Otto-von-Guericke-Universität Magdeburg

FAKULTÄT FÜR
INFORMATIK

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Bachelorarbeit

Untersuchung der Nutzung mehrdimensionaler Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme

Verfasser:

Sara Kunze

15. Dezember 2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich,
Dipl.-Inform. Martin Kuhlemann

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

**Korregierte Version 2011,
Korrekturen auf den Seiten 68-70**

Kunze, Sara:
Untersuchung der Nutzung mehrdimensionaler Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2010.

Danksagung

Ich möchte mich bei Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und Dipl.-Inform. Martin Kuhlemann für die Betreuung meiner Arbeit bedanken. Zahlreiche Kommentare und kritische Fragen ihrerseits haben entscheidend zur Verbesserung der Arbeit beigetragen. Zudem möchte ich meiner Familie danken, die mir erst ein Studium ermöglicht und mich immer unterstützt. Besonderer Dank gilt meiner Mutter Kornelia Kunze für die Hinweise zu Rechtschreibung und Grammatik. Mein Dank gilt auch der Arbeitsgruppe von Prof. Dr. rer. nat. Jörg Kaiser, die mir für die Arbeit freundlicherweise einen Roboter als Testsystem zu Verfügung gestellt hat. Schließlich möchte ich mich bei allen, auch den nicht namentlich benannten, für ihre Hilfe, Motivation und Unterstützung herzlich bedanken.

Inhaltsverzeichnis

Inhaltsverzeichnis iii

Abbildungsverzeichnis v

Tabellenverzeichnis vii

Verzeichnis der Abkürzungen ix

1 Einleitung 1

1.1 Motivation 1

1.2 Zielsetzung 2

1.3 Gliederung 2

2 Grundlagen 3

2.1 Eingebettete Systeme 3

2.1.1 Begriffserklärung 4

2.1.2 Anwendungsgebiete 6

2.1.3 Anforderungen 8

2.2 Datenmanagement in eingebetteten Systemen 10

2.2.1 Begriffserklärung 11

2.2.2 Daten in eingebetteten Systemen 12

2.2.3 Datenspeicherung in eingebetteten Systemen 14

2.2.4 Datenbankenmanagementsysteme in eingebetteten Systemen . . . 16

3 Nutzung mehrdimensionaler Daten 23

3.1 Mehrdimensionale Daten 23

3.2 Mehrdimensionale Daten in klassischen Rechnersystemen 24

3.2.1 Geodatenbanksysteme 24

3.2.2 Multimedia-Datenbanken 25
3.3 Anwendungen in eingebetteten Systemen 25
3.3.1 Intelligente, eingebettete Systeme 26
3.3.2 Automobilbereich 30
3.4 Anforderungsanalyse 34

4 Mehrdimensionale Indexstrukturen 37
4.1 Übersicht und Einteilung 37
4.2 Anfragearten 38
4.3 Betrachtung verschiedener mehrdimensionaler Indexstrukturen 41
4.3.1 Der R-Baum 41
4.3.2 Der LSD-Baum 42
4.3.3 Das Grid-File 44
4.3.4 Das VA-File 45
4.4 Auswahl 46

5 Implementierung 49
5.1 Erweiterung von RobbyDBMS 49
5.1.1 Feature-orientierte Programmierung von RobbyDBMS 49
5.1.2 Integration der neuen Features in RobbyDBMS 50
5.2 Allgemeine Festlegungen 52
5.3 Implementierung mehrdimensionaler Anfragen ohne Index 54
5.4 Implementierung des VA-Files 55
5.5 Implementierung des LSD-Baum 60

6 Analyse 65
6.1 Durchführung und Grenzen der Analyse 65
6.2 Ergebnisse 67
6.2.1 Speicherplatzbedarf 67
6.2.2 Ausführungszeit 70
6.3 Diskussion 76

7 Zusammenfassung und Ausblick 77

Literaturverzeichnis 79

Abbildungsverzeichnis

2.1 Struktur eingebetteter Systeme mit möglichen Wirkungsketten nach [Sch05b] . . . 4
2.2 Allgemeiner Aufbau eines Mikrocontrollers nach [SZ06] . . . 5
2.3 Einteilung der Datenmanagementsysteme nach dem Funktionsumfang [RLAS07] . . . 12
2.4 Übersicht von Speichertechnologien [SZ06] . . . 15
2.5 Aufbau von FlashDB [NK07] . . . 17
2.6 Symbole eines Feature-Diagramms [CE99] . . . 18
2.7 Feature-Diagramm COMET DBMS [Lie08] . . . 19
2.8 Feature-Diagramm FAME-DBMS [Lie08] . . . 20
2.9 Feature-Diagramm RobbyDBMS ohne mehrdimensionale Indexstrukturen und Anfragen [Lie08] . . . 21
2.10 Schichtenmodell von RobbyDBMS [Lie08] . . . 22

3.1 Ablauf einer Mustererkennung am Beispiel von Bilddaten nach [Nau05] . 26
3.2 Beispiel eines Aufbaus eines Sensornetzwerks für Anwendungen im betreuten Wohnen: Intelligente Umgebung mit Lokalisierungsfunktion [Röh10] 28
3.3 Einteilung der eingebetteten Systeme im Auto in Subsysteme mit Beispielen nach [SZ06] . . . 31
3.4 Zündwinkelkennfeld einer Motorsteuerung [DG98] . . . 32
3.5 Tabellarisches Ablageschema eines zweidimensionalen Kennfeldes [SZ06] . 32
3.6 Innere Struktur von Tupel . . . 34

4.1 Graphische Darstellung der Bereichssuche im zweidimensionalen Fall . . . 39
4.2 Graphische Darstellung der Nächsten-Nachbarsuche im zweidimensionalen Fall mit der euklidischen Distanz [Sch05a] . . . 40
4.3 R-Baum mit Punktdaten im zweidimensionalen Raum [Sch05a] . . . 41

4.4 Graphische Darstellung eines MBRs im zweidimensionalen Raum mit dem beschreibenden Punktepaar (s,t) [Sch05a] 42
4.5 Mögliche Datenraumaufteilung und zugehöriger LSD-Baum [HSW89] . . 43
4.6 Aufbau eines Grid-Files [SHS05] 45
4.7 Datenraumaufteilung und Approximation im VA-File für einen zweidimensionalen Beispieldatenraum [WB97] 46

5.1 Feature-Diagramm von RobbyDBMS mit mehrdimensionalen Anfragen und unterstützenden Indexstrukturen (grau hinterlegt) 51
5.2 Aufbau eines Datensatzes 53
5.3 Pseudocode für mehrdimensionale Exact-Match-Suche ohne Indexunterstützung 55
5.4 Variante mit Speicherung im EEPROM: Speicherung der Listen 56
5.5 Obere und untere Grenze der Distanz nach [WB97] 58
5.6 Pseudocode für SSA nach [WB97] 59
5.7 Pseudocode für NOA nach [WB97] 60
5.8 Pseudocode für mehrdimensionale Exact-Match-Suche mit VA-File 61
5.9 Pseudocode für die Exact-Match-Suche mit Unterstützung durch den LSD-Baum nach [HSW89] 62
5.10 Pseudocode für die Nächste-Nachbarsuche mit Unterstützung durch den LSD-Baum nach [Hen94] 63
5.11 Pseudocode für die Bereichssuche mit Unterstützung durch den LSD-Baum 64

6.1 Analyse der maximalen Anzahl von Datensätzen für $D = 2$, $D = 3$ und $D = 4$ 68
6.2 Ergebnisse der Analyse der Exact-Match-Suche 71
6.3 Ergebnisse der Analyse der Bereichssuche 73
6.4 Ergebnisse der Analyse der Nächsten-Nachbarsuche mit SSA für VA-Files 74
6.5 Vergleich der Anfragezeiten für NOA und SSA für zweidimensionale Daten 75

Tabellenverzeichnis

2.1 Basisdatentypen mit Schlüsselwort, Wertebereich und Größe in C [Wie07] 13
2.2 Zuordnung Beispieldaten zu Speichertechnologie [NTN+02] 16
5.1 Mögliche Werte für b und ihre Eignung 57
6.1 Übersicht über den Programmspeicherbedarf der Indexstrukturen nach Anfrage-Features (in Bytes) 69
6.2 Intervalle (min-max) des Speicherbedarfs einzelner Anfragen im Bezug zum verwendeten Index (in Bytes) 70

Verzeichnis der Abkürzungen

ABS Antiblockiersystem

ACC daptive Cruis Control

API Application Programming Interface

DBMS Datenbankmanagementsystem

EBV elektronische Bremskraftverteilung

ESP elektronische Stabilitätsprogramm

MBR Minimum Bounding Rectangle

NOA Near-Optimal Algorithm

SSA Simple-Search Algorithm

TID Tupel Identifier

Kapitel 1

Einleitung

1.1 Motivation

Das Datenaufkommen in eingebetteten Systemen steigt allgemein an. Zum Beispiel bei eingebetteten Systemen im Auto ist eine jährliche Steigerung um 7 bis 10% festzustellen[NTN+02]. Dieser Anstieg ist bei gleichbleibend geringen Ressourcen eine Herausforderung an die Datenhaltung in tief eingebetteten Systemen [RLAS07]. In klassischen Rechnersystemen werden große Datenmengen häufig in Datenbanksystemen verwaltet. Datenbanksysteme bestehen aus einer Datenbank, welche die Daten speichert, und einem Datenbankmanagementsystem (DBMS) zum Zugriff und zur Verwaltung der Daten [SSH08]. Datenbankmanagementsysteme für klassische Rechnersysteme besitzen in der Regel eine Vielzahl an Funktionen, die für den Einsatz auf eingebetteten Systemen nicht notwendig beziehungsweise nicht immer notwendig sind [RLAS07]. Zudem sind diese DBMS nicht auf die Heterogenität der Systeme und die Ressourcenbeschränkungen zugeschnitten [RLAS07]. Dies führte zur Entwicklung spezieller DBMS-Lösungen für den Einsatz auf eingebetteten Systemen, die oftmals für konkrete Anwendungen entwickelt wurden [RLAS07]. Dabei werden in diesen speziellen Datenbankmanagementlösungen in der Regel nicht alle Funktionen klassischer DBMS, wie Anfragesprachen und Mehrbenutzerbetrieb, implementiert [RLAS07].

Indexstrukturen unterstützen den Zugriff auf die Daten [SHS05] und werden häufig auch in Datenbankmanagementsystemen für eingebettete und tief eingebettete Systeme verwendet [NTNH03, RSS+08, NK07, Lie08]. Die Indexstrukturen lassen sich in eindimensionale und mehrdimensionale Indexstrukturen untergliedern. Eindimensionale Indexstrukturen unterstützen dabei den Zugriff auf die Daten über genau ein Attribut oder den Tupel Identifier (TID) [SHS05]. Realisierungen solcher Indexstrukturen sind in verschiedenen DBMS-Lösungen bereits vertreten [NTNH03, RSS+08, NK07, Lie08]. Mehrdimensionale Indexstrukturen unterstützen den Zugriff auf die Daten über mehrere Attribute und ermöglichen spezielle mehrdimensionale Anfragen wie die Nächste-Nachbarsuche [SHS05]. Diese Indexstrukturen werden in klassischen Rechnersystemen zum Beispiel in Multimedia- und in Geodatenbanken eingesetzt [Pag96, Sch05a, SHS05] und sind für den Einsatz in tief eingebetteten Systemen noch nicht untersucht wurden. Daraus ergibt sich die Frage, ob mehrdimensionale Indexstrukturen auch in einem DBMS für tief eingebettete Systeme einen Nutzen haben. Dazu gehört es, zu klären, ob in tief eingebetteten Systemen entsprechende mehrdimensionale Daten verwalten werden

müssen und ob eine mehrdimensionale Indexstruktur eine Verbesserung der Anfragezeiten im Vergleich zum sequentiellen Durchsuchen aller Datensätze bewirkt. Dabei ist zu beachten, dass die Anzahl an Datensätzen im Vergleich zu klassischen Einsatzgebieten mehrdimensionaler Indexstrukturen eher als gering einzustufen ist [SHS05]. Zudem ist zu klären, ob die Speicherkosten in den ressourcenbeschränkten Systemen die möglicherweise zu erzielenden Verbesserungen rechtfertigen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, herauszufinden in wieweit mehrdimensionale Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme sinnvoll sind. Dabei muss untersucht werden, ob mehrdimensionale Daten in tief eingebetteten Systemen so genutzt werden, dass eine Speicherung in einer Datenbank möglich ist, und wie dieser Daten strukturiert sind. Danach gilt es herauszufinden, welche mehrdimensionalen Indexstrukturen für die Implementierung in einem tief eingebetteten System geeignet sind. Anhand der exemplarisch implementierten Indexstrukturen ist eine Analyse erforderlich, welche die Kosten und den Nutzen der Indexstrukturen vergleicht. Diese Analyse muss besonders für Datenbanken mit wenigen Datensätzen erfolgen, da tief eingebettete Systeme typischerweise starken Ressourcenbeschränkungen unterliegen und somit im Vergleich zu klassischen Rechnersystemen nur wenig Datensätze speichern können. Schlussendlich werden Aussagen zur Verwendung der implementierten Indexstrukturen und zur Verwendung mehrdimensionaler Indexstrukturen in tief eingebetteten Systemen mit einer geringen Anzahl an Datensätzen im Vergleich zu klassischen Rechnersystemen getroffen.

1.3 Gliederung

Im Kapitel 2 wird in die Thematik der eingebetteten Systeme und der Datenhaltung in eingebetteten Systemen eingeführt. Das darauf folgende Kapitel 3 beschäftigt sich mit der Nutzung mehrdimensionaler Daten. Dazu wird auf spezielle Datenbanksysteme für mehrdimensionale Daten in klassischen Rechnersystemen eingegangen. Auf Grundlage dieser Betrachtungen werden dann Anwendungen eingebetteter Systeme, die mit mehrdimensionalen Daten arbeiten, näher erläutert und die Möglichkeit des Einsatzes eines Datenbanksystems mit einer mehrdimensionalen Indexstruktur zur Verwaltung der vorhandenen Daten betrachtet. Schließlich wird in diesem Kapitel eine Anforderungsanalyse durchgeführt, welche die Anforderungen an die Indexstruktur aufzeigt. Diese Anforderungen werden im Kapitel 4 dazu genutzt zwei mehrdimensionale Indexstrukturen zur Implementierung und Analyse auszuwählen. Die Auswahl erfolgt auf Grundlage einer vorhergehenden Vorstellung mehrerer mehrdimensionaler Indexstrukturen. Die Implementierung der ausgewählten Indexstrukturen und der mehrdimensionalen Anfragen ohne Indexunterstützung wird im Kapitel 5 erklärt. Das Kapitel 6 beschreibt die Analyse der implementierten Indexstrukturen bezüglich des Speicherbedarfs und der Ausführungszeiten für die Anfragen. Im Kapitel 7 folgt eine Zusammenfassung dieser Arbeit und ein Ausblick auf weitere mögliche Arbeiten.

Kapitel 2

Grundlagen

Dieses Kapitel macht mit den Grundlagen, die zum Verständnis dieser Arbeit notwendig sind, vertraut. Dazu ist es erforderlich, zuerst eine kurze Einführung in die Thematik der eingebetteten Systeme zu geben. Der Abschnitt 2.1 erläutert eingebettete Systeme im Allgemeinen. Dazu gehören die Definition von wichtigen Begriffen und ein Überblick über den Aufbau eingebetteter Systeme. Des Weiteren werden Anwendungsgebiete und Beispielsysteme sowie Anforderungen betrachtet.

Indexstrukturen gehören zum Bereich des Datenmanagements [SHS05]. Aufgrund dessen folgt im Abschnitt 2.2 ein Einblick in die Thematik des Datenmanagements in eingebetteten Systemen. Dort werden die grundlegenden Begriffe erklärt, anfallende Daten beschrieben und eine Übersicht über Speichertechnologien und ihre Auswirkungen auf das Datenmanagement gegeben. Zum Schluss werden einige Datenbankmanagementsysteme für eingebettete Systeme vorgestellt. Dies dient dazu, einen kurzen Überblick über die Datenbankmanagementsysteme selbst und die implementierten Indexstrukturen zu bekommen.

2.1 Eingebettete Systeme

Eingebettete Systeme finden in vielen sehr unterschiedlichen Bereichen Anwendung [Mar07]. Damit einhergehen teilweise stark unterschiedliche Anforderungen und damit unterschiedliche Ausprägungen solcher Systeme [Mar07]. Aufgrund dieser Vielfalt ist es notwendig, im Kontext dieser Arbeit mit einer einheitlichen Definition des Begriffes *„eingebettetes System“* zu arbeiten.

Abschnitt 2.1.1 definiert die Begriffe „eingebettetes System“ und „tief eingebettetes System“ und erläutert den allgemeinen Aufbau eingebetteter Systeme. Des Weiteren werden Unterschiede zu klassischen Rechnersystemen herausgearbeitet, die für den Einsatz mehrdimensionaler Daten von Bedeutung sind. Auf die unterschiedlichen Anwendungsbereiche wird im darauf folgenden Abschnitt 2.1.2 kurz eingegangen. Im Abschnitt 2.1.3 werden danach allgemeine Anforderungen an eingebettete Systeme thematisiert. Der Bezug Anforderungen und Anwendung wird dort gleichfalls thematisiert. Im Anschluss an diesen Abschnitt folgt ein Überblick über das Datenmanagement in eingebetteten Systemen.

2.1.1 Begriffserklärung

Marwedel definiert den Begriff *„eingebettetes System“* wie folgt:

> *„Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.“ [Mar07]*

Der Begriff *„tief eingebettetes System“* beschreibt eine Gruppe eingebetteter Systeme, die sich durch *„extreme Ressourcenbeschränkungen hinsichtlich Speicher, CPU und Energieverbrauch“ [SSPSS98]* auszeichnet. Beispiele für solche tief eingebetteten Systeme sind Herzschrittmacher, Waschmaschinen, Subsysteme von Autos und Flugzeugen [SSPSS98, BIT10]. Die Beschränkungen beruhen zumeist auf Anforderungen bezüglich des Preises wie auch des Gewichts des Endproduktes [Mar07]. Näheres zu den Anforderungen an eingebettete Systeme wird im Abschnitt 2.1.3 erklärt.

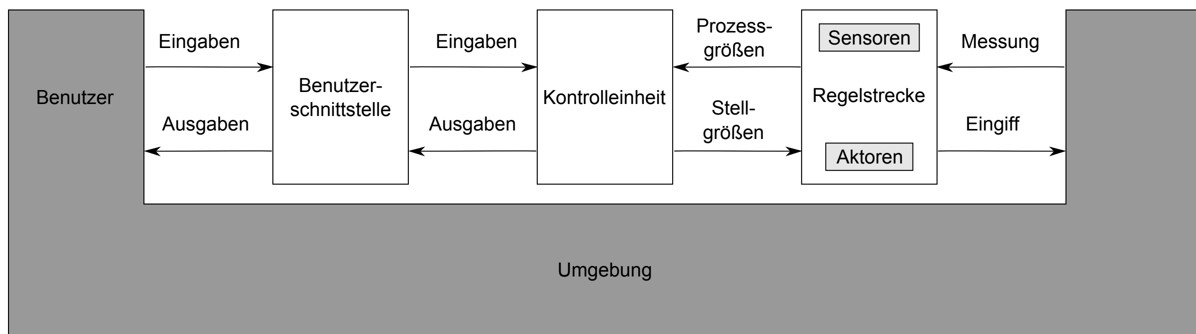


Abbildung 2.1: Struktur eingebetteter Systeme mit möglichen Wirkungsketten nach [Sch05b]

Eingebettete Systeme bestehen aus Hard- und Softwarekomponenten, die gemeinsam die Funktion erfüllen [Sch05b]. Der grundlegende Aufbau ist bei den meisten eingebetteten Systemen gleich und wird in drei Komponenten gegliedert [Sch05b]. Abbildung 2.1 veranschaulicht diese Struktur mit ihren möglichen Wirkungsketten. Diese Komponenten sind die Kontrolleinheit, die Regelstrecke und die Benutzerschnittstelle. Das eingebettete System ist in die Systemumgebung, integriert [Sch05b].

Der Benutzer als Teil der Systemumgebung nimmt in vielen Systemen eine Sonderrolle ein und ist daher extra aufgeführt [Sch05b]. Diese Sonderrolle wird durch die Benutzerschnittstelle unterstützt. Diese dient der Kommunikation zwischen Benutzer und Kontrolleinheit [Sch05b]. Die Kontrolleinheit erhält so Benutzereingaben, um diese entsprechend zu verarbeiten, und liefert Ausgaben. Dadurch erhält der Benutzer Informationen zum Systemzustand [Sch05b]. Die Benutzerschnittstelle wird durch Sonderformen von Sensoren und Aktoren realisiert. Dies sind beispielsweise Taster, Drehregler, Leuchtdioden (LEDs) oder Anzeigen [Sch05b].

Die Kontrolleinheit bildet das Zentrum des Systems und ist das eigentliche Rechnersystem, das über die Regelstrecke und die Benutzerschnittstelle mit der Umgebung interagiert [Sch05b]. Die Kontrolleinheit kann in Form eines Mikrocontrollers realisiert sein [SZ06]. Die Hauptoperationen von Mikrocontrollern werden zu Datenverarbeitung, Datenspeicherung und Datenaustausch zusammengefasst [SZ06]. Der allgemeine Aufbau eines Mikrocontrollers ist in Abbildung 2.2 dargestellt.

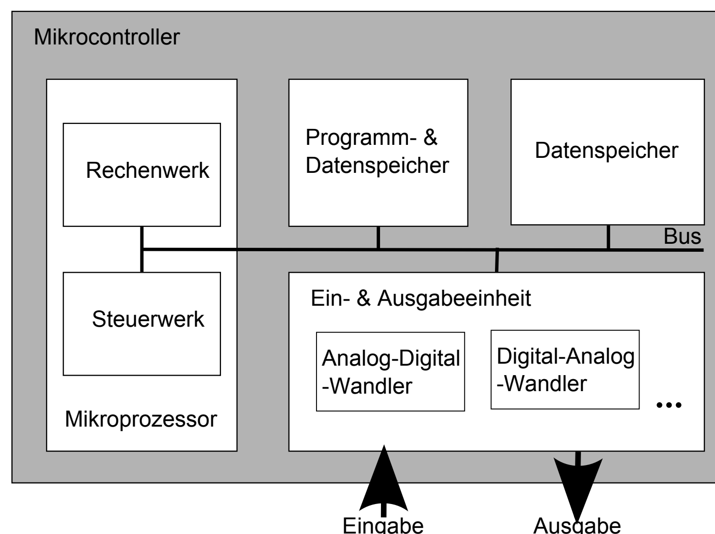


Abbildung 2.2: Allgemeiner Aufbau eines Mikrocontrollers nach [SZ06]

Mikroprozessor Der Mikroprozessor ist die zentrale Recheneinheit des Controllers und enthält als Hauptkomponenten das Rechenwerk und das Steuerwerk [SZ06]. Die Ausführung der arithmetischen und logischen Operationen erfolgt im Rechenwerk [SZ06]. Das Steuerwerk steuert die Ausführung der Operationen, die in Form von Befehlen im Programmspeicher stehen [SZ06].

Ein- und Ausgabeeinheit Die Ein- und Ausgabeeinheit dient der Kommunikation mit den angeschlossenen Geräten [SZ06]. Alle Ein- und Ausgaben des Mikrocontrollers werden über diese Funktionseinheit abgewickelt [SZ06]. Ein- und Ausgaben sind analoge oder digitale Werte [SZ06]. Analoge Werte müssen in der Einheit umgewandelt werden. Für analoge Eingaben wird der Analog-Digital-Wandler genutzt, für Ausgaben der Digital-Analog-Wandler oder Pulsweitenmodulation [SZ06]. Des Weiteren werden in der Ein- und Ausgabeeinheit externe Programmunterbrechungen, serielle Schnittstellen, die Bussteuerung für externe Bausteine und digitale Ein- und Ausgänge realisiert [SZ06].

Programm- und Datenspeicher Im Programm- und Datenspeicher werden der Programmcode sowie bestimmte konstante Parameter gespeichert [SZ06]. Meist erfolgt eine Trennung von Programm und Daten [SZ06].

Datenspeicher Der Datenspeicher dient zur Speicherung von allen veränderlichen Daten [SZ06]. Die Daten können sich dabei während der Laufzeit ändern [SZ06]. Der Speicher muss daher Schreiboperationen zur Programmlaufzeit ermöglichen [SZ06].

Bussystem Das Bussystem ermöglicht die interne Kommunikation der Elemente untereinander [SZ06].

Die über die Ein- und Ausgabeeinheit angeschlossene Regelstrecke besteht aus Sensoren und Aktoren, welche die Umgebung direkt beobachten und steuern können [Sch05b]. Sie stellt somit die Schnittstelle des eingebetteten Systems zur Umgebung dar.

> *Ein Sensor wird definiert, als „eine Einrichtung zum Feststellen von physikalischen oder chemischen Eingangsgrößen, die optimal eine Messwertzuordnung der Größe treffen kann, sowie gegebenenfalls ein digitales beziehungsweise digitalisierbares Ausgangssignal liefern kann.“ [Sch05b]*

Ein Aktor ist eine technische Einheit, die elektrische Energie in mechanische Arbeit, thermische Energie oder Strahlung umwandelt [Sch05b]. Die elektrische Energie wird von der Kontrolleinheit zum Beispiel über Pulsweitenmodulation oder über Digital-Analog-Wandlung an die Regelstrecke geliefert. Der Aktor wirkt so auf die Umgebung ein und verändert diese. Sensoren und Aktoren sind selbst technische Systeme [SZ06]. Erfolgt in diesen Systemen eine Vor- oder Nachbearbeitung der Daten, so werden diese *intelligente Sensoren* beziehungsweise *Aktoren* genannt [SZ06]. Das eingebettete System übernimmt mit Hilfe dieser Struktur komplexe Steuerungs-, Regelungs-, Überwachungs- und Datenverarbeitungsaufgaben [Sch05b].

Der Aufbau ist bereits ein großer Unterschied zu klassischen Rechnersystemen wie Personal Computern. Einer der weiteren Hauptunterschiede ist die häufige Optimierung eingebetteter Systeme auf eine vordefinierte Anwendung [Mar07]. Das heißt, das System ist hard- und softwareseitig auf diese Anwendung ausgerichtet. Dies dient der Verbesserung der Verlässlichkeit und der Effizienz [Mar07]. Klassische Rechnersysteme werden typischerweise für viele verschiedene, nicht immer vordefinierte Anwendungen entworfen [Mar07].

Ein weiterer großer Unterschied zu klassischen Rechnersystemen ist die Wirkung des Systems auf die Umgebung. Eingebettete Systeme wirken in der Regel auf die physikalische Umgebung ein, währenddessen klassische Rechnersysteme nicht darauf ausgelegt sind [Hor05]. Das heißt, eingebettete Systeme sollen auf die Umgebung einwirken und diese so verändern [Hor05].

Die Erzeugung des Programmcodes unterscheidet beide Systeme ebenfalls stark. Programme für klassische Rechnersysteme können in der Regel direkt auf dem Zielsystem programmiert und getestet werden [PR05]. Der Compiler kann auf dem System selbst ausgeführt werden [PR05]. Bei eingebetteten Systemen ist dies meist nicht möglich [PR05]. Daher werden dort Crosscompiler angewendet [PR05]. Mit diesen ist es möglich, den Programmcode auf einem anderen System zu erstellen und dann auf das Zielsystem zu übertragen. Das Debuggen des erstellten Codes ist ebenfalls auf einem eingebetteten System nicht immer möglich [PR05]. Um diesen Programmierschritt zu erleichtern, werden häufig Simulatoren eingesetzt [PR05].

Im folgenden Abschnitt 2.1.2 werden die Anwendungsgebiete eingebetteter Systeme und einige konkrete Beispiele der Anwendung eingebetteter Systeme genannt. Das Kapitel 3 beschäftigt sich dann weiterführend mit dieser Thematik.

2.1.2 Anwendungsgebiete

Aktuell ist ein Trend, der *„verschwindende Computer“* genannt wird, festzustellen [Mar07]. Der Name bezieht sich dabei auf das scheinbare Verschwinden der Computer [Mar07]. Zu Bemerken ist dabei, dass die Rechnersysteme nicht im wörtlichen Sinne verschwinden, sondern unsichtbar für die Benutzer werden und doch gleichzeitig überall vorhanden sind [Mar07]. Damit ist gemeint, dass die den Benutzer umgebenden Produk-

te eingebettete Systeme in der Form beinhalten, dass sie nach außen hin nicht sichtbar werden [Mar07].

Einige Anwendungsgebiete eingebetteter Systeme mit Beispielsystemen sind [Mar07, Sch05b, BIT10]:

- *Automobilbereich*: Airbag, Motorsteuerung, Fahrassistenzsysteme
- *Bereich des öffentlichen Verkehrs*: Fahrkartenautomaten, Zugleitsysteme, Zugbeeinflussungssysteme, Ampelanlagen
- *Luft- und Raumfahrtsektor*: Bordelektronik von Flugzeugen, Raketensteuerungen, Satelliten
- *Telekommunikationsbereich*: Mobiltelefone, Sende- und Verteilungsanlagen, Router
- *Medizintechnik*: Herzschrittmacher, mobile Diagnostiksysteme, Röntgenanlagen, Computertomographen
- *militärische Anwendungen*: Systemen in Panzer, Flugzeugen, Schiffen und U-Booten, Aufklärungsdrohnen, mobile Radargeräte
- *Sicherheitstechnik*: Authentifizierungssysteme, Überwachungsanlagen
- *Unterhaltungselektronik*: Fotoapparate, Fernseher, MP3-Player, Spielkonsolen
- *industrielle Anwendungen*: Fabriksteuerungen, Förderanlagen, automatisierte Fertigungsanlagen wie CNC-Fräsen
- *intelligente Gebäudetechnik*: Klimaanlagen, Lichtsteuerung, Zugangskontrollen, Heizsysteme
- *Robotik*: Steuerung der Mechanik, der Mensch-Maschine Interaktion
- *Haushaltsgeräte*: Waschmaschinen, Mikrowellen, Kühlschränke, Staubsauger, Rasenmäher
- *Versorgungstechnik*: Energieversorgung, Gasversorgung, Wasserversorgung

Diese Auflistung stellt einen groben Überblick über die Anwendungsgebiete eingebetteter Systeme im täglichen Leben dar. Die Auflistung ist aufgrund der Vielfalt nicht vollständig. Eine Beschränkung der Ressourcen ist in fast allen Beispielsystemen festzustellen. Daher gelten diese Systeme auch als Beispiele für tief eingebettete Systeme. Zu diesen Systemen zählen zum Beispiel Fahrassistenzsysteme, Satelliten, Herzschrittmacher und Rasenmäher. Nicht in die Gruppe gehören in der Regel Computertomographen und Spielkonsolen. In Kapitel 3 wird der Automobilbereich und seine konkreten Anwendungen bezüglich der Nutzung mehrdimensionaler Daten vertiefend betrachtet. Hinzu kommen Beispielanwendungen aus verschieden Bereichen, die mit Mustererkennung arbeiten.

Einige der oben genannten Anwendungen, wie der Automobilbereich, stellen besondere Anforderungen an das System, insbesondere bezüglich der Verlässlichkeit, der Effizienz, des Gewichtes und des Preises [Mar07]. Diese speziellen Anforderungen erwachsen

aus den Rahmenbedingungen des Systems. Beispiele für solche Rahmenbedingungen sind die Art der Produktion, die Marktsituation, die Umwelt- und Umgebungsbedingungen, gesetzliche Regelungen und Sicherheitsvorschriften [Gri05]. Der folgende Abschnitt 2.1.3 beschäftigt sich mit den allgemeinen Anforderungen und einigen Beispielen zu oben genannten Rahmenbedingungen.

2.1.3 Anforderungen

Die Einbettung in das jeweilige Produkt führt zu speziellen Anforderungen an die Hard- und Software eingebetteter Systeme [Mar07]. Einige Anforderungen beschränken sich auf das jeweilige Anwendungsgebiet, andere können allgemein für eingebettete Systeme ausgemacht werden. Diese allgemeinen Anforderungen treten dann in den jeweiligen Anwendungen mit unterschiedlicher Relevanz auf [Mar07].

In eingebetteten Systemen ist die Verlässlichkeit wichtig [Mar07]. Dies ist unter anderem darin begründet, dass viele eingebettete Systeme sicherheitskritisch sind oder die Nutzer eine hohe Verlässlichkeit erwarten [Mar07]. Das bedeutet, dass nicht nur die algorithmische Korrektheit der Programmierung sichergestellt sein muss, sondern das Gesamtsystem muss verlässlich arbeiten [Mar07]. Die Verlässlichkeit lässt sich dabei in die Unterpunkte Zuverlässigkeit, Wartbarkeit, Verfügbarkeit, Sicherheit und Integrität gliedern [Mar07].

Zulässigkeit Die Zuverlässigkeit ist eine statistische Größe, welche die Wahrscheinlichkeit des Nichtausfalls eines Systems beschreibt [Mar07]. Jeder Systemausfall wirkt sich negativ auf die Zuverlässigkeit aus [Mar07]. Die Zuverlässigkeit ist in sicherheitskritischen Systemen besonders wichtig [BIT10]. Würde das Bremsassistenzsystem eines Autos beispielsweise nur mit einer Wahrscheinlichkeit mit 50% funktionieren, wenn er gebraucht wird, so würden Unfälle nicht immer verhindert oder abgemildert werden können. Der Hersteller hätte im Vergleich mit anderen Anbietern, deren Systeme besser funktionieren, sicher einen Marktnachteil. Ein weiteres Beispiel für ein eingebettetes System, das hoch zuverlässig sein muss, ist im Bereich der Luft- und Raumfahrt zu finden. Beispielsweise Satelliten müssen ausfallfrei arbeiten. Ein vollständiger Systemausfall wäre mit hohen Kosten sowie starken Zeitverzögerungen verbunden.

Wartbarkeit Die Wartbarkeit gibt an, wie wahrscheinlich es ist, dass ein System nach einem Ausfall innerhalb einer bestimmten Zeit repariert werden kann [Mar07]. Diese Eigenschaft ist besonders wichtig bei Systemen, die schnell wieder verfügbar sein müssen. Beispiele dafür sind der Telekommunikationsbereich und der industrielle Anwendungsbereich. Bei diesen Bereichen wirken sich lange Ausfallzeiten direkt auf den Umsatz des Unternehmens aus, welches das System nutzt [BIT10]. Aber auch Bereiche wie der automobile Anwendungsbereich und die Unterhaltungselektronik setzen hohe Anforderungen an die Wartbarkeit, da hohe Ausfallzeiten schlecht für den Ruf des Herstellerunternehmens sind.

Verfügbarkeit In die Verfügbarkeit fließen sowohl die Zuverlässigkeit als auch die Wartbarkeit ein [Mar07]. Somit wird mit der Verfügbarkeit die Wahrscheinlichkeit angegeben, dass ein System genutzt werden kann und seine Aufgaben korrekt erfüllt

[Mar07]. Diese Eigenschaft ist in der Regel immer wichtig für ein eingebettetes System, da geringe Verfügbarkeiten den Nutzer und damit in der Regel auch den Käufer verstimmen würden.

Sicherheit Die Sicherheit eines Systems wird in zwei Bereiche geteilt [BIT10]. Das ist zum einen die Sicherheit von Schäden, die durch das System entstehen und zum anderen die Sicherheit des Systems vor Angreifern [BIT10].

Die Sicherheit vor Schäden ist eine weitverbreitete Anforderung an eingebettete Systeme, da diese Systeme in Wechselwirkung mit der Umgebung stehen und diese beeinflussen [Mar07, BIT10]. Ein eingebettetes System kann auf zweierlei Arten unbeabsichtigte Schäden an der Umgebung anrichten, zum einen durch den Ausfall von Komponenten und zum anderen durch einen Fehler in der Software [Sch05b]. Ein Fehler in der Software wird häufig durch fehlerhafte Eingabedaten oder eine fehlerhafte Programmierung verursacht. Tritt ein Fehler auf, so muss sich das System selbst in einen sicheren Zustand bringen.

Die Sicherheit eines eingebetteten Systems vor Angreifern geht stark einher mit der Integrität der Daten [BIT10]. Werden die Software oder die Daten durch einen Angriff verändert, so ist das Systemverhalten nicht mehr vorhersagbar. Somit ist die Funktion des Systems nicht sicher gestellt [BIT10].

Eine weitere wichtige Anforderung an viele eingebettete Systeme betrifft die Effizienz [Mar07]. Besonders tief eingebettete Systeme unterliegen hier starken Restriktionen. Die Effizienz betrifft dabei die Bereiche Energie, Speicher, Zeit, Gewicht und Kosten [Mar07].

Energie Viele eingebettete Systeme werden mobil betrieben [Mar07]. Das heißt, sie arbeiten mit Akkumulatoren oder Batterien. Daher muss der Strom optimal genutzt werden, um möglichst lange die geforderten Funktionen ausführen zu können [Mar07].

Speicher Speicher ist aus Gründen des Gewichts und der Kosten in der Regel knapp [Mar07]. Dies gilt sowohl für den Programm- als auch den Datenspeicher [Mar07]. In Anbetracht dessen, muss der Programmcode möglichst gering sein [Mar07]. Das bedeutet, dass nicht mehr Code als zur Erfüllung der Funktion notwendig auf dem System gespeichert werden sollte [Mar07]. Das Gleiche gilt für Daten, die zur Übersetzungszeit oder zur Laufzeit erzeugt werden [Mar07].

Zeit Die Zeit ist generell ein wichtiger Faktor in Rechnersystemen. Hinzu kommt, dass viele eingebettete Systeme zeitkritisch sind [Sch05b]. Das bedeutet, dass das System innerhalb einer bestimmten Zeit auf seine Umgebung reagieren muss [Sch05b]. Dies ist unter anderem darin begründet, dass Sensordaten aus einem kontinuierlichen Signalfluss nur eine gewisse Zeit gültig sind. Eine stark verzögerte Ausführung führt zu einer verzögerten Beeinflussung der Umgebung. In der Zwischenzeit kann die benötigte Beeinflussung bereits stark von der eingeleiteten Beeinflussung abweichen. Ein Beispiel für ein stark zeitkritisches System ist ein Bremsassistenzsystem von Autos. Das Auto muss in kürzester Zeit richtig auf die Situation reagieren und dementsprechend handeln. Lange Ausführungszeiten hätten hier möglicherweise

katastrophale Folgen. Ein solches eingebettetes System wird Echtzeitsystem genannt [Mar07]. Das heißt, dieses System muss weiche oder harte Echtzeitbedingen einhalten [Mar07].

Zeit kann oft durch Speicherung von Zwischenwerten eingespart werden, da eine erneute Berechnung nicht notwendig wird. Der dadurch benötigte Speicher wirkt sich jedoch auf den Speicherbedarf aus. Dadurch lässt sich die Aussage treffen, dass die Forderung nach möglichst schneller Ausführung in Konkurrenz zur Forderung nach möglichst wenig Speicherbedarf steht.

Gewicht Mobile eingebettete Systeme haben oft starke Einschränkungen bezüglich des Gewichtes [Mar07]. Diese Systeme sollen möglichst leicht und ressourcenschonend transportiert werden. Ein geringes Gewicht bietet so Wettbewerbsvorteile [Mar07]. Bei Systemen in Autos, Flugzeugen, Zügen und anderen Transportmitteln wird mit weniger Gewicht auch weniger Kraftstoff verbraucht [Gri05]. Dies wirkt sich auf die Umwelt sowie auf die Betriebskosten aus. Zudem sind gesetzliche Restriktionen bezüglich des Kraftstoffverbrauchs und der CO2-Emission beim Auto einzuhalten. Gerade diese beiden Faktoren spielen heute eine wichtige Rolle beim Kauf solcher Produkte. Bei kleineren mobilen Geräten besonders im Bereich der Unterhaltungselektronik und der Medizintechnik trägt das Gewicht des Gesamtproduktes maßgeblich zum Nutzen bei. Um das Gewicht zu sparen, müssen die notwendigen Ressourcen, wie zum Beispiel der Speicher, minimiert werden [Mar07]. Dies setzt eine optimale Nutzung dieser Ressourcen voraus [Mar07].

Kosten Ein weiterer Grund für die Minimierung der verwendeten Ressourcen ist der Preis [Mar07]. Viele eingebettete Systeme werden in Massenproduktion hergestellt [Mar07]. Diese Produkte unterliegen oft stark dem Wettbewerb [Mar07]. Die Kosten sind daher ein wichtiger Faktor für eingebettete Systeme. Beispiele dafür sind in fast allen Bereichen zu finden.

Durch diese Beschränkung der Ressourcen müssen die anfallenden Daten optimal verwaltet werden. Dies gilt insbesondere für die stark ressourcenbeschränkten tief eingebetteten Systeme. Der Abschnitt 2.2 beschäftigt sich mit einem Überblick zu dieser Problematik.

2.2 Datenmanagement in eingebetteten Systemen

Seit der Entwicklung der ersten Mikrocontroller Ende der 70er/ Anfang der 80er Jahre hat sich der Bereich der eingebetteten Systeme stark entwickelt [PR05]. Eingebettete Systeme besonders im Automobilbereich müssen eine steigende Zahl von Funktionen unterstützen [SZ06, Gri05]. Damit einher geht ein kontinuierlich steigendes Datenaufkommen [RLAS07]. Die dabei noch immer knappen Ressourcen machen ein effizientes Datenmanagement auch für diese Systeme notwendig [RLAS07].

Dieser Abschnitt beschäftigt sich mit dem Datenmanagement in eingebetteten Systemen. Dabei wird zuerst im Abschnitt 2.2.1 auf wichtige Begriffe eingegangen. Danach im Abschnitt 2.2.2 folgt eine Übersicht über die zu verwaltenden Daten in eingebetteten Systemen. Der Abschnitt 2.2.3 geht auf die Möglichkeit der Datenspeicherung in eingebetteten Systemen ein. Einige Datenbankmanagementsysteme für eingebettete Systeme

werden dann im Abschnitt 2.2.4 vorgestellt, darunter auch das für die Implementierung (Kapitel 5) und die Analyse (Kapitel 6) genutzte RobbyDBMS.

2.2.1 Begriffserklärung

Daten sind für den Rechner aufbereitete, abstrahierte Informationen, die codiert, gespeichert oder verarbeitet werden [FH08]. Diese Informationen werden zur Übersetzungszeit oder zur Laufzeit gewonnen und besitzen verschiedene Gültigkeitsbereiche. Diese Gültigkeitsbereiche können lokal einzelne Programmblöcke, Funktionen und Methoden, innerhalb von Klassen bei objektorientierter Programmierung oder global das ganze Programm betreffen [Wie07].

Das *Datenmanagement* ist ein Konzept zur Verwaltung von Daten. Dabei ist zu beachten, dass der Begriff *eingebettetes Datenmanagement* für ein in eine Anwendung integriertes Datenmanagement verwendet wird [RLAS07]. Eingebettete Systeme sind ein häufiges Anwendungsgebiet des eingebetteten Datenmanagements [RLAS07].

Eine *Datenbank* ist ein strukturiert gespeicherter Datenbestand, der von einem *Datenbankmanagementsystem (DBMS)* verwaltet wird [SSH08]. Ein DBMS ist somit eine Software, die eine Datenbank verwaltet [SSH08]. Die wichtigsten Eigenschaften dieser Verwaltung werden nach Codd in neun Regeln zusammengefasst [SSH08]:

1. Integration
2. Operationen
3. Katalog
4. Benutzersichten
5. Integritätssicherung
6. Datenschutz
7. Transaktionen
8. Synchronisation
9. Datensicherung

Diese Regeln werden im Bereich der DBMS für eingebettete Systeme nicht immer vollständig umgesetzt [RLAS07]. Gründe dafür sind in den Anwendungen und den daraus resultierenden Anforderungen an das Zielsystem zu finden [RLAS07].

Allgemein können Datenmanagementsysteme nach [RLAS07] auf Grundlage ihres Funktionsumfanges in verschiedene Gruppen eingeteilt werden. Abbildung 2.3 zeigt diese Einteilung und einige typische Anwendungsgebiete. Diese Arbeit beschäftigt sich hierbei mit einem DBMS, das in die Gruppe der Micro-Datenmanagementsysteme einzuordnen ist.

Ein Index ist im Kontext zu DBMS eine Zugriffsstruktur, die den Zugriff auf die Daten über Attribute unterstützt [SHS05]. Der Zugriff kann dabei über ein oder mehrere

	Datenmodelle/ Speicherformen	Anfragen	Einsatzgebiet	Typische Systeme
Macro	objektrelationales-, multidimensionales-, und relationales Datenmodell	SQL 3	Serversysteme	Oracle, IBM, MS SQL-Server
Mini	relationales Datenmodell	SQL 1 (Ad-hoc-Anfragen möglich)	Desktop, Laptop	MySQL, Oracle Lite
Micro	persistente Tabellen	1-Relationen Zugriff meist über eine API	PDA, Smartphones	Berkeley DB, RDM-Embedded, COMET DBMS
Nano	einfache persistente Speicherstrukturen (Array, Hash-Map,..)	API	(tief) eingebettete Systeme, Smartcards	Prevayler

Abbildung 2.3: Einteilung der Datenmanagementsysteme nach dem Funktionsumfang [RLAS07]

Attribute erfolgen [SHS05]. Unterstützt der Index den Zugriff über genau ein Attribut, so wird er eindimensional genannt [SHS05]. Ein Mehr-Attribut-Zugriff kann durch mehrere eindimensionale Indizes oder durch einen mehrdimensionalen Index unterstützt werden [SHS05]. Ein mehrdimensionaler Index unterstützt somit den Zugriff auf mehrere Attribute mit einer Zugriffsstruktur. Kapitel 4 beschäftigt sich vertiefend mit mehrdimensionalen Indexstrukturen.

2.2.2 Daten in eingebetteten Systemen

Die Daten in einem eingebetteten System werden grundsätzlich nach ihrer Veränderlichkeit klassifiziert [SZ06]. Die Unterscheidung in Variablen und nicht veränderliche Parameter ist möglich [SZ06]. Für die Datenhaltung sind beide Gruppen von Bedeutung, da beide im eingebetteten System verarbeitet, gespeichert und weitergeleitet werden. Daten liegen in verschiedenen Strukturen vor, als skalare Größen oder als komplexe aus skalaren Größen zusammengesetzte Strukturen [SZ06]. Die einfachsten zusammengesetzten Strukturen sind Vektoren (oft auch Felder genannt) und Matrizen [SZ06].

Skalare Größen werden in der Regel einem Datentyp zugeordnet. *Datentypen* dienen der richtigen Interpretation des Wertes [Wie07]. Die grundlegenden Basisdatentypen sind in Tabelle 2.1 aufgelistet. Wertebereich und Speicherbedarf der Datentypen sind abhängig von der verwendeten Programmiersprache. Viele Programmiersprachen ermöglichen die einfache Anordnung von skalaren Größen zu Feldern und Matrizen. Allgemein dienen zusammengesetzte Strukturen dazu, Beziehungen zwischen Daten abzubilden [SZ06]. Eine Schachtelung verschiedener zusammengesetzter Strukturen ist meist ebenfalls möglich. In der Programmiersprache C sind des Weiteren Zeiger auf Daten möglich [Wie07]. Die Zeiger werden in der Regel nach dem Datentyp des Ziels charakterisiert [Wie07]. Der Type „char“, der ursprünglich der Aufnahme von Zeichen dient, kann

in C auch zum Zählen verwendet werden und somit eine ganze Zahl ressourcenschonend speichern, da ein char einem Byte entspricht [Wie07].

Datentyp	Schlüsselwort	Wertebereich	Größe in Byte
ganze Zahlen			
vorzeichenbehaftet	int	$-32768 \ldots 32767$	2
	long int	$-2 \cdot 10^9 \ldots 2 \cdot 10^9$	4
vorzeichenlos	unsigned int	$0 \ldots 65535$	2
	long unsigned int	$0 \ldots 4 \cdot 10^9$	4
Gleitkommazahlen			
	float	$\pm 10^{-37} \ldots \pm 10^{38}$	4
	double	$\pm 10^{-307} \ldots \pm 10^{308}$	8
Zeichen			
vorzeichenbehaftet	char	$-128 \ldots 127$	1
vorzeichenlos	unsigned char	$0 \ldots 255$	1

Tabelle 2.1: Basisdatentypen mit Schlüsselwort, Wertebereich und Größe in C [Wie07]

In dieser Arbeit wurde die Programmiersprache C++ mit dem Cross-Compiler AVR-GCC verwendet. Alle entsprechenden Aussagen in dieser Arbeit beziehen sich auf diese Sprache. Tabelle 2.1 ordnet den Basisdatentypen die entsprechenden Wertebereiche und den jeweiligen Speicherbedarf zu. Mit der AVR-libc Bibliothek sind für ganzzahlige Datentypen zusätzlich ein, zwei, vier und acht Byte Varianten in vorzeichenbehafteter oder vorzeichenloser Form möglich. Diese werden mit „[u]intBitzahl_t“ initialisiert. Das „[u]“ steht dabei für unsigned und wird nur bei vorzeichenlosen Daten gebraucht. Die Bitzahl muss entsprechend eingetragen werden [1].

Um Daten effizient zu verwalten, ist es wichtig zu wissen, woher die Daten kommen, wozu sie genutzt werden und wie häufig sie benötigt werden [NTN+02]. In eingebetteten Systemen sind die Datenquellen in der Regel fest bestimmt [Lie08]. Datenquellen in eingebetteten Systemen sind beispielsweise:

- Sensoren,
- Aktoren,
- Benutzer,
- das System selbst,
- andere verbundene Systeme und
- die Programmierung beziehungsweise das Programm.

Aus diesen Datenquellen lassen sich wichtige Daten, die in einem eingebetteten System anfallen, ableiten. Hinzu kommen Daten, die für eine Analyse des Systems benötigt werden [NTN+02]. Einige Beispiele für anfallende Daten sind [Lie08, NTN+02]:

- Sensor- und Aktordaten,

[1] avr-libc 1.7.0 user manual, Juni 2010, Quelle: http://avr-libc.nongnu.org/

- Konfigurationsdaten,
- Kalibrierdaten,
- Wartungsdaten,
- Ereignisprotokolle,
- Störungsprotokolle,
- Verbindungsdaten,
- Daten des Programmablaufs und
- Programmdaten wie Referenzdaten.

Je nach Verwendung dieser Daten werden verschiedene Anforderungen an den Speicher gestellt [NTN+02]. Der folgende Abschnitt beschäftigt sich mit dieser Thematik.

2.2.3 Datenspeicherung in eingebetteten Systemen

Allgemein soll Speicher möglichst kostengünstig, leicht, schnell und energieeffizient sein. Einige Anwendungen fordern zudem eine nichtflüchtige Speicherung der Daten [SZ06]. Das heißt, dass die gespeicherten Daten auch nach dem Abschalten der Betriebsspannung nicht verloren gehen dürfen [SZ06]. Flüchtige Speicher benötigen im Gegensatz dazu die Spannung, um die Daten zu halten [SZ06].

Grundlegend müssen zur Laufzeit je nach Verwendung Daten nur gelesen, nur geschrieben oder gelesen und geschrieben werden können. Der lesende Zugriff kann auf alle Speichertechnologien erfolgen [SZ06]. Das Schreiben wird hier in das Schreiben eines Bits und das Löschen eines Bits unterschieden. Damit wird der Übergang von 0 zu 1 und andersherum beschrieben. Das Schreiben und Löschen von Bits wird von den verschiedenen Speichern unterschiedlich gut unterstützt [SZ06]. Da diese Aktionen für die Anwendung der Speichertechnologie von wesentlicher Bedeutung sind, ist eine Einteilung nach diesen Gesichtspunkten sinnvoll [SZ06]. Abbildung 2.4 zeigt solch eine Übersicht über verschiedene Speichertechnologien nach [SZ06].

ROM und PROM Diese beiden Technologien sind dabei nur für Daten geeignet, die nur einmal geschrieben werden und vom eingebetteten System nur gelesen werden [SZ06]. Eine Veränderung der Daten bedeutet den Austausch der Speichereinheit. Der PROM ist dabei auch nach der Herstellung mit einem speziellen Programmiergerät programmierbar [SZ06]. Dabei ist darauf zu achten, dass nur zusätzliche Daten geschrieben werden können. Das Löschen von Daten ist nicht möglich [SZ06].

EPROM Der EPROM ermöglicht es, unter großem Aufwand, Daten zu löschen und den frei gewordenen Speicher erneut zu nutzen [SZ06]. Dadurch ist es möglich, die Daten zu ändern, ohne eine neue Speichereinheit nutzen zu müssen. Das Löschen und Schreiben erfolgt auf einem speziellen Programmiergerät [SZ06]. Das heißt, der Speicher muss aus dem System ausgebaut werden. Eine schnelle Neuprogrammierung ist nicht möglich [SZ06]. Daher ist dieser Speicher nur für Daten einsetzbar, die extrem selten verändert werden müssen.

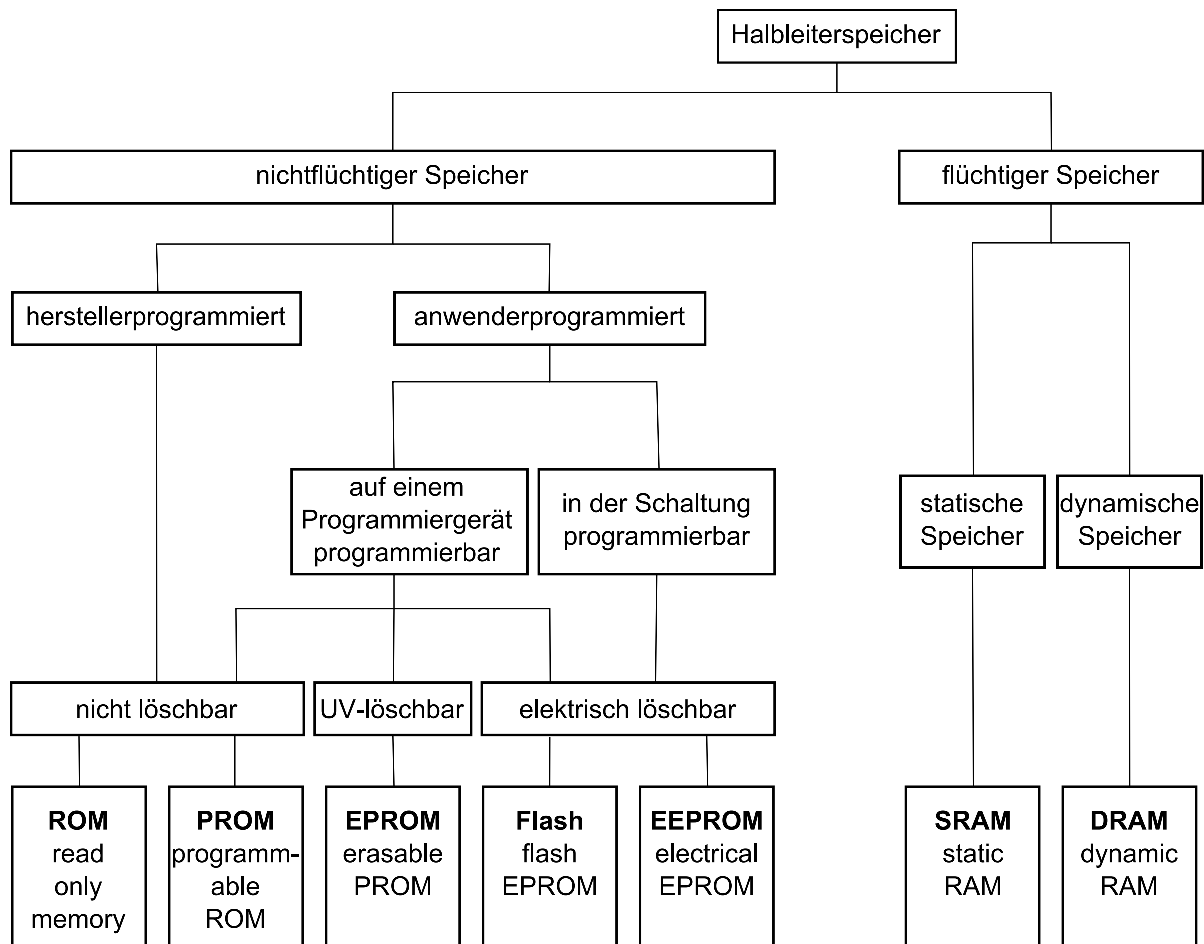


Abbildung 2.4: Übersicht von Speichertechnologien [SZ06]

EEPROM Der EEPROM kann auf dem eingebetteten System beschrieben und gelöscht werden [SZ06]. Dies erfolgt zeilenweise [SZ06]. Der EEPROM ist somit für veränderliche Daten geeignet, die fest gespeichert werden müssen [SZ06]. Lesen, Schreiben und Löschen sind wahlfrei und zeilenweise möglich, wobei eine Zeile einem Byte entsprechen kann [Lie08, Sch06]. Eine EEPROM-Speicherzelle kann nur begrenzt oft beschrieben werden, typischerweise zwischen 10.000 und 1.000.000-mal [Lie08].

Flash Diese Speichertechnologie ist eine Weiterentwicklung des EPROM und des EEPROM [SZ06]. Der Flash-Speicher wird grundlegend in zwei Kategorien unterteilt, den NAND- und den NOR-Flash [NK07, KBL+06]. Diese beiden Kategorien unterscheiden sich im Aufbau und der Adressierung voneinander [NK07, KBL+06]. Beide Flash-Speicher sind in Blöcke untergliedert [NK07, KBL+06]. Diese Blöcke sind wiederum in Seiten geteilt, welche eine bestimmte Anzahl an Speicherzellen enthalten [NK07, KBL+06]. Das Löschen entspricht dem Setzen von Bits auf 1 und erfolgt auf Blockebene [KBL+06]. Das Schreiben von Bits entspricht dem Setzen auf 0 [KBL+06]. Der NOR-Speicher unterstützt bitweises Schreiben und Lesen, während der NAND-Flash nur seitenweise arbeitet [KBL+06]. NOR-Speicher hat jedoch eine geringere Speicherkapazität als NAND-Speicher [NK07]. Eine Speicherzelle hat eine begrenzte Lebensdauer, ähnlich dem EEPROM, die sich nach der Zahl an Löschvorgängen richtet [KBL+06].

RAM Der RAM ist ein flüchtiger Kurzzeitspeicher und dient zum schnellen Lesen und Schreiben von Daten [SZ06]. Dynamischer RAM muss ständig neu beschrieben werden, während statischer RAM die Daten bis zum Neuschreiben oder zum Wegfall der Betriebsspannung halten kann [SZ06]. Bei Mikrocontrollern wird zwischen internem und externem RAM unterschieden [Wie07].

Der C-Compiler übernimmt einen Teil der Speicherverwaltung des RAM, indem er den Programmstack und die globalen Variablen im internen RAM organisiert [Wie07]. Der Heap enthält alle allokierten Speicherbereiche [Wie07]. Das heißt, er enthält alle dynamisch erzeugten Variablen. Der Heap kann sowohl im internen als auch im externen RAM liegen [Wie07]. Wenn der Heap im internen RAM liegt, kann es dazu kommen, dass der Stack in den Heap hinein wächst. Dies führt zu Fehlern im Programm und ist zu vermeiden. Des Weiteren ist zu beachten, dass es bei starker Nutzung des Heap zu einer Heap-Fragmentierung kommen kann [Wie07]. Spezielle Verfahren, um dem entgegen zu wirken, sind der Literatur zu entnehmen [Wie07].

Die Betrachtung des RAM ist für die Implementierung und Analyse der Indexstrukturen in Kapitel 5 und Kapitel 6 wichtig. Im Folgenden wird für den RAM auch der Begriff *„Arbeitsspeicher“* genutzt.

Die im Abschnitt 2.2.2 aufgeführten Beispiele für anfallende Daten können grundlegend nach dem benötigten Zugriff den Speichertechnologien zugeordnet werden. Diese Zuordnung ist jedoch nicht bindend. Eine Änderung ist je nach Hard- und Software, nach Anwendung und nach Ausprägung möglich. Tabelle 2.2 stellt die beispielhafte Zuordnung einiger Daten des Abschnittes 2.2.2 dar.

Daten	Zugriff	Technologie
Sensordaten	Lesend und Schreibend	RAM, EEPROM, ggf. Flash
Aktordaten	Lesend und Schreibend	RAM, EEPROM, ggf. Flash
Konfigurationsdaten	vornehmlich Lesend	EEPROM, Flash, ggf. PROM, EPROM
Wartungsdaten	vornehmlich Lesend	EEPROM, Flash, PROM, EPROM
Protokolle	vornehmlich Schreibend	EEPROM, ggf. Flash
Programmdaten	Lesend und Schreibend	RAM, EEPROM, ggf. Flash, ROM, PROM

Tabelle 2.2: Zuordnung Beispieldaten zu Speichertechnologie [NTN+02]

2.2.4 Datenbankenmanagementsysteme in eingebetteten Systemen

Eine Studie von Volvo aus dem Jahr 1998 ermittelte, einen Anstieg des Datenaufkommens in einem Fahrzeug von 7-10% pro Jahr [NTN+02]. Dieser enorme Anstieg von Daten erfordert ein effizientes Datenmanagement [RLAS07]. Hinzu kommt die steigende

Vernetzung von eingebetteten Systemen innerhalb des Produktes und mit anderen externen Rechnersystemen [NTN+02]. Bei gleichzeitigem Zugriff auf die Daten wird somit eine konsistente und effiziente Datenhaltung benötigt. Eine Möglichkeit dies zu gewährleisten, ist der Einsatz von Datenbanksystemen, wie sie im Abschnitt 2.2 definiert werden.

Herkömmliche Mehrzweck-DBMS können im Bereich der eingebetteten Systeme in der Regel nicht verwendet werden [RLAS07]. Gründe dafür sind in den knappen Ressourcen, der Heterogenität der Hard- und Software sowie den variablen Anforderungen zu sehen [RLAS07]. Das Problem der Ressourcenbeschränkungen wird sich in den nächsten Jahren trotz der steigenden Rechenleistung nicht wirklich lösen [RLAS07]. Dies ist damit zu begründen, dass mit steigender Leistung auch der Energieverbrauch und die Wärmeentwicklung steigen [RLAS07]. Zudem sind leistungsstärkere Systeme in der Regel teurer als die Leistungsschwächeren [RLAS07]. Somit sind spezielle DBMS-Lösungen für den Bereich der eingebetteten Systeme und tief eingebetteten Systeme notwendig [RLAS07].

Einige Vertreter von DBMS für eingebettete Systeme werden im Folgenden kurz vorgestellt. Dabei wird auf die Anwendungsgebiete, die Variabilität und die Architektur dieser Systeme eingegangen. Das Datenbankmanagementsystem RobbyDBMS wird hierbei besonders intensiv betrachtet, da es in dieser Arbeit für die Implementierung (Kapitel 5) und Analyse (Kapitel 6) mehrdimensionaler Indexstrukturen verwendet wird.

FlashDB

Diese DBMS-Lösung wurde von Microsoft Research für die Nutzung von NAND-Flash-Speicher entwickelt [NK07]. Die Anwendung ist auf Sensornetzwerke beschränkt [NK07].

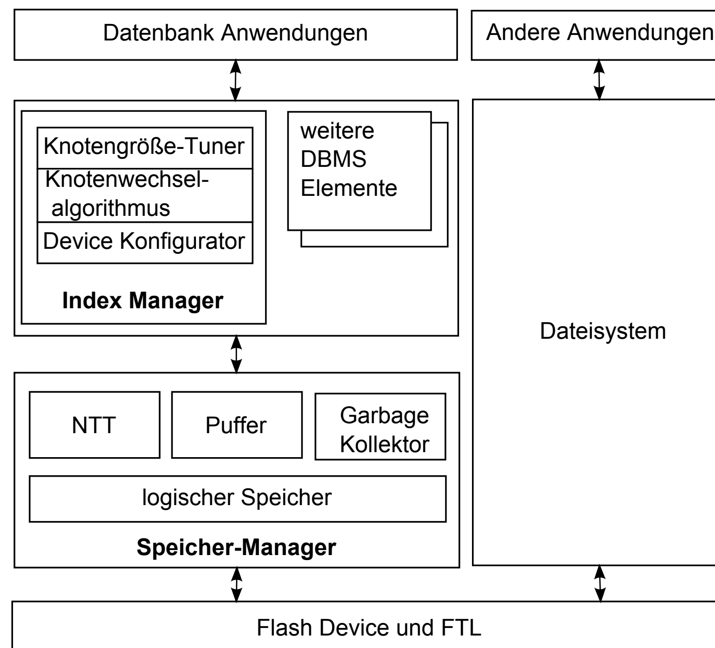


Abbildung 2.5: Aufbau von FlashDB [NK07]

FlashDB passt sich dynamisch an den geforderten Leistungsumfang und das Speichergerät an [NK07]. Der Leistungsumfang bezieht sich dabei auf das Verhältnis der Nutzung von Lese- und Schreibzugriffen. Das Verhältnis des Zeitaufwands für Lese- und

Schreibzugriffe geht über das Speichergerät ein [NK07]. Als Index wird ein modifizierter B^+-Baum verwendet [NK07]. Ein Knoten des Baums kann dabei entweder im Disk-Modus oder im Log-Modus sein [NK07]. Ein im Disk-Modus befindlicher Knoten wird als ein Datenblock abspeichert. Dieser Datenblock kann sich dabei über mehrere Speicherseiten und -blöcke erstrecken [NK07]. Dies führt zu einem großen Aufwand bei der Modifikation der Daten [NK07]. Ist der Knoten im Log-Modus, so wird er log-structured abgespeichert. Das bedeutet, der Knoten wird wie ein Transaktionsprotokoll organisiert. Dies führt zu einem geringeren Aufwand bei der Modifikation der Daten [NK07]. Jedoch verursacht das Lesen von Daten einen höheren Aufwand [NK07]. Aus dem aktuellen Leistungsumfang und dem genutzten Speichergerät wird dann für jeden neuen Knoten der Modus bestimmt [NK07].

FlashDB unterstützt Lookups, eindimensionale und mehrdimensionale Bereichsanfragen sowie Verbundanfragen [NK07]. Der Speicherplatzbedarf ist mit 6 KB für 30.000 Tupel, unabhängig von der Tupelgröße, relativ gering [NK07]. Der Aufbau von FlashDB ist in Abbildung 2.5 schematisch dargestellt. Daran ist die Realisierung der Indexstruktur als ein zentraler Bestandteil in den Komponenten des DBMS zu erkennen [NK07].

Ein mehrdimensionaler Index wird nicht verwendet. Die Hauptmerkmale von FlashDB sind der modifizierte B^+-Baum und die Nutzung des NAND-Flash-Speichers [NK07]. Eine Erweiterung um mehrdimensionale Indexstrukturen ist daher für diese DBMS-Lösung nicht zielführend.

COMET DBMS

COMET DBMS ist ein DBMS für den automotiven Einsatz [Nys03, NTNH03, Nys05]. Zu diesem Zweck besteht COMET DBMS aus einer Sammlung von Werkzeugen, mit denen ein Echtzeit-Datenbankmanagementsystem generiert werden kann [NTNH03]. Die Software kann in einer heterogenen Umgebung arbeiten. Die Echtzeitanforderungen werden direkt im Design mit Hilfe von Konfigurationsparametern umgesetzt [Lie08].

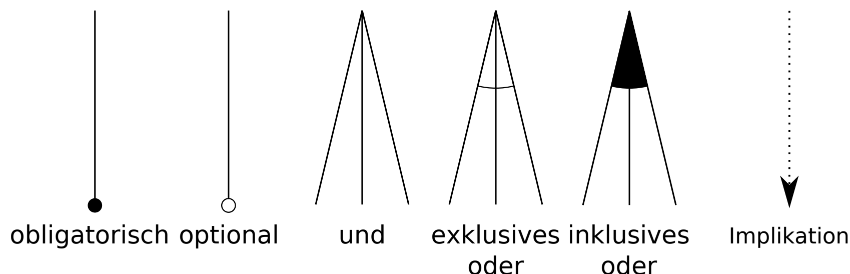


Abbildung 2.6: Symbole eines Feature-Diagramms [CE99]

Die Abkürzung COMET DBMS steht für „componenten-based embedded real-time database management system“ [NTNH03]. Die Software ist aus vordefinierten Softwarebausteinen aufgebaut [NTNH03]. Diese sind obligatorisch, das heißt verpflichtend, oder optional (nicht verpflichtend) für die Generierung des maßgeschneiderten DBMS-Lösung und realisieren in der Regel je genau ein Merkmal. Die Bausteine werden in einem Feature-Diagramm dargestellt. Die Bedeutung der Symbole des Feature-Diagramms sind in Abbildung 2.6 dargestellt. Das Feature-Diagramm in Abbildung 2.7 veranschaulicht somit den Aufbau und die möglichen Merkmale des Systems.

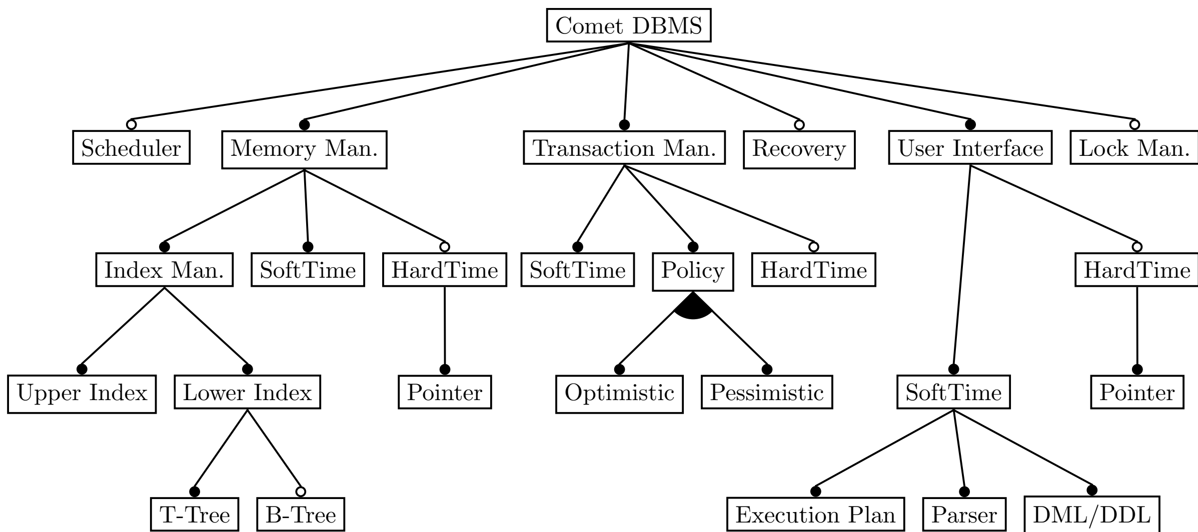


Abbildung 2.7: Feature-Diagramm COMET DBMS [Lie08]

Die Basisimplementierung benötigt einen Speicherplatz von rund 20 KB [Nys03, Nys05]. Darin sind eine relationale Anfragesprache, eine Transaktionsverwaltung und eine Speicherverwaltung enthalten. Viele der Merkmale sind obligatorisch, das heißt verpflichtend [Nys03, Nys05].

Die implementierten Indexstrukturen sind eindimensional [Lie08]. Eine Erweiterung um mehrdimensionale Indexstrukturen ist möglich, wird in dieser Arbeit aufgrund des beschränkten Anwendungsgebietes nicht vorgenommen.

FAME-DBMS

FAME-DBMS entstand in einem von der German Resarch Foundation gegründeten Forschungsprojekt. An dem Projekt waren die Otto-von-Guericke Universität Magdeburg, die Technische Universität Dortmund, die Friedrich-Alexander Universität Erlangen-Nürnberg, die Universität Passau und die METOP GmbH beteiligt [2].

Das Ziel des FAME-DBMS Projektes war, Techniken und Werkzeuge zur Entwicklung und Anpassung von DBMS zu entwickeln, zu erweitern und zu bewerten [RSS+08]. Für die Entwicklung wurde ein Softwareproduktlinien-Ansatz gewählt, der eine hohe Zahl an Konfigurationen ermöglicht [RSS+08]. Die Umsetzung fand in Form einer Programmfamilie statt. Die Programmierung erfolgte feature- und aspekt-orientiert [RSS+08]. Bei der *feature-orientierten Programmierung* steht das Feature im Mittelpunkt [Lie08]. Ein *Feature* ist ein Merkmal, das die Spezifikation und Unterscheidung von Software ermöglicht [Ape07, CE00, KCH+90]. Es ist dabei in der Regel direkt aus den Anforderungen abgeleitet [Ape07, CE00, KCH+90]. Die *aspekt-orientierte Programmierung* beschäftigt sich mit der Kapselung von Programmfunktionalitäten [KLM+97, Spi02]. Besonderer Wert wird dabei auf sich überschneidende Funktionen gelegt [KLM+97, Spi02]. Als Programmiersprachen wurden die Spracherweiterungen FeatureC++ und AspectC++ von C++ verwendet [Lie08]. Das Feature-Diagramm ist in Abbildung 2.8 zu sehen. Die Bedeutung der Symbole des Feature-Diagramms sind in Abbildung 2.6 (Seite 18) dargestellt. Eine

[2] http://fame-dbms.org

Komposition verschiedener Features gemäß dem Feature-Diagramm zu einem funktionsfähigen Programm wird im folgenden *„Ausprägung“* genannt. Nähere Erläuterungen zur feature-orientierten Programmierung folgen im Kapitel 5.

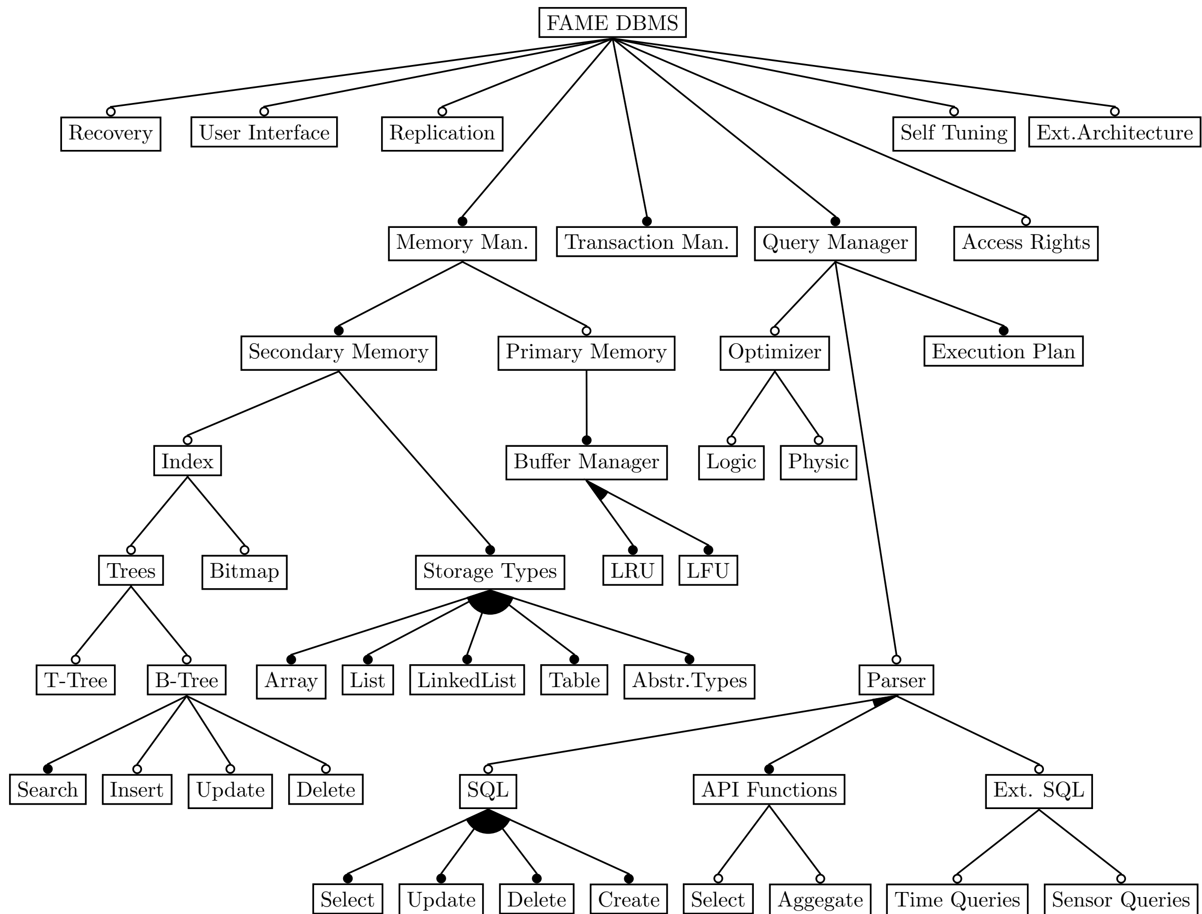


Abbildung 2.8: Feature-Diagramm FAME-DBMS [Lie08]

Die DBMS-Lösung ist für verschiedene Zielanwendungen und -systeme einsetzbar [Lie08, RSS+08]. Das heißt, die Software ist auf kein Anwendungsszenario ausgerichtet. Dies wird durch die vielen Konfigurationsmöglichkeiten erreicht [Lie08]. FAME-DBMS bildet dabei die Basisimplementierung für eine Softwareanwendung, die durch zusätzliche Implementierungen gemäß den Anforderungen der Anwendung und des Zielsystems erweitert wird [Lie08]. Als Beispielzielsysteme sind Windows, Linux und NutOS implementiert [Lie08]. Damit kann FAME-DBMS auf eingebetteten Systemen wie auch auf anderen Rechnersystemen genutzt werden [Lie08, RSS+08]. Dieser Umstand führt jedoch dazu, dass die Anforderungen an tief eingebettete Systeme nicht immer konsequent umgesetzt werden konnten [Lie08].

RobbyDBMS

Mit *RobbyDBMS* wird der Ansatz von FAME-DBMS weiter verfolgt [Lie08]. Auch dieses DBMS ist modular aufgebaut. Abbildung 2.9 zeigt das Feature-Diagramm von RobbyDBMS ohne die Erweiterungen dieser Arbeit. RobbyDBMS ist darauf ausgelegt, möglichst viele Merkmale optional (nicht verpflichtend) zu realisieren [Lie08]. Da-

durch und durch eine sparsame Programmierung wird der Speicherplatzbedarf minimiert [Lie08]. Eine voll ausgestattete Ausprägung von RobbyDBMS benötigt rund 6045 Byte [Lie08]. Somit ist RobbyDBMS für tief eingebettete Systeme geeignet [Lie08].

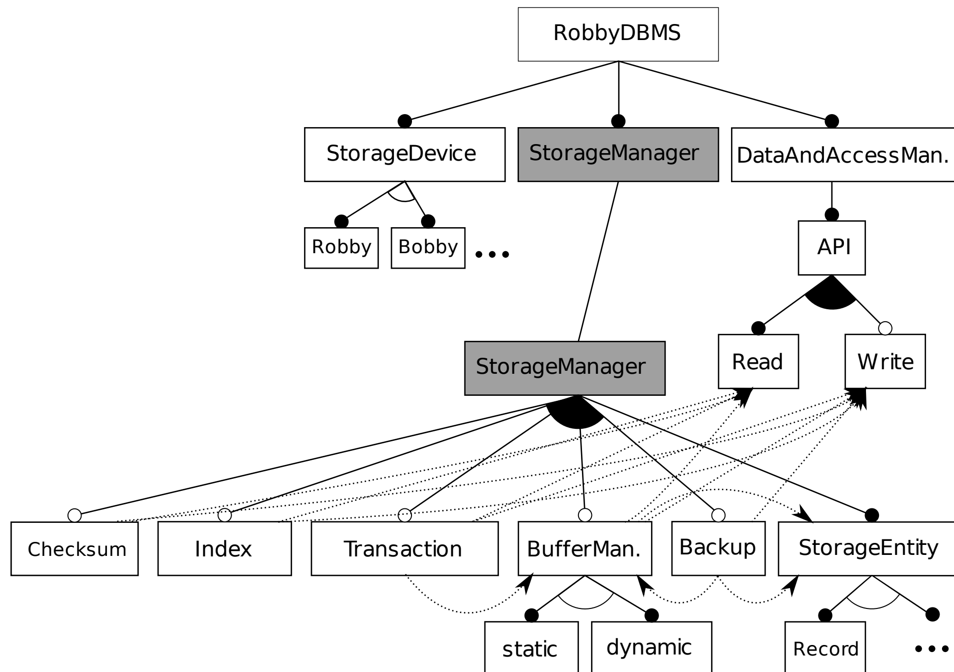


Abbildung 2.9: Feature-Diagramm RobbyDBMS ohne mehrdimensionale Indexstrukturen und Anfragen [Lie08]

RobbyDBMS von Liebig besteht aus 16 Features, die optional oder obligatorisch sind [Lie08]. Einige dieser Features unterteilen sich in weitere Features. Zwischen den einzelnen Merkmalen bestehen Abhängigkeiten [Lie08]. Diese sind in der Abbildung 2.9 durch Pfeile gekennzeichnet. Die Abhängigkeiten erfordern zudem bei einigen Merkmalen eine geteilte Implementierung [Lie08]. Ein Beispiel dafür sind die Features „Index“, „Read“ und „Write“. Da „Write“ optional ist, muss das Feature „Index“ in eine „Read“ und eine „Write“ Komponente geteilt werden [Lie08].

Bei der Programmierung von RobbyDBMS konnte auf aspektorientierte Programmierung verzichtet werden [Lie08]. Daher wurde nur FeatureC++ als Spracherweiterung von C++ und zur Umsetzung des modularen Aufbaus eingesetzt [Lie08]. Weitere Erläuterungen zur feature-orientierten Programmierung von RobbyDBMS folgen in Kapitel 5.

Der Aufbau von RobbyDBMS orientiert sich am Fünf-Schichten-Modell von Härder [Lie08]. Es konnte jedoch aufgrund der Anforderungen und bestimmter getroffener Annahmen, wie der als fest angenommenen Anzahl an Informationen, umfangsmäßig zu einem Drei-Schichten-Modell vereinfacht werden [Lie08]. Dieses vereinfachte Schichtenmodell ist in Abbildung 2.10 dargestellt. Der Zugriff auf die Datenbank erfolgt nicht über eine Anfragesprache, sondern über eine API. Das Speichersystem regelt das Lesen und Speichern der Daten und das Betriebssystem realisiert den Zugriff auf die Hardwarekomponenten [Lie08]. RobbyDBMS nutzt einen EEPROM zur Speicherung der Datensätze [Lie08].

Das von Liebig entwickelte RobbyDBMS unterstützt als Indexstruktur nur das eindimensionale, lineare Hashen [Lie08]. Anfragen sind nur in Form von Exact-Match-

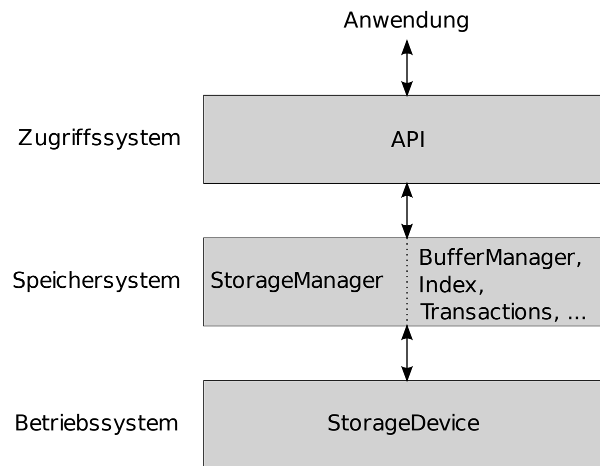


Abbildung 2.10: Schichtenmodell von RobbyDBMS [Lie08]

Anfragen über die TID möglich [Lie08]. Auf dieser Grundlage wurden die Transaktionsverwaltung und die Pufferverwaltung implementiert. Zur Unterstützung weiterer Anfragen ist eine Veränderung dieser Features jedoch nicht zwingend notwendig, da beide Komponenten optional sind [Lie08]. Dies macht die Erweiterung von RobbyDBMS um mehrdimensionale Anfragen und unterstützende Indexstrukturen einfacher. Im Kapitel 5 wird die Erweiterung von RobbyDBMS um mehrdimensionale Anfragen und unterstützende Indexstrukturen erläutert.

Kapitel 3

Nutzung mehrdimensionaler Daten

Ziel dieses Kapitels ist herauszufinden, inwieweit mehrdimensionale Daten in Anwendungen von eingebetteten Systemen genutzt werden. Zu diesem Zweck wird in Abschnitt 3.1 einführend auf mehrdimensionale Daten eingegangen. Der darauf folgende Abschnitt 3.2 beschäftigt sich mit Datenbankmanagementsystemen, die auf bestimmte Arten mehrdimensionaler Daten in klassischen Rechnersystemen ausgelegt sind. Aus den Anwendungsgebieten dieser DBMS lassen sich mögliche Parallelen zu Anwendungen in eingebetteten Systemen ziehen. Beispielhaft wird in Abschnitt 3.3 auf Anwendungen mehrdimensionaler Daten in eingebetteten Systemen eingegangen. Alle betrachteten Systeme werden als tief eingebettete Systeme, wie sie im Abschnitt 2.1.1 definiert werden, betrachtet, da eine Ressourcenbeschränkung in den Beispielsystemen angenommen wird. Dies ist durch die allgemeinen Anforderungen aus Abschnitt 2.1.3 und den Charakter der beschriebenen Systeme begründed. Mit Hilfe dieser Anwendungsbeispiele wird im Abschnitt 3.4 eine Analyse der Anforderungen an die Verwaltung der mehrdimensionalen Daten durchgeführt.

3.1 Mehrdimensionale Daten

Daten wurden in Abschnitt 2.2.1 als abstrahierte Informationen definiert. Die Struktur dieser Informationen ist, wie in Abschnitt 2.2.2 beschrieben, einfach oder komplex. Komplexe Daten bestehen aus mehreren, in Beziehung stehenden Informationen [SZ06]. Diese Informationen können als Merkmale oder Attribute bezeichnet werden. Eine Teilmenge der Attribute kann zum Vergleich verschiedener Daten genutzt werden und als *„Suchattribute"* bezeichnet werden. Die Anzahl dieser Suchattribute bestimmt die Anzahl der Dimensionen des übergeordneten Datums [Sch05a]. Der Werte eines Suchattributs entspricht dabei dem Wert in einer Dimension [Sch05a]. Das übergeordnete Datum wird entsprechend der Anzahl an Dimensionen als *„mehrdimensional"* bezeichnet. Bei einer sehr großen Anzahl von Merkmalen wird auch der Begriff *„hochdimensional"* verwendet [Sch05a].

Beispiele für mehrdimensionale Daten sind Raumdaten, die zwei oder drei Dimensionen haben. Die Attribute sind entsprechend der Achsen, x-, y- und z-Werte [Pag96]. Hochdimensionale Daten sind beispielsweise Bild-, Video- und Audiodaten, die in der Regel eine große Menge von Merkmalen besitzen [Sch05a].

Mehrdimensionale Daten erfordern je nach Anwendung eine kurzfristige oder eine

permanente Speicherung. Ist die Menge der permanent zu speichernden Daten groß oder unterliegt bestimmten Kriterien wie zum Beispiel der Kosistenz der Daten beim Mehrbenutzerbetrieb, bietet sich eine Speicherung in einer Datenbank an [Sch05a]. Der folgende Abschnitt beschäftigt sich mit speziellen DBMS für mehrdimensionale Daten in klassischen Rechnersystemen. Dabei wird auf die Daten und Anwendungen der Systeme eingegangen.

3.2 Mehrdimensionale Daten in klassischen Rechnersystemen

In klassischen Rechnersystemen ist eine Vielzahl von verschiedenen mehrdimensionalen Daten zu finden. Beispiele dafür sind Audiodaten, Texte, Bilder, Programmcode und Kontaktdaten, die jeweils mehrere Merkmale besitzen. In klassischen Rechnersystemen werden häufig Datenbanken zur Verwaltung von Daten genutzt [Sch05a]. Für einige mehrdimensionale Daten, wie Multimediadaten, ist die Verwendung von klassischen Datenbanksystemen häufig ungeeignet, da spezielle Anfragen auf den Daten nicht optimal unterstützt werden [Sch05a]. Aus diesem Grund werden häufig spezielle Datenbanksysteme genutzt [Sch05a]. Für Geodaten sind dies beispielsweise Geodatenbanken und für Multimediadaten Multimedia-Datenbanken [Pag96, Sch05a]. Diese speziellen DBMS-Lösungen verwenden häufig mehrdimensionale Indexstrukturen, um einen effizienten Zugriff zu ermöglichen [Pag96, Sch05a]. Daher werden im Folgenden diese beiden Datenbanksysteme betrachtet.

3.2.1 Geodatenbanksysteme

Geodatenbanksysteme verwalten in der Regel eine große Menge georeferenzierter Daten [Pag96]. Georeferenzierte Daten, auch Geo-Objekte genannt, bestehen aus geometrischen Attributen und Sachattributen [Pag96]. Die geometrischen Attribute dienen der Form- und Lagebeschreibung, während über die Sachattribute alle weiteren Eigenschaften beschrieben werden [Pag96]. Geodatenbanksysteme unterstützen spezielle Anfragen, die über Exact-Match-Anfragen nach dem Schlüssel hinausgehen [Pag96]. Beispiele für raumbezogene Anfragen sind räumliche Bereichsanfragen und Nächste-Nachbarsuchen. Es sind jedoch in vielen Geodatenbanksystemen auch kombinierte Anfragen über Raum- und Sachattribute möglich [Pag96]. Nähere Erläuterungen zu Anfragearten sind im Abschnitt 4.2 zu finden.

Geodatenbanksysteme dienen der effizienten, sicheren und redundanzfreien Verwaltung der Daten ebenso wie der Unterstützung eines schnellen, fachbezogenen Zugriffs [Pag96]. Zu diesem Zweck werden in Geodatenbanksystemen häufig spezielle, mehrdimensionale Indexstrukturen wie der R-Baum und der LSD-Baum, die in Kapitel 4 vorgestellt werden, verwendet [Pag96].

Anwendungsgebiete von Geodatenbanksystemen sind beispielsweise die Geo- und Umweltwissenschaften, der Katastrophenschutz, Versicherungen und im Touristikbereich [Bre07]. Neuere Anwendungsgebiete sind Frühwarnsysteme für Naturereignisse, Navigationssysteme und Fernerkundungsanwendungen wie Google Earth [Bre07].

3.2.2 Multimedia-Datenbanken

Multimedia-Datenbanken wurden zur Anpassung an die speziellen Herausforderungen mehrdimensionaler Mediendaten entwickelt [Sch05a]. Mediendaten sind unter anderem Texte, Grafiken, Bilder, Audiodaten und Videodaten. Die Suche solcher Daten erfolgt oft inhaltsbasiert über die Ähnlichkeit zu gegebenen Werten [Sch05a]. Daher sind klassische relationale Datenbanken nicht optimal zur Verwaltung geeignet [Sch05a]. Information-Retrieval-Systeme ermöglichen solch eine inhaltsbasierte Suche, besitzen jedoch viele der geforderten Datenbankfunktionalitäten, wie Anfragesprachen, nicht [Sch05a]. Die Kombination beider Techniken ist eine Herausforderung der Multimedia-Datenbanken [Sch05a].

Multimediadaten, wie Videodaten, sind häufig hochdimensional [Sch05a]. Dies ist durch ihre meist komplexe Informationsstruktur zu begründen, die zu einer hohen Zahl an Merkmalen führt [Sch05a]. Wichtige Anfragearten sind dabei die Nächste-Nachbarsuche, die Bereichssuche, die mehrdimensionale Exact-Match-Suche und Partial-Match-Suche [Sch05a]. Diese Anfragen werden im Abschnitt 4.2 näher erläutert.

Die Anwendungen von Multimedia-Datenbanken sind vielfältig [Sch05a]. Dies ist unter anderem in dem variablen Datenformat begründet [Sch05a]. Einige Beispielanwendungen sind multimediale Enzyklopädien, Umweltinformationssysteme, digitale Bibliotheken, Lernsoftware, medizinische Informationssysteme und Anwendungen des Teleshopping-Bereichs [Sch05a].

Die Daten in den einzelnen Anwendungen sind in der Regel sehr komplex [Sch05a]. Dadurch und durch die große Datenmenge sind spezielle Indexstrukturen für einen effizienten Zugriff notwendig [Sch05a]. Beispiele für solche Indexstrukturen sind der R-Baum als ein Vertreter der Baumverfahren und das VA-File als ein Vertreter der Signaturverfahren [Sch05a]. Beide Indexstrukturen werden im Kapitel 4 vorgestellt.

3.3 Anwendungen in eingebetteten Systemen

Dieser Abschnitt untersucht, inwieweit mehrdimensionale Daten in eingebetteten Systemen von Bedeutung sind. Allgemein steigt das Datenaufkommen in eingebetteten Systemen stetig an [RLAS07]. Besonders ist dieser Trend im Bereich der automotiven Anwendungen sichtbar, mit einer Steigerung des Datenaufkommens um 7 bis 10% jährlich [NTN+02]. Ein weiterer aktueller Trend bei der Entwicklung eingebetteter Systeme ist in Richtung intelligenter, eingebetteter Systeme zu beobachten [Nau05]. Diese Systeme benötigen oft, beispielsweise bei der Mustererkennung, große Mengen an Daten. [Nau05] Im Folgenden werden daher intelligente, eingebettete Systeme sowie eingebettete Systeme im Auto als Beispiele herangezogen.

Der Abschnitt 3.3.1 beschäftigt sich mit intelligenten, eingebetteten Systemen. Dabei werden zuerst wichtige Begriffe geklärt und es wird auf die Mustererkennung eingegangen. Danach werden konkrete Anwendungsbeispiele für Mustererkennung in eingebetteten Systemen beschrieben. Dabei wird auch auf die zu speichernden Daten eingegangen. Im Abschnitt 3.3.2 wird zu Beginn allgemein auf die Verwendung eingebetteter Systeme im Auto eingegangen. Darauf folgt eine Betrachtung von eingebetteten Systemen des Autos, die mehrdimensionale Daten nutzen.

3.3.1 Intelligente, eingebettete Systeme

Intelligente, eingebettete Systeme sind eingebettete Systeme, auf denen Algorithmen aus den Bereichen der künstlichen Intelligenz und der Mustererkennung implementiert sind [Nau05]. Typische Beispiele sind sprach- und bilderkennende sowie lernende und autonome Systeme [Nau05].

Bezüglich mehrdimensionaler Daten ist hier der Bereich der *Mustererkennung* besonders interessant, da zur Erkennung von Mustern Referenzdaten benötigt werden und diese häufig mehrdimensional sind [Nau05]. Mustererkennung wird definiert als:

> *„Die Analyse eines komplexen Signals (=Muster), um Objekte samt ihren Eigenschaften zu erkennen oder um dessen Bedeutung im Zusammenhang mit einer Aufgabe und Umgebungsbedingungen zu verstehen.“ [Nau05]*

Komplexe Signale sind in der Regel mehrdimensionale Daten, die mehrere unterscheidbare Merkmale aufweisen. Abbildung 3.1 zeigt den Ablauf einer Mustererkennung am Beispiel von Bilddaten. Grundsätzlich wird zwischen der Lern- beziehungsweise Parametrisierungsphase und der Erkennungsphase unterschieden [Nau05].

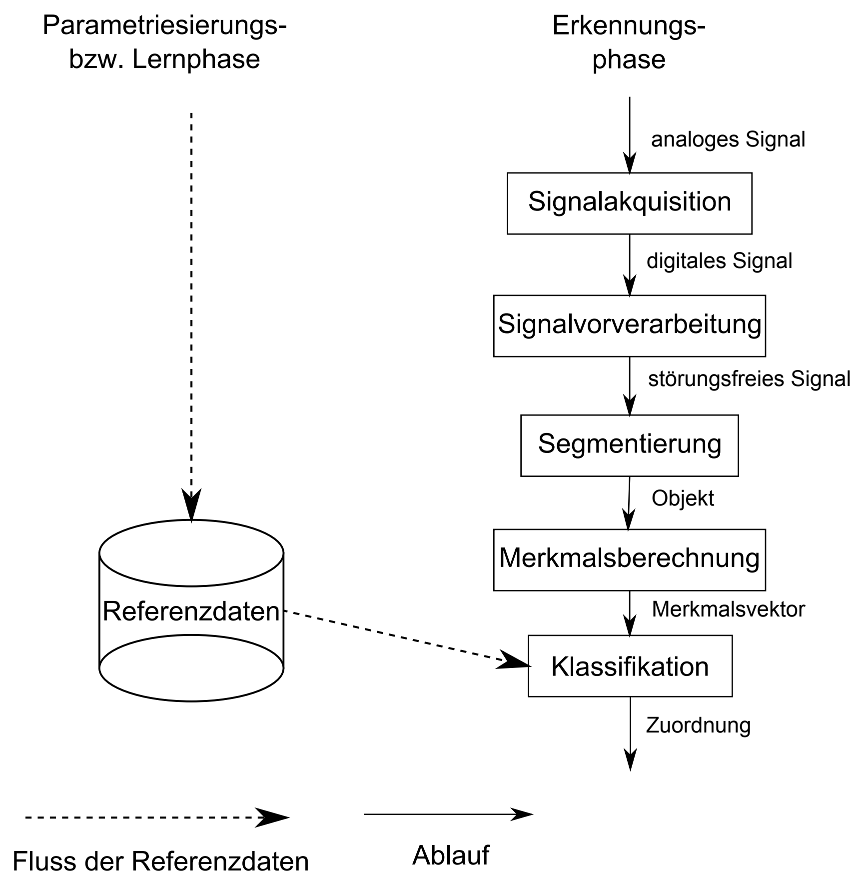


Abbildung 3.1: Ablauf einer Mustererkennung am Beispiel von Bilddaten nach [Nau05]

In der Lern- beziehungsweise Parametrisierungsphase werden die notwendigen Daten für die Klassifikation festgelegt [Nau05]. Eine Parametrisierung erfolgt bei arithmetisch beschreibbaren Unterscheidungsmerkmalen [Nau05]. Während das Lernen für vorher unbekannte Muster oder Merkmale mit statistischen Variationen genutzt wird [Nau05].

Ein Beispiel dafür sind gesprochene Wörter, die das System lernt zu unterscheiden. Diese Phase beeinflusst die Referenzdaten, die in einer Datenbank abgelegt sein können [Nau05, Mar07]. Die Referenzdaten liegen in der Regel in Form von Vektoren vor und werden daher auch als Referenzvektoren bezeichnet [Nau05].

Die Erkennungsphase gliedert sich in die Signalakquisition, die Vorverarbeitung, die Segmentierung, die Merkmalsberechnung und die Klassifikation [Nau05]. Bei der Signalakquisition erfolgt die Umwandlung von physikalischen, analogen Signalen in elektrische, digitale Signale [Nau05]. Danach kann eine Vorverarbeitung erfolgen, bei der Störungen oder Inhomogenitäten entfernt werden können [Nau05]. Dies erfolgt beispielsweise durch Filter. Die Segmentierung trennt dann die wichtigen Informationen für die Erkennung vom Rest ab [Nau05]. Ein Beispiel hierfür ist das Schwellwertverfahren, bei dem Schwellwerte zur Abgrenzung genutzt werden [Nau05]. Das Ergebnis diese Phase sind ein oder mehrere Objekte. Bei der Merkmalsberechnung wird dann von jedem Objekt ein Merkmalsvektor gebildet [Nau05]. Dieser enthält n Merkmale, die möglichst signifikant sein müssen, da diese zur Klassifikation genutzt werden [Nau05]. Die Auswahl der Erkennungsparameter basiert auf empirischen Auswertungen [Nau05]. Bei der Klassifizierung erfolgt die Zuordnung von Objekten anhand der Merkmale zu Mustern [Nau05]. Die Hauptoperation ist dabei ein Vergleich zwischen den Merkmalsvektoren und den Referenzvektoren [Nau05].

Um eine zuverlässige Zuordnung zu ermöglichen, ist eine ausreichende Zahl von Referenzvektoren für jedes Muster erforderlich [Nau05]. Bei der statistischen Optimalklassifikation wird aus den Referenzvektoren ein Mittelvektor ermittelt und gemeinsam mit einem Diskriminanzgrenzwert gespeichert [Nau05]. Dieser Grenzwert gibt den Randbereich des Musters an [Nau05]. In der Erkennungsphase wird für jeden Merkmalsvektor und jedem Mittelwertvektor die Diskriminanz berechnet [Nau05]. Das Objekt wird dabei dem Muster zugeordnet, zu welchem es die geringste Diskriminanz hat und bei welchem die Diskriminanz kleiner als der entsprechende Diskriminanzgrenzwert ist [Nau05]. Wird dieser Grenzwert für alle Muster überschritten, gilt das Objekt als nicht erkannt [Nau05].

Die Diskriminanz kann auch als Distanz zwischen zwei Vektoren aufgefasst werden. Wobei der nächste Nachbar zu einem gegebenen Merkmalsvektor gesucht wird und die Distanz einen bestimmten Grenzwert nicht überschreiten darf. Die Nächste-Nachbarsuche ist dabei direkt in einer Datenbank möglich und kann durch entsprechende mehrdimensionale Indexstrukturen unterstützt werden.

Mustererkennung wird in eingebetteten Systemen zum einen zur Steuerung und zum anderen zur Erfüllung der Funktionalität genutzt [Nau05]. Das heißt, die Steuerung kann beispielsweise durch Spracherkennung erfolgen [Nau05]. Die Funktionalität kann zum Beispiel durch Bilderkennung zum Teil realisiert werden [Nau05]. Es sind jedoch auch weitere komplexe Anwendungsszenarien möglich. Beispiele dafür sind Farb- und Formerkennung von Produkten für vielfältige Zwecke und Verhaltensmustererkennung in der Schnittstelle zwischen Medizin- und intelligenter Gebäudetechnik [Nau05, Röh10].

Farb- und Formerkennung Das Erkennen von Farben und Formen wird in vielfältigen Anwendungen genutzt, um die Erfüllung von Aufgaben zu unterstützen [Nau05]. Eine mögliche Anwendung in der pharmazeutischen Industrie ist das Erkennen, Überprüfen und Sortieren von Tabletten und Ampullen [Lin06]. So werden falsche oder beschädigte Tabletten erkannt und aussortiert [Lin06]. Ampullen werden dabei oft durch Farbringe gekennzeichnet. Werden die Farben und die Ring-

kombination richtig erkannt, so erfolgt eine Klassifizierung der Ampulle [Nau05]. Weitere Anwendungen sind das Sortieren und Zuordnen von Bauteilen und Kabeln in der Elektronik- und in der Automobilfertigung [Nau05]. Farberkennen kann aber auch zur Qualitätssicherung eingesetzt werden [Lin06]. Anwendungsbereiche sind hier beispielsweise die Textil- und Druckindustrie [Lin06].

Bei der Erkennung der Farbe wird ein dreidimensionaler Vektor benötigt, der aus den Grundfarben Rot, Grün und Blau besteht [Nau05]. Jeder Farbe werden dabei die entsprechenden Anteile der Grundfarben zugeordnet [Nau05]. Farbsensoren bestimmen aus den Aufnahmen einer gegebenen Farbe diese Anteile. Die Mustererkennung ordnet dann die entsprechende Merkmalskombination einer gespeicherten Referenzfarbe zu [Nau05]. Somit ist bei der Farberkennung ein Zugriff auf gespeicherte dreidimensionale Daten erforderlich. Kommen dazu noch Formmerkmale, Farbkombinationen oder weitere Merkmale, so wird in einigen Fällen von hochdimensionalen Daten gesprochen.

Verhaltensmustererkennung Die Erkennung von Verhaltensmustern kann besonders im Bereich der Betreuung von behinderten und alten Menschen eine gute Grundlage zur Unterstützung dieser Menschen und zur Erkennung von Notfallsituationen sein [Röh10]. Diese Anwendung kann innerhalb einer intelligenten Gebäudetechnik durch ein Sensornetzwerk realisiert werden [Röh10]. Abbildung 3.2 zeigt den möglichen Aufbau eines solchen Sensornetzwerkes. Es besteht dabei aus fest installierten sowie mobilen Sensorknoten [Röh10].

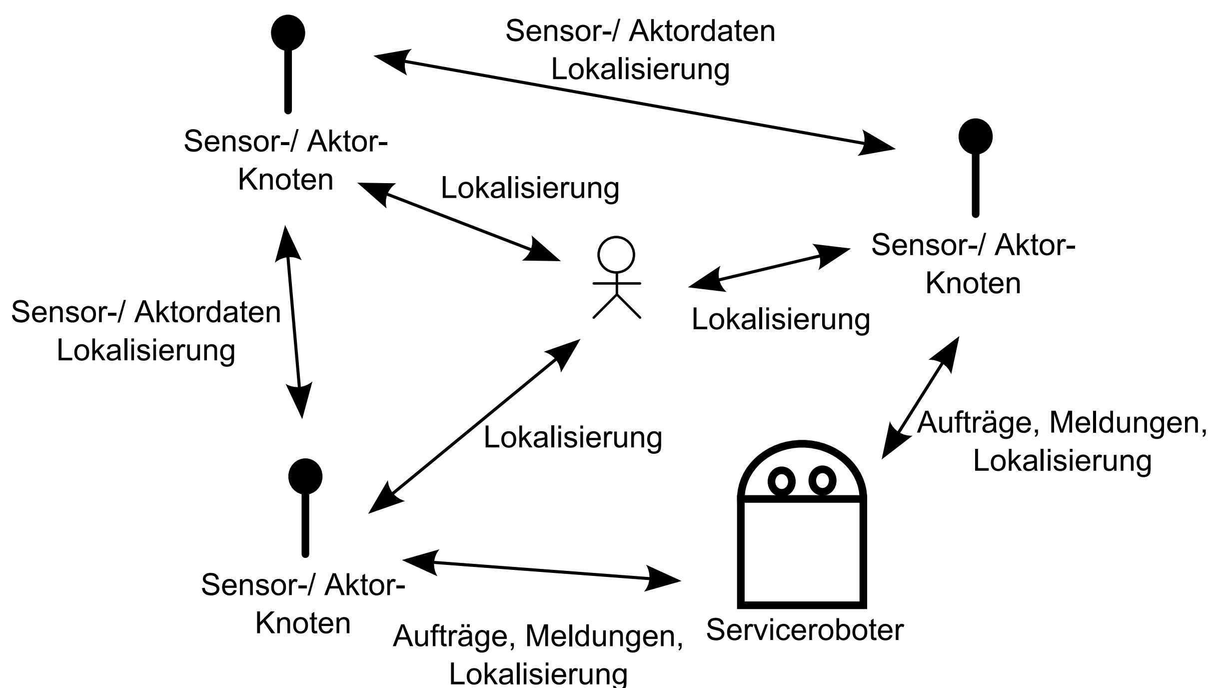


Abbildung 3.2: Beispiel eines Aufbaus eines Sensornetzwerks für Anwendungen im betreuten Wohnen: Intelligente Umgebung mit Lokalisierungsfunktion [Röh10]

Die Sensoren nehmen das Verhalten des Bewohners auf und analysieren es [Röh10]. Dabei kann ein Abgleich mit vorhergehenden Mustern geschehen, um eine Notfallsituation zu erkennen. Aus dem allgemeinen Verhaltensmuster können zudem

personalisierte Hilfestellungen für den Menschen abgeleitet werden, die dann durch Aktoren realisiert werden [Röh10]. So kann beispielsweise analysiert werden, wenn der Bewohner, der gegebenenfalls im Rollstuhl sitzt, seine Straßenschuhe anzieht und sich in Richtung Tür bewegt, die Tür automatisch für ihn geöffnet wird. Im Winter kann die Anpassung des Verhaltensmusters durch Hinzufügen des Anziehens einer Jacke erfolgen. Diese Ansätze lassen sich auch für allgemeine intelligente Gebäudetechnik weiter verfolgen, wo in einem Haushalt die Heizungs- und Beleuchtungstechnik beispielsweise an das Verhalten der Bewohner angepasst wird [Röh10].

Das beschriebene System kann im medizinischen Bereich durch Vitalwertsensoren erweitert werden, die einen Notfall möglichst schnell erkennen und einen Alarm auslösen können [Röh10]. Diese Sensoren können dabei entsprechend dem Patienten ausgewählt werden [Röh10]. Das Erkennen von solchen Notfallsituationen erfordert durch den Vergleich des Ist-Zustands mit den Referenzwerten.

Daten und Muster sind in diesem Anwendungsbereich vielfältig. Allein das Erkennen von Gefahrensituation, wie eines vergessenen Bügeleisens, ist komplex, da viele verschiedene Faktoren berücksichtigt werden müssen. Dies erfordert eine Vielzahl verschiedener Referenzdaten, die gespeichert und wieder gelesen werden müssen. Die Anpassung des Verhaltensmodells an ein verändertes Verhaltensmuster erfordert eine dynamische Veränderung vieler Referenzdaten. Daher bietet es sich an, diese Daten in einer Datenbank zu speichern und einen mehrdimensionalen Zugriff auf die Daten zu ermöglichen.

Dieser Abschnitt hat gezeigt, dass intelligente, eingebettete Systeme ein hohes Datenaufkommen besitzen können. Viele dieser Daten sind mehrdimensional und müssen im System gespeichert und verwaltet werden. Für die Mustererkennung konnte zudem gezeigt werden, dass spezielle mehrdimensionale Anfragearten nützlich sein können.

Bei den Referenzdaten für die Mustererkennung sind Parallelen zu Daten in Multimedia-Datenbanken und Geodatenbanken zu finden. So sind die Attribute von Referenzdaten zur Erkennung von Bildern oder von Sprache oft ähnlich zu Attributen von Bildern und Audiodaten in Multimedia-Datenbanken. Die Struktur der Daten ist denen in Geodatenbanken ähnlich. Dort besteht ein Objekt aus geometrischen Attributen und Sachattributen. Referenzdaten bestehen aus der Referenz und den Nutzdaten. Nutzdaten sind zum Beispiel der Name und weitere Eigenschaften des erkannten Musters. Aufgrund dieser Gemeinsamkeiten könnten Indexstrukturen, die in Multimedia-Datenbanken oder in Geodatenbanken genutzt werden, auch im Bereich der Mustererkennung in eingebetteten Systemen eine Unterstützung sein.

Dieser Abschnitt konnte außerdem zeigen, wie vielfältig die Anwendungsmöglichkeiten intelligenter, eingebetteter Systeme mit Mustererkennung sind. Nicht eingegangen wurde hier auf Anwendungsmöglichkeiten intelligenter, eingebetteter Systeme im Automobilbereich wie durch Fahrassistenzsysteme. Die Betrachtung von Anwendungen eingebetteter Systeme im Automobilbereich erfolgt im nächsten Abschnitt.

3.3.2 Automobilbereich

Eingebettete Systeme sind mittlerweile ein wichtiger Bestandteil moderner Autos [BS07]. Dies ist bereits an der Zahl der Steuergeräte pro Auto deutlich erkennbar. So sind in modernen Autos heute etwa 40 Steuergeräte und in Fahrzeuge der Oberklasse sogar deutlich mehr enthalten [BS07].

Ein Steuergerät ist ein Bestandteil eines eingebetteten Systems und erfüllt in etwa die Funktionen der Kontrolleinheit, wie sie im Aufbau eingebetteter Systeme im Abschnitt 2.1.1 aufgeführt ist. Die Besonderheit im Auto ist, dass teilweise mehrere Steuereinheiten auf gemeinsame Sensoren und Aktoren zugreifen [SZ06]. Bei Sensoren ist dies in der Regel unproblematisch [SZ06]. Bei einem gemeinsamen Zugriff auf Aktoren stellt das jedoch eine Herausforderung an die Entwickler dar [SZ06]. Dies wird in der Regel durch konkrete Schnittstellen versucht zu bewältigen [SZ06]. Im Weiteren wird vereinfachend immer die Steuereinheit mit ihren benötigten Sensoren und Aktoren als das eingebettete System betrachtet. Somit kann es möglich sein, dass Sensoren und Aktoren mehreren eingebetteten Systemen zugeordnet werden.

Die eingebetteten Systeme und ihre Software realisieren im steigenden Anteil Innovationen im Automobilbereich, die häufig stark zum Erfolg und zur Wettbewerbsfähigkeit des Produkts Auto beitragen [Gri05]. Studien haben ermittelt, dass 80% aller Innovationen von der Elektronik realisiert werden und davon 90% durch die Software [Gri05]. Ziel der Innovationen ist es, eine Erhöhung der Sicherheit, des Komforts und der Umweltverträglichkeit zu erreichen [Gri05].

Die Abbildung 3.3 zeigt die Einteilung der eingebetteten Systeme im Auto in Subsysteme. Zu Beginn des Einsatzes eingebetteter Systeme in Autos, ließen diese sich eindeutig Subsystemen zuordnen [SZ06]. Die Zuordnung eingebetteter Systeme im Auto wird heute durch übergeordnete Funktionen ergänzt, die eine Vernetzung der einzelnen Systeme und Subsysteme erfordern [SZ06]. Wird eine Funktion durch mehrere Steuergeräte innerhalb eines Subsystems realisiert, so wird dieses als *„integriertes System“* bezeichnet [SZ06]. Eine Realisierung von Funktionen über die Grenze von Subsystemen hinaus wird in der Regel mit Hilfe von Gateway-Steuergeräten realisiert [SZ06].

Alle eingebetteten Systeme im Auto werden als tief eingebettete Systeme betrachtet, da eine Beschränkung der Ressourcen im Auto generell notwendig ist. Die Gründe dafür sind im Gewicht, im Kraftstoffverbrauch sowie im Preis zu finden. Der Energieverbrauch ist zudem durch das Bordnetz begrenzt, das in Wechselwirkung mit dem Kraftstoffverbrauch steht [BS07, Rei09]. Daher muss der Energieverbrauch in allen Systemen möglichst gering gehalten werden [Rei09].

Im Folgenden wird an den Beispielen Antriebssteuerung und Fahrassistenz exemplarisch erläutert, inwieweit mehrdimensionale Daten benötigt werden. Auf die konkrete Realisierung der Funktionen wird dabei nicht genauer eingegangen.

Motor- und Getriebesteuerung Die Motor- und Getriebesteuerung gehören zum Subsystem Antriebsstrang [SZ06]. Die Funktionen beider Steuerungen hängen stark von der Fahrzeugkonfiguration ab [SZ06]. Die Aufgaben der Motorsteuerung richten sich stark nach der verwendeten Antriebseinheit [SZ06]. Das heißt, ob ein Ottomotor, ein Benzinmotor, ein Elektromotor oder ein hybrides System verwendet wird. Die Getriebesteuerung hängt stark von der verwendeten Getriebeart und der Art der Schaltung ab [SZ06]. So muss die Getriebesteuerung in einem Fahrzeug mit

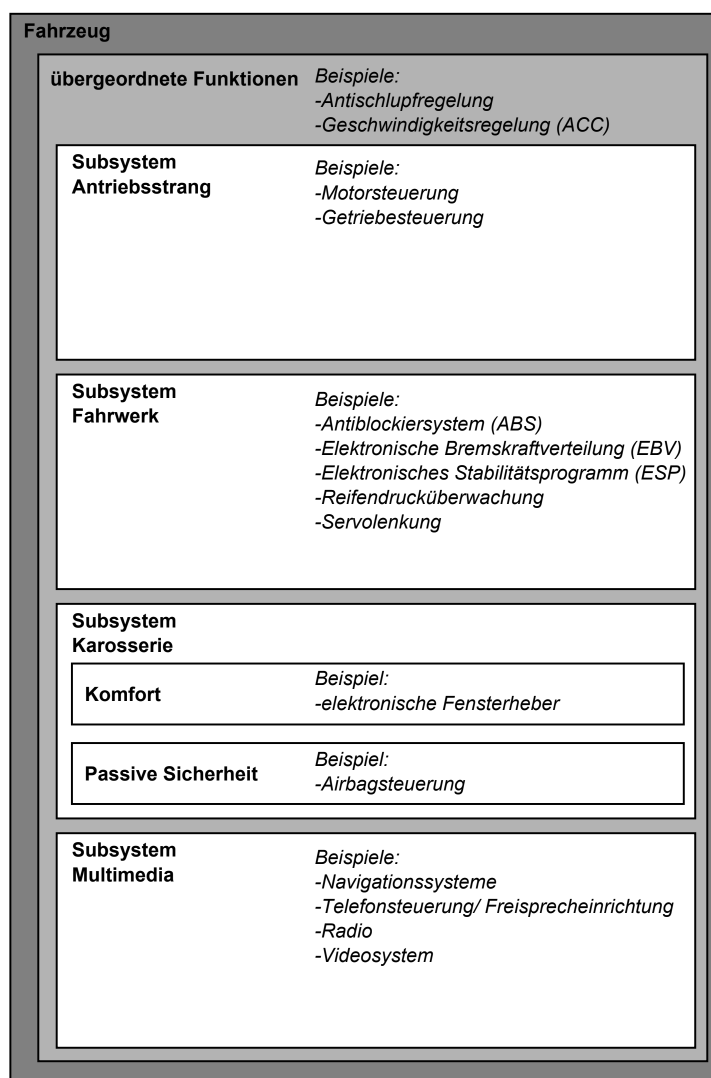


Abbildung 3.3: Einteilung der eingebetteten Systeme im Auto in Subsysteme mit Beispielen nach [SZ06]

Automatikschaltung in der Regel mehr Funktionen erfüllen als in einem Fahrzeug mit Handschaltung [SZ06].

Aufgrund der hohen Zahl an zu erfüllenden Funktionen benötigen beide Steuerungen viele Parameter, die in Form von Kennfeldern vorliegen können [SZ06]. Ein Beispiel dafür ist das Zündwinkelkennfeld, das in Abbildung 3.4 dargestellt ist. Dieses Kennfeld ist Teil der Motorsteuerung und für jeden Motor spezifisch [SZ06]. Es muss daher für jeden Motor neu gemessen und gespeichert werden [SZ06]. Das Zündwinkelkennfeld bildet den Betriebszustand, bestehend aus Last und Drehzahl, auf den bestmöglichen Zündwinkel ab [SZ06]. Dieser Wert beeinflusst den Kraftstoffverbrauch und das Abgasverhalten des Motors [SZ06].

Parameter sind allgemein ein wichtiger Aspekt, der zur Erfüllung der Funktion eines eingebetteten Systems beiträgt [SZ06]. Diese Parameter müssen effizient gespeichert werden und einen schnellen Zugriff erlauben [SZ06]. Parameter sind Kennwerte, Kennlinien oder Kennfelder [SZ06]. Kennlinien und Kennfelder sind dabei mehr-

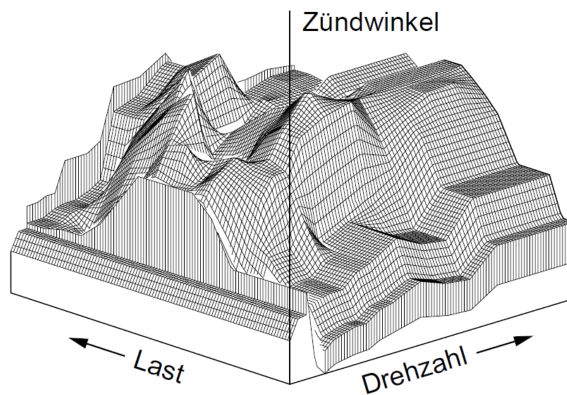


Abbildung 3.4: Zündwinkelkennfeld einer Motorsteuerung [DG98]

dimensional [SZ06]. So werden Parameterwerte für verschiedene Betriebszustände gespeichert und entsprechend abgerufen [SZ06]. Diese Parameter werden in der Regel durch Messungen bestimmt und häufig an das Endprodukt angepasst [SZ06]. Zur optimierten Speicherung werden verschiedene Maßnahmen eingesetzt [SZ06]. Unter anderem wird häufig ein tabellarisches Ablageschema gewählt, wobei eine Dimension monoton steigend gespeichert wird [SZ06]. Eine andere Möglichkeit ist die Speicherung in einer Datenbank mit einem mehrdimensionalen Zugriff. Somit könnten auch nachträglich einfach weitere Parameterpunkte eingetragen werden. Damit wäre eine nachträgliche Verfeinerung möglich. Dies scheint besonders bei mehrdimensionalen Kennfeldern eine gute Möglichkeit zu sein, da bei einer tabellarischen Abspeicherung immer alle Punkte entsprechend des Rasters gespeichert werden müssen. Abbildung 3.5 zeigt eine tabellarische Abspeicherung eines zweidimensionalen Kennfeldes. Eine Speicherung in einer Datenbank ermöglicht eine flexiblere Auswahl der Punkte. In wichtigen Bereichen könnten so einfach weitere Punkte genutzt werden.

Eingabe : x-Wert und y-Wert
Ausgabe: z-Wert

	x_1	x_2	x_3	x_4	x_5
y_1	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
y_2	z_{21}	z_{22}	z_{23}	z_{24}	z_{25}
y_3	z_{31}	z_{32}	z_{33}	z_{34}	z_{35}

Abbildung 3.5: Tabellarisches Ablageschema eines zweidimensionalen Kennfeldes [SZ06]

Das Beispiel der Kennfelder zeigt, dass auch im Bereich der Steuerungen des Antriebsstrangs mehrdimensionale Daten benötigt werden.

Fahrassistenzsysteme Viele Innovationen werden im Bereich der Fahrassistenzsysteme entwickelt, da Sicherheit ein entscheidender Kauffaktor für ein Auto geworden ist [BS07]. Die Fahrassistenzsysteme gehören zum Bereich der aktiven Sicherheit [SZ06]. Sie sollen den Fahrer in so weit unterstützen, dass Unfälle vermieden oder abgemildert werden [SZ06]. Passive Sicherheitsmechanismen streben dage-

gen an, Unfallfolgen zu minimieren [SZ06]. Eine Kombination beider Bereiche ist wünschenswert, da Fahrassistenzsysteme Unfälle nicht gänzlich verhindern können. Dies Umstand behält auch in Zukunft seine Gültigkeit [Rei09, BS07].

Fahrassistenzsysteme selbst werden teilweise dem Subsystem Fahrwerk zugeordnet [SZ06]. Neue Fahrassistenzsysteme arbeiten meist systemübergreifend [SZ06]. Beispiele für Fahrassistenzsysteme des Fahrwerks sind das Antiblockiersystem (kurz ABS), die elektronische Bremskraftverteilung (kurz EBV) und das elektronische Stabilitätsprogramm (kurz ESP) [SZ06]. Ein Beispiel für systemübergreifende Fahrassistenzsysteme ist die verkehrsflussangepasste Geschwindigkeitsregelung (englisch: Adaptive Cruis Control kurz ACC), die über den Abstand, die Relativgeschwindigkeit und die Winkellage des vorausfahrenden Fahrzeuges für einen konstanten Sicherheitsabstand sorgt [SZ06].

Es gibt Ansätze, in denen bildliche Aufnahmen zur Analyse der Fahrzeugumgebung genutzt werden [Rei09]. Dazu kann eine Mustererkennung, wie unter Abschnitt 3.3.1 beschrieben, genutzt werden. Mögliche Anwendungen sind Spurhaltesysteme, Systeme zum Erkennen von kritischen Hindernissen im toten Winkel und Systeme zur Erkennung von Verkehrsschildern [Rei09].

Allen Fahrassistenzsystemen ist gemein, dass die aktuelle Situation analysiert, eine mögliche Gefahrensituation erkannt und eine entsprechende Reaktion ausgeführt werden muss. Alle diese Aufgaben enthalten mehrdimensionale Daten, die jedoch nicht immer eine permanente Speicherung und einen späteren effizienten Zugriff benötigen. Ein Beispiel dafür sind die aktuellen Situationsdaten, auf die nur zum aktuellen Zeitpunkt ein effizienter Zugriff erforderlich ist. Die Speicherung erfolgt in der Regel nur zu Protokollzwecken [SZ06, NTN+02]. Eine Indexstruktur ist daher nicht notwendig.

Jedoch könnte der Zugriff über eine Indexstruktur für die Ausgabe einer Warnung an den Fahrer als eine auszuführende Reaktion des Systems stattfinden [NTN+02]. Die Warnung kann in Form von Text- oder Bildanzeigen, Sprachausgaben, Alarmleuchten oder sonstigen Tonsignalen erfolgen [NTN+02]. Bei Text-, Bild- oder Sprachausgaben ist die Auswahl der richtigen Sprache wichtig [NTN+02]. Bildausgaben können dabei im Tag- oder Nachtmodus angezeigt werden. Damit kann eine Warnmeldung über die Warnung und gegebenenfalls über die entsprechende Sprache und den Modus ausgewählt werden. Somit wird es möglich, eine Warnmeldung als ein mehrdimensionales Datum darzustellen, auf das anhand seiner Attribute (Warnung, Sprache, Modus) zugegriffen werden kann.

Fahrassistenzsysteme benötigen somit viele verschiedene mehrdimensionale Daten. Die Art der Daten und des benötigten Zugriffs hängt dabei von der konkreten Aufgabe und Funktionsweise des Systems ab.

Dieser Abschnitt konnte zeigen, dass mehrdimensionale Daten in eingebetteten Systemen verwaltet werden müssen und ein Zugriff über eine mehrdimensionale Indexstruktur durchaus sinnvoll sein kann. Der folgende Abschnitt führt eine kurze Analyse der allgemeinen Anforderungen an eine Verwaltung von mehrdimensionalen Daten, wie sie in den hier aufgeführten Beispielen genutzt werden, durch. Dabei wird nicht auf spezielle Anforderungen der konkreten Beispielanwendungen eingegangen. Es wird eher ein Überblick über die Anforderungen geschaffen.

3.4 Anforderungsanalyse

Anhand der bereits beschriebenen Anwendungsbeispiele werden allgemeine Anforderungen an die Verwaltung der Daten abgeleitet. Dieser Abschnitt beschränkt sich auf die allgemeinen Anforderungen, die bezüglich der Verwaltung mehrdimensionaler Daten eine besondere Relevanz aufweisen und somit für die Auswahl geeigneter Indexstrukturen von Bedeutung sind.

Allgemein lässt sich für alle aufgeführten Anwendungen feststellen, dass eine hohe Effizienz erforderlich ist. Diese Effizienz betrifft sowohl die Ausnutzung des vorhandenen Speichers als auch die Geschwindigkeit des Zugriffs. Diese beiden Anforderungen stehen häufig in Konkurrenz zueinander. Rechenzeit lässt sich häufig durch die Speicherung von Zwischenergebnissen sparen. Dies führt jedoch zu einem höheren Speicherbedarf. Dieses Phänomen ist auch bei der Verwendung von Indexstrukturen sichtbar. Indexstrukturen können die Zugriffszeit verkürzen, führen aber zu einem höheren Speicherbedarf [SHS05]. Da im Bereich der eingebetteten Systeme und insbesondere im Bereich der tief eingebetteten Systeme Speicherressourcen begrenzt sind, ist ein Vergleich der Speicherkosten und des Zeitgewinns notwendig. Eine Betrachtung bezüglich der implementierten Strukturen erfolgt in Kapitel 6.

Aufgrund der aufgeführten Anwendungen wird im Weiteren von Punktdaten ausgegangen. Diese beschreiben immer genau einen Punkt im Merkmalsraum und bestitzen keine räumliche Ausdehnung. Ein Attribut wird dabei als ein Merkmal aufgefasst. Ein Merkmal ist dabei auch ein Parameter, wie im Falle eines Kennfeldes. Aus den verwendeten mehrdimensionalen Daten lässt sich eine innere Struktur der Tupel wie in Abbildung 3.6 bestimmen. Der mehrdimensionale Zugriff erfolgt dabei über die Suchattribute, die je ein Merkmal darstellen. Die Nutzdaten sind dabei in der Regel das eigentliche Ziel der Anfrage. Es gibt jedoch auch die Möglichkeit, nur das Vorhandensein der Kombination der Suchattribute zu prüfen oder den nächsten Nachbarn zu einem gegebenen Punkt zu finden. In diesen Fällen können die Nutzdaten weniger relevant sein, beziehungsweise entfallen. Die Suchattribute sind obligatorische Werte, die nicht entfallen dürfen.

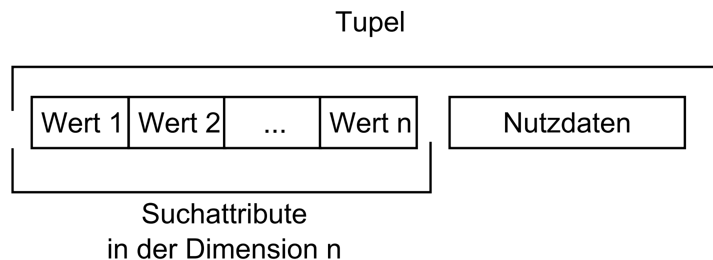


Abbildung 3.6: Innere Struktur von Tupel

Die Anzahl der Dimensionen der Daten ist in einigen Anwendungen hoch, wie oft im Falle der Bilderkennung. Dies führt dazu, dass Indexstrukturen für hochdimensionale Daten ebenfalls untersucht werden müssen.

Als benötigte Anfragearten stellen sich die Nächste-Nachbarsuche, die mehrdimensionale Exact-Match-Suche und die mehrdimensionale Bereichssuche heraus. Die Nächste-Nachbarsuche ist besonders für Anwendungen mit Mustererkennung von Bedeutung, während Exact-Match-Suchen zum Beispiel für Kennfelder wichtig sind. Mehrdimensionale Bereichsanfragen für Sensornetzwerke wurden bereits realisiert, um interessante

Ereignisse zu lokalisieren [LKGH03]. In eingebetteten Systemen können diese Bereichsanfragen ebenfalls zum Zweck der Erkennung interessanter Daten genutzt werden. Dies ist zum Beispiel innerhalb von Protokolldaten denkbar. Die Anfragearten werden im Abschnitt 4.2 des Kapitels 4 genauer erläutert.

Im folgenden Kapitel werden einige mehrdimensionale Indexstrukturen vorgestellt, aus denen dann anhand der aufgeführten Anforderung zwei zur Implementierung und Analyse ausgewählt werden.

Kapitel 4

Mehrdimensionale Indexstrukturen

Nachdem im vorherigen Kapitel die möglichen Anwendungen und Anforderungen erläutert wurden, beschäftigt sich dieses Kapitel mit den mehrdimensionalen Indexstrukturen selbst. Dabei wird zu Beginn in Abschnitt 4.1 ein Überblick über mehrdimensionale Indexstrukturen geschaffen. Der Abschnitt 4.2 beschäftigt sich dann mit den bereits im Kapitel 3 genannten Arten von mehrdimensionalen Anfragen. Auf die Anfragearten wird bei der Betrachtung der Indexstrukturen Bezug genommen, die im Abschnitt 4.3 erfolgt. Es werden verschiedene mehrdimensionale Indexstrukturen mit ihren Vor- und Nachteilen vorgestellt. Auf Grundlage der Betrachtung der Indexstrukturen werden im Abschnitt 4.4 zwei Indexstrukturen ausgewählt, die implementiert und analysiert werden.

4.1 Übersicht und Einteilung

Mehrdimensionale Indexstrukturen dienen zur Unterstützung des mehrdimensionalen Zugriffs auf Daten in einer Datenbank [SHS05]. Grundsätzlich arbeiten diese Indexstrukturen entweder auf *Punkten oder ausgedehnten Objekten* [Pag96]. Ausgedehnte Objekte sind beispielweise Linien, Kreisflächen und Rechteckflächen. Die Ausdehnung führt zu speziellen möglichen Anfragen wie Schnittanfragen, aber auch zu besonderen Herausforderungen an die Indexstruktur [Pag96]. Aufgrund der Anforderungsanalyse aus Abschnitt 3.4 beschränkt sich diese Arbeit auf mehrdimensionale Indexstrukturen für Punktdaten. Die Punktdaten können als Vektor dargestellt werden.

Mehrdimensionale Indexstrukturen für Punktdaten lassen sich unterscheiden in [SHS05, Bal04]:

- hierarchische Verfahren,
- mehrdimensionale Hashverfahren,
- Grid-File Verfahren und
- Approximationsverfahren.

Bedeutende Vertreter der *hierarchischen, mehrdimensionalen Indexstrukturen* sind die Baumverfahren [Sch05a]. Hierarchische Verfahren arbeiten auf Clustern [Sch05a].

Cluster sind Regionen, die eine bestimmte Menge von Punktdaten beschreiben. Anfragen erfolgen zuerst über den Clustern. Punktdaten in Clustern, die aufgrund der Beschreibung ausgeschlossen werden können, müssen nicht extra geprüft werden. Eine Hierarchie entsteht dadurch, dass die Cluster sich gegenseitig enthalten können [Sch05a]. Bei Baumverfahren kann ein Cluster nur in einem übergeordneten Cluster enthalten sein. Das übergeordnete Cluster kann jedoch mehrere Cluster enthalten [Sch05a]. Als Beispiele für Baumverfahren werden der R-Baum (4.3.1) und der LSD-Baum (4.3.2) vorgestellt. Dabei ist zu bemerken, dass der R-Baum auch ausgedehnte Objekte unterstützt [Sch05a].

Mehrdimensionale Hashverfahren nutzen eine Hashfunktion zum Zugriff auf Daten [SHS05]. Die Hashfunktion bildet von den Schlüsselattributen des Datensatzes auf ein Hashbucket in Form eines Hashwertes ab [SHS05]. Dieser Vorgang wird Schlüsseltransformation genannt [SHS05]. Ein Hashbucket liegt dabei direkt im Speicher liegen und speichert die entsprechenden Datensätze oder liegt in einer Tabelle und verweist von dort aus auf die Adresse im Speicher. Bei mehrdimensionalen Hashverfahren besteht der Hashwert aus einem Tupel von Zahlen [SHS05]. Mehrdimensionale Hashverfahren unterstützen Exact-Match-Suchen und teilweise Partial-Match-Suchen [SHS05]. Nächste-Nachbarsuchen und mehrdimensionale Bereichssuchen werden in der Regel nicht unterstützt, daher werden Hashverfahren in dieser Arbeit nicht weiter betrachtet.

Die *Grid-File Verfahren* nutzen eine Kombination aus den Grundprinzipien der Hashverfahren und der hierarchischen Verfahren [SHS05]. Näheres zur Arbeitsweise von Grid-Files folgt im Abschnitt 4.3.3.

Approximationsverfahren wurden für hochdimensionale Daten entwickelt, da bei hierarchischen Indexstrukturen mit steigender Anzahl der Dimensionen eine Verschlechterung des Zugriffes festgestellt wurde [Bal04]. Dieses Phänomen wird in der Literatur häufig als „Fluch der hohen Dimensionen“ bezeichnet [Bal04]. Approximationstechniken nutzen eine kompakte Darstellung der Daten, die an die konkreten Daten angenähert ist [Bal04]. Die Liste der Approximationen der Daten wird sequentiell durchlaufen. Aufgrund der Approximation können während des Durchlaufs Datensätze ausgeschlossen werden, die danach nicht mehr konkret untersucht werden müssen [Bal04]. Als Beispielverfahren wird in Abschnitt 4.3.4 das VA-File vorgestellt.

4.2 Anfragearten

Dieser Abschnitt betrachtet die in Kapitel 3 bereits genannten mehrdimensionalen Anfragearten. Im Folgenden beschränken sich die Betrachtungen auf rein selektive Anfragen. Verbundanfragen werden von allen weiteren Betrachtungen ausgeschlossen, da davon ausgegangen wird, dass die Datenbank über nur eine Tabelle mit entsprechenden Daten verfügt. Dies ist durch die Struktur der Anwendungen, durch die beschränkten Ressourcen und durch den Aufbau von RobbyDBMS zu begründen.

Zum besseren Verständnis der folgenden Erläuterungen zu den Anfragearten ist es notwendig, den Begriff der *Distanz* zu klären. Die Distanz beschreibt in dieser Arbeit den Abstand zweier Punktdaten und wird durch die *Distanzfunktion* berechnet [SHS05]. Mit der Distanzfunktion wird die Metrik für den Vergleich der mehrdimensionalen Daten festgelegt [SHS05]. Somit wird die Ähnlichkeit der Datenpunkte ermittelt. Eine Distanzfunktion δ muss die folgenden Eigenschaften erfüllen. PO ist die Menge der Punktdaten

[SHS05]:

- Die Distanz eines Punktes zu sich selbst ist null. $\forall p \ni PO : \delta(p, p) = 0$
- Die Distanz zweier Punkte ist kommutativ. $\forall p_1, p_2 \ni PO : \delta(p_1, p_2) = \delta(p_1, p_2)$
- Es gilt die Dreiecksungleichung. $\forall p_1, p_2, p_2 \ni PO : \delta(p_1, p_2) + \delta(p_2, p_3) \geq \delta(p_1, p_3)$

Beispiele für bedeutende Distanzmetriken sind die euklidische Distanz und die Manhattan-Distanz [SHS05]. Es sind jedoch auch komplexe, auf die Anwendung bezogene Metriken möglich [SHS05].

Bereichssuche Eine Bereichssuche, im Englischen *range query*, sucht Datensätze deren Attribute innerhalb eines bestimmten Suchbereichs liegen [SHS05]. Es gibt verschiedene Varianten der Bereichssuche [Sch05a]. Häufig verwendet werden Bereichssuchen im Umkreis um einen gegebenen Punkt und Bereichssuchen in einem gegebenen Wertintervall [Sch05a, SHS05].

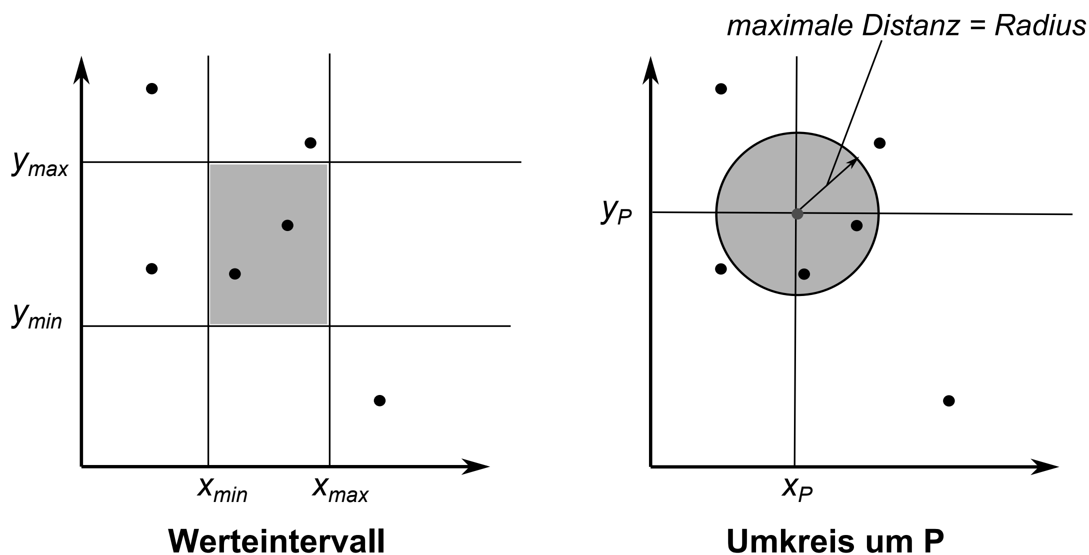


Abbildung 4.1: Graphische Darstellung der Bereichssuche im zweidimensionalen Fall

Bei der Suche im Umkreis eines Punktes müssen der Punkt selbst und eine maximale Distanz gegeben werden. Wird als Metrik die euklidische Distanz genutzt, so wird der Bereich als Hyperkugel entsprechend der Anzahl an Dimensionen betrachtet [Sch05a]. Bei der Bereichssuche in einem Wertintervall muss kein Punkt angegeben werden. Dafür müssen für jede Dimension der minimale und der maximale Wert der Attribute, als untere und obere Grenze der Attributwerte, angegeben werden. Es entsteht so ein Hyperquader als Suchbereich [SHS05]. Abbildung 4.1 veranschaulicht beide Arten der Bereichssuche für einen zweidimensionalen Datenraum.

Im Folgenden wird unter Bereichssuche immer die Variante mit einem Wertintervall verstanden, da bereits die Nächste-Nachbarsuche mit der Distanz arbeitet. Die Veränderung der Implementierung zu einer Bereichssuche mit Umkreis ist jedoch ohne Weiteres möglich und sollte ähnliche Ergebnisse bezüglich der Analyse liefern.

Exact-Match-Suche Bei der Exact-Match-Suche wird nach einem Punkt gesucht, dessen Suchattribute exakt mit denen des gegebenen Punktes übereinstimmen [SHS05]. Die Exact-Match-Suche ist ein Spezialfall der Bereichssuche und der Nächsten-Nachbarsuche [SHS05]. Bei der Variante der Bereichssuche mit einem Wertintervall sind jeweils die obere und die untere Grenze gleich. Bei der Variante mit dem Umkreis und der Nächsten-Nachbarsuche ist die Distanz null. In dieser Arbeit wird die Exact-Match-Suche jedoch aus Performance- und Relevanzgründen als eigenständige Anfrage implementiert. So können unnötige Vergleiche und die Berechnung der Distanz entfallen.

Partial-Match-Suche Die Partial-Match-Suche ist ein Spezialfall der Bereichssuche mit Wertintervallen [Sch05a]. Bei der Partial-Match-Suche werden nicht für alle Suchattribute Werte angegeben. Diese können in der Bereichssuche mit Wertintervallen als Intervalle über den gesamten Wertebereich der Dimension aufgefasst werden [Sch05a]. Aus diesem Grund und aufgrund mangelnder Anwendungen wird die Partial-Match-Suche in dieser Arbeit nicht näher betrachtet.

Nächste-Nachbarsuche Bei der Nächsten-Nachbarsuche wird zu einem gegebenen Punkt der nächstgelegene Nachbar gesucht [SHS05]. Das heißt, es wird der Punkt gesucht, der zu einem gegebenen Punkt die geringste Distanz hat. Werden bei dieser Suche mehrere Punkte mit der gleichen minimalen Distanz zum gegebenen Punkt gefunden, so werden entweder alle Punkte oder ein Punkt, der nichtdeterministisch bestimmt wird, ausgegeben [SHS05]. Abbildung 4.2 veranschaulicht die Nächste-Nachbarsuche graphisch.

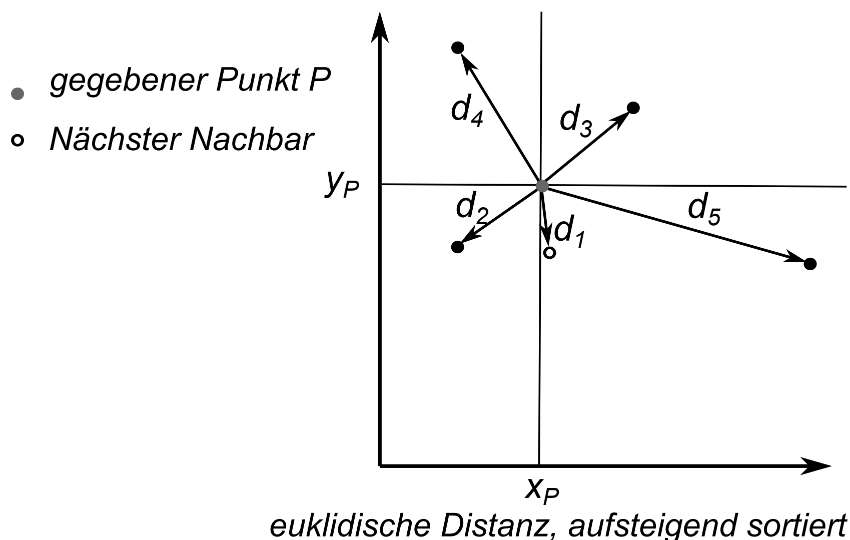


Abbildung 4.2: Graphische Darstellung der Nächsten-Nachbarsuche im zweidimensionalen Fall mit der euklidischen Distanz [Sch05a]

Eine Verallgemeinerung der Nächsten-Nachbarsuche ist die k-Nächste-Nachbarsuche, bei der nicht nur der Punkt mit der minimalen Distanz, sondern auch k weitere Punkte, mit jeweils der nächstgrößeren Distanz, ausgegeben werden [Sch05a]. Die k-Nächste-Nachbarsuche wird in dieser Arbeit nicht weiter untersucht, da keine Anwendungen in tief eingebetteten Systemen gefunden werden konnten.

4.3 Betrachtung verschiedener mehrdimensionaler Indexstrukturen

Dieser Abschnitt stellt exemplarisch einige mehrdimensionale Indexstrukturen vor. Ziel ist es, die Kerngedanken sowie die Arbeitsweise der einzelnen Strukturen herauszuarbeiten. Auf Grundlage dieser Beschreibungen erfolgt im Abschnitt 4.4 die Auswahl zweier Indexstrukturen, deren Implementierung im folgenden Kapitel beschrieben wird.

4.3.1 Der R-Baum

Der *R-Baum* gilt als Prototyp geometrischer Suchbäume, die eine nachbarschaftserhaltende Speicherung ermöglichen [Sch05a]. Seitdem wurden verschiedene Erweiterungen und Varianten des R-Baums entwickelt, darunter der R^+ sowie der X-Baum [Sch05a]. Der R-Baum selbst ist eine Erweiterung des B-Baums, bei der mehrere Dimensionen unterstützt werden [Sch05a]. Die Nutzung des R-Baums beschränkt sich auf niedrigdimensionale Daten [Sch05a]. Ein häufiges Einsatzgebiet sind Geodatenbanken [Sch05a]. Der R-Baum ist eine Indexstruktur für beliebige, ausgedehnte Objekten, wie beliebigen Regionen, aber auch Punkten [Sch05a]. Die Vorstellung hier erfolgt aufgrund seiner Bedeutung in Geodatenbanken.

Analog zum B-Baum ist der R-Baum ein balancierter Mehrwegbaum, dessen Objekte wie beim B^+-Baum in den Blättern liegen [Sch05a]. Abbildung 4.3 zeigt einen R-Baum mit Punktdaten. Jeder Knoten enthält mehrere Einträge, deren Anzahl durch einen Minimalwert und Maximalwert begrenzt wird [Sch05a]. Der Maximalwert richtet sich nach der Seitengröße des Hintergrundspeichers, damit ein Knotenzugriff genau einem Zugriff auf den Hintergrundspeicher entspricht [Sch05a].

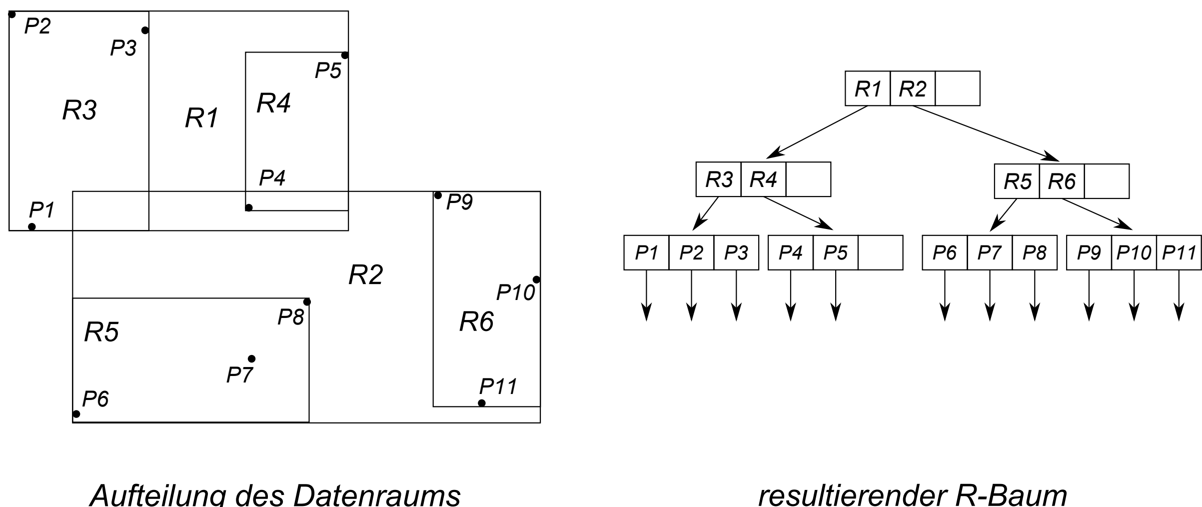


Abbildung 4.3: R-Baum mit Punktdaten im zweidimensionalen Raum [Sch05a]

Die Daten im R-Baum werden lokal zu Hyperquadern gruppiert [Sch05a]. Als Hyperquader werden *MBRs (Minimum Bounding Rectangle)* genutzt [Sch05a]. Abbildung 4.4 stellt einen MBR graphisch dar. Die Cluster-Geometrie wird durch ein Punktepaar *(s,t)* beschrieben. Der Punkt *s* beschreibt die dem Koordinatenursprung zugewandte

Ecke des Quaders und der Punkt t die dazu gegenüberliegende Ecke. Eine $(n-1)$-dimensionale Hyperfläche eines n-dimensionalen MBRs muss mindestens ein Objekt des MBRs berühren. So wird die Minimalität des Hyperquaders sichergestellt, die für die Effizienz der Zugriffsalgorithmen wichtig ist [Sch05a].

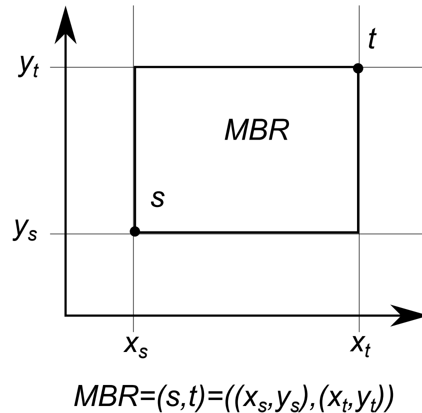


Abbildung 4.4: Graphische Darstellung eines MBRs im zweidimensionalen Raum mit dem beschreibenden Punktepaar (s,t) [Sch05a]

Der R-Baum besteht aus zwei Arten von Knoten, den inneren Knoten und den Blattknoten [Sch05a]. Die Einträge der inneren Knoten bestehen jeweils aus dem Punktepaar des MBRs und einem Verweis auf den enthaltenen Kindknoten. Ein Blattknoten besteht ebenfalls aus dem Punktepaar des MBRs und einem Verweis auf das entsprechende Tupel. Bei Punktdaten kann anstelle des Punktepaares des MBRs der Punkt gespeichert werden, wie in Abbildung 4.3 dargestellt [Sch05a].

Eine Überlappung der Cluster ist im R-Baum erlaubt [Sch05a]. Somit müssen bei einer Suche gegebenenfalls mehrere Teilbäume parallel untersucht werden. Bei einem hohen Grad der Überlappung können so weniger Teilbäume bei der Anfragebearbeitung ausgeschlossen werden und somit steigt die Zahl der Zugriffe auf den Hintergrundspeicher. Deswegen ist ein hoher Grad an Überlappung zu vermeiden [Sch05a]. Im niedrigdimensionalen Fall gelingt dies noch gut durch Verwendung spezieller Einfüge- und Splitalgorithmen, die der Literatur zu entnehmen sind [Sch05a]. Ab einer Dimensionsanzahl von etwa 10 wird der R-Baum jedoch aufgrund der zu erwartenden Überlappungen ineffizient [Sch05a]. Auch Erweiterungen, wie der R^+- und der X-Baum, können dieses Problem nicht beheben, sondern nur das Auftreten hinauszögern [Sch05a].

R-Bäume unterstützen unter anderem mehrdimensionale Exact-Match-Suchen, mehrdimensionale Bereichssuchen und Nächste-Nachbarsuchen. Beim Auftreten von Überlappungen wird zusätzlicher Arbeitsspeicher benötigt, um die parallele Bearbeitung der Teilbäume zu realisieren.

4.3.2 Der LSD-Baum

Der *LSD-Baum (Local Split Decision)* wurde 1989 von Henrich, Six und Widmayer eingeführt und ist eine k-d-Baum-basierte Indexstruktur für mehrdimensionale Daten [HSW89]. Eine Erweiterung des LSD-Baums ist der LSD^h-Baum, der für hochdimensionale Daten gedacht ist [Hen98].

Der LSD-Baum unterstützt mehrdimensionale Punktdaten. Die Unterstützung von beliebigen, ausgedehnten Objekten wird mit Hilfe von Transformationstechniken jedoch ebenfalls realisiert [HSW89]. Die Ausführungen dieser Arbeit beschränken sich aufgrund der Anforderungsanalyse auf die Unterstützung von Punktdaten.

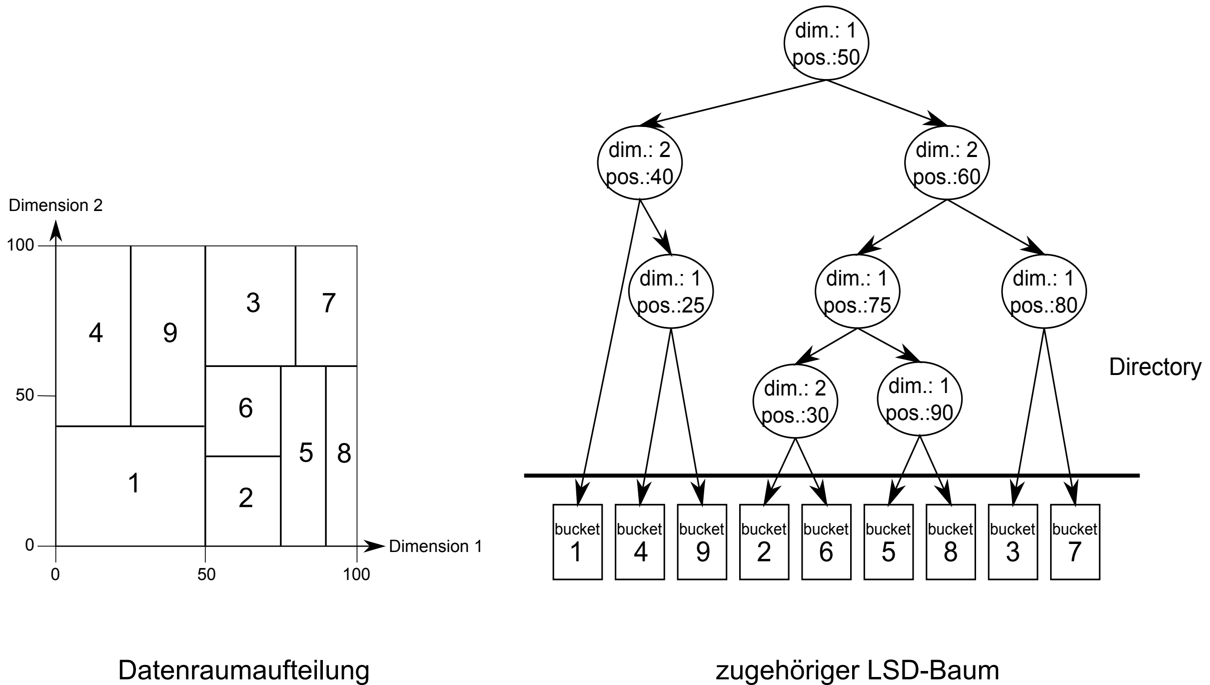


Abbildung 4.5: Mögliche Datenraumaufteilung und zugehöriger LSD-Baum [HSW89]

Beim LSD-Baum wird der Datenraum in vollständig disjunkte Teilräume entlang der Achsen untergliedert [HSW89]. Abbildung 4.5 zeigt eine mögliche Aufteilung des Datenraums und den dazugehörigen LSD-Baum. Die Entscheidung der Teilung eines Teilraums erfolgt nach dem lokalen Optimum. Daher stammt auch der Name des LSD-Baums [HSW89].

Ein LSD-Baum besteht aus inneren Knoten und Blattknoten. Die inneren Knoten dienen als Verzeichnisknoten und speichern die Verweise auf die Kindknoten, die Dimension der Teilung als *Splitdimension* sowie die Position der Teilung als *Splitposition*. Alle inneren Knoten bilden zusammen das *Verzeichnis (Directory)* [HSW89]. Die Punkte eines Teilraums werden in den Blättern, genannt Buckets, gespeichert. Die Größe eines Buckets ist festgelegt und richtet sich, ähnlich der Größe eines Knotens beim R-Baum, nach der Größe der Hintergrundspeicherseiten, sodass nur eine Hintergrundspeicherseite pro Bucket gelesen werden muss [HSW89].

Die Buckets befinden sich im Hintergrundspeicher, während das Verzeichnis im Hauptspeicher liegt [HSW89]. Wird das Verzeichnis zu groß für den Hauptspeicher, so werden Teilbäume in den Hintergrundspeicher ausgelagert [HSW89]. Diese Auslagerung erfolgt in Ebenen, sodass innerhalb einer Ebene immer ein Teilbaum auf eine Speicherseite passt [HSW89]. So entstehen ein internes Verzeichnis und ein externes Verzeichnis, das gegebenenfalls mehrere Ebenen hat. Die genaue Vorgehensweise der Auslagerung ist der Literatur zu entnehmen [HSW89].

Die Teilung eines Teilraums erfolgt lokal nach einer Strategie, die *Splitstrategie* ge-

nannt wird [HSW89]. Das Ziel der Splitstrategie ist es, die lokal beste Position und Dimension für die Teilung zu finden [HSW89]. Vorherige Teilungsentscheidungen haben keinen Einfluss auf die lokale Entscheidung. Die Splitstrategien werden in zwei grundlegende Gruppen unterschieden, die Gruppe der datenabhängigen Splitstrategien und die Gruppe der verteilungsabhängigen Splitstrategien [HSW89].

Datenabhängige Splitstrategien Die datenabhängigen Splitstrategien arbeiten direkt mit den im Bucket gespeicherten Daten und ermitteln mit deren Hilfe die Splitdimension und -position [HSW89].

Verteilungsabhängige Splitstrategien Ohne Berücksichtigung der gespeicherten Daten werden bei einer verteilungsabhängigen Splitstrategie die Splitdimension und -position bestimmt. Ein Beispiel ist die Teilung des Datenraums immer in der Mitte einer Achse [HSW89].

Die Splitstrategie beeinflusst nachhaltig den Aufbau eines Verzeichnisses und den Füllgrad der Buckets [HSW89]. Datenabhängige Splitstrategien sind dabei für unsortierte Einfügereihenfolgen und unbekannte Datenverteilungen geeignet, während verteilungsabhängige Strategien für sortierte Einfügereihenfolgen von Daten mit bekannter Verteilung gut sind [HSW89].

Der LSD-Baum unterstützt unter anderem Exact-Match-Suchen, Bereichssuchen und Nächte-Nachbarsuchen [HSW89, Hen94]. Zu beachten ist, dass Bereichssuchen und Nächste-Nachbarsuchen zusätzlichen Arbeitsspeicher benötigen, um gegebenenfalls beide Teilbäume untersuchen zu können.

Beim LSD-Baum sind, ähnlich dem R-Baum, Probleme bei der Unterstützung hochdimensionaler Daten zu erwarten [Bal04]. Diese Probleme sind durch den steigenden Aufwand der Anfragebearbeitung zu begründen [Bal04]. Auch der LSD^h-Baum kann diese Probleme nicht lösen, sondern nur deren Auftreten hinauszögern [Bal04].

4.3.3 Das Grid-File

Ein *Grid-File* ist eine Indexstruktur, die nach dem Prinzip der Nachbarschaftserhaltung arbeitet und eine Exact-Match-Suche mit zwei Plattenzugriffen realisiert. Der Datenraum wird zu diesem Zweck in n-dimensionale Quader zerlegt, ähnlich der Aufteilung des Datenraums beim LSD-Baum. Die Struktur wird dynamisch an die enthaltenen Daten beim Einfügen und Löschen angepasst [SHS05].

Ein Grid-File besteht aus mehreren Teilen, dem Grid, den Skalen, dem Grid-Directory, den Grid-Zellen und den Grid-Regionen [SHS05]. Abbildung 4.6 veranschaulicht diesen Aufbau. Das Grid repräsentiert den Datenraum, der in mehrere eindimensionale Felder geteilt ist. Diese eindimensionalen Felder sind die Skalen, die den Achsen des Datenraums entsprechen und somit je ein Attribut repräsentieren. Die Skalen sind in Intervalle geteilt. Die Skalen mit ihren entsprechenden Intervallen werden in der Regel im Arbeitsspeicher gehalten. Bei einer Exact-Match-Anfrage wird mit Hilfe der Skalen vom gesuchten Punkt auf die Intervalle abgebildet. Die Kombination der Intervalle ergibt dann einen Indexwert. Dieser Indexwert wird zum Zugriff auf das Grid-Directory genutzt. Das Grid-Directory besteht aus den Grid-Zellen. Diese Zellen teilen den Datenraum entsprechend der Intervalle in n-dimensionale Quader. Eine Grid-Region besteht

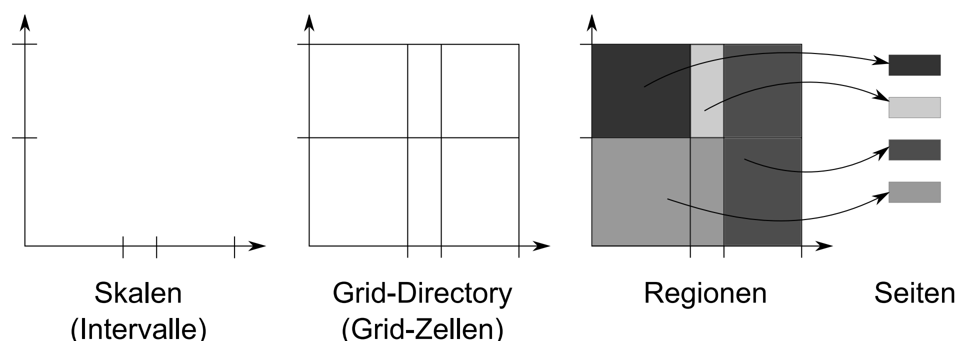


Abbildung 4.6: Aufbau eines Grid-Files [SHS05]

aus mehreren Grid-Zellen und verweist direkt auf eine Datensatzseite im Hintergrundspeicher. Das Grid-Directory mit den entsprechenden Regionen liegt in der Regel im Hintergrundspeicher. Aus dem berechneten Indexwert wird so die Adresse der Datensatzseite ermittelt. Der zweite Zugriff auf den Hintergrundspeicher ist das Lesen dieser Datensatzseite [SHS05].

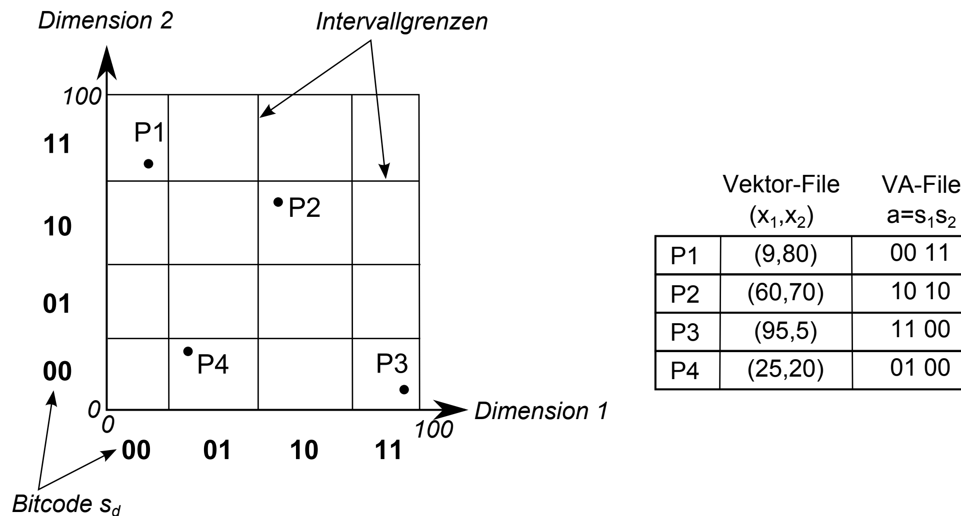
Exact-Match-Suchen können mit einem Grid-File effizient realisiert werden [SHS05]. Eine symmetrische Behandlung aller Dimensionen ermöglicht zudem eine effiziente Behandlung von Partial-Match-Anfragen [SHS05]. Bereichsanfragen und Nächste-Nachbaranfragen können jedoch nur mit größerem Aufwand realisiert werden, da mehrere Zugriffe auf das Grid-Directory notwendig werden, um alle relevanten Regionen zu untersuchen. Das Grid-File ist für hohe Dimensionsanzahlen aufgrund des Aufwands weniger geeignet [WB97].

4.3.4 Das VA-File

Das Grundprinzip der Approximationsverfahren ist es, die sequentielle Suche durch Ausschluss von Datensätzen zu beschleunigen [Sch05a]. Zu diesem Zweck werden kompakte Approximationen genutzt [Sch05a]. Das *VA-File* wurde 1997 von Weber und Blott zur Unterstützung von Nächste-Nachbarsuchen in hochdimensionalen Datenräumen vorgestellt und ist eines der populärsten Approximationsverfahren [WB97, Sch05a]. „VA“ steht dabei für „vector-approximation“ (auf Deutsch „Vektorapproximation“) [WB97]. Weiterentwicklungen des VA-Files sind unter anderem das AV-File, das VA^+-File und der A-Baum [Bal04].

Das VA-File nutzt für die Approximation der Datensätze die Aufteilung des Datenraums in Hyperquader [WB97]. Jede Dimension d wird dabei in 2^{b_d} Intervalle gegliedert, die dann binär durchnummeriert werden und damit durch b_d Bits identifiziert werden. Üblich sind 4 bis 6 Bits pro Dimension [WB97]. Abbildung 4.7 veranschaulicht eine mögliche Aufteilung eines zweidimensionalen Datenraums. Die Identifikation eines Hyperquaders erfolgt dann durch die Aneinanderreihung aller Bitcodes s_d. Diese Aneinanderreihung wird im Weiteren als Signatur s bezeichnet. Der Hyperquader, der durch seine Signatur eindeutig zu identifizieren ist, wird als Approximation a für alle in ihm enthaltenen Punkte genutzt [WB97].

Bei der Verwendung des VA-Files werden alle Datensätze sequentiell gespeichert. Analog zu dieser Reihenfolge werden alle Approximationen in einer Liste gespeichert. Das



	Vektor-File (x_1,x_2)	VA-File $a=s_1s_2$
P1	(9,80)	00 11
P2	(60,70)	10 10
P3	(95,5)	11 00
P4	(25,20)	01 00

Abbildung 4.7: Datenraumaufteilung und Approximation im VA-File für einen zweidimensionalen Beispieldatenraum [WB97]

heißt, zu jedem Datensatz wird die Signatur des umgebenden Hyperquaders gespeichert. Die entstehende Liste mit den Approximationen wird in der Regel als *VA-File* bezeichnet und die Liste der Datensätze als *Vektor-File* [WB97]. Abbildung 4.7 zeigt beide Listen für die Beispieldaten.

Die Intervallgrenzen eines Hyperquaders müssen von vornherein festgelegt und gespeichert werden [WB97]. Sie sind für den Wirkungsgrad des VA-Files ausschlaggebend [Bal04] und sollten gemeinsam mit der Zahl der Intervalle so gewählt werden, dass möglichst eine konstante Anzahl von Punkten in einem Hyperquader enthalten sind [WB97]. Zur Bestimmung der Grenzen können laut Weber und Blott etablierte Techniken des Grid-Files genutzt werden [WB97].

Das VA-File wurde ursprünglich nur zur Unterstützung von Nächste-Nachbarsuchen entwickelt, kann aber auch für Exact-Match-Suchen, Partial-Match-Suchen und Bereichssuchen eingesetzt werden.

4.4 Auswahl

In diesem Abschnitt werden zwei Indexstrukturen ausgewählt, die im weiteren Verlauf dieser Arbeit implementiert und analysiert werden. Da die Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme genutzt wird, gelten besondere Anforderungen, die bereits im Abschnitt 3.4 analysiert wurden. Demnach müssen entsprechende Indexstrukturen hauptsächlich Punktdaten effizient unterstützen. Die Effizienz bezieht sich sowohl auf die Anfragezeiten wie auf den Speicherplatzbedarf. Die möglichen Datenräume können niedrig- oder hochdimensional sein. Als Anfragen müssen Exact-Match-Suchen, Bereichssuchen und Nächste-Nachbarsuchen unterstützt werden.

Alle vorgestellten Indexstrukturen unterstützen den Zugriff auf Punktdaten über Exact-Match-Suchen, Bereichssuchen und Nächste-Nachbarsuchen. Das Grid-File unterstützt Bereichssuchen und Nächste-Nachbarsuchen jedoch nur unter größerem Aufwand.

Der Speicherplatzbedarf beim R-Baum ist im Vergleich zu den anderen vorgestellten Indexstrukturen in der Regel am Größten, da für jede MBR die Punkte s und t sowie ein Verweis gespeichert werden müssen. Im Vergleich dazu speichert der LSD-Baum für jeden Knoten nur die Splitdimension, die Splitposition und zwei Verweise. Die Höhe, und damit die Anzahl der Knoten des R-Baums, ist abhängig von der Anzahl der maximalen Einträge und der Anzahl der Tupel. Beim LSD-Baum ist die Anzahl der Knoten abhängig von der Anzahl der Buckets und damit auch von der Tupelzahl. Beim Grid-File müssen das Grid-Directory und die Skalen gespeichert werden. Für jede Dimension muss eine Skala bestehend aus den Splitwerten gespeichert werden. Das Grid-Directory speichert für jede Zelle zudem eine Seitenadresse. Die Skalen und das Grid-Directory sind abhängig von den Buckets und somit von der Tupelzahl. Der R-Baum, der LSD-Baum und das Grid-File benötigen für die Bearbeitung einiger Anfragen, insbesondere bei Nächste-Nachbaranfragen, zusätzlichen Arbeitsspeicher, um die parallele Bearbeitung von Teilbäumen beziehungsweise Grid-Regionen zu ermöglichen. Im Gegensatz dazu müssen beim VA-File die Liste der Approximationen sowie die Intervalleinteilung gespeichert werden, wobei die Intervalleinteilung fest ist und im Programmspeicher abgelegt werden kann. Die Größe des VA-Files ist direkt abhängig von der Tupelzahl, der Dimension und der Zahl der Intervalle pro Dimension.

Der R-Baum, der LSD-Baum und das Grid-File sind für niedrigdimensionale Datenräume geeignet. Das VA-File wurde für hochdimensionale Datenräume entwickelt. Unter anderem aus diesem Grund wird das VA-File als eine mehrdimensionale Indexstruktur implementiert und analysiert. Als zweite Indexstruktur wird ein Verfahren für niedrigdimensionale Daten untersucht. Dabei ist dem LSD-Baum wegen seines geringeren Speicherbedarfs dem R-Baum vorzuziehen. Das Grid-File wird aufgrund des größeren Aufwands bei Bereichssuchen und Nächsten-Nachbarsuchen nicht weiter betrachtet.

Kapitel 5

Implementierung

Dieses Kapitel beschäftigt sich mit der Realisierung mehrdimensionaler Anfragen und unterstützender Indexstrukturen auf Grundlage der im vorherigen Kapitel getroffenen Auswahl. Dabei wird besonderer Wert auf die Erfüllung der Anforderungen aus Abschnitt 3.4 für die Nutzung in tief eingebetteten Systemen gelegt. Die Indexstrukturen werden zu diesem Zweck an den Anwendungsbereich angepasst. Abschnitt 5.1 beschäftigt sich mit der Integration in das bestehende Datenbankmanagementsystem RobbyDBMS. Es wird dabei auf die Wechselwirkungen der neu implementierten Programmteile mit den bestehenden Teilen eingegangen. Auf die Abhängigkeiten der bestehenden Features untereinander wird nicht eingegangen. Näheres dazu ist der Literatur zu entnehmen [Lie08]. Danach werden in Abschnitt 5.2 allgemeine Festlegungen zur Realisierung des mehrdimensionalen Zugriffes erläutert. Der Abschnitt 5.3 beschäftigt sich mit der Implementierung mehrdimensionaler Anfragen ohne die Unterstützung einer Indexstruktur. Die Implementierung der Indexstrukturen und der damit unterstützten Anfragen wird in den Abschnitten 5.4 und 5.5 beschrieben. Dabei wird zuerst auf das VA-File und danach auf den LSD-Baum eingegangen.

5.1 Erweiterung von RobbyDBMS

RobbyDBMS, wie es im Abschnitt 2.2.4 beschrieben wird, ist feature-orientiert aufgebaut. Das heißt, dass alle Programmfunktionalitäten gemäß der Erfüllung der gestellten Anforderungen in Merkmalen, den Features, gekapselt sind. Dieser Ansatz wird auch bei der Erweiterung von RobbyDBMS um mehrdimensionale Anfragen und unterstützende Indexstrukturen weiter verfolgt. Daher wird im folgenden einiges zur feature-orientierten Programmierung von RobbyDBMS erläutert. Danach im Abschnitt 5.1.2 wird die Integration der für diese Arbeit realisierten Features in RobbyDBMS beschrieben.

5.1.1 Feature-orientierte Programmierung von RobbyDBMS

Bei der feature-orientierten Programmierung steht, wie bereits im Kapitel 2 (Seite 19) beschrieben, das Feature als Merkmal der Software im Mittelpunkt. Die feature-orientierte Programmierung ist ein Programmierparadigma zur Entwicklung von *Softwareproduktlinien*. Eine Softwareproduktlinie umfasst mehrere Software-Produkte, die durch Merkmale voneinander unterschieden werden können [Nor02]. Die Software-Produkte gehören

einem gemeinsamen Softwarebereich an [Nor02]. Ziel bei Softwareproduktlinien ist die Wiederverwendung von Code [Nor02].

Ein Feature-Diagramm stellt, wie im Kapitel 2 (Seite 18) beschrieben, den Zusammenhang der Features dar. Die Symbole eines Feature-Diagramms werden in Abbildung 2.6 (Seite 18) erläutert.

Für die Programmierung von RobbyDBMS wird FeatureC++ als Spracherweiterung von C++ zur Realisierung feature-orientierter Programmierung genutzt [Lie08]. Ein Feature wird durch ein oder mehrere Software-Einheiten, den *Mixins*, realisiert [ALRS05]. Alle Mixins eines Features werden zu einem *Mixin Layer* zusammengefasst [ALRS05]. Dieses umfasst alle Erweiterungen, die zur Realisierung des Features notwendig sind. Ein Mixin erweitert und verfeinert eine gegebene Klasse durch Code-Fragmente [ALRS05]. Die Erweiterung von Klassen wird Kollaboration genannt [ALRS05]. Ein Mixin Layer kann dabei eine vollkommen neue Klasse enthalten, die dann von nachfolgenden Features als Basis zur Erweiterung genutzt werden kann. Alle Mixins werden in FeatureC++ zu einer Ordnerstruktur zusammengefasst, wobei ein Ordner immer genau einem Mixin Layer entspricht [ALRS05]. Die Reihenfolge der Features wird durch eine Datei festgelegt. Der FeatureC++-Compiler ist ein Precompiler, der aus dem FeatureC++-Code C++-Code erzeugt und der dann durch einen beliebigen C++-Compiler übersetzt wird [ALRS05].

5.1.2 Integration der neuen Features in RobbyDBMS

Die mehrdimensionalen Anfragen und Indexstrukturen werden in Featuren gekapselt und in die Featurestruktur von RobbyDBMS integriert. Dabei ist eine Anpassung der Struktur von RobbyDBMS notwendig. Abbildung 5.1 zeigt das entstehende Feature-Diagramm.

Das Feature „Read“ der API wird untergliedert in die Features „one-dimensional“ und „multi-dimensional“, welche die eindimensionalen beziehungsweise die mehrdimensionalen Anfragen realisieren. Die bestehende Exact-Match-Suche über die TID wird den eindimensionalen Anfragen durch das Feature „exactTID“ zugeordnet und ist verpflichtend, da das Löschen von Tupeln darüber implementiert wird. Die mehrdimensionale Bereichssuche, die mehrdimensionale Exact-Match-Suche und die Nächste-Nachbarsuche gehören zu den mehrdimensionalen Anfragen. Sie werden durch die Features „range“ für die Bereichssuche, „exact“ für die Exact-Match-Suche und „nns“ für die Nächste-Nachbarsuche realisiert und sind optional. Das heißt, sie können gemäß der Anwendung ausgewählt werden. Das Feature „Write“ der API wird nicht dementsprechend untergliedert. Die durch dieses Feature realisierten Operationen sind das Einfügen und das Löschen von Daten. Das Löschen erfolgt über die TID des Datensatzes. Eine Einteilung nach der Dimension ist somit nicht notwendig.

Die Erweiterung von RobbyDBMS um mehrdimensionale Indexstrukturen erfordert eine Neuordnung des optionalen Features „Index“. Das Feature wird in die Features „one-dimensional“ und „multi-dimensional“ untergliedert, dabei erfolgt eine weitere Unterteilung dieser Features in die einzelnen implementierten Indizes. In RobbyDBMS wurde als Index das lineare Hashen implementiert. Dieser Index wird in der neuen Untergliederung dem Feature „one-dimensional“ zugeordnet. Die mehrdimensionalen Indexstrukturen VA-File und LSD-Baum werden entsprechend dem Feature „multi-dimensional“ zu-

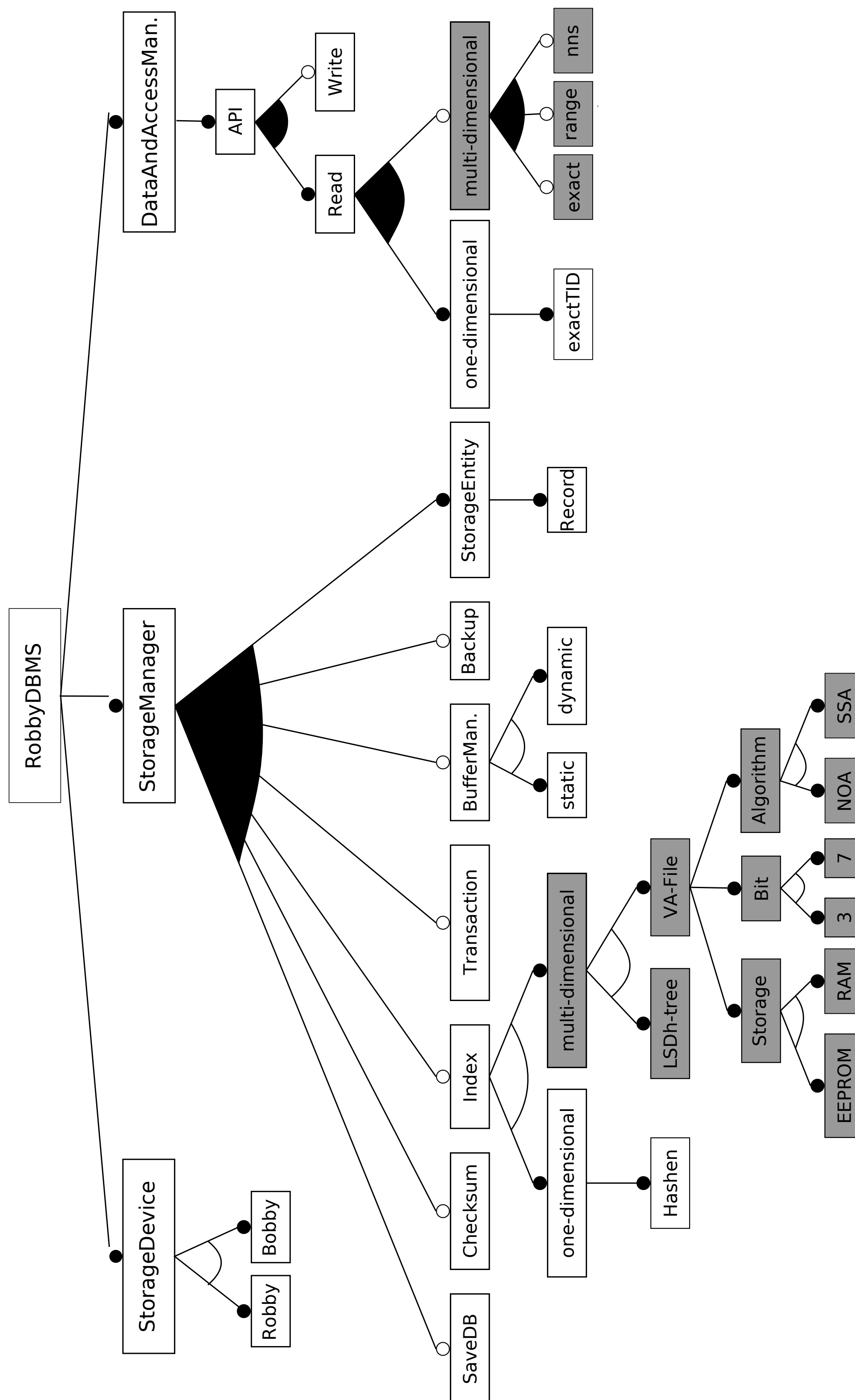


Abbildung 5.1: Feature-Diagramm von RobbyDBMS mit mehrdimensionalen Anfragen und unterstützenden Indexstrukturen (grau hinterlegt)

geordnet. Für bestimmte Merkmale des VA-Files sind mehrere Varianten möglich. Diese Varianten werden durch untergeordnete Features realisiert. Weiteres zu den Varianten des VA-Files wird im Abschnitt 5.4 beschrieben. Beim LSD-Baum sind verschiedene Varianten bezüglich der Kapazität der Buckets möglich. Eine Ausgliederung in weitere Features erfolgt jedoch nicht, da eine Änderung der Algorithmen nicht notwendig ist. Die verwendete Variante des LSD-Baums wird durch den Parameterwert festgelegt. Nähere Erläuterungen dazu folgen im Abschnitt 5.5. Eine Ausprägung von RobbyDBMS kann immer nur eine Indexstruktur enthalten.

Zwischen den neuen und den bestehenden Features von RobbyDBMS sind bestimmte Abhängigkeiten und Wechselwirkungen festzustellen. So bestehen Wechselwirkungen zwischen dem Feature „Write“ und den einzelnen Indexstrukturen, die eine gegliederte Realisierung der Indexstrukturen notwendig machen. Somit wird zum Beispiel der LSD-Baum durch eine Read- und eine Write-Komponente realisiert, um auch bei einer Ausprägung ohne „Write“ den Index nutzen zu können. Weitere Wechselwirkungen sind zwischen den mehrdimensionalen Anfragen und den Indexstrukturen festzustellen. So ändern sich die Algorithmen, die zur Realisierung der Anfragen genutzt werden, mit der verwendeten mehrdimensionalen Indexstruktur und im Falle des VA-Files und der Nächsten-Nachbarsuche mit dem ausgewählten Algorithmus. Eine mehrdimensionale Indexstruktur ist zudem nur dann sinnvoll, wenn mehrdimensionale Anfragen genutzt werden.

Des Weiteren bestehen Wechselwirkungen zwischen der Transaktionsverwaltung, realisiert durch das Feature „Transaction“, und den Anfragearten. Die bestehende Transaktionsverwaltung ist darauf ausgelegt, ohne mehrdimensionale Anfragen möglichst ressourcenschonend zu arbeiten. Dies führt dazu, dass insbesondere die Nächste-Nachbarsuche und die Bereichssuche nicht ohne große Veränderungen in die Transaktionsverwaltung integriert werden können. Dies ist vor allen auf der Tatsache begründet, dass das Löschen oder Einfügen von Datensätzen nachhaltig den Ablauf der Anfragen beeinflussen kann. Da die Transaktionsverwaltung optional ist und für die weiteren Betrachtungen wenig Bedeutung hat, wird auf die Implementierung einer angepassten Transaktionsverwaltung innerhalb dieser Arbeit verzichtet.

Der folgende Abschnitt beschäftigt sich mit den allgemeinen Festlegungen, die für die Implementierung der mehrdimensionalen Anfragen sowie der unterstützenden Indexstrukturen getroffen wurden.

5.2 Allgemeine Festlegungen

In der Anforderungsanalyse im Abschnitt 3.4 wurden die Exact-Match-Suche, die Bereichssuche und die Nächste-Nachbarsuche als wichtige Anfragen herausgearbeitet und in Abschnitt 4.2 näher erläutert. Zur Implementierung mit und ohne Indexunterstützung ist es wichtig, festzulegen welche Parameter von vornherein im DBMS gespeichert werden müssen und welche für die spezielle Anfrage an das DBMS übergeben werden müssen.

Ein Datensatz ist im Folgenden immer wie in Abbildung 5.2 aufgebaut. Dieser Aufbau bezieht sich auf die innere Struktur eines Tupels, wie sie im Abschnitt 3.4 vorgestellt wurde. Ein Datensatz besteht immer aus der TID, der Größe des Tupels, den Suchattributen und den weiteren Attributen, welche die Nutzdaten bilden. Die TID sowie jedes

Attribut hat eine Größe von 16 Bit und somit einen für viele Anwendungen ausreichend großen Wertebereich von 0 bis 65535. Die Größe des Tupels entspricht der Summe der Größen aller Attribute. Die Anzahl an Dimensionen, das heißt die Anzahl der Suchattribute, ist innerhalb der Datenbank konstant, um eine Vergleichbarkeit der Datensätze untereinander zu garantieren. Die Dimensionsanzahl wird im DBMS gespeichert. Die Datensätze sind sequentiell ohne Lücken im EEPROM abgelegt. Beim Löschen von Datensätzen werden diese nicht physisch gelöscht, sondern erhalten als TID den Wert 0. Damit wird der entsprechende Datensatz als gelöscht gekennzeichnet. Als gelöscht gekennzeichnete Datensätze müssen bei der Bearbeitung der mehrdimensionalen Anfragen übersprungen werden.

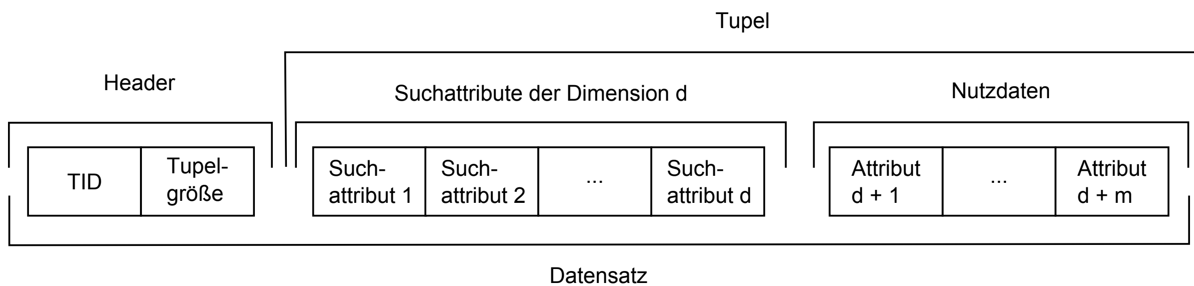


Abbildung 5.2: Aufbau eines Datensatzes

Die Exact-Match-Suche benötigt einen Wert für jedes der d Suchattribute und gibt den entsprechenden Datensatz zurück, falls es ihn gibt. Um dies zu realisieren, ist es günstig für die Programmierung in C++, der Funktion einen Zeiger auf einen leeren Datensatz im Arbeitsspeicher zu geben und in den Suchattributen dieses leeren Datensatzes die entsprechenden Werte zu hinterlegen. Die Funktion schreibt dann in den bis auf die Suchattribute leeren Datensatz die Werte des gefundenen Datensatzes. Somit wird der leere Datensatz um die restlichen Attribute sowie die TID und die Größe des Datensatzes ergänzt. Wichtig ist, dass der leere Datensatz immer größer oder genauso groß wie der gefundene Datensatz ist, da es sonst zu einem sogenannten Speicherleck im RAM kommen kann und weiterer nicht zum Datensatz gehörender Arbeitsspeicher beschrieben wird. Dies kann Folgen für das gesamte Programm des eingebetteten Systems haben. Der Zugriff auf den leeren Datensatz erfolgt mit Hilfe des Zeigers. Wird kein entsprechender Datensatz gefunden, wird der leere Datensatz nicht verändert. Zusätzlich gibt eine boolean-Variable den Erfolg der Anfrage zurück. Gibt es mehrere Datensätze mit den gleichen Suchattributwerten, so wird immer der zuerst gefundene Datensatz zurückgegeben.

Die Übergabe- und Rückgabewerte der Nächste-Nachbarsuche werden ähnlich der Exact-Match-Suche realisiert. Der Funktion wird ein Zeiger auf einen bis auf die Suchattribute leeren Datensatz übergeben. Der leere Datensatz enthält als Suchattribute die Koordinaten des Punktes, zu dem der nächste Nachbar gesucht wird. Ist der nächste Nachbar gefunden, werden alle Werte des leeren Datensatzes durch die des gefundenen ersetzt. Die Koordinaten des Ausgangspunktes gehen verloren. Ausgegeben wird so immer ein Punkt, auch wenn mehrere Punkte die gleiche Distanz haben. Zur Bestimmung des Erfolgs der Anfrage wird wiederum eine boolean-Variable zurückgegeben.

Die Bereichssuche besteht aus zwei zusammengehörigen Funktionen. Die erste Funktion führt mit Hilfe eines übergebenen, mehrdimensionalen Felds, das die minimalen

und maximalen Werte für jedes Suchattribut enthält, die Bereichssuche aus und speichert die Adressen aller gefundenen Datensätze innerhalb des DBMS in einem Stack. Dieser Stack muss dabei so groß sein, dass die Adresse jedes Datensatzes im EEPROM hineinpasst. Die zweite Funktion gibt bei jedem Aufruf den Datensatz zurück, dessen Adresse zu oberst im Stack liegt, und löscht diese Adresse vom Stack. Diese Funktion bekommt einem Zeiger auf einen leeren Datensatz im Arbeitsspeicher und schreibt in diesen den gefundenen Datensatz aus dem EEPROM. Beide Funktionen geben durch eine boolean-Variable zurück, ob sie erfolgreich waren. Vor jedem Einfügen, Löschen oder einer anderen mehrdimensionalen Anfrage wird der Stack gelöscht. Dies geschieht zum einen, um die Konsistenz zu gewährleisten und zum anderen um den Arbeitsspeicher, der durch den Stack belegt wird, freizugeben.

Für die Bearbeitung aller Anfragen und die Dimensionierung des Stacks für die gefundenen Tupel der Bereichssuche ist es hilfreich und teilweise notwendig die Anzahl der Datensätze in der Datenbank zu kennen. Diese Anzahl wird daher in einer Variable gespeichert.

Als Distanzmetrik für die Nächste-Nachbarsuche wird die Manhattan-Distanz festgelegt, die sich für zwei Punkte p und q mit d Dimensionen wie folgt berechnen lässt [SHS05]:

$$Distanz = \sum_{i=1}^{d} |p_i - q_i|$$

Die Suchattributwerte werden dabei unabhängig ihres Datentyps als 16 Bit lange vorzeichenlose, ganze Zahlen interpretiert. Eine Anpassung der Implementierungen an eine andere, an die Anwendung angepasste Metrik ist ohne Veränderung der Grundalgorithmen möglich. Der Vorteil der Manhattan-Distanz gegenüber anderen Distanzmetriken ist der relativ geringe Rechenaufwand [SHS05].

5.3 Implementierung mehrdimensionaler Anfragen ohne Index

Ohne die Unterstützung einer Indexstruktur erfolgt der Zugriff auf die Datensätze der Datenbank sequentiell. Das bedeutet, dass jeder Datensatz untersucht werden muss, ob er der Gesuchte ist. Die dazu notwendigen Algorithmen für die Realisierung sind einfach. Der Nachteil sind die hohen Zugriffskosten auf den Hintergrundspeicher [SHS05].

Abbildung 5.3 zeigt den Pseudocode für die mehrdimensionale Exact-Match-Suche ohne unterstützenden Index. Das Vorgehen bei der sequentiellen Bereichssuche und bei der Nächsten-Nachbarsuche ist analog. Es sind lediglich einige Anpassungen bei der Bearbeitung eines gültigen Datensatzes notwendig (Zeile 10 bis 19).

Bei der sequentiellen Realisierung der Exact-Match-Suche muss jeder gespeicherte Datensatz betrachtet werden, bis der gesuchte gefunden ist. Dieser Vorgang wird im Pseudocode durch die Zeilen 10 bis 19 beschrieben. Der Zugriff auf die gespeicherten Datensätze wird über die EEPROM-Adresse realisiert. In der Variable `cur` wird die Adresse des aktuell betrachteten Datensatzes gespeichert. Da die Datensätze sequentiell ohne Lücken im EEPROM abgelegt sind, erfolgt der Sprung zum nächsten Datensatz über das Addieren der aktuellen Adresse `cur` und der Größe des durch den aktuellen

```
1  cur: Adresse des aktuell untersuchten Datensatzes im Speicher
2  r: übergebener leerer Datensatz mit den Suchattributen
3  N: Anzahl der Datensätze in der Datenbank
4  D: Anzahl an Dimensionen
5
6  bool exactMulti (r){
7    for (n=1 to N){
8      cur= Addresse des nten Datensatzes;
9
10     if (Datensatz an Adresse cur nicht gelöscht){
11       for (d=1 to D){
12         if (Attribut d von r nicht gleich Attribut d vom Datensatz an cur){
13           Datensatz an cur nicht der gesuchte
14           abbrechen der for-Schleife;
15         }
16       }
17
18       if (for-Schleife nicht Abgebrochen){
19         Daten von Datensatz an Adresse cur in r schreiben;
20         return true;
21       }
22     }
23   }
24
25   keiner der N Datensätze ist der gesuchte Datensatz;
26   return false;
27 }
```

Abbildung 5.3: Pseudocode für mehrdimensionale Exact-Match-Suche ohne Indexunterstützung

Datensatz belegten Speichers. Die Größe des durch den aktuellen Datensatz belegten Speichers setzt sich aus der Größe des Headers und der Größe des Tupels zusammen. Die Größe des Tupels ist im Header gespeichert und wird somit aus dem Speicher ausgelesen. Im Pseudocode wurde dieser Vorgang in Zeile 7 zusammengefasst zur Bestimmung der Adresse des n'ten Datensatzes im EEPROM.

Ist ein entsprechender Datensatz gefunden, so wird die Suche abgebrochen (Zeile 16 bis 19). Bei der Bereichssuche und der Nächsten-Nachbarsuche entfällt dies. Dafür müssen dort entsprechende EEPROM-Adressen zwischengespeichert werden. Bei der Nächsten-Nachbarsuche erfolgt die Zwischenspeicherung der EEPROM-Adresse des Datensatzes mit der bisher kleinsten gefundenen Distanz in einer Variable. Bei der Bereichssuche werden alle EEPROM-Adressen entsprechender Datensätze im Stack gespeichert.

Die Implementierung der mehrdimensionalen Anfragen ohne die Unterstützung einer Indexstruktur bildet die Basis für die Analyse der Indexstrukturen im folgenden Kapitel.

5.4 Implementierung des VA-Files

Das VA-File, welches im Abschnitt 4.3.4 vorgestellt wurde, dient ursprünglich zur Unterstützung der Nächsten-Nachbarsuche in hochdimensionalen Datenräumen, wird jedoch auch zur Unterstützung der mehrdimensionalen Bereichssuche sowie der mehrdimensionalen Exact-Match-Suche genutzt. Die Algorithmen zur Realisierung der mehrdimensionalen Anfragen basieren ebenso wie die Algorithmen ohne Indexunterstützung auf dem sequentiellen Durchlauf aller Datensätze. Der Unterschied ist, dass nicht die konkreten Datensätze sequentiell durchlaufen werden, sondern die Liste der Approximationen

[Sch05a].

Die Position einer Approximation im VA-File entspricht der Position des dazugehörigen Datensatzes im EEPROM. Ist die Größe der Tupel variabel, führt dies dazu, dass zur Bestimmung der Adresse eines Datensatzes an beliebiger Stelle jede Tupelgröße der vorherigen Datensätze aus dem EEPROM geholt werden muss. Auf diese Weise ist das Bestimmen der Adresse eines Datensatzes im EEPROM sehr aufwendig. Die Bestimmung der Adresse eines Datensatzes im EEPROM an beliebiger Stelle ist jedoch für die Realisierung der Anfragen wichtig. Eine Möglichkeit ist, die Adressen der gespeicherten Datensätze in einer Liste zu verwalten. Diese Lösung benötigt jedoch relativ viel Speicher, da für jeden Datensatz zwei Byte gespeichert werden müssen. Eine andere, ressourcensparendere Möglichkeit ist, die Tupelgröße von vornherein für jede Anwendung festzulegen und die EEPROM-Adressen der Datensätze somit zu berechnen. Diese Variante wird für das in dieser Arbeit implementierte VA-File genutzt, bringt jedoch einen Nachteil bezüglich der Verwendung gegenüber dem LSD-Baum und der Anfragebearbeitung ohne Indexunterstützung, die beide variable Tupelgrößen erlauben, mit sich.

Für das VA-File gibt es verschiedene Möglichkeiten der Realisierung, so kann zum Beispiel der Speicherort des VA-Files entweder der RAM oder der EEPROM sein. Beide Varianten haben Vor- und Nachteile.

Die Vorteile bei der Speicherung im RAM sind die Geschwindigkeit des Zugriffes, die im Vergleich zum EEPROM höher ist, und dass kein Speicher im EEPROM für Datensätze blockiert wird. Diese Variante setzt jedoch einen ausreichend großen Arbeitsspeicher voraus. Ein Nachteil der Speicherung im RAM ist, dass die maximale Anzahl der zu speichernden Datensätze vor dem ersten Einfügen festgelegt werden muss, um entsprechend viel Arbeitsspeicher reservieren zu können. Auf die Liste der Approximationen wird über zwei Zeiger zugegriffen, von denen einer auf den Anfang und einer auf das Ende der Liste zeigt.

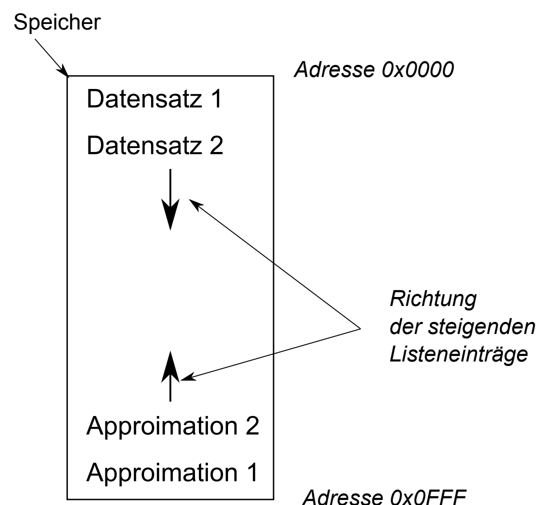


Abbildung 5.4: Variante mit Speicherung im EEPROM: Speicherung der Listen

Die zweite Variante mit der Speicherung im EEPROM ist unabhängig von der Größe des Arbeitsspeichers, dafür sind längere Zugriffszeiten zu erwarten sowie weniger Datensätze bei gleicher Speichergröße speicherbar. Die Liste der Datensätze und die Liste

der Approximationen sind gegenläufig zueinander implementiert. Das heißt, die Listen wachsen aufeinander zu. Abbildung 5.4 veranschaulicht diesen Aufbau. Eine Trennung beider Listen muss ebenfalls auf Betriebssystemebene erfolgen und wird durch einen Vergleich der Adressen vor der Speicherung realisiert. Erfolgt eine Überschneidung der Adressen, so wird der Einfügevorgang abgebrochen und das Einfügen schlägt fehl.

Ein wichtiger Parameter für das VA-File ist die Anzahl der Bits b_d, die den Wert einer Dimension d approximieren [WB97]. Im Abschnitt 4.3.4 wurde bereits das Vorgehen bei der Approximation beschrieben. Die 2^{b_d} Intervalle jeder Dimension werden durchnummeriert und werden anhand dieser Nummer eindeutig identifiziert. Durch diese Intervalle wird der Datenraum in Hyperquader gegliedert, deren Signatur die Approximation aller enthaltenen Punkte bildet. Die Signatur und somit auch die Approximationen bestehen dabei aus den Nummern der Intervalle der einzelnen Dimensionen. Für jede Dimension d werden daher b_d Bits benötigt [WB97]. In dieser Arbeit werden alle Dimensionen symmetrisch behandelt und durch b Bits approximiert. Dies erfolgt aufgrund dessen, dass die Verteilung von Realdaten unbekannt ist. So wird keine Dimension benachteiligt behandelt.

b	Intervalle	Eignung
1	2	ungeeignet, da zu geringe Teilung
2	4	ungeeignet, da schlecht in einem Byte unterzubringen
3	8	geeignet
4	16	ungeeignet, da schlecht in einem Byte unterzubringen
5	32	ungeeignet, da schlecht in einem Byte unterzubringen
6	64	ungeeignet, da schlecht in einem Byte unterzubringen
7	128	geeignet
8	256	ungeeignet, da kein Platz für Löschbit
≥ 9	≥ 512	ungeeignet, da zu große Teilung und zu großes VA-File

Tabelle 5.1: Mögliche Werte für b und ihre Eignung

Die Größe des Wertes b bestimmt die Teilung des Datenraums sowie die Größe der Approximationen [WB97]. Für jede Approximation wird ein Löschbit benötigt, um gelöschte Datensätze zu kennzeichnen. Die kleinste, adressierbare Speichereinheit im Arbeitsspeicher und im EEPROM umfasst 8 Bit (=1 Byte). Die Aufteilung eines Wertes für eine Dimension auf zwei Byte sollte vermieden werden, da für den vollständigen Wert zwei Byte gelesen werden müssen. Dies ist zum Beispiel der Fall, wenn 4 Bit pro Dimension für die Approximation genutzt werden. Das sind mit dem Löschbit 5 Bit, die pro Dimension gespeichert werden müssen. Bereits der zweite zu speichernde Wert muss auf zwei Byte aufgeteilt werden. Tabelle 5.1 listet mögliche Werte für b, die dazugehörige Anzahl der Intervalle und die Eignung auf. Als Varianten werden 3 und 7 Bit ausgewählt, die auch vergleichend analysiert werden. Die Realisierung der 3-Bit-Variante erfolgt mit einem Puffer, um ein Byte nicht zweimal hintereinander lesen zu müssen.

Wichtig für das VA-File und somit für den Wirkungsgrad der Indexstruktur ist die Verteilung der Intervalle über den Wertebereichen [WB97]. Ein Attribut egal welcher Dimension ist eine 16 Bit lange, vorzeichenlose, ganze Zahl. Somit ist der Wertebereich für jede Dimension gleich groß. Das führt dazu, dass die Verteilung der Intervalle nur von der Verteilung der Daten abhängig ist. Die Verteilung von Daten in Anwendungen ist nicht

bekannt. Daher wird die Verteilung der Intervalle als gleichverteilt festgelegt. Dies eröffnet die Möglichkeit, die Intervallgrenzen nicht fest speichern zu müssen, da sie einfach berechnet werden können. Auf diese Weise wird zusätzlich Speicherplatz eingespart.

Das Prinzip des VA-Files beruht auf der Filterung der Datensätze mit Hilfe der Approximationen [Sch05a]. Die Abarbeitung der Anfragen wird allgemein in zwei Phasen gegliedert, in die Filterung und in die Untersuchung der Datensätze selbst [WB97].

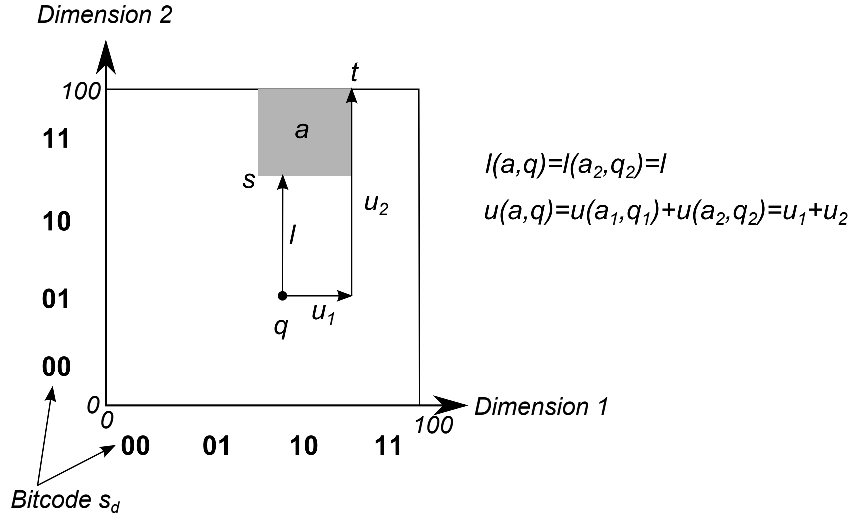


Abbildung 5.5: Obere und untere Grenze der Distanz nach [WB97]

Die Approximation eines Datensatzes beschreibt einen Hyperquader, in dem der Datensatz liegt. Für die Nächste-Nachbarsuche wird die Distanz zum gegebenen Datenpunkt q benötigt, um zu entscheiden, ob der Datensatz der nächste Nachbar ist. Mit Hilfe des Hyperquaders wird ein Intervall für die Distanz festgelegt [WB97]. Dieses Intervall wird für einen Datensatz p mit der Approximation $a = a_1 a_2 \dots a_d$ beschrieben durch die *untere Schranke* $l(a, q)$ und die *obere Schranke* $u(a, q)$ [WB97]. Die untere Schranke $l(a, q)$ und die obere Schranke $u(a, q)$ werden mit der Manhattan-Distanz bestimmt durch [WB97]:

$$l(a,q) = \sum_{i=1}^{d} l(a_i, q_i) \text{ mit } l(a_i, q_i) = \begin{cases} q_i - t_i & \text{wenn } q_i > t_i \\ s_i - q_i & \text{wenn } s_i > q_i \\ 0 & \text{sonst} \end{cases}$$

$$u(a,q) = \sum_{i=1}^{d} u(a_i, q_i) \text{ mit } u(a_i, q_i) = \begin{cases} q_i - s_i & \text{wenn } q_i > t_i \\ t_i - q_i & \text{wenn } s_i > q_i \\ max(s_i - q_i, q_i - t_i) & \text{sonst} \end{cases}$$

Die Werte von s_i und t_i entsprechen den Intervallgrenzen [WB97] und lassen sich durch die Gleichverteilung leicht berechnen. Abbildung 5.5 veranschaulicht die obere und untere Grenze der Distanz graphisch.

Weber und Blott schlagen bei der Vorstellung des VA-Files zwei Algorithmen für die k-Nächste-Nachbarsuche vor [WB97]. In dieser Arbeit werden beide Algorithmen in vereinfachter Form als Nächste-Nachbarsuchen implementiert.

```
1  cur: Adresse des aktuell untersuchten Datensatzes im Speicher
2  r: übergebener leerer Datensatz mit den Suchattributen
3  N: Anzahl der Datensätze in der Datenbank
4  D: Anzahl an Dimensionen
5  ans: Adresse des Datensatzes im Speicher mit der bisher kleinsten Distanz
6  dis: Distanz von ans zu r
7
8  bool nns(r) {
9    ans=ungültige Adresse;
10   dis=maximale Distanz+1;
11
12   for(n=1 to N){
13     if(l(n,r) < dis){
14       cur=Adresse von Datensatz n;
15       tmp= Distanz vom Datensatz an cur zu gegebenem r;
16       if(tmp < dis){
17         ans=cur;
18         dis=tmp;
19       }
20     }
21   }
22
23   if (ans gültige Adresse){
24     Daten von Datensatz an Adresse ans in r schreiben;
25     return true;
26   }
27
28   return false;
29 }
```

Abbildung 5.6: Pseudocode für SSA nach [WB97]

Der erste Algorithmus wird als *'Simple-Search' Algorithmus (SSA auf Deutsch „einfacher Suchalgorithmus“)* bezeichnet [WB97]. Abbildung 5.6 zeigt den Algorithmus für die Nächste-Nachbarsuche. Das VA-File wird dabei als einfacher Filter genutzt. Bei ungünstiger Reihenfolge der Datensätze im Speicher kann es vorkommen, dass für viele Datensätze die konkrete Distanz berechnet werden muss, da der nächste Nachbar erst relativ spät gefunden wird. Dafür wird kein zusätzlicher Arbeitsspeicher benötigt. Die Phasen Filterung und Untersuchung gehen ineinander über und sind nicht klar getrennt [WB97].

Der zweite Algorithmus wird als *'Near-Optimal' Algorithmus (NOA auf Deutsch „beinahe optimaler Algorithmus“)* bezeichnet [WB97]. Dieser Algorithmus trennt klar zwischen der Filterung und der Untersuchung der Datensätze. Abbildung 5.7 veranschaulicht die Arbeitsweise. In der Filterungsphase werden alle Approximationen sequentiell durchsucht und mögliche Kandidaten in einer sortierten Warteschlange gespeichert. In dieser Arbeit wird ein Heap als sortierte Warteschlange genutzt, der anhand der unteren Schranke der Distanz die Position des relevantesten Datensatzes sortiert ausgibt. In der zweiten Phase werden die relevantesten Kandidaten untersucht, solange es noch Kandidaten im Heap gibt, deren Distanz möglicherweise kleiner ist als die bisher gefundene. Ziel dieses Algorithmus ist, den Zugriff auf den EEPROM zu minimieren. Dies geht zulasten des Arbeitsspeichers sowie der Anzahl durchzuführender Berechnungen [WB97].

Die Realisierung der mehrdimensionalen Exact-Match-Suche und der mehrdimensionalen Bereichssuche sind ähnlich dem SSA, da eine Sortierung nach der Relevanz nicht möglich ist. Abbildung 5.8 zeigt den Pseudocode für die Exact-Match-Suche. Die Suche wird, ebenso wie bei der Exact-Match-Suche ohne Indexunterstützung, beendet, wenn ein entsprechender Datensatz gefunden wurde. Die Bereichssuche wird mit einigen Anpassungen mit demselben Grundalgorithmus realisiert.

```
cur: Adresse des aktuell untersuchten Datensatzes im Speicher
r: übergebener leerer Datensatz mit den Suchattributen
N: Anzahl der Datensätze in der Datenbank
D: Anzahl an Dimensionen
ans: Adresse des Datensatzes im Speicher mit der bisher kleinsten Distanz
dis: Distanz von ans zu r
h: sortierte Warteschlange

bool nns(r) {
  initialisiere h;
  ans=ungültige Adresse;
  dis=maximale Distanz+1;

  for(n=1 to N){
    if(l(n,r) < dis){
      if(u(n,r) < dis) dis= u(n,r);
      einfügen von (l(n,r),n) in h;
    }
  }

  dis=maximale Distanz+1;
  hole (l,i) aus h;

  while( l < dis ){
    cur= Adresse von Datensatz i;
    tmp= Distanz vom Datensatz an cur zu gegebenem r;
    if(tmp<dis){
      ans=cur;
      dis=tmp;
    }
    if(h leer) breche Schleife ab;
    hole nächstes (l,i) aus h;
  }

  if (ans gültige Adresse){
    Daten von Datensatz an Adresse ans in r schreiben;
    return true;
  }

  return false;
}
```

Abbildung 5.7: Pseudocode für NOA nach [WB97]

5.5 Implementierung des LSD-Baum

Der LSD-Baum wurde im Abschnitt 4.3.2 vorgestellt. Der Baum besteht aus inneren Knoten und Buckets. Die inneren Knoten liegen typischerweise im Arbeitsspeicher und die Buckets im Hintergrundspeicher. Die Implementierung dieser Arbeit weicht etwas von diesem Aufbau ab, indem die Speicherung der Buckets verändert wird. Dies hat den Vorteil, dass die Datensätze weiter sequentiell im EEPROM abgelegt werden. Bei der klassischen Realisierung, entsprechen die Buckets Seiten im Speicher, die mehr oder weniger gut gefüllt sind [HSW89]. Der Zugriff auf dem EEPROM erfolgt jedoch byteweise. Um Buckets im EEPROM zu realisieren, müssen entweder Lücken bei der Speicherung der Datensätze gelassen werden oder ein Bucket wird als verkettete Liste im EEPROM realisiert. Bei der Variante mit der verketteten Liste muss die Zahl der Verkettungen an geeigneter Stelle hinterlegt werden, um zu erkennen ob ein Bucket voll ist. Beide Varianten bedeuten einen höheren Speicherplatzbedarf im EEPROM und sind komplex zu implementieren. Eine einfachere Möglichkeit ist, in einem Bucket die Adressen der enthaltenen Datensätze zu speichern und dieses Bucket im Arbeitsspeicher zu halten. Die

```
1  cur: Adresse des aktuell untersuchten Datensatzes im Speicher
2  r: übergebener leerer Datensatz mit den Suchattributen
3  N: Anzahl der Datensätze in der Datenbank
4  D: Anzahl an Dimensionen
5
6  bool exactMulti (r){
7    for(n=1 to N){
8
9      if(Approximation an der Stelle n im VA-File gleich der Approximation von r){
10       cur=Adresse von Datensatz n;
11
12       for (d=1 to D){
13         if (Attribut d von r nicht gleich Attribut d vom Datensatz an cur){
14           Datensatz an cur nicht der gesuchte
15           abbrechen der for-Schleife;
16         }
17       }
18
19       if (for-Schleife nicht Abgebrochen){
20         Daten von Datensatz an Adresse cur in r schreiben;
21         return true;
22       }
23     }
24   }
25
26   return false;
27 }
```

Abbildung 5.8: Pseudocode für mehrdimensionale Exact-Match-Suche mit VA-File

letzte Variante wird in dieser Arbeit genutzt. Dies ermöglicht auch die einfache Realisierung einer individuell einstellbaren Kapazität der Buckets. Die Kapazität der Buckets wird vor dem Kompilieren festgelegt und hat Einfluss auf die Größe des Baums sowie auf die Geschwindigkeit des Zugriffes [HSW89].

Maßgeblich für die Größe des Baums sowie für die Zugriffsgeschwindigkeit ist die Splitstrategie [HSW89]. Die Verteilung von Daten in Anwendungen ist nicht bekannt. Daher wird in dieser Arbeit eine datenabhängige Splitstrategie, wie sie für den LSD^h-Baum vorgeschlagen wurde, genutzt [Hen98]. Diese Splitstrategie kann sowohl auf bekannten Verteilungen wie auch auf unbekannten Verteilungen arbeiten. Die Einfügereihenfolge hat jedoch Einfluss auf die Bucketauslastung und somit auf die Speicherauslastung. Ein Split erfolgt immer nur in einer Dimension und möglichst immer in einer anderen Dimension. Der Split in der Dimension erfolgt nur, wenn sich die Attributwerte der Datensätze im Bucket in dieser Dimension unterscheiden, sonst wird die nächste, mögliche Dimension genutzt. Die Position des Splits erfolgt genau im Mittelwert der Attributwerte der Splitdimension von allen Datensätzen im Bucket und dem Datensatz, der nicht mehr hineinpasst. Das heißt, es wird die Summe der Attributwerte der Splitdimension über den Datensätzen gebildet und durch die Kapazität der Buckets +1 dividiert und abgerundet auf die ganze Zahl.

Wird ein Datensatz gelöscht, so wird der Datensatz im EEPROM mit der TID 0 gekennzeichnet und die Adresse im entsprechenden Bucket gelöscht. Leere Buckets werden gelöscht und der Baum entsprechend angepasst. Das Löschen von Buckets und Knoten kann eine Fragmentierung des Speichers zu Folge haben. Die Fragmentierung des Speichers ist ein Problem des LSD-Baums. Das Löschen von Datensätzen sollte daher und aufgrund der Speicherbelegung des EEPROM die Ausnahme sein.

Die Bearbeitung der mehrdimensionalen Anfragen wird durch die Partitionierung

des Datenraums in Regionen unterstützt [HSW89]. Mit Hilfe der hierarchischen Struktur können teilweise komplette Teilbäume von der Betrachtung ausgeschlossen werden [HSW89]. Dies soll den Aufwand für Anfragen minimieren. Die Datensätze in den Buckets werden sequentiell durchsucht [HSW89]. Dieses Grundprinzip wird bei allen implementierten, mehrdimensionalen Anfragen angewendet.

```
1  cur: Adresse des aktuell untersuchten Datensatzes im Speicher
2  r: übergebener leerer Datensatz mit den Suchattributen
3  D: Anzahl an Dimensionen
4  k: aktuell betrachteter Knoten
5
6  bool exactMulti (r){
7    k=Wurzel;
8
9    while(k kein Bucket){
10     if(Attribut k.dim von r <= k.position)
11       k=linkes Kind von k;
12     else k= rechtes Kind von k;
13   }
14
15   for(i=1 to Anzahl Datensätze in k){
16     cur=Adresse an Stelle i des Buckets k;
17
18     for (d=1 to D){
19       if (Attribut d von r nicht gleich Attribut d vom Datensatz an cur){
20         Datensatz an cur nicht der gesuchte,
21         abbrechen der for-Schleife;
22       }
23     }
24
25     if (for-Schleife nicht Abgebrochen){
26       Daten von Datensatz an Adresse cur in r schreiben;
27       return true;
28     }
29   }
30
31   return false;
32 }
```

Abbildung 5.9: Pseudocode für die Exact-Match-Suche mit Unterstützung durch den LSD-Baum nach [HSW89]

Der Algorithmus zur Realisierung der mehrdimensionalen Exact-Match-Suche ist in Abbildung 5.9 dargestellt. Zuerst muss das Bucket gesucht werden, in dem der Datensatz gespeichert sein könnte (Zeile 9 bis 13). Danach werden alle Datensätze des gefundenen Buckets mit den gegebenen Suchattributwerten im Datensatz `r` verglichen. Wird ein Datensatz gefunden, so wird die Suche abgebrochen und der Datensatz ausgegeben. Wurden alle Datensätze des Buckets untersucht und kein passender Datensatz gefunden, dann ist kein gültiger Datensatz mit den gegebenen Suchattributen im Speicher [HSW89].

Die Realisierung der Nächsten-Nachbarsuche basiert auf dem von Henrich vorgestelltem Algorithmus für die k-Nächste-Nachbarsuche, der für den Spezialfall k=1 vereinfacht werden konnte [Hen94]. Abbildung 5.10 veranschaulicht das Vorgehen. Ein zentraler Bestandteil dieses Algorithmus ist eine sortierte Warteschlange, die in dieser Arbeit durch einen Heap realisiert wurde. Der Heap gibt die potentiell noch zu untersuchenden Knoten in aufsteigender Reihenfolge nach ihrer minimalen Distanz zum gegebenen Datenpunkt aus. Die minimale Distanz eines Knoten `k`, egal ob innerer Knoten oder Bucket, zu einem gegebenen Datenpunkt `r` wird durch die Funktion `l(k,r)` berechnet. Der Algorithmus beginnt bei der Wurzel, die immer eine minimale Distanz zum Datenpunkt von

```
cur: Adresse des aktuell untersuchten Datensatzes im Speicher
r: übergebener leerer Datensatz mit den Suchattributen
N: Anzahl der Datensätze in der Datenbank
D: Anzahl an Dimensionen
ans: Adresse des Datensatzes im Speicher mit der bisher kleinsten Distanz
dis: Distanz von ans zu r
q: sortierte Warteschlange
k: Knoten
lb: untere Schranke der Distanz von k zu r

bool nns(r){
  ans=ungültige Adresse;
  dis=maximale Distanz +1;
  initialisieren von q;

  k=Wurzel;
  lb=l(Wurzel,r);
  while(lb < dis){
    if(k ist Bucket){
      for(i=1 to Anzahl Datensätze in k){
        cur=Adresse an Stelle i des Buckets k;
        tmp= Distanz vom Datensatz an cur zu gegebenem r;
        if (tmp<distanz){
          ans=cur;
          dis=tmp;
        }
      }
    }else{
      if(l(linkes Kind von k,r)<dis)
        einfügen von(linkes Kind von k,l(linkes Kind von k,r)) in S;
      if(l(rechtes Kind von k,r)<dis)
        einfügen von(rechtes Kind von k,l(rechtes Kind von k,r)) in S;
    }
    hole nächstes (lb,k) aus S;
  }

  if (ans gültige Adresse){
    Daten von Datensatz an Adresse ans in r schreiben;
    return true;
  }
  return false;
}
```

Abbildung 5.10: Pseudocode für die Nächste-Nachbarsuche mit Unterstützung durch den LSD-Baum nach [Hen94]

0 hat, da der Datenpunkt im Datenraum enthalten sein muss. In der while-Schleife wird untersucht, ob der aktuell betrachtete Knoten ein Bucket oder ein innerer Knoten ist. Bei einem Bucket werden alle in ihm enthaltenen Datensätze untersucht. Ist der aktuelle Knoten ein innerer Knoten, so wird die minimale Distanz seiner Kinder zum Datenpunkt bestimmt und die Kinder gegebenenfalls in den Heap eingefügt. Nach Bearbeitung eines Knotens wird der nächste Knoten aus dem Heap geholt. Ist dessen minimale Distanz größer als die letzte gefundene kleinste Distanz eines konkreten Datensatzes, so wird die Suche abgebrochen. Der nächste Nachbar ist der letzte gefundene Datensatz. Der Algorithmus realisiert eine Tiefensuche durch den Baum [Sch05a].

Die Bereichssuche mit der Unterstützung des LSD-Baums arbeitet mit einem Stack, der die Knoten zwischenspeichert, die den Suchbereich schneiden. Ein Stack ist besonders einfach und speicherplatzschonend implementierbar. Abbildung 5.11 veranschaulicht den Algorithmus. Bei inneren Knoten wird untersucht, ob die Kinder den Suchbereich schneiden. Die Kinder werden dann gegebenenfalls bei Überschneidung im Stack zwi-

```
cur: Adresse des aktuell untersuchten Datensatzes im Speicher
find: Stack mit den Adressen der gefundenen Datensätze im Speicher
D: Anzahl an Dimensionen
search [D][2]: Feld mit dem Minimal- und dem Maximalwert jedes Suchattributes
S: Stack mit den abzuarbeitenden Knoten


bool range(search[][2]) {
  initialisiere find;
  initialisiere S;
  einfügen der Wurzel in S;

  while(S nicht leer){
    hole von Knoten k aus S;

    if(k ist Bucket){
        for(i=1 to Anzahl Datensätze in k){
          cur=Adresse an Stelle i des Buckets k;
          if(Datensatz von cur in search) hinzufügen von cur zu find;
        }
    }else{
      if(linkes Kind von k search schneidet)  einfügen des linken Kindes von k in S;
      if(rechtes Kind von k search schneidet) einfügen des rechten Kindes von k in S;
    }
  }

  return (find nicht leer);
}
```

Abbildung 5.11: Pseudocode für die Bereichssuche mit Unterstützung durch den LSD-Baum

schengespeichert. Die Untersuchung von Buckets bedeutet das sequentielle Durchsuchen aller enthaltenen Datensätze.

Kapitel 6

Analyse

Die Analyse der implementierten Indexstrukturen ist Gegenstand dieses Kapitels. Dabei wird auf die Varianten des VA-Files eingegangen. Die Analyse des LSD-Baums erfolgt anhand der beiden Extremwerte für die Bucketkapazität. Zu Beginn dieses Kapitels werden Vorbetrachtungen bezüglich der Durchführung und der Grenzen der Analyse erläutert. Der Abschnitt 6.2 betrachtet im Anschluss die Ergebnisse der Analyse. Am Ende dieses Kapitels findet eine Diskussion der Ergebnisse statt.

6.1 Durchführung und Grenzen der Analyse

Zum Verständnis und zur richtigen Beurteilung der Analyseergebnisse ist es notwendig, bestimmte Vorbetrachtungen anzustellen. Dieser Abschnitt beschäftigt sich dahin gehend mit der Durchführung sowie den Grenzen der Analyse.

Die Analyse erfolgt auf einem Roboter, der mit einem AT90CAN128 Mikrocontroller von Atmel ausgestattet ist. Dieser 8-bit Mikrocontroller wird für die Analyse im 16 MHz-Modus betrieben. Als Speicher stehen 128 KB Flash für Programmcode, 4 KB SRAM als Arbeitsspeicher für den Programmstack und Daten und 4 KB EEPROM für permanent zu speichernde Daten bereit. Auf Grundlage dieser Einschränkungen wird das Testsystem als tief eingebettetes System betrachtet. Der Mikrocontroller verfügt über mehrere 8- und 16-bit Timer zur Zeitmessung und Pulsweitenmodulation.[1]

Für die Analyse der Ausführungszeit wird ein 16-bit Timer genutzt. Der Timer wird mit den 16 MHz der internen Clock und einem Vorteiler von 1024 betrieben. Die Überläufe des Zählers werden über einen Interrupt in einer extra Variable gezählt. So ist eine zuverlässige Zeitmessung möglich. Geringe Abweichungen sind durch die Behandlung des Überlaufens zu erwarten.

Die genutzte Ausprägung von RobbyDBMS enthält neben den mehrdimensionalen Anfragen und den entsprechenden Indexstrukturen die obligatorischen Features und das optionale Feature „Write“ zum Einfügen der Datensätze. Die zugrunde liegende Geräteanbindung erfolgt durch das Feature „Robby“. Auf alle weiteren Features wird verzichtet.

Für das VA-File werden alle Varianten untersucht, das beinhaltet alle Kombinationen der möglichen Features des VA-Files. Die Kennzeichnung der Varianten des VA-Files

[1] Datenblatt des AT90CAN, Quelle: http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

erfolgt im Weiteren durch „VA - Bitzahl b - gegebenenfalls Algorithmus der Nächsten-Nachbarsuche". Die Bitzahl b gibt die Zahl der Bits, die zur Approximation einer Dimension genutzt werden, an.

Der LSD-Baum wird mit den Extremwerten für die Bucketkapazität untersucht, da diese großen Einfluss auf die Datenraumaufteilung hat. Die größte Datenraumaufteilung tritt bei einer Kapazität von einem Datensatz pro Bucket auf und die kleinste Aufteilung bei einem Bucket für alle Datensätze. Im Weiteren wird der LSD-Baum mit einer Bucketkapazität von einem Datensatz als *„LSD-1"* bezeichnet. Der andere LSD-Baum wird mit *„LSD-voll"* bezeichnet. Beim LSD-1 ist mit der schnellsten Anfrageausführung zu rechnen, da eine sequentielle Suche innerhalb der Buckets nicht notwendig ist und der Datenraum die größte Aufteilung erfährt. Dafür ist beim LSD-1 das Directory durch die große Aufteilung am größten. Dies führt zu einem hohen Arbeitsspeicherverbrauch. Beim LSD-voll sind Ausführungszeiten in der Nähe der Ausführungszeiten der Anfragen ohne Indexunterstützung zu erwarten, da das Bucket, in dem alle Datensätze enthalten sind, sequentiell durchsucht werden muss. Eine Datenraumaufteilung erfolgt praktisch nicht. Die Buckets sind bei beiden Varianten voll ausgelastet.

Als Analysedaten werden gleichverteilte Daten genutzt, da aus Umfangsgründen keine Realdaten gewonnen werden konnten. Ein Tupel wird für die Analyse auf die Suchattribute reduziert. Dadurch ist es möglich, eine größere Zahl von Datensätzen zu speichern und die Indexstrukturen dahin gehend zu untersuchen. Die Analyse der Ausführungszeit erfolgt für zwei-, drei- und vierdimensionale Daten. Der Grund für die Beschränkung auf diese Dimensionsanzahlen ist die mögliche Anzahl an Datensätzen, die bei Gleichverteilung gespeichert werden können, und die damit mögliche Anzahl der Analysen. Die Anzahl der Datensätze N bei einer Analyse hängt sowohl von der maximalen Anzahl der speicherbaren Datensätze N_{max} als auch von der Dimensionsanzahl D und der Teilung t des Wertebereichs einer Dimension ab. Die Teilung t des Wertebereichs einer Dimension ist aufgrund der Gleichverteilung für alle Achsen gleich. Es gilt:

$$N = t^D \leq N_{max}$$

Im Abschnitt 6.2.1 wird näher auf die maximale Anzahl an Datensätzen eingegangen.

Die Daten werden sortiert in die Datenbank eingefügt. Der LSD-Baum arbeitet mit einer datenabhängigen Splitstrategie. Die Höhe des Baums hängt somit unter anderem von der Einfügereihenfolge ab. Dies kann bei den gleichverteilten, sortierten Daten der Analyse ein Nachteil für den LSD-1-Baum sein. Jedoch hat die Höhe des LSD-Baums in der Regel keinen Einfluss auf den Zugriff auf den EEPROM, wenn das Directory ausschließlich im Arbeitsspeicher liegt. Die Untersuchung des LSD-1-Baums mit verschiedenen Splitstrategien erfolgt aus Gründen des Umfangs in dieser Arbeit nicht. Ebenso ist eine Analyse mit konkreten Realdaten nicht möglich. Für Schlussfolgerungen bezüglich Realdaten und bezüglich verschiedener Splitstrategien beim LSD-1-Baum sind separate Analysen notwendig.

Die Analyse in diesem Kapitel bewegt sich an der unteren Grenze des Einsatzes mehrdimensionaler Indexstrukturen zur Unterstützung mehrdimensionaler Anfragen. Dies bezieht sich auf die Anzahl der Datensätze sowie auf die Größe des verfügbaren Speichers. Die Interpretation der Ergebnisse sollte daher diese besondere Situation beachten. Die gezogenen Schlussfolgerungen können daher nicht ohne Prüfung auf andere Systemgrößen übertragen werden.

Die untersuchten Indexstrukturen vertreten nur beispielhaft die mehrdimensionalen Indexstrukturen. Schlussfolgerungen auf nicht untersuchte Indexstrukturen sind daher nicht möglich. Mögliche weitere Optimierungen der implementierten Indexstrukturen erfordern eine erneute Analyse.

6.2 Ergebnisse

Dieser Abschnitt beschäftigt sich mit den Ergebnissen der durchgeführten Analyse. Dabei wird zuerst auf die Kosten der Indexstrukturen und ihrer Algorithmen in Form von Speicherplatzbedarf eingegangen. Danach wird im Abschnitt 6.2.2 auf die Ausführungszeiten eingegangen. Der Vergleich basiert dabei immer auf der Implementierung der mehrdimensionalen Anfragen ohne die Unterstützung einer Indexstruktur.

6.2.1 Speicherplatzbedarf

Die Unterstützung der Anfragebearbeitung führt zu einem höheren Bedarf an Speicherplatz gegenüber der Ausprägung ohne Indexstrukturen. Damit kann der Speicherplatzbedarf als die Kosten für die schnellere Bearbeitung der Anfragen gesehen werden. Die Indexstrukturen benötigen zusätzlichen Speicher für den Programmcode sowie für die Daten. Der Bedarf an zusätzlichem Speicher für Daten wirkt sich auf die maximal mögliche Anzahl an Datensätzen aus. Diese Anzahl wird im folgenden Abschnitt analysiert. Danach wird der Programmspeicherbedarf für die Indexstrukturen untersucht.

Maximale Anzahl von Datensätzen

Die Indexstrukturen benötigen Datenspeicher zur Unterstützung der Anfragen. Da der Datenspeicher, das heißt der Arbeitsspeicher und der EEPROM, begrenzt ist, wirkt sich der Bedarf an zusätzlichem Speicher auf die maximale Anzahl möglicher Datensätze aus. Die maximal mögliche Anzahl an Datensätzen gibt für die Analyse der Ausführungszeit der Anfragen die obere Grenze der Anzahl an Analysedaten an. Die Dimensionsanzahl D der Datensätze hat Einfluss auf die maximale Anzahl von Datensätzen, daher wird diese Anzahl für die zwei-, drei- und vierdimensionale Daten analysiert. Gleichverteilte Daten dieser Dimensionen werden auch bei der Analyse der Ausführungszeit genutzt. Abbildung 6.1 zeigt das Ergebnis der Analyse der maximalen Anzahl an Datensätzen.

Die Werte für die maximale Anzahl N_{max} von Datensätzen ohne die Unterstützung einer Indexstruktur ergeben sich aufgrund der Größe des EEPROM, der ausschließlich mit Datensätzen gefüllt ist. Diese Werte geben die obere Grenze der maximalen Anzahl an. Der Arbeitsspeicher wird bei der Ausführung der Bereichssuche mit einem Feld der Größe $2 \cdot N$ Byte für die Adressen der gefundenen Datensätze belegt, wobei N die Anzahl der Datensätze in der Datenbank ist. Im Testsystem mit der Analyseanwendung bedeutet das, dass eine Einschränkung der Anzahl an Datensätzen, um die Bereichssuche ohne Indexunterstützung zu ermöglichen, nicht notwendig ist.

Die Variante des VA-Files mit der Speicherung im Arbeitsspeicher ermöglicht eine vollständige Füllung des EEPROM ausschließlich mit Datensätzen, da das VA-File im RAM gespeichert wird. Der RAM ist dabei ausreichend für das VA-File und das Feld

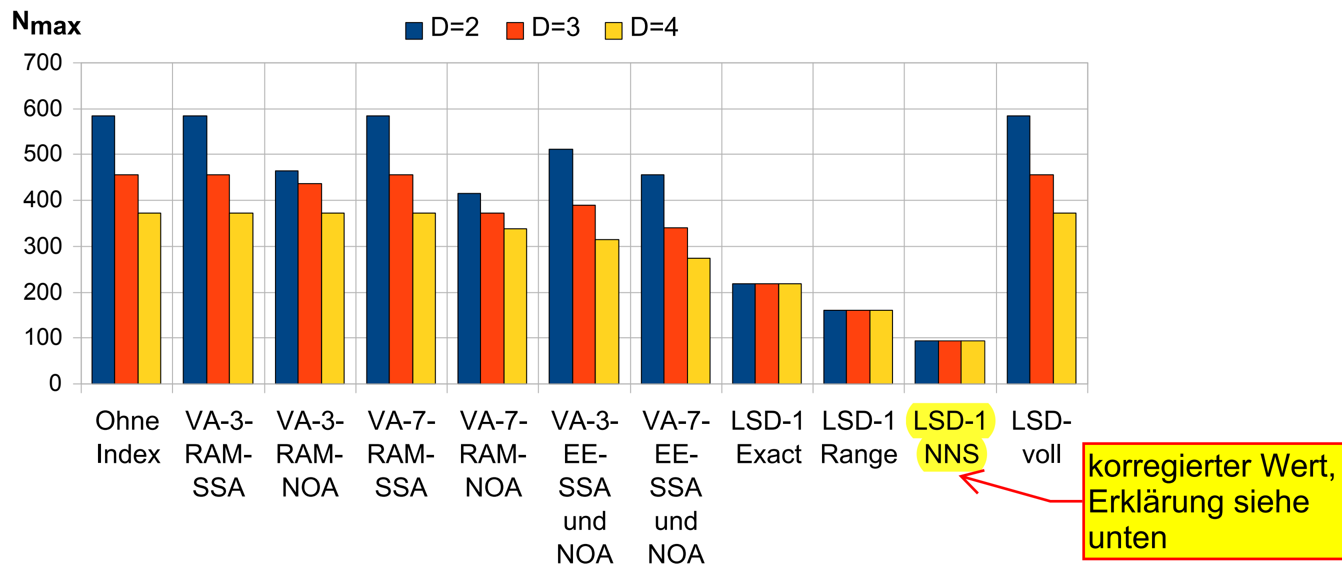


Abbildung 6.1: Analyse der maximalen Anzahl von Datensätzen für $D = 2$, $D = 3$ und $D = 4$

für die Bereichssuche. In der Variante mit $b = 3$ ergibt sich eine Größe des VA-Files von $N \cdot D \div 2$ mit Aufrunden auf ganze Byte und bei $b = 7$ von $N \cdot D$ Byte. Für NOA als Algorithmus zur Bearbeitung der Nächsten-Nachbarsuche sind zusätzlich $N \cdot 7$ Byte Arbeitsspeicher für die sortierte Warteschlange notwendig. Dies führt zu einer Einschränkung der maximal möglichen Tupelzahlen, da zu wenig Arbeitsspeicher für VA-File und die Warteschlange vorhanden ist. Der Speicher, den das Feld für die Bereichssuche belegt, wird zur Bearbeitung der Nächsten-Nachbarsuche freigegeben, um möglichst viele Datensätze speichern zu können.

Die maximale Anzahl von Datensätzen verringert sich bei der Variante des VA-Files mit der Speicherung im EEPROM, da weniger Datensätze in den EEPROM passen. Die Größe des VA-Files richtet sich ebenfalls nach dem Wert von b, der Anzahl der Datensätze und der Dimension. Zusätzlicher Arbeitsspeicher wird nur bei der Variante mit NOA für die sortierte Warteschlange benötigt. Der Speicher für das Feld der Bereichssuche wird für die Nächste-Nachbarsuche freigegeben. Durch die Speicherung des VA-Files im EEPROM ist eine Einschränkung der Anzahl an Datensätzen für den NOA nicht notwendig.

Der LSD-Baum mit einem Bucket für alle Datensätze belegt zusätzlichen Arbeitsspeicher durch das Bucket, das alle Adressen der enthaltenen Datensätze speichert. Das sind $2 \cdot N + 3$ Byte. Weitere Knoten existieren in dieser Variante nicht. Der zusätzliche Speicher für den Stack der Bereichssuche beträgt 2 Byte und für die Warteschlange der Nächsten-Nachbarsuche 7 Byte. Dieser Speicherbedarf entsteht dadurch, dass die Algorithmen nicht für den Extremfall angepasst sind. Ein Einfluss auf die maximale Anzahl von Datensätzen entsteht nicht. Somit können genauso viele Datensätze gespeichert werden, wie ohne Indexstruktur.

Der LSD-1 besteht aus genauso vielen Buckets wie gespeicherten Datensätzen. Für jedes Bucket werden 5 Byte gespeichert. Dies beinhaltet den Knotentyp, die Adresse des Datensatzes sowie die Füllung des Buckets. Die Füllung wird ab einer Kapazität von zwei Datensätzen benötig, bleibt jedoch bei dem Extremfall von einem Datensatz pro Bucket erhalten. Jeder der $(N - 1)$ inneren Knoten benötigt je 8 Byte und enthält die Verweise

Bei der Berechnung der Funktion I(k,r) wurde ein spezieller Fall nicht richtig behandelt. Der korregierte Code benötigt zusätzlichen Arbeitsspeicher. Daher sinkt die Zahl der möglichen Tupel f ̈ur den LSD-1-NNS Baum. Ein Einfluss auf die Analyse der Ausf ̈uhrungszeit kann ausgeschlossen werden, da der nicht richtig behandelte Fall bei der Analyse nicht auftritt.

auf die Kinder, den Typ des Knoten, die Splitposition und die Splitdimension. Damit benötigt der LSD-1-Baum $5 \cdot N + 8 \cdot (N - 1)$ Byte Arbeitsspeicher für die Indexstruktur. Der EEPROM enthält ausschließlich die Datensätze. Der hohe Bedarf an Arbeitsspeicher schränkt die maximale Anzahl an Datensätzen stark ein. Bei der Bereichssuche werden zusätzlich $2 \cdot (N + N - 1)$ Byte Arbeitsspeicher für den Stack benötigt, was eine weitere Einschränkung der maximalen Anzahl an Datensätzen zur Folge hat. Die Warteschlange der Nächsten-Nachbarsuche belegt $7 \cdot (N + N - 1)$ Byte Arbeitsspeicher. Dadurch verringert sich der Wert für die maximale Anzahl noch einmal. Jedoch ist zu bemerken, dass die maximale Anzahl von Datensätzen unabhängig von der Anzahl der Dimensionen und der Größe der Datensätze ist, solange der EEPROM ausreicht. Beim VA-File steigt die Größe der Indexstruktur mit der Anzahl der Dimensionen. Der LSD-Baum bleibt bei gleichbleibender Anzahl an Datensätzen gleich groß.

Programmspeicherbedarf

Das verwendete Testsystem verfügt mit 128 KB über genügend Programmspeicher. Doch tief eingebettete Systeme können auch weniger Programmspeicher besitzen, sodass der Programmspeicherbedarf ein wichtiges Kriterium für den Einsatz wird. Die Tabelle 6.1 zeigt eine Übersicht über den Programmspeicherbedarf der untersuchten Indexstrukturen. Die Übersicht beschränkt sich dabei auf die wichtigsten Ausprägungen. Die Übersetzung erfolgte mit der Optimierungsstufe `-O2`.

Index	nur exact	nur range	nur nns (SSA/ NOA)	alle (SSA/ NOA)
ohne Index	2274	4126	2634	6354
VA-3-RAM	3688	4348	3630/ 5566	8920/ 11052
VA-7-RAM	3484	4262	3466/ 5222	8304/ 10410
VA-3-EEPROM	3178	4894	3502/ 5528	8810/ 10870
VA-7-EEPROM	2976	4684	3332/ 5890	8718/ 10792
LSD-Baum	3980	5766	7980	11846

korregierte Werte, Erklärung siehe Seite 68

Tabelle 6.1: Übersicht über den Programmspeicherbedarf der Indexstrukturen nach Anfrage-Features (in Bytes)

In der Übersicht wird ersichtlich, dass der LSD-Baum am meisten Programmspeicher benötigt und ohne Index am wenigsten Speicher notwendig ist. Beim VA-File benötigt der NOA gegenüber dem SSA immer mehr Speicher. Auffällig ist bei den Varianten des VA-Files, dass die einzelnen Varianten in den einzelnen Ausprägungen unterschiedliche Veränderungen der Werte aufweisen. So benötigt bei der Ausprägung nur mit der Exact-Match-Suche das VA-File mit $b = 3$ und der Speicherung im RAM am meisten Speicher, während bei der Ausprägung nur mit der Bereichssuche das VA-File mit $b = 3$ und der Speicherung im EEPROM am meisten Speicher benötigt. Allgemein ist jedoch festzustellen, dass die VA-Files mit $b = 3$ mehr Programmspeicher benötigen als mit $b = 7$. Einzige Ausnahme bildet das VA-File mit $b = 7$, Speicherung im EEPROM und dem NOA in der Ausprägung nur mit der Nächsten-Nachbarsuche. Dort ist der Programmspeicherbedarf höher als bei der Variante mit $b = 3$. Dies ist auf die Optimierung durch den Compiler zurückzuführen. Die Optimierung erklärt auch die wechselnden Unterschiede zwischen den Varianten mit der Speicherung im EEPROM und denen im RAM.

Index	exact	range	nns
ohne Index	(882-1112)	(2670-2964)	(1116-1410)
VA-3-RAM-SSA	(2150-2774)	(2516-2990)	(1798-2422)
VA-3-RAM-NOA	(2150-3012)	(2474-3018)	(3692-4554)
VA-7-RAM-SSA	(1916-2492)	(2346-2832)	(1550-2126)
VA-7-RAM-NOA	(1916-2722)	(2346-2924)	(3426-4232)
VA-3-EEPROM-SSA	(1670-1780)	(3386-3612)	(2104-2220)
VA-3-EEPROM-NOA	(1670-2312)	(3030-3644)	(4278-4662)
VA-7-EEPROM-SSA	(1576-1646)	(3284-3748)	(1994-2458)
VA-7-EEPROM-NOA	(1552-1670)	(3232-3350)	(4438-4532)
LSD-Baum	(970-1112)	(2748-2794)	(4994-5192)

korregierte Werte, Erklärung siehe Seite 68

Tabelle 6.2: Intervalle (min-max) des Speicherbedarfs einzelner Anfragen im Bezug zum verwendeten Index (in Bytes)

Bei der Analyse der weiteren, nicht in der Übersicht aufgeführten Anfragekombinationen wurde festgestellt, dass sich der Speicherbedarf durch ein Anfrage-Feature in Kombination mit den weiteren Features ändert. Diese Wechselwirkungen der Features sind unter anderem durch den compilerinternen Optimierer zu erklären. Tabelle 6.2 veranschaulicht die Intervalle des Speicherbedarfs für die einzelnen Anfragen.

6.2.2 Ausführungszeit

Das Ziel bei der Nutzung von Indexstrukturen ist es, die Ausführung der Anfragen zu beschleunigen. In diesem Abschnitt werden die Ergebnisse der Analyse der Indexstrukturen vorgestellt. Die Ergebnisse sind nach den Anfragen gegliedert. Im folgenden Abschnitt werden die Ergebnisse für die Exact-Match-Suche vorgestellt. Danach folgen die Ergebnisse für die Bereichssuche und dann für die Nächsten-Nachbarsuche. Die Analyse der Ausführungszeiten erfolgt auf gleichverteilten Daten unter Angabe der Sekunden, berechnet aus den Timerticks.

Exact-Match-Suche

Die Ergebnisse der Analyse der Exact-Match-Suche für zwei-, drei- und vierdimensionale Daten sind in Abbildung 6.2 dargestellt. Dabei wurde die Zeit gemessen, die benötigt wird, um nach jedem Datensatz in der Datenbank einmal zu suchen. Dies geschieht, um einen exakten Wert zu erhalten, da die Position eines Datensatzes im Speicher Vor- und Nachteile haben kann. Das heißt, es werden bei n Datensätzen in der Datenbank n Exact-Match-Suchen hintereinander ausgeführt. Es erfolgt eine gleichverteilte Abfrage der Daten.

Die Abbildung 6.2 zeigt das, dass die Variante des VA-Files mit $b = 3$ und der Speicherung im EEPROM für die Exact-Match-Suche ungeeignet ist. Mögliche Gründe dafür sind eine zu geringe Datenraumaufteilung, der Rechenaufwand zur Extraktion der 4 Bit großen Approximation einer Dimension aus einem Byte und die Zugriffskosten für den Zugriff auf das VA-File. Durch die geringe Datenraumaufteilung und den zusätzlichen Rechenaufwand sind auch die Ergebnisse der Analyse mit zweidimensionalen Daten für

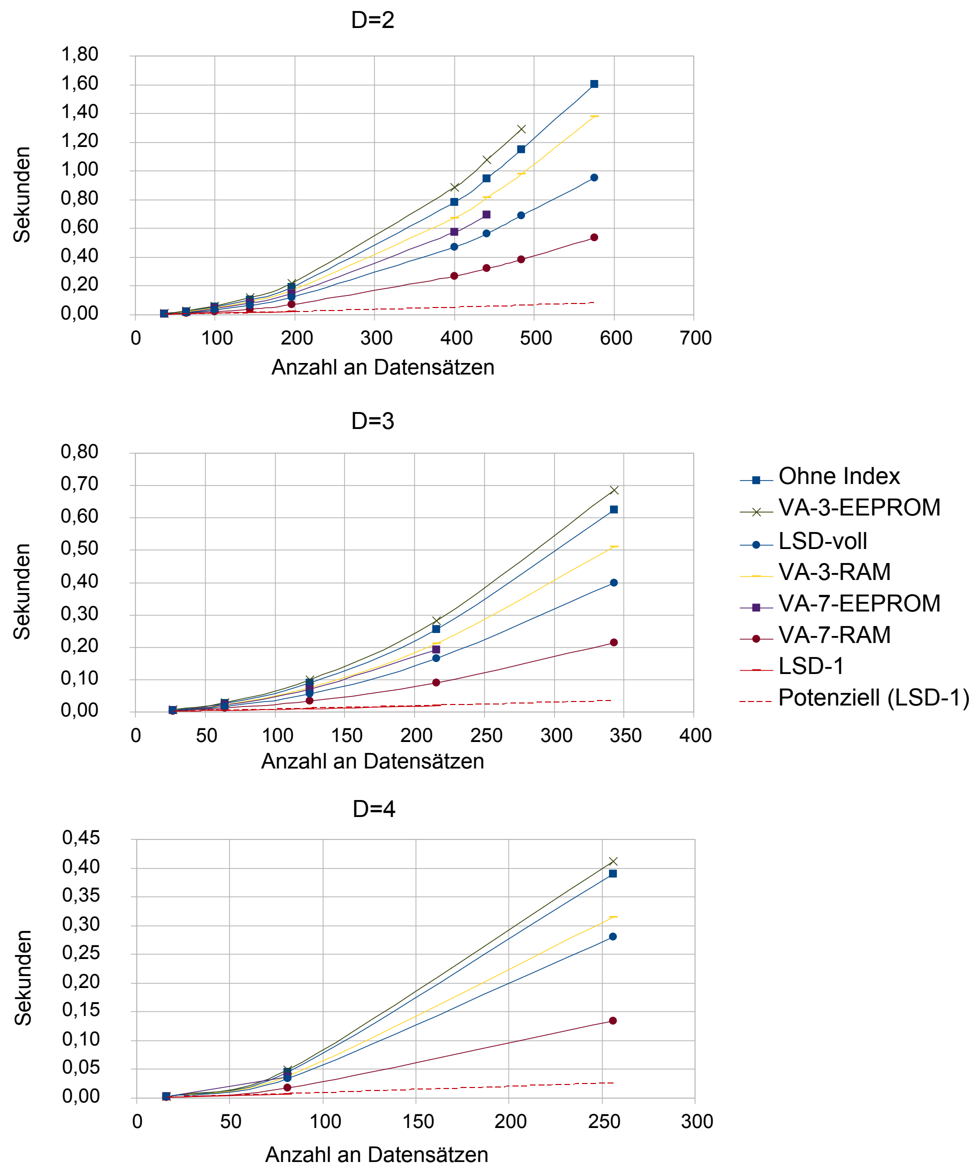


Abbildung 6.2: Ergebnisse der Analyse der Exact-Match-Suche

die Variante mit der Speicherung im RAM zu erklären. Das Argument der zu geringen Datenraumaufteilung ist jedoch erst ab einer Anzahl von über 3^D Datensätzen gültig, da darunter die Datenraumaufteilung vollkommen ausreichend ist und etwa ein Datensatz in einem Hyperquader liegt. Bis zu dieser Anzahl sind die Unterschiede allein durch den zusätzlichen Rechenaufwand für das Auslesen der Approximation zu begründen. Dieses Phänomen ist in den Ergebnissen für die Analyse dreidimensionaler und vierdimensionaler Daten besonders zu sehen, da dort die Anzahl nicht überschritten wird. Der Unterschied zwischen den beiden VA-Files mit $b = 7$ ist mit dem langsameren Zugriff auf das VA-File im EEPROM zu erklären.

Beim LSD-voll-Baum ist auffällig, dass die Suche schneller ist als bei der Suche ohne Index, obwohl alle Datensätze sequentiell durchsucht werden müssen. Dieses Phänomen ist durch die Speicherung der Adresse jedes Datensatzes im Bucket zu erklären und durch die nicht benötigte Prüfung nach der Gültigkeit des Datensatzes. Damit müssen nicht für jeden Datensatz die Größe des Tupels und die TID ausgelesen werden. Bei einer großen

Anzahl an Datensätzen summiert sich dieser kleine Zeitgewinn auf.

Der LSD-1-Baum weist im Vergleich zu den anderen analysierten Indexstrukturen die größte Verbesserung auf. Dies ist durch die starke Aufteilung des Datenraums zu erklären. Bei der Exact-Match-Suche muss immer nur auf einen Datensatz im EEPROM zugegriffen werden. Die Hauptoperation besteht im Folgen des Pfades im Baum.

Die Ergebnisse der Analyse der vierdimensionalen Daten lassen für den LSD-1-Baum und das VA-7-EEPROM-File keine konkreten Aussagen zu, da zu wenig verschiedene Datensatzanzahlen analysiert werden können. Eine Verbesserung der Anfragezeiten ist jedoch zu erkennen. Um weitere Aussagen treffen zu können, ist eine erneute Analyse auf einem größeren System notwendig.

Unabhängig von der untersuchten Dimensionsanzahl ist festzustellen, dass bei einer geringen Anzahl von Datensätzen die Ausführungszeiten sich aneinander annähern. Unterschiede zwischen den Anfragen mit Unterstützung durch eine Indexstruktur und der Anfrage ohne Indexunterstützung sind gering.

Bereichssuche

Die Analyse der Bereichssuche wurde mit einem Bereich, der 2^D Datensätzen umfasst und in der Mitte des Datenraums liegt, durchgeführt. Die Zeit zur Ausführung immer genau einer Anfrage wurde gemessen. Die Ergebnisse für die Analyse mit zwei-, drei- und vierdimensionalen Daten sind in Abbildung 6.3 dargestellt. Die Analyse mit vierdimensionalen Daten konnte nur für zwei verschiedene Anzahlen von Datensätzen durchgeführt werden. Für den LSD-1-Baum und das VA-7-EEPROM-File war nur eine Zeitmessung möglich. Daher sind nur grobe Aussagen für die Analyse mit vierdimensionalen Daten möglich. Trends müssen aus den Analysen mit zwei- und dreidimensionalen Daten abgeleitet werden.

Die Abbildung 6.3 zeigt, dass alle Indexstrukturen zu einer Verbesserung der Anfragezeit führen. Beim LSD-voll ist dies wiederum durch die geringere Zahl an Zugriffen auf den EEPROM zu erklären, da die TID und die Tupelgröße nicht für jeden Datensatz ausgelesen werden müssen.

Der LSD-1-Baum führt aufgrund seiner großen Datenraumaufteilung zur größten Verbesserung. Danach folgt direkt das VA-7-RAM-File. Der Unterschied zum VA-3-RAM-File ist durch den höheren Rechenaufwand zur Extraktion der 4-Bit großen Approximation einer Dimension aus einem Byte und ab einer Anzahl von 3^D Datensätzen mit der geringeren Datenraumaufteilung zu begründen. Die Unterschiede beim VA-3-EEPROM-File und VA-7-EEPROM-File heben sich bei geringeren Anzahlen an Datensätzen nahezu auf, durch die größeren Zugriffskosten auf das VA-File. Die Ergebnisse der Analyse der zweidimensionalen Daten zeigen, dass sich dieses Verhalten bei größeren Datensatzanzahlen zugunsten des VA-7-EEPROM-Files ändert.

Die Analyse der vierdimensionalen Daten zeigt, dass sich der LSD-voll-Baum bei steigender Anzahl an Tupeln gegen die VA-3-Files durchsetzt. Bei den Analysen mit zwei- und dreidimensionalen Daten ist dieses Verhalten bereits in den ersten Ergebnissen festzustellen.

Allgemein ist festzustellen, dass Verbesserungen der Anfragezeit für die Bereichssuche vorhanden sind. Bei einer sinkenden Anzahl an Datensätzen werden diese Verbesserungen jedoch immer geringer. Zudem sind die Verbesserungen allgemein für die untersuchten

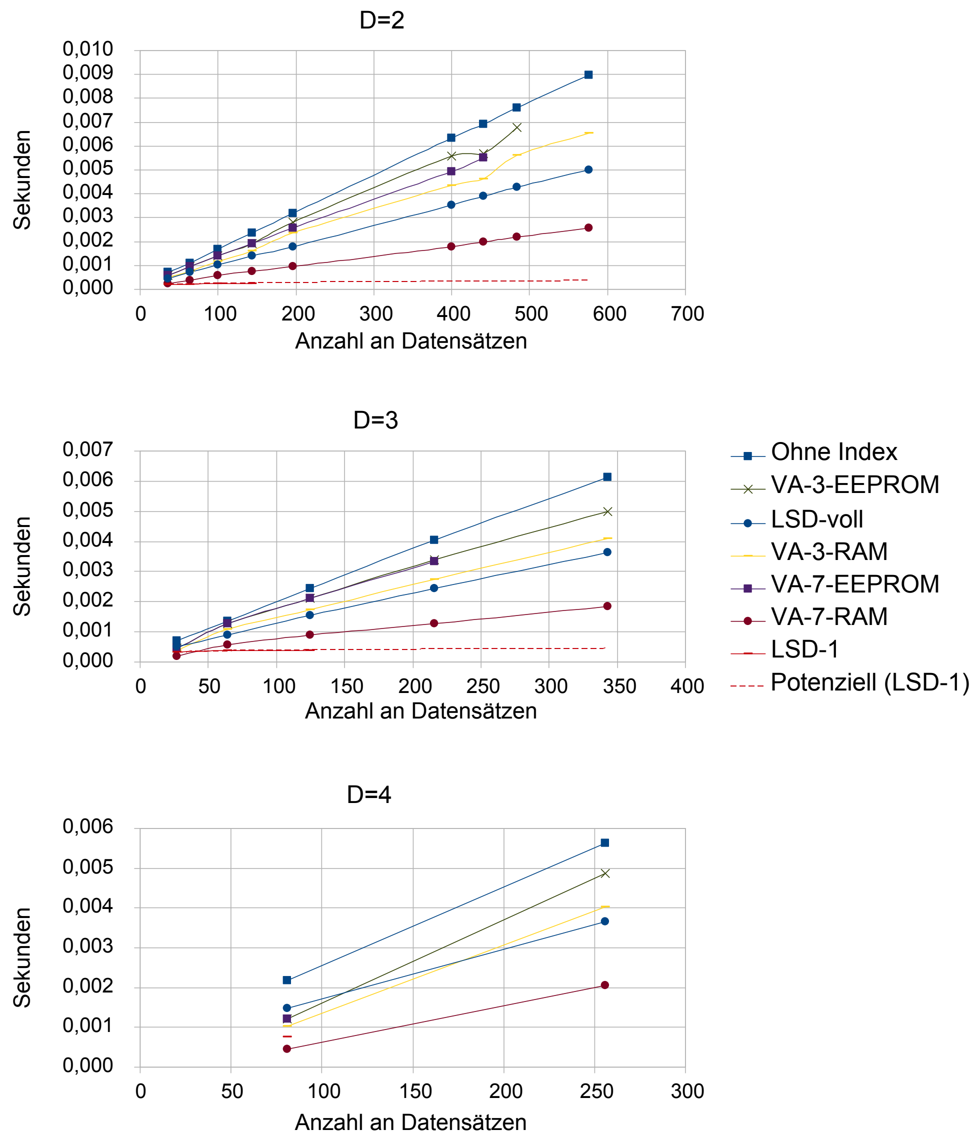


Abbildung 6.3: Ergebnisse der Analyse der Bereichssuche

Anzahlen als gering einzuschätzen, da die Bearbeitungszeit einer Anfrage schon ohne Indexunterstützung gering ausfällt.

Nächste-Nachbarsuche

Die Ergebnisse der Untersuchung der Indexstrukturen bezüglich der Verbesserungen der Nächsten-Nachbarsuche sind in Abbildung 6.4 dargestellt. Die Untersuchung erfolgte durch Messen der Zeit, die benötigt wird ,um zu jedem n Datensatz den nächsten Nachbarn, also den Datensatz selbst zu finden. Damit wurden n Nächste-Nachbarsuchen hintereinander ausgeführt und die Zeit, die dafür benötigt wurde, gemessen.

Abbildung 6.4 zeigt, dass alle Indexstrukturen bis auf den LSD-voll-Baum zu einer Verbesserung der Anfragezeiten führen. Die Vorteile des LSD-voll-Baums gegenüber der Suche ohne Indexunterstützung unterliegen dem zusätzlichen Aufwand des LSD-Baums bei der Nächsten-Nachbarsuche durch die sortierte Warteschlange. Die Warteschlange

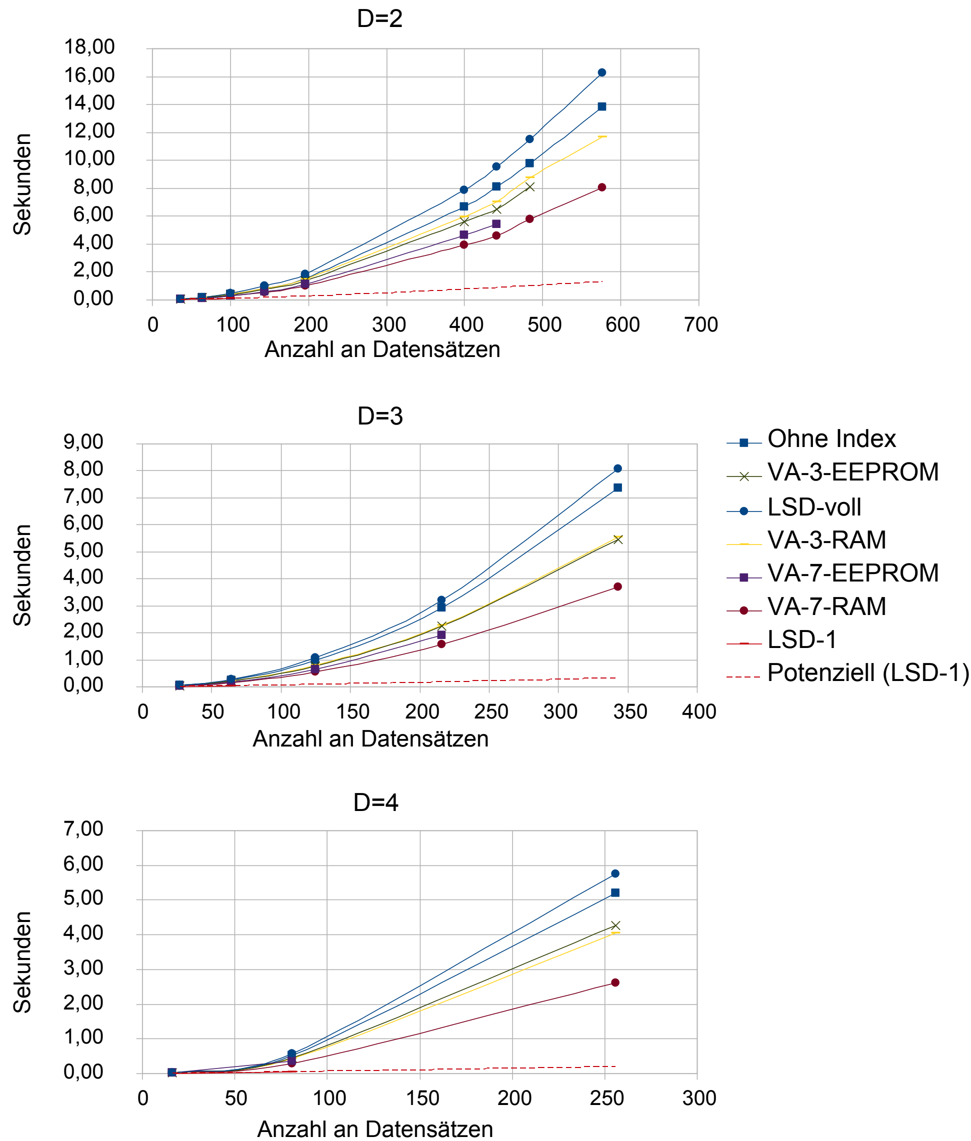


Abbildung 6.4: Ergebnisse der Analyse der Nächsten-Nachbarsuche mit SSA für VA-Files

führt zu zusätzlichem Aufwand. Dieser wird nicht durch die Einsparungen beim Zugriff auf die Datensätze ausgeglichen. Daher sind die Ausführungszeiten beim LSD-voll-Baum größer als ohne Indexstruktur.

Der LSD-1-Baum führt zur größten Verbesserung der Anfragezeit. Dies ist durch die Datenraumaufteilung zu begründen. Zusätzlich entsteht bei der gewählten Anfrage die Situation, dass der nächste Nachbar genau im ersten untersuchten Bucket liegt und somit keine weiteren Buckets untersucht werden müssen. Doch auch das Untersuchen mehrerer Buckets zur Bestimmung des nächsten Nachbarn führt zu nur gering höheren Anfragekosten, da immer nur wenige Buckets und damit wenige Datensätze genauer untersucht werden müssen. Dies ist in der Datenraumaufteilung begründet.

In der Abbildung 6.4 sind für das VA-File nur die Ergebnisse für die Nächste-Nachbarsuche mit dem SSA angegeben. Abbildung 6.5 zeigt den Vergleich der Anfragezeiten für die Nächste-Nachbarsuche mit den beiden Algorithmen für zweidimensionale Daten. Es ist deutlich erkennbar, dass die Suche mit dem NOA länger dauert als

mit dem SSA, obwohl der NOA eine Weiterentwicklung des SSA darstellt. Bei drei- und vierdimensionalen Daten sehen die Ergebnisse ähnlich aus. Dieses Verhalten lässt sich durch den großen Aufwand, der durch die sortierte Warteschlange entsteht und den im Gegensatz zu klassischen Datenbanksystemen geringen Zugriffskosten auf den Hintergrundspeicher, erklären. Der NOA wurde für klassische Datenbanksysteme entwickelt und stellt sich für das Testsystem als nicht geeignet heraus.

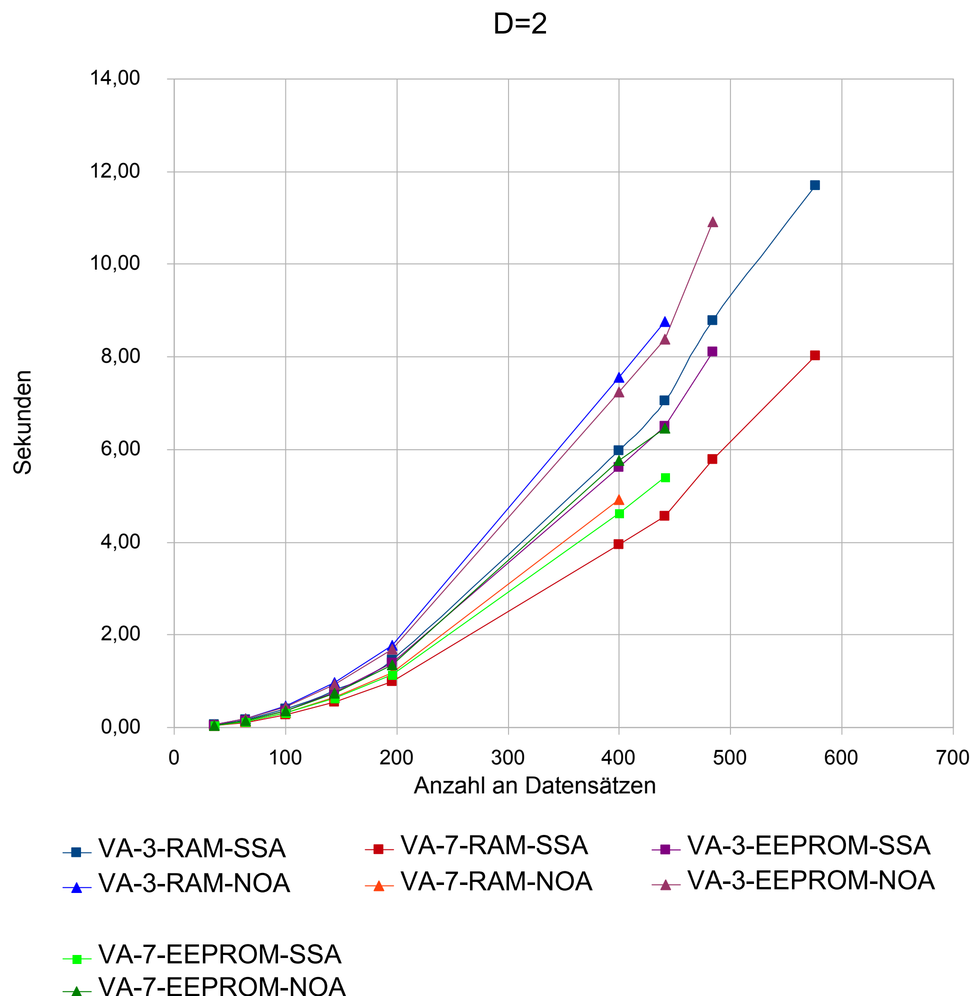


Abbildung 6.5: Vergleich der Anfragezeiten für NOA und SSA für zweidimensionale Daten

Die beiden Varianten des VA-7-Files mit dem SSA führen nach dem LSD-1-Baum zu den größten Verbesserungen. Dabei ist die Variante mit der Speicherung im RAM besser als die mit der Speicherung im EEPROM, da der Zugriff auf den EEPROM länger dauert als der Zugriff auf den RAM. Im Gegensatz zu den Varianten mit $b = 3$ ist kein zusätzlicher Rechenaufwand beim Auslesen der Approximation notwendig. Bei einer Anzahl von über 3^D Datensätzen führt zudem die größere Datenraumaufteilung zu bessern Ausführungszeiten. Dies ist zum Beispiel bei den Ergebnissen für zweidimensionale Daten sichtbar.

Bei den Varianten des VA-3-Files fällt auf, dass die Ausführungszeiten der Nächsten-Nachbarsuche beider Speicherorte annähernd gleich sind. Dieses Verhalten ist unabhängig bei beiden Algorithmen zu beobachten. Ein möglicher Grund dafür ist, dass

die wenigen Zugriffe auf das VA-File nur geringen Einfluss auf die Ausführungszeit besitzen und dieser Einfluss durch die compilerseitige Optimierung noch verringert wird. Die Anzahl der Zugriffe auf das VA-File sinkt im Vergleich zu den Varianten mit $b = 7$ aufgrund des Puffers, der das letzte gelesene Byte des VA-Files zwischenspeichert. Der Rechenaufwand zur Extraktion der Approximation bleibt jedoch hoch, da jede 4-bit Approximation einer Dimension einzeln extrahiert werden muss. Dies erklärt die größeren Ausführungszeiten gegenüber den VA-7-Files.

Allgemein ist festzustellen, dass Verbesserungen der Anfragezeiten durch die Indexstrukturen erreicht werden, doch fallen diese Verbesserungen insgesamt für eine einzelne Anfrage gering aus. Bei sinkender Anzahl an Datensätzen ist zudem eine Annäherung der Anfragezeiten an die Anfragezeiten der Suche ohne Indexunterstützung zu beobachten.

6.3 Diskussion

Die Analyse der einzelnen mehrdimensionalen Anfragen zeigt, dass Verbesserungen der Anfragezeit durch die implementierten, mehrdimensionalen Indexstrukturen auch für wenige Datensätze möglich sind. Die Verbesserungen der Anfragezeiten gehen zu Kosten von Speicherplatz. Die Analyse zeigt jedoch auch, dass besonders im Bereich weniger Datensätze die Anfragezeiten auch ohne Indexunterstützung gering sind. Die Zeitersparnisse sind daher insgesamt gering. Aus diesem Grund ist eine Prüfung, ob kürzere Anfragezeiten wirklich notwendig sind, für jede mögliche Anwendung notwendig. Der Grad der Verbesserung durch die implementierten Indexstrukturen sollte individuell für die Anwendungsdaten bestimmt werden, da die Analyse nur für gleichverteilte Daten erfolgt ist.

Die Analyse der implementierten Varianten des VA-Files zeigte die unterschiedlich gute Eignung der Varianten. Besonders zwischen den Varianten mit $b = 3$ und $b = 7$ wird dies deutlich. Die Varianten mit $b = 3$ sind aufgrund des zusätzlichen Rechenaufwandes weniger geeignet. Aus dem gleichen Grund ist auch der NOA nicht passend. Das Phänomen, das der Rechenaufwand ähnlich wichtig für den gesamten Aufwand der Anfrage ist wie die Zahl der Zugriffe auf den Hintergrundspeicher, ist eine Besonderheit, die durch die Prozessorleistung und die verwendeten Speichertechnologien begründet ist. In klassischen Rechnersystemen kann der Rechenaufwand in der Regel vernachlässigt werden [SHS05]. Dies zeigt, dass Indexstrukturen, die in klassischen Rechnersystemen verwendet werden, nicht ohne Prüfung und gegebenenfalls Anpassung auf tief eingebettete Systeme übertragen werden können.

Der LSD-1-Baum erwies sich in der Analyse der Anfragezeiten als die schnellste Indexstruktur. Doch benötigt der LSD-1-Baum auch am meisten Speicher, sowohl Datenspeicher als auch Programmspeicher. Somit konnten nur wenige Datensätze in die Datenbank eingefügt werden. Diese Zahl ist jedoch unabhängig von der Anzahl an Dimensionen der Daten. Dennoch eignet sich der LSD-1-Baum aufgrund des Speicherbedarfs nur bedingt für die Verwendung in tief eingebetteten Systemen.

Die Analyse des LSD-voll-Baums zeigt, dass der sequentielle Durchlauf durch alle Datensätze mit einer Liste der EEPROM-Adressen aller gültigen Datensätze beschleunigt werden kann.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Nutzung mehrdimensionaler Indexstrukturen in einem Datenbankmanagementsystem für ressourcenbeschränkte, tief eingebettete Systeme zu untersuchen. Nachdem im Kapitel 2 die Grundlagen zu eingebetteten Systemen und der Datenhaltung in diesen Systemen gelegt wurden, beschäftigte sich das Kapitel 3 mit Beispielanwendungen, in denen mehrdimensionale Daten gespeichert und gelesen werden müssen. Dabei wurden intelligente, eingebettete Systeme und eingebettete Systeme im Auto näher beleuchtet. Im Ergebnis konnte festgestellt werden, dass Anwendungen für tief eingebettete Systeme bestehen, die effizient mit mehrdimensionalen Daten umgehen müssen. Mit Hilfe dieser Anwendungen wurde dann eine Anforderungsanalyse, welche die Anforderungen an eine mehrdimensionale Indexstruktur festlegt, durchgeführt.

Das Kapitel 4 beschäftigte sich mit den mehrdimensionalen Indexstrukturen allgemein. Es wurde auf die Einteilung sowie auf die wichtigsten mehrdimensionalen Anfragearten eingegangen. Danach folgte eine Vorstellung verschiedener Vertreter mehrdimensionaler Indexstrukturen, von denen im letzten Abschnitt des Kapitels das VA-File und der LSD-Baum zur Implementierung und Analyse ausgewählt wurden. Im Kapitel 5 wurde die Implementierung der mehrdimensionalen Anfragen und der beiden Indexstrukturen thematisiert. Dabei wurde auf die verschiedenen Varianten des VA-Files und die Anpassung des LSD-Baums an die Speichertechnologie eingegangen. Die Analyse der implementierten Indexstrukturen erfolgte in Kapitel 6. Verglichen wurden die Speicherkosten und die Zeitgewinne der implementierten Indexstrukturen untereinander und gegenüber einer Implementierung der Anfragen ohne Indexunterstützung. Zur Bestimmung der Speicherkosten wurden jeweils die maximale Anzahl an Datensätzen und der Bedarf an Programmspeicher ermittelt. Das Ergebnis der Analyse ist, dass die Indexstrukturen auch bei wenigen Datensätzen Verbesserungen der Ausführungszeit bringen. Dies ist jedoch nur zulasten des Speichers möglich. Aufgrund der geringen Anzahlen an Datensätzen sind die Ausführungszeiten für eine Anfrage auch ohne Index gering. Dies führt dazu, dass Verbesserungen nur im kleinen Maßstab möglich sind. Daher und aufgrund der Speicherkosten sind für jede konkrete Anwendung der Nutzen und die Kosten abzuwiegen.

Diese Arbeit konnte zeigen, dass sowohl Anwendungen mehrdimensionaler Indexstrukturen in tief eingebetteten Systemen möglich sind, als auch, dass mehrdimensionale Indexstrukturen bei relativ geringen Anzahlen an Datensätzen und bei Ressourcenbeschränkung Verbesserungen der Anfragezeiten erreichen. Das VA-File und der LSD-Baum

wurden als mehrdimensionale Indexstrukturen für ein DBMS für tief eingebettete Systeme erfolgreich implementiert. Um einen vollständigen Blick auf die Nutzung mehrdimensionaler Indexstrukturen in tief eingebetteten Systemen zu erlangen, sind jedoch weitere Analysen mit weiteren mehrdimensionalen Indexstrukturen und konkreten Realdaten notwendig.

Literaturverzeichnis

[ALRS05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *4th International Conference of Generative Programming and Component Engineering, 29 September - 1 October 2005, Tallinn, Estonia*, S. 125–140. Springer Verlag, 2005.

[Ape07] Apel, S.: *The Role of Features and Aspects in Software Development.* Dissertation, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Germany, 2007.

[Bal04] Balko, S.: *Grundlagen, Entwicklung und Evaluierung einer effizienten Approximationstechnik für Nearest-Neighbor-Anfragen im hochdimensionalen Vektorraum.* Dissertation, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Germany, March 2004.

[BIT10] BITKOM: Eingebettete Systeme - Ein stategisches Wachstumsfeld für Deutschland: Anwendungsbeispiele, Zahlen und Trends, 2010. published online http://www.bitkom.org/de/publikationen/38338_62539.aspx.

[Bre07] Breunig, M.: Geodatenbankforschung: Rückblick und Perspektiven aus Sicht der Informatik. *Datenbank-Spektrum*, Band 7, Nr. 21, S. 5–14, 2007.

[BS07] Braess, H.-H.; Seiffert, U. H.: *Vieweg Handbuch Kraftfahrzeugtechnik.* Vieweg + Teubener Verlag, 2007.

[CE99] Czarnecki, K.; Eisenecker, U. W.: Components and generative programming. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1999, Toulouse, France*, S. 2–19. Springer Verlag, 1999.

[CE00] Czarnecki, K.; Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Verlag, 2000.

[DG98] Decker, H.; Gehlen, H.-P.: *Zündsysteme. Motorsteuerung für Ottomotoren.* Robert Bosch Verlag, 1998.

[FH08] Fischer, P.; Hofer, P.: *Lexikon der Informatik.* Springer Verlag, 2008.

[Gri05] Grimm, K.: Software-Technologie im Automobil. In *Software-Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen*, S. 407–430. Spektrum Akademischer Verlag, 2005.

[Hen94] Henrich, A.: A Distance Scan Algorithm for Spatial Access Structures. In *ACM-GIS*, S. 136–143. ACM, 1994.

[Hen98] Henrich, A.: The LSD^{h}-Tree: An Access Structure for Feature Vectors. In *14th International Conference on Data Engineering, 23-27 February 1998, Orlando, Florida, USA*, S. 362–369. IEEE Computer Society, 1998.

[Hor05] Horn, E.: Software- und Systemarchitektur. In *Software-Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen*, S. 179–203. Spektrum Akademischer Verlag, 2005.

[HSW89] Henrich, A.; Six, H.-W.; Widmayer, P.: The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *15th International Conference on Very Large Data Bases, 22-25 August 1989, Amsterdam, The Netherlands*, S. 45–53. Morgan Kaufmann, 1989.

[KBL+06] Kim, G.-J.; Baek, S.-C.; Lee, H.-S.; Lee, H.-D.; Joe, M. J.: LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *32nd International Conference on Very Large Data Bases, 12-15 September 2006, Seoul, Korea*, S. 1255–1258. ACM, 2006.

[KCH+90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technischer Bericht Nr. CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, November 1990.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming, 9-13 June 1997, Jyväskylä, Finland*, S. 220–242. Springer Verlag, 1997.

[Lie08] Liebig, J.: Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme. Master's thesis, Otto-von-Guericke-Universität Magdeburg, 28.11.2008.

[Lin06] Lin, J.: Zuverlässiger als das menschliche Auge. *Der Konstrukteur*, Band 2006, Nr. 05, 2006.

[LKGH03] Li, X.; Kim, Y.-J.; Govindan, R.; Hong, W.: Multi-dimensional range queries in sensor networks. In *1st International Conference on Embedded Networked Sensor Systems, 5-7 November 2003, Los Angeles, California, USA*, S. 63–75. ACM, 2003.

[Mar07] Marwedel, P.: *Eingebettete Systeme*. Springer Verlag, 2007.

[Nau05] Nauth, P.: *Embedded Intelligent Systems*. Oldenbourg Verlag, 2005.

[NK07] Nath, S.; Kansal, A.: FlashDB: dynamic self-tuning database for NAND flash. In *6th International Conference on Information Processing in Sensor Networks, 25-27 April 2007, Cambridge, Massachusetts, USA*, S. 410–419. ACM, 2007.

[Nor02] Northrop, L. M.: SEI's Software Product Line Tenets. *IEEE Software*, Band 19, Nr. 4, S. 32–40, 2002.

[NTN+02] Nyström, D.; Tesanovic, A.; Norström, C.; Hansson, J.; Bånkestad, N.-E.: Data Management Issues in Vehicle Control Systems: A Case Study. In *14th Euromicro Conference on Real-Time Systems, 19-21 June 2002, Vienna, Austria*, S. 249–256. IEEE Computer Society, 2002.

[NTNH03] Nyström, D.; Tesanovic, A.; Norström, C.; Hansson, J.: The COMET Database Management System. Technical Report Nr. ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE, Mälardalen University, April 2003.

[Nys03] Nyström, D.: COMET: A Component-Based Real-Time Database for Vehicle Control-Systems. Licentiate Thesis, Mai 2003.

[Nys05] Nyström, D.: *Data Management in Vehicle Control-Systems.* Dissertation, Mälardalen University, Department of Computer Science and Electronics, Västerås, Sweden, 2005.

[Pag96] Pagel, B.-U.: *Analyse und Optimierung von Indexstrukturen in Geo-Datenbanksystemen.* Infix Verlag, 1996.

[PR05] Polze, A.; Rasche, A.: Programmierung eingebetteter Software. In *Software-Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen*, S. 179–203. Spektrum Akademischer Verlag, 2005.

[Rei09] Reif, K.: *Automobilelektronik.* Vieweg + Teubner Verlag, 2009.

[Röh10] Röhrig, C.: Drahtlose Sensornetzwerke mit Lokalisierungsfunktion für Anwendungen im betreuten Wohnen. In *Tagungsband – Ambient Assisted Living – 3. Deutscher AAL-Kongress, 26-27 January 2010, Berlin, Germany.* VDE-Verlag, 2010.

[RLAS07] Rosenmüller, M.; Leich, T.; Apel, S.; Saake, G.: Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen. *Datenbank-Spektrum*, Band 7, Nr. 20, S. 33–47, 2007.

[RSS+08] Rosenmüller, M.; Siegmund, N.; Schirmeier, H.; Sincero, J.; Apel, S.; Leich, T.; Spinczyk, O.; Saake, G.: FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management, 29 March 2008, Nantes, France*, S. 1–6. University of Magdeburg, 2008.

[Sch05a] Schmitt, I.: *Ähnlichkeitssuche in Multimedia-Datenbanken - Retrieval, Suchalgorithmen und Anfragebehandlung.* Oldenbourg Verlag, 2005.

[Sch05b] Scholz, P.: *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Springer Verlag, 2005.

[Sch06] Schmitt, G.: *Mikrocomputertechnik mit Controllern der Atmel-AVR-RISC-Familie: Programmierung in Assembler und C-Schaltungen und Anwendungen*. Oldenbourg Verlag, 2006.

[SHS05] Saake, G.; Heuer, A.; Sattler, K.-U.: *Datenbanken: Implementierungstechniken*. mitp-Verlag, 2005.

[Spi02] Spinczyk, O.: *Aspektorientierung und Programmfamilien im Betriebssystembau*. Dissertation, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Germany, 2002.

[SSH08] Saake, G.; Sattler, K.-U.; Heuer, A.: *Datenbanken: Konzepte & Sprachen*. mitp-Verlag, 2008.

[SSPSS98] Schön, F.; Schröder-Preikschat, W.; Spinczyk, O.; Spinczyk, U.: Design Rationale of the PURE Object-Oriented Embedded Operation System. In *Distributed and Parallel Embedded Systems, IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems , 5-6 October 1998, Schloß Eringerfeld, Germany*, S. 231–234. Kluwer, 1998.

[SZ06] Schäuffele, J.; Zurawka, T.: *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Vieweg + Teubner Verlag, 2006.

[WB97] Weber, R.; Blott, S.: An Approximation-Based Data Structure for Similarity Search. In *ESPRIT Project HERMES, no. 9141, technical report number 24*, oct 1997.

[Wie07] Wiegelmann, J.: *Softwareentwicklung in C für Mikroprozessoren und Mikrocontroller*. Hüthig Verlag, 2007.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 15. Dezember 2010

Sara Kunze