Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Institut für Technische und Betriebliche

Informationssysteme

Diplomarbeit

Klassifizierung von Ansätzen für column oriented DBMS

Verfasser :
Carsten Wittwer
Matrikelnummer 91349

Betreuer :
Prof. Dr. rer. nat. habil.Gunter Saake,
Dipl.-Inf. Andreas Lübcke

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39106 Magdeburg

Danksagung

An dieser Stelle möchte ich mich bei Herrn Professor Dr. Saake und Herrn Andreas Lübcke von der Arbeitsgruppe Datenbanken am Institut für Technische und Betriebliche Informationssysteme der Otto-von-Guericke-Universität für die Betreuung von zunächst meiner Studienarbeit und nun meiner Diplomarbeit bedanken, insbesondere für die stets gute Unterstützung während der Erstellung von Studien- und Diplomarbeit.

Da die Diplomarbeit auch gleichzeitig der Abschluss meines Fernstudiums an der Otto-von-Guericke-Universität ist, möchte ich an dieser Stelle auch allen Professoren und weiteren Mitarbeitern der Universität danken, die durch ihr persönliches Engagement dieses Fernstudium ermöglicht haben, wozu unter anderem das Abhalten von Vorlesungen am Wochenende und das äußerst flexible Vereinbaren von Prüfungsterminen mit den (teilweise berufstätigen) Studenten gehörten. Insbesondere danken möchte ich an dieser Stelle Herrn Professor Paul, Herrn Professor Dumke sowie Frau Grosche vom Studentensekretariat, die meinen Kommilitonen und mir während des gesamten Fernstudiums stets mit Rat und Tat zur Seite standen.

Inhaltsverzeichnis

Danksagung.....I

Inhaltsverzeichnis.....III

Abbildungsverzeichnis.....IX

Tabellenverzeichnis.....XIII

Abkürzungsverzeichnis.....XVII

1 Einleitung.....1

1.1 Motivation.....1

1.2 Zielsetzung.....3

1.3 Gliederung der Diplomarbeit und Formatierungen.....3

2 Grundlagen und Unterschiede roDBMS / coDBMS.....5

2.1 9 Codd'sche Regeln für ein DBMS.....5

2.2 Architektur von DBMS.....6

2.3 Speicherhierarchie.....13

2.4 Verwaltung des Hintergrundspeichers.....16

2.4.1 Interne Speicherung von Relationen und Zugriffspaden.....17

2.4.2 Einpassung von Tupeln/Datensätzen in Blöcke gleicher Länge (roDBMS).....18

2.5 Seiten, Sätze und Adressierung.....19

2.5.1 (Daten-)Satztypen.....19

2.5.1.1 Sätze fester Länge.....20

2.5.1.2 Sätze variabler Länge.....21

2.6 Datenbankpuffer.....22

2.7 Adressierung von Datensätzen (Tupel-Identifier).....23

2.8 Grundlegendes Prinzip spaltenorientierter DBMS und Unterschiede zu roDBMS.....23

2.8.1 Unterschiede bei der Anordnung der Daten.....24

2.8.2 Unterschiede bei der Verwaltung des Hintergrundspeichers.....24

2.8.2.1 Interne Speicherung von Relationen und Zugriffspaden.....24

2.8.2.2 Einpassung von Tupeln/Datensätzen in Blöcke gleicher Länge.....25
2.8.3 Unterschiede zu roDBMS bei Sätzen fester/variabler Länge.....25
2.8.4 Effiziente Nutzung des Datenbankpuffers.....26
2.9 Existierende Implementierungen von coDBMS.....27
2.10 OLTP vs. OLAP.....28
2.10.1 OLTP.....28
2.10.2 OLAP.....28
2.10.3 Unterschiede im Überblick.....30
2.10.4 Konsequenzen für die Speicherarchitektur eines DBMS.....32
2.11 coDBMS vs. leseoptimierte roDBMS.....34
2.11.1 Zugriffsstrukturen.....34
2.11.2 Dateiorganisationsformen.....34
2.11.2.1 Heap-Organisation.....34
2.11.2.2 Sequenzielle Organisation.....35
2.11.2.3 Hash-Organisation.....35
2.11.3 Zugriffspfade.....35
2.11.3.1 B-Baum.....36
2.11.3.2 Hash-Index.....37
2.11.3.3 Bitmap-Index.....38
2.11.3.4 Verbundindex.....38
2.11.4 Weitere Ansätze zur Performancesteigerung.....39
2.11.4.1 Clustering.....39
2.11.4.2 Materialisierte Sichten.....39
2.11.5 Klassische Optimierungstechniken vs. Column-oriented Ansatz.....39
2.12 Kompressionstechniken „klassischer“ und spaltenorientierter DBMS.....40
2.12.1 Lauflängenkodierung.....41
2.12.2 Statistische Kompressionsverfahren (Huffman, Arithmetische Kodierung).....41
2.12.2.1 Huffman-Kodierung.....42
2.12.2.2 Arithmetische Kodierung.....42
2.12.3 Wörterbuchkompression (Lempel-Ziv).....43
2.12.3.1 Lempel-Ziv 77 (LZ77).....44
2.12.3.2 Lempel-Ziv 78 (LZ78).....45
2.12.4 Einsatz von Kompression in roDBMS und coDBMS.....45

2.12.4.1 Komprimierung in roDBMS am Beispiel von Oracle und IBM DB 2.....49
2.12.4.2 Komprimierung in coDBMS.....50
2.13 Verteilte und parallele Datenbanken.....52
2.13.1 Horizontale und vertikale Partitionierung.....56
2.14 Anfrageausführung und -optimierung (Query Processor).....58

3 Klassifizierung der Ansätze für coDBMS.....63
3.1 Vorstellung und Klassifizierung der Implementierungen.....65
3.1.1 C-Store und Vertica.....65
3.1.1.1 Logisches und physisches Datenmodell.....69
3.1.1.2 Leseoptimierter Speicher (RS).....72
3.1.1.3 Schreiboptimierter Speicher.....75
3.1.1.4 Speicherverwaltung.....76
3.1.1.5 Transaktionen.....76
3.1.1.6 Anfrageausführung.....77
3.1.1.7 Materialisierungsstrategien in C-Store.....81
3.1.1.8 Ansätze von C-Store/Vertica.....87
3.1.2 Infobright Data Warehouse (basierend auf MySQL).....88
3.1.2.1 Anfragebearbeitung.....91
3.1.2.2 Laden der Daten und Kompression.....94
3.1.2.3 Schnittstellen.....95
3.1.2.4 Systemarchitektur.....96
3.1.2.5 Ansätze von Infobright.....97
3.1.3 LucidDB.....98
3.1.3.1 Physische Speicherung.....99
3.1.3.2 Spaltenorientierte Speicherung.....100
3.1.3.3 Indizierung.....100
3.1.3.4 Datenkompression.....101
3.1.3.5 Page-level-Multiversioning.....102
3.1.3.6 Optimierung der Dateiorganisation.....102
3.1.3.7 Anfrageoptimierung und -ausführung.....103
3.1.3.8 Ansätze von LucidDB.....105
3.1.4 1010data's Tenbase.....106

3.1.5 Bigtable, Hbase, Hypertable und Cassandra Project....107
3.1.5.1 Allgemeines....109
3.1.5.2 Datenmodell....110
3.1.5.3 Physische Speicherung....112
3.1.5.4 Dateiformat und Indizierung....115
3.1.5.5 Bloomfilter....116
3.1.5.6 Anwendungsschnittstellen....117
3.1.5.7 Konkurrierende Zugriffe....118
3.1.5.8 Systemarchitektur....118
3.1.5.9 Google File System (GFS), Hadoop Distributed File System (HDFS) und Amazon Dynamo....120
3.1.5.10 Google und Hadoop MapReduce....121
3.1.5.11 Kompression....121
3.1.5.12 Beispiel-Anwendung für Bigtable, Hbase, Hypertable und Cassandra....122
3.1.5.13 Ansätze von Bigtable, Hbase, Hypertable und Cassandra....123
3.1.6 Valentina Database....124
3.1.7 Vectornova / Vectorstar High-Speed Data Engine....125
3.1.8 kdb(+)....126
3.1.8.1 Die Programmiersprache K (jetzt q)....128
3.1.8.2 Ansätze von kdb+....128
3.1.9 Skytide XOLAP-Server....129
3.2 Nicht berücksichtigte Implementierungen....130
3.2.1 S und GNU R Programmiersprache....130
3.2.2 EaseDB....131
3.2.3 FastBit....131
3.2.4 DataProbe....131
3.2.5 Weitere nicht betrachtete Implementierungen....132
3.3 Gesamtübersicht über die Klassifizierungen....132
4 Fazit und Ausblick....137
4.1 Ausblick....139
A Ergänzende Grundlagen....141

A.1 Seiten, Sätze, Adressierung....141
A.1.1 Datensatztypen....141
A.1.1.1 Pinned Records (fixierte Sätze)....141
A.1.1.2 Unpinned Records (unfixierte Sätze)....141
A.1.1.3 Lange Felder oder große unstrukturierte Sätze....141
A.2 Adressierung von Datensätzen mittels Tupel-Identifier (TID)....143
A.3 18 Codd'sche Regeln für Anforderungen an OLAP-Produkte....144
A.4 Klassifikation von Zugriffsstrukturen von Datensätzen....146
A.4.1 Primärschlüssel vs. Sekundärschlüssel....146
A.4.2 Primärindex vs. Sekundärindex....146
A.4.3 Dateiorganisation vs. Zugriffspfad....147
A.4.4 Dünnbesetzter vs. Dichtbesetzter Index....147
A.4.5 Geclusterte vs. Nicht-geclusterte Indexe....147
A.4.6 Schlüsselzugriff vs. Schlüsseltransformation....148
A.4.7 Ein-Attribut- vs. Mehr-Attribut-Index....148
A.4.8 Ein- vs. mehrdimensionale Zugriffsstruktur....149
A.4.9 Statische vs. Dynamische Zugriffsstruktur....149
A.5 Zugriffspfade....150
A.5.1 Hash-Index....150
A.5.2 Bitmap-Index....150
A.5.3 Verbund-Index....151
A.6 Kompressionstechniken in DBMS....152
A.6.1 Klassifizierung von Kompressionstechniken....152
A.6.2 Lauflängenkodierung....153
A.6.3 Huffman-Kodierung....153
A.6.4 Entropie....154
A.6.5 Arithmetische Kodierung....155
A.6.6 Wörterbuchkompression....161
A.6.6.1 Übersicht über existierende Lempel-Ziv-Implementierungen....161
A.6.6.2 Beispiel für Lempel-Ziv 77....161
A.6.6.3 Beispiel für Lempel-Ziv 78 (LZ78)....163
A.6.7 Datenkompression bei Oracle DBMS....164
A.6.8 Datenkompression in IBM DB2....171

A.6.9 Vergleich der Datenkompression in Oracle und DB2.....172
A.6.10 Vergleich von zeilen- und spaltenorientierter Kompression.....172
A.7 Verteilte und parallele Datenbanken.....176
A.7.1 Verteilte Datenbanken.....176
A.7.2 Horizontale und vertikale Partitionierung.....181

B Glossar.....183

Literaturverzeichnis.....187

Selbstständigkeitserklärung.....199

Abbildungsverzeichnis

Abbildung 1: Drei-Ebenen-Schema-Architektur (ANSI), vgl. [HS00].....7
Abbildung 2: ANSI-SPARC-Architektur eines DBS (gem. [HS00]).....8
Abbildung 3: Klassifikation von Komponenten eines DBMS (Seite 32 [HS00]).....9
Abbildung 4: Systemarchitektur eines DBMS (aus [SK08]).....9
Abbildung 5: Fünf-Schichten-Architektur der Transformationskomponenten des DBMS.....10
Abbildung 6: Speicherung einer Relation in physischer Datei ([SHS05]).....12
Abbildung 7: Festplatten-Zugriffszeiten (aus [TH07]).....15
Abbildung 8: Festplatten-Lese-Transferraten (aus [TH07]).....15
Abbildung 9: Nichtspannsatz und Spannsatz (Seite 100 [SHS05]).....19
Abbildung 10: Aufbau von Seiten (Abbildung 3.11 aus [SHS05]).....20
Abbildung 11: Sätze variabler Länge: Variante 1 (Seite 105 [SHS05]).....22
Abbildung 12: Sätze variabler Länge: Variante 2 (Seite 105 [SHS05]).....22
Abbildung 13: B-Baum der Ordnung 2 mit 3 Stufen.....37
Abbildung 14: LZ77-Schiebefenster (aus [LZAlg1]).....44
Abbildung 15: Topologie eines DDBMS (aus [CB05]).....54
Abbildung 16: Phasen der Anfragebearbeitung (Seite 390 [SHS05]).....58
Abbildung 17: Arbeitsweise des DB-Designers von Vertica (Seite 10 [VertWhitePaper]).....67
Abbildung 18: Architektur von C-Store und Vertica ([VertWhitePaper]).....69
Abbildung 19: Physische Speicherung einer Relation in Vertica (aus [VertWhitePaper]).....70
Abbildung 20: Join-Index für zwei Projektinen ([Mic05]).....72
Abbildung 21: Entscheidungsweg für Kompressionsverfahren in C-Store (Seite 65 [Aba08]) 75
Abbildung 22: Ein spaltenorientierter Anfrageplan (Seite 43 [Aba08]).....78
Abbildung 23: Positionsliste bei Verbundoperation [Aba06].....79
Abbildung 24: Anfragepläne für frühe Materialisierung [Aba08].....84
Abbildung 25: Anfragepläne für späte Materialisierung [Aba08].....85
Abbildung 26: Multicolumn-Datenstruktur in C-Store [Aba08].....87
Abbildung 27: Architekturebenen von Infobright ([IBWP08]).....89
Abbildung 28: Beispiel für Datenpakete (Data Nodes) bei einer Relation mit 3 Attributen ([IBWP08]).....89

Abbildung 29: Anlegen von DN und DPN beim Laden von Daten in das DW [IBWP08].....96
Abbildung 30: Einbindung von Infobright in MySQL-Gesamtarchitektur [IBWP08].....97
Abbildung 31: Systemarchitektur von LucidDB aus [LucidArch09].....99
Abbildung 32: B-Baum-Indizierung in LucidDB [LucStor].....101
Abbildung 33: B-Baum-Index und komprimierter Bitmap-Index [LucStor].....101
Abbildung 34: Page-level-Multiversioning in LucidDB [LucStor].....102
Abbildung 35: Tenbase Backend Service [1010Arch].....106
Abbildung 36: Tenbase-Architektur [TBFeat].....107
Abbildung 37: Schlüssel-Wert-Paare in Hypertable [HTArch].....114
Abbildung 38: Logische Verwaltung der Regionen in Hbase [Kel07].....119
Abbildung 39: Systemarchitektur von Hypertable [HTArch09].....120
Abbildung 40: BLOB als verkettete Seitenliste (Seite 107 [SHS05]).....142
Abbildung 41: BLOB-Directory (Seite 109 [SHS05]).....142
Abbildung 42: Einstufiger Tupel-Identifikator.....143
Abbildung 43: Geclusterter Index.....148
Abbildung 44: Nichtgeclusterter Index.....148
Abbildung 45: Hash-Implementierung [Seite 185 [SHS05]].....150
Abbildung 46: Huffman-Kodierung für Beispiel aus Tabelle 7.....154
Abbildung 47: Arithmetische Kodierung für Zeichenkette "baaaaaacaa".....160
Abbildung 48: Arithmetische Dekodierung für Zeichenkette "baaaaaacaa".....160
Abbildung 49: Lempel-Ziv-Algorithmenfamilie (aus [LZAlg1]).....161
Abbildung 50: Unterschied zwischen Tabelle mit Index und IOT (aus [OraCompr6]).....165
Abbildung 51: Kompression indexorganisierter Tabellen in Oracle 8i (aus [OraCompr6]). .166
Abbildung 52: Kompression auf Blockebene mit Symboltabelle in Oracle ([OraCompr4])..167
Abbildung 53: Batchprocessing bei OLTP-Blockkompression in Oracle 11g ([OraCompr4]) 168
Abbildung 54: Abfrage der komprimierten Tabellen in Oracle 11g (Seite 17 [OraCompr3]) 170
Abbildung 55: Compression Advisor in Oracle 11g (Seite 16 [OraCompr3]).....171
Abbildung 56: Kompression von zwei Tupeln in IBM DB 2 ([Ah06]).....171
Abbildung 57: Shared-Memory-Umgebung (aus [CB05]).....178
Abbildung 58: Shared-Disk-Umgebung (aus [CB05]).....180
Abbildung 59: Shared-Nothing-Umgebung (aus [CB05]).....180

Abbildung 60: Logische Dekomposition einer Relation, [Seite 10 [ETH1]].....181
Abbildung 61: Physische Speicherung der Dekomposition aus Abbildung 53, [Seite 11 [ETH1]].....181

Tabellenverzeichnis

Tabelle 1: Abbildung der konzeptuellen Ebene auf die interne Ebene und das Dateisystem ([SHS05])....11
Tabelle 2: Klassifizierung der Speicherhierarchien (Kapitel 2.3 [SHS05])....13
Tabelle 3: Kennzahlen von Magnetplattenspeichern (aus [SHS05])....15
Tabelle 4: Lesen von 1.000 Blöcken à 8 KB (aus [ETH1])....16
Tabelle 5: Übersicht über spaltenorientierte DBMS....27
Tabelle 6: Anfragecharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])....31
Tabelle 7: Datencharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])....31
Tabelle 8: Anwendercharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])....31
Tabelle 9: Beispiel für Differenz-Komprimierung (aus [ROTH93])....48
Tabelle 10: Kompressionsfunktionalität in Oracle (aus [OraCompr5])....49
Tabelle 11: Beispielrelation ANG [Mic05]....71
Tabelle 12: Projektionen für Beispielrelation in Tabelle 11....71
Tabelle 13: Ansätze / Klassifizierungskriterien für C-Store/Vertica....88
Tabelle 14: Data Pack Nodes (DPN) für Datenpakete in Abbildung 25 ([IBWP08])....90
Tabelle 15: Datenpaketknoten (DPN) für Datenpakete in Abbildung 25 ([IBWP08])....93
Tabelle 16: Datenpaketknoten für Relation X ([IBWP08])....94
Tabelle 17: Pack-to-Pack-Wissensknoten für Attribute B und C [IBWP08]....94
Tabelle 18: Features von Infobright Community Edition [ICE31Specs]....95
Tabelle 19: Ansätze / Klassifizierungskriterien für Infobright....98
Tabelle 20: Ansätze / Klassifizierungskriterien für LucidDB....106
Tabelle 21: Komponenten von BigTable, Hbase, Hypertable und Cassandra....110
Tabelle 22: Konzeptuelle Sicht auf eine BigTable/HBase/HyperTable/Cassandra-Tabelle ([HBArch])....111
Tabelle 23: Online-Shop realisiert mit Hypertable [GoCo09]....112
Tabelle 24: Physische Speicherung der Daten aus Tabelle 20 [HBArch]....113

Tabelle 25: Ansätze / Klassifizierungskriterien für Bigtable, Hbase, Hypertable und Cassandra 124
Tabelle 26: Klassifizierungskriterien für Valentina Database.....125
Tabelle 27: Ansätze / Klassifizierungskriterien für Vectornova.....126
Tabelle 28: Ansätze von kdb+.....129
Tabelle 29: Ansätze von Skytide XOLAP.....130
Tabelle 30: Klassifizierung der Ansätze für coDBMS.....135
Tabelle 31: Eindimensionaler Index über zwei Attributen.....149
Tabelle 32: Beispielrelation für Bitmap-Indizierung [Seite 1284 [CB05]].....150
Tabelle 33: Bitmap-Indizes für Beispielrelation aus Tabelle 27 [Seite 1284 [CB05]].....151
Tabelle 34: Relation 1 für Verbundindex [Seite 1285 [CB05]].....151
Tabelle 35: Relation 2 für Verbundindex [Seite 1285 [CB05]].....151
Tabelle 36: Verbundindex für Tabellen 29 und 30.....152
Tabelle 37: Bitraster mit Lauflängenkodierung (Abbildung 22.1 [Sed92]).....153
Tabelle 38: Beispiel-Kodierung der Symbole für "ABRACADABRA".....153
Tabelle 39: Huffman-Kodierung für Wort "ABRACADABRA".....154
Tabelle 40: Tabelle 3.11 aus [Say06].....156
Tabelle 41: Tabelle 4.2 aus [Say06].....157
Tabelle 42: Algorithmus der Arithmetischen Kodierung (aus [Bo09]).....158
Tabelle 43: Algorithmus der Arithmetischen Dekodierung (aus [Bo09]).....159
Tabelle 44: Übersicht über Lempel-Ziv-Algorithmen (aus [LZAlg]).....161
Tabelle 45: LZ77-Algorithmus (aus Kapitel 3.2 [LZAlg]).....162
Tabelle 46: Beispiel für eine Kompression nach LZ77 (aus [Wander05]).....162
Tabelle 47: Dekompression der Daten aus Tabelle 41.....163
Tabelle 48: LZ78-Algorithmus (aus [LZAlg])].....163
Tabelle 49: Beispiel für LZ78-Kodierung (aus [BELZ78]).....164
Tabelle 50: Dekodierung des Beispiels aus Tabelle 14.....164
Tabelle 51: Oracle SQL-Syntax für OLTP-Kompression ([OraCompr4]).....169
Tabelle 52: Oracle SQL-Syntax für Bulkload-Kompression ([OraCompr4]).....169
Tabelle 53: Vergleich der Kompression von Oracle und IBM DB2 (aus [OraCompr3]).....172
Tabelle 54: Beispiel für zu komprimierende Daten ([OraCompr6]).....173
Tabelle 55: Symboltabelle für Beispiel aus Tabelle 28.....174
Tabelle 56: Relation kodiert mit Symboltabelle (Oracle-Ansatz).....174

Tabelle 57: Kodierungen für Spalte 1 für Beispiel aus Tabelle 28.....175
Tabelle 58: Kodierungen für Spalte 2 für Beispiel aus Tabelle 28.....175
Tabelle 59: Kodierungen für Spalte 3 für Beispiel aus Tabelle 28.....175
Tabelle 60: Kodierungen für Spalte 4 für Beispiel aus Tabelle 28.....176
Tabelle 61: Klassifikation von parallelen DBMS (ähnlich Kapitel 3 [Rahm94]).....177
Tabelle 62: Unterschiede zwischen parallelen und verteilten DBMS (aus [Gon09]).....177
Tabelle 63: Beispiel für Natural Join aus [WikiRA].....185
Tabelle 64: Semi-Join für Relation R und S [WikiRA].....185
Tabelle 65: SQL-Anfrage für einen Star-Join [Seite 640 [SSH08]].....186

Abkürzungsverzeichnis

ANSI	American National Standards Institute
ANSI-SPARC	ANSI-Scalable Processor Architecture
ATA	Advanced Technology Attachment
BI	Business Intelligence
coDBMS	column oriented (spaltenorientiertes) DBMS
CPU	Central Processing Unit
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
DDL	Data Definition Language
DFS	Distributed File System
DML	Data Manipulation Language
DS	Data Source
DSM	Decomposite Storage Model
GB	Gigabyte
HDFS	Hadoop Distributed File System
HOLAP	Hybrides OLAP
ICE	Infobright Community Edition
IOT	Indexorganisierte Tabellen
JDBC	Java Database Connectivity
KB	Kilobyte
ODBC	Open Database Connectivity
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
PB	Petabyte
RDBMS	Relationales DBMS
RLE	Run Length Encoding
roDBMS	row oriented (zeilenorientiertes) DBMS
SAN	Storage Area Network
SATA	Serial ATA
SID	Segment Identifier

SQL	Structured Query Language
TB	Terabyte
TID	Tupel-Identifikator
URL	Uniform Resource Locator

1 Einleitung

In dieser Diplomarbeit soll ein Überblick über das relativ junge Forschungsgebiet der sogenannten spaltenorientierten (engl. *column oriented*) Datenbankmanagementsysteme (**DBMS**) gegeben werden. Weitere übliche englischsprachige Bezeichnungen für diese Art von **DBMS** sind *Column Stores* oder *Columnar Stores*. Das Pendant dazu sind die „klassischen" zeilenorientierten **DBMS**, zu denen auch die der großen Hersteller Oracle, Microsoft (SQL Server) und IBM (DB 2) gehören. Diese werden im Englischen auch *Row Stores* genannt. Im weiteren Verlauf der Diplomarbeit wird für den Begriff „column oriented **DBMS**" die Abkürzung **coDBMS** verwendet. Für zeilenorientierte **DBMS** (engl. *row oriented DBMS*) wird die Abkürzung **roDBMS** verwendet.

1.1 Motivation

Datenbanksysteme, auch die „klassischen" relationalen, setzen sich aus mehreren Ebenen zusammen. Auf der externen und auch der konzeptuellen Ebene werden die Datenstrukturen durch Sichten und Relationenschemata bzw. Tabellen beschrieben, die ein Modell eines Ausschnitts der realen Welt repräsentieren. Ein Tupel einer Relation bzw. eine Tabellenzeile beschreibt dabei über seine Attributwerte ein konkretes Objekt (bzw. eine Instanz) dieses modellierten Ausschnitts. Unabhängig von dieser konzeptuellen Darstellung müssen die als Daten abgebildeten Informationen aber auf physischer Ebene in eine eindimensionale Struktur, nämlich einen linearen Adressraum des Primär- oder Sekundärspeichers, abgebildet werden.

Insbesondere das persistente Ablegen von Daten im Sekundärspeicher und umgekehrt das Laden dieser abgelegten Daten in den Primärspeicher zum Zweck der Auswertung (beziehungsweise Beantwortung teilweise komplexer Anfragen an das **DBMS**) ist aufgrund der existierenden Zugriffslücke zwischen Sekundär- und Primärspeicher schon immer Betrachtungsgegenstand der Forschung im Bereich Datenbanken hinsichtlich möglicher Schreib- und Leseoptimierungen gewesen.

Insbesondere die Leseoptimierung ist aufgrund weltweit stark wachsender Datenbestände und den dadurch gestiegenen Anforderungen an eine zeiteffiziente Analyse und Auswertung dieser (großen, integrierten) Datenmengen im Rahmen des Online Analytical Processing

(**OLAP**) eins der wesentlichen Forschungsgebiete in diesem Bereich. Bei sogenannten transaktionalen Datenbanksystemen werden Daten in kurzen und einfachen Transaktionen im Rahmen des Online Transaction Processing (**OLTP**) gelesen, erzeugt, modifiziert oder gelöscht, in analysorientierten Systemen (z.B. Data Warehouses) werdenInformationen durch lange Lesetransaktionen bzw. komplexe Anfragen gewonnen, die Millionen von Datensätzen betreffen und ad hoc[1] formuliert werden, vgl. [Seite 9 [BG09]].

Data-Warehouse-Größen betragen heutzutage mehrere Terabyte (**TB**) bis zu 100 **TB** (Stand 2005) für die weltweit größten (Beispiele unter anderem [BNET1], [Seite 525 ff. [BG09]], [Seite 80-81 [SHS05]] , [VERT1] und [WinterCorp05]), wobei die Daten in der Regel auf Festplatten gespeichert sind. Zwar haben die heute handelsüblichen **SATA**-Festplatten eine Übertragungsrate bei sequenziellem Zugriff von 150 bis 300 **MB**/s, jedoch ist diese bei wahlfreiem Zugriff erheblich geringer (vgl. Seite 18 [ETH1]). So stellt sich die Frage, wie die physische Struktur der Daten so optimiert werden kann, dass trotz sich stetig ändernder Analyseanforderungen und den damit verbundenen Ad-hoc-Anfragen ein sequenzieller Zugriff nur auf die benötigten Daten erfolgen kann.

Denn immerhin ist trotz der heutigen hohen Übertragungsraten selbst bei optimalen 300 **MB**/s (**SATA** 2) für 1 **TB** Daten (= 1.000 **GB** bzw. 1.000.000 **MB**) eine Übertragungszeit von 55 Minuten notwendig. Müssen aufgrund der physikalischen (zeilenorientierten) Anordnung im Sekundärspeicher sämtliche Attribute der vorhandenen Relationen zunächst vollständig eingelesen werden, obwohl nur wenige von Ihnen zur Beantwortung einer Anfrage benötigt werden, ist in diesem Punkt ein Optimierungspotenzial vorhanden. Hier scheinen **coDBMS** ein Ansatz zu sein, dessen Betrachtung lohnend ist, wie zum einen verschiedene Performance- und Speicherplatzverbrauchs-Messungen (vgl. [Seite 29 [Aba08]], [VERT1], [TPC-H1], [ParAccel1]) von Data Warehouses zeigen, in denen mittlerweile einige **coDBMS** in den vorderen Rängen zu finden sind, und zum anderen die Tatsache, dass ein Hersteller wie Sun eine Data-Warehouse-Architektur für über 1 Petabyte (**PB**) Daten unter Verwendung des **coDBMS** Sybase IQ aufgebaut hat, vgl. [Sybase1]. Dass der spaltenorientierte Ansatz mehr und mehr in den Fokus im Bereich der **DBMS** zu rücken scheint, zeigen auch die stetigen Ergänzungen des entsprechenden Wikipedia-Artikels, die während der Erstellung der Diplomarbeit zu beobachten waren ([WikiCODBMS]).

[1] Spontan, nicht vorhersehbar

1.2 Zielsetzung

Ziel dieser Literaturarbeit ist es, im ersten Schritt die Grundlagen von **coDBMS** zu erläutern und eine Übersicht über existierende Implementierungen zu geben. Auch werden Unterschiede zu „klassischen“ zeilenorientierten, in der Regel relationalen, **DBMS** dargestellt.

Im nächsten Schritt werden einige dieser spaltenorientierten Implementierungen vorgestellt, und es wird versucht, anhand von sich aus der Beschaffenheit der Implementierungen ergebenden Kriterien eine Klassifizierung der **coDBMS** vorzunehmen. Kriterien können dabei z.B. die sich aus der Dokumentation ergebenden Einsatzgebiete bzw. Funktionalitäten der **DBMS**, architektonische Prinzipien oder verwendete Algorithmen sein.

Abschließend werden die wesentlichen Unterschiede, die sich im Rahmen der Recherchen ergeben haben, sowie die Klassifizierungskriterien zusammenfassend dargestellt.

1.3 Gliederung der Diplomarbeit und Formatierungen

In Kapitel 2 werden die zur Klassifizierung von **coDBMS** notwendigen Grundlagen beschrieben. Ebenfalls in Kapitel 2 werden bereits grundlegende Unterschiede zu **roDBMS** dargestellt, die sich noch nicht auf eine konkrete Implementierung der **coDBMS** beziehen. In Kapitel 3 werden die konkreten Implementierungen zunächst vorgestellt und anhand ihrer Eigenschaften klassifiziert. Dabei werden auch Unterschiede zu anderen Implementierungen von **coDBMS** und zu **roDBMS** herausgestellt. Abschließend wird in Kapitel 4 ein Gesamtfazit der Betrachtungen gezogen.

Detaillierte Beschreibungen der Grundlagen, die einem fachkundigen Leser der Diplomarbeit in der Regel geläufig sind, sind in den Anhang ausgegliedert. Auf diesen kann bei Bedarf zurückgegriffen werden.

Abkürzungen, die im Abkürzungsverzeichnis erläutert werden, werden in dieser Arbeit **fett** hervorgehoben. Hervorgehobene Begriffe (Schlagwörter) werden *kursiv* dargestellt. Angaben zu Literaturquellen werden in [eckigen Klammern] dargestellt. Unterstrichene Wörter sind im Glossar (Anhang B) erläutert.

2 Grundlagen und Unterschiede roDBMS / coDBMS

In diesem Kapitel werden die für das Verständnis der nachfolgenden Kapitel notwendigen Grundlagen beschrieben sowie Unterschiede zwischen **roDBMS** und **coDBMS** dargestellt, die sich nicht auf eine konkrete Implementierung von **coDBMS** beziehen.

2.1 9 Codd'sche Regeln für ein DBMS

Da in dieser Arbeit verschiedene Implementierungen betrachtet werden, die alle laut recherchierter Literaturquellen unter den Oberbegriff **coDBMS** fallen, sollen an dieser Stelle zunächst die Aufgaben eines **DBMS** erläutert werden. So kann in Kapitel 3 beurteilt werden, ob es sich bei der jeweiligen Implementierung um ein **DBMS** im Sinne der Codd'schen Regeln handelt, welche sind (vgl. [Seite 7-8 [HS00]]):

1. *Integration*: Ein **DBMS** soll die von allen Anwendungen benötigten Daten einheitlich und nicht-redundant verwalten.
2. *Operationen*: Ein **DBMS** muss Operationen zum Sichern, Suchen und Ändern der Daten anbieten.
3. *Katalog*: Der Katalog (engl. *Data Dictionary*) enthält die Datenbeschreibungen der Datenbank.
4. *Benutzersichten*: Ein **DBMS** muss die Erzeugung und Verwaltung verschiedener Benutzer- bzw. Anwendungssichten auf den (einheitlich verwalteten) Datenbestand ermöglichen und diese Sichten verwalten.
5. Konsistenzüberwachung: Ein **DBMS** muss die Integrität (Korrektheit und Vollständigkeit) der Daten gewährleisten.
6. *Zugriffskontrolle*: Ein **DBMS** muss sicherstellen, dass nur authorisierte Anwender auf die gespeicherten Daten zugreifen können.
7. *Transaktionen*: Ein **DBMS** muss die Ausführung von Operationen im Rahmen von Transaktionen anbieten. Eine *Transaktion* ist eine Zusammenfassung von Datenbankoperationen, die nur als Ganzes ausgeführt werden können und deren Änderungen bei Erfolg persistent gespeichert werden.

8. *Synchronisation*: Konkurrierende Transaktionen verschiedener Benutzer (auf denselben Daten) müssen synchronisiert werden, um gegenseitige Beeinflussungen zu vermeiden.
9. *Datensicherung*: Das DBMS muss im Fall von Systemfehlern die Wiederherstellung von Daten gewährleisten können.

2.2 Architektur von DBMS

Um zu verdeutlichen, wo die wesentlichen Unterschiede zwischen zeilen- und spaltenorientierten **DBMS** liegen, wird an dieser Stelle kurz die Architektur von **DBMS** beschrieben. Diese Architektur kann man aus verschiedenen Blickwinkeln betrachten, gemäß [Seite 25 ff. [HS00]] aus Sicht der Schemata (*Schemaarchitektur*), des Datenbanksystems bzw. seinen Komponenten (*Systemarchitektur*) und aus Sicht der Anwendungsentwicklung (*Anwendungsarchitektur*).

Zum besseren Verständnis der Einordnung des Diplomthemas in den Gesamtkontext von Datenbankkonzepten und -implementierungstechniken werden an dieser Stelle Schema- und Systemarchitektur beschrieben.

Ein Datenbanksystem **(DBS)** setzt sich nach der Drei-Ebenen-Schemaarchitektur (siehe Abbildung 1) aus drei Ebenen zusammen, und zwar der externen, konzeptuellen und internen Ebene, vgl. [Seite 27-28 [HS00]]. Die externe Ebene liefert (anwendungsspezifische) Sichten auf den Gesamtdatenbestand, wohingegen die konzeptuelle Ebene eine logische, anwendungsunabhängige Gesamtsicht auf denselbigen liefert. Sowohl externe als auch konzeptuelle Ebene sind noch implementierungsunabhängig. Die interne Ebene realisiert die Implementierung von Datenstrukturen (d.h. *Dateiorganisationsformen* und *Zugriffspfaden*).

Die beschriebene Drei-Ebenen-Schemaarchitektur nach **ANSI** führt zur physischen und logischen Datenunabhängigkeit. Die *physische Datenunabhängigkeit* beschreibt den Umstand, dass die konzeptuelle Sicht auf die Daten unabhängig ist von den gewählten Datenstrukturen. Sie kapselt damit auch die interne Realisierung einer zeilen- oder spaltenorientierten Datenablage (welche den Gegensatz von zeilen- und spaltenorientierten Datenbanken ausmacht) von der konzeptuellen Sicht auf den Datenbestand ab, sodass ein **DBS** nicht zwangsweise zeilen- *oder* spaltenorientiert sein muss, sondern durchaus beide Techniken vereinen und je nach Anwendungsszenario zwischen ihnen wechseln kann. Auf [Seite 19 [Aba08]] z.B. wird beschrieben, dass auch bei einer spaltenorientierten

Speicherarchitektur durchaus ein (zeilenorientierter) Query Processor[2] zum Einsatz kommen kann, da die für eine Anfrage benötigten Spalten (bzw. *Attribute* im relationalen Datenmodell) auf der (architektonischen) Ebene des Query Processors des **DBMS** bereits zu Tupeln verbunden sind (vgl. auch [DBM06]). Ebenso verweist [Seite 130 [Aba08]] auf Untersuchungen eines kombinierten (hybriden) Ansatzes von Column- und Row Store in [RDS02]. Die Betrachtung eines hybriden Ansatzes und die Abwägung, wann welcher Ansatz zum Einsatz kommen sollte, sind jedoch nicht Betrachtungsgegenstand dieser Arbeit.

Die *logische Datenunabhängigkeit* beschreibt die Trennung von Datenbank und Anwendungsschnittstelle. Deswegen kann auch ein spaltenorientiertes **DBMS** „tupelorientierte" Anwendungsschnittstellen wie **ODBC** und **JDBC** anbieten (vgl. u.a. [Seite 67 [Aba08]]) (und natürlich auch **SQL** als Anfragesprache).

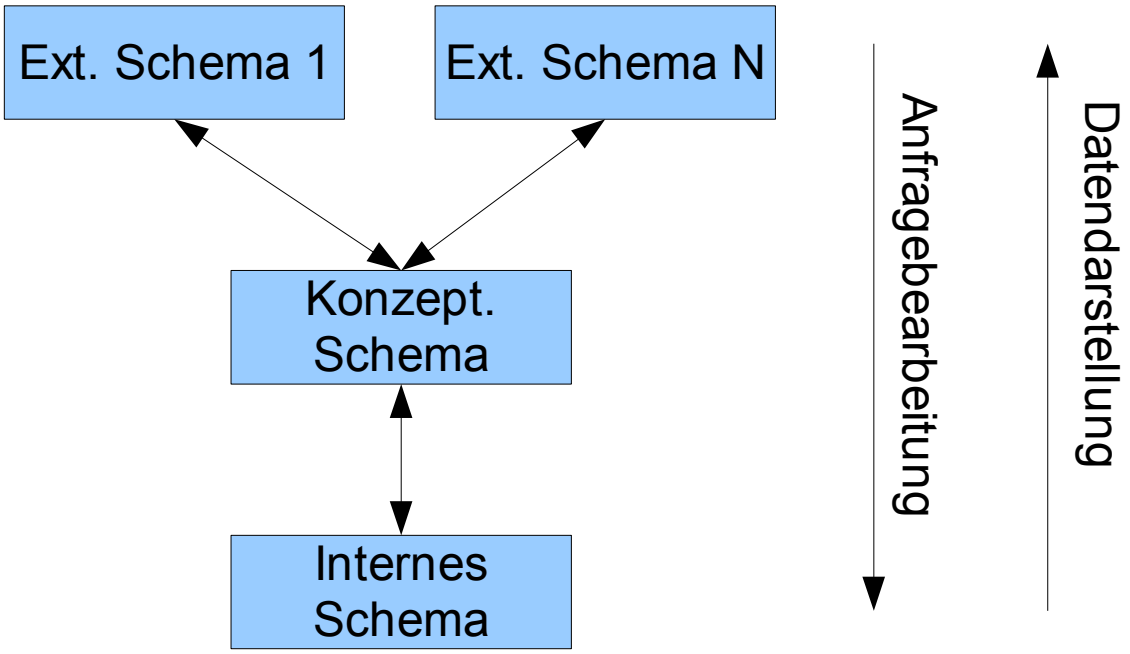

Abbildung 1: Drei-Ebenen-Schema-Architektur (ANSI), vgl. [HS00]

Anhand der *Systemarchitektur* eines **DBS** kann das Diplomthema noch genauer eingegrenzt werden. Die Systemarchitektur beschreibt die Komponenten eines **DBMS** bzw. **DBS**.

[2]Ein Query Processor ist die Komponente in einem DBMS, die für eine gegebene Anfrage (z.B. in SQL) einen Anfrageplan bestehend aus Datenbankoperationen erstellt, diesen optimiert und schlussendlich ausführt, vgl. u.a. [Seite 701 [GMUW09]]

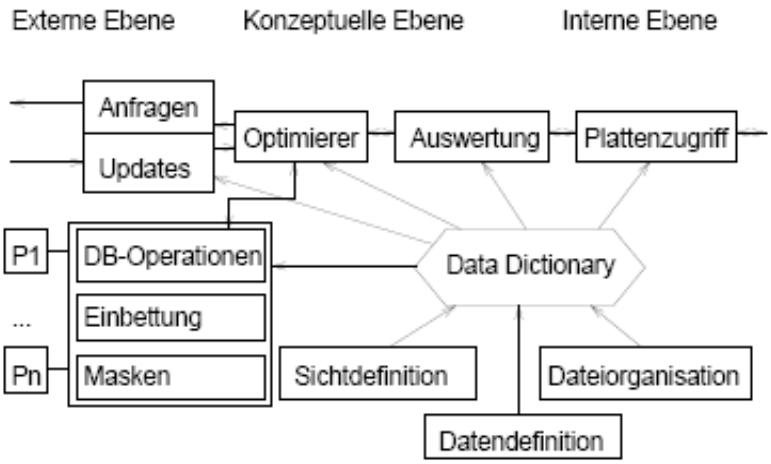

Abbildung 2: ANSI-SPARC-Architektur eines DBS (gem. [HS00])

Gemäß [Seite 31 [HS00]] gibt es für die Systemarchitektur zwei wichtige Architekturvorschläge, nämlich die **ANSI-SPARC**-Architektur als detaillierte Version der oben beschriebenen Drei-Ebenen-Schemaarchitektur sowie die Fünf-Schichten-Architektur nach Senko/Härder (siehe u.a. [Seite 11 ff. [HR99]]). Die **ANSI-SPARC**-Architektur mit ihren Komponenten ist in Abbildung 2 dargestellt.

Die in Abbildung 2 aufgeführten Komponenten, die an dieser Stelle nicht weiter erläutert werden sollen, kann man z.B. wie in Abbildung 3 gezeigt klassifizieren in Benutzerkomponenten, Programmierkomponenten, Data Dictionary, Definitions- und Transformationskomponenten. Von besondererem Interesse bei der Betrachtung des Themas der Diplomarbeit sind dabei die Transformationskomponenten, die Anfrage- und Änderungsoperationen an das **DBS** schrittweise über Optimierung und Auswertung in Plattenzugriffsoperationen umwandeln und in umgekehrter Richtung die im (persistenten) Speicher abgelegten Daten in die externe Darstellung (Sichten) transformieren. Zu den Transformationskomponenten gehören der Query Processor (inklusive Optimierer), die Transaktionsverwaltung sowie die Plattenzugriffs- und Dateiorganisationskomponente. In den Transformationskomponenten, genauer der Dateiorganisation, wird entschieden, wie die Daten in den Sekundärspeicher geschrieben und von dort wieder gelesen werden (unter Berücksichtigung der Speicherhierarchie des Rechners), somit auch ob (im Sekundärspeicher) die Daten zeilen- bzw. tupelweise (engl. *Row Store*) oder spalten- bzw. attributweise (engl. C*olumn Store*) geschrieben und gelesen werden..

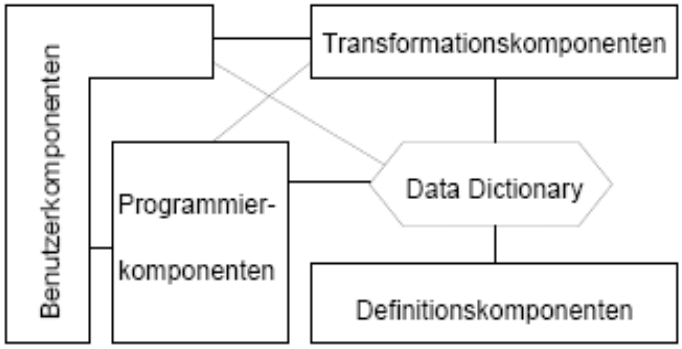

Abbildung 3: Klassifikation von Komponenten eines DBMS (Seite 32 [HS00])

Die Transformationskomponenten lassen sich nach [Seite 40 [SHS05]], in fünf Schichten mit jeweils dazwischenliegenden Schnittstellen aufteilen. Eine an [Seite 34 [SHS05]] angelehnte Darstellung in Abbildung 5 zeigt die Transformationskomponenten *Datensystem*, *Zugriffssystem*, *Speichersystem*, *Pufferverwaltung* und *Betriebssystem*. Abbildung 4 zeigt eine aus [SK08] entnommene ähnliche Struktur, die einzelne Komponenten wie z.B. den im Speichersystem enthaltenen *Lock Manager* (zuständig für die Sperrverwaltung) grafisch hervorhebt.

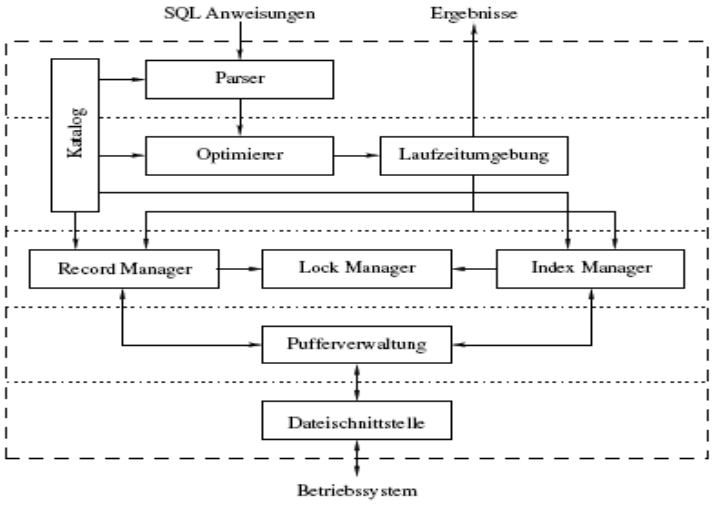

Abbildung 4: Systemarchitektur eines DBMS (aus [SK08])

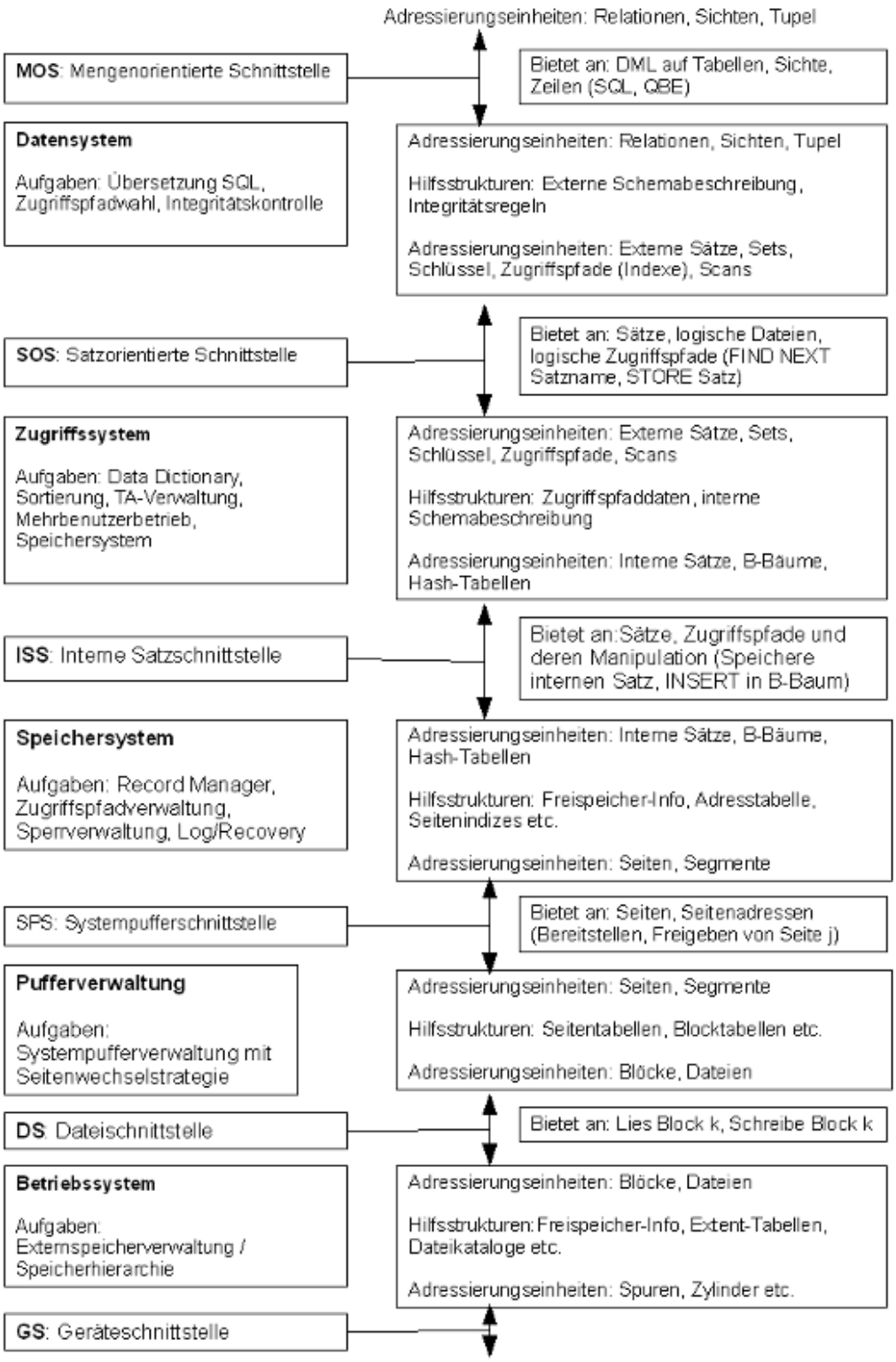

Abbildung 5: Fünf-Schichten-Architektur der Transformationskomponenten des DBMS

Die unterste zu einem **DBMS** gehörende Komponente ist dabei die Pufferverwaltung. Sie bildet interne Seiten auf *Blöcke* der Dateischnittstelle des Betriebssystems ab und speichert zur Überwindung der Zugriffslücke zwischen Sekundär- und Primärspeicher zumindest einen Teil der Datenbank im Hauptspeicher. Ein Block ist ein reiner Bytecontainer (in der Regel 512 bis 4.096 Bytes), eine Folge von Blöcken ist das abstrakte Modell eines Sekundärspeichers (bei einer Festplatte z.B. bestehend aus Platten, Spuren, Zylindern und Sektoren), vgl. [Seite 95 [SHS05]]. Eine Seite wiederum ist in der Pufferverwaltung das Modell eines Blocks eines Sekundärspeichermediums wie z.B. einer Festplatte. Sämtliche Tupel, die vom Sekundärspeicher in den Primärspeicher (Hauptspeicher bzw. Cache-Speicher) geladen werden, werden über ihre Seiten in den Puffer geladen. Durch das Dateisystem des Betriebssystems erfolgt die Zuordnung physischer Blöcke zu Seiten, meist werden 1,2,4 oder 8 Blöcken zu 1 Seite zusammengefasst, [Seite 97 [SHS05]]. Die höheren Schichten eines **DBS** adressieren nur die Seiten (über die Seitennummer), nicht die Blöcke.

Über die Transformationskomponenten werden die Daten der konzeptuellen Ebene (dies sind Relationen bei relationalen Datenbanksystemen) auf Seiten auf physischen Dateien abgebildet. Umgekehrt müssen die internen Seiten bzw. ihr Inhalt (Bytes) auch wieder als Relation mit Tupeln und Attributwerten interpretiert werden können. Die Abbildungshierarchie wird in Tabelle 1 dargestellt.

Konzeptuelle Ebene	Relation	Tupel	Attributwerte
	↓	↓	↓
Interne Ebene (Datensystem/Zugriffssystem)	Logische Datei	Datensätze (Records)	Felder
	↓	↓	↓
Dateisystem/Platte	Physische Datei	Seiten/Blöcke	Bytes

Tabelle 1: Abbildung der konzeptuellen Ebene auf die interne Ebene und das Dateisystem ([SHS05])

Die Art der Abbildung ist gem. [Seite 98 [SHS05]] abhängig von den Dateiorganisationsformen und internen Schemata des **DBS**, der konkreten Speicherung und Adressierung auf physischer Ebene und der Kodierung der Bytes. Abbildung 6 zeigt ein Beispiel aus [SHS05], wobei hier der zeilenorientierte Ansatz dargestellt wird, das heißt jedes Tupel bzw. jeder Datensatz (Record) wird mit all seinen Attributen sequenziell in den Seiten bzw. Blöcken der physischen Datei abgelegt, entweder als Spannsatz oder Nicht-Spannsatz

(sog. „Blockungstechnik“). In den Folgekapiteln 2.3 bis 2.7 wird die Speicherarchitektur eines **roDBMS** bis zur Pufferverwaltung erläutert, Kapitel 2.8 geht dann auf die Unterschiede zu einem **coDBMS** ein und Kapitel 2.9 gibt einen Überblick über existierende Implementierungen von **coDBMS.**

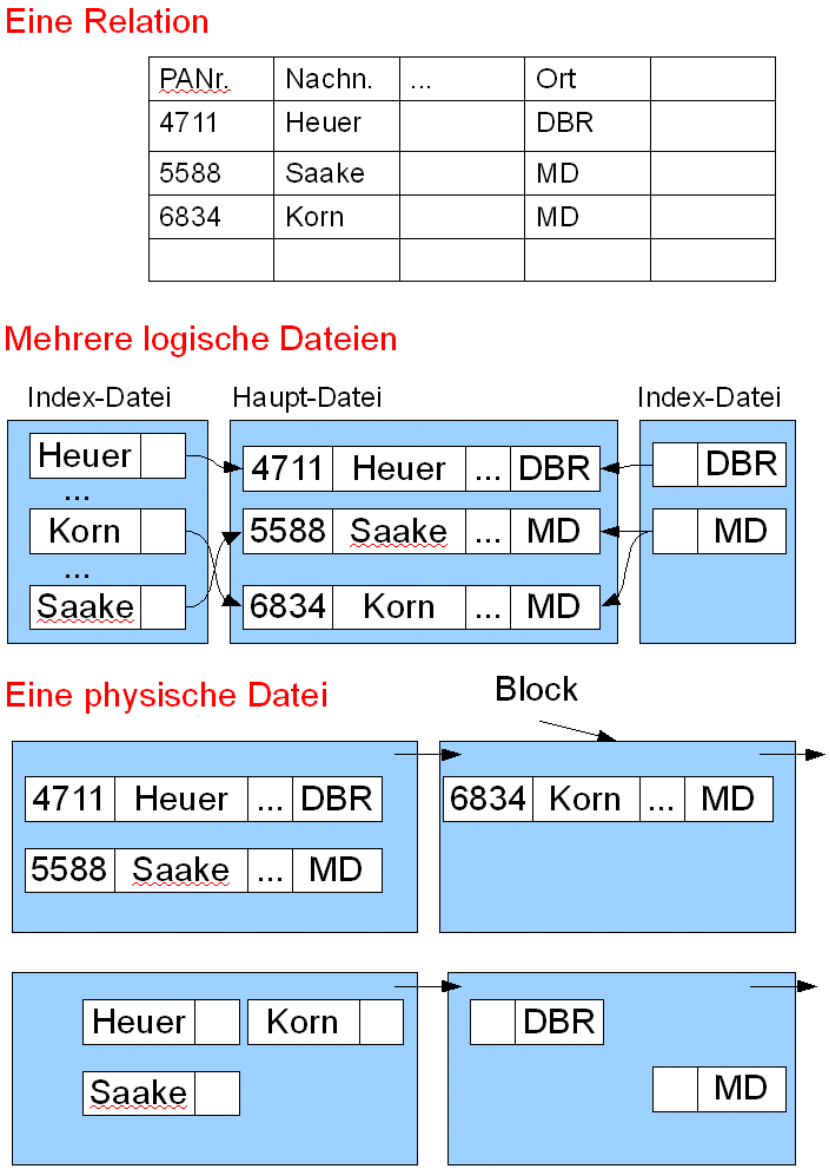

Abbildung 6: Speicherung einer Relation in physischer Datei ([SHS05])

2.3 Speicherhierarchie

Um die Vorteile von spaltenorientierten **DBMS** zu verstehen, ist ein Verständnis der Speicherhierarchie eines Rechners notwendig. Ein **DBMS** erzeugt, speichert und ändert seine Daten in einer dreistufigen Speicherhierarchie, bestehend aus Primär-, Sekundär- und Tertiärspeicher. Zum Primärspeicher gehören der Cache des Prozessors und der Hauptspeicher des Rechners, der Sekundärspeicher ist der Plattenspeicher des Rechners und der Tertiärspeicher besteht aus optischen Platten sowie Magnetbändern. Ein **DBMS** nutzt dabei in der Regel nur Primär- und Sekundärspeicher, der Zugriff auf Tertiärspeicher, zumindest auf optische Platten, kommt nur beim Zugriff auf selten benötigte oder ältere Daten bzw. zur Datenwiederherstellung und -archivierung vor. Tabelle 2 zeigt eine Klassifizierung der Speicherhierarchien.

Speicher	Typ	Geschwin-digkeit	Preis	Stabilität	Größe	Granulate[3]
Primär	Cache Haupt	schnell	teuer	Flüchtig	klein	fein
Sekundär	Platte	langsam	preiswert	Stabil	groß	grob
Tertiär	Optische Platte Magnetbänder	Sehr langsam	billig	Stabil	Sehr groß	grob

Tabelle 2: Klassifizierung der Speicherhierarchien (Kapitel 2.3 [SHS05])

Grundsätzlich gilt, dass Hierarchieebene *x* gegenüber Hierarchieebene *x+1* zwar eine schnellere Zugriffszeit, aber dafür einen höheren Preis und deshalb in der Regel eine geringere Kapazität aufweist. Die sogenannte *Zugriffslücke* (verschiedene Zugriffsgeschwindigkeiten auf Daten in den verschiedenen Hierarchieebenen) versucht man durch *Cache-Speicher* zu vermindern. Lt. [Seite 76 [SHS05]] beträgt die Zugriffslücke zwischen Hauptspeicher und Plattenspeicher 10^5.

Genauer betrachtet werden soll hier der Sekundärspeicher, da die Unterscheidung zwischen R*ow Store* und C*olumn Store* sich im Wesentlichen auf die Art der Sicherung und des Ladens der Daten im Sekundärspeicher bezieht (auch wenn die Spaltenorientierung der Daten von einigen **coDBMS** auch im Hauptspeicher noch aufrechterhalten wird). **DBMS** sichern ihre Daten in der Regel auf nicht flüchtigem Sekundärspeicher (Festplatten) und laden sie von

[3]Das Granulat (Tabelle 2) beschreibt die Genauigkeit der Adressierbarkeit eines Speicherbereichs. Bei Cache und Hauptspeicher ist jedes Byte adressierbar, beim Sekundärspeicher (Festplatte) nur Blöcke von Bytes (in der Regel 512 Bytes)

diesem über den Primärspeicher (Pufferverwaltung) in die **CPU**. Der Puffer ist der bereits erwähnte Cache, der Ausschnitte des Sekundärspeichers im Hauptspeicher (temporär) zwischenspeichert. Dieses Prinzip gilt sowohl für R*ow Stores* als auch für C*olumn Stores*. Ist die gesamte Datenbank im Puffer geladen, spricht man von einer *Hauptspeicherdatenbank* (vgl. [Seite 46 [SHS05]]).

Festplatten als Form des Sekundärspeichers bestehen aus rotierenden Magnetplatten, die in Spuren und Sektoren unterteilt sind, sowie aus Zugriffsarmen, Schreib- und Leseköpfen und einem Controller (heute sind **SATA**-Controller üblich, vgl. [WikiIDE]) mit Mikroprozessor und eigenem Cache-Speicher, der die Schnittstelle zum Übertragungssysten des Rechners bildet und den Zugriff auf die Sektoren der Magnetplatten steuert. Die kleinste adressierbare Einheit für wahlfreien Zugriff ist dabei der *Sektor*, der wiederum 512 Bytes an Daten beinhaltet. Auch wenn nur ein einzelnes Byte benötigt wird, müssen mindestes 512 Bytes eingelesen werden. Mitentscheidend für die Performance eines **DBMS** ist die Art und Weise des Festplattenzugriffs, welcher wahlfrei oder sequenziell erfolgen kann. Unter sequenziellem Zugriff wird dabei verstanden, dass die Daten in aufeinanderfolgenden Sektoren liegen und mit nur einem Lesezugriff eingelesen werden. Grundsätzlich ist jedes Anfordern von Blöcken zunächst einmal ein wahlfreier Zugriff. Bei diesem müssen 3 verschiedene Zeitfaktoren berücksichtigt werden, nämlich die Zeit zum Bewegen der Köpfe zur Spur (engl. S*eek Time*), die Zeit, bis der Sektor der rotierenden Platte an der Kopfposition angekommen ist (engl. L*atency Time*) und die Übertragungszeit (engl. T*ransfer Time*). Wie Tabelle 3 zeigt (Stand 2004), hat es bei der mittleren Zugriffsbewegungszeit (S*eek Time*) und der Umdrehungszeit (*Latency Time*) wenig bis keine Verbesserungen in den letzten 10 Jahren gegeben, jedoch hat die Übertragungszeit durch **SATA(-2)** einen Schub erfahren, wobei Abbildungen 7 und 8 eines Performancetests aus 2007 ([TH07]) zeigen, dass die maximal möglichen Übertragungszeiten bei **SATA** (150 bzw. 300 MB/s) in der Praxis nicht erreicht werden müssen.

Merkmal	2004	1998	1985	1970
Mittlere Zugriffsbewegungszeit (seek time)	5 ms	8 ms	16 ms	30 ms
Umdrehungszeit	6 ms	6 ms	16,7 ms	16,7 ms
Spurkapazität	420 KB	100 KB	47 KB	13 KB
Plattenoberflächen	4	20	15	19
Zylinder	90000	5000	2655	411
Kapazität	150 GB	10 GB	1,89 GB	93,7 MB

Tabelle 3: Kennzahlen von Magnetplattenspeichern (aus [SHS05])

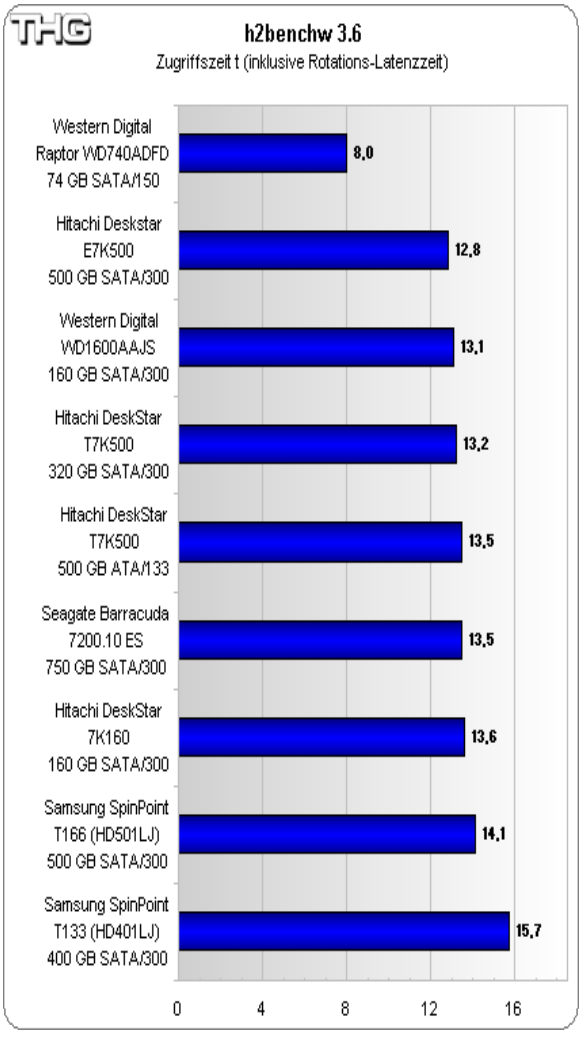

Abbildung 7: Festplatten-Zugriffszeiten (aus [TH07])

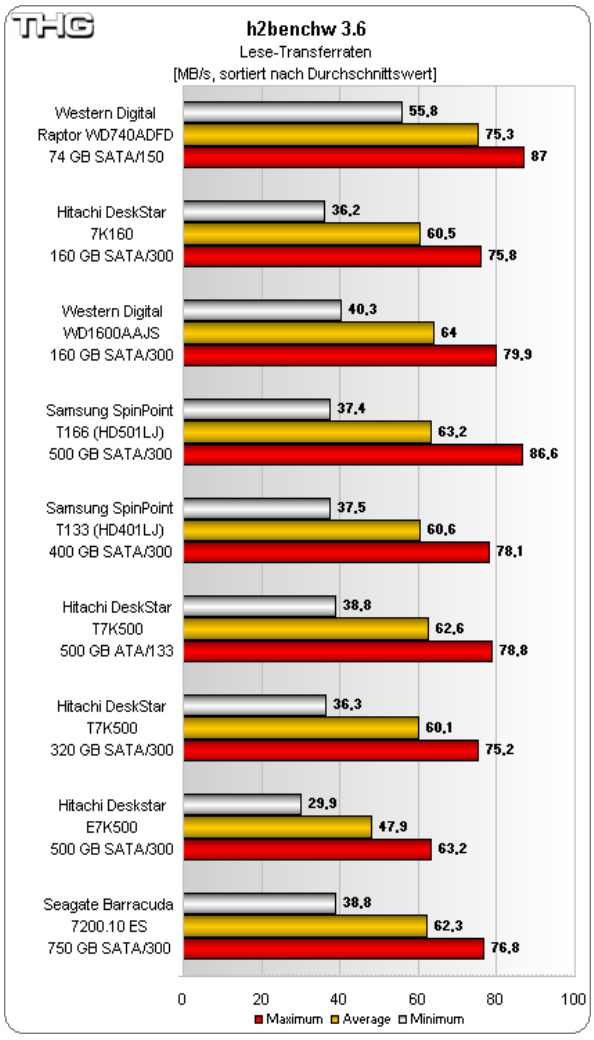

Abbildung 8: Festplatten-Lese-Transferraten (aus [TH07])

Nimmt man die (aus 2007 stammenden) Messungen in Tabelle 4 hinzu, die die Unterschiede von wahlfreiem und sequenziellem Zugriff auf eine Datenmenge von 8 MB darstellt, so zeigt

sich eine Verbesserung des sequenziellen Zugriffs gegenüber dem wahlfreien von ca. 18,5 (147,4 / 8). Das Ziel von Performanceoptimierungen sollte daher sein, möglichst S*eek Time* und L*atency Time* zu vermeiden, d.h. möglichst viele Daten sequenziell einzulesen (ohne dass der Lesekopf an eine neue Position geführt werden muss).

	1970	**2007**	**Verbesserung**
Wahlfrei	48.275 ms	6.000 ms	8
Sequenziell	10.315 ms	70 ms	147,4
Verhältnis	4,7	85,7	

Tabelle 4: Lesen von 1.000 Blöcken à 8 KB (aus [ETH1])

Bei Datenvolumen von mehreren **TB** bei Data Warehouses spielt die Übertragungszeit natürlich auch eine Rolle, die so gering wie möglich für die benötigten Daten sein sollte. Dies erreicht man dadurch, möglichst viel der benötigten Daten in aufeinanderfolgenden Sektoren der Festplatte abzulegen, zum einen um möglichst viel sequenziell einlesen zu können, zum anderen um die Dichte der relevanten Informationen im (begrenzten) Datenbankpuffer zu erhöhen. Hierzu gibt es mehrere Ansätze, unter anderem das *Clustering* von Daten (bei Verbundanfragen), das Erzeugen *materialisierter Sichten* (engl. *Materialized Views*) und eben auch das spaltenorientierte Ablegen von Informationen im Sekundärspeicher, erreicht durch die sogenannte *vertikale Partitionierung*, die lediglich ein anderer Begriff für die spaltenorientierte Speicherarchitektur, also den Betrachtungsgegenstand dieser Arbeit, ist. Ein anderer Begriff für vertikale Partitionierung ist *Decomposite Storage Model (**DSM**)* und wird im Folgekapitel 2.13.1 erläutert. Auf alle relevanten Ansätze zur Performanceoptimierung wird in den nachfolgenden Unterkapiteln eingegangen und der Unterschied zu **coDBMS** herausgearbeitet, die bei allen Ansätzen notwendige Reorganisation (nach dem Hinzufügen einer großen Menge neuer Daten) wird dabei nicht betrachtet.

2.4 Verwaltung des Hintergrundspeichers

Bei der Verwaltung des Hintergrundspeichers (Sekundärspeicher) gibt es sowohl Gemeinsamkeiten als auch Unterschiede zwischen **roDBMS** und **coDBMS**. Unter Bezugnahme auf [Seite 95 ff. [SHS05]] soll deshalb die Verwaltung des Hintergrundspeichers für **roDBMS** kurz beschrieben werden, damit in Kapitel 2.8 die Unterschiede bei **coDBMS** (ohne Bezugnahme auf eine konkrete Implementierung) dargestellt werden können.

Die Verwaltung des Hintergrundspeichers ist Aufgabe des Betriebssystems (siehe Abbildung 5 auf Seite 10). Das Betriebssystem abstrahiert vom Speicherungs- oder Sicherungsmedium

auf das Modell des Hintergrundspeichers, nämlich eine Folge von *Blöcken* (vgl. Kapitel 2.2) . Zu lösende Problematiken sind:

1. Wie werden Relationen und Zugriffspfade (Indizes) intern gespeichert ?
2. Wie werden Tupel bzw. Datensätze (Records) in Blöcke gleicher Länge eingepasst ?

2.4.1 Interne Speicherung von Relationen und Zugriffspaden

Zur Beantwortung der ersten Frage müssen die Strukturen der konzeptuellen Ebene auf die internen Strukturen (siehe Tabelle 1, Seite 11) abgebildet werden. Umgekehrt müssen die internen Seiten bzw. ihr Inhalt (Bytes) auch wieder als Relation mit Tupeln und Attributwerten interpretiert werden können. Zum Zwecke der Abbildung werden die physischen *Blöcke* auf den Spuren der Magnetplatte in größeren Organisationseinheiten, den *Dateien*, verwaltet. Alternativen sind dabei:

1. Das **DBS** schreibt jede Relation bzw. jeden Zugriffspfad in eine eigene Betriebssystemdatei, die Struktur der Datenbank ist somit im Dateisystem des Betriebssystems sichtbar,
2. das **DBS** verwaltet die Relationen und Zugriffspfade selbst in einer/mehreren Dateien, die Struktur ist im Dateisystem des Betriebssystems nicht sichtbar, oder
3. das **DBS** steuert die Magnetplatte an und arbeitet selbst mit den Blöcken (sogenanntes R*aw Device,* d.h. ein eigenes Dateisystem des **DBS** am Betriebssystem vorbei).

Die 3. Alternative ist von Vorteil

- bei betriebssystemunabhängigen Implementierungen, oder
- wenn die Dateigröße > 4 GB sein soll (Beschränkung bei einem Dateisystem wie FAT16/32), oder
- bei mediumübergreifenden Dateien (die nicht vom Betriebssystem unterstützt werden).

Die Abbildung in beide Richtungen erfolgt durch Metadaten, verwaltet im Data Dictionary des **DBS**. Die Abbildung ist nicht bijektiv, d.h. viele Relationen einer Datenbank (bzw. mehrere logische Dateien) können in eine physische Datei geschrieben werden (die sich wiederum aus Blöcken zusammensetzt, siehe Abbildung 6 auf Seite 12).

Die Art der Abbildung ist abhängig von

- den Organisationsformen und internen Schemata des **DBS**,
- der konkreten Speicherung und Adressierung in physischen Dateien und
- der Kodierung der Bytes.

2.4.2 Einpassung von Tupeln/Datensätzen in Blöcke gleicher Länge (roDBMS)

Die zweite Frage, wie die Daten der internen Ebene auf Seiten/Blöcke der Magnetplatte eingepasst werden, wird durch die Technik des *Blockens* beantwortet: Datensätze fester oder variabler Länge werden auf interner Ebene in die Blöcke des Sekundärspeichers eingepasst. Die Anzahl der Bytes der Blöcke der Platte ist fest vorgegeben.

Ein *Datensatz* ist eine Folge von Datenfeldern mit bestimmtem Typ und bestimmter Länge. Bei *Row Stores* ist der Datensatz das interne Abbild eines Tupels. Das Blocken ist abhängig von der variablen oder festen Feldlänge der Datenfelder. Bei *variabler Satzlänge* ist ein höherer Verwaltungsaufwand beim Lesen/Schreiben gegeben, da die Satzlänge pro Datensatz zu ermitteln ist. Bei *fester Satzlänge* ist höherer Speicheraufwand gegeben, da auch für Datensätze mit kurzen Feldinhalten die maximale Zahl an Bytes gespeichert wird.

Für das Einpassen gibt es 2 *Blockungstechniken* (Abbildung 9):

- *Nichtspannsätze* speichern den Datensatz in maximal einem Block. Ist ein Datensatz zu lang, um im aktuellen Block noch gespeichert werden zu können, so wird er komplett im neuen Block gespeichert.
- *Spannsätze* speichern den Datensatz evtl. in mehreren Blöcken. Bei einem zu großem Datensatz wird ein noch passender Anteil in diesem Block und der Rest im neuen gespeichert.

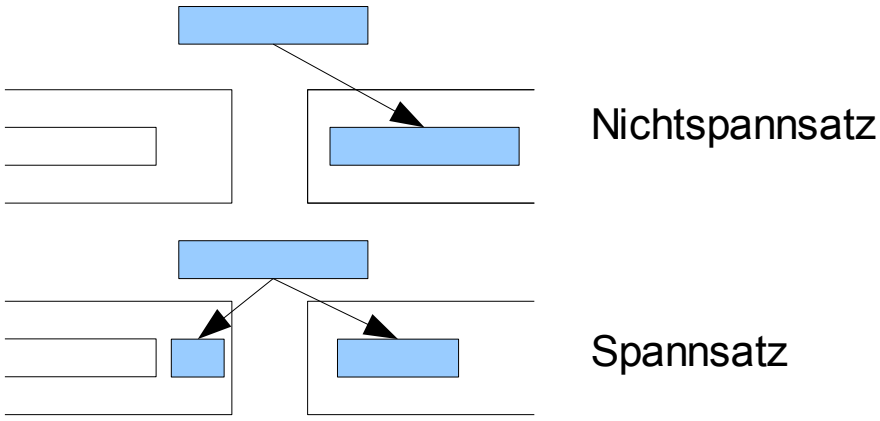

Abbildung 9: Nichtspannsatz und Spannsatz (Seite 100 [SHS05])

2.5 Seiten, Sätze und Adressierung

Nachdem das Prinzip von Blöcken, Seiten und des Blockens erläutert wurde, wird nun der Aufbau der Seiten und die Adressierung der Datensätze (interne Darstellung der konzeptuellen Tupel) innerhalb der Seiten beschrieben mit dem Hinweis, dass die Beschreibungen für die Speicherarchitektur von **roDBMS** gelten. Die Unterschiede bei **coDBMS** werden in Kapitel 2.8 erläutert.

Neue Seiten/Blöcke holt sich das DBMS von der *Freispeicherverwaltung*[4] (doppelt verkettete Liste freier Seiten), vgl. [Seite 100 [SHS05]]. Eine Seite besteht aus einem Header (speichert Vorgänger- und Nachfolgerseite, Seitennummer, Infos über Typ der Sätze, freien Platz), aus Datensätzen (als eigentlichen Informationsträgern) und unbelegten Bytes. Die Adresse eines Datensatzes besteht aus Seiten- bzw. Blocknummer und Offset des Datensatzes innerhalb der Seite, also z.B. (115,142) beim Beispiel in Abbildung 10.

2.5.1 (Daten-)Satztypen

Die in den Seiten gespeicherten Datensätze können nach verschiedenen Kriterien klassifiziert werden, [Seite 102 ff. [SHS05]], z.B.

[4] Die Freispeicherverwaltung ist Teil des Betriebssystems und organisiert den zur Verfügung stehenden Speicher für Programme und Daten. Hier ist allerdings die Verwaltung freier Blöcke gemeint.

1. Ob sie eine feste Positionsadresse auf der Platte oder nicht besitzen (engl. *pinned* / *unpinned records*)[5] ,
2. Ob sie eine *feste* oder *variable* Länge haben, oder
3. Ob sie *lange Attributwerte* (Text, Binärdaten) aufnehmen

Zur Darstellung der Unterschiede zu **coDBMS** ist lediglich der zweite Punkt von Interesse. Die Speicherung von fixierten / unfixierten Sätzen und langer Attributwerte wird in Anhang A.1 erläutert.

2.5.1.1 Sätze fester Länge

In der Data Definition Language (**DDL**) **SQL** existieren Datentypen mit fester und variabler Länge. So deklariert `char(n)` eine Zeichenkette feste Länge, `varchar(n)` hingegen eine Zeichenkette variabler Länge mit maximaler Länge n.

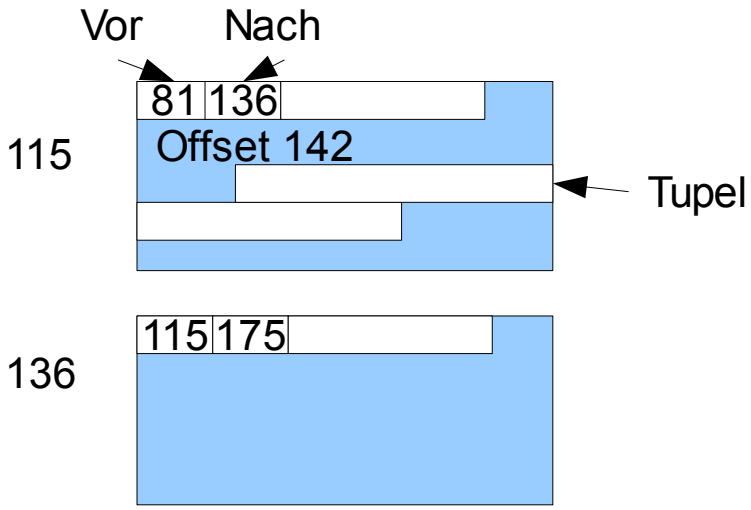

Abbildung 10: Aufbau von Seiten (Abbildung 3.11 aus [SHS05])

Datensätze bestehen bei Datentypen fester Länge aus

- einem Verwaltungsblock mit

[5]Die *Adresse* eines Records besteht entweder aus der Blockadresse und einem *Offset* (Anzahl der Bytes vom Blockanfang bis zum Record) oder wird durch den sogenannten *Tupel-Identifikator* (**TID**) gegeben.

- Angabe des Satztyps, wenn unterschiedliche Satztypen (d.h. Recordtyp bzw. Attributtyp) auf einer Seite gespeichert werden sollen (z.B. in Form eines Zeigers auf das Schema der Relation des Tupels, [Seite 590 [GMUW09]],
- ggf. der Länge des Satzes (um beim Überspringen von Sätzen nicht das Schema konsultieren zu müssen), sowie einem
- Löschbit (gibt an, ob der Inhalt des Datensatzes noch aktuell ist),

- einem Freiraum zur Justierung des Offsets des folgenden Bereichs sowie den
- Nutzdaten (Folge der Datenfelder).

Nachteil von Sätzen fester Länge ist, dass sich wegen fixierter Feldlängen der Speicherplatzbedarf dem längstem Feldinhalt (bezogen auf ein Attribut) anpassen muss (durch Auffüllen mit Nullzeichen).

2.5.1.2 Sätze variabler Länge

Der Unterschied zu Sätzen fester Länge ist hier, dass der Verwaltungsblock des Satzes bei Feldern (und damit Sätzen) variabler Länge die Satzlänge l (d.h. Tupellänge) speichern muss. Hierzu gibt es 2 alternative Strategien:

1. Die Attributlänge al_i steht vor jedem Attribut A_i mit variabler Länge (siehe Abbildung 11), oder
2. der Attributzeiger $ap_1,...,ap_n$ steht für alle variabel langen Datenfelder am Anfang im Zeigerfeld (wodurch bessere Navigation innerhalb des Satzes ermöglicht wird). Der Zeiger ap_{n+1} verweist auf den Anfang des ersten Feldes fester Länge oder auf den nächsten Datensatz (Abbildung 12).

Vorteil von Sätzen mit variabler Länge ist der geringere Speicherplatzverbrauch gegenüber Sätzen mit fester Länge, bei denen nicht benötigter Speicherplatz mit füllenden Nullen belegt wird. Nachteil ist der höhere Verwaltungsaufwand beim Lesen und Schreiben von Datensätzen, da die aktuelle Satzlänge pro Datensatz immer wieder neu ermittelt werden muss.

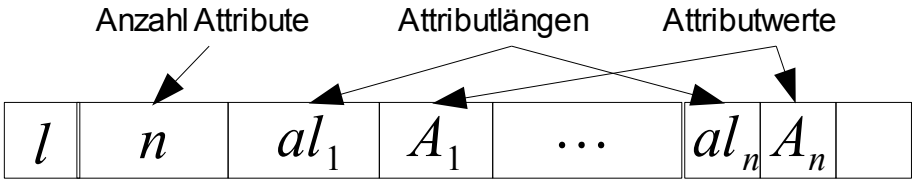

Abbildung 11: Sätze variabler Länge: Variante 1 (Seite 105 [SHS05])

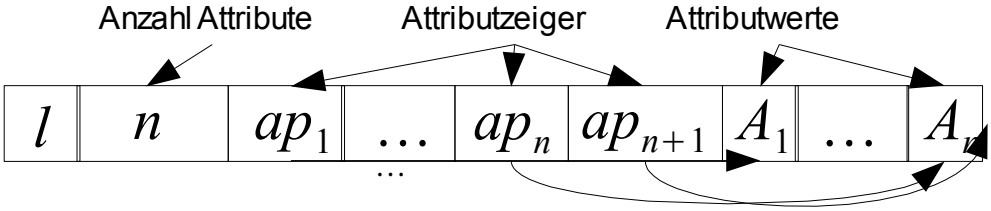

Abbildung 12: Sätze variabler Länge: Variante 2 (Seite 105 [SHS05])

Variabel lange Datenfelder kommen bei allen Datentypen, die *Wiederholgruppen* sind, vor. Eine *Wiederholgruppe* ist eine Datenstruktur für eine Liste von Worten des gleichen Datentyps. Der Begriff der Wiederholgruppe wird bei den Grundlagen zu spaltenorientierten **DBMS** in Kapitel 2.8 noch verwendet werden, deswegen wird er an dieser Stelle erläutert. Beispiele für Wiederholgruppen sind:

- `(char)*`: `varchar (n)` ist Wiederholgruppe mit char als Basistyp
- `(integer)*`: Mengen- oder listenwertige Attributwerte, die im Datensatz selbst denormalisiert gespeichert werden sollen (z.B. Liste von Integern)
- `(pointer)*`: Adressfelder für Indexe, die zu einem Attributwert auf mehrere Datensätze zeigen (Sekundärindex)

2.6 Datenbankpuffer

Nachfolgend soll auf die Bedeutung des Datenbankpuffers eingegangen werden. Auch wenn Sinn und Zweck des Datenbankpuffers als Hauptspeichercache zur Überwindung der Zugriffslücke zwischen Sekundär- und Primärspeicher bereits erläutert wurde, soll an dieser Stelle noch ein weiterer Aspekt des Datenbankpuffers herausgestellt werden, der später in Kapitel 2.8 den Nutzen von **coDBMS** gegenüber **roDBMS** deutlich werden lässt. Alle Lese-/ Schreibvorgänge des **DBS** werden auf Seiten im Puffer ausgeführt. Das Speichersystem als höhere Schicht des **DBMS** fordert bei der Pufferverwaltung eine Seite (z.B. mit benötigten Tupeln) an, die vom Inhalt her identisch mit dem gelieferten Block (oder Blöcken) der

Dateischnittstelle ist. Ist diese noch nicht im Puffer geladen, wird sie nachgeladen. Bei **roDBMS** werden aus Gründen der zeilenorientierten Speicherarchitektur die gespeicherten Tupel komplett mit allen Attributen vom Sekundärspeicher geladen und im Datenbankpuffer zwischengespeichert, auch wenn nicht alle Attribute eines Tupels z.B. zur Beantwortung einer Anfrage benötigt werden.

Aus Performancegründen ist es das Ziel, möglichst alle oder zumindest alle häufig benötigten Daten/Tupel im Puffer zu halten. Das Problem hierbei ist aber die Größenbegrenzung des Datenbankpuffers (Hauptspeicher) gegenüber dem Festplattenspeicher. In der Regel ist es unvermeidbar, nicht mehr benötigte Seiten wieder aus dem Puffer zu verdrängen. Zu diesem Zweck gibt es verschiedene *Seitenverdrängungsstrategien*, diese werden aber nicht näher erläutert, da hier kein Unterschied zu **coDBMS** vorhanden ist. **coDBMS** sind jedoch ein Ansatz, den Datenbankpuffer effizienter zu nutzen. Dies wird in Kapitel 2.8.4 beschrieben.

2.7 Adressierung von Datensätzen (Tupel-Identifier)

Das Problem der physischen Adressierung von Datensätzen bzw. der Änderung ihrer Adressen beim Verschieben ihrer Position und der Lösung durch sogenannte Tupel-Identifikatoren (**TID**, engl. *RowID*) ist nicht Betrachtungsgegenstand der Klassifizierung in Kapitel 3. Es wird deshalb an dieser Stelle lediglich auf Anhang A.2 verwiesen, wo das **TID**-Prinzip beschrieben wird. Es wird jedoch an dieser Stelle darauf hingewiesen, dass dieselben Begriffe (**TID**, *RowID*) bei **coDBMS** mit anderer Bedeutung verwendet werden.

In der recherchierten Literatur zu **coDBMS** wird der Begriff des **TID** bzw. der *RowID* nicht zur physischen Adressierung (Seitennummer, Offset des Listeneintrags) eines Datensatzes (Tupels) aus anderen Datensätzen heraus verwendet (vgl. [Seite 108 [SHS05]]), sondern als logische Adresse, über die die Zuordnung eines bzw. mehrerer Attributwerte zu einem bestimmten Tupel einer Relation sichergestellt wird (eindeutige Tupelnummer eines Tupels innerhalb einer Relation). Dies wird in Kapitel 2.8.1 beschrieben.

2.8 Grundlegendes Prinzip spaltenorientierter DBMS und Unterschiede zu roDBMS

Nachdem in den bisherigen Kapiteln die Grundlagen der physischen Speicherung in **roDBMS** erläutert wurden, soll nun die Umsetzung für **coDBMS** dargestellt werden. Die nachfolgende

Darstellung stammt im Wesentlichen aus [WikiCODBMS] und [GMUW09]. Wie bereits in Kapitel 1.1 und 2.2 dargestellt, speichern **DBMS** ihre (bei relationalen **DBMS** zweidimensionalen) Daten im eindimensionalen Adressraum des darunterliegenden Speichers, bei Sekundärspeicher in einem Block. Die Adressen der Werte einer Relation sind nicht mehr Zeilennummer und Spaltennummer einer Zelle, sondern die Werte sind wie auf einem Band aufeinanderfolgend als Blöcke geschrieben. Die folgenden Unterkapitel beschreiben die Unterschiede zwischen der Speicherarchitektur **roDBMS** und **coDBMS**.

2.8.1 Unterschiede bei der Anordnung der Daten

Der klassische Ansatz, die zeilenorientierte Speicherarchitektur (engl. *Row Store Architecture*), wurde bereits in Kapitel 2.2 dargestellt. Sie wird von den meisten **DBMS**, darunter auch die bekanntesten, Oracle, IBM DB2 und SQL-Server, verwendet (vgl. [Seite 17 [Aba08]]. **roDBMS** speichern den jeweiligen Datensatz (Tupel mit allen Attributwerten) zusammenhängend.

coDBMS speichern hingegen die Attributwerte eines Attributs zusammenhängend. Bei spaltenorientierter Speicherarchitektur wird die in Abbildung 6 gezeigte Relation z.B. in den Datensätzen/Records (alternativ)

1. (4711,5588,6834), (Heuer,Saake,Korn), (DBR,MD,MD) oder
2. ((1,4711), (2,5588),(3,6834)), ((1,Heuer),(2,Saake),(3,Korn)), ((1,DBR),(2,MD), (3,MD))

gespeichert werden, wobei 1. den Fall der gleichen Reihenfolge der Attributwerte und 2. den Fall der Tupel-ID pro Attributwert darstellt.

Die Zuordnung der einzelnen Attributwerte zu einem (logischen) Tupel, welche bei zeilenorientierter Speicherung automatisch gegeben ist, ergibt sich bei Spaltenorientierung also z.B. aus der Reihenfolge der physischen Speicherung oder aus sogenannten Tupel-ID (vgl. Kapitel 2.7), die mit jedem Attributwert zusammen abgelegt werden. Auch eine Zuordnung über den Primärschlüssel der Relation ist denkbar, siehe [Seite 28 [Aba08]]

2.8.2 Unterschiede bei der Verwaltung des Hintergrundspeichers

2.8.2.1 Interne Speicherung von Relationen und Zugriffspaden

Welchen Ansatz ein **coDBMS** bei der physischen Speicherung (Kapitel 2.4.1) wählt (*Betriebssystemdatei* oder *Raw Device*), wird bei den konkreten Implementierungen in Kapitel 3 betrachtet. Grundsätzlich ist zu sagen, dass die eben beschriebenen Alternativen der physischen Dateiwahl (Betriebssystemdatei oder Raw Device) sowohl bei **roDBMS** und **coDBMS** Anwendung finden, hier also keine Unterschiede zwischen beiden Ansätzen vorhanden sind. Wie bei **roDBMS** stellt sich aber auch bei **coDBMS** die Frage, wie die logischen Dateien auf physische Dateien abgebildet werden. Grundsätzlich kann eine Relation, wie in Kapitel 2.8.1 beschrieben, in mehrere Records, einer pro Attribut aufgeteilt werden, wobei diese Records dann hintereinander in Blöcke einer physischen Datei geschrieben werden. Bei Einfügen eines neuen Tupels ist dann aber das Problem, dass zwischen den Records freier Speicherplatz geschaffen werden muss, um die Tupelwerte an mehreren Stellen in der Datei einzufügen. Ein günstigerer Ansatz, den auch einige **coDBMS** wählen, ist die Erzeugung einer eigenen Betriebssystemdatei pro Attribut einer Relation. Auch hier ist der Unterschied, dass bei Einfügen eines neuen Tupels bei **roDBMS** nur eine Einfügeoperation (für das Tupel) nötig ist, bei **coDBMS** mehrere (einer pro Attribut-Record) nötig sind.

2.8.2.2 Einpassung von Tupeln/Datensätzen in Blöcke gleicher Länge

Während bei **roDBMS** der Datensatz das interne Abbild eines Tupels ist, kann man bei **coDBMS** den Datensatz als das interne Abbild eines Attributs der Relation betrachten [Seite 610 [GMUW09]]. Ausgehend von dieser Betrachtungsweise, stellt sich die Frage der Behandlung des Datensatzes als Spannsatz oder Nicht-Spannsatz (Kapitel 2.4.2) bei **coDBMS** nicht, denn ein Datensatz, der ein ganzes Attribut umfasst, übersteigt in der Regel die Kapazität eines Blocks bzw. einer Seite und muss deshalb aufgeteilt werden.

2.8.3 Unterschiede zu roDBMS bei Sätzen fester/variabler Länge

Betrachtet man Datensätze (engl. *Records*) bei **coDBMS** als internes Abbild der Werte eines Attributs, dann handelt es sich gemäß [Seite 610 [GMUW09]] bzw. [Seite 104 [SHS05]] um Sätze variabler Länge, da sich die Größe einer Relation (d.h. Anzahl Tupel und damit auch Attributwerte eines Attributs) dynamisch ändert. Die Attributwerte eines Attributs, die logischerweise gleichen Typs sind, können als Wiederholgruppen (mengen- oder listenwertige Attributwerte) im Sinne von [Seite 105 [SHS05]] betrachtet werden (vgl.

Kapitel 2.5.1.2). Selbst wenn die Datentypen der Attribute einer Relation alle eine feste Länge hätten (z.B. int, char(n) etc.), hätten die Datensätze beim **coDBMS** unterschiedliche Länge im Gegensatz zu den Datensätzen eines **roDBMS**, da der Datensatz für die Spalte mit int-Werten, für die z.B. 4 Byte pro Wert belegt werden, kürzer wäre als der Record für die Spalte mit char(n)-Werten bei einer angenommenen festen Länge von 20 Zeichen (char(20)).

2.8.4 Effiziente Nutzung des Datenbankpuffers

An dieser Stelle soll noch einmal auf den in Kapitel 2.6 beschriebenen Datenbankpuffer und die Unterschiede zwischen **roDBMS** und **coDBMS** eingegangen werden. Der wesentliche Unterschied ergibt sich aus der in Kapitel 2.8.1 beschriebenen Datenanordnung, sodass die Seiten im Puffer bei **roDBMS** tupelorientierte, bei **coDBMS** attributorientierte Datensätze enthalten. Da es das Ziel ist, zur Performancesteigerung möglichst viele benötigte Daten dauerhaft im Puffer vorzuhalten, ist die Frage, welcher Ansatz günstiger ist. Dies ergibt sich aus dem jeweiligen Einsatzszenario des **DBS** und wird in Kapitel 2.10 beschrieben. Bezogen auf den Datenbankpuffer lässt sich sagen, dass der in Kapitel 2.3 beschriebene Performancevorteil des sequenziellen Zugriffs auf benötigte Daten umso mehr zum Tragen kommt, je höher die Dichte der benötigten Informationen in den eingelesenen Seiten ist, da zum einen natürlich umso weniger Seiten eingelesen werden müssen, und zum anderen bezogen auf die Pufferauslastung umso mehr Daten im Puffer dauerhaft ohne Seitenverdrängung vorgehalten werden können. In diese Richtung zielen bekannte Ansätze wie Clusterung, Materialisierte Sichten und Verbundindizes (engl. *Join Index*), die in Kapitel 2.11 erläutert und **coDBMS** gegenübergestellt werden.

2.9 Existierende Implementierungen von coDBMS

Eine Übersicht aus [WikiCODBMS] , [ETH1] und [Hen08] über Implementierungen von **coDBMS** liefert Tabelle 5.

Kommerzielle coDBMS	**Nichtkommerzielle coDBMS**
Sybase IQ	C-Store (kein neues Release seit Oktober 2006)
Vertica (ähnlich C-Store)	FastBit Open Source
Valentina Database	Infobright Community Edition (ICE)
Vectornova/Vectorstar Highspeed Data Engine	LucidDB Open Source
kdb+	MonetDB Academic Open Source Project
Sensage Scalable Log Server (früher Addamark)	Metakit Open Source
1010data's Tenbase Database	S und GNU R Programmiersprache (beinhalten column-oriented Datenstrukturen für statistische Analysen)
DataProbe	
Bigtable (proprietäres DBMS von Google, basierend auf dem ebenfalls proprietärem Google File System)	Hbase (Apache Foundation Projekt), basierend auf Hadoop DFS (ähnlich Bigtable)
EXASolution	Hypertable (ähnlich BigTable)
Skytide XOLAP Server	The Cassandra Project (Facebooks verteiltes Speichersystem ähnlich BigTable)
SuperSTAR from Space-Time Research	EaseDB
ParAccel Analytic Database	
Infobright (ehemals Brighthouse) Data Engine in MySQL	
RC21 Commercial Open Source Project	
Xplain Semantic Database	
SAP BI Accelerator	

Tabelle 5: Übersicht über spaltenorientierte DBMS

Im Verlauf der Diplomarbeit sind weitere Implementierungen zur Übersicht über **coDBMS** ([WikiCODBMS]) hinzugekommen, nämlich „Lemur Bitmap Index C++-Library“ und „Ingres Vectorwise“. Diese konnten aufgrund der verbliebenen knappen Bearbeitungszeit aber nicht mehr bei der Klassifizierung in Kapitel 3 berücksichtigt werden.

2.10 OLTP vs. OLAP

Wie bereits in Kapitel 1 erwähnt, sind **coDBMS** ein zu den herkömmlichen **roDBMS** alternativer Ansatz zur Performanceoptimierung bei **OLAP**-Anwendungen, wohingegen letztere nach herrschender Literaturmeinung für **OLTP**-Anwendungen die geeignetere Lösung darstellen. Dementsprechend werden auch viele der in Tabelle 5 aufgeführten **coDBMS**, wie in Kapitel 3 noch gezeigt wird, von den Herstellern als **OLAP**-Systeme (genauer gesagt Data Warehouses) deklariert. Nachfolgend soll deshalb ein kurzer Überblick über die Begriffe **OLTP** und **OLAP** und ihre wesentlichen Unterschiede gegeben werden.

2.10.1 OLTP

Der Begriff des *Online Transaction Processing* (**OLTP**) ist das Gegenstück zum *Online Analytical Processing* und beschreibt eine Nutzungsform von Datenbanken bzw. eine Klasse von Datenbankanwendungen, bei der die Verarbeitung von Transaktionen[6] im Vordergrund steht (vgl. [Seite 620 [SSH08]] und [WikiOLTP]). Jede Transaktion beinhaltet dabei ein oder mehrere eingefügte, geänderte oder gelöschte Tupel in 1-n Relationen. Im Mittelpunkt steht hier die Verarbeitung der Daten täglich erfasster Geschäftsvorfälle bzw. das operationale Tagesgeschäft wie z.B. Personalwirtschaft, Lagerbestandsverwaltung, Kundenverwaltung sowie Ein- und Verkauf, Kreditoren- und Debitorenverwaltung und Buchführung. Online Transaction Processing bedeutet dabei, dass die bei den Geschäftsvorfällen gesammelten Daten nicht in (nächtlichen) Batchläufen, sondern aktuell (online) verarbeitet und gebucht werden. Charakteristisch für das **OLTP** sind wenige Schreib- und Lesezugriffe pro Transaktion.

2.10.2 OLAP

Im Gegensatz zum **OLTP** steht beim *Online Analytical Processing* (**OLAP**) eine vergleichende oder auswertende (analytische) Verwendung der Daten im Vordergrund, bei

[6] Siehe Anhang B (Glossar)

der auf große Datenmengen (d.h. nicht nur auf wenige, sondern auf eine Vielzahl von Datenbankeinträgen) lesend zugegriffen wird (vgl. [Seite 7 [BG09]]).

Grundlage des **OLAP** ist in der Regel der in einem *Data Warehouse* integrierte, historische Gesamtdatenbestand einer Organisation (vgl. [Seite 105 [BG09]]), der z.B. jeden Verkauf jedes beliebigen Produkts in jedem Geschäft erfasst hat.

OLAP beschäftigt sich mit der Aufbereitung, Gruppierung, Aggregation und Auswertung operativer Daten eines längeren Zeitraums, um Muster und Trends zu erkennen und daraus Strategien für die Zukunft zu entwickeln ([Seite 465 [GMUW09]]) und dabei z.B. mit Fragestellungen wie „Wie hat sich der Umsatz einer Produktgruppe im Vergleich zum Vormonat geändert ?“, „Wie zuverlässig haben Personen der Altersgruppe x und Einkommen y in Bundesland z ihre Kredite abbezahlt ?“ (zur Abschätzung der Kreditwürdigkeit) oder „Wie gut schnitt eine bestimmte Verkaufskampagne im Weihnachtsgeschäft 2007 im Vergleich zu 2006 ab ?“. Solche Fragen bzw. Anfragen, die im Hinblick auf unternehmerische Entscheidungen (nach Bewertung der Anfrageergebnisse) gestellt werden, werden als entscheidungsunterstützende Anfragen (engl. D*ecision-Support Queries*) bezeichnet. Im Gegensatz zu den Methoden des *Data Mining*, wo durch Anwendung mathematischer Methoden nach (bisher unbekannten) Mustern im Datenbestand gesucht wird, muss der Anfragesteller bei **OLAP** vor der Untersuchung wissen, welche Anfrage er an das System stellen möchte. Die von ihm in diesem Zusammenhang aufgestellte Hypothese wird dann durch das Analyseergebnis entweder bestätigt oder abgelehnt. **OLAP** zählt deshalb zu den hypothesengestützten Analysemethoden (vgl. [WikiOLAP]).

Der Begriff OLAP wurde 1993 durch zwölf Regeln von Codd ([CoCS93]) zu Anforderungen an **OLAP**-Produkte definiert, welche er 1995 um weitere sechs Regeln erweiterte ([Seite 105-108 [BG09]]). Die insgesamt achtzehn Regeln sind wegen ihre hohen Detailgrades, der für die Klassifizierung nicht benötigt wird, in Anhang A.3 aufgeführt.

Eine andere, bekannte Anforderungsliste ist die Definition *Fast Analysis of Shared Multidimensional Information* (**FASMI**) von Pendse ([Pen95], [Pen97]). Sie fasst die Regeln von Codd zu fünf Schlüsselwörten *Geschwindigkeit*, *Analysemöglichkeit*, *Sicherheit*, *Multidimensionalität* und *Kapazität* zusammen, vgl. [Seite 624 [SSH08]]:

- *Fast*: Die Zugriffsdauer (z.B. Dauer einer Anfrage) muss im Sekundenbereich liegen.

- *Analysis*: Ein OLAP-System muss analytische und statistische Funktionalitäten anbieten. Ad-hoc-Anfragen müssen ohne Programmieraufwand realisierbar sein.
- *Shared*: Ein OLAP-System soll für Mehrbenutzerbetrieb ausgelegt sein und entsprechende Zugriffskontrolle, Transaktionssynchronisation (für die Phasen der Aktualisierung) und Recovery bieten.
- *Multidimensional*: Daten müssen mehrdimensional strukturierbar sein mit voller Unterstützung der Dimensionshierarchien.
- *Information*: Alle benötigten Daten sollen dem Anwender transparent zur Informationsgewinnung Verfügung stehen.

2.10.3 Unterschiede im Überblick

Nach [Seite 9 [BG09]] bzw. [Seite 621 [SSH08]] lassen sich die Unterscheidungsmerkmale zwischen **OLTP-** und **OLAP**-Systemen in die Bereiche *Anfragen*, *Daten* und *Anwender* klassifizieren.

Bei *Anfragen* lesen, erzeugen, modifizieren und löschen **OLTP**-Anwendungen Daten in kurzen Transaktionen (auf wenigen Datensätzen), während OLAP-Systeme Informationen aus (in Data Warehouses integrierten) Daten durch lange Lesetransaktionen auf Millionen von Datensätzen mit komplexen (Ad-hoc-)Anfragen gewinnen [Seite 1153 [CB05]], [Seite 9 [BG09]] und [Seite 621 [SSH08]]). Diese analyseorientierten Anfragen arbeiten in der Regel auf mehreren Relationen, aus denen Daten nach bestimmten Kriterien selektiert werden, und diese Kriterien sind in der Regel spaltenbasiert. Die Unterscheidungsmerkmale sind in Tabelle 6 aufgeführt.

Daten eines **OLTP**-Systems stammen aus einer Produktionsumgebung mit einer operativen Datenbank, Daten eines **OLAP**-Systems sind aus in der Regel mehreren operativen Datenbanken abgeleitete Daten. Transaktional verarbeitete Daten sind zeitaktuell, nicht abgeleitet und dynamisch, d.h. von vielen Modifikationen betroffen. Die Daten in **OLAP**-Systemen hingegen sind, da für Analysezwecke benötigt, konsolidiert, historisiert, integriert und stabil. Das Volumen der Daten in **OLAP**-Systemen liegt wegen der Datenintegration im Terabyte und nicht im Gigabyte-Bereich (Tabelle 7).

Anfragen	OLTP	OLAP
Fokus	Lesen, Schreiben, Modifizieren, Löschen	Lesen, periodisches Hinzufügen
Transaktionsdauer und -typ	Kurze Lese- und Schreibtransaktionen	Lange Lesetransaktionen (analysegetrieben)
Anfragestruktur	Einfach strukturiert, wiederholt dieselben Anfragen	Komplex, ad hoc
Datenvolumen einer Anfrage	Wenige Datensätze	Viele Datensätze
Datenmodell	Anfrageflexibles Datenmodell	Analysebezogenes Datenmodell
Entscheidungsunterstützung	Unterstützung von Entscheidungen im Rahmen täglicher Geschäftsprozesse	Unterstützung strategischer Entscheidungen

Tabelle 6: Anfragecharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])

Daten	OLTP	OLAP
Datenquellen	Meist eine	Mehrere
Eigenschaften	Nicht abgeleitet, zeitaktuell, autonom, dynamisch	Abgeleitet, konsolidiert, historisiert, integriert, stabil
Datenvolumen	MB bis GB	GB bis TB
Zugriffe	Einzelne Tupel	Bereichsanfragen
Indexierung	Änderungseffizient	Leseeffizient

Tabelle 7: Datencharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])

Anwender eines **OLTP**-Systems sind in der Regel Sachbearbeiter, die Daten erfassen oder abfragen, bei **OLAP**-Systemen sind dies Manager, Controller oder Analysten, die Daten in verdichteter Form zur Entscheidungsunterstützung benötigen. Die Anwenderzahlen sind bei OLAP-Systemen, wie man sich leicht vorstellen kann, also bedeutend geringer. Aufgrund der Komplexität der Anfragen liegt die erwartete Antwortzeit von Anfragen bei **OLAP**-Systemen zwischen Sekunden und Minuten (wenn nicht sogar Stunden), bei **OLTP**-Systemen höchstens im Sekundenbereich (gefühlt „sofort“), vgl. Tabelle 8.

Anwender	OLTP	OLAP
Anwendertyp	Ein- und Ausgabe durch Sachbearbeiter	Auswertungen durch Manager, Controller und Analysten
Anwenderzahl	Sehr viele	Wenige bis einige hundert
Antwortzeit	Millisekunden – Sekunden	Sekunden bis Minuten/Stunden

Tabelle 8: Anwendercharakteristika von OLTP- und OLAP-Systemen (aus [BG09] bzw. [SSH08])

2.10.4 Konsequenzen für die Speicherarchitektur eines DBMS

Entsprechend des Themas der Diplomarbeit stellt sich die Frage, welche Speicherarchitektur (zeilen- oder spaltenorientiert) für welchen Anwendungszweck (**OLTP** oder **OLAP**) eher geeignet ist.

Datenbanksysteme (u.a. Oracle, IBM DB2, Microsoft SQL-Server) wurden ursprünglich für die transaktionale Bearbeitung von Geschäftsvorfällen konzipiert und mit zeilenorientierten Speicherarchitekturen versehen. Bei analyseorientierten Anwendungen, deren Ziel ist, große Datenmengen zu scannen und daraus Informationen für Entscheidungen und Planungen zu gewinnen, ist diese Speicherarchitektur nicht unbedingt die optimale.
Unabhängig von der Problematik, für **OLTP** und **OLAP** unterschiedliche Datenbestände zu pflegen, unter anderem weil die lange dauernden analyseorienten Anfragen die **OLTP**-Schreibzugriffe sonst blockieren würden und für OLAP konsoldierte, integrierte Daten benötigt werden, ist die Frage bei der Wahl der Speicherarchitektur, wie nah die Daten, auf die zugegriffen wird, in der jeweiligen Speicherhierarchie beieinander liegen (vgl. Kapitel 2.3).
Denn Ziel bei der Performanceoptimierung von Datenbankoperationen (sowohl schreibend als auch lesend) ist das sequenziellen Lesen der benötigen Blöcke (wegen vergleichsweise hoher S*eek* und L*atency time)* und generell die Minimierung von Block- bzw. Seitenzugriffen, vgl. Kapitel 7.1 [SHS05]
Sind die Daten, auf die zugegriffen wird, hauptsächlich (einzelne) Tupel, so ermöglicht die zeilenorientierte Speicherung, bei der die Tupelwerte aneinander angrenzend in den Blöcken des Sekundärspeichers bzw. in den Seiten des Datenbankpuffers liegen (vgl. Kapitel 2.6), kurze Zugriffszeiten. Bei diesem Ansatz reicht ein suchender Zugriff im Sekundärspeicher, um alle Felder eines Records (Tupelwerte) zu sichern oder zu lesen (wenn man verschiedene mögliche Dateiorganisationsformen und Zugriffspfade zunächst einmal außer Acht lässt).
Die zeilenorientierte Speicherarchitektur ist deshalb geeignet für Applikationen, die regelmäßig Schreibzugriffe (z.B. Einfügen vieler neuer Tupel) oder lesenden Zugriff auf wenige, spezielle Tupel erfordern, eben typische **OLTP**-Anwendungen.

Wird jedoch wie bei **OLAP** auf ganze Attribute der Relationen zugegriffen, so ist die Zugriffszeit (die sich nach der örtlichen Verteilung der einzulesenden Seiten und ihrer Anzahl richtet) bei spaltenorientierter Speicherung am effizientesten, siehe z.B. [Seite 17 [Aba08]]. Neben der physikalischen Anordnung der Daten sind natürlich noch andere

Optimierungsmöglichkeiten wie z.B. die Anpassung des *Query Processors* an die spaltenorientierte Speicherarchitektur möglich, dies wird in Kapitel 2.14 erläutert.
Bei einer spaltenorientierten Speicherarchitektur werden die Werte jedes Attributs aneinander angrenzend in die Blöcke des Sekundärspeichers geschrieben (Kapitel 2.8.1). Bei Anfragen, die sämtliche Werte von 1-n Attributen auswerten, muss das **DBS** nur die Bereiche des Sekundärspeichers lesen, in denen die für die Anfrage relevanten Attribute abgelegt sind. Vor allen Dingen können sämtliche Werte eines Attributs mit nur einem Lesezugriff eingelesen werden (vorausgesetzt, der Datenbankpuffer ist ausreichend groß). Das Einlesen irrelevanter Daten, in diesem Fall nicht benötigter Attribute, wird vermieden, was I/O-Zeit (weniger Blockzugriffe, vgl. [Seite 610 [GMUW09]], und Platz im Datenbankpuffer spart.

Geht man davon aus, dass in einer zeilenorientierten Speicherarchitektur sämtliche Attribute der Records eingelesen werden, obwohl nur wenige benötigt werden, so hat dies grundsätzliche Performancenachteile gegenüber einer spaltenorientierten Architektur, die in der bereits in Kapitel 2.3 erwähnten Zugriffslücke und der Begrenztheit des Datenbankpuffers (Kapitel 2.6 und 2.8.4) begründet ist.

Genauso muss natürlich erwähnt, dass das (transaktionale) Schreiben von Tupeln (Inserts / Updates) und dass tupelorientierte Anfragen bei einer spaltenorientierten Speicherarchitektur höhere I/O-Kosten verursacht, da sich die Attributwerte eines Tupels auf mehr Records[7] (und damit mehr Seiten) verteilen als bei zeilenorientierter. Dies gilt nicht für Inserts/Updates vieler oder aller Tupel, die sich nur auf ein Attribut beziehen. Hier ist wiederum der spaltenorientierte Ansatz günstiger.

[7]Ein Record kann z.B. sämtliche Werte eines Attributs umfassen

2.11 coDBMS vs. leseoptimierte roDBMS

In diesem Kapitel wird auf die möglichen Zugriffsstrukturen, d.h. *Dateiorganisationsformen* und *Zugriffspfade (Indizes)*, sowie weitere Strukturen zur Steigerung der Leseperformance von **DBMS** eingegangen. Dies erfolgt zum einen, weil bestimmte in **roDBMS** bereits etablierte Ansätze auch in **coDBMS** eingesetzt werden und hier vorgestellt werden sollen, zum anderen um zu zeigen, wo der Nachteil von Ansätzen zur Performanceoptimierung für **roDBMS** (Cluster, Materialisierte Sichten) gegenüber **coDBMS** liegt, deren Ansatz es gerade ist, auf die letztgenannten Hilfsstrukturen zu verzichten. Die folgenden Darstellungen beziehen sich im Wesentlichen auf [Seite 139 ff. [SHS05]], [Lue07], [Kue09] und [Appendix C, [CB05]].

2.11.1 Zugriffsstrukturen

Über Zugriffsstrukturen erfolgt der (effiziente) Zugriff auf die (in den Blöcken des Sekundärspeichers) gespeicherten Datensätze. Anhang A.4 klassifiziert die Zugriffsstrukturen. Die Datei- und Indexverwaltung wird durch den Record- und Index-Manager im Speichersystem des **DBMS** (Abbildung 6 auf Seite 12) realisiert.

Über die Systempufferschnittstelle fordert das Speichersystem Seiten an (siehe Abbildung 5 auf Seite 10) und interpretiert diese als interne Datensätze. Das Zugriffssystem wiederum abstrahiert nach Darstellung in der allgemeinen Literatur, die sich auf **roDBMS** bezieht, von der konkreten Realisierung durch externe Datensätze. An dieser Stelle wird schon einmal darauf hingewiesen, dass dieser Grundsatz bei **coDBMS** im Prinzip auch noch gültig ist, da von den Seiten der Pufferverwaltung abstrahiert wird, jedoch nicht zwangsläufig die Spalten bereits im Speichersystem zu Tupeln zusammengefügt werden. Dies ist abhängig vom implementierten Query Processor, wie im noch folgenden Kapitel 2.14 beschrieben wird.

2.11.2 Dateiorganisationsformen

Eine Dateiorganisationsform ist eine Form der Speicherung der internen Relation. Grundlegende Organisationsformen für Dateien sind die *Heap*-, die *Hash*- und die *sequenzielle* Dateiorganisation.

2.11.2.1 Heap-Organisation

Die *Heap-Datei* ist die einfachste Dateiorganisationsform. Die Datensätze werden unsortiert, nur entsprechend ihrer zeitlichen Reihenfolge ihrer Aufnahme, auf einem „Haufen“ gespeichert.

Das Einfügen neuer Datensätze (insert) geschieht mit geringem Aufwand (*O(1)*), das Suchen oder Löschen (lookup, delete) hingegen mit großem (*O(n)*), da es das Durchsuchen aller Datensätze erfordert (sogenannter *Full-Table-Scan*).

2.11.2.2 Sequenzielle Organisation

Unter der sequenziellen Dateiorganisation versteht man die sortierte Speicherung einer Relation nach einem Schlüsselattribut. Dies erfordert beim Einfügen neuer Datensätze ggf. das Verschieben bereits vorhandener Datensätze, was gelöst wird durch das nur teilweise Füllen der Seiten mit Datensätzen, um so genug Platz für neu einzufügende zu haben. Üblich ist, wie noch gezeigt wird, eine *indexsequenzielle Organisationsform*, d.h. eine sortierte Speicherung der Relation mit einem (dünnbesetzten) Primärindex über dem Primärschlüssel.

2.11.2.3 Hash-Organisation

Idee der Hash-Organisation ist es, die Speicheradresse eines Datensatzes durch eine Hashfunktion (z.B. Modulo- bzw. Divisions-Rest-Funktion) über einer bestimmten Attributkombination zu berechnen, vgl. [Seite 10 [Kue09]]. Dabei kann es zu Kollisionen kommen, d.h. dass für verschiedene Attributwerte dieselbe Speicheradresse berechnet wird. Durch die Kollisionen kann die Seite, die diese Tupel speichern soll, überlaufen. In einem solchen Fall wird eine neue Seite aufgenommen und mit der Ursprungsseite verkettet. Mit der Hashfunktion können aufgrund der schnellen Adressberechung für ein Tupel einzelne Datensätze effizient gefunden werden bzw. neu eingefügt werden.

2.11.3 Zugriffspfade

Ein *Zugriffspfad* ist jede Zugriffsstruktur, die über eine Dateiorganisationsform hinausgeht, z.B. eine Indexdatei über einer internen Relation. Eine Indexdatei besteht aus Einträgen (K,K↑) mit

- K: Suchschlüssel bzw. Zugriffsattribut (Wert eines Primär- oder Sekundärschlüssels) sowie

- K↑: Datensatz oder Verweis auf einen Datensatz..

Wenn K↑ der Datensatz selbst ist, dann ist der Zugriffspfad zur Dateiorganisationsform entartet (da interne Tupel nach K sortiert abgelegt werden). Ist K↑ ein Verweis, dann ist er entweder die Adresse eines internen Tupels oder eine Liste von Tupeladressen. Bei einem (eindeutigen) Primärschlüssel K gibt es jeweils nur ein K↑, bei einem (nicht eindeutigen) Sekundärschlüssel gibt mehrere Einträge $(K,K_1↑),\ldots,(K,K_n↑)$ für denselben Wert K, d.h. derselbe Wert kann in verschiedenen Datensätzen vorkommen. K↑ kann jedoch auch als Liste von Tupeladressen implementiert werden, in diesem Fall gibt es auch bei Sekundärschlüsseln nur jeweils ein Paar (K,K↑).

Bei sehr großen Indizes kann man diese zur Effizienzsteigerung des Zugriffs auch mehrstufig gestalten. Ein *Index zweiter Stufe* z.B. ist die Speicherung der Indexdatei selbst in einer der möglichen Dateiorganisationsformen, wobei die Datei nochmal mit einem Zugriffspfad versehen wird.

2.11.3.1 B-Baum

Der B-Baum [BM72] dient in Datenbanksystemen als Zugriffsverfahren für Primär- und Sekundärindizes. Er ist ein dynamischer, nicht binärer, balancierter Indexbaum. Balanciert bedeutet, dass alle Pfade von der Wurzel zu den Blättern gleich lang sind. Das Balancekriterium ist, dass jeder Baumknoten mindestens m und maximal 2m Elemente (Schlüsselwerte) enthalten muss. Die Konstante m ist die Ordnung des B-Baums. In den Knoten sind die (K,K↑)-Paare gespeichert. Speichert man statt dieser Paare ganze Datensätze, so wird der B-Baum zur Dateiorganisationsform. Der B-Baum ist ein Suchbaum, d.h. er speichert die Schlüsselwerte K_n sortiert ab. Neben den Indexeinträgen sind in den Knoten Zeiger auf die Kindknoten gespeichert. Die Anforderungen an einen B-Baum sind:

1. Jede Seite enthält höchstens 2m Elemente
2. Jede Seite, außer der Wurzelseite, enthält mindestens m Elemente
3. Jede Seite ist entweder eine Blattseite oder hat i+1 Nachfolger, wenn i die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.

Die Knoten des B-Baums sind von ihrer Kapazität her zugeschnitten auf die Seitenstruktur des DBMS.

Das Suchen von Schlüsselwerten erfolgt ausgehend von der Wurzel und endet spätestens auf der Blattebene. Der gesuchte Schlüsselwert k wird mit den Schlüsselwerten des Wurzelknotens verglichen. Wird der Wert gefunden, ist die Suche beendet. Die Suche wird dabei für alle Werte $K_1,K_2,...,K_m$ des Wurzelknotens wie folgt ausgeführt:

- Falls $k < K_1$, folge dem ersten Zeiger
- Falls $K_i < K < K_{i+1}$, folge dem i+1-ten Zeiger
- Falls $K_n < k$, folge dem letzten Zeiger

Der Vorgang wiederholt sich auf dem Kindknoten, wenn der Schlüsselwert dort auch nicht gefunden wird. Abbildung 13 zeigt einen B-Baum.

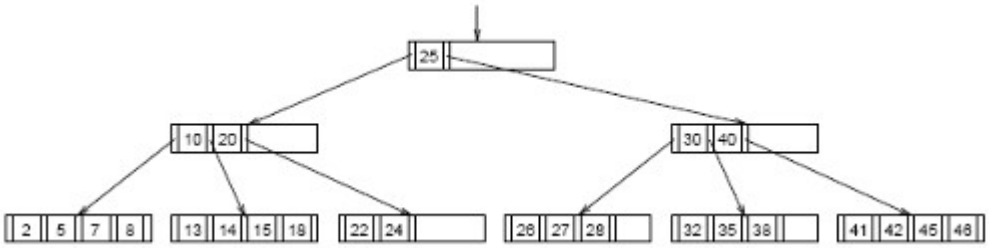

Abbildung 13: B-Baum der Ordnung 2 mit 3 Stufen

2.11.3.2 Hash-Index

Das der Hash-Dateiorganisation zugrundeliegende Prinzip wird auch bei der Hash-Indizierung, also dem Einsatz des Hashverfahrens für Zugriffspfade, in diesem Fall Sekundär-Indizes[8], angewandt. Ein Hash-Index speichert die (K,K↑)-Einträge eines Indexes in einer hash-organisierten Struktur (*Hash-Table*). Er wird über einem oder mehreren anderen Attributen als dem Primärschlüssel gebildet. Beim Hashen werden die Datensätze anhand des gewählten Schlüsselattributs sogenannten *Buckets* (deutsch Eimer) zugeordnet, die aus einer oder mehreren (einzeln adressierbaren) Seite(n) bestehen. Die Zuordnung der Paare (K,K↑) zu den Buckets erfolgt über eine Hashfunktion *h(K)*. Der (feste) Wertebereich der Hashfunktion gibt die möglichen Bucketadressen vor. Da der Wertebereich begrenzt ist, kann es auch hier für zwei Hashwerte $h(K_1)$ und $h(K_2)$ eine Kollision geben. In diesem Fall wird eine Überlaufseite erzeugt, die mit dem jeweiligen Bucket B_i=h(K) verknüpft wird. Das Problem bei Kollisionen ist, dass sich in einer Bucketadresse sehr viele Verweise auf Datensätze befinden können, von denen aber nur einer der gesuchte ist. Ein Lösungsansatz zur Behebung

[8]Denn der Hash-Primärindex entspricht bereits der Hash-Dateiorganisation.

Kollisions-Problematik sind die dynamischen Hashverfahren (im Gegensatz zu dem gerade beschriebenen statischen), hierzu wird auf [Seite 184 ff. [SHS05]] verwiesen. Das Hashverfahren ist in Anhang A.5.1 illustriert.

2.11.3.3 Bitmap-Index

Bitmap-Indizes ([NQ97]) werden insbesondere bei dünnbesetzten Attributen, d.h. solchen mit wenigen verschiedenen Attributwerten, verwendet [Seite 1284 [CB05]]. Sie werden sowohl in **roDBMS** als auch in **coDBMS** als Zugriffspfad eingesetzt und ähneln der bei **coDBMS** verwendeten Bitvektor-Komprimierung, die im Folgekapitel 2.12 beschrieben wird. Sie speichern für jedes Attribut einen Bitvektor, der für jedes Tupel anzeigt, ob es einen bestimmten Wert enthält. Jedes Bit (0 oder 1) stimmt dabei mit einer bestimmten Tupel-ID überein. Bei wenigen verschiedenen Werten verbrauchen die Bitmap-Indizes wenig Speicher und können darüber hinaus, da dünnbesetzt (viele Null-Werte) gut komprimiert werden, wie im Folgekapitel 2.12 noch gezeigt wird. Mit Bitmap-Indizes können Selektionen mit mehreren Prädikaten durch eine boolesche AND- bzw. OR-Verknüpfung effizient ausgeführt werden. **CoDBMS** sind bei der Anwendung von Bitmap-Indizes gegenüber **roDBMS** klar im Vorteil denn sie haben entweder die Daten selbst bereits als Bitvektor gespeichert (bei Bitvektorkompression) oder können zumindest bei Anfragebearbeitung diese mit weniger I/O-Aufwand dynamisch erzeugen und haben dafür entsprechende Operatoren. Hierauf wird in Kapitel 3 eingegangen. In Anhang A.5.2 ist die Bitmap-Indizierung beispielhaft beschrieben.

2.11.3.4 Verbundindex

An dieser Stelle soll auch auf den Verbundindex (engl. *Join Index*) eingegangen werden, der zumindest von einem der in Tabelle 5 genannten **coDBMS** benutzt wird. Ein Verbundindex ist ein Index über Attribute von zwei oder mehreren Relationen derselben Domäne (bzw. desselben Datentyps, [Seite 1284 [CB05]]. In Anhang A.5.3 wird der Verbundindex beispielhaft erklärt. Er ist bei typischen Anfragen in Data Warehouses vorteilhaft, wenn man Fakten über zusammenhängende Daten herausfinden möchte (hier: Wieviele Objekte kommen aus Städten mit Zweigbüros ?), siehe auch [Seite 332 [SHS05]], wo der Verbundindex als Zugriffsstruktur für Star Joins (Erläuterung im Anhang) erwähnt wird. Der Verbundindex ist ein vorberechneter Verbund und erspart das Berechnen des Verbundes mit den bekannten (und zeitaufwändingen) Methoden bei jeder erneuten Anfrage. Nachteil ist jedoch wie bei

allen Zugriffspfaden, dass sie vorberechnet sind (materialisiert), die Anfrage also im Vornhinein bekannt sein muss und nicht ad hoc formuliert werden kann. Außerdem kostet die Wartung dieses Indexes zusätzlichen Speicherplatz und zusätzliche I/O-Operationen. Oracle kombiniert den bereits vorgestellten Bitmapindex mit dem Verbundindex zum *Bitmap Join Index*. Nachteil ist jedoch unter anderem auch hier, dass der Index nicht automatisch, sondern nur per **DDL**-Kommando angelegt wird (siehe z.B. [Burl09]). In Kapitel 3.1.1 wird gezeigt, wie Verbundindizes zur Unterstützung von Ad-hoc-Anfragen bei vertikaler Partitionierung eingesetzt werden.

2.11.4 Weitere Ansätze zur Performancesteigerung

2.11.4.1 Clustering

Unter *Clustering* (deutsch Ballung) versteht man die zusammenhängende physische Speicherung von Datensätzen, die in typischen Anfragen zusammen benötigt werden. Beispiele sind das Clustering nach Schlüsselattributen zur Unterstützung von Bereichsanfragen und Gruppierungen (Ablage der Datensätze in Sortierreihenfolge) und das Clustering (Ballung) basierend auf Fremdschlüsselattributen zur Unterstützung von *Verbundanfragen*.

2.11.4.2 Materialisierte Sichten

Sichten sind normalerweise logische Beschreibungen von Relationen [Seite 359 [GMUW09]]. Materialisierte Sichten (engl. *Materialized Views*) hingegen sind physisch abgelegte Sichten. Diese Materialisierung kann zur Performanceverbesserung bei häufig benutzen Sichten von Vorteil sein, jedoch muss die materialisierte Sicht bei Änderung der entsprechenden Basisrelationen ebenfalls angepasst werden.

2.11.5 Klassische Optimierungstechniken vs. Column-oriented Ansatz

Durch Indizes, Clustering und Materialisierte Sichten soll der Zugriff auf die Daten beschleunigt werden. Das Problem bei **OLAP**-Anwendungen ist aber, dass die Anfragen, die an ein **DBMS** gestellt werden, nicht immer vorhersagbar sind, da sich die Anforderungen ändern und gerade die nicht vorhersagbaren Recherchen (bzw. die der Firma noch nicht bekannten) die sind, deren Ergebnisse den größten Nutzen bringen und somit auch Attribute

umfassen können, für die kein Index, kein Cluster und keine materialisierte Sicht existiert (was dann einen full table scan erfordert).

Zusätzlich verbrauchen die genannten Hilfsstrukturen weiteren Speicherplatz. Möchte man vorausschauend eine Vielzahl zukünftig möglicher Anfragen abdecken, kann der durch die Basisrelationen verbrauchte Speicherplatz durch den Speicherplatz der Hilfsstrukturen um ein Vielfaches übertroffen werden.

coDBMS sind ein Ansatz, der Ad-hoc-Anfragen für **OLAP** ohne die Nutzung solcher Hilfsstrukturen ermöglichen soll. Durch die spaltenorientierte Speicherung ist die Lokalität der Daten, die zu effizientem I/O und effizienter Puffernutzung führt, für den genannten Anwendungszweck automatisch automatisch gegeben.

2.12 Kompressionstechniken „klassischer“ und spaltenorientierter DBMS

Datenkompression ist in heutigen IT-Anwendungen nicht mehr wegzudenken. Sie wird genutzt für Audio-, Bild- und Videodaten, für das Speichern von Backups und für Archive wie Zip-, Jar- und War-Dateien. In den vergangenen Jahren ist die Datenkompression beim Einsatz von **DBMS** immer mehr in den Fokus gerückt. Insbesondere **coDBMS** setzen Kompressionstechniken quasi standardmäßig ein, kaum eine der in dieser Arbeit betrachteten Implementierungen verzichtete auf Datenkompression.
Sowohl in spalten- (u.a. C-Store, Vertica, Sybase IQ und Infobright) als auch in zeilenorientierten (u.a. Oracle ab Version 9i, vgl. [OraCompr] und [OraCompr2]) DBMS werden Kompressionstechniken zur sowohl zur Performancesteigerung als auch zum Einsparen von benötigtem Speicherplatz eingesetzt (siehe u.a. [Seite 21 ff. [Aba08]] oder [Seite 1 [WKHM]]). Die Performancesteigerung ergibt sich dabei aus den verringerten Schreib- und Lesezugriffen (weil mehr Informationen in weniger Seiten enthalten sind), gleichzeitig darf der CPU-Overhead für Kompression und Dekompression diesen Performancegewinn nicht aufwiegen (vgl. [Seite 2 [WKHM]] und [Seite 49 [Aba08]]). Ein Kostenmodell muss beide Faktoren berücksichtigen (vgl. [Seite 11 [Aba06]]). Ein Ansatz hierfür ist z.B. die Entwicklung eines Query Processors, der auf komprimierten Daten arbeiten kann, und das Vorhalten der komprimierten Daten im Datenbankpuffer (und Dekodierung der Daten erst, wenn wirklich benötigt).

Ein allgemeiner Überblick über das Thema Datenkompression inklusive Klassifizierung (u.a. in verlustfreie und verlustbehaftete Kompression) wird in Anhang A.6.1 gegeben.
Im nachfolgenden sollen nur Methoden zur verlustfreien Kompression vorgestellt werden, da die recherchierten Literaturquellen zur Datenkompression in DBMS (insbesondere in **coDBMS**) lediglich auf diese Verfahren abzielen. Verlustfreie Kompression wird überall dort eingesetzt, wo ein Informationsverlust der Dateien nicht akzeptabel ist. Hierzu gehören z.B. Textinformationen, Programmcode und generell die im Rahmen von Datenbankanwendungen erfassten Informationen. Die wichtigsten Arten verlustfreier Kompression sind dabei *Lauflängenkodierung* (engl. Run Length Encoding (**RLE**)), Kompression basierend auf *statistischen Methoden (Huffmann* und *Arithmetische Kodierung*) sowie *Wörterbuchkompression (Lempel-Ziv).*
Lauflängenkodierung z.B. erreicht die Kompression durch Ersetzung aufeinanderfolgender sich wiederholender Zeichen eine kürzere Darstellung, die *statistische Kompression* wendet statistische Methoden auf die Eingabedaten an und die *Wörterbuchkompression* beruht auf der Beobachtung, dass sich Zeichenmuster in Eingabedaten wiederholen.

2.12.1 Lauflängenkodierung

Durch Lauflängenkodierung werden lange Folgen sich wiederholender Symbole komprimiert, indem sie durch eine kürzere Darstellung ersetzt werden. So kann z.B. die Zeichenfolge $\underbrace{AAAA}_{4}\underbrace{BBB}_{3}\underbrace{AA}_{2}\underbrace{BBBBB}_{5}\underbrace{CCCCCCCC}_{8}$ ersetzt werden durch die Zeichenfolge 4A3BAA5B8C . Bei binären Dateien kann die Lauflängenkodierung durch Ausnutzung der Tatsache, dass 0 und 1 abwechselnd auftreten. Anhang A.6.2 enthält ein aus [Seite 373 [Sed92]] entnommenes Beispiel, das den Ansatz verdeutlicht.

Lauflängenkodierung arbeitet nur effizient, wenn lange Ketten sich wiederholender Zeichen vorhanden sind, da nur dann die kodierte Information weniger Speicherplatz verbraucht als die Unkodierte.

2.12.2 Statistische Kompressionsverfahren (Huffman, Arithmetische Kodierung)

Statistische Methoden erreichen eine Komprimierung der Daten durch statistische Analyse der Häufigkeit von Symbolen oder Wörtern (als Konkatenation einzelner Symbole). Die Idee

ist, häufig vorkommende Informationen durch kürzere, weniger häufige Informationen durch längere Kodierungen zu ersetzen. Der Ansatz basiert auf der Idee des Morsecodes, der das am häufigsten vorkommende Symbol „e“ in der englischen Sprache durch einen einzigen Punkt darstellt. Zusätzlich zu den kodierten Daten muss die Codetabelle mit gesichert werden, damit die Daten wieder dekodiert werden können (vgl. [Coding1]).

2.12.2.1 Huffman-Kodierung

Bei der Huffman-Kodierung [Huff52] werden Daten fester Länge (Symbole) zu Daten variabler Länge kodiert. Häufig wiederkehrende Symbole sollen dabei durch möglichst wenig, selten auftretende Daten durch mehr Informationen kodiert werden. Die Kodierung wird anhand eines Binärbaums entwickelt, dessen Blätter die Symbole mit ihren Gewichtungen sind, welche sich nach der Häufigkeit ihres Auftretens richten. Durch Anwendung des Binärbaums erhält man eindeutige Kodierungen, siehe Anhang A.6.3.

Gegeben ist also ein Alphabet $A=\{a_1, a_2, \ldots, a_n\}$ mit den zu kodierenden Symbolen als Elemente. Ebenfalls gegeben ist die Menge $W=\{w_1, w_2, \ldots, w_n\}$ der (positiven) Gewichtungen der Symbole (die die Häufigkeit ihres Auftretens ausdrücken), d.h. $w_i = weight(a_i), 1 \le i \le n$. Ergebnis ist die Menge $C(A,W)=\{c_1, c_2, \ldots, c_n\}$ der (binären) Kodierungen, wobei c_i die Kodierung für $a_i, 1 \le i \le n$ ist. Beispiele für die Huffman-Kodierung finden sich in Anhang A.6.3.

2.12.2.2 Arithmetische Kodierung

Genau wie die Huffman-Kodierung gehört die Arithmetische Kodierung zu den statistischen Kompressionsverfahren. Im Gegensatz zu Huffman werden hier jedoch nicht die Eingabesymbole durch einen bestimmten Code ersetzt, sondern die Eingabedaten werden insgesamt in eine einzige reelle Zahl zwischen 0 und 1 umgewandelt. Je länger die Nachricht, umso mehr Bits werden zur Kodierung der Zahl verwendet. Es sollen aber nur so viel Bits wie nötig verbraucht werden. Hierzu werden die Wahrscheinlichkeiten des Auftretens der Symbole verwendet. Beide Verfahren haben gemeinsam, dass sie bei häufigem Auftreten eines Symbols für dieses nur sehr wenige Bits, bei selten auftretenden Symbolen entsprechend mehr Bits produzieren (vgl. [ct00]). Die Häufigkeit des Auftretens der einzelnen Symbole in den Eingabedaten erhält der Algorithmus z.B. durch Zählen aller Symbole vor dem Packen (wie bei der Huffman-Kodierung). Die arithmetische Kodierung versucht sich der *Entropie* möglichst optimal anzunähern. Anhang A.6.4 definiert den Begriff

der Entropie. Anhang A.6.5 beschreibt den Algorithmus der arithmetischen Kodierung und zeigt beispielhaft, dass die Huffman-Kodierung die Entropie nicht immer erreicht.

2.12.3 Wörterbuchkompression (Lempel-Ziv)

Die Huffman-Kodierung nutzt die Häufigkeit von Symbolen (eines vorgegebenen Alphabets) in den Eingabedaten aus, um eine Komprimierung der Daten zu erzielen (statistische Kodierung). Die Wörterbuchkompression hingegen arbeitet nicht mit (in der Regel einzelnen) Symbolen und deren statistischer Häufigkeit, sondern beruht auf der Beobachtung, dass sich ganze Zeichenmuster (konkatenierte Symbole des Eingabealphabets, bestehend aus 1-n Symbolen) in Eingabedaten wiederholen. Die Idee ist, die Wiederholungen dieser (möglichst langen) Muster durch (kürzere) Referenzierungen auf das Original, welches sich in einem aufgebautem Wörterbuch befindet, zu ersetzen [LZAlg]. Es wird dabei zwischen statischen und adaptiven (dynamisch aufgebauten) Wörterbüchern unterschieden. Ein statisches Wörterbuch enthält im Vorhinein festgelegte Wörter beliebiger Länge, ohne dass dafür aber die Eingabedaten analysiert wurden. Adaptive Verfahren hingegen bauen ihr Wörterbuch aus den Eingabedaten heraus beim Durchlauf derselben dynamisch (zur Laufzeit) auf. Die Lempel-Ziv-Kodierung gehört zu den adaptiven Wörterbuch-Kodierverfahren. Im Gegensatz zur Huffman-Kodierung, die Symbole fester Länge auf Kodierungen variabler Länge abbildet, wandelt die Lempel-Ziv-Kompression Wörter variabler Länge in Kodierungen fester Länge um. Bezüglich der Kompressionsrate ist Lempel-Ziv umso effektiver, je länger die sich wiederholenden Wörter sind. Im Gegensatz zur Huffman-Kodierung benötigt Lempel-Ziv keinen ersten Durchgang zum Analysieren der Eingabedaten bzw. kein Wissen über die Häufigkeit des Vorkommens der Muster. Das Wörterbuch wird zeitgleich mit der Kodierung beim Parsen der Eingabedaten erzeugt und ist nicht wie bei der Huffman-Kodierung statisch vorgegeben (vgl. [LZ]).

Das Lempel-Ziv-Verfahren ist kein einzelner Algorithmus, sondern eine ganze Familie an Algorithmen, die auf den beiden grundlegenden Algorithmen von Lempel und Ziv aus den Jahren 1977 und 1978 basieren ([LZ77], [LZ78]). Sie sind Bestandteil vieler Anwendungen wie *zip*, *gzip*, *compress*, *GIF*-Kompression, PNG-Kompression, PDF etc. Abbildung 49 und Tabelle 46 (Anhang A.6.6.1) geben eine Übersicht über die beiden Algorithmenfamilien. Nachfolgend vorgestellt werden sollen die beiden wesentlichen Entwicklungen LZ77 und LZ78.

2.12.3.1 Lempel-Ziv 77 (LZ77)

Lempel-Ziv 77 arbeitet mit einem Schiebefenster (engl. S*liding Window*) konstanter Größe, welches in einen *Vorschaupuffer* und einen *Suchpuffer* (auch *Lexikon*- oder *Wörterbuchpuffer* genannt, enthält bereits kodierte Daten) eingeteilt ist. Dieses wandert über die Eingabedaten . Dabei ist der Inhalt des Vorschaupuffers V zu komprimieren, während im Suchpuffer S bereits komprimierte Zeichenketten liegen, die als Wörterbuch zu benutzen sind. Der Suchpuffer ist also ein „implizites“ Wörterbuch (vgl. [Wander05]).
Der Algorithmus sucht dabei im Suchpuffer nach der längsten Zeichenkette in S (bereits kodierte Daten), die mit einem Präfix von V (Zeichenkette im Vorschaupuffer) übereinstimmt.

Wird eine Übereinstimmung gefunden, wird diese als (Referenzierungs-)Tripel, bestehend aus *Position* (Anzahl der Schritte, die man im bereits gelesenen Teilzeichenkette zurückgehen muss, um eine mit dem aktuellen Präfix des Vorschaupuffers übereinstimmende Zeichenkette zu erhalten) und *Länge* des Matches im Suchpuffer sowie dem *Zeichen* im Vorschaupuffer, das dem übereinstimmenden Präfix folgt (also nicht mehr übereinstimmt), kodiert. Es handelt sich hierbei um einen Verweis auf eine (Teil-)Zeichenkette, die im bisher durchsuchten Teil der Zeichenkette bereits vorhanden ist. Das für LZ77 verwendete Schiebefenster ist in Abbildung 14 dargestellt.

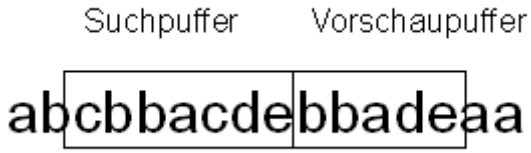

Abbildung 14: LZ77-Schiebefenster (aus [LZAlg1])

Der Algorithmus für Lempel-Ziv 77 sowie ein Beispiel für die Funktionsweise von LZ77 befinden sich in Anhang A.6.6.2.

Die Schwierigkeit bei LZ77 ist es, die optimale Größe des Schiebefensters zu finden. Verwendet man ein kleines Fenster, ist die Kompression schlechter, weil lange Wörter, die sich wiederholen, evtl. nicht entdeckt werden, wenn sie nicht vollständig in den Suchpuffer passen. Große Schiebefenster hingegen sorgt für eine lange Laufzeit, da bei jeder Suche nach der längsten Übereinstimmung des Präfixes ein großer Suchpuffer durchsucht werden muss.

2.12.3.2 Lempel-Ziv 78 (LZ78)

Im Gegensatz zu LZ77-Algorithmus besitzt der LZ78-Algorithmus kein „implizites“ Wörterbuch (bei LZ77 der Vorschaupuffer), sondern ein explizites Wörterbuch, welches jedoch nur zur Laufzeit der Kodierung bzw. Dekodierung existiert und nicht Bestandteil der kodierten Daten ist, sondern aus diesen bei Dekodierung generiert wird. Im Gegensatz zum Schiebefenster-Ansatz von LZ77 (implizites Wörterbuch), bei dem immer nur ein Teil der bereits gelesenen Daten als Wörterbuch verwendet wurde (abhängig von der Größe des Suchpuffers), speichert das Wörterbuch bei LZ78 alle bisher gelesenen Zeichenketten unterschiedlicher Länge und nutzt sie für Referenzierungen zum Einsparen von Speicherplatz. Zur effizienten Suche von übereinstimmenden Zeichenketten im Wörterbuch kann dieses als Suchbaum implementiert werden, vgl. [LZ77], [LZ78], [BELZ78] oder [Seite 5 [IW94]] (wo die Erzeugung des Parsebaums und das Suchen der zu kodierenden Wörter mit einem solchen erläutert wird). Diese Art der Implementierung soll jedoch nicht weiter erläutert werden, da sie über das Grundprinzip des LZ78-Verfahrens hinausgeht.

Die von LZ78 erzeugten Kodierungen bestehen aus einer Referenz zum längsten übereinstimmenden Eintrag im Wörterbuch und dem ersten nicht-übereinstimmendem Symbol. Die Menge $M = \{(i, N)\}$ der Paare von Referenz (Index) i und nicht übereinstimmendem Symbol N bildet die kodierten Daten. Der Algorithmus für LZ 78 (Pseudocode) sowie ein Beispiel für Kodierung und Dekodierung befinden sich in Anhang A.6.6.3.

2.12.4 Einsatz von Kompression in roDBMS und coDBMS

Der Einsatz von Kompressionsverfahren in Datenbanken war bereits in den Siebziger und Achtziger Jahren Gegenstand der Forschung (u.a. [Als75], [Ba85]), jedoch wurden sie erst wesentlich später tatsächlicher Bestandteil der kommerziellen **DBMS**, bei IBM DB2 bereits in den Neunziger Jahren (vgl. [IW94], [DB2Compr1]), bei Oracle erst seit Version 8i bzw. 9i.

In den in dieser Diplomarbeit untersuchten **coDBMS** werden sie, wie später noch gezeigt wird, fast ausnahmslos eingesetzt. Roth und van Horn [ROTH93] stellten bereits in den 90ern fest, dass der Einsatz von Kompressionsverfahren (damals noch bezogen auf **roDBMS**) aus zwei Gründen interessant sei, nämlich wegen des geringeren Speicherplatzverbrauchs und der Performanceverbesserung, da weniger Seiten vom Sekundärspeicher eingelesen bzw. in den

Sekundärspeicher geschrieben werden müssen. Die Einsparung von Seiten im Sekundärspeicher gilt nach [ROTH93] nicht nur für die gespeicherten Relationen, sondern auch für Indizes. Nachfolgend einige Beispiele, bei denen die Anfrageverarbeitung durch den *Query Processor* (Kapitel 2.14) sogar direkt mit den komprimierten Daten erfolgen kann:

1. Bei Prüfung auf Wertegleichheit (Exact-Match-Queries (bzw. Index-Lookups[9]) oder Natural Joins, z.B. von indizierten Schlüsseln mit einem gesuchten Wert), indem der gesuchte Wert zunächst komprimiert und dann mit den komprimierten Daten verglichen wird (vgl. u.a [ROTH93] und [GS91]).
2. Bei Projektion mit Duplikateliminierung (von Attributwerten), da gleiche unkomprimierte Daten gleiche (komprimierte) Abbilder haben [GS91].
3. Bei Gruppierungen (SQL: `GROUP BY`) und Aggregierungen (SQL: z.B. `SUM`) zumindest beim Gruppierungs-Attribut, jedoch nicht beim Attribut, auf dem die Aggregierung ausgeführt wird. Ausnahme ist die (insbesondere bei sortierten Spalten in **coDBMS** günstige) Lauflängenkodierung, bei der bei einer `SUM`-Aggregierung der `SUM`-Operator den aggregierten Wert einfach durch Multiplikation von Wert und Lauflänge bestimmen kann, ohne die Daten zu dekomprimieren, siehe [Seite 47 [Aba08]]. Gleiches gilt auch für die `AVG`-Aggregierung, die ebenfalls direkt auf **RLE**-Daten arbeiten kann, [Seite 8 [VertWhitePaper]].
4. Bei Verbunden (engl. *Joins*) müssen weder das Join-Attribut noch die anderen Attribute dekomprimiert werden. Fast alle Joins sind Equality Joins[10] auf Schlüsseln und Fremdschlüsseln. Da diese (Schlüssel) aus derselben Domäne stammen (d.h. Denselben Datentyp haben) und die Kompressionsverfahren für jede Domäne einheitlich sind, ergibt ein Join auf komprimierten Werten dasselbe Ergebnis wie ein Join auf unkomprimierten. Auch das Ausführen eines Merge Joins [Seite 370 [SHS05]] in der Reihenfolge der komprimierten Werte erzeugt korrekte Ergebnisse. Dieselben Argumente gelten auch für Semi-Joins, Outer-Joins, Unions (Vereinigung), Intersections (Schnittmenge) und Differenzen (vgl. [GS91]).

Weitere Vorteile von Datenkompression sind:

1. Das DBMS kann beim Durchführen des Verbunds mehr Daten (Tupel) im Hauptspeicher halten (in Form von Seiten im Datenbankpuffer ([Aba06]), z.B.

[9]Das Suchen von Datensätzen unter Zuhilfenahme eines Index

[10]Der Equality-Join wird auch Equi-Join genannt

vorteilhaft beim Hash Join [Seite 372 [SHS05]] der Join benötigt dadurch weniger I/O-Operationen, der Merge-Join ggf. entsprechend weniger Merge-Level (wenn das Datenvolumen den zur Verfügung stehenden Hauptspeicher übersteigt).

2. Bei der Transaktionsverarbeitung verkürzen sich die für das Recovery angelegten *Log Records* (durch das Verwenden komprimierter Werte oder durch Komprimieren der Logrecords selbst ([GS91])).
3. Die Navigationsfunktionen von *B-Bäumen* wie Range Queries (z.B. `Gehalt <= 90.000` oder `Gehalt BETWEEN 40.000 AND 50.000`) oder Vorsortierung für Sort-Merge-Joins können ohne Dekodierung von komprimierten Schlüsseln ausgeführt werden, wenn das Kompressionsverfahren die Sortierreihenfolge einhält (vgl. [IW94], wo verschiedene Kompressionsverfahren diesbezüglich erwähnt werden).

Ziel des Einsatzes von Kompressionsverfahren in **DBMS** ist es wie überall, redundante Daten zu finden, die keine „neue“ Information darstellen. Nach [ROTH93] gibt es vier Typen von *Redundanz* in Datenbanken, wobei bereits das für die jeweilige Redundanz geeignet Kompressionsverfahren erwähnt wird:

1. *Zeichenverteilung* (engl. *Character Distribution*): Hier wird das Prinzip der statistischen Kompressionsverfahren (Kapitel 2.12.2) aufgegriffen, d.h. häufig auftretende Zeichen können durch kürzere Kodierungen ersetzt werden.
2. *Zeichenwiederholung* (engl. *Character Repetition*): Hiermit sind kürzere Repräsentationen für sich wiederholende Zeichen, also das Prinzip der Lauflängenkodierung (Kapitel 2.12.1), gemeint.
3. *Wiederkehrende Muster* (engl. *High Usage Patterns*): Beschreibt das Kodieren von wiederkehrenden Zeichenketten (statt einzelner Zeichen wie bei der Zeichenverteilung), also die Wörterbuchkodierung (Kapitel 2.12.3) und das Kodieren von Zahlen (in einem Text) als Zahl (und nicht als Zeichen), was bedeutend weniger Bits in Anspruch nehmen würde.
4. Auftreten an vorhersagbaren Stellen (engl. *Positional Redundancy*): Tritt das gleiche Zeichen wiederholt an vorhersagbaren Stellen auf, muss lediglich einmal das (wiederholt auftretende) Zeichen und dann seine Positionen gespeichert werden.

Dieses Verfahren kann insbesondere in einer spaltenorientierten Speicherarchitektur angewendet werden, wie später noch anhand der *Bitvektor-Kodierung* gezeigt wird.

Neben den Arten von Redundanz werden in [ROTH93] und Kapitel 3.2 [WKHM] folgende grundlegende Komprimierungsverfahren beschrieben:

1. Verwenden von *Abkürzungen* (engl. *Abbreviations*): Ersetzen von Attributwerte durch eindeutig zugeordnete Abkürzungen (*statisches Wörterbuchverfahren* (Kapitel 2.12.3)). Da das Wörterbuch auch gespeichert werden muss, ist das Verfahren nur bei vielen Wiederholungen gleicher Werte effizient.
2. *Null-Unterdrückung* (engl. *Null Suppression*) und *Lauflängenkodierung*: Die Nullunterdrückung ist eine Lauflängenkodierung für Leerzeichen und Nullen.
3. *Musterersetzung*: Ersetzen von Muster mit 2-n Zeichen durch kürzere, noch nicht vorhandene. Zur gleichen Zeit wird ein Ersetzungs-Wörterbuch aufgebaut (ähnlich den adaptiven Wörterbuchverfahren wie LZ77 und LZ78 (Kapitel 2.12.3)).
4. *Differenz-Komprimierung (*alternativ*: Delta-Kodierung)*: Ersetzen einer Zeichenfolge unter Bezugnahme auf einen vorherigen oder einen spezifischen Wert. Diese Methode kann deshalb nur bei gleichartigen Daten angewendet werden, die nur geringfügig variieren (z.B. bei spaltenorientiert und sortiert angeordneten). Tabelle 9 zeigt die ein Beispiel für die Anwendung der Differenz-Komprimierung, wobei sich bei der Komprimierung immer auf den direkten Vorgänger bezogen wird.
5. *Statistische Kodierverfahren:* Siehe Huffman-Kodierung und Arithmetische Kodierung (Kapitel 2.12.2)
6. Lempel-Ziv-Komprimierung (wie bereits beschrieben)

Original-Daten	**Komprimierte Daten**
Johnson, Jonah, Jones, Jorgenson	(0)Johnson,(2)nah,(3)es,(2)rgenson
1500,1520,1600,1550,1570,1610	1500,20,80,-50,20,40

Tabelle 9: Beispiel für Differenz-Komprimierung (aus [ROTH93])

Bei der Anwendung von Kompressionsverfahren in DBMS stellen sich zusammenfassend folgende Fragen, nämlich *welche Dateneinheiten* komprimiert werden (Tupel/Seiten/Attribute), *wann* komprimiert und dekomprimiert wird (bei Laden der Seite, beim Zugriff auf das Tupel oder möglichst gar nicht), und *welche Kompressionsverfahren* (als

geeignetste[11]) eingesetzt werden. An dieser Stelle wird auf die entsprechende Literatur verwiesen, in der diese Fragen diskutiert werden [GS91], [ROTH93], [IW94].

2.12.4.1 Komprimierung in roDBMS am Beispiel von Oracle und IBM DB 2

Nach den einführenden Erläuterungen zur Kompression in DBMS soll nun am Beispiel der bekannten **roDBMS** Oracle und IBM DB2 die konkrete Implementierung von Kompressionsverfahren gezeigt werden.
Das DBMS Oracle setzt Kompressionsverfahren seit Version 8i ein. Die Funktionalitäten wurden dabei bis zur aktuellen Version stetig erweitert. Tabelle 10 aus [OraCompr5] zeigt die Historie der Kompressionsfunktionalitäten der verschiedenen Versionen des Oracle DBMS.

Kompressionsfunktionalität	**Verfügbar ab Version**
Index Compression	8i
Index-organisierte Tabellen (IOT)	8i
Tabellenkomprimierung für Bulk Load	9i Release 2
Tabellenkomprimierung für OLTP	11g
SecureFiles (LOB)	11g

Tabelle 10: Kompressionsfunktionalität in Oracle (aus [OraCompr5])

In Anhang A.6.7 sind die verschiedenen Kompressionsverfahren von Oracle dargestellt. Für die Tabellenkomprimierung wird eine Blockkompression eingesetzt. Hierbei handelt es sich um eine Wörterbuchkompression auf Blockebene, bei der mehrfach vorkommende Werte in eine pro Block existierende Symboltabelle ausgelagert werden und darauf bei Wiederholungen datensatzübergreifend referenziert wird.

Im Gegensatz zu Oracle 11g bietet IBM DB2 keine block-, sonder eine zeilenorientierte Kompression [Ah06]. DB2 9 komprimiert Tupel aber auch nach dem gleichen Prinzip wie Oracle durch das Einsparen von Duplikaten. Dazu wird eine gesamte Relation gescannt und ein Wörterbuch (für die gesamte Relation) mit den doppelten Einträgen aufgebaut. Die Tupel werden dann mit variabler Länge gespeichert [IW94]. Für die Freiraumbehandlung werden bekannte Algorithmen für Tupel variabler Länge verwendet [IW94]. IBM DB2 wendet (Stand 1994) nichtadaptive Kompression mit apriori-Wissen an, wobei dieses Wissen bei jeder Reorganisation aktualisiert werden kann [IW94]. Ob sich an dieser Implementation etwas geändert hat, ließ sich leider nicht weiter klären.

[11]Delta-Kodierung z.B. nur bei gleichartigen Daten möglich, daher schlecht bei Tupel mit unterschiedlichen Attributtypen

Ein Beispiel und weitere Features der DB2-Kompression finden sich in Anhang A.6.8. In Anhang A.6.9 ist abschließend ein Vergleich zwischen Oracle- und DB2-Kompression aufgeführt.

2.12.4.2 Komprimierung in coDBMS

Die recherchierte Literatur zum Thema „Kompression in **coDBMS**“ bezieht sich jeweils auf eine konkrete Implementierung (z.B. [Aba08] und [Aba06] auf das **coDBMS** „C-Store“ (siehe Tabelle 5), wobei alle betrachteten coDBMS-Implementierungen Kompressionsverfahren einsetzen. Implementierungsunabhängig können dabei folgende Aussagen getroffen werden:

1. Datenkompression wird in **coDBMS** zum Einsparen von Speicherplatz (weniger Platzverbrauch im Sekundärspeicher und effizientere Ausnutzung des Datenbankpuffers) und I/O-Bandbreite (Reduzierung von *Seek Time* und *Transfer Time*, da die Daten örtlich näher zusammen liegen und weniger unbenötigte Information übertragen werden muss, vgl. Kapitel 1 [Aba06]) eingesetzt.
2. Das Komprimieren von (aufgrund der spaltenorientierten Speicherarchitektur) gleichartigen (aufeinanderfolgenden) Daten bietet Vorteile gegenüber der zeilenorientierten Speicherarchitektur mit nicht gleichartigen Daten, In einem **roDBMS** ist jeder Attributwert Teil eines Tupels, das Kombinieren von Attributwerten (des gleichen Attributs) mehrerer Tupel erfordert mehr Aufwand als bei **coDBMS**. Insbesondere Kodierschemata wie z.B. Lauflängen- und Bitvektor-Kodierung sind in **coDBMS** effizient ermöglichen eine Verwendung der komprimierten Daten durch den Query Processor[12]. Durch das Arbeiten auf komprimierten Daten bei Ausführung des Anfrageplans wird CPU-Zeit eingespart. Wie mit komprimierten Daten gearbeitet werden kann, wird weiter oben in diesem Unterkapitel erläutert. Bei **coDBMS** kommt bei Einsatz von Bitvektor-Kodierung auch noch das Verwenden kodierter Daten für verknüpfte Prädikate (AND, OR) hinzu, wie noch gezeigt wird.

[Aba06] gibt einen Überblick über in **coDBMS** eingesetzten Kompressionsverfahren und erläutert, wo die Vorteile des Einsatzes in einer spaltenorientierten Speicherarchitektur liegen. Zwar bezieht sich [Aba06] auf das **coDBMS** „C-Store“, jedoch sind die beschriebenen

[12]Natürlich ist auch die Wörterbuchkodierung in coDBMS effizient, insbesondere bei sortierten Spalten (falls Blockkompression eingesetzt wird).

Kompressionsverfahren allgemeiner Art und für alle spaltenorientierten Speicherarchitekturen geeignet:

1. Nullunterdrückung (engl. *Null Suppression*): Wie bereits in Kapitel 2.12.4.1 beschrieben. Nullunterdrückung bezieht sich insbesondere auf das Komprimieren führender Nullen bei fester Satzlänge (wobei die feste Satzlänge dabei aufgegeben wird), beispielsweise werden für Integer-Werte nicht die vollen 4 Byte verwendet, sondern 1-4 Bytes mit vorangehenden 2 Bits für die Angabe der verwendeten Bytezahl.

2. *Wörterbuchkodierung*: Bei der Wörterbuchkodierung werden Muster durch Kodierungen, die weniger Speicherplatz benötigen, ersetzt. Zeilenorientierte **DBMS** wie Oracle oder DB2 haben hier den Block oder die Relation als Bezugsgröße, bei **coDBMS** ist es die jeweilige Spalte. Während bei den in Kapitel 2.12.4.1 beschriebenen **roDBMS** der Ansatz der Referenzierung bereits vorhandener Daten verfolgt wird (der in **coDBMS** natürlich genauso umsetzbar ist), wird in Kapitel 4.2 [Aba06] der Ansatz beschrieben, dass anhand Anzahl der eindeutigen Werte in der Spalte die minimale Anzahl von Bits zur Kodierung berechnet wird (bei 32 unterschiedlichen Werten also 5 Bits).

3. *Lauflängenkodierung (RLE):* Wie bereits beschrieben, gut bei sortierten Spalten mit wenigen unterschiedlichen Werten, an. Die Folgen werden ersetzt durch Tripel $(Wert, Startposition, Lauflänge)$. In **roDBMS** wird **RLE** nur zum Komprimieren von langen Zeichenketten mit vielen Leerzeichen oder aufeinanderfolgenden Zeichen eingesetzt bzw. bietet sich dort an [Aba06].

4. *Bitvektorkodierung*: Bitvektorkodierung bietet sich bei **coDBMS** an, wenn eine Spalte eine überschaubare Anzahl von Werten hat (also non-unique). Ein Bitvektor beschreibt das Vorkommen der Werte in der Spalte durch den Wert 1 und das Nichtvorkommen durch den Wert 0. Ein anderer Begriff für diese Art der Kodierung ist „Tokenization“ (Seite 12 [Bloor08]). Die Werte 1132231 werden als Bitstrings *(1,1100001),(2,0001100),(3,0010010)* kodiert. Bei dünnbesetzten Bitstrings ist eine weitere Komprimierung durch Lauflängenkodierung möglich. Bitstrings werden, wie in Kapitel 3 noch gezeigt wird, bei spaltenoptimierter Anfrageausführung im Anfrageplan verwendet. Wie man sehen kann, ist diese Art der Kodierung nicht ideal bei einer hohen Anzahl unterschiedlicher Werte, da für jeden Wert der gesamte

Bitvektor gespeichert werden muss (Seite 12 [Bloor08]). Eine Version der Bitvektorkodierung, die sogenannten Bitmap-Indizes (Kapitel 2.11.3.3), werden zum Indizieren in **roDBMS** verwendet, aber explizit und separat zu den Basisrelationen angelegt. **coDBMS** hingegen speichern bei Bitvektorkodierung so gesehen ihre Daten gleich mit im Index [Seite 4 [Los09]].

5. Delta-Kodierung: Delta- oder Differenz-Kodierung ist in coDBMS aufgrund der Gleichartigkeit aufeinanderfolgender Werte in den Seiten des Sekundärspeichers effizienter umsetzbar als in roDBMS, das Beispiel in Tabelle 9 macht dies deutlich.

6. *Lempel-Ziv-Kodierung:* Die bereits erläuterte Lempel-Ziv-Kodierung kann natürlich auch in coDBMS verwendet werden (vgl. [Aba06]). Lempel-Ziv gehört zu den wörterbuchbasierten Komprimierungsverfahren und ähnelt somit der unter Punkt 2 genannten Wörterbuchkodierung. Lempel-Ziv (Kapitel 2.12.3.1 und 2.12.3.2) könnte man jedoch insbesondere in **coDBMS** so einsetzen, dass das Wörterbuch nicht nur ganze Attributwerte, sondern kann z.B. auch granularere sich wiederholende Textteile enthält. Denkbar wäre, um den Unterschied zur bisherigen Wörterbuchkompression in DBMS aufzuzeigen, folgendes Beispiel mit Autokennzeichen: Gegeben sind die drei Werte „MD-AX 4659“, „MD-AY 2159“, „MD-AY 2278“. Bei der bisher vorgestellten Implementierung der Wörterbuchverfahren in DBMS wären dies drei unterschiedliche Werte. Jedoch gibt es in den einzelnen Zeichenketten Teilzeichenketten, die übereinstimmen („MD-“), sodass eine noch effizientere Kompression denkbar wäre.

7. Huffmann-, Arithmetische Kodierung: Sind grundsätzlich auch einsetzbar, jedoch ist gem. [Aba06] der CPU-Overhead groß

In Anhang A.6.10 wird zeilen- und spaltenorientierte Kompression beispielhaft verglichen.

2.13 Verteilte und parallele Datenbanken

Ein Teil der in Tabelle 5 aufgeführten Implementierungen von **coDBMS** ist aus Gründen der Performance und der Skalierbarkeit[13] als parallele **DBMS** bzw. für den Einsatz in Rechner-Grid-Umgebungen (siehe u.a. [Mic05] und [DMS09]) konzipiert. Nachfolgend soll deshalb

[13]Unter Skalierbarkeit wird hier die Möglichkeit der Ressourcenerhöhung (mehr Rechner und/oder Prozessoren) zur Steigerung der Verarbeitungsgeschwindigkeit bei wachsenden Eingabedaten verstanden.

ein kurzer Überblick über die wichtigsten Begriffe von verteilten/parallelen DBMS gegeben werden. Zwischen beiden Begriffen gibt es Überschneidungen.

Eine *verteilte Datenbank* ist eine logisch verknüpfte Menge von Daten (und ihrer Beschreibung), die physisch auf mehrere Rechner in einem Netzwerk verteilt sind (Kapitel 22.1 [CB05]). Die Gründe für die Verteilung (statt einer zentralen Datenhaltung mit zentralem DBMS) können z.B. darin liegen, bei einer verzweigten Struktur einer Organisation durch lokales Vorhalten z.B. filialspezifischer Daten eine bessere örtliche Verfügbarkeit und Performance zu gewährleisten oder die Administration der Daten in den jeweils zuständigen Fachabteilungen zu dezentralisieren (Kapitel 1.1 [Rahm94]).

Ein *verteiltes DBMS* (engl. *Distributed DBMS* (**DDBMS**)) ist das Softwaresystem, dass die Verwaltung der verteilten Datenbank übernimmt. Es besteht aus einer einzigen *logischen* Datenbank, die in Fragmente aufgeteilt ist. Die Verteilung ist dabei transparent (unsichtbar) für den Anwender. Jedes Fragment ist (ggf. repliziert) auf einen oder mehrere (vernetzte) Rechner verteilt und steht dort jeweils unter Kontrolle eines eigenständigen DBMS (lokale Autonomie). Jede *Seite* (hierunter wird eines der DBMS mit den jeweiligen Fragmenten verstanden) kann unabhängig von den anderen Benutzeranfragen zu den lokalen Daten und Daten auf anderen Rechnern verarbeiten, ist aber auch in eine globale Applikation eingebunden, die die Fragmente des DDBMS in ihrer Gesamtheit nutzt. Abbildung 15 zeigt die Topologie eines **DDBMS**.

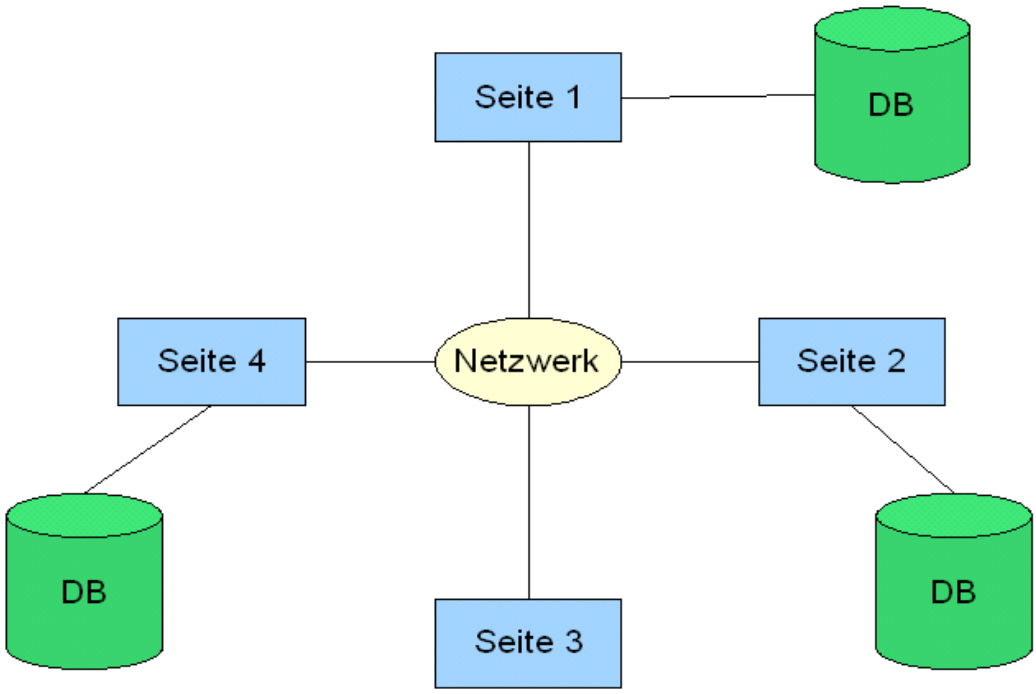

Abbildung 15: Topologie eines DDBMS (aus [CB05])

Parallele Datenbanken sind Realisierungen von DBMS auf Parallelrechnerarchitekturen. Der Anwender hat zwar eine globale Sicht wie auf ein normales DBS, intern werden aber parallelisierte Algorithmen eingesetzt (vgl. Kapitel 12 [SHS05]). In diesem Kapitel sollen nur einige wenige Grundlagen, die zum Verständnis der weiteren Ausführungen notwendig sind, erläutert werden.

Parallel-Computing im allgemeinen ist der Einsatz mehrerer Computer oder Prozessoren für die Lösung eines Problems oder die Ausführung einer Funktion [Lex09].

Unter einer *parallelen Datenbank* versteht man ein DBS, das seine Operationen auf mehrere gleichzeitig ablaufende Prozesse bzw. mehrere Prozessoren verteilt und parallel (und damit in kürzerer Zeit) ausführen lässt. Diese parallele Ausführung kann entweder auf einem Rechner oder auf mehreren Rechnern (in einem Netzwerk) stattfinden. Auch die parallele Verarbeitung von Transaktionen bzw. Datenbankabfragen wird unterstützt. Inwieweit **DDBMS** und parallele **DBMS** zu unterscheiden sind, wird im weiteren Verlauf des Kapitels beschrieben.

Grundlage paralleler Maschinen ist das Vorhandensein einer Anzahl p von Prozessoren (p > 1), wobei die Anzahl bei größeren Systemen mehrere hundert oder tausend beträgt (vgl. [Seite 986 [GMUW09]]). In Parallelsystemen haben diese Prozessoren die Möglichkeit, miteinander zu kommunizieren bzw. sich Information zu senden, z.B. über ein Bussystem oder Switches.

Parallele DBMS verbinden viele kleine Maschinen, um dieselbe Leistung wie eine einzige, große Maschine zu erreichen. Die Prozessoren teilen sich dabei die Ressourcen. Hierfür gibt es drei verschiedene Ansätze: Shared Memory[14], Shared Disk und Shared Nothing:

- *Shared Memory:* In einer Shared-Memory-Umgebung teilen sich mehrere Prozessoren den Hauptspeicher. Dieser Ansatz wird auch Symmetric Multiprocessing (**SMP**) genannt und findet z.B. auf Mehrprozessor-Rechnersystemen Anwendung. Flaschenhals ist hier die Verbindung zwischen Prozessoren und Hauptspeicher, sodass die Umgebung auf eine Höchstzahl von Prozessoren begrenzt ist (Abbildung 57).
- *Shared Disk:* Die Shared-Disk-Umgebung (Abbildung 58) ist für zentralisierte, hochverfügbare Umgebungen konzipiert. Jeder Prozessor kann direkt auf alle Disks zugreifen, hat aber seinen eigenen Hauptspeicher. Üblich ist hier der Einsatz von *Network Attached Storage*[15] (**NAS**) oder *Storage Area Networks* (**SAN**)[16]. Der Flaschenhals Shared Memory ist hier also nicht vorhanden. Im Gegensatz zu Shared Nothing-Architekturen sind die Relationen hier nicht genau einem Rechner, sondern allen Rechnern zugeordnet. Hiervon zu unterscheiden ist eine physische Fragmentierung bzw. Partitionierung der Relationen, die auch bei Shared-Disk-Umgebungen möglich ist, jedoch mit dem Unterschied zu Shared-Nothing, dass nicht nur ein Rechner Zugriff auf das jeweilige Fragment hat. Die Partitionierung der Plattenperipherie impliziert nämlich nicht notwendigerweise die (horizontale und/oder vertikale) *Partitionierung* der Datenbank [Seite 28 [Rahm94].
- *Shared Nothing:* Shared Nothing (Abbildung 59) ist eine Multiprozessor-Umgebung, wobei jeder Prozessor Teil eines Gesamtsystems mit eigenem Haupt- und Sekundärspeicher ist. Die logische Datenbank ist über alle Sekundärspeicher hinweg physisch partitioniert (d.h. Partitionierung ist bei Shared Nothing erforderlich, Kapitel 3.1.2 [Rahm94]) und für den Anwender transparent (Kapitel 1.1 [Rahm94]), d.h. die Partitionierung bleibt ihm verborgen. Jeder Prozessor kann nur auf seine Fragmente der Relationen zugreifen (im Gegensatz zu Shared Disk, soweit die Datenbank dort fragmentiert wird). Neben verteilten Ausführungsplänen muss auch ein rechnerübergreifendes Commit-Protokoll für Transaktionen zur Verfügung stehen. Die

[14]Wird in der Fachliteratur teilweise auch „Shared Everything“ genannt.

[15]Disk-Farm, die Dateien speichert und überträgt (vgl. Kapitel 20.1.1 [GMUW09])

[16]Überträgt Blöcke zu den Prozessoren (Kapitel 20.1.1 [GMUW09])

Architektur ist skalierbarer als Shared Memory und kann eine große Anzahl von Prozessoren unterstützen. Die Performance ist jedoch nur bei Lokalität der Daten beim jeweiligen Prozessor optimal. Die Shared-Nothing-Umgebung erfüllt die Eigenschaften eines **DDBMS**, wenn die Knoten (Prozessoren mit Hauptspeicher) des parallelen DBMS nicht im selben Rechner oder in räumlicher Nähe zueinander liegen, sondern geographisch[17] verteilt sind (Kapitel 3.1.1 [Rahm94]). Bei lokaler räumlicher Anordnung erfolgt die physische Verteilung der Daten (Partitionierung) aus reinen Performancegründen.

Parallele DBMS werden in der Regel bei sehr großen Datenbanken im Terabyte-Bereich eingesetzt, die pro Sekunde tausende von Transaktionen verarbeiten müssen. Durch die parallele Architektur können Scans, Joins und sortierungen durch mehrere Prozessoren parallel ausgeführt werden. Parallele DBMS werden deshalb häufig für Data Warehouses eingesetzt, wie im nachfolgenden Kapitel noch gezeigt wird, vgl. Kapitel 31.4.2 und 22.1.1 [CB05]. Eine zusammenfassende Klassifizierung der drei möglichen Architekturen von parallelen DBMS (angelehnt an Abbildung 3-1 [Rahm94]) sowie Abbildungen zu Shared-Nothing, Shared-Disk und Shared-Memory befindet sich in Anhang A.7.1.

2.13.1 Horizontale und vertikale Partitionierung

Der Entwurf eines Datenbanksystems ist unterteilt in die sieben Phasen Anforderungsanalyse, Konzeptioneller Entwurf, Verteilungsentwurf, Logischer Entwurf, Datendefinition, Physischer Entwurf und Implementierung, siehe [Seite 122 [SSH08]]. Beim *Verteilungsentwurf* wird festgelegt, wie die Daten in einem verteilten System mit mehreren Knoten aufgeteilt werden sollen. Unterschieden wird zwischen *horizontaler Verteilung* (Speicherung der Tupel einer Relation auf unterschiedlichen Knoten, waagerechter Schnitt durch eine Relation) und *vertikaler Verteilung*, d.h. die einzelnen Attribute der Tupel werden auf verschiedene Rechnerknoten verteilt (senkrechter Schnitt durch die Relation). Bezüglich der horizontalen Verteilung wird nur auf die bekannten Strategien Bereichs-, Round-Robin- und Hash-Partitionierung verwiesen [Seite 221 [SHS05]]. Das Prinzip der vertikalen Verteilung findet bei **coDBMS** Anwendung, jedoch nicht bei der Verteilung auf verschiedene Rechnerknoten, sondern bei der Festlegung der *Zugriffsstrukturen* auf der *physischen Ebene* des Sekundärspeichers. **CoDBMS** speichern die Tupel attributweise ab.

[17] z.B. aus Gründen lokaler Administration

Die Fragmentierung der Relationen auf Ebene der Zugriffsstrukturen wird auch *Decomposition Storage Model* (**DSM**) (deutsch: Dekomposition) genannt, siehe [CoKho85]. Es ist ein Synonym für die spaltenorientierte Speicherarchitektur, die bereits mit ihren Unterschieden zur zeilenorientierten vorgestellt wurde (Kapitel 2.8 und 2.10.4), und teilt logische Relationen in mehrere physische Tabellen auf, siehe auch Anhang A.7.2 für Illustrationen. Zu ergänzen ist, dass bei Verwendung mehrerer Attribute in einer Anfrage zur Tupelbildung über das verbindende Attribut intern ein Verbund gebildet wird. Wie dieser Verbund effizient durchgeführt wird, wird in Kapitel 3 bei den Implementierungen betrachtet, sofern dies aus der Literatur ersichtlich ist. Ein möglicher Ansatz ist z.B. die Durchführung eines Merge-Joins über gleichsortierten Attributen, siehe [AbaVP].

Der (etablierte) zeilenorientierte Ansatz wird synonym auch als n-äres Speichermodell (**NSM**) bezeichnet, bezogen auf die zusammenhängend gespeicherten n Attribute einer Relation (Kapitel 2.5.1), siehe [ADHS01]. Die Leseperformance wird bekanntermaßen durch Zusatzstrukturen wie Indizes gesteigert, [Seite 2 [Mo04]. Kapitel 2.10.4 gilt entsprechend für **NSM** und **DSM**.

Die Begriffe **NSM** und **DSM** werden als etablierte Begriffe an dieser Stelle eingeführt um zu verdeutlichen, dass ein **DBMS** nicht grundsätzlich „row oriented“ bzw. nicht grundsätzlich „column oriented“ ist, sondern es durch Dekomposition möglich ist, ein sogenanntes[18] zeilenorientiertes DBMS in ein spaltenorientiertes umzuwandeln, vgl. [Seite 19 und 28 [Aba08]] bzw. [Kho87]. Die Frage stellt sich dann natürlich, ob eine Organisation aus Gründen der Wartbarkeit nicht nur ein zeilenorientiertes **DBMS** einsetzen sollte, denn schließlich könnte sie bei Bedarf das **DBMS** entweder für ihre **OLTP**-Anwendungen verwenden oder für ihre **OLAP**-Erfordernisse anpassen, wohingegen sie bei einem **coDBMS** auf einen Einsatzzweck eingeschränkt wäre. Auf diese Frage wird in Kapitel 4 Bezug genommen.

Erwähnt sei auch noch der sogenannte Projektionsindex, der die Vorteile von **NSM** und **DSM** nutzt, ohne die Nachteile zu übernehmen. Ein DBMS setzt dabei **NSM** ein, ergänzt dies jedoch um redundante Projektionen auf einzelne Attribute (Seite 13 [ETH1]]), die äquivalent zum **DSM**-Modell sind. Je nach Kosten wird **NSM** oder der Projektionsindex genutzt.

[18]Der Begriff „sogenannt“ soll deutlich machen, dass es eigentlich keine klare Trennung zwischen den Begriffen gibt, denn es gibt kein Maß welches festlegt, ab welchem Grad der Dekomposition die Zeilenorientierung aufhört und die Spaltenorientierung anfängt.

2.14 Anfrageausführung und -optimierung (Query Processor)

Der wesentliche Unterschied zwischen **roDBMS** und **coDBMS** besteht neben der unterschiedlichen physischen Anordnung der Daten und den Unterschieden beim Einsatz von Kompression in der Umsetzung der Anfragebearbeitung. Deshalb werden nachfolgend die Grundlagen der Anfragebearbeitung beschrieben.

Eine vom Anwender an das **DBMS** (z.B. in **SQL**) formulierte Anfrage wird von diesem geparst und in eine Folge von auszuführenden Datenbankoperationen (z.B. Full-Table-Scan, Index-Scan, Verbundoperation) umgewandelt. Dies erfolgt in der Transformationskomponente des Datensystems, vgl. Abbildungen 4 und 5 (Seite 9 und 10). Der Ablauf der Anfragebearbeitung ist in Abbildung 16 dargestellt.

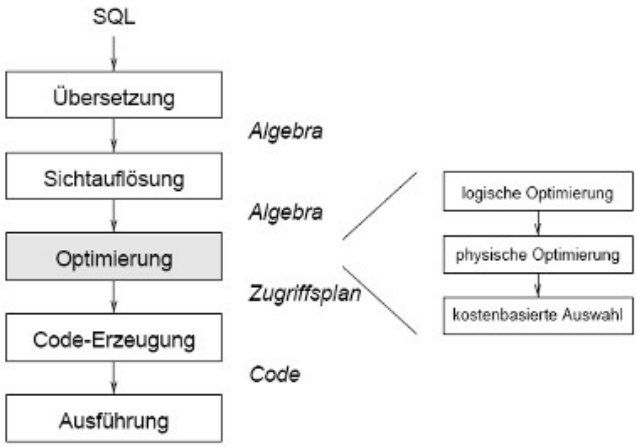

Abbildung 16: Phasen der Anfragebearbeitung (Seite 390 [SHS05])

Die wesentlichen Phasen der Anfragebearbeitung sind:

1. *Übersetzung und Sichtauflösung***:** Die (SQL-)Anfrage (inklusive Unteranfragen) wird in einen unoptimierten Term der Relationenalgebra (bei einem **RDBMS**) übersetzt. Zugriffe auf Sichten (Views) werden bei der Sichtauflösung durch die entsprechenden Anfragen ersetzt. Arithmetische Ausdrücke werden vereinfacht, Unteranfragen aufgelöst. Bereits hier entsteht ein (unoptimierter) Anfrageplan in Form eines Operatorbaums. Die innern Knoten des Baums repräsentieren Operatoren der Relationenalgebra (u.a. Vereinigung, Differenz, Schnittmenge, Kartesisches Produkt, Projektion, Selektion, Verbund), die Blätter repräsentieren Basisrelationen, die Kanten beschreiben den Datenfluss zwischen den Operatoren. Der Wurzelknoten repräsentiert

das Anfrageergebnis. Die Operationen werden von den Blattknoten in Richtung Wurzelknoten ausgeführt. Selektionen, Projektionen und Aggregationen haben einen einzelnen Kindknoten, wohingegen Verbundoperationen mehrere Kindknoten haben (S. 393 [SHS05]). Der Operatorbaum wird ab dem Zeitpunkt der physischen Optimierung als *Anfrageplan* bezeichnet (Seite 427 [SHS05]).

2. *Optimierung*: Unter Optimierung versteht man die Auswahl einer effizienten Ausführungsstrategie für die Verarbeitung einer Anfrage [Seite 632 [CB05]]. Bei der Optimierung wird unterschieden zwischen *logischer* und *physischer* Optimierung.

    1. *Logische (algebraische) Optimierung:* Der unoptimierte Anfrageplan wird (bei RDBMS) unabhängig von der physischen Speicherform der Relationen auf Basis von Regeln optimiert. Die Größe von Relationen oder die Existenz von Indizes wird z.B. nicht berücksichtigt. Die Optimierung erfolgt im Wesentlichen durch das „Kleinhalten“ von Zwischenergebnismengen durch Hineinziehen (d.h. frühzeitiges Ausführen) von Selektionen und auch Projektionen (um nicht benötigte Attribute zu entfernen und so den Datenbankpuffer effizienter nutzen zu können). Eine Selektion, die nur Attribute von einer Relation prüft, wird z.B. zu der Basisrelation im Operatorbaum hineingezogen (d.h. nach unten verschoben).

    2. *Physische (interne) Optimierung:* Im Operatorbaum werden die Operatoren der Relationenalgebra ersetzt durch konkrete Algorithmen. Bei der Wahl der Algorithmen (Full-Table-Scan, Index-Scan, Verbundoperationen) werden die vorhandenen Zugriffsstrukturen (Indizes, Cluster) berücksichtigt. In der Regel entstehen aus dem Operatorbaum der Relationenalgebra mehrere alternative Anfragepläne. Das gebräuchlichste physische Datenflussmodell ist, dass sich der Wurzelknoten mit *getNextTuple* das nächste Tupel von seinen Kindknoten abfragt und diese den Befehl weiterreichen (Iterator-Prinzip, Seite 432 [SHS05]). Die Tupel werden dann verarbeitet und das Ergebnis an den Elternknoten weitergereicht. In der Regel werden die Tupel einzeln übergeben, manche **DBMS** geben zur Performanceoptimierung auch mehrere Tupel zurück (Seite 421 [Aba08]).

3. Kostenbasierte Auswahl: Für die erstellten alternativen Anfragepläne wird anhand eines Kostenmodells eine kostenbasierte Auswahl getroffen wird. Die Kostenbasierte Auswahl nutzt Statistikinformationen (Relationengröße, Selektivität von Attributen)

für die Auswahl eines internen Plans. Das Ziel ist, den Plan mit dem geringsten Ressourcenverbrauch zu wählen bzw. den mit der geringsten Ausführungszeit, die sich aus den Ausführungszeiten aller individuellen Operationen der Anfrage zusammensetzt [Seite 632 [CB05]]. Grundsätzlich ist immer das Ziel, sowenig Seitenzugriffe wie möglich auszuführen.

Ein *Query Processor* ist die Komponente in einem DBMS, die eine gegebene Anfrage parst, den Anfrageplan bestehend aus Datenbankoperationen erstellt, diesen optimiert und schlussendlich ausführt, vgl. u.a. [Seite 701 [GMUW09]] und [Seite 631 [CB05]].

Grundsätzlich ist die Frage von Zeilen- oder Spaltenorientierung eines **DBMS** eine Frage der Speicherarchitektur, der in der höheren Schicht (Datensystem) liegende Anfrageausführer und -optimierer (engl. *Query Processor)* ist von dieser Problematik nicht zwangsläufig betroffen, denn bei **coDBMS** sind zwei Ansätze für die Implementierung des *Query Processor* möglich, nämlich

1. zum einem ein „normaler“ (zeilenorientierter) *Query Processor*, der auf der spaltenorientierten Speicherarchitektur aufsetzt, und zum anderen,
2. ein für eben jene Speicherarchitektur konzipierter *Query Processor*, siehe [Seite 19-20 [Aba08]].

Der erste Ansatz wird dadurch realisiert, dass vom Speichersystem (Kapitel 2.2) die Tupel der für eine Anfrage benötigten Spalten zusammengefügt wurden, die dann von einem herkömmlichen (zeilenorientierten) *Query Processor* weiterverwendet werden.

Dabei bleiben alle Tabellendefinitionen und SQL-Statements gleich, lediglich die physische Speicherung ändert sich. Dadurch bedingt muss vor der Verarbeitung der Anfrage die Tupelrekonstruktion durch Heranziehen der benötigten Attribute durchgeführt werden. Im Gegensatz zu **roDBMS** ist jedoch auch hier eine Optimierung möglich, da lediglich die für die Anfrage benötigten Spalten gelesen und zu Tupeln zusammengesetzt werden müssen, z.B. indem der i-te Wert jeder Spalte in das i-te Tupel der zu erzeugenden Zwischenrelation kopiert wird. Dieser Ansatz ist jedoch nur bei einer Heap-Speicherung im jeweiligen Attribut-Record (Kapitel 2.8.1) möglich, ansonsten müssen die logischen Tupel-IDs zum Zusammenfügen herangezogen werden. Wählt man den sortierten Ansatz (der auf Seitenebene vorhanden sein muss), ist ein Merge Join (ohne Sortierung) die offensichtliche Wahl für Tupelrekonstruktion, ansonsten ein indexbasierter Merge-Join.

Der zweite Ansatz ist, einen spaltenorientierten *Query Processor* zu entwickeln. Dieser könnte z.B., wird ein Prädikat[19] nur auf eine Spalte angewendet, nur diese isoliert bearbeiten (anstatt das ganze Tupel), wodurch weniger Daten durch die CPU verarbeitet werden müssten. Darüber hinaus können Anfrageprozessoren für das Arbeiten auf komprimierten Daten modifiziert werden, wie in Kapitel 2.12.4 bereits erwähnt.

Die Mehrzahl der in dieser Diplomarbeit betrachteten Implementierungen von **coDBMS**, u.a. C-Store/Vertica und Infobright, besitzen einen Anfrageoptimierer, der die Besonderheiten der spaltenorientierten Speicherarchitektur zur Anfrageoptimierung ausnutzt.

[19]Ein Prädikat ist ein Selektionskriterium

3 Klassifizierung der Ansätze für coDBMS

Aufbauend auf den in Kapitel 2 beschriebenen Grundlagen und Unterschieden zu **roDBMS** werden nun die in Tabelle 5 genannten **coDBMS** auf Basis der verfügbaren Literatur vorgestellt und anhand ihrer beschriebenen Eigenschaften klassifiziert.

Zunächst wurde für die bekanntesten **coDBMS** eine vertiefende Darstellung von deren Architektur und Ansätzen in den Unterkapiteln dieses Kapitels vorgenommen. Ausgewählt wurden dafür die **coDBMS** C-Store, Vertica, Google Bigtable und Nachbildungen (Hbase, Hypertable, Cassandra Project), Infobright (MySQL-Plugin) und LucidDB. Dies diente der Gewinnung von Schwerpunkten bei der Betrachtung dieser und der weiteren Implementierungen.

Anhand der Beschreibungen in der recherchierten Literatur für eben genannte **coDBMS** und einem Beispielansatz für die Klassfizierung von **DBMS** [DBClasses] wurden folgende Betrachtungsschwerpunkte entwickelt:

1. *Zugriffsstrukturen*:
    a. Wie wird die Dateiorganisation durch das **coDBMS** umgesetzt ? Inwieweit ist die Dateiorganisation spaltenorientiert ?
    b. Welche Indextypen sonstigen Ansätze bietet das **coDBMS** zur Verbesserung der Anfrageperformance (B-Bäume, Bitmap-Indizes etc.) ?
2. *Algorithmen*:
    a. Welche Algorithmen setzt das **coDBMS** zur effizienten Anfragebearbeitung ein ?
    b. Setzt das **coDBMS** Datenkompression ein und wenn ja, wie ?
3. *Anwendungen*:
    a. Für welche Anwendungsszenarien wurde das coDBMS konzipiert (Data Warehouse oder anderer Zweck) ?
    b. Welche Schnittstellen bietet das DBMS für die nutzenden Anwendungen ? Bietet es eine **DDL** und eine **DML** (z.B. SQL) und/oder **APIs** (**ODBC**-, **JDBC**-Anbindung, Anbindungen für Programmiersprachen (C, C++, Java, PHP, Perl etc.).

c. Für welche Art von Daten wurde das **coDBMS** konzipiert bzw. welches Datenmodell wird verwendet ((objekt-)relationalen DBMS mit üblichen Datentypen oder nur ein Datentyp)) ?

4. *Anfrageverarbeitung*:
   a. Arbeitet das **coDBMS** mit einem Optimierer, der die spaltenorientierte Speicherarchitektur ausnutzt ?
   b. Kann der Optimierer direkt auf den komprimierten Daten arbeiten (siehe Kapitel 2.12.4.1 und 2.12.4.2)?
   c. Welche sonstigen Ansätze werden zur Anfrageoptimierung eingesetzt ?
5. *Verteilungsschema***:**
   a. Handelt es sich um eine lokake Datenbank (auf einem Workstation-PC), eine Single-Server-Datenbank, eine parallele oder eine verteilte Datenbank ?
   b. Abhängig von 5a: Wird eine Shared Nothing-, Shared Disk- oder Shared Everything-Architektur eingesetzt ?
   c. Abhängig von 5a: Wird Datenfragmentierung (horizontal/vertikal) eingesetzt ? Wenn ja, wie ?
6. *Transaktionsverwaltung (nebensächlich)*: Wie wird die Update-Problematik (bei Data Warehouses) umgesetzt, d.h. dass ein lesender Zugriff auf die Datenobjekte trotz gleichzeitigem schreibenden Zugriff möglich ist (Stichwort: *Snapshot Isolation*) ?
7. *Plattform (nebensächlich)*: Für welche Betriebssysteme ist das **coDBMS** verfügbar ?

Es handelt sich dabei um die Punkte, die in der recherchierten Literatur am signifikantesten zum einen die Unterschiede zwischen **roDBMS** und **coDBMS** und zum anderen zwischen den verschiedenen Ansätzen der **coDBMS** deutlich werden lassen. Einschränkend ist jedoch anzumerken, dass nicht für jede Implementierung Aussagen zu allen genannten Schwerpunkten gefunden wurden.

Für die Schwerpunkte wurden pro Implementierung ihre Umsetzungen ermittelt, welche zugleich Ansätze und auch Klassifizierungskriterien sind. Kapitel 3.3 gibt eine Gesamtübersicht über die recherchierten Ansätze. Diese unterteilen die Betrachtungsschwerpunkte 1-7 in mehrere Untergruppen, welche die Klassifizierungen für die betrachteten **coDBMS** darstellen.

Implementierungen, die mangels (ausreichender) Literatur zu den Schwerpunkten oder aus anderen Gründen nicht bei der Klassifizierung berücksichtigt wurden, sind in Kapitel 3.2 aufgeführt.

3.1 Vorstellung und Klassifizierung der Implementierungen

Nachfolgend werden die Implementierungen vorgestellt, die sich anhand ihrer Eigenschaften und der vorhandenen Literatur für eine Klassifizierung eignen. Am Ende jedes Unterkapitels werden die genannten Klassifizierungskriterien auf die jeweilige Implementierung angewendet.

3.1.1 C-Store und Vertica

Auch wenn C-Store und Vertica grundsätzlich zwei eigenständig implementierte **coDBMS** sind (das erste nichtkommerziell, das zweite kommerziell), so wurde nach Literaturrecherche deutlich, dass beide **coDBMS** zusammen betrachtet werden können, denn Vertica ist der kommerzielle Nachfolger von C-Store, letzteres wird nicht mehr weiterentwickelt.
C-Store ist ein spaltenorientiertes Open-Source-**DBMS**, das an der Brown University, Brandeis University und am Massachussetts Institute of Technoly von Michael Stonebraker, Samuel Madden, Daniel Abadi, David DeWitt und anderen entwickelt wurde (siehe u.a. [CSWik]). Es liegt aktuell in der Version 0.2 (für Linux) vor, wird aber seit Oktober 2006 (vgl. [CSHome]) nicht mehr weiterentwickelt und ist damit für den produktiven Einsatz nicht mehr ernsthaft verwendbar, zumal es lt. [CSHome] nur noch mit alten Versionen von BerkeleyDB und GNU C Compiler (gcc) kompilierbar ist. Jedoch sind die Ideen und Erfahrungen bei der Entwicklung von C-Store in das kommerzielle Produkt Vertica eingeflossen, deshalb ist eine Betrachtung von C-Store durchaus noch von Interesse. Auf der Homepage wird empfohlen, statt C-Store nunmehr Vertica einzusetzen.
Die Vertica Analytic Database (Hersteller: Vertica) ist die kommerzielle Variante des seit 2006 nicht mehr weiterentwickelten DBMS C-Store ([VERTWIKI]). Die Firma Vertica wurde 2005 gegründet. Die Eigenschaften (vgl. u.a. [VERTWIKI] und [VertWhitePaper]) von Vertica sind (logischerweise) im Wesentlichen identisch mit denen von C-Store.

Vertica Analytic Database und C-Store sind gridbasierte, spaltenorientierte **DBMS** und haben ihre Anwendungsschwerpunkte bei Anfragen an große Datenmengen, wie sie bei Data Warehouses üblich sind.

Die wesentlichen Eigenschaften sind

1. Eine spaltenorientierte Speicherarchitektur mit darauf abgestimmter Anfragebearbeitung und -optimierung,
2. eine hybride[20] Speicherarchitektur, die sowohl lese- als auch schreiboptimiert ist (also kontinuierliche transaktionale Updates bei gleichzeitigem lesenden Zugriff auf denselben Daten unterstützt),
3. Komprimierung der gespeicherten Daten mit einem für die jeweilige Spalte geeignetem Kodierverfahren,
4. automatisierte Tools zum Entwerfen des optimalen physischen Datenbankdesigns (horizontale und vertikale Partitionierung der Tabellen in sogenannte Projektionen über mehrere Rechner hinweg),
5. Physisches Datenmodell bestehend aus sich überlappenden Projektionen der logischen Relationen statt Tabellen und Sekundärindexen,
6. Ausfallsicherheit durch *K-Safety*, d.h. Replizierung der Daten auf den Knoten, sodass K Rechnerknoten ausfallen können, ohne dass die Funktionen gestört sind,
7. Unterstützung einer skalierbaren, grid-basierten Shared-Nothing-Umgebung (Standard-Linux-Server), die auf lokale Disks und/oder **SAN** zugreift,
8. verteilte Transaktionen ohne REDO-Log oder 2-Phase-Commit,
9. **SQL** als **DDL**, **DML** und Anfragesprache, Integration bekannter ETL und OLAP-Werkzeuge, **JDBC**- und **ODBC**-Anbindung (Seite 10 [VertCompr]),
10. sowie eine effiziente Schnappschuss-Isolierung (engl. S*napshot Isolation*) zum Vermeiden transaktionaler Konflikte bzw. permanenter Ermöglichung von Readonly-Transaktionen.

Die nachfolgenden Beschreibungen sind u.a. [Mic05], [Aba08] und [VertWhitePaper] entnommen. Bei C-Store und Vertica handelt es sich im Gegensatz zu den (aufgrund ihres Einsatzgebietes **OLTP**) meist schreiboptimierten relationalen DBMS [Mic05] um leseoptimierte relationale **DBMS**. Die auf konzeptueller Ebene vorhandenen Relationen werden physisch als Menge sich überlappender spaltenorientierter *Projektionen* gespeichert. Eine *Projektion* ist hierbei eine Menge von Attributen, die nach demselben Attribut sortiert sind. Dasselbe Attribut kann unterschiedlich sortiert in mehreren Projektionen vorkommen. Auf diese Art und Weise kann bei Anfragen die am günstigsten sortierte Projektion gewählt werden. Die Projektionen werden in Vertica durch den Befehl `CREATE PROJECTION` erzeugt und beziehen sich auf eine zuvor mit `CREATE TABLE` erzeugt logische Tabelle.

[20]Hybride bezieht sich hier nicht auf einen kombinierten Column- und Row Store-Ansatz

Daten können erst gespeichert werden, wenn für alle Attribute Projektionen angelegt wurden [Seite 319 [VertSQL]].

Der Database Designer erstellt auf Basis des logischen Modells, einer Beispieldatenmenge und „Trainings"-Anfragen Vorschläge für das physische Modell (Projektionen, Kompressionsverfahren, Partitionierung auf mehrere Rechner, K-Safety) und die entsprechenden Skripte für das Erzeugen des physischen Modells [VertAdmin], siehe Abbildung 17.

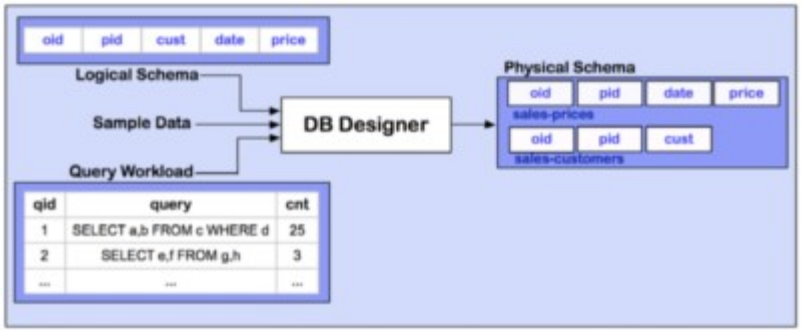

Abbildung 17: Arbeitsweise des DB-Designers von Vertica (Seite 10 [VertWhitePaper])

C-Store (und vermutlich auch Vertica) speichert jede Spalte einer Tabelle (bzw. Projektion) in einer separaten Datei. Jede Spalte wird dabei in Blöcke von 64K unterteilt (Seite 41 [Aba08]). Wenn die Daten einer Spalte komprimiert sind oder variable Länge haben, können die Blöcke eine unterschiedliche Anzahl an Werten für eine Spalte beinhalten. Jeder Block besitzt am Anfang einen Header, der den verwendeten Kompressionsalgorithmus angibt, die Anzahl der gespeicherten Werte und die Ordinalzahl des ersten Wertes im Block. Die Adressen weiterer Werte (für wahlfreien Zugriff) wird lt. Vorliegender Literatur noch nicht im Header gespeichert. Die Speicherung der Blöcke erfolgt entweder direkt im Filesystem des Betriebssystems oder als 64K-Attribute in indizierten Berkeley DB-Dateien (Seite 41 [Aba08]). Bei Speicherung einer Spalte mit Berkeley DB können zwei Arten von Indizes genutzt werden:

1. Bei einer sortierten Spalte wird ein dünnbesetzter Index erzeugt, was bedeutet, dass der letzte Wert des 64K-Blocks im Index gespeichert wird.
2. Ein zweiter dünnbesetzter Index wird für die Position jedes Werts (die der (logischen) Tupel-ID[21] entspricht) angelegt, im Index gespeichert wird die letzte Position im

[21]Zeilennummer innerhalb der Relation

Block. Dies ist z.B. sinnvoll für Operatoren, die ein komplettes Tupel rekonstruieren möchten.

Dichtbesetzte Indizes, wie sie in spaltenorientierten DBMS üblich sind, finden bei C-Store keine Verwendung (Seite 45 [Aba08]). Auch die SQL-Manual für Vertica ([VertSQL])enthält keinen `CREATE INDEX`-Befehl.

Auch in Data Warehouses müssen transaktionale Updates durchgeführt werden, z.B. zur Fehlerkorrektur und zum Einfügen neuer Daten. Problematisch dabei ist, dass lesende Transaktionen (welche in OLAP-Anwendungen lange dauern können) die schreibenden blockieren und umgekehrt schreibende Transaktionen alle anderen Transaktionen blockieren.

Es stellt sich also die Frage, wie man in einem spaltenorientierten **DBMS** einerseits eine Leseoptimierung, andererseits aber keine großen Verzögerungen bei schreibenden Zugriffen in Kauf nehmen zu muss, denn das eine wirkt in der Regel entgegengesetzt zum anderen[22]. C-Store und Vertica bieten hierfür zwei ineinandergreifende Lösungsansätze:

1. Verbessern der Schreib- und gleichzeitig der Leseperformance durch Bereitstellen einer hybriden Speicherarchitektur, bestehend aus schreiboptimiertem Speicher (engl. *Writeable Store (WS)*) und leseoptimiertem Speicher (engl. *Readable Store (RS)*).
2. Beseitigen der Behinderung von Readonly-Transaktionen durch Schreibtransaktionen im RS durch *Snapshot Isolation*.

Sowohl WS als auch RS sind spaltenorientiert. Die beiden Komponenten sind durch einen sog. *Tuple Mover* miteinander verbunden. Abbildung 18 veranschaulicht diese Architektur. Inserts und Updates finden grundsätzlich nur im WS statt und werden dann in Batch-Läufen in den (persistenten) RS übertragen. Deletes hingegen werden im RS markiert und später vom *Tuple Mover* entfernt. Updates sind als kombinierter Delete und Insert mit geänderten Werten implementiert. Das Konzept des *Tuple Movers* macht deutlich, dass Vertica und C-Store keine direkten Updates (engl. *In-Place updates*) unterstützen (Seite 4 [VertWC08]).

Für den *Tuple Mover* wird eine Variante des Konzept des *LSM*-Baums verwendet. Da hierzu aber keine weiteren Beschreibungen erhältlich waren, wird nur auf die entsprechende Grundlagenliteratur verwiesen [ONEI96].

Auch trotz der hybriden Speicherarchitektur sind Transaktionskonflikte vorhanden, nämlich beim Batch-Insert des Tuple Movers vom WS in den RS. Das herkömmliche Locking würde

[22]In roDBMS verlangsamen Leseoptimierungen wie z.B. Indizes die Schreiboperationen, da diese zusätzlich zur Tabelle geupdatet werden müssen.

in Lese-Schreib-Konflikten enden. Stattdessen werden lesende Anfragen im sogenannten historischen Modus mittels *Snapshot Isolation* [Bere95] ausgeführt. Die Anfrage wird dabei mit einer Zeitmarke (engl. Timestamp) versehen, berücksichtigt werden nur die bis dahin beendeten Transaktionen. Die Schnappschuss-Isolierung erfordert das Versehen eingefügter Datenelemente mit Timestamps. Durch *Snapshot Isolation* erhalten Lesetransaktionen immer eine konsistente Sicht auf den Datenbestand.

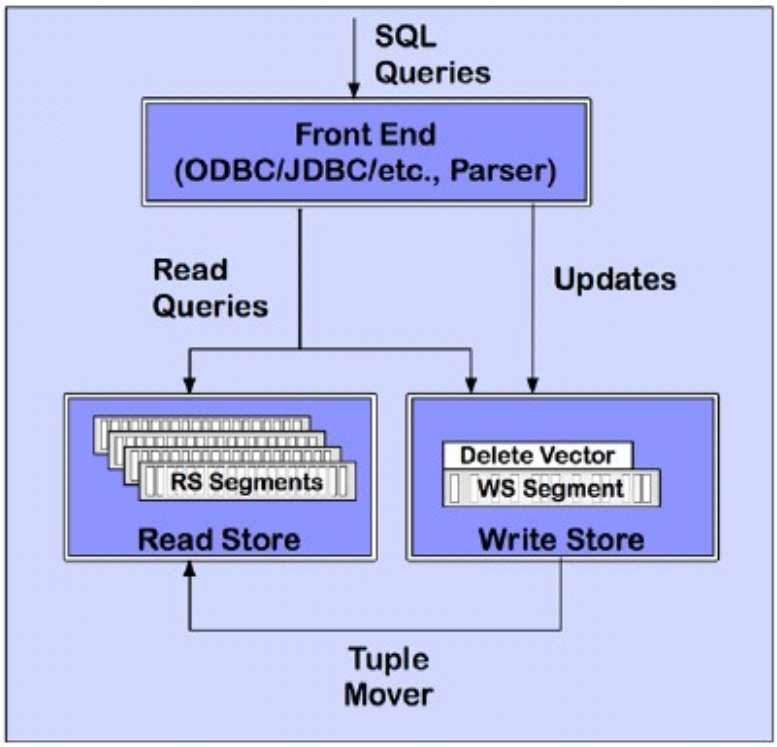

Abbildung 18: Architektur von C-Store und Vertica ([VertWhitePaper])

3.1.1.1 Logisches und physisches Datenmodell

C-Store und Vertica unterstützen auf konzeptueller Ebene das relationale Datenmodell mit Relationen, Attributen und Tupeln (siehe u.a. Kapitel 4.1 [HS00]) und damit auch Primär- und Fremdschlüssel. Als Anfragesprache, **DDL** und **DML** wird die Structured Query Language (**SQL**) verwendet (u.a. Seite 3 [VertWP]). Auf physischer Ebene werden Projektionen verwendet, die mit einer (logischen) Relation T verankert sind (sog. *Ankerrelation*) und eine Teilmenge der Attribute enthalten. Unter Aufbewahrung doppelter

Zeilen werden die gewünschten Attribute aus T projeziert. Eine Projektion kann außerdem Attribute aus anderen Relationen bei vorhandenen Fremdschlüsselbeziehungen der Ankerrelation enthalten [Seite 3 [Mic05]]. Auf diese Art und Weise werden Verbunde materialisiert (Seite 3 [VertWC08]). Anzumerken ist, dass der Begriff „Projektion“ hier insoweit vom etablierten Begriff abweicht, dass hier keine Basisrelation gespeichert ist, von der die Projektion gebildet wird.

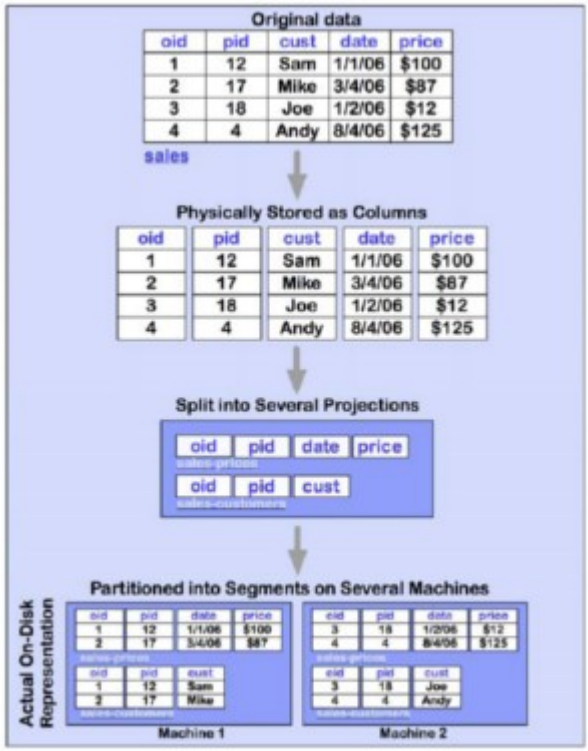

Abbildung 19: Physische Speicherung einer Relation in Vertica (aus [VertWhitePaper])

Jede Spalte kann in 1-n Projektionen gespeichert werden, die partial redundante Kopien darstellen. Abbildung 19 zeigt die physische Speicherung einer Relation in mehreren Projektionen (die die Relation sowohl horizontal als auch vertikal partitionieren) auf verschiedenen Rechnerknoten. Die Sortierreihenfolge kann dabei pro Projektion festgelegt werden. Die in Abbildung 19 genannte Projektion `sales-customers` kann z.B. nach der `customer-id` sortiert sein, um möglichst effizient alle Produkte eines Kunden zu finden.

Für die Beispielrelation ANG (Angestellte) [Mic05] in Tabelle 11 werden in Tabelle 12 mögliche Projektionen angegeben. Der Sortierschlüssel wird dabei durch eine vertikale Trennlinie kenntlich gemacht. Auch eine Sortierung mit einem Schlüssel einer anderen Relation (Fremdschlüssel-Beziehung) ist möglich (Projektion ANG2).

Name	Alter	Abteilung	Gehalt
Bob	25	Math	10000
Bill	27	EECS	50000
Jill	24	Biologie	80000

Tabelle 11: Beispielrelation ANG [Mic05]

```
ANG1 (Name, Alter|Alter)
ANG2 (Abteilung, Alter, ABT.GESCHOSS|ABT.GESCHOSS)
ANG3 (Name, Gehalt|Gehalt)
ABT (Name, Geschoss|Geschoss)
```

Tabelle 12: Projektionen für Beispielrelation in Tabelle 11

Jede Projektion wird horizontal in 1-n Segmente mit je einem Segment Identifier (**SID**) partitioniert (bzw. fragmentiert). Unterstützt wird wertebasierte Partitionierung anhand des Sortierschlüssels.

Die Anforderungen an die Projektionen sind:

1. Jedes Attribut der Relationen muss durch die Projektionen abgebildet sein, um SQL-Anfragen beantworten zu können.
2. Darüber hinaus müssen aus der Menge der Segmente alle Tupel vollständig rekonstruiert werden können.

Problem bei der Umsetzung der zweiten Anforderung ist, dass die Attribute in verschiedenen Projektionen unterschiedlich sortiert sind bzw. sein können. D.h. Auch wenn für eine (logische) Ankerrelation die existierenden Projektionen bekannt sind, kann die Bildung kompletter Tupel aufwändig sein, denn es handelt sich hier um einen Verbund (Join) mehrerer Projektionen, die bekannten Methoden hierfür sind *Nested Loop*-, *Merge*- oder *Hash-Join*. Der Merge Join würde gleichartig sortierte Projektionen voraussetzen.

Um die zweite Anforderung effizient erfüllen zu können, müssen die Segmente (horizontale Partitionen) verschiedener Projektionen miteinander verbunden werden. Dies wird durch die Verwendung von *Speicherschlüsseln* (engl. S*torage Keys*) und *Verbundindexen* (Join Indexes, siehe Kapitel 2.11.3.4) erreicht.

- *Speicherschlüssel* (S*torage Keys*): Jedes Segment verbindet jeden Wert jeder (Projektions)Spalte mit einem Speicherschlüssel *SK*. Werte aus verschiedenen Spalten desselben Segments, die den gleichen *SK* -Wert haben, gehören zum selben Tupel der Relation. Der Speicherschlüssel ist also das gleiche wie ein logischer TID. Speicherschlüssel *SK* werden 1,2,3,... im RS nummeriert, jedoch nicht physisch gespeichert, sondern aus der physischen Position des (Projektions)Tupels in der Spalte abgeleitet. Die physische Speicherung findet ausschließlich im WS statt.

- *Join-Indizes*: zum Rekonstruieren aller Tupel von Tabelle T aus seinen Projektionen benutzt C-Store (und vermutlich auch Vertica) Join-Indizes. Angenommen die Projektionen T1, T2 decken Tabelle T ab, dann ist ein Join-Index von m Segmenten in T1 zu n Segmenten in T2 logisch eine Menge von m Tabellen, eine pro Segment S, mit den Zeilen (s: SID in T2, k: Storage Key des korrespondierenden Wertes in T2). Im Prinzip werden durch den Join-Index die Speicherschlüssel und SIDs zweier Projektionen miteinander verbunden. Abbildung 20 zeigt einen Join-Index, der Projektion ANG3 auf ANG1 abbildet bzw. diese verknüpft, wobei die Annahme ist, dass es je Projektion nur ein Segment (SID=1) gibt. Der erste Eintrag von ANG3 korrespondiert mit dem zweiten von ANG1, daher hat der erste Eintrag des Join-Index den Speicherschlüssel 2. Änderungen an Projektionen erfordern natürlich auch Änderungen der Join-Indizes.

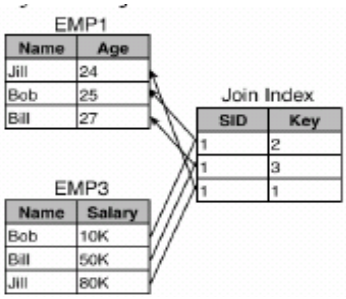

Abbildung 20: Join-Index für zwei Projektinen ([Mic05])

3.1.1.2 Leseoptimierter Speicher (RS)

Im RS werden die Daten im Gegensatz zum WS sortiert und komprimiert gespeichert. Auch werden die Daten des RS im Sekundärspeicher gesichert, während die Seiten des WS vornehmlich im Datenbankpuffer liegen. Im RS wird jede Spalte einer Projektion mit deren Sortierschlüssel sortiert. Die Daten werden im RS dicht abgelegt, d.h. es wird kein Freiraum

in den Seiten gelassen (wie z.B. `pctfree`-Parameter bei Oracle), da Inserts im RS nur durch sogenanntes Bulk Loading (also Inserts großer Datenmengen) durch den Tuple Mover stattfinden (Seite 8 [VertWhitePaper]). Dadurch wird im Vergleich zu **roDBMS** weiterer Platz eingespart.

Der Speicherschlüssel SK eines Tupels wird hier im Gegensatz zum WS nicht physisch gespeichert, sondern ist die Ordinalzahl des Tupels im Segment (d.h. ergibt sich aus der Anordnung der Tupel).

Prefetching

Da die Werte einer Spalte zwar in aufeinanderfolgenden Blöcken, die verschiedenen Spalten aber separat gespeichert werden, besteht wegen der im Vergleich zur Übertragungszeit hohen Suchzeit die Gefahr, dass der Lesekopf bei Einlesen mehrerer Spalten relativ viele Suchoperationen durchführen muss. Um dies zu verhindern, liest C-Store per Default 64 Blöcke (entspricht 4 MB) einer Spalte im Voraus ein (vgl. Kapitel 3.3.1 [Aba08]). Der Wert ist allerdings konfigurierbar. Beim Einlesen mehrerer Spalten ist eine Suche also nur alle 64 Blöcke notwendig. Dieser Ansatz ist natürlich nur sinnvoll bei Anfragen, die nicht nur ein bestimmtes Tupel suchen (**OLTP**-Anfragen), z.B. aggregierende und summierende Anfragen im Rahmen von Data Warehouses.

Kodierschemata (Kompression)

Zum Einsatz von Datenkompression in **DBMS** und den Gründen dafür siehe Kapitel 2.12.4.

Spalten werden im RS durch eine von vier möglichen Kodierungen komprimiert. Die Wahl der Kodierung ist dabei abhängig von der Sortierung der Spalte und wird unterschieden nach

1. der Sortierung (nach der eigenen (Selbst-) oder einer anderen Spalte der Projektion (Fremd-Sortierung)) sowie dem
2. Anteil unterschiedlicher Werte in der Spalte (wenig/viele).

1. Kodierungstyp (Selbst-Sortierung, wenig unterschiedliche Werte):

Die Spalte ist eine Sequenz von Tripeln *(v,f,n)* mit *v:* Wert in der Spalte, *f:* Position des ersten Auftretens und *n:* Anzahl des Auftretens von *v* in Spalte. Es handelt sich also um eine Lauflängenkodierung, die insbesondere bei den sortierten Projektionen von C-Store effizient genutzt werden kann [Seite 49 [Aba08]].

Beispiel: (4,12,7) komprimiert den Wert 4, der in den Zeilen 12-18 der Spalte auftritt.

Bei Selbst-Sortierung erfordert dies lediglich ein Tripel für jeden unterschiedlichen Wert. Zur Anfrageunterstützung haben Typ-1-kodierte Werte geclusterte B-Baum-Indizes (Kapitel 2.11.3) über ihren Werten [Mic05]. Da der RS keine Online-Updates erfordert, können die Seiten des Indexes dicht gepackt werden (wie die übrigen Seiten auch). Bei einer Blockgröße von 64k-128k muss der Index maximal zweistufig sein.

2. Kodierungstyp (Fremd-Sortierung, wenig unterschiedliche Werte):

Eine Spalte mit Typ-2-Kodierung ist eine Sequenz von Tupeln (v,b) mit *v:* Wert in der Spalte und *b:* Bitmap, die die Positionen des Wertes in der Spalte angibt.

Beispiel: Die Integer-Werte 0,0,1,1,2,1,0,2,1 in einer Spalte werden kodiert als 3 Paare (0,110000100) (1,001101001) und (2,000010010). Da jede dieser Bitmap dünn besetzt ist (engl. sparse bitmap), werden sie mit Lauflängenkodierung komprimiert.

Zur Beschleunigung des Zugriffs werden Offset-Indizes (B-Bäume) eingesetzt, die Positionen in einer Spalte auf die Werte abbilden, um auf den i-ten Wert einer Spalte schneller zugreifen zu können.

3. Kodierungstyp (Selbst-Sortierung, viele unterschiedliche Werte):

Der Ansatz ist, jeden Spaltenwert als Delta vom vorigen Wert darzustellen (Delta-Kodierung, siehe Kapitel 2.12.4).

Beispiel: Eine Spalte mit den Werten 1,4,7,7,8,12 wird kodiert als 1,3,3,0,1,4. Es wird eine an VSAM-B-Baum-Indexschlüsseln angelehnte blockorientierte Form verwendet, bei dem der erste Wert des Blocks ein Wert der Spalte mit dem dazugehörigen Speicherschlüssel ist, und jeder nachfolgende Wert ein Delta [Mic05]. Der Zugriff auf die komprimierten Werte kann auch hier durch einen B-Baum-Index auf Blocklevel-Ebene beschleunigt werden.

4. Kodierungstyp (Fremd-Sortierung, viele unterschiedliche Werte):

Ggf. ist es hier besser, auf Komprimierung zu verzichten. Aber ein B-Baum kann auch hier für die Indizierung verwendet werden

Neben den genannten Kodierverfahren implementiert Vertica auch die Lempel-Ziv-Kodierung als Kompressionsverfahren ein, jedoch wird Lempel-Ziv nur im Notfall angewendet, wenn die Kodierungstypen 1-3 nicht anwendbar sind (Seite 6 [VertCompr]).

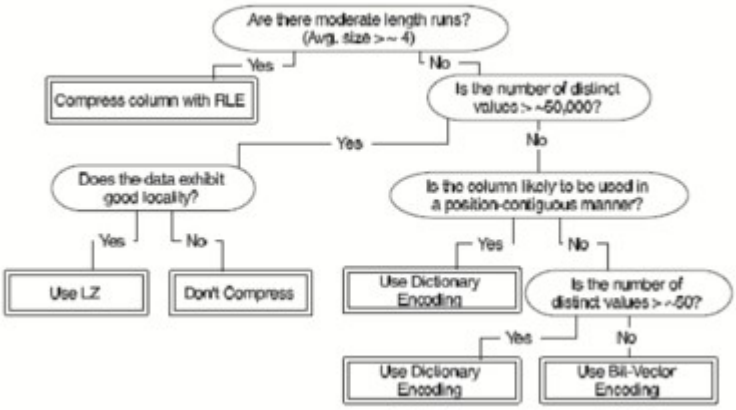

Abbildung 21: Entscheidungsweg für Kompressionsverfahren in C-Store (Seite 65 [Aba08])

Abbildung 21 zeigt das Auswahlschema von C-Store für die Kompression einer Spalte. Inwieweit das Auswahlschema bei Vertica Anwendung findet, ließ sich nicht klären. „Exhibits good locality“ bedeutet entweder, dass die Spalte eine der Sortierspalten in der Projektion ist oder anderweitig sich wiederholende Muster enthält. Die komprimierten Daten werden in Blöcken a 64K abgelegt (wie bei Vertica), das Kompressionsverfahren ist einheitlich für den Block. „Likely to be used in an position contiguous manner“ bezieht sich auf die Art der Positionsspeicherung der Daten. Ein Bitvektor ist non-position contiguous, da er in der Regel die nicht aufeinanderfolgenden Positionen eines Wertes enthält. Wird die Spalte in einer WHERE-Klausel herangezogen, ist ein „Position Contiguous“-Zugriff nicht nötig, als Bestandteil der SELECT-Klausel ist er insbesondere bei Einsatz der GROUP BY-Klausel vorteilhaft.

3.1.1.3 Schreiboptimierter Speicher

Der schreiboptimierte Speicher (engl. *Writeable Store* (WS)) ist ebenso wie der RS spaltenorientierter Speicher und beinhaltet dieselben physischen Dateiorganisationsformen (Projektionen) wie der RS [VertWC08], [Mic05]. Aufgrund des Ziels effizienter transaktionaler Updates findet aber keine Datenkomprimierung und Sortierung der Daten statt [Seite 4 [VertWhitePaper]]. Auf die Datenkomprimierung wird wegen der im Verhältnis zum RS trivialen Größe des WS verzichtet.

Der Speicherschlüssel *SK* (logische Tupel-ID) jedes Tupels ist im Gegensatz zum RS in jedem Segment (einer Projektion) gespeichert. Jedem Insert eines logischen Tupels (d.h. jeder

Zeile) in Tabelle T wird ein eindeutiger SK gegeben. Dieser SK muss in jeder Projektion, die dieses logische Tupel speichert, verwendet werden. Der Grund für die Speicherung des SK im WS und die Nichtspeicherung im RS wird leider nicht näher erläutert. Im RS wird aufgrund der Sortierung und Verwendung von Join-Indizes darauf verzichtet. Denkbar ist, dass im (unsortierten) WS durch Verwendung des SK für den Tuple Mover kenntlich gemacht wird, auf welches Tupel des RS sich ein Update bezieht.

Der WS wird horizontal genauso partitioniert wie der RS, es gibt also eine 1:1-Abbildung zwischen WS- und RS-Segmenten. Ein Paar *(SID,SK)* identifiziert also sowohl in RS als auch im WS die gleichen Tupelwerte.

Bezüglich des Einsatzes von Indizes im WS gibt es widersprüchliche Aussagen für C-Store und Vertica. Gemäß [Mic05] werden zur Herstellung einer logischen Sortierung des WS in C-Store B-Bäume eingesetzt, gemäß [Seite 12 [VertWhitePaper] und Seite 51 [VertConcepts]] ist der WS von Vertica nicht indiziert. Es ist zu vermuten, dass die Indizierung im WS von C-Store prototypischen Charakter hatte. Zum Nachlesen wird auf [Seite 6 [Mic05]] verwiesen.

Der WS wird durch den Tuple Mover regelmäßig geleert bzw. die Updates werden in den RS überführt [Seite 4 [VertWC08]].

3.1.1.4 Speicherverwaltung

Grundsätzlich ist bei einer spaltenorientierten Speicherarchitektur der Speicher eine Menge von Spalten. Die Speicherverwaltung verteilt die Segmente auf die Knoten eines Gridsystems. Diese Aufgabe erledigt der *Storage Allocator* von C-Store. Alle Spalten eines Segments einer Projektion sollten zusammen angeordnet sein, inklusive der dazugehörigen Join-Indizes. Außerdem wird jedes WS-Segment zusammen mit seinem entsprechenden RS-Segment angeordnet.

3.1.1.5 Transaktionen

C-Store ist auf viele Ad-hoc-Anfragen mit großen Datenmengen und wenige **OLTP**-Transaktionen mit wenigen Tupeln ausgelegt.

Bei konventionellem Locking würden, wie oben bereits geschrieben, Sperr-Konkurrenzsituationen zu schlechter Performance führen. Daher werden Readonly-Transaktionen durch Schnappschuss-Isolierung (engl. *Snapshot Isolation*) unterstützt.

Schnappschuss-Isolierung erlaubt es Readonly-Transaktionen, auf einen historischen Stand der Datenbank zuzugreifen, von dem garantiert werden kann, dass es keine uncommitteten Transaktionen gibt. Bei Schnappschuss-Isolierung müssen Readonly-Transaktionen keine Locks setzen.

Updates werden in zeitbasierten *Buckets* (deutsch Eimern) fester Länge, den sogenannten Epochen, gesammelt. Die Zeitdauer einer Epoche ist dabei konfigurierbar. Zu festen Intervallen schließen C-Store/Vertica eine Epoche und öffnen die nächste. Neue Updates werden in der neuen Epoche gespeichert (im WS), Daten in älteren Epochen stehen für Anfragen und eine Migration in den RS zur Verfügung, wenn keine Schreibtransaktionen mehr im WS auf ihnen arbeiten. Sind alle schreibenden Transaktionen in einer Epoche n-1 beendet, können Readonly-Transaktione auf die Daten dieser Epoche zugreifen.

3.1.1.6 Anfrageausführung

Zur allgemeinen Funktionsweise der Anfrageverarbeitung in einem DBMS wird auf Kapitel 2.14 verwiesen.

Das dort beschriebene Interator-Prinzip ist in C-Store (und vermutlich auch Vertica) so umgesetzt, dass statt einzelnen Tupeln 64K-Datenblöcke (Werte einer Spalte) von den Operatoren weitergegeben. Außerdem bilden die Anfragepläne keine Baumstruktur mehr, sondern Operatoren können mehrere Elternknoten haben. Beispiel hierfür ist die Anfrage

```
SELECT shipdate, linenum
FROM lineitem
WHERE shipdate < CONST1 AND linenum < CONST2
```

, deren Anfrageplan in Abbildung 22 dargestellt ist.

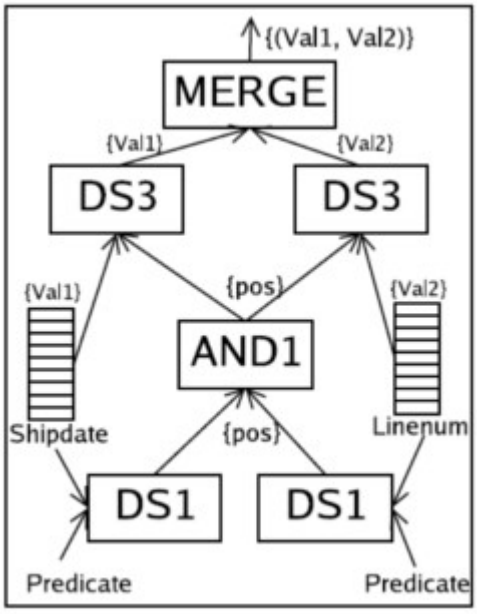

Abbildung 22: Ein spaltenorientierter Anfrageplan (Seite 43 [Aba08])

DS1 und DS3 sind dabei zwei von 4 möglichen *Data Source (**DS**)*-Operatoren. Diese Operatoren lesen Atttribute/Spalten vom Sekundärspeicher und führen Filteroperationen (Selektionen) auf ihnen durch. Sie haben außerdem Wissen über die Komprimierung der Daten. Nachfolgend werden die 4 Operatoren beschrieben (Seite 71 [Aba08]):

1. DS1 liest eine Spalte bzw. deren Blöcke ein und wendet ein Prädikat auf die Spalte an. Der Operator gibt als Ergebnis eine Spalte mit Positionen der gültigen Werte aus.
2. DS2 arbeitet wie DS 1, liefert aber als Ergebnis eine Spalte mit Paaren (Position,Wert) der gültigen Werte der Spalte.
3. DS3 liest eine Spalte bzw. deren Blöcke ein und filtert die Werte nach einer Positionsliste. Ausgegeben wird eine Liste mit den entsprechenden Werten.
4. DS4 liest eine Spalte bzw. deren Blöcke ein

Bei Einsatz von Kompressionsverfahren wie z.B. Lempel-Ziv werden die Daten vom DataSource-Operator zuerst dekomprimiert (vgl. Kapitel 4.4.1 [Aba08]), bevor sie an Eltern-Operatoren weitergegeben werden.

In Abbildung 22 liest DS1 eine Spalte und wendet das Prädikat auf die Spalte an. Das Ergebnis ist eine Liste mit Positionen der Werte, die das Prädikat erfüllen. Diese werden dann mit dem AND1-Operator verknüpft. Bis zu diesem Knoten bilden die Operatoren einen Baum. Nun müssen aus den Spalten die Positionen extrahiert werden, die beide Prädikate erfüllen. Dies wird durch die Operatoren DS3 ausgeführt, die aber beide die Ausgabe des

AND-Operators benötigen. Somit hat der AND-Operator zwei Elternknoten, die Operatoren bilden also keinen Baum mehr. Die Gefahr, dass ein Eltern-Operator (z.B. DS3) die Funktion *getNextBlock* schneller aufrufen könnte als ein langsamerer Nachbarelternknoten wird dadurch gelöst, dass es weiterhin nur einen Wurzelknoten gibt, der die Aufrufe weitergibt und dies erst dann macht, wenn alle Daten des aktuellen Blocks verarbeitet wurden. Der durch den aktuellen Block belegte Speicher kann dann vollständig freigegeben werden. Deutlich wird bei dem beschriebenen Beispiel, dass bei der Anfragebearbeitung

- nur die benötigten Spalten eingelesen werden (geringerer I/O/Aufwand), Projektionen also praktisch entfallen [Aba06],
- lediglich Positionslisten (z.B. als Bitstrings) statt kompletter Tupel zwischen den Operatoren weitergegeben werden und die Verbindung zum Ergebnistupel erst am Ende stattfindet, was die Zwischenergebnisse klein hält,
- das Erzeugen von Positionen (der beiden Spalten) statt Werten (d.h. zusammengebauter Tupel) bei Verbunden, z.B. nach einem Nested-Loop-Join [Aba06], und an andere DS-Operatoren zur Extraktion weiterer Werte aus anderen Spalten weitergereicht werden, siehe Abbildung 23.

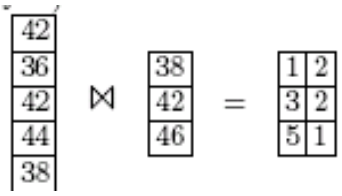

Abbildung 23: Positionsliste bei Verbundoperation [Aba06]

Weitere Optimierungsmöglichkeiten sind:

- Bei einem Nested-Loop-Join von 2 Spalten, bei denen einen unkomprimiert und eine RLE-komprimiert ist, kann die Anzahl der Operationen um den Faktor k vermindert werden, wobei k die Lauflänge des RLE-Tripels ist, das mit einem Wert der unkomprimierten Spalte übereinstimmt. Die resultierenden Positionsspalten (siehe obige Abbildung) können für die rechte Spalte direkt in RLE ausgegeben werden (siehe auch Kapitel 5.2 [Aba06]).

- Ebenfalls bei Verbunden kann, wenn einer der Spalten bitvektor-kodiert ist, kann der Bitvektor des übereinstimmenden Wertes in das Ergebnis kopiert werden. Auch dies spart etliche Vergleichsoperationen (vgl. Abb. 4-1 [Aba08]).
- Selektionsprädikate können in die DataSource-Operatoren weitergereicht werden,
- die Kodierung kann bei der Ausführung der Operationen berücksichtigt werden und zu einer Effizienzsteigerung führen, bei einem Gleichheits-Prädikat z.B. kann der DS-Operator für bitvektor-kodierte Daten (Typ 2) nur den Bitvektor für den anfragten Wert zurückgeben, die Selektion wird dadurch zeitsparend.
- Bei einer wörterbuchkodierten Spalte wandelt die DataSource den Prädikatwert in seinen Wörterbucheintrag um und führt den Vergleich ohne Dekomprimierung durch.
- Auch ein direktes Arbeiten auf wörterbuchkodierten Daten z.B. bei der Gruppierung ist denkbar. Angenommen die Werte 2, 4 und 8 würden durch die Symbole 000,001 und 002 repräsentiert werden, dann könnte bei Gruppierung und Aggregierung von 001,001,000,001,002,002 zunächst ein Zählen der Werte stattfinden mit dem Ergebnis: 001:3, 000:1,002:2 und danach würde erst die Dekodierung anhand des Wörterbuchs und die Summenbildung durchgeführt werden.

Das Problem des doppelten Zugriffs auf die Spalten durch DS1 und DS3 in Abbildung 22 wird durch Materialisierungsstrategien gelöst, auf die noch eingegangen wird. Die wesentlichen weiteren Operatoren neben den DS-Operatoren sind ([Mic05] und Anhang A [Aba08]):

1. Decompress: Dekomprimiert eine Spalte,
2. Select: Wie Selektion bei relationaler Algebra, erzeugt aber statt Restriktion der Eingabe eine Bitstring-Darstellung,
3. Mask: Akzeptiert Bistring B und Projektion Cs, beschränkt Cs durch Zulassen nur der Werte, dessen Bits in B 1 sind.,
4. Project: Wie bei relationaler Algebra,
5. Sort: Sortiert Spalten einer Projektion nach einer Untermenge der Spalten,
6. Aggregation Operators: Berechnen Aggregation über einer Spalte,
7. Concat: Verbinden 1-n Projektionen mit selber Sortierung zu einer Projektion,
8. Permute: Wandelt eine Projektion nach der Ordnung eines Join-Index,
9. Join: Verbindet 2 Projektionen anhand des Prädikats,

10. Bitstring Operators: Bitweises AND und Bitweises OR von Bitstrings sowie NOT (Komplement eines Bitstrings).

Iteratoren

Die Operatoren übergeben nicht die Blöcke selbst, sondern Iteratoren. Ein Iterator ist eine Zeigerliste auf Werte eines geladenen Blocks. Er hat hat eine Start- und Endeposition und gibt Blockwerte durch die Methode *getNext* zurück.

3.1.1.7 Materialisierungsstrategien in C-Store

Durch physische und logische Datenunabhängigkeit ist es auch bei einem spaltenorientierten DBMS möglich, relationale Datenbankschnittstellen wie **JDBC** und **ODBC** zu unterstützen (Kapitel 2.2). Das bedeutet aber, dass die auszugebenden Attribute zu Tupeln zusammengefügt werden müssen. Wann dies im Anfrageplan geschehen muss, ist genau das Gegenteil des Problems in zeilenorientierten DBMS, wann Projektionen stattfinden müssen[23], denn anstatt ein Attribut aus einem Zwischenergebnis so früh wie möglich herauszulösen, muss hier eins eingefügt werden. Eine Spalte wird hinzugefügt, sobald sie von einem späteren Operator oder im Ergebnis benötigt wird. Das Hinzufügen von Spalten zu Zwischenergebnissen zwecks Tupelbildung wird in [Aba08] *Materialisierung*[24] genannt. Eine Materialisierungsstrategie wird benötigt, sobald auf mehr als ein Attribut einer Relation zugegriffen wird.

Die möglichen Ansätze sind frühe und späte Materialisierung. Frühe Materialisierung bedeutet, dass die Spalten möglichst früh zu Tupeln verbunden werden, späte Materialisierung das Gegenteil. Es ist abzuwägen, welcher Ansatz der effizienteste ist.

Wenn eine Anfrage aus drei Selektionen $\sigma_1, \sigma_2, \sigma_3$ über den Spalten R.a, R.b und R.c (abgespeichert in 3 separaten Dateien und in gleicher Reihenfolge sortiert) ausgeführt wird, wobei o_1 die selektivste und o_3 die am wenigsten selektive ist, würde eine frühe Materialisierung wie folgt ablaufen: Es werden die Blöcke von R.a, R.b und R.c eingelesen, zu Tupeln zusammengefügt und $\sigma_1, \sigma_2, \sigma_3$ angewendet. Diese Strategie ist ineffizient, denn da

[23]Projektione sollen möglichst früh ausgeführt werden (genau wie Selektionen), um Zwischenergebnisse durch Entfernen nicht benötigter Attribute (Hinweis: Bei Selektionen werden Tupel entfernt) klein zu halten (siehe u.a. Seite 774 [GMUW09])

[24]Unter Materialisierung wird in der Fachliteratur in der Regel das Speichern von Zwischenergebnissen im Sekundärspeicher verstanden (Seite 830 [GMUW09])

die Selektivität von σ_1 und σ_3 unterschiedlich ist, wurden viele Tupel umsonst zusammengebaut, denn sie werden kurz danach wieder aussortiert. Die späte Materialisierung, bei der die Tupel zum spätestmöglichen Zeitpunkt zusammengebaut werden, ist hier effizienter. Im genannten Beispiel könnte der Anfrageplan wie folgt aufgebaut sein:

1. Lese R.a und gebe die Positionen aus, die σ_1 erfüllen (z.B. in Form eines Bitstrings) und wiederhole dies für R.b und R.c (respektive σ_2 und σ_3).
2. Wende einen AND-Operator auf die Positionslisten an.
3. Greife danach wiederum auf R.a, R.b und R.c zu und füge die Werte anhand der Positionsliste (die Positionen oder Tupel-Identifier beinhaltet) zu Tupeln zusammen.

Dieses Vorgehen kann effizienter sein, da weniger Tupel zusammengefügt werden müssen (vergleichbar mit einer Join-Operation) und Positionslisten einfache Datenstrukturen sind, auf denen direkt gearbeitet werden kann. Nachteil ist aber, dass erneut auf die Basisrelationen (bzw. bei C-Store Projektion) zugegriffen werden muss.

Späte Materialisierung erlaubt es dem *Query Processor*, neben der Erzeugung von weniger Tupeln außerdem Operationen auf komprimierten Spalten auszuführen und ggf. weniger Tupel zu erzeugen. Die meisten Anfragen beinhalten ein oder mehrere Selektionsprädikate. Das Ergebnis der Anwendung der Prädikate ist eine Menge von Positionen einer Spalte, deren Werte das Prädikat erfüllen. Wenn mehrere Prädikate angewendet werden, werden verschiedene Positionsmengen für verschiedene Spalten erzeugt. Positionen können durch Komprimierungstechniken dargestellt werden, z.B. durch [startpos,endpos] bei aufeinanderfolgenden Positionen oder durch Bitvektoren. Ein Bitvektor 0111010001 für die Positionen 11-20 würde z.B. anzeigen, dass die Positionen 12, 13, 14, 16 und 20 das Selektionsprädikat erfüllen. Auf diesen Darstellungen kann direkt gearbeitet werden, z.B. mit AND- und OR-Verknüpfungen ([Seite 68 [Aba08]]).

Der Nachteil der späten Materialisierung ist, dass auf Spalten im Anfrageplan mehrere Male zugegriffen werden muss. Zum Beispiel, um beim ersten Zugriff die Positionen zu erhalten, die das Prädikat erfüllen, und beim zweiten Zugriff, um die Werte zu erhalten (wenn man die Positionen nicht aus einem Index erhalten kann). Bei früher Materialisierung werden die Werte sofort in Zwischenergebnis-Tupel geschrieben. Wenn trotz der Performance-Optimierungen der späten Materialisierung (Arbeiten mit Positionsdaten, Erzeugen nur relevanter Tupel, Arbeiten mit komprimierten Daten) die Kosten für einen erneuten Zugriff zur Tupelkonstruktion überwiegen, hat die frühe Materialisierung Vorteile. Ein Vergleich von

früher und später Materialisierung anhand eines Kostenmodells findet sich in [Kapitel 5.3 [Aba08]], steht aber nicht im Fokus der Betrachtungen dieser Arbeit.

Nachfolgend sollen Ansätze in C-Store (und ggf. auch Vertica[25]) für frühe und späte Materialisierung aufgezeigt werden, nämlich *pipelined*- und *parallele* Verarbeitung. Mit Pipelining ist jedoch nicht das in der Datenbankliteratur bekannte *Pipelining* bei der Anfrageverarbeitung gemeint (als Gegensatz zur Materialisierung von Zwischenergebnissen, siehe u.a. [Seite 665 [CB05]]), sondern die Aufteilung der Tupelbildung in mehrere Schritte (statt einmalig stattfindender). Die Anfrage

```
SELECT shipdate, linenum FROM lineitem
WHERE shipdate < CONST1 AND linenum < CONST 2
```

kann je nach Verwendung von pipelined- und paralleler Verarbeitung zu unterschiedlichen Anfrageplänen führen. Anfrageplan a (Abbildung 24) scannt mit einem DS2-Operator die shipdate-Spalte und erzeugt (pos,shipdate)-Tupel, die das Prädikat shipdate < CONST1 erfüllen. Diese Zwischenrelation wird als Eingabe zusammen mit der linenum-Spalte und dem Prädikat linenum < CONST2 vom DS4-Operator verwendet, um (shipdate,linenum)-Ergebnistupel zu erzeugen. Das zweite Prädikat wird nur noch bei den Werten angewandt, für die es im Eingabestrom entsprechende Werte aus shipdate gibt. Die Tupel werden inkrementell erzeugt.

Der parallele Anfrageplan b (Abbildung 24) erzeugt die Tupel am Anfang des Plans (Kartesisches Kreuzprodukt) und wendet dann darauf die Prädikate an (*SPC-Operator*: Scan, Predicate and Construct).

Der erste Ansatz wendet die beiden Prädikate separat an, d.h. Bei 100 Zeilen fallen 100 Vergleichsoperationen mehr an, dafür aber, wenn nicht gerade alle Zeilen selektiert werden, weniger.

Der zweite Ansatz muss alle Blöcke scannen, der erste Ansatz kann je nach Selektivität ggf. ganze Blöcke überspringen, muss dafür aber pro DS-Operator ein Prädikat anwenden (d.h. Einen Vergleich durchführen).

Bei niedriger Selektivität (d.h. wenn der DS2-Operator wenige Werte das Prädikat erfüllen) ist ggf. der erste Ansatz performanter).

[25]Ist mangels Literatur nicht nachvollziehbar

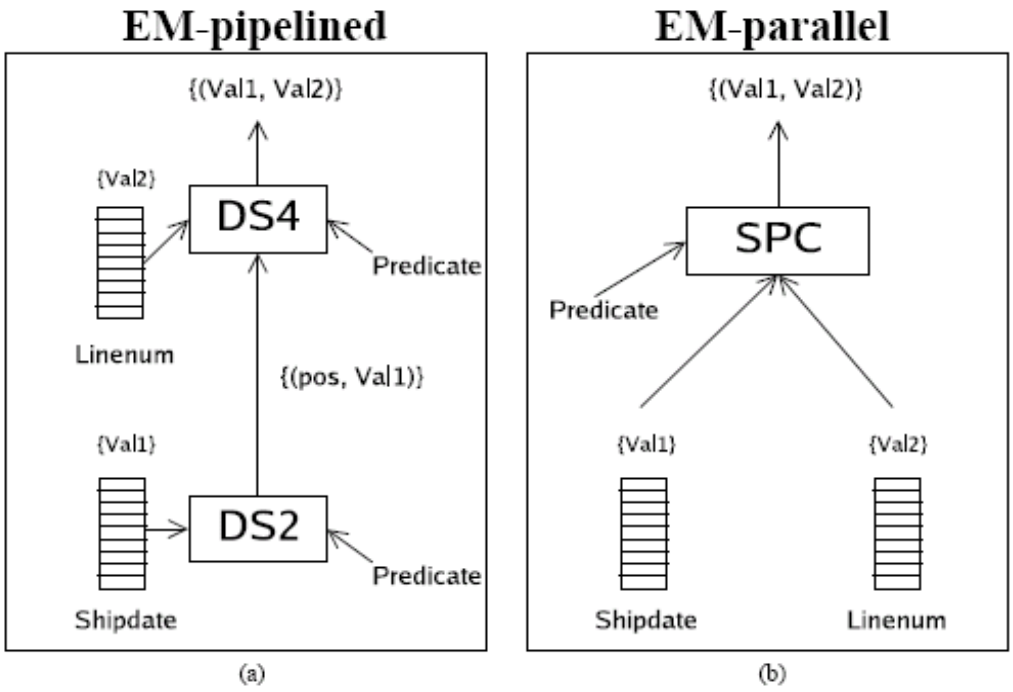

Abbildung 24: Anfragepläne für frühe Materialisierung [Aba08]

Wie bei früher Materialisierung, gibt es auch bei später Materialisierung den Pipeline- und Parallel-Ansatz, siehe Abbildung 25. Der parallele Ansatz (a) beginnt beim genannten Beispiel mit zwei DS1-Operatoren, einen für die Spalte *shipdate* und einen für *linenum*. Jeder DS1-Operator scannt seine Spalte und erzeugt unter Anwendung des Prädikats eine Positionsliste. Die Listen werden nach Verknüpfung zu zwei DS3-Operatoren weitergeleitet, die aus beiden Spalten die entsprechendenWerte extrahieren. Diese Werte werden durch den MERGE-Operator zu Ergebnistupeln zusammengefügt.

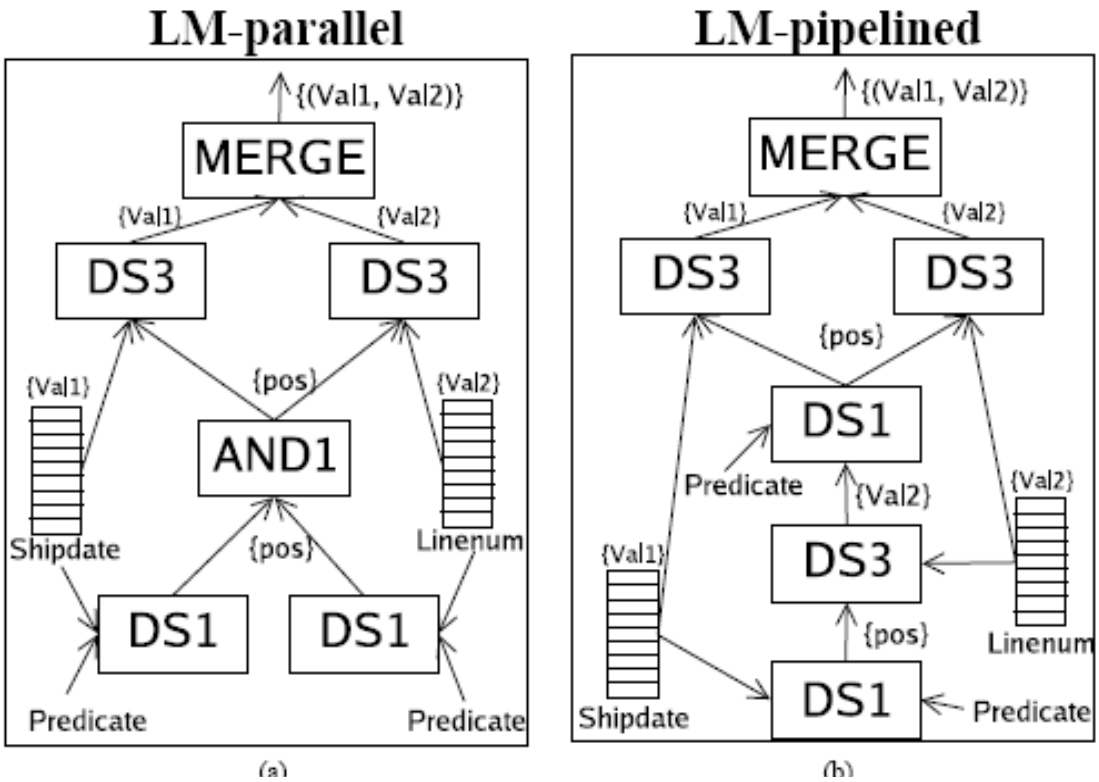

Abbildung 25: Anfragepläne für späte Materialisierung [Aba08]

Der Pipeline-Ansatz (b) in Abbildung 25 arbeitet ähnlich. Hier wird die durch DS1 erzeugte Positionsliste einer Spalte an DS3 gesandt, die für diese Positionen die Werte aus der *linenum*-Spalte liest und diese an einen weiteren DS1-Operator sendet, der auf diese *linenum*-Teilmenge (und nicht auf alle Werte der Spalte) das Prädikat anwendet. Als Seiteneffekt wird kein *UND*-Operator mehr benötigt.

Zur Vermeidung von I/O-Kosten bei später Materialisierung beim erneuten Zugriff auf die Spalten durch den DS3-Operator (der Werte aus der gemeinsamen Positionsliste erzeugt), werden diese in Form von *Multicolumns* (Zusammenschluss mehrerer *Minicolumns*[26]) mit entsprechenden Positionslisten, die die Gültigkeit des Werts nach Anwendungs des Prädikats angeben, im Hauptspeicher abgelegt, sodass später wieder auf sie zugegriffen werden kann. Eine *Multicolumn* (Abbildung 26) ist eine Datenstruktur zum Abspeichern von Zwischenergebnissen bzw. eine im Hauptspeicher hinterlegte horizontale Partition einer Teilmenge von Attributen einer Relation. Sie besteht aus

- einem Positionsbereich, der die virtuelle Start- und Endposition der horizontalen Partition angibt,

[26]Das entsprechende Gegenstück zu den C-Store-Minicolumns nennt sich bei MonetDB „Vector“

- einem Array von Minicolumns[27]. Der *Grad* einer Multicolumn ist die Größe des Minicolumn-Arrays , d.h. die Anzahl seiner Attribute. Jede Minicolumn wird im Hauptspeicher auf dieselbe Art komprimiert wie im Sekundärspeicher.
- Einer Positionsbeschreibung, die anzeigt welche Positionen im Positionsbereich gültig bleiben. Positionen werden ungültig, wenn Prädikate auf die Multicolumns angewendet werden.

Die *Positionsbeschreibung* kann eine von drei Formen haben:

1. *Positionsgültigkeitsbereich*: Alle Positionen zwischen einer angegebenen Start- und Endeposition sind gültig
2. *Bitmappositionen*: Ein Bitvektor der Größe des des Positionsbereichs der Multicolumn zeigt durch '1' die Positionen an, die das Prädikat erfüllen
3. *Positionsliste*: Eine Liste gültiger Positionen innerhalb des Positionsbereichs.

Wenn eine Seite eines Attributs gelesen wird (z.B. durch DS1-Operator), wird eine Minicolumn (die ein Zeiger auf die Seite im Datenbankpuffer ist) mit einer Positionsbeschreibung erzeugt, die zunächst alle Positionen als gültig markiert. Der DS1-Operator durchläuft dann die Attributwerte, wendet das Prädikat auf jeden Wert an und erzeugt eine neue Liste gültiger Positionen.Die Multicolumn ersetzt ihre Positionsbeschreibung durch die neue Liste. Ein UND-Operator erzeugt aus zwei Multicolumns mit überlappenden Positionsbereichen eine neue Multicolumn, bei der der Positionsbereich und die Positionsbeschreibung gekreuzt sind. Die UND-Verknüpfung von Multicolumns ist im Endeffekt das gleiche wie die UND-Verknüpfung von Positionslisten.

Der Unterschied ist, dass hier zusätzlich Zeiger auf die Minicolumns bei der Ergebnis-Multicolumn kopiert werden müssen. Wenn der UND-Operator Multicolumns statt Positionslisten erzeugt und an den DS3-Operator weiterleitet, muss dieser nicht auf die Spalte zurückgreifen, sondern kann direkt auf dem Multicolumn-Block arbeiten. Durch die Multicolumn-Prinzip können Spaltenteilmengen im Anfragebaum zu den oberen Knoten durchgereicht werden.

[27]Eine Minicolumn ist eine Menge korrespondierender Werte für einen Positionsbereich

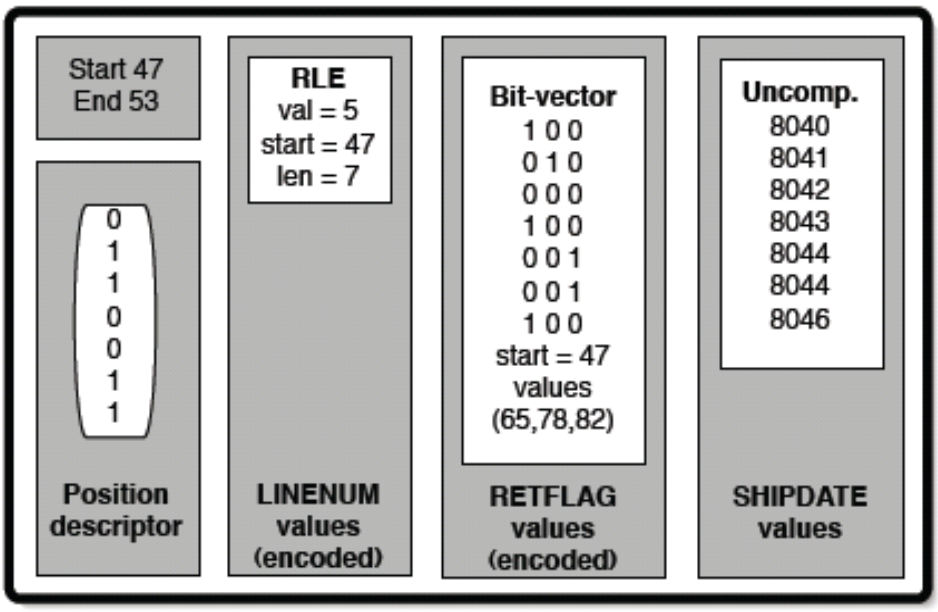

Abbildung 26: Multicolumn-Datenstruktur in C-Store [Aba08]

3.1.1.8 Ansätze von C-Store/Vertica

Tabelle 13 führt die ermittelten Ansätze von C-Store bzw. Vertica auf.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Unkomprimierte Speicherung sich überlappender unsortierter Projektionen mit Fremdattributen (Fremschlüsselbeziehung) im Writeable Store, unterschiedlich sortierter, komprimierter, überlappender Projektionen im Readable Store. Batch-Updates des Readable Stores durch den Tuple Mover. Speicherung jeder Spalte in separater Datei mit 64K-Blöcken. Automatische interne Indexgenerierung, Anwender muss sich darum nicht kümmern. Dünnbesetzte Indizes für Spaltenwerte und logische Tupel-IDs. Verwendung von Offset-Indizes bei Einsatz von Bitvektor-Kodierung zur Performanceoptimierung.
2	*Algorithmen* (DB-Operationen, Kompression)	Verschiedene Kompressionsverfahren, abgestimmt auf die Gegebenheiten der jeweiligen Spalte
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Relationales Datenmodell SQL als Anfragesprache, DDL und DML Anbindung verschiedener OLAP-Werkzeuge JDBC- und ODBC-Treiber
4	*Anfrageverarbeitung*	Prefetching von Blöcken

Nr.	Betrachtungsschwerpunkt	Ansatz
	(spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Spaltenorientierter Optimierer mit später Materialisierung, Pipelining und Multicolumns zur Vermeidung wiederholten Laden
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Nothing-Architektur. Verteilte Speicherung sich überlappender (horizontal partitionierter) Projektionen.
6	*Transaktionsverwaltung* (Update-Problematik)	Snapshot Isolation zum Lesen eines konsistenten Datenbank-Standes
7	*Betriebssystem*	Linux

Tabelle 13: Ansätze / Klassifizierungskriterien für C-Store/Vertica

3.1.2 Infobright Data Warehouse (basierend auf MySQL)

Das als Data Warehouse konzipierte relationale **coDBMS** der Firma Infobright (ehemals Brighthouse) ist in das bekannte DBMS MySQL integriert, besitzt aber eine eigene Datenspeicher- und Anfrageoptimierungsschicht ([WikiIB]).

Die Daten der Relationen werden von Infobright in den drei Schichten *Datenpakete* (engl. *Data Packs*), *Datenpaketknoten* (engl. *Data Pack Nodes (DPN)*) und *Wissensknoten* (engl. *Knowledge Nodes (KN)*) vorgehalten. Datenpaketknoten und Wissensknoten sind Bestandteil des sogenannnten *Knowledge Grid* von Infobright. Abbildung 27 zeigt die Architekturebenen von Infobright. Das *Knowledge Grid* ist eine Metadatenschicht, das bedeutet es enthält Daten über die Daten, nämlich über Inhalte von und Beziehungen zwischen den Datenknoten. Das Knowledge Grid wird automatisiert, d.h. ohne Zutun des Anwenders, erstellt. Infobright nutzt keine traditionellen Indizes zur Beantwortung analytischer Anfragen. Die Elemente des Knowledge Grid, die Wissensknoten und Datenpaketknoten, beschreiben im Gegensatz zu Indizes nicht einzelne Datensätze, sondern große Datenmengen (Indizierung auf Datenpaketebene). Das Knowledge Grid wird in [Bright08] als Mediator zwischen dem Query Processor und der Speicherschicht beschrieben. Nachfolgend werden die einzelnen Schichten von Infobright beschrieben.

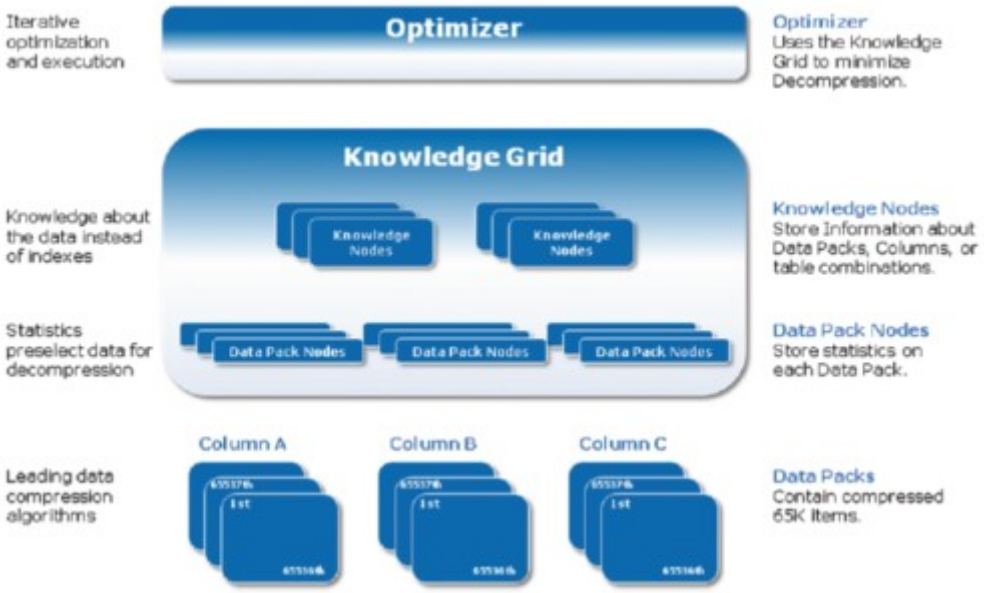

Abbildung 27: Architekturebenen von Infobright ([IBWP08])

Datenpakete enthalten die Tupel der Relationen. Jedes Datenpaket besteht aus 2^{16} (65.536) Zeilen eines Attributs einer Relation unter Verwendung von Datenkompression (vgl. [WikiIB] und [IBWP08]). Ein Datenpaket ist also ein Bereich eines Attributs. Datenpakete sind so gesehen horizontal und vertikal partitionierte Relationen. Ein Beispiel für Datenpakete aus [IBWP08] für eine Relation T mit 300.000 Tupeln und den Attributen A, B und C ist in Abbildung 28 dargestellt. Die Pakete $A_1,...A_5$ enthalten z.B. die (komprimierten) Werte des Attributs A, aufgeteilt in Einheiten zu je 65.536 Werten. Gemäß [Seite 2 [Bright08]] werden die Zeilen von Infobright niemals sortiert, um beim Laden der Daten in das Data Warehouse (aus den Datenquellen) eine bessere Performance zu erreichen. Eine Begründung für die Wahl der Größeneinheit 2^{16} wird in der recherchierten Literatur nicht gegeben.

Abbildung 28: Beispiel für Datenpakete (Data Nodes) bei einer Relation mit 3 Attributen ([IBWP08])

Datenpaketknoten (DPN) enthalten analytische Informationen über die in jedem Datenpaket gespeicherten und komprimierten Daten. Zwischen Datenpaketen und Datenpaketknoten besteht eine 1:1-Beziehung. Für nummerische Datentypen werden z.B. die Angaben minimaler und maximaler Wert, Summe sowie Anzahl der Elemente die nicht NULL sind. Angenommen, die Attribute A und B aus Abbildung 28 enthalten nummerische Werte, dann könnten die Informationen der insgesamt 10 Datenpaketknoten (für 10 Datenpakete) wie in Tabelle 14 beispielhaft aufgeführt aussehen. Auf DPN kann ohne vorherige Dekomprimierung der Datenpakete zugegriffen werden. Infobright untersucht, wie noch gezeigt wird, bei der Anfragebearbeitung den DPN eines Datenpakets, um festzustellen, ob das Datenpaket dekomprimiert werden muss. Manchmal reicht zur Anfrageverarbeitung auch der Inhalt des DPN. DPN verringern also bereits den Bedarf an Datendekompression bei der Anfrageverarbeitung, siehe [Seite 8 [IBWP08]].

Paket-Nummer Spalte A+B	**DPN von A**			**DPN von B**		
	Min	**Max**	**Summe**	**Min**	**Max**	**Summe**
Paket A1 + B1	0	5	10000	0	5	1000
Paket A2 + B2	0	2	2055	0	2	100
Paket A3 + B3	7	8	500000	0	1	100
Paket A4 + B4	0	5	30000	0	5	100
Paket A5 + B5	-4	10	12	-15	0	-40

Tabelle 14: Data Pack Nodes (DPN) für Datenpakete in Abbildung 25 ([IBWP08])

Wissensknoten (KN) sind weitere Elemente des Knowledge Grids und enthalten tiefergehende Informationen zu den Datenpaketen, z.B. wie sie zu anderen Daten in Beziehung stehen. Wissensknoten wurden für die Optimierung von Anfragen über mehrere Relationen (Joins, Subqueries (verschachtelte Anfragen)) entwickelt Das Ziel von Wissensknoten ist die Identifizierung von Datenpaketen, die zur Anfragebearbeitung nicht dekomprimiert werden müssen ([Bright08]).. Sogenannte Multi-Table-Pack-to-Pack-Wissensknoten geben an, welche Datenpaket-Paare von verschiedenen Tabellen bei der Verknüpfung der Tabellen tatsächlich berücksichtigt werden sollen. Es gibt verschiedene Typen von Wissensknoten, Beispiele dafür sind (aus [IBWP08], [Bright08])

1. *Histogramme*, die das Auftreten bestimmter nummerischer Werte in den Datenpaketen abbilden, ebenso Informationen über das Auftreten bestimmter alphanummerischer Werte im Datenpaket. Dazu wird für jedes Datenpaket der Bereich zwischen MIN- und MAX-Wert (gespeichert im Datenpaketknoten) in

1.024 Intervalle unterteilt. Enthält ein Datenpaket Werte, die innerhalb eines Intervalls liegen, wird der Wert für das Intervall auf 1 gesetzt, sonst auf 0. So kann der Optimierer feststellen, ob ein bestimmter Wert vorkommen kann, ohne das Datenpaket selbst dekomprimieren zu müssen. Die Wissensknoten sind also mit Bitmap-Indizes vergleichbar, arbeiten aber gemäß [IBWP08] mit Datenpaketen statt Tupeln, wobei aber anzumerken ist, dass Datenpakete nichts anderes als horizontal partitionierte Attribute sind. Wissensknotens werden automatisch während des Ladens der Daten erstellt.

2. *Character Maps (CMAP)* für alphanummerische Attribute. Die CMAP speichert Informationen über das Auftreten von Zeichen in Zeichenketten. Wenn es z.B. kein Wort mit einem „b“ an dritter Position gibt, ist der Wert hierfür 0 ([Bright08]).

3. *Pack-to-Packs* werden für Paare von Relationen gebaut.. Sie beziehen sich auf einen Joinbeziehung zweier Relationen basierend auf der Gleichheit zweier Attributwerte und sind nur Bestandteil des Knowledge Grid, wenn eine Joinbeziehung bereits in einer vergangenen Anfrage verwendet wurde. Ein Pack-to-Pack ist eine Binärmatrix, die mit sogenannten Row-Pack-IDs (ein Row Pack ist ein Synonym für ein Datenpaket, siehe [Bright08]) mit den Matrixwerten

   - 1, wenn es mindestens ein Wertpaar gibt, das die Verbundbedingung erfüllt oder
   - 0, wenn es kein solches Wertpaar gibt.

   Ein Pack-to-Pack-Wissensknoten ist in etwa vergleichbar mit dem in Kapitel 2.11.3.4 beschriebenen Verbundindex, jedoch mit dem Unterschied, dass der Pack-to-Pack-Knoten Verbundbeziehungen zwischen Datenpaketen (mit je 2^{16} Werten) und nicht zwischen einzelnen Tupeln beschreibt und dass der Pack-to-Pack-Knoten als Zugriffsstruktur für die Unterstützung von Ad-hoc-Anfragen konzipiert wurde (denn er unterstützt bei der Entscheidung, welche Datenpakete für eine Verbundoperation zu dekomprimieren sind), während der klassische Verbundindex lediglich den Aufwand für Verbundoperationen bei bekannten, wiederkehrenden Anfragen vermeiden soll.

3.1.2.1 Anfragebearbeitung

Der Infobright-Optimierer nutzt das *Knowledge Grid*, um die Menge der Datenpakete zu bestimmen, die dekomprimiert werden müssen, um eine Anfrage in kürzestmöglicher Zeit zu beantworten bzw. andersherum um zu entscheiden, welche Datenpakete zur Anfragebearbeitung nicht herangezogen, das heißt nicht dekomprimiert werden müssen. In manchen Fällen genügt die im *Knowledge Grid* enthaltene Information, um eine Anfrage ohne Dekomprimierung eines Datenpakets beantworten zu können. Gemäß [IBWP08] setzt der Optimierer die Theorie von Rough Sets ([Paw91], [Paw07]) unter Nutzung der Daten des Knowledge Grid ein, um zu entscheiden, welche Datenpakete dekomprimiert werden müssen. Rough Sets arbeiten mit Wertepaaren, die die unteren und oberen Näherungswerte einer Menge von Datensätzen angeben, sodass Datensätze nicht mehr sorgfältig in ihrer Gänze geprüft werden müssen. In Rough Sets werden Mengen in positive, negative und grenzwertige Regionen entsprechend einer Interessenslage (decision) eingeteilt. Positive Regionen enthalten die Daten, die mit der Entscheidung übereinstimmen, negative die nicht und grenzwertige solche, bei denen man nicht weiß, ob sie die Entscheidung voll, teilweise oder gar nicht stützen. Die Rough Sets als grobe Richtlinien ermöglichen eine schnelle Annäherung an die relevanten Datensätze für Datenbankabfragen, die Rechenleistung kann auf eine präzise Analyse einer relativ kleinen Datenmenge fokussiert werden.

Der Optimierer nutzt Informationen der Datenpaket- und Wissensknoten, um bei Anfragen die Datenpakete in die drei folgenden Kategorien einteilen zu können, welche den Regionen der Rough Sets entsprechen:

1. *Relevante Pakete*: Jedes Element (Werte) wird als anwendbar für die gegebene Anfrage identifiziert.
2. *Irrelevante Pakete*: Kein Element des Datenpakets wird als anwendbar für die Beantwortung der Anfrage identifiziert.
3. *Suspekte Pakete*: Es kann nicht festgestellt werden, ob Datenpaket vollständig relevant oder vollständig irrelevant für die Beantwortung der Anfrage ist.

Datenpaketknoten (bzw. ihre Min-Max-Werte eines Datenpakets) können z.B. zusammen mit den Histogrammen (Bitmap-Index auf Min-Max-Intervall) verwendet werden, um zu sagen, welche Datenpakete bei einer (nummerischen) `BETWEEN`-Bedingung ignoriert werden können (d.h. Irrelevant sind). *CMAP*-Wissensknoten können bei Prüfung einer `LIKE`-Bedingung bei der Entscheidung helfen, welche Datenpakete irrelevant sind. Die Arbeitsweise des Query Processors wird am nachfolgenden Beispiel erläutert.

Paket-Nummer Attribut A+B	DPN von A			DPN von B		
	Min	**Max**	**Summe**	**Min**	**Max**	**Summe**
Paket A1 + B1	0	5	10000	0	5	1000
Paket A2 + B2	0	2	2055	0	2	100
Paket A3 + B3	7	8	500000	0	1	100
Paket A4 + B4	0	5	30000	0	5	100
Paket A5 + B5	-4	10	12	-15	0	-40

Tabelle 15: Datenpaketknoten (DPN) für Datenpakete in Abbildung 25 ([IBWP08])

Bei dem SQL-Statement `SELECT SUM (B) FROM T WHERE A>6;` ergibt sich anhand der Datenpaketknoten in Tabelle 17 folgendes:

1. Relevant ist Datenpaket A3, da alle Werte `A>6` erfüllen. Daher ist auch Paket B3 relevant. Durch den DPN weiß der Optimierer, dass `SUM(B)=100` ist und muss Datenpakete A3 und B3 nicht dekomprimieren.
2. Irrelevant sind Datenpakekte A1, A2 und A4, da kein Paket `A>6` erfüllt. Folglich sind B1, B2 und B4 ebenso irrelevant.
3. Suspekt ist Datenpaket A5. Einige Tupel erfüllen `A>6`, aber es ist nicht bekannt welche. Folglich ist B5 auch suspekt. Deshalb müssen sowohl A5 als auch B5 dekomprimiert werden.

An diesem Beispiel sieht man, dass es sich bei Datenpaketknoten neben dem theoretischen Ansatz eines Wertepaars der Rough-Set-Methode praktisch um ein Art Materialized View (Kapitel 2.11.4.2) handelt (die in zeilenorientierten DBMS wie Oracle, IBM DB2 und Microsoft SQL-Server implementiert ist), jedoch mit dem Unterschied, dass die Datenpaketknoten automatisch für alle Datenpakete (und damit Attribute) angelegt werden und im Gegensatz zu Materialized Views für die Optimierung von Ad-hoc-Anfragen konzipiert sind, wie das obige Beispiel zeigt.

Die Tatsache, dass ein Datenpaket suspekt ist, heißt jedoch nicht, dass es in jedem Fall dekomprimiert werden muss, wie ein weiteres Beispiel zeigt. Bei der Anfrage `SELECT MAX (B) FROM T WHERE A > 6;` bleiben die dieselben Datenpakete relevant, irrelevant oder suspekt. Paket B5 mit `MAX(B)=0` muss nun aber nicht mehr dekomprimiert werden, weil das bereits vorher betrachtete Paket B3 den Wert `MAX(B)=1` hat.

Ein weiteres Beispiel soll die Nutzung der *Pack-to-Pack*-Wissensknoten durch den Optimierer verdeutlichen. Gegeben ist die Anfrage `SELECT MAX (X.D) FROM T JOIN X ON T.B = X.C WHERE T.A > 6;`. Relation X mit Attributen C und D umfassst 150.000 Tupel und damit 3 Datenpakete pro Attribut. Über der bereits genannten Relation B und X wird über den Attributen B und C ein Verbund durchgeführt für Werte `A>6`. Dabei wird für den Verbund der Max-Wert in Attribut D ausgegeben. Tabelle 16 zeigt die Datenpaketknoten für die Spalten C und D.

Bei der Ausführung des Anfrageplans wird wie folgt vorgegangen: Lt. Tabelle 15 sind die einzigen in Frage kommenden Datenpakete für die Bedingung `T.A>6` A3 + B3 und A5 + B5.

Paket-Nummer Attribut C + D	**DPN von C**			**DPN von D**		
	Min	Max	Summe	Min	Max	Summe
Paket C1+D1	-1	5	100	0	5	100
Paket C2+D2	-2	2	100	0	33	100
Paket C3+D3	-7	8	1	0	8	100

Tabelle 16: Datenpaketknoten für Relation X ([IBWP08])

In einem Pack-to-Pack-Wissensknoten, der von Infobright automatisch angelegt wird, sobald eine entsprechende Anfrage einmal gestellt wurde, ist nun zusätzlich beschrieben, welche Datenpakete der Attribute X.C und T.B übereinstimmende Werte haben. Das Beispiel in Tabelle 17 (0=es gibt Übereinstimmungen, 1=es gibt keine Übereinstimmungen) wird als gegeben angenommen. Da bei T.B3 alle Werte 0 sind, ist klar, dass T.B3 nicht mehr berücksichtigt werden muss, da es keine Übereinstimmungen mit X.C gibt. Infobright muss also nur einen Join der Datenpakete X.C2 und T.B5 durchführen und dann für Datenpaket X.D2 den Wert `MAX(D)` des Joins ermitteln. Zur Prüfung von `A>6` muss außerdem A5 dekomprimiert werden.

	T.B1	**T.B2**	**T.B3**	**T.B4**	**T.B5**
X.C1	0	1	0	1	0
X.C2	1	0	0	0	1
X.C3	1	1	0	1	0

Tabelle 17: Pack-to-Pack-Wissensknoten für Attribute B und C [IBWP08]

3.1.2.2 Laden der Daten und Kompression

Der Prozess des Ladens der Daten von den Datenquellen in das Data Warehouse sowie der Erstellung und Speicherung der Datenpakete und der Datenpaketknoten ist in Abbildung 29 dargestellt. Infobright verwendet einen Multi-Thread-Ladealgorithmus zum parallelen Laden der Attribute. Die Nullmaske speichert die Positionen von Nullwerten, die nicht mitkomprimiert werden. Es wird der für den Datentyp des Attributs Spalte und die auftretenden Regelmäßigkeiten der Daten geeignetste Kompressionsalgorithmus gewählt, wobei aus der recherchierten Literatur [Bright08] nicht ersichtlich war, welche Kompressionsverfahren Infobright einsetzt.

3.1.2.3 Schnittstellen

Infobright nutzt die Treiber von MySQL mit (C, JDBC, ODBC, .NET, Perl usw.). Die Infobright Community Edition (ICE) 3.1 bietet die in Tabelle 18 aufgeführten Features ([ICE31Specs]).

Feature	Umsetzung
SQL-Support	ANSI SQL-92 mit einigen SQL-99-Erweiterungen (Views, Stored Procedures)
Flexibler Schema-Support	ICE unterstützt alle Schemadesigns
Interfaces	JDBC (nativer Java-Treiber), ODBC, native Schnittstellen
APIs	C,C++,C#,Delphi, Eiffel, SmallTalk, Java, Lisp, Perl, PHP, Python, Ruby, REALbasic, FreeBasic, Tcl
Userzahl	Bis zu 500 User mit 24 konkurrierenden Anfragen
Betriebssysteme	Windows XP (32-bit), Red Hat Enterprise Linux 5 (64-Bit), Debian „Lenny“ (64-Bit), CentOS 5 (64-Bit), Fedora 9 (32-bit), Ubuntu 8.04 (32-bit)
Prozessoren	Intel und AMD 32- und 64-bit
Data Loader	Paralleler DataLoad über mehrere Tabellen
Kompression	Durchgängiger Einsatz von Kompression der Datenpakete, Wahl des optimalen Kompressionsverfahrens je nach Datentyp
Skalierbarkeit	Skalierbarkeit bis 30TB Rohdaten pro Server
Optimierung	Spaltenorientierte Architektur, keine Indizes, aber Einsatz von Rough Sets (Wissensknoten, Knowledge Grid)
Administration	Administration über MySQL-Tools
Wartung	Keine Indizes, kein manuelles Tuning, keine manuelle Partitionierung
Ladeformate	ICE Loader kann Textdateien laden
BI-Unterstützung	Unterstützung vielfältiger BI-Tools wie Cognos, Business Objects, SAS, Pentaho, JasperSoft, MicroStrategy etc.

Tabelle 18: Features von Infobright Community Edition [ICE31Specs]

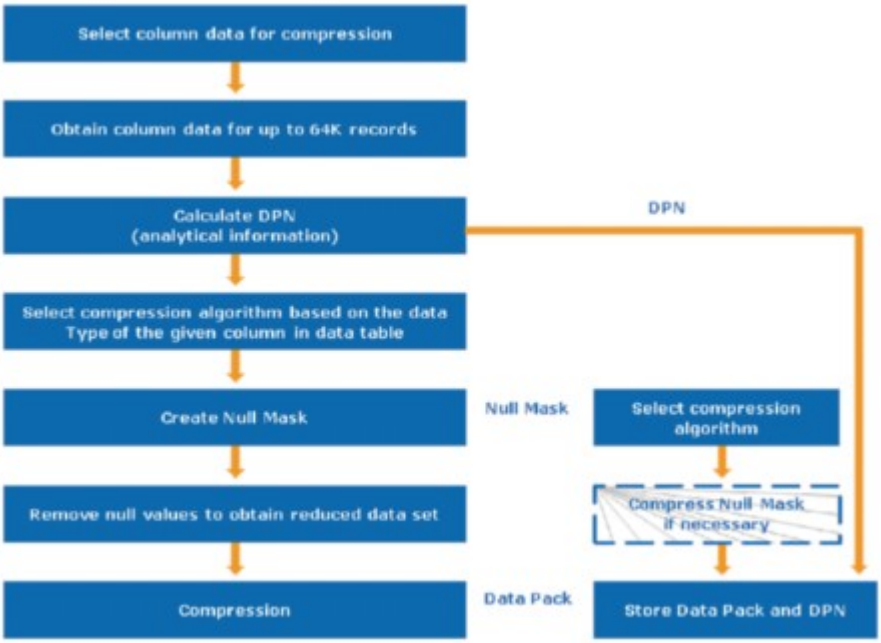

Abbildung 29: Anlegen von DN und DPN beim Laden von Daten in das DW [IBWP08]

3.1.2.4 Systemarchitektur

Infobright ist eine Storage Engine für MySQL, siehe [Bright08] und [MySQLSE09]. Die Konnektoren von MySQL (JDBC, ODBC etc.) können mitgenutzt werden. Die Katalogdaten (Tabellendefinitionen, Viewdefinitionen etc.) werden hingegen in der MyISAM-Storageengine gespeichert. Brighthouse hat einen eigenen Query Processor und einen eigenen Loader zum Laden von Daten in das Data Warehouse (wegen Nutzung des Knowledge Grid und der Kompression). Das Knowledge Grid schaltet den MySQL-Indexsupport aus. Abbildung 30 zeigt die Systemarchitektur von Infobright.

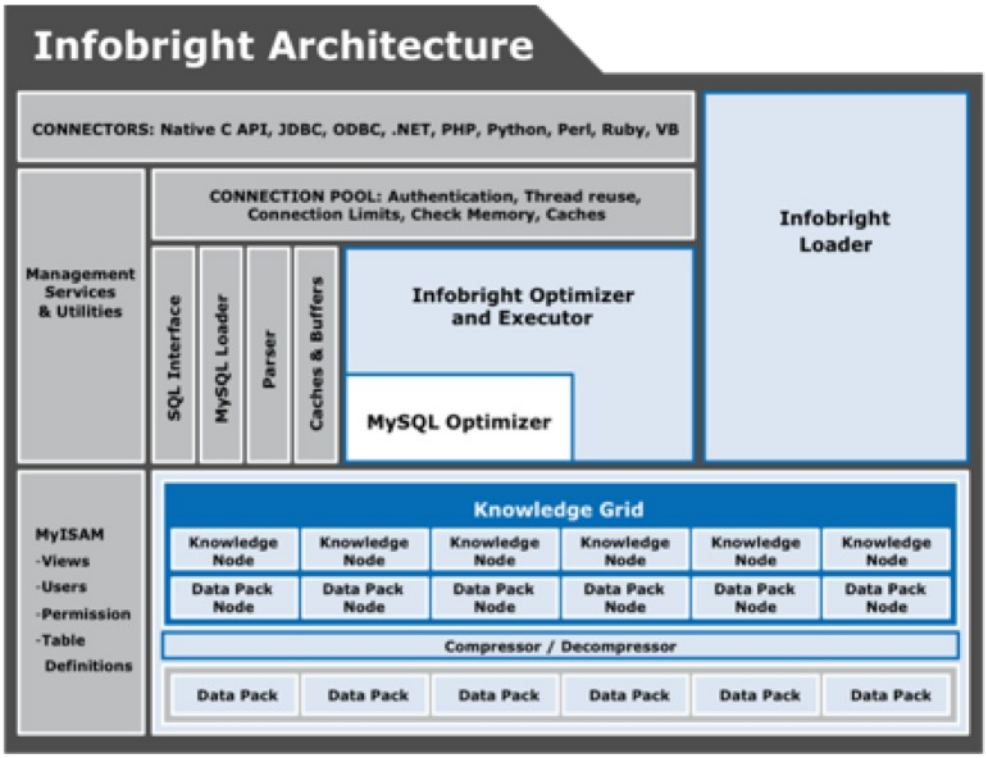

Abbildung 30: Einbindung von Infobright in MySQL-Gesamtarchitektur [IBWP08]

Gemäß [IBForum] verwendet Infobright eine SMP-Architektur, also einen Shared-Memory-Ansatz, siehe Kapitel 2.13.

3.1.2.5 Ansätze von Infobright

Tabelle 19 fasst die bei dieser Implementierung vorhandenen Ansätze nach den genannten Betrachtungsschwerpunkten zusammen.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Vollständige Dekomposition der Attribute aller Relationen. Aufteilung der Werte eines Attributes in Datenpakete à 65.536 Zeilen (unsortiert). Automatisierte Aggregierungen (Rough Sets) der Datenpakete in Form von Datenpaketknoten. Indizierung auf Datenpaketebene durch Wissensknoten (Verbunde zwischen Datenpaketen, vorkommende Werte) durch interne Nutzung von Bitmap-Indizes (automatisch erstellt). Ausschalten der Indizes für Anwender (läuft alles automatisiert, Zugriffsstrukturen sind für Ad-hoc-Anfragen ausgelegt).
2	*Algorithmen* (DB-Operationen, Kompression)	Einsatz von Kompression für die Datenpakete mit für den jeweiligen Datentyp des Attributs optimalem

Nr.	Betrachtungsschwerpunkt	Ansatz
		Kompressionsverfahren
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Data Warehouse für OLAP-Anwendungen, relationales Datenmodell. Verwendung von SQL als Anfragesprache, Standard-Schnittstellen wie JDBC und ODBC, verschiedene API (C,C++,PHP etc.)
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Optimierer basierend auf der Theorie der Rough Sets setzt Metadaten (Wissensknoten und Datenpaketknoten) ein, um Menge der zu verarbeitenden Datenpakete (und damit Datensätze) zu reduzieren. D.h. der Query Processor liest nicht die gesamte Spalte ein. Er arbeitet aber nicht auf komprimierten Daten (Kapitel 2.12.4), sondern reduziert die Menge zu dekomprimierender Daten. Zum Zeitpunkt der Materialisierung werden in der Literatur keine Aussagen getroffen.
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Memory-Architektur, Datenpakete sind horizontal und vertikal partitionierte Relationen
6	*Transaktionsverwaltung* (Update-Problematik)	Aus der Dokumentation nicht ersichtlich, ob z.B. Snapshot Isolation eingesetzt wird
7	*Betriebssystem*	Windows XP (32-bit), Red Hat Enterprise Linux 5 (64-Bit), Debian „Lenny“ (64-Bit), CentOS 5 (64-Bit), Fedora 9 (32-bit), Ubuntu 8.04 (32-bit)

Tabelle 19: Ansätze / Klassifizierungskriterien für Infobright

3.1.3 LucidDB

LucidDB ist ein als Data Warehouse konzipiertes relationales Open-Source-**DBMS**. Die aktuelle Version ist 0.9.1. Seine wesentlichen Features sind

- eine spaltenbasierte Speicherarchitektur,
- der Einsatz von Kompressionstechniken,
- der Einsatz verschiedener Indizes je nach Datenverteilung, entweder B-Bäume oder (komprimierte) Bitmap-Indizes, ggf. beide für verschiedene Bereiche einer Relation,
- *Page-level-Multiversioning* für konkurrierende Lese- und Schreibzugriffe, das den lesenden Zugriff auf einen konsistenten Stand der Relation während eines Bulkloads oder eines Updates ermöglicht,
- eine Star-Join-Optimierung auf der Basis von Semi-Join-Techniken, die nur die benötigten Zeilen der Faktentabelle heranziehen, sowie eine kostenbasierte Anordnung von Verbundoperationen und Indexauswahl,

- der Einsatz von Hashverfahren für Verbundoperationen und Aggregationen sowie die Verknüpfung von Bitmap-Indizes,
- der Einsatz von Prefetching-Strategien,
- sowie Unterstützung der Standards **SQL**:2003 und **JDBC**.

Im Moment ist es für den Betrieb auf einem einzigen Windows- oder Linux-Server konzipiert, siehe [Lucid] und [LuFeat]. LucidDB ist nicht als **OLTP**-, sondern als eine auf Bulkloads ausgelegte **OLAP**-Datenbank konzipiert, siehe [LucArch]). Abbildung 31 zeigt die hybride Java- und C++-Architektur von LucidDB.

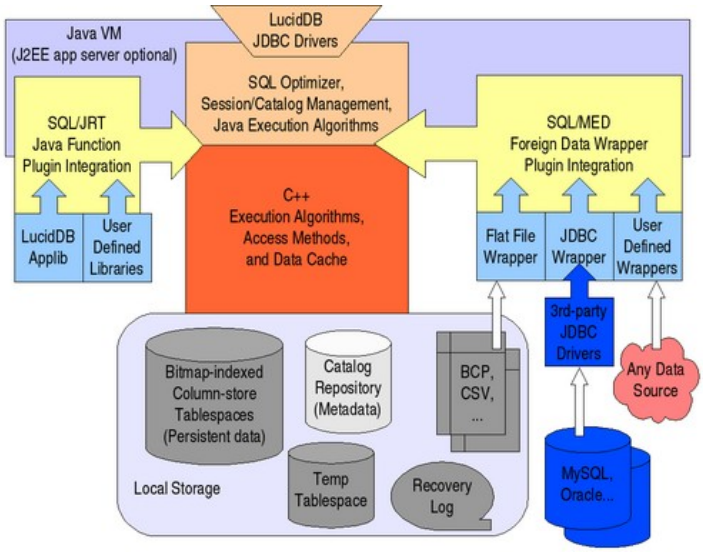

Abbildung 31: Systemarchitektur von LucidDB aus [LucidArch09]

Nachfolgend sollen insbesondere die Ansätze der Datenablage, Indizierung und Anfrageoptimierung betrachtet werden.

3.1.3.1 Physische Speicherung

LucidDB speichert die Daten in 32KB großen Seiten in einer Betriebssystemdatei *db.dat* ([LucStor]). Weitere .dat-Dateien werden angelegt für temporäre Daten der Anfragebearbeitung, für das Transaktions-Logging und Recovery.
Auf die Seiten wird über eine eindeutige Block-ID zugegriffen, die einem Offset in der physischen Datei zugeordnet sind. Die Defaultgröße des Datenbankpuffers umfasst 5.000 Seiten (ca. 160 MB). Zum effizienten Zugriff auf Seiten im Puffer werden *Hash-Buckets*

(Kapitel 2.11.3.2) verwendet. Seiten mit geänderten Inhalten (engl. *Dirty Pages*) werden bei jedem Transaktions-Commit und regelmäßig durch einen sogenannten *Lazy Page Writer*, der aktiv wird, wenn ein gewisser Prozentsatz des Datenbankpuffers geändert wurde, in den Sekundärspeicher geschrieben. Seiten werden in LucidDB in sogenannten Segmenten abgespeichert (Synonym für die Pufferrahmen der Pufferverwaltung, siehe [Seite 110 [SHS05]]). Die Segmente bilden die logische Page-ID auf die physischen Block-ID ab, siehe [LucStor].

3.1.3.2 Spaltenorientierte Speicherung

LucidDB partitioniert Tabellen vertikal und speichert sie in komprimierter Form mit den bereits erwähnten Vorteilen bei der Anfrageausführung (Ausblenden nicht benötigter Attribute, bessere Pufferausnutzung), siehe [LucArch]. Eine Tabelle besteht aus einer Menge von *Clustern*. Ein *Cluster* legt die Granularität fest, mit der die Spalten vertikal partitioniert werden. Standardmäßig wird jede Spalte in ein Cluster abgebildet, es können aber auch mehrere Spalten in ein Cluster abgebildet werden. Eine Cluster-Seite speichert folglich die Werte einer bestimmten Menge logischer Tupel-IDs für *alle* Spalten des Clusters.

3.1.3.3 Indizierung

LucidDB setzt B-Baum- und Bitmap-Indizierung (Kapitel 2.11.3) ein. Bitmap-Indizes werden direkt auf den komprimierten Daten gebildet und selbst komprimiert gespeichert, siehe [LucArch]. Über die Bitmap-Indizes werden die relevanten Bereiche der Relation bei Ausführung einer Anfrage (mit Prädikaten) ermittelt. Die Kompression wird durch das Entfernen von Bytes erreicht, die keine gesetzten Bits enthalten. Die entfernten Bytes werden durch einen Deskriptor im Bitmap-Index beschrieben.

Um auf Spaltenwerte über den Tupel-ID-Wert schnell zugreifen zu können, sind die Tupel-ID-Werte mit den logischen Page-IDs der Segmente über einen B-Baum-Index pro Cluster verbunden, siehe Abbildung 32. Der Begriff Rid ist dabei identisch mit der Tupel-ID. Gespeichert werden die jeweils ersten Tupel-ID-Werte jeder Seite eines Clusters, die über ihre Page-ID identifiziert wird. Wenn man z.B. auf Tupel-ID 1.000 zugreift, und der B-Baum für Page-ID 50 als Tupel-ID den Wert 1.248 speichert, für Page-ID 49 aber 984, dann ist klar dass Page-ID 49 benötigt wird.

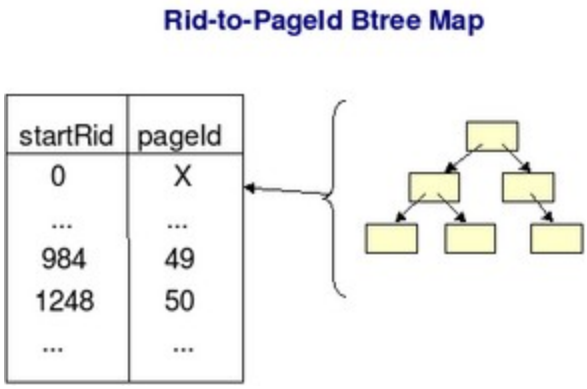

Abbildung 32: B-Baum-Indizierung in LucidDB [LucStor]

Abbildung 33 zeigt einen Vergleich zwischen einem B-Baum-Eintrag sowie einem unkomprimierten und komprimierten Bitmap-Index. Ein Wert kommt in den Tupeln 10,12 und 8001 vor (wobei ein B-Baum normalerweise die physische Tupel-ID speichert). Beim unkomprimierten Bitmap-Index besteht das erste Byte aus 8 Leerbits, bevor im zweiten Byte an den Stellen 3 und 5 gesetzten Bits kommen, die die Zeilen 10 und 12 darstellen. Bei einem unkomprimierten Bitmap-Index kommen dann 998 Leer-Bytes, damit im letzen Byte an Stelle 2 ein Bit gesetzt ist, das Zeile 8001 repräsentiert.

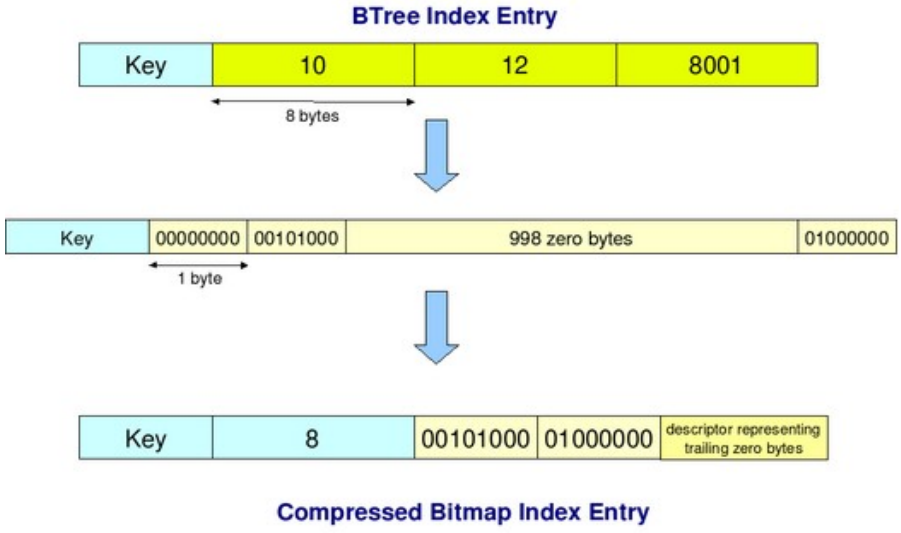

Abbildung 33: B-Baum-Index und komprimierter Bitmap-Index [LucStor]

3.1.3.4 Datenkompression

Die Spaltenwerte des Clusters werden komprimiert gespeichert. Der Beschreibung nach wird (wie bei Oracle, Kapitel 2.12.4.1) eine wörterbuchbasierte Seiten- bzw. Blockkompression für Duplikate verwendet, siehe [LucStor]. Vor jeder Komprimierung wird geprüft, ob die Kompression wirklich lohnend ist oder ob wenig Duplikate in der Seite enthalten sind. Weitere Kompression wird durch das Streichen führender Nullen bei den Typen BIGINT, DATE, TIME und DECIMAL erreicht. Dies wird aber nur bei 8-Byte-Integern gemacht.

3.1.3.5 Page-level-Multiversioning

Auch wenn LucidDB als Read-Only-Data Warehouse konzipiert ist, sind Schreiboperationen zum Laden und Ändern von Daten notwendig, vgl. [LucStor]. Um während dieser Zeit das Lesen von Daten weiterhin zu ermöglichen, wird von den Seiten ein lesbarer Schnappschuss bei Beginn der Schreibtransaktionen angelegt. Hierzu wird eine neue Version der Seite angelegt und mit der existierenden (sogenannten) *Ankerseite* verknüpft. Jede folgende Schreibtransaktion legt eine neue Seitenversion am Anfang der Seitenkette an, d.h. die jeweils neueste Seite ist direkt mit der Ankerseite verknüpft. Langandauernde Lesetransaktionen können so ältere Schnappschüsse lesen, während neuere Lesetransaktionen neuere Schnappschüsse heranziehen.Die Ketten wachsen nicht ohne Begrenzung an, denn nicht mehr von einer Transaktion verwendete Schnappschüsse werden freigegeben. Der Zeitstempel wird auf jeder Seite der Kette vermerkt und entspricht dem COMMIT-Zeitpunkt der Schreibtransaktion. Anhand des Zeitstempels können Lesetransaktionen erkennen, welche Seite den für sie aktuellsten konsistente Stand enthält. Mit dem Befehl `ALTER SYSTEM DEALLOCATE OLD` werden nicht mehr im Zugriff befindliche alte Seiten (außer der Ankerseite) deallokiert. Abbildung illustriert das Page-level-Multiversioning in LucidDB.

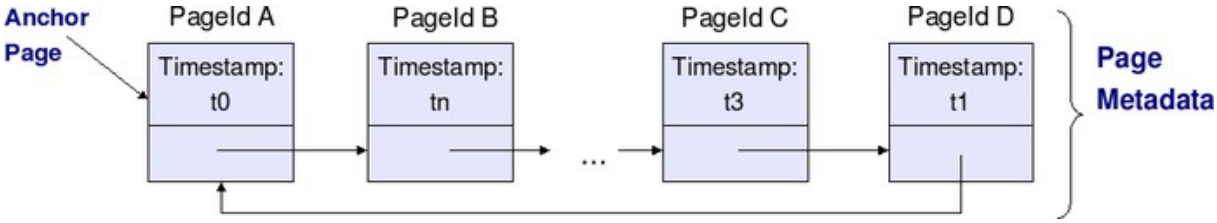

Abbildung 34: Page-level-Multiversioning in LucidDB [LucStor]

3.1.3.6 Optimierung der Dateiorganisation

Löschen von Tupeln

LucidDB löscht Zeilen nicht physisch aus den Seiten und Indizes, sondern die gelöschten Zeilennummern werden in einem *Deletion-Index* (pro Relation) vermerkt, siehe [LucStor]. Der Deletion-Index ist als Bitmap-Index realisiert. Der Startwert ist hier nicht ein bestimmter Wert der Spalte, sondern der Startzeilenwert der ersten gelöschten Zeile. Der Deletion-Index erweitert sich dynamisch, wenn weiter hinten liegende Zeilen gelöscht werden. Durch den Deletion-Index wird das Dekomprimieren einer Seite bei Löschen einer Zeile vermieden. Der Deletion-Index wird beim Full-Table-Scan verwendet, um so wenig Daten wie nötig einlesen zu müssen.

Beim Löschen der Tabelle mit TRUNCATE TABLE hingegen wird der Deletion-Index nicht verwendet, sondern auf Seitenebene gearbeitet, ebenso werden die Wurzelseiten jedes B-Baums der Cluster sowie die Bitmap-Indizes der Tabelle angepasst. Die Wurzelseiten enthalten danach keine Einträge mehr. Das Page-level-Multiversioning ermöglicht für laufende Lesetransaktionen einen Zugriff auf die alte Seite.

Reorganisation

Durch Deletes und Updates (welches ein Delete gefolgt von einem Insert ist) wächst der Deletion-Index. Durch das Aufbewahren der gelöschten Zeilen in den Tabellen und das Anwachsen der Versionsliste der Seiten wird nicht mehr benötigter Speicherplatz belegt. Dies wird durch den Befehl `ALTER TABLE REBUILD` freigegeben. Eine neue leere Tabelle wird angelegt und die Zeilen der bisherigen dort (ohne die gelöschten) hineinkopiert. Durch die Seitenversionierung können auch während des Kopierens ausgeführte Lesetransaktionen auf die Daten zugreifen. Die Wurzelseiten der B-Bäume und Bitmap-Indizes werden ebenfalls versioniert.

3.1.3.7 Anfrageoptimierung und -ausführung

Zeitpunkt der Materialisierung

Der LucidDB-Optimierer liest die in einer Anfrage nicht genannten Spalten bzw. deren Cluster nicht ein. Die Spalten werden individuell gelesen und dann lt. [LucStor] zu Tupeln zusammengefügt, was einer frühen Materialisierung im Sinne von C-Store (Kapitel 3.1.1.7) entspricht.

Nutzung des Deletion-Indexes

Row Scans bezeichnen das Einlesen einer Spalte ohne Indizes. Beim Lesen einer Tabelle ohne Verwendung eines Bitmap-Index (d.h. Full Table Scan) werden sämtliche Zeilen einer Spalte sequenziell gelesen, es werden jedoch die Zeilen gefiltert, die im Deletion Index vermerkt sind. Dabei wird versuchAlternativ können die Bitmap-Indizes als Input beim Scannen der Spalte herangezogen werden, dies wird im folgenden Unterkapitel erläutert.

Filterung

Beim Scannen einer Spalte (ohne Index) wird von LucidDB ein einfacher Filter eingesetzt. Dieser testet auf Gleichheit eines Wertes mit einer Konstante (z.B. `col=10)`, auf einem Bereich (z.B. `col BETWEEN 20 and 30)` oder auf eine Werteliste (z.B. `col IN (40,50,60)`). Der Zeilenscan blendet bei der Ausgabe die Zeilen aus, die diese Prädikate nicht erfüllen. Erfüllt eine Spalte das Prädikat komplett nicht, werden die weiteren Spalten für diese Zeile nicht mehr herangezogen. Beim Scannen komprimierter Daten wird der Filter auf jeden eindeutigen Wert (d.h. jeden Wörterbucheintrag) der Spalte angewendet und das Ergebnis in einer Bitmap abgespeichert.

LucidDB-Joinoptimierung

Der Optimierer unterstützt Star-Joins mit Bitmap-Indizes durch Anwendung von Semi-Joins, d.h. dass nur auf die Zeilen der Faktentabelle zugegriffen wird, die wirklich zum Berechnen des Anfrageergebnisses benötigt werden. Zur Bestimmung der optimalen Reihenfolge der Verbundberechnung und der Bestimmung der benötigten Bitmap-Indizes wird eine kostenbasierte Analyse durchgeführt. Der Query Processor setzt unter anderem Hash-Verbundoperationen und Hash-Aggregationen ein.

In diesem Unterkapitel wird anhand der vorhandenen Literatur ([LucJoinOpt]) die Optimierung von Joinanfragen durch LucidDB gezeigt. Das Ziel von Anfrageoptimierung ist, Selektionen zur Minimierung der Datenmengen so früh wie möglich auszuführen. Es wird insbesondere beschrieben,

Star-Joins

LucidDB optimiert Star-Joins durch Semijoins. Die Idee ist: Bei einem Join zwischen einer Faktentabelle und einer Dimensionstabelle sowie einer Selektion der Dimensionstabelle muss nicht die gesamte Faktentabelle gelesen werden. Stattdessen muss nur die Untermenge der Faktentabelle gelesen werden, die mit der selektierten Untermenge der Dimensionstabelle verbunden wird. Die reduzierte Untermenge der Faktentabelle, die Ergebnis des Semijoins ist,

kann mit der selektierten Dimensionstabelle verbunden werden, um die notwendigen Tupel für das Ergebnis zu erhalten.

Bei der Anfrage SELECT * FROM sales s, product p WHERE s.product_id = p.id and p.size = 'S'; mit der Faktentabelle sales und der Dimensionstabelle product würde bei der Anfrage der Semijoin zum Einlesen nur der Zeilen aus sales führen, die mit den selektierten Zeilen von product (welche deutlich geringer in der Anzahl sind) einen Verbund bilden.

LucidDB führt Semijoins durch eine Reihe von Bitmap-Index-Lookups durch. Daher muss es einen Bitmap-Index auf der Spalte der Faktentabelle geben, die mit der Dimensionstabelle verbunden wird. Die eindeutige Menge selektierten Werte der Dimensionsspalten, die mit der Faktentabelle verbunden Werte, dienen als Suchschlüssel beim Indexscan. Dabei wird wie folgt vorgegangen, vgl. [LucJoinOpt]:

1. Auf den Attributen der Dimensionen werden die Selektionen durchgeführt.
2. Die Verbund-Attribute der Dimensionen werden aus dem Ergebnis projeziert.
3. Nullwerte werden herausrausgefiltert, damit anschließenede Index-Lookup nicht Nullwerten in der Faktentabelle eine Übereinstimmung findet. Bei NOT-NULL-Spalten ist kein Filtern notwendig
4. Doppelte Werte werden zum Vermeiden redundanter Lookups entfernt.
5. Zum Schluss werden die Schlüssel sortiert und die Bitmap-Index-Lookups so lokalisiert.

3.1.3.8 Ansätze von LucidDB

Tabelle 20 führt die eben ermittelten Ansätze von LucidDB zusammenfassend auf.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Clusterung von 1-n Spalten, B-Baum-Indizes für Zuordnung von Tupel-ID zu Page-ID, Bitmap-Indizes für Anfrageoptimierung (insbesondere Star-Join)
2	*Algorithmen* (DB-Operationen, Kompression)	Einheitliches Kompressionsverfahren (Wörterbuchkompression für Blöcke) Anwenden von Selektionsfiltern auf komprimierte Daten. Erzeugen von Bitmap-Indizes für komprimierte Daten.
3	*Anwendungen* (Anwendungsszenarien,	Relationales Datenmodell. JDBC, SQL-2003

Nr.	Betrachtungsschwerpunkt	Ansatz
	Schnittstellen, Datenmodelle)	
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Filteroperationen werden auf komprimierten Daten ausgeführt. Frühe Materialisierung (Tupelbildung).
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Single-Server-Datenbank. Vertikale Partitionierung in Clustern.
6	*Transaktionsverwaltung* (Update-Problematik)	Page-level-Multiversioning
7	*Betriebssystem*	Linux, Windows

Tabelle 20: Ansätze / Klassifizierungskriterien für LucidDB

3.1.4 1010data's Tenbase

Die Firma 1010data bietet seinen Kunden eine Data-Warehouse-Lösung bestehend aus einem **coDBMS** und einer webbasierten Analyseplattform, 1010vision, als Service an. Die Lösung wird für verschiedene Anwendungszwecke eingesetzt, siehe [Wiki1010].

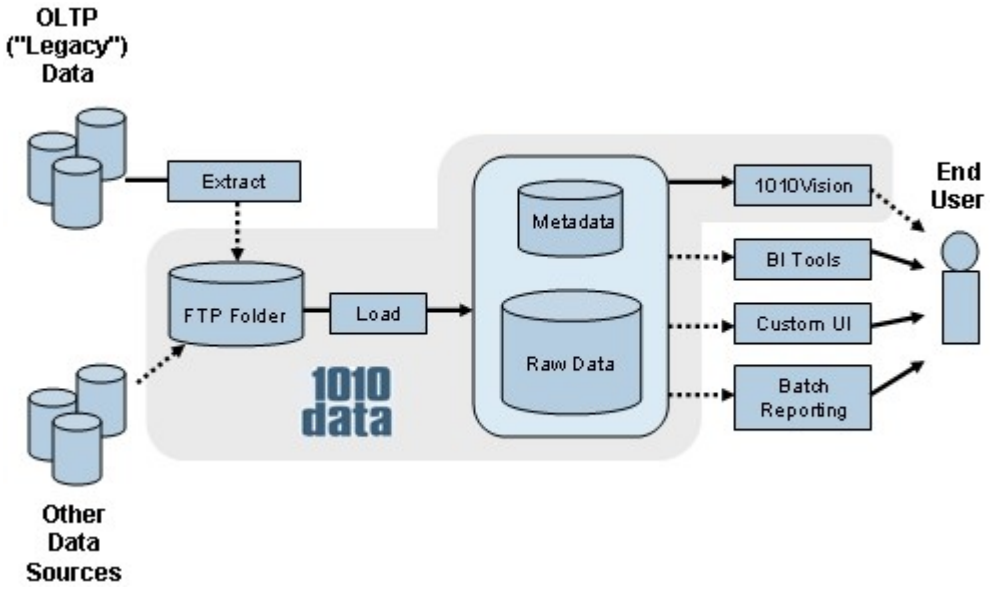

Abbildung 35: Tenbase Backend Service [1010Arch]

Der Kunde sendet die Rohdaten üblicherweise per File Transfer Protocol (FTP) an 1010data und kann sie dann mit einer Webanwendung analysieren, siehe [1010Arch] und Abbildung 35.

Über das Webfrontend stellt der Anwender Anfragen an das Tenbase-Backend. Die Features von Tenbase sind u.a. eine spaltenorientierte Speicherarchitektur, eine Shared-Nothing-

Architektur (paralleles und verteiltes DBMS), Verzicht auf die Erstellung von Indizes und auf Präaggregation beim ETL-Prozess, Verfügbarkeit während des Ladeprozesses (konsistenter Stand für Lesetransaktionen), sowie eine SQL-, eine ODBC-Schnittstelle und eine eigene XML-Anfragesprache mit 150 fertigen Funktionen, siehe [TBFeat]. Eine API auf Basis des http- und eines XML-Protokolls liegt für verschiedene Programmiersprachen (Java, C++ etc.) vor.

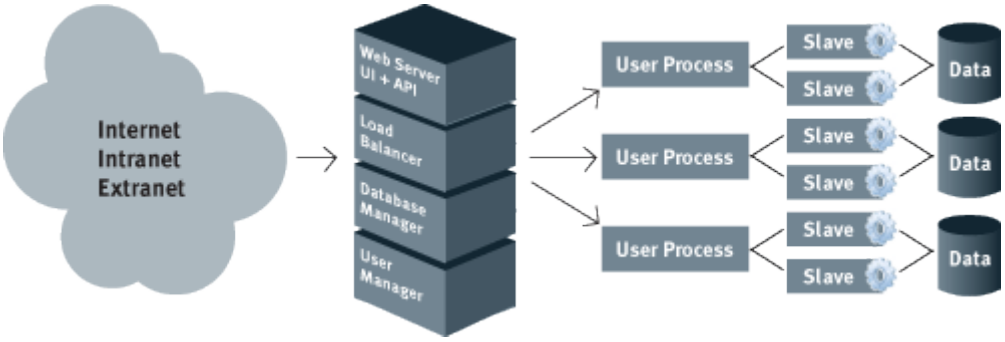

Abbildung 36: Tenbase-Architektur [TBFeat]

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Spaltenorientierung
2	*Algorithmen* (DB-Operationen, Kompression)	
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	SQL, JDBC
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Nothing
6	*Transaktionsverwaltung* (Update-Problematik)	
7	*Betriebssystem*	Software as a Service (Saas)

3.1.5 Bigtable, Hbase, Hypertable und Cassandra Project

Die **coDBMS** Bigtable, Hbase, Hypertable und Cassandra Project werden zusammen betrachtet, da die drei letztgenannten Bigtable nachempfunden sind und alle im Wesentlichen die gleichen Ansätze haben.

Googles *Bigtable* ist ein komprimiertes, proprietäres, spaltenorientiertes und skalierbares verteiltes **DBMS,** das auf dem *Google File System* [GFS2003], dem *Chubby Lock Service* und einigen anderen Google-Programmen basiert, siehe [BTWiki] und [Hit05]. Es ist als Schlüssel-Wert-Speicher (Map) implementiert. Die Entwicklung begann in 2004. Es ist für die (durch automatische Replizierung) ausfallsichere Verwaltung von Petabytes unstrukturierter Daten auf tausenden von Commodity-Servern konzipiert, vgl. [Chang06]. Der Zugriff auf das **DBMS** erfolgt über eine proprietäre C++-**API**.

Hbase ([HBHome]) ist ein Bigtable nachempfundenes, spaltenorientiertes, verteiltes Open-Source-**DBMS** der Apache Software Foundation und wurde mit der Programmiersprache *Java* entwickelt ([HBWiki]). Den ersten Prototypen gab es im Februar 2007, die aktuelle Version ist 0.19.3. Ziel ist ebenfalls die Speicherung großer Datenmengen (Petabytes) mit Milliarden von Tabellenzeilen und Millionen von Spalten in tausenden von Versionen. Hbase setzt auf dem *Hadoop Distributed File System* (**HDFS**) von Apache auf und setzt genau wie Bigtable Kompression und Bloom-Filter[28] auf Spaltenbasis ein. Als Standalone-Version kann es auch auf dem lokalen Dateisystem eingesetzt werden ([HBAPI]). Auf die Tabellen kann mit einer Java-**API** und einem Kommandozeilenprogramm zugegriffen werden.

Hypertable ([HTHome]) ist ebenfalls ein Open-Source-**DBMS**, dass Bigtable nachempfunden ist ([HTWiki]). Es ist in der Programmiersprache C++ implementiert und setzt wahlweise auf einem der verteilten Dateisysteme **HDFS**, *GlusterFS* [GlusterFS] oder Cloudstore (früher *Kosmos File System (**KFS**))* auf, [ClStore]. Hypertable liegt aktuell in der Version 0.9.2.6 alpha vor. Es ist für verschiedene Linux-Derivate (Ubuntu, Gentoo, Fedora) und Mac OS/X vorkompiliert über die Homepage [HT09] erhältlich.

Cassandra ist lt. [CasProj] genau wie die anderen genannten **DBMS** ein skalierbarer, verteilter Schlüssel-Wert-Speicher. Es wird produktiv von *Facebook* eingesetzt und setzt auf Amazons verteiltem Dateisystem *Dynamo* (mittlerweile Open Source) auf. Es wurde von Facebook 2008 als Open-Source-**DBMS** veröffentlicht.. Die aktuelle Version ist 0.3.0.

Da Hbase, Hypertable und Cassandra Nachbildungen von BigTable sind, siehe [HTWiki], und bei der Literaturrecherche deutlich wurde, dass sich die Ansätze weitestgehend

[28]Bloom-Filter werden später noch erläutert.

überschneiden, werden die Implementierungen nicht getrennt, sondern zusammen vorgestellt. Nur im Bedarfsfall werden Unterschiede deutlich macht. Gleiches gilt auch für die abschließend durchgeführte Klassifizierung der Implementierungen. Die gemeinsame Betrachtung ist auch deshalb sinnvoll, weil bei der Literaturrecherche deutlich wurde, dass Informationen zur einen Implementierung teilweise in der Literatur der anderen Implementierung gegeben werden. So ist in der Literatur für Bigtable z.B. nicht unbedingt verständlich beschrieben, inwieweit Bigtable einen spaltenorientierten Ansatz verfolgt. Dies z.B. wurde erst nach Betrachtung von Hbase und Hypertable deutlich.

3.1.5.1 Allgemeines

Die in Kapitel 3.1.5 vorgestellten **DBMS** sind keine relationalen **DBMS**, sondern *Maps*. Eine Map ist ein abstrakter Datentyp, der aus eine Menge von Schlüsseln und einer Menge von Werten besteht. In der Programmiersprache *PHP* z.B. wird hierfür der Begriff „assoziatives Array“ verwendet.

Es existieren gibt keine unterschiedlichen Datentypen, und Daten können nicht über Fremdschlüssel zueinander in Beziehung gesetzt werden. SQL wird nicht unterstützt, stattdessen steht eine einfache, jeweils proprietäre Anfragesprache und **DML**/**DDL** zur Verfügung.

Ebenso wenig sind Verbundoperationen (Joins) möglich, auch Sekundärindizes und tupelübergreifende Transaktionen werden nicht unterstützt [Stack09], genau wie referenzielle Integrität und statitische Analyse durch GROUP BY und ORDER BY nicht unterstützt werden.

Ein optimierender Query Processor (im Sinne von algebraischer oder kostenbasierter Optimierung) ist gemäß [Stack09] ebenfalls nicht implementiert und auch nicht nötig, denn es werden, wie noch gezeigt wird, lediglich einfache `get`- und `scan`-Operationen auf den Tabellen bezogen auf den Zeilenschlüssel angeboten. Die Formulierung eines Prädikats ist ebenfalls nicht vorgesehen. Ein Zugriff auf einen oder mehrere Datenwerte erfolgt immer nur über den Zeilenschlüssel, den Spaltenbezeichner und den Zeitstempel.

Daten werden in Tabellen mit spaltenorientierten Zeilen abgelegt. Es ist sowohl ein Batch- als auch ein wahlfreier Zugriff auf die Daten möglich. Für die Lösungen, die Google zur Skalierung von Bigtable entwickelt hat, gibt es seitens der anderen Implementierungen jeweils ein Pendant (siehe Tabelle 21).

Google	Apache	Hypertable	Cassandra
Google File System	Hadoop DFS	Hadoop DFS oder andere	Amazon Dynamo
Google Map Reduce ([GoogleMR]	Hadoop Map Reduce	Google Map Reduce	Zur Zeit noch nicht implementiert ([MArch09]
BigTable	HBase	Hypertable	Cassandra

Tabelle 21: Komponenten von BigTable, Hbase, Hypertable und Cassandra

3.1.5.2 Datenmodell

Alle Implementierungen verwenden ein multidimensionales Datenmodell mit den vier Dimensionen *Zeilenschlüssel (*engl. *Row Key)*, *Spaltenfamilie (*engl. *Column Family), Spalte (*engl. *Column)* und *Zeitstempel (*engl. *Timestamp)*. Jede Datenzeile hat einen sortierten Zeilenschlüssel als Primärschlüssel der Tabelle und eine frei wählbare Anzahl von Spalten, siehe u.a. [HTArch09]. Jede Spalte kann über den Zeitstempel beliebig viele Versionen eines Wertes haben, siehe [HBArch]. Ein Spaltenname hat die Form „<familie>:<bezeichnung>“, wobei „<familie>“ und „<bezeichnung>“ beliebige Bytearrays sind. „<familie>“ bezeichnet die sogenannte *Spaltenfamilie* der Spalte. Wie noch gezeigt wird, wird eine Spaltenfamilie lokal zusammenhängend im Sekundärspeicher gespeichert. Apache empfiehlt für Hbase, bei der Definition der Spaltenfamilien (und der darin enthaltenen Spalten) darauf zu achten, dass die Daten der Spaltenfamilie ähnlich sind und vergleichbare Schreib-/Lesecharakteristiken besitzen. Insofern wird die Drei-Ebenen-Schemaarchitektur (Abbildung 1) bzw. die physische Datenunabhängigkeit aufgeweicht, weil man schon bei der konzeptuellen Modellierung das physische Datenmodell und seine Auswirkungen auf die Lese- und Schreibperformance berücksichtigen muss.

Die Tabelle wird *dünnbesetzt* gespeichert, d.h. die Zeilen können eine variierende Anzahl von Spalten (aber nicht von Spaltenfamilien) haben, und nicht jede Spalte muss einen Wert speichern. Die Spalten werden dynamisch zu einer Spaltenfamilie beim Einfügen von Werte hinzugefügt, die Spaltenfamilien hingegen sind Bestandteil der Tabellenschemata.

Jede Zelle der Tabellen speichert ein uninterpretiertes Byte-Array, z.B. den Inhalt von Webseiten. Der Zugriff auf die Zellenwerte erfolgt über den Zeilenschlüssel, der ebenfalls ein uninterpretiertes Byte-Array ist. Die Zeilen im Sekundärspeicher sind anhand ihrer Zeilenschlüssel sortiert abgelegt, die Sortierung erfolgt bei Hbase z.B. durch ein Java-

Comparator-Objekt. Das Datenmodell besteht also aus denormalisierten Daten und weiten, dünnbesetzten Tabellen.

Tabelle 22 zeigt den konzeptuellen Ansatz einer Tabelle mit einer Menge dünnbesetzter Zeilen, die zu einem Zeilenschlüssel gehören und bei der nicht jede Spalte einen Wert für einen bestimmten Schlüssel hat. Die Tabelle besteht aus den Spaltenfamilien „contents“, „anchor“ und „mime“ sowie den Spalten „cnnsi.com“ und „my.loo.ca“ in der Spaltefamilie „anchor“. Der konzeptuelle Ansatz ist für Bigtable, Hbase, Hypertable und Cassandra identisch.

Row Key	**Time Stamp**	**Column „contents:“**	**Column „anchor:“**		**Column „mime:“**
„com.cnn.www“	t9		„anchor:cnnsi.com“	„CNN“	
	t8		„anchor:my.look.ca“	„CNN.com“	
	t6	„<html>...“			„text/html“
	t5	„<html>...“			
	t3	„<html>...“			

Tabelle 22: Konzeptuelle Sicht auf eine BigTable/HBase/HyperTable/Cassandra-Tabelle ([HBArch])

Ein weiteres Beispiel für einen Online-Shop aus [GoCo09], in der Waren (die die Zeilenschlüssel bilden) mit ihren von den Kunden zugeordneten Merkmalen (Tags) und Anmerkungen der Kunden hinterlegt sind, zeigt Tabelle 23. In den *tag*-Spalten ist die Anzahl der Kunden gespeichert, die der Ware diese Eigenschaften zugeordnet haben. Deutlich wird der Unterschied zu **RDBMS** bei den Spaltenfamilien *tag* und *review*, da die Spaltenbezeichnungen *review:jürgen* und *review:susanne* in einem **RDBMS** Tupel und nicht (wie hier) Attribute sind.

Row Key	tag: gestreift	tag: vierbeinig	tag: samten	tag: angenehm	Review: Jürgen	Review: Susanne	Preis:US	Preis:EU
Stofftier	803	212	0	6	„Ich mag das Stofftier“		25	20
Sessel	1	343	298	0		„Dies ist ein hervor-ragender Sessel“	200	175
Filzhut	3	0	145	0		„Ein sehr schöner Hut“	50	40

Tabelle 23: Online-Shop realisiert mit Hypertable [GoCo09]

3.1.5.3 Physische Speicherung

Alle Daten einer Spaltenfamilie werden wie beschrieben physisch zusammen gespeichert. Die physische Dateistruktur z.B. von Hbase ist [Seite 16 [Gray08]]: *SortedMap (RowKey,List (Sorted Map (Column,List (Value,Timestamp))))*. D.h. es werden Schlüssel-Wert-Paare für eine Spaltenfamilie in einer Dateistruktur (z.B. *SSTable* bei Bigtable, *MapFile* bei Hbase) gespeichert, siehe [Nab09].

Tabelle 24 zeigt die physische Speicherung der Daten in Tabelle 22. Die konzeptuelle Sicht wird physisch in drei Dateien (für drei Spaltenfamilien) gesichert, wobei jede Datei aus Schlüssel-Wert-Paaren besteht, die nach dem Zeilenschlüssel sortiert sind. Der Schlüssel für einen Wert setzt sich aber insgesamt aus dem Zeilenschlüssel, der Spaltenfamilie, der Spaltenbezeichnung und dem Zeitstempel zusammen, er ist für den Wert „CNN.com“ (beispielhaft) „com.cnn.www anchor:my.look.ca t8“.

Die leeren Zellen der konzeptuellen Sicht werden nicht in der physischen Datei gesichert. Eine Anfrage des Wertes der „contents:“-Spaltenfamilie für Zeitstempel t8 würde keinen Wert zurückerhalten, genauso wie eine Anfrage an die Spalte „anchor:my.look.ca“ für t9. Wird kein Zeitstempel bei der Anfrage übergeben, wird der jüngste Wert zurückgegeben, da die Zeitstempel in absteigender Folge gespeichert werden.

Eine Anfrage für alle Werte der Zeile „com.cnn.www“ gibt den Wert von „contents:“ für t6, den Wert von „anchor:cnnsi.com“ für t9, den Wert von „anchor:my.look.ca“ für t8 und den Wert von „mime:“ für t6 zurück.

Row Key	Time Stamp	Column „contents:“
„com.cnn.www“	t6	„<html>...“
	t5	„<html>...“
	t3	„<html>...“

Row Key	Time Stamp	Column „anchor:“	
„com.cnn.www“	t9	„anchor:cnnsi.com“	„CNN“
	t8	„anchor:my.look.ca“	„CNN.com“

Row Key	Time Stamp	Column „mime:“
„com.cnn.www“	t6	„text/html“

Tabelle 24: Physische Speicherung der Daten aus Tabelle 20 [HBArch]

Abbildung 37 illustriert die Schlüsselbildung und die Speicherung der Schlüssel-Wert-Paare. Der Zeilenschlüssel ist so gesehen nicht vergleichbar mit der logischen Tupel-ID bei einer vertikal partitionierten Relation. Sinn der Tupel-ID ist die Sicherstellung der Zusammengehörigkeit der Attributwerte der auf konzeptueller Ebene definierten Attribute einer Relation nach physischer Dekomposition. Sinn des Zeilenschlüssels als Präfix des Gesamtschlüssels ist die Sortierung der Werte in den Datenbankdateien. Die Zusammengehörigkeit der Spaltendaten wird durch den Gesamtschlüssel (Zeilenschlüssel, Spaltenfamilie, Spalte, Zeitstempel) sichergestellt. Nachfolgend werden noch eine Besonderheit von Hypertable und die vertikale Partitionierung der Daten am Beispiel von Hbase (die aber in allen Implementierungen identisch erfolgt) beschrieben.

crawldb Table

key	value
com.facebook.www title *2008-02-11 15:14:01*	Facebook I Home
com.facebook.www title *2008-02-03 19:27:57*	Facebook I Home
com.facebook.www title *2008-01-22 08:46:28*	Facebook I Home
com.facebook.www content *2008-02-11 15:14:01*	<!DOCTYPE html PUBLIC "-//W3C//DTD...
com.facebook.www content *2008-02-03 19:27:57*	<!DOCTYPE html PUBLIC "-//W3C//DTD...
com.facebook.www content *2008-01-22 08:46:28*	<!DOCTYPE html PUBLIC "-//W3C//DTD...
com.facebook.www anchor:com.apple.www/ *2008-02-11 15:14:01*	Facebook
com.facebook.www anchor:com.apple.www/ *2008-02-03 19:27:57*	Facebook
com.facebook.www anchor:com.apple.www/ *2008-01-22 08:46:28*	Facebook
com.facebook.www anchor:com.redherring.www/ *2008-02-11 15:14:01*	Facebook
com.facebook.www anchor:com.redherring.www/ *2008-02-03 19:27:57*	Facebook
com.yahoo.www title *2008-02-10 21:12:09*	Yahoo!
com.yahoo.www title *2008-02-04 03:46:22*	Yahoo!
com.yahoo.www title *2008-01-22 08:46:28*	Yahoo!
com.yahoo.www content *2008-02-10 21:12:09*	<html><head><meta http-equiv="Content-...
com.yahoo.www content *2008-02-04 03:46:22*	<html><head><meta http-equiv="Content-...
...	...

Abbildung 37: Schlüssel-Wert-Paare in Hypertable [HTArch]

Zugriffsgruppen

In Hypertable ermöglichen sogenannte *Access Groups*, Spaltenfamilien physikalisch zu gruppieren, siehe [HTTable]. Alle Spaltenfamilien einer AccessGroup werden physikalisch zusammenhängend im selben CellStore (das Dateiformat von Hypertable) gespeichert. Ein zeilenorientierter Ansatz kann durch das Gruppieren aller Spaltenfamilien in einer einzigen AccessGroup simuliert werden, ein spaltenorientierter durch das Erzeugen einer eigenen Access Group pro Spaltenfamilie. Durch AccessGroups kann die Lese- und Schreibperformance gesteigert werden, indem die Spaltenfamilien, auf die am häufigsten zugegriffen wird, zusammenhängend gespeichert werden, [GoCo09]. Das gleiche Konzept bietet Bigtable in Form der sogenannten *lokalen Gruppen*.

Der Optimierungsansatz für Ad-hoc-Anfragen anderer **coDBMS** im Rahmen des OLAP, der ja gerade zum Ziel hat, dass der Anwender ohne Planung der physikalischen Speicherstruktur solche Anfragen effizient ausführen kann, wird hier also nicht verfolgt. Bei den vier Implementierungen handelt es sich mangels GROUP BY-Funktion und Aggregierungsfunktionen auch nicht um Datenbanken, die für **OLAP** konzipiert wurden, wie noch gezeigt wird.

Horizontale Paritionierung

Aus Sicht einer Anwendung erscheint eine Tabelle als Liste von Tupeln, die nach ansteigendem Zeilenschlüssel, Spaltennamen und Zeitstempel sortiert sind. Physisch fragmentiert werden diese Zeilenbereiche (engl. *Row Ranges*) von Tabellen bei Hbase z.B. in sogenannte *Regionen* (engl. *Regions*), bei Bigtable in *Tablets*. Jeder Zeilenbereich enthält Zeilen von einem Startschlüssel bis zu einem Endeschlüssel. Eine Menge von Regionen/Tablets bildet eine Tabelle. Im Gegensatz zu BigTable, das einen Zeilenbereich durch den Tabellennamen und den Endeschlüssel identifiziert, ist es bei Hbase der Tabellenname und Startschlüssel.

3.1.5.4 Dateiformat und Indizierung

Die gespeicherten Daten werden anhand des Schlüssels indiziert, wobei nicht geklärt werden konnte, ob nur der Zeilenschlüssel oder der ganze Schlüssel eines Wertes indiziert wird. Die Indizierung wird beispielhaft für Hbase, Hypertable und Bigtable beschrieben.

Hbase

Jede Spaltenfamilie (vertikale Partition, Kapitel 2.13.1) einer Region (horizontale Partition) wird von Hbase in einem separaten sogenannten *Hstore* gespeichert. Ein Hstore besteht aus eine oder mehrerer Mapdateien (engl. *MapFiles*), eine Google-SSTable-ähnliche Dateistruktur des HDFS zum Speichern von sortierten Schlüssel-Wert-Paaren. Ein MapFile wartet wie SSTable einen dünnbesetzten (bezogen auf die Anzahl der Schlüssel) In-Memory-Index, d.h. der Index ist vollständig im Hauptspeicher geladen. Über eine Suche im Index gelangt man zum Block mit dem naheliegendsten Schlüssel, von dort aus, wird die Datei sequenziell durchsucht, siehe [MFIndex].

Hypertable

Bei *Hypertable* werden die Schlüssel-Wert-Paare in sogenannten CellStore-Dateien gespeichert. Physikalisch werden sie als Folge komprimierter Blöcke (à 65KB) abgespeichert. Am Ende der Blockfolge (der Datei) wird ein Index mit einer Liste des jeweils letzten Schlüssels jedes Blocks und der dazugehörige Block-Offset gespeichert. Auch hier wird der Index wird beim Laden des CellStore in den Hauptspeicher geladen, siehe [HTArch09].

BigTable

Das Dateiformat zum Speichern der Bigtable-Daten ist das Dateiformat SSTable des Google File Systems. Das Indizierungsprinzip ist identisch mit Hbase und Hypertable. Für Cassandra wurde auf eine Recherche verzichtet, da das Prinzip vermutlich das gleiche ist.

3.1.5.5 Bloomfilter

Die Implementierungen ermöglichen pro Spaltenfamilie den Einsatz eines Bloomfilter-Index [Bloom]. Hierbei handelt es sich um eine Hashstruktur, mit der die Existenz von Wörtern in einer Datenmenge effizient geprüft werden kann. Im Fall von Bigtable und den weiteren Implementierungen wird dadurch die Prüfung, ob ein Spaltenname in einer Spaltenfamilie enthalten ist, verkürzt und damit auch die Anfragezeit, siehe [HBBloom]. Dies ist bei diesen vier **coDBMS** insofern von Bedeutung, da sie auf das Erzeugen von Millionen von Spalten in den Spaltenfamilien ausgelegt sind. Da der in Kapitel 3.1.5.4 genannte Index ein dünnbesetzter Hauptspeicherindex ist und aus der recherchierten Literatur auch nicht hervorgeht, ob der Spaltenname Teil dieses Index ist, müssten bei der Suche nach Spaltennamen unter Umständen viele Blöcke erfolglos durchsucht werden. Der erfolglose Zugriff auf den Sekundärspeicher wird durch den im Hauptspeicher existenten Bloomfilter verhindert. Die nachfolgende Beschreibung des Bloomfilters stammt aus [WBloom].

Der Bloomfilter erzeugt mit Hashfunktionen einen „Fingerabdruck“ eines Datensatzes, in diesem Fall eines Spaltenbezeichners. Die Hash-Werte (der Spaltenbezeichner) werden nun nacheinander in einen zunächst mit Nullen gefüllten Bitstring geschrieben, der dieselbe Länge hat wie jeder Wert. Dort, wo ein Hash-Wert eine 1 enthält, wird eine 1 in das Array geschrieben, bei einer 0 bleibt der bisherige Wert erhalten. Es handelt sich also um eine binäre Oder-Funktion. Damit nicht sehr bald im Array nur noch Einsen stehen, sollte die Hash-Funktion Werte liefern, die überwiegend Nullen enthalten.

Soll nun überprüft werden, ob ein beliebiges Wort im Vokabular enthalten ist, wird auch dessen Hash-Wert ermittelt. Hat er irgendwo eine Eins, wo im Bitstring eine Null steht, kann das Wort nicht enthalten sein. Ist dies aber nicht der Fall, muss das Wort dennoch nicht zwingend im Vokabular enthalten sein, denn das übereinstimmende Bitmuster kann durch die Überlappung mehrerer anderer Hash-Werte zustande kommen, oder auch dadurch, dass zwei Wörter den gleichen Hash-Wert haben. Es kann also Falsch-Positiv-Prüfungen geben, die dann einen erfolglosen Zugriff auf den Sekundärspeicher zur Folge haben. Dennoch verringert der Bloomfilter diese Zugriffe.

3.1.5.6 Anwendungsschnittstellen

Nachfolgend werden beispielhaft für drei Implementierungen (Hbase, Bigtable, Hypertable) die (ähnlichen) Anwendungsschnittstellen vorgestellt.

Bigtable

Die C++-API erlaubt das Erzeugen, Ändern und Löschen von Tabellen und Spaltenfamilien und das Ändern von Metadaten wie z.B. Zugriffsrechten. Ebenso kann damit über alle mehrere Spaltenfamilien iteriert werden. Es existieren mehrere Mechanismen, um die durch einen Scan erzeugten Zeilen, Spaltem und Zeitstempel einzuschränken. Weiterhin unterstützt Bigtable Single-Row-Transaktionen für Daten, die unter einem bestimmten Row Key abgelegt sind, siehe [Chang06].

Hbase

Es gibt bei Hbase keine Möglichkeit, eine Liste aller Spalten einer Tabelle abzufragen. Hierfür ist ein FullTable-Scan notwendig. Das Abfragen der Spaltenfamilien ist hingegen möglich, da diese nichtveränderlich sind ([Jimbojw09]).

Hbase bietet neben einer proprietären Java-**API** noch eine sogenannte Hbase-Shell zum Erzeugen, Löschen, Ändern und Anfragen der Tabellendaten, siehe [HbShell] und [Seite 22 [Kel07]]. Es wird dabei eine proprietäre, nicht SQL-konforme Syntax verwendet. Der Zugriff auf die Daten erfolgt durch einen get-Befehl unter Angabe des Zeilenschlüssels, optional der Spaltenfamilie, des Spalten-Identifiers und des Zeitstempels (je nachdem, wie granular die Suche sein soll). Als Erweiterung des get-Befehls kann der scan-Befehl nicht nur die Werte für einen Zeilenschlüssel, sondern für einen Bereich von Zeilenschlüsseln ausgeben, siehe [ScanClass]. Verbunde von Daten sind wie weiter oben bereits beschrieben über die Anfrageschnittstellen nicht möglich, sondern nur auf Applikationsebene über *MapReduce*, welches noch beschrieben wird, möglich. MapReduce realisiert einen Batch-Zugriff auf die Daten (zu Analysezwecken von z.B. Webcrawlingdaten), ein wahlfreier Zugriff, z.B. für Crawler-Updates, erfolgt über die Hbase-Shell oder die Java-**API**.

Bigtable bietet statt einer Java-API eine C++-API mit identischen Funktionen [Chang06], Hypertable bietet neben der C++-API genau wie Hbase ebenfalls eine Shell mit im Gegensatz z.B. zu Hbase SQL-ähnlicher Syntax (CREATE, DELETE, DROP, SELECT, SHOW TABLES).

3.1.5.7 Konkurrierende Zugriffe

Die Implementierungen bieten ein Locking auf Zeilenebene, d.h. die Zeile wird während eines Updates für andere Zugriffe gesperrt [Chang06]. Dies gilt auch für Zeilen mit hunderten von Spalten.

3.1.5.8 Systemarchitektur

Die Systemarchitektur ist bei allen Implementierungen bis auf Cassandra identisch, sie wird deshalb nur einmal anhand von Hbase beschrieben. Für die Beschreibung von Cassandras Systemarchitektur war die Bearbeitungszeit leider nicht mehr ausreichend.

Tabellen sind wie in Kapitel 3.1.5.3 beschrieben horizontal in Regionen/Tablets partitioniert. Jede Region (bei Bigtable: Tablet) kann auf einem anderen Knoten liegen und besteht aus mehreren replizierten Hadoop-MapFiles (Bigtable: SSTables). Es gibt drei (physische) Rechnerknotentypen, den *Masterserver,* den *Regionserver (Bigtable: Tabletserver)* und den *Client.*

Die (logische) Verwaltung der bereits vorgestellten Regionen folgt einem dreistufigen Aufbau, bestehend aus *ROOT*-Region (als Wurzel), *META*-Regionen und *User*-Regionen.

Die *ROOT*-Region lokalisiert alle *META*-Regionen. Jede META-Region lokalisiert wiederum eine Menge von *User*-Regionen, die sämtliche Tabellen einer Hbase-Datenbankinstanz umfassen, siehe [HBArch].

Spezielle Tabellen, *ROOT* und *META*, speichern Schemainformationen (also die definierten Tabellen und Spaltenfamilien) und die Speicherorte der Regionen. Die ROOT-Tabelle enthält ein sogenanntes HregionInfo-Objekt für jede META-Region und den Ort des Hregion-Servers, der die META-Region anbietet, siehe [HBArch]. Die META-Tabelle enthält ebenfalls HregionInfo-Objekte, die Informationen wie Start- und Endeschlüssel einer User-Region, Status der User-Region (on- oder off-line) und die Adresse des Hregion-Servers, auf dem die Region zu finden ist, speichern. Jede Zeile der META-Tabelle zeigt also auf eine User-Region. Abbildung 38 zeigt die Verwaltung der User-Regionen in Hbase.

Der Masterserver ordnet den Regionservern User-Regionen zu. Ein Regionserver ist verantwortlich für das Verarbeiten von Lese- und Schreibanfragen der Clients (für die User-Regionen des Servers). Die in den Leseanfragen angeforderten Daten werden in den MapFiles der Hstores (die die Tabellendaten einer Region speichern) gesucht. Jeder Region-Server

kommuniziert mit dem Master-Server, um eine Liste von Regionen zu erhalten, die er anbieten soll, und um seine Aktivität anzuzeigen.

Wie ein Master-Server das Loadbalancing durchführt, die Region-Server überwacht und wie der Region-Server überlaufende Regionen aufsplittet und der Master- diese dann neu auf die Region-Server neu verteilt, wird in dieser Arbeit nicht näher erläutert, Informationen hierzu finden sich in den genannten Literaturquellen dieses Unterkapitels.

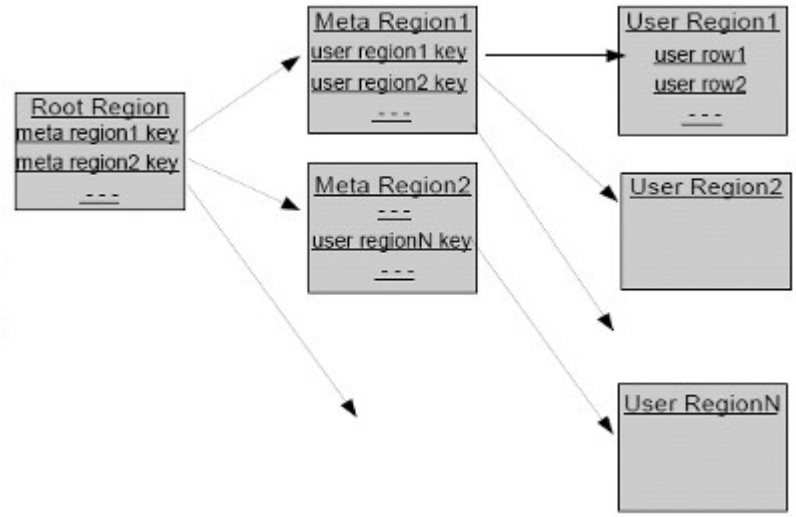

Abbildung 38: Logische Verwaltung der Regionen in Hbase [Kel07]

Der Hbase-*Client* ist verantwortlich für das Auffinden von Regionservern, die den benötigten Zeilenbereich einer Tabelle anbieten. Er kommuniziert einmalig mit dem Masterserver, um den physischen Ablageort der ROOT-Region (bzw. der ROOT-Tabelle) zu finden. Sobald diese gefunden ist, kontaktiert der Client den entsprechenden Regionserver und scannt die ROOT-Region, um die META-Region zu finden, die den Ort der User-Region mit dem gewünschten Zeilenbereich enthält. Der Client kontaktiert dann den Regionserver, der die META-Region anbietet und scannt diese META-Region (bzw. die META-Tabelle), um den Ort der User-Region zu bestimmen. Nach dem Lokalisieren der User-Region kontaktiert der Client den Region-Server dieser Region und stellt die Lese- oder Schreibanfrage.

Abbildung 39 zeigt die Umsetzung für Hypertable, wobei der Regionserver von Hbase hier Rangeserver heißt [HTArch09]. Der Masterserver hat bei Hypertable einen Hotstandby-Server, der bei Ausfall des regulären Masterservers dessen Funktion übernimmt. Der **DFS**-Broker abstrahiert die Schnittstelle zum Dateisystem, damit Hypertable auf einem beliebigen

Dateisystem arbeiten kann. Er übersetzt standardisierte Dateisystemnachrichten in Systemaufrufe des jeweiligen Dateisystems. **DFS**-Broker existieren für **HDFS**, KFS (jetzt Cloudstore) und das lokale Dateisystem.

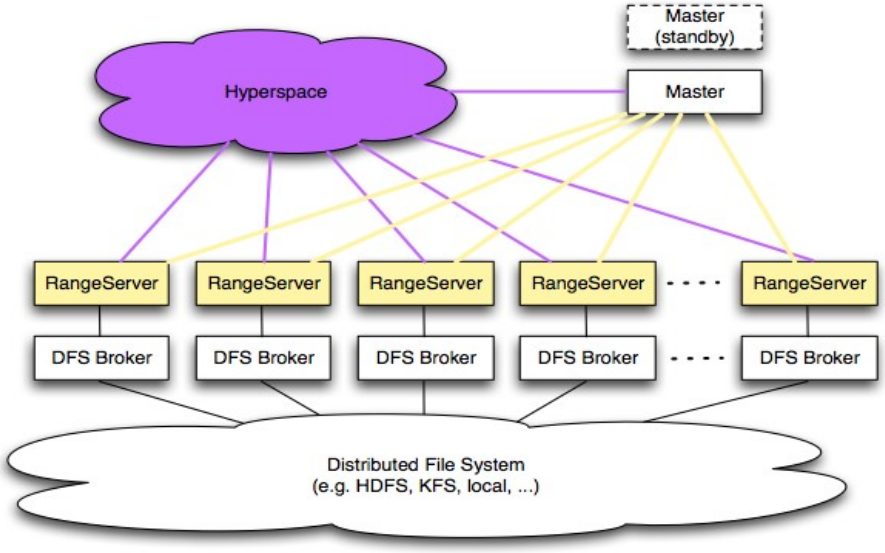

Abbildung 39: Systemarchitektur von Hypertable [HTArch09]

3.1.5.9 Google File System (GFS), Hadoop Distributed File System (HDFS) und Amazon Dynamo

Die **coDBMS** setzen auf verschiedenen verteilten Dateisystem auf und nutzen die angebotenen Dateistrukturen für Schlüssel-Wert-Paare, die Skalierbarkeit und die Replizierungsmechanismen. Sie erweitern die DFS z.B. um den wahlfreien Zugriff auf die Daten, siehe für Hbase [Seite 10 [Gray08]]). Die genannten Dateisysteme sind für den Einsatz auf Commodity-Hardware konzipiert. Die Replizierung der Datenblöcke erfolgt ohne Einsatz von RAID-Technologie. HDFS z.B. wird z.B. bei Yahoo auf mehreren zehntausend Rechnern eingesetzt [Seite 5 [Gray08]], Google hat mit dem GFS sein eigenes verteiltes Dateisystem entwickelt.

HDFS und **GFS** haben eine Master-Slave-Architektur, bei der ein *MasterNameNode*-Rechner die Orte aller Blöcke verwaltet und *SlaveDataNodes*-Rechner die Blöcke im lokalen Dateisystem speichern. Cassandra nutzt das verteilte Dateisystem *Amazon Dynamo*. Dieses soll nur insoweit erläutert werden, dass es hier im Gegensatz zu **GFS** und **HDFS** keinen zentralen Knoten gibt, sondern ein Verteilungsmodell gleichberechtigter Knoten, siehe

[MArch09]. Hieraus ergeben sich auch Unterschiede zu der in Kapitel 3.1.5.8 vorgestellten Systemarchitektur, die aber nicht näher erläutert werden sollen.

3.1.5.10 Google und Hadoop MapReduce

MapReduce ist ein (proprietäres) Software-Framework von Google [MapRed], das vom Hadoop Software-Projekt (Open Source) der Apache Software Foundation nachgebildet wurde. Es dient der verteilten Berechnung großer Datenmengen in Batch-Läufen.

Map Reduce verarbeitet die Daten mit sogenannten Map- und Reduce-Funktionen. Aus einer Liste von Schlüssel-Wert-Paaren (Eingabedaten) wird eine Liste von Schlüssel-Wert-Paaren (Ausgabedaten) erzeugt, siehe [WikiMR].

Das Framework ruft auf jedem Paar der Eingabedaten die Map-Funktion auf, die daraus ein Zwischenergebnis, ebenfalls eine Liste von Schlüssel-Wert-Paaren (Zwischenwertliste), erzeugt. All diese Map-Berechnungen sind voneinander unabhängig, so dass man sie nebenläufig und verteilt auf einem Rechnerverbund ausführen kann. Die Reduce-Funktion berechnet aus der Zwischenwertliste eine Liste von Ergebniswerten. Auch die Aufrufe von Reduce können unabhängig auf verschiedene Prozesse im Computercluster verteilt werden. Die Liste von Ergebniswerten wird vom Framework in einer Ausgabeliste bestehend aus Schlüssel-Wert-Paaren gesammelt.

Ein Beispiel ist das Zählen des Vorkommens von Wörtern in einer großen Textdatei. Hierfür durchläuft die Map-Funktion (oder die parallelen Funktionen) das (ggf. gesplittete) Dokument Wort für Wort. Jedes Mal, wenn ein Wort *w* angetroffen wird, wandert eine 1 in die *w*-Zwischenergebnisliste (falls diese noch nicht existiert, wird sie angelegt). Ist man mit allen Wörtern durch und hat der Text insgesamt *n* verschiedene Wörter, so endet die Map-Phase mit *n* Zwischenergebnislisten, jede für ein anderes Wort sammelnd, welche so viele 1-Einträge enthält, wie das entsprechende Wort im Dokument gefunden wurde. Die Reduce-Funktion wird jeweils für das Wort *word* und seine Zwischenergebnisliste aufgerufen. Die Zwischenergebnisliste wird durchlaufen, alle gefundenen Zahlen werden addiert. Die Summe wird an das Framework zurück gegeben, sie enthält, wie oft das Wort *word* im Dokument gefunden wurde [WikiMR].

3.1.5.11 Kompression

Die vorgestellten Implementierungen setzen jeweils ein einheitliches Kompressionsverfahren für die (unstrukturierten) Daten ein, sofern die Verwendung von Kompression bei der Erzeugung der Tabelle für eine Spaltenfamilie angegeben wurde.

Hbase z.B. setzt mit Lempel-Ziv-Oberhumer (LZO, [LZO]) ein auf Lempel-Ziv basierendes, blockorientiertes Kompressionsverfahren ein, dessen Schwerpunkt auf einer schnellen Komprimierung und Dekomprimierung von Daten liegt. LZO wird seit 1996 entwickelt. Es setzt ein Schiebefenster ein, basiert also auf LZ77 (und nicht LZ78), siehe Kapitel 2.12.3.1 und [LZODoc]. Die Dekomprimierung findet direkt beim Lesen der Blöcke statt, die Daten liegen also in dekomprimierter Form im Datenbankpuffer. Es werden also die bereits erwähnten Vorteile geringerer I/O-Kosten genutzt, ein Arbeiten mit komprimierten Daten durch den Query Processor ist jedoch nicht vorgesehen (da ja auch keine Verbund-, Gruppierungs- und Aggregierungsfunktionen ausgeführt werden, bei denen die Vorteile des Arbeitens mit komprimierten Daten zum Tragen kommen). Hbase nutzt die native LZO-Bibliothek, die Bestandteil vieler Linux-Systeme ist. Beim Anlegen einer Tabelle z.B. in der Hbase-Shell muss angegeben werden, das Kompression eingesetzt werden soll. Die Festlegung wird dabei pro Spaltenfamilie getroffen. Beispiel ist der Befehl `create 'mytable', {NAME=>'colfam:', COMPRESSION=>'lzo'}`.

Auch Google kann optional Kompression einsetzen, es verwendet das Verfahren BMDiff (Wörterbuchkompression für mehrfach vorkommende Worte) oder alternativ das proprietäre Verfahren Zippy (ebenfalls LZO-ähnlich).

3.1.5.12 Beispiel-Anwendung für Bigtable, Hbase, Hypertable und Cassandra

Da es sich bei den vorgestellten DBMS nicht um solche handelt, die für OLAP konzipiert wurde (weil analytische Funktionen fehlen), soll nachfolgend soll beschrieben werden, für welche Art von Anwendungen diese Art von coDBMS verwendet werden.

Gemäß [Chang06] wird Bigtable unter anderem für die Webindizierung, Google Maps, Google Book Search, Google Earth und Youtube genutzt

[Seite 17 [Gray08]] beschreibt ein Beispiel einer Anwendung basierend auf Hbase für die Speicherung der Daten eines Webcrawlers und vergleicht dies mit einer entsprechenden Umsetzung in einem **RDBMS**. In Hbase wird dazu eine Tabelle namens *crawl* und die Spaltenfamilie *content angelegt.* Pro Zeile wird ein **URL** mit den Spalten *content:data* (speichert Rohdaten), *content:language* (speichert http-language-header) und *content:type*

(speichert http-content-header) gespeichert. Bei Hyperlinks und Bildern werden die Spaltenfamilien *links:<url>* für jeden Link (**URL** ist der Schlüssel, der Link der zugeordnete Wert) und *images:<url>* für jedes Bild hinzugefügt.

In einem **RDBMS** wird dafür beispielhaft eine Relation *crawl* mit den Attributen *url*, *data*, *language* und *type*, eine Relation *links* mit den Spalten *url* und *links* und *e*ine Relation *images* mit den Attributen *url* und *image*. Bei 10 Millionen Webseiten mit jeweils 10 Links und 10 Bildern ergibt dies gemäß [Seite 20 [Gray08]) im RDBMS 210 Millionen Tupel in den Relationen und, soweit eingesetzt, einem großen Index aufgrund der Tabellen links und images, in einer Map wie BigTable oder Hbase 10 Millionen Zeilen. Hier ist jedoch anzumerken ist, dass ein Vergleich nur bezogen auf die Zeilenanzahl nicht vollständig ist, denn letztendlich sind die jeweils 100 Millionen Zeilen für die Relationen *links* und *images* bei Bigtable, Hbase, Hypertable oder Cassandra in Form von jeweils 100 Millionen Spalten für die genannten Spaltenfamilien vorhanden.

Deutlich wird jedoch, dass neben dem Anwendungsgebiet des klassischen **OLAP** auch noch das Anwendungsgebiet des Webcrawling mit anschließender Webindizierung (ausgeführt in Batch-Läufen durch Map Reduce), als weiteres Einsatzgebiet von **coDBMS** vorhanden ist. Inwieweit die Webindizierung ebenfalls als analysierende Verarbeitung betrachtet werden kann, wird hier nicht weiter diskutiert, jedoch festgestellt, dass hierbei keine auswertenden Funktionen (Gruppierungen, Aggregierungen) auf konsolodierten und strukturierten, sondern unstrukturierten Daten (Bytearrays) ausgeführt werden. Die Webindizierung wird im Gegensatz zum klassischen OLAP nicht mit dem Ziel der Unterstützung für (unternehmens-)strategische Entscheidungen (decision support) durchgeführt, sondern ist lediglich eine Auswertung darüber, welche Wörter in welchen Webseiten vorkommen (z.B. zum Einsatz für eine Websuchmaschine).

Interessant wäre eine vergleichende Betrachtung für dieses Einsatzgebiet zwischen (ggf. dekomposierten) **roDBMS**, relationalen **coDBMS** und den Map-Ansätzen, diese steht jedoch inhaltlich nicht im Fokus dieser Diplomarbeit. Im Ausblick (Kapitel 4) wird darauf noch einmal eingegangen.

Ergänzend wird auf die Erweiterung *Hbase Writer* hingewiesen, die ein Webcrawler ist, der seine Ergebnisse direkt in eine Hbase-Tabelle schreibt (Seite 20 [Gray08]).

3.1.5.13 Ansätze von Bigtable, Hbase, Hypertable und Cassandra

Tabelle 25 fasst die bei dieser Implementierung vorhandenen Ansätze nach den genannten Betrachtungsschwerpunkten zusammen.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Speicherung der horizontal partitionierten Spaltenfamilien sortiert nach dem Zeilenschlüssel. Verwendung eines dünnbesetzten Hauptspeicherindex für die Spaltenfamilien und optional eines Bloomfilter-Indexes für die Anfrageoptimierung nach Spaltenbezeichern.
2	*Algorithmen* (DB-Operationen, Kompression)	Optionaler Einsatz eines einheitlichen Kompressionsverfahrens (LZO) pro Spaltenfamilie.
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Jeweils proprietäre Java- oder C++-API oder proprietäre Kommandozeilenanwendung mit ebenfalls proprietärem Befehlssatz zum Erzeugen, Ändern und Löschen von Tabellen sowie zum wahlfreiem Zugriff auf die Dimensionen (Zeilenschlüssel, Spaltenfamilie, Spalte, Zeitstempel). Alternativ Scan-Funktion für das Scannen von Zeilenschlüsselbereichen (statt einzelner Zeilenschlüssel). MapReduce zum Batchzugriff auf große Datenmengen. Verwendung eines multidimensionalen Datenmodells mit vier Dimensionen, jedoch ohne Datentyp (unstrukturierte Daten) und ohne Verbund zwischen Daten. Keine Verbundoperationen, keine GROUP BY- oder Aggregierungsfunktionen (d.h. kein OLAP).
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Kein spaltenorientierter Optimierer, sondern lediglich effizienter Zugriff auf Schlüssel-Wert-Paare (ohne Prädikatformulierung oder Verbunde).
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Nothing-Architektur (Kapitel 2.13) mit entweder Master-/Slave-Rechnerknoten[29] oder gleichberechtigten Knoten (P2P-Modell[30])
6	*Transaktionsverwaltung* (Update-Problematik)	Schreibtransaktion umfasst jeweils nur eine Tabellenzeile, diese wird komplett gesperrt.
7	*Betriebssystem*	Linux bzw. plattformunabhängig

Tabelle 25: Ansätze / Klassifizierungskriterien für Bigtable, Hbase, Hypertable und Cassandra

3.1.6 Valentina Database

Valentina ist ein objektrelationales (vgl. [Seite 536 [SSH08]]), spaltenorientiertes DBMS der Firma Paradigm, das sowohl eine lokale, eingebettete Datenbank-Engine zum Zugriff auf

[29]Bei Google File System oder Hadoop Distributed File System

[30]Bei Verwendung von Amazon Dynamo

lokale Datenbanken als auch auch auf einen Datenbankserver zugreifen kann. Der Produktseite nach arbeitet es nicht mit B-Bäumen, sonderen anderen Datenstrukturen für Indizes und benutzt Bitmaps bei der Anfragebearbeitung ([Val09]). Es unterstützt SQL-92 und bietet eine API für die Datenbank-Engine. Inwieweit der Optimierer die Spaltenorientierung nutzt, ist aus der Literatur nicht erkennbar. Die vorhandene Dokumentation bezieht sich im Wesentlichen auf die SQL-Syntax, die Schwerpunkte dieser Diplomarbeit (Spaltenoptimierung, Anfrageoptimierung, verwendete Zugriffsstrukturen) werden mit wenigen Worten behandelt ([Val09-2]). Insofern kann eine Klassifizierung nur eingeschränkt durchgeführt werden, siehe Tabelle 26.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Spaltenorientierung
2	*Algorithmen* (DB-Operationen, Kompression)	-
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Objektrelationales Datenmodell SQL-92
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Verwendung von Bitmaps bei Anfrageverarbeitung
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Embedded-Datenbank für lokale Datenbank oder Single-Server-Datenbank
6	*Transaktionsverwaltung* (Update-Problematik)	-
7	*Betriebssystem*	Windows (verwendet DLLs)

Tabelle 26: Klassifizierungskriterien für Valentina Database

3.1.7 Vectornova / Vectorstar High-Speed Data Engine

Vectorstar ist ein als Data Warehouse konzipiertes relationales, vektorisiertes coDBMS der Firma Vectornova mit dem Schwerpunkt der Unterstützung von OLAP-Anwendungen. Es bietet einen multidimensialen Data Cube sowie wahlweise eine Shared-Nothing- oder Shared-Disk-(**SAN**)-Architektur. Es ist für die Betriebssysteme Windows, Linux, FreeBSD und MacOS/X verfügbar und bietet neben Datenkompression und verschiedenen Indizes ANSI-SQL und VectorSQL als eigenen Dialekt ([VSHome]). VectorStar speichert jede Spalte als eigene Datei ab ([VSOverview]). Dabei wird das logische Schema im Dateisystem abgebildet,

d.h. die Datenbankobjekte (Datenbank, Tabellen, Spalten) erhalten jeweils eigene Verzeichnisse. Die unterste Verzeichnisebene bilden dabei die Verzeichnisse für die Spalten. Jedes dieser Unterverzeichnisse enthält eine Datei, die wiederum die Daten einer Spalte enthält. Jede Datenbank wird inklusive ihrer Metadaten in einem Verzeichnis gespeichert und kann mit Betriebssystem-Bordmitteln komplett an eine andere Stelle kopiert werden. Auch wenn die Ausführungen relativ knapp gehalten sind, so scheint VectorStar beim Ausführen von Selektionen (WHERE-Statement) Operatoren genau wie z.B. C-Store zu haben, die Bitmaps im Anfrageplan für die Spalten, auf die das Prädikat angewendet wird, erzeugen, nur dass diese hier *Result Set Index* (NDX) heißen, siehe [VSOverview]. Diese repräsentieren die Menge der selektierten Zeilen.
Weitere Informationen insbesondere zur Spaltenarchitektur sind lt. Hersteller leider erst in der Erstellung [VectorStarDoc]. Tabelle führt die Ansätze von Vectornova auf.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Pro Spalte eine Datei
2	*Algorithmen* (DB-Operationen, Kompression)	-
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Relationales Datenmodell Data Warehouse SQL
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	Query Processor arbeitet mit Bitstrings
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Nothing oder Shared-Disk
6	*Transaktionsverwaltung* (Update-Problematik)	-
7	*Betriebssystem*	-

Tabelle 27: Ansätze / Klassifizierungskriterien für Vectornova

3.1.8 kdb(+)

Die nachfolgenden Beschreibungen wurden der zu kdb+ recherchierten Literatur entnommen. kdb+ (aktuelle Version ist 2.5) ist ein auf der Programmiersprache K (siehe Unterkapitel

3.1.8.1) basierendes, relationales, skalierbares, hauptspeicherbasierendes (engl. *In-Memory*) **coDBMS** ([KWiki09], [Kx09]), das sowohl für die In-Memory-Analyse von Streamingdaten als auch großer historisierter persistenter Datenbestände in Echtzeit konzipiert wurde. Insbesondere können Streaming-Daten mit historischen Daten verglichen bzw. Rohdaten historisiert gesichert werden.

Der Schwerpunkt sowohl von K als auch von kdb+ liegt bei Anwendungen im Finanz- und Bankgewerbe (z.B. Wertpapierhandel basierend auf Echtzeitanalyse zur Unterstützung von Kaufs- und Verkaufsentscheidungen, siehe Seite 4 [Kx09]), es handelt sich um ein Decision-Support-System (siehe Kapitel 2.10), das aber nicht auf einem Data Warehouse (also einen integrierten, materialisierten Datenbestand) aufsetzt.

Der Vorgänger von kdb+, kdb, wurde auf Basis von K 1998 entwickelt. kdb beinhaltet als DDL, DML und Anfragesprache *Ksql*, eine Sprache mit einer SQL92-ähnlichen Syntax, die für spaltenbasierte Anfragen und Array-Analysen konzipiert wurde. Der Nachfolger kdb+ ist die in 2003 entwickelte 64-Bit-Version von kdb. Kdb+ wiederum beinhaltet die Sprache *q*, die die Funktionalitäten der Sprachen K und *Ksql* vereint. Weitere unterstützte Programmiersprachen sind R, Matlab, C/C++, Java, Visual Basic, .NET, Perl und Python. Auch ODBC und JDBC werden unterstützt.

Die Features von kdb+ sind:

- Spaltenorientierte Architektur, die Spalten werden als ganzes jeweils einer Datei im Dateisystem des Betriebssystems gespeichert.
- Echtzeit-Indizierung der Spalten. Neuankommende Streamdaten werden automatisch indiziert.
- Die Sprache q als integrierte Sprache sowohl zum Entwickeln von Programmen als auch zum Anfragen der Daten
- Beim Entwickeln des DB-Schemas existiert die Zeitdimension bereits.
- Datentypen: Date, time, timestamp (für Zeitreihenanalysen), Array-Datentypen
- Unterstützung für Grid-Architekturen
- Multi-Threading zum Verteilen der Prozesse auf mehrere CPUs eines Servers oder verteilter Server.

- Skalierbarer Einsatz auf Commodity-Servern (Standardserver)

Kdb+ bietet eine in-memory-Datenbank mit der Fähigkeit, Kdb ist auf das Speichern und Wiederverwenden von nichtkonsoldierten Rohdaten spezialisiert (Seite 4 [Kx09]).Rohdaten historisiert zu sichern. Die Algorithmen arbeiten auf Streaming-Daten der Börsen, Handel werden automatisch in der In-Memory-Datenbank ausgeführt und gespeichert, diese wird historisiert geloggt. Kdb+ ist verfügbar für Solaris, Linux, OSX und Windows (32 oder 64-bit).

Die Speicherung von Daten erfolgt im Dateisystem des Betriebssystems. Die Daten werden dabei von kdb+ spaltenweise abgelegt. Big tables (billions of rows) are horizontally partitioned (often by date) and then splayed across a directory.

3.1.8.1 Die Programmiersprache K (jetzt q)

K ist eine Variante der Programmiersprachen *APL* und *Scheme*. Die Programmiersprache K wurde 1993 durch Arthur Whitney entwickelt und wird von dessen Firma Kx-Systems kommerziell vertrieben. K ist eine Interpreter-Programmiersprache zum Verarbeiten von Arrays. Es bietet mathematische und andere Funktionen, die auf den Arrays arbeiten, z.B. Sortierung und Umkehrung der Reihenfolge. Durch das Verketten von Ausdrücken können Transformationsketten für Daten aufgebaut werden.

Die Funktionen und Operatoren werden durch einzelne oder doppelte ASCII-Zeichen ausgedrückt. Die Funktionen sind dabei überladen, d.h. ein Funktionssymbol kann abhängig vom Kontext verschiedene Bedeutungen haben. An dieser Stelle wird auf die Beispiele in [KWiki09] verwiesen.

Durch die geringe Größe des Interpreters können K-Applikationen im Prozessor-Cache ausgeführt werden. K besaß außerdem eine integrierte GUI-Bibliothek, ab Version K4 ist dies nicht mehr der Fall.

3.1.8.2 Ansätze von kdb+

Tabelle 28 führt die Kriterien von kdb+ auf.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	Spaltenorientierte Ablage indizierter Daten im Dateisystem
2	*Algorithmen* (DB-Operationen, Kompression)	Hauptspeicherdatenbank
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Relationales Datenmodell Echtzeit-Streaminganalyse aus heterogenen Quellen. Historisierte, spaltenorientierte Ablage JDBC, ODBC
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	Shared-Nothing-Umgebung
6	*Transaktionsverwaltung* (Update-Problematik)	
7	*Betriebssystem*	Solaris, Linux, Windows

Tabelle 28: Ansätze von kdb+

3.1.9 Skytide XOLAP-Server

Auf der Homepage der Firma Skytide [SkyHome] findet sich kein direkter Hinweis auf den XOLAP-Server, der auf [WikiCODBMS] genannt ist. Die angebotenen Produkte heißen „Insight for Digital Media“ und „Insight for Content Delivery Networks (CDN)“ und sind Analysewerkzeuge für Medienunternehmen zum Messen der Profitabilität von digitalen Angeboten (Insight for Digital Media), [SkyIDM], und zum Messen der Servicequalität (Datenfluss über die Netzwerkknoten, abgerufene Datenvolumen), [SkyICDN].

„Insight for Digital Media“ kann verschiedene Datenbestände mit einem definierten Analysemodell oder Ad-hoc-Anfragen auswerten, u.a. die Logdateien verschiedener Medienserver (Microsoft Media Server, RealServer, Apache-Webserver, Adobe Flash) und verschiedene DBMS (Oracle, MS-SQL, DB2, Sybase etc.). Skytide bietet dieses Produkt als *Software as a Service* (**Saas**) an, d.h. das Produkt wird bei Skytide selbst betrieben.

Technische Plattform beider Produkte ist die *Skytide Analytical Platform* (eXtensible OLAP (XOLAP)), siehe [Sky451]. Sie dient zum Analysieren semistrukturierter Daten (Logfiles, HTML, XML, HTTP), aber auch strukturierter relationaler. Ein Data Warehouse wird nicht benötigt. Es wird auf die originären Datenquellen zugegriffen. Daten werden über alle

Datenquellen hinweg durch eine In-Memory-Datenbank in Echtzeit analysiert (aggregiert) und anhand des XML-Analysemodells zueinander in Verbindung gesetzt. Ad-hoc-Anfragen sind aber über ein GUI genauso möglich. Es existiert weiterhin eine JDBC-Schnittstelle und eine API zum Einbinden in eigene Applikationen. Das Speichern der analysierten Daten zum Zwecke der Historisierung wird über ein integriertes **coDBMS** realisiert, dabei werden nur die für die Analyse bedeutenden Daten gesichert. Tabelle 29führt die Ansätze von Skytide XOLAP auf.

Nr.	Betrachtungsschwerpunkt	Ansatz
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)	-
2	*Algorithmen* (DB-Operationen, Kompression)	-
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)	Hauptspeicherdatenbank, Zugriff direkt auf Datemquellen. Auswertung semistrukturierter Daten.
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, frühe oder späte Materialisierung, Sonstiges)	-
5	*Verteilungsschema* (Parallelisierung, Partitionierung)	-
6	*Transaktionsverwaltung* (Update-Problematik)	-
7	*Betriebssystem*	Software as a Service

Tabelle 29: Ansätze von Skytide XOLAP

3.2 Nicht berücksichtigte Implementierungen

Nachfolgend wird auf die Implementierungen von coDBMS eingegangen, die nicht bei der Klassifizierung berücksichtigt wurden. Es wird pro Implementierung jeweils eine kurze Begründung für die Nichtberücksichtigung gegeben.

3.2.1 S und GNU R Programmiersprache

S und GNU R sind Programmiersprachen für statistische und graphische Anwendungen. S ist 1976 an den Bell Laboraties entstanden, siehe [SWiki]. Mittlerweile liegt Version 4 der Programmiersprache vor.

GNU R ist S nachempfunden und wurde 1993 veröffentlicht. Die aktuelle Version ist 2.9.2, siehe [WikiR]. In [WikiCODBMS] werden beide Programmiersprachen den **coDBMS** zugeordnet, da sie spaltenorientierte Datenstrukturen für die statistische Analyse einsetzen. Gemäß [RData] bietet GNU R die Möglichkeit, Matrizen zeilen- oder spaltenorientiert in Textdateien zu schreiben.

Ziel dieser Arbeit ist es, **coDBMS** im Sinne von Kapitel 2.1 zu klassifizieren und die Unterschiede zu dem bekannten Ansatz **roDBMS** darzustellen. S und GNU R sind keine **DBMS** im Sinne dieser Definition. Die einzige Gemeinsamkeit mit coDBMS ist die physische Speicherung einer bestimmten Datenstruktur der Sprache in einer spaltenorientierten Form, was sich jedoch prinzipiell mit vielen anderen Programmiersprachen auch realisieren lässt. Im Fokus dieser Arbeit stehen Implementierungen, die die Eigenschaften eines DBMS zumindest überwiegend erfüllen. S und R sind keine fertigen Systeme zur Verwaltung großer Datenmengen mit den bekannten Zugriffsstrukturen zur Performancesteigerung des Zugriffs und externen Sichten auf das konzeptuelle Datenschema, sondern Programmiersprachen, mit denen sich ggf. ein solches DBMS implementieren lässt. Aus diesem Grund werden die Programmiersprachen nicht in der Klassifizierung berücksicht.

3.2.2 EaseDB

Nach den recherchierten Dokumenten des Entwicklers von EaseDB handelt es sich um einen Row Store, insofern ist die Information in [ETH1] anscheinend nicht korrekt. Es wurden in der recherchierten Literatur auch keinerlei Hinweise auf einen spaltenbasierten Ansatz gefunden.

3.2.3 FastBit

FastBit ist kein DBMS (im Sinne der Codd'schen Regeln, siehe Kapitel 2.1), sondern eine Toolsammlung zum Importieren, spaltenorientiertem Speichern und Indizieren großer Datenmengen durch Bitmap-Indizes, [Wu09]. SQL wird größtenteils nicht unterstützt. Es ist aber kein Plug-In, sonder eine eigenständige Anwendung zum Verarbeiten von Daten und auch kein Client/Server-System. Es existieren Libraries in C++ und Commandline-Tools zur Bedienung. Da hier nur **DBMS** betrachtet werden sollen, wird FastBit nicht klassifiziert.

3.2.4 DataProbe

Der Hyperlink von [WikiCODBMS] für das **coDBMS** *DataProbe* (Tabelle 5, Seite 27) verweist auf die Homepage der amerikanischen Firma Thomson Reuters, die IT-Lösungen für das Gesundheitswesen anbietet [ThRHome]. Ein Hinweis auf das gesuchte **coDBMS** ließ sich dort jedoch nicht finden. Weitere, aber wenige Informationen wurden nach Recherche mit der Suchmaschine Google gefunden (Suchbegriff: „DataProbe Thomson“), siehe [DPRes]. Es handelt sich bei DataProbe um ein Analyse- und Decision-Support-System (OLAP, siehe Kapitel 2.10.2), basierend auf Microsoft Windows Workstation und Server, für integrierte Datenquellen im Bereich des Gesundheitswesens, z.B. für Krankheitsstatistiken, Geburtenraten etc. Mehr Informationen, insbesondere zur spaltenorientierten Speicherarchitektur, waren nicht verfügbar. Eine weiterführende Recherche (Anfrage an die Firma) war in der verbleibenden Bearbeitungszeit leider nicht mehr möglich. Die recherchierten Informationen waren für eine Klassifizierung nicht ausreichend bzw. unterscheiden sich nicht von anderen Systemen.

3.2.5 Weitere nicht betrachtete Implementierungen

Aufgrund der vorgegebenen Bearbeitungszeit war es leider nicht mehr möglich, die in Tabelle 5 aufgeführten **coDBMS** RC21 Commercial Open Source Project, SAP BI Accelerator, Super STAR, EXA Solution, Sensage 4, Metakit Open Source und Xplain Semantic Database.

Bei ParAccel Analytic Database und Sybase IQ handelt es sich genau wie bei Vertica um ein verteilte, parallele und spaltenorientierte Data Warehouses mit relationalem Datenmodell, die Datenkompression einsetzen. Jedoch waren dies die einzigen nennenswerten Informationen, die gewonnen werden konnten. Auf eine Klassifizierung wurde deshalb verzichtet.

3.3 Gesamtübersicht über die Klassifizierungen

Nachfolgend werden zu jedem der in Kapitel 3 genannten Betrachtungsschwerpunkte die ermittelten Ansätze der in Kapitel 3.1 betrachteten Implementierungen genannt. Die Ansätze sind also die Kriterien, die die Betrachtungsschwerpunkte in verschiedene Untergruppen unterteilen bzw. klassifizieren. In diese Untergruppen werden die Implementierungen dann eingeordnet. Tabelle 30 fasst die für die einzelnen Implementierungen ermittelten Ansätze/Klassifizierungskritierien zusammen. Aus der Übersicht geht hervor, dass die meisten Implementierungen das relationale Datenmodell und SQL verwenden, lediglich die Schlüssel-Wert-Speicher Bigtable (und weitere) machen hier eine Ausnahme und haben auch ein anderes Anwendungsgebiet als **OLAP**. Auch verfolgen coDBMS wie Vertica und

Infobright den Ansatz, dass der Anwender für den Einsatzzweck von ad hoc formulierten OLAP-Anfragen selbst keine Optimierungen am **DBMS** vornehmen muss. Bezüglich der Komprimierung und der Anfrageausführung nutzt nur Vertica die späte Materialisierung. Nicht überraschend ist, dass die meisten **coDBMS**, jedenfalls die für OLAP konzipierten, Bitstrings zur Anfrageverarbeitung einsetzen.

Nr.	Betrachtungs-schwerpunkt	Ansatz (Klassifizierungskriterium)	coDBMS-Implementierungen
1	*Zugriffsstrukturen* (Dateiorganisation, Indizes)		
a)	Zeilen- vs. Spaltenorientierung	Reiner Columnstore	Infobright, Vertica, LuciDB, kdb+
		Kombinierter Column- und Rowstore	Bigtable und weitere (Derselbe Zeilenschlüssel sortiert jede Spaltenfamilie)
b)	Grad der Unterstützung für Ad-hoc-OLAP-Anfragen[31]	Indizes für Anwender transparent (d.h. er muss sich nicht darum kümmern)	Infobright (automatisierte Vorberechnungen für Daten), Bigtable und andere (für dünnbesetzte Indizes), C-Store/ Vertica, kdb+ (Live-Indizierung)
		Anwender muss gezielt Indizes erzeugen	Bigtable und andere (Bloomfilter für Beschleunigung der Spaltensuche), LucidDB
		coDBMS verzichtet auf Indizes	Infobright (bedingt[32])
c)	Dateiorganisation	Sortierte Ablage der Attributwerte, ggf. mit dünnbesetztem Index	Bigtable, Hbase, Hypertable, Cassandra, C-Store/Vertica (sortierte Projektion)
		Unsortierte Ablage der Attributwerte	Infobright, LucidDB (Sortierung nach Tupel-ID)
2	*Algorithmen* (DB-Operationen, Kompression)		
a)	Wahl des Kompressionsverfahrens	Individuelle Datenkompression je nach Datentyp des Attributs	Infobright, C-Store/Vertica
		Einheitliches Kompressionsverfahren	Bigtable und weitere (LZO), LucidDB
		Keine Kompression	kdb+
3	*Anwendungen* (Anwendungsszenarien, Schnittstellen, Datenmodelle)		
a)	a) Data Warehouse oder nicht	Integriertes Data Warehouse für (Ad-hoc-)OLAP	Infobright, C-Store/Vertica, LucidDB

[31]Gilt nur für coDBMS, die für OLAP konzipiert wurden

[32]Infobrights Wissensknoten verwenden Bitmap-Index-ähnliche Strukturen ein, sind aber lt. Infobrigt keine Indizes (weil nicht für einzelne Datensätzen, sondern Datenpakete erzeugt)

Nr.	Betrachtungs-schwerpunkt	Ansatz (Klassifizierungskriterium)	coDBMS-Implementierungen
		OLAP-Analysewerkzeug für heterogene Datenquellen	Kdb+, Skytide XOLAP
b)	Verwendetes Datenmodell	Relationales Datenmodell	Infobright, C-Store/Vertica, LucidDB
		Objektrelationales Datenmodell	Valentina
		Multidimensionales Datenmodell	Bigtable und weitere[33]
		Semistrukturiertes Datenmodell	Skytide XOLAP
c)	Anfragesprache, DML und DDL	SQL-Dialekt (SQL-92, -99, -2003)	Infobright, Valentina, C-Store/ Vertica, LucidDB, kdb+
		Proprietäre Anfragesprache	Bigtable und weitere, kdb+
d)	Anwendungs- und Programmierschnittstellen	Standard-Programmierschnittstellen (JDBC, ODBC)	Infobright, C-Store/Vertica, LucidDB, kdb+
		Ausschließlich proprietäres Programmier-API oder Kommandozeilenumgebung	Bigtable und weitere
4	*Anfrageverarbeitung* (spaltenorientierter Optimierer mit Kompression, Sonstiges)		
a)	Optimierung	Algebraische oder kostenbasierte.	Infobright, C-Store/Vertica, LucidDB
		Optimierung nur bei vorhandenen Zugriffsstrukturen.	Bigtable und andere (Bloomfilter)
b)	Zeitpunkt der Materialisierung	Frühe Materialisierung. Optimierer arbeitet ansonsten zeilenorientiert.	LucidDB
		Späte Materialisierung	C-Store/Vertica
c)	Durchführung von Selektionen und Verbunden mit Hilfsstrukturen	Query Processor arbeitet direkt auf Attributwerten	Infobright, C-Store/Vertica
		Query Processor nutzt Hilfsstrukturen	Infobright[34], LucidDB (StarJoin mit Bitmap)
d)	Query Processor nutzt Kompression	Query Processor kann auf komprimierten Daten arbeiten	Infobright, C-Store/Vertica, LucidDB (Filter)
		Query Processor arbeitet auf unkomprimierten Daten	Infobright, Bigtable und andere
5	*Verteilungsschema* (Parallelisierung, Partitionierung)		
a)	Grad der Dekomposition	Vollständige Dekomposition (jedes Attribut separat)	Infobright

[33]Das Datenmodell von Bigtable beinhaltet aber keine Fremdschlüsselbeziehungen, demzufolge keine referenzielle Integrität, und keine Datentypen (alle Daten sind Bytearrays)

[34]Infobright trifft Vorauswahl der zu verwendenden Attribute anhand von Metadaten

Nr.	Betrachtungs-schwerpunkt	Ansatz (Klassifizierungskriterium)	coDBMS-Implementierungen
		Teil-Dekomposition	Bigtable und weitere[35], Infobright, C-Store/Vertica, LucidDB (Cluster)
b)	Art der Partitionierung	Horizontale und vertikale Partitionierung	Infobright, Bigtable und weitere, Infobright, C-Store/Vertica
		Nur vertikale Partitionierung	LucidDB (?)
c)	Art des DDBMS	Shared-Memory	Infobright
		Shared-Nothing	Bigtable und weitere, Infobright, C-Store/Vertica, kdb+
		Shared-Disk	
d)	Server-DBMS oder lokales DBMS	Server-DBMS	Infobright, LucidDB
		Embedded-DBMS	Valentina
6	*Transaktionsverwaltung* (Update-Problematik)		
a)	Sperrung von Datenbankobjekten	Lesender Zugriff trotz Schreib-Transaktion (Snapshot Isolation)	Infobright, C-Store/Vertica, LucidDB
		Sperrung für andere Transaktionen bei Schreibzugriff	Bigtable und weitere
7	*Betriebssystem*		
		Windows	Infobright, Valentina, LucidDB
		Unix/Linux	Infobright, Bigtable, Hypertable, Cassandra (?)[36],, C-Store/Vertica, LucidDB
		MacOS	Hypertable
		Plattformunabhängig	Hbase

Tabelle 30: Klassifizierung der Ansätze für coDBMS

[35]Teilweise Dekomposition in Bigtable und weiteren durch Zusammenfassung mehrerer Spaltenfamilien in AccessGroups möglich

[36]Es ließ sich nicht genau ermitteln, ob Cassandra Linux voraussetzt oder (weil es Java voraussetzt) plattformunabhängig ist, die Administrationsanleitungen beziehen sich auf Linux-Shellskripte

4 Fazit und Ausblick

Im abschließenden Kapitel der Diplomarbeit werden die Motivation der Diplomarbeit, die Zielstellung, das Vorgehen und die gewonnenen Erkenntnisse zusammengefasst und eine abschließende Bewertung durchgeführt. Daran anschließend wird ein Ausblick gegeben, wie man die Erkenntnisse für weitere Betrachtungen, die über den Rahmen der Diplomarbeit hinausgehen, verwenden kann.

In Kapitel Fehler: Referenz nicht gefunden wurden zunächst die Motivation der Diplomarbeit begründet und die Ziele erläutert. Ziel der Diplomarbeit war die Klassifizierung von Ansätzen für **coDBMS** und ein Vergleich mit **roDBMS**. Die Motivation für die Klassifizierung war darin begründet, dass bei weltweit wachsenden Datenbeständen, die in einer Organisation mehrere Terabytes oder sogar Petabytes umfassen, und der Anforderung an eine möglichst ad hoc erfolgende Analyse das Problem gelöst werden muss, in einer großen Datenmenge auf die benötigten Daten möglichst effizient lesend zuzugreifen und sie zu analysieren. Denn trotz der Fortschritte in der Entwicklung der Speichertechnologie sind der Sekundärspeicher und Datenbankpuffer immer noch die Flaschenhälse eines **DBMS**. Insbesondere bei **OLAP** werden eher spaltenbasierte Auswertungen ganzer Attribute auf großen Datenmengen durchgeführt, bei denen die bekannten zeilenorientierten Ansätze hohe I/O-Kosten verursachen und den Datenbankpuffer ineffizient nutzen.

In Kapitel 2 wurden dann die Grundlagen von **roDBMS**, die Grundlagen von **coDBMS** sowie die Unterschiede sowie Vor- und Nachteile von beiden Ansätzen aufgeführt.

Es wurde deutlich gemacht, dass ein spalten- oder zeilenorientierter Ansatz aufgrund der physischen Datenunabhängigkeit grundsätzlich austauschbar ist und keinen Einfluss auf die konzeptuelle und externe Sicht eines **DBMS** haben und noch nicht einmal auf die Art der Anfrageverarbeitung haben müssen. Auch wurden die recherchierten vorhandenen Implementierungen von **coDBMS** vorgestellt.

Der spaltenorientierte Ansatz sorgt für eine andere physische Anordnung der Daten im Sekundärspeicher mit dem Ziel, die Datendichte in den auf physischer Ebene angeforderten Daten zu erhöhen, um so I/O-Kosten und Platz im Datenbankpuffer einzusparen und möglichst viele Daten mit einem sequenziellen Zugriff auf den Sekundärspeicher einzulesen. Als weiteres Mittel zur Einsparung von I/O-Kosten und Pufferplatz wurde die

Datenkompression vorgestellt, die von vielen **coDBMS** (ggf. spaltenindividuell) implementiert ist. Hier wurden ein Vergleich mit Ansätzen bekannter **roDBMS** durchgeführt und Unterschiede deutlich gemacht.

Außerdem wurden **coDBMS** mit bekannten Optimierungstechniken für **roDBMS** (Clustering, Materialisierte Sichten) verglichen, dabei wurden die Vorteile von **coDBMS** für **OLAP** gegenüber diesen Ansätzen herausgearbeitet. **CoDBMS** sind ein Ansatz, der Ad-hoc-Anfragen für **OLAP** ohne die Nutzung solcher Hilfsstrukturen ermöglichen soll. Durch die spaltenorientierte Speicherung ist die Lokalität der Daten, die zu effizientem I/O und effizienter Puffernutzung führt, für den genannten Anwendungszweck automatisch gegeben.

Außerdem wurde gezeigt, wie ein *Query Processor* in einem **coDBMS** aufgrund der physischen Datenunabhängigkeit sowohl zeilen- als auch spaltenorientiert und auf komprimierten Daten arbeiten kann.
Bei der Erläuterung der *vertikalen Dekomposition* von Daten wurde jedoch auch verdeutlicht, dass eine spaltenorientierte Architektur letztendlich auch mit einem herkömmlichen roDBMS zu realisieren ist.
In Kapitel 3 wurde dann anhand der Betrachtung konkreter Implementierungen eine Klassifizierung der Ansätze von **coDBMS** vorgenommen. Hierbei wurde sich auf die Schwerpunkte konzentriert, die sich nach eingehender Literaturrecherche und Beschäftigung mit verschiedenen Implementierungen als charakteristisch für den spaltenorientierten Ansatz erwiesen haben. Die Qualität der erhältlichen Literatur schwankte zwischen den Implementierungen sehr stark. Insbesondere die kommerziellen Hersteller (bis auf Infobright und Vertica) liefern wenig bis gar keine Informationen, die sich auf den spaltenorientierten Ansatz beziehen (Funktionsweise des Optimierers, wie wird die Spaltenorientierung umgesetztm welche Dateiorganisation und sonstige Optimierungsansätze gibt es, welches Kompressionsverfahren wird eingesetzt). Sie wurden deshalb bei der Klassifizierung nicht berücksichtigt.
Bei der Klassifizierung stellte sich heraus, dass die untersuchten Implementierungen eine Bandbreite bei den Betrachtungsschwerpunkten lieferten, die eine Klassifizierung der Ansätze in verschiedene Gruppen ermöglichte. Die meisten coDBMS sind entweder als Data Warehouse konzipierte relationale **DBMS** oder Schlüssel-Wert-Speicher. Insbesondere beim Optimierer werden verschiedene Ansätze verfolgt. Die Schlüssel-Wert-Speicher wie Google haben überhaupt keine ausgeklügelte Optimierung auf Basis eines Kostenmodells, Infobright nutzt automatisiert angelegte Materialisierungen zur Optimierung, und C-Store , Vertica und

LucidDB setzen Bitmaps zur effizienten Anfrageverarbeitung ein. Wobei aber anzumerken ist, dass C-Store und LucidDB hinsichtlich des Zeitpunkts der Materialisierung (also des Konstruierens der Tupel) unterschiedliche Ansätze haben, denn Lucid setzt grundsätzliche frühe Materialisierung ein. Fast jedes relationale coDBMS setzt Bitmaps zur Steigerung der Anfrageperformance ein.
Neben Ad-hoc-**OLAP** tat sich sogar ein weiteres Einsatzgebiet von **coDBMS** auf, nämlich das Speichern und Auswerten Webcrawling-Daten im Rahmen der Webindexierung erkannt.

4.1 Ausblick

Durch die Erkenntnisse dieser Diplomarbeit haben sich Erkenntnisse ergeben, bei denen eine weitere Betrachtung von Interesse wäre.
Auf Basis der Erkenntnisse dieser Arbeit kann man Anwendungsszenarien für **OLAP**, **OLTP** und auch das Webcrawling entwickeln und mit diesen Performancevergleiche zwischen den genannten Implementierungen und z.B. einem vertikal partitionieren **roDBMS** und einem normal zeilenorientierten **roDBMS** durchführen unter Anwendung verschiedener Optimierungsansätze wie z.B. Zugriffspfaden. Hierfür könnten die bekannten Benchmarks *TPC-D* (für **OLTP**-Anwendungen) und *TPC-H* (für **OLAP**-Anwendungen) herangezogen werden.
Auch die Entwicklung eines Kostenmodells für einen Optimieres, welches abschätzt, wann Daten spaltenweise und wann zeilenweise einzulesen sind und wie die Daten dann mit welchen Methoden am effizientesten verarbeitet werden, abhängig von der gestellten Anfrage, ist eine weitere Möglichkeit, wie die Erkenntnisse dieser Arbeit weiterverwendet werden können.

Anhang

A Ergänzende Grundlagen

In diesem Kapitel sollen Aspekte der Grundlagen in Kapitel 2 ausführlicher beschrieben werden, die in Hinblick auf die Klassifizierung lediglich erwähnt, aber nicht tiefergehend betrachtet werden müssen.

A.1 Seiten, Sätze, Adressierung

A.1.1 Datensatztypen

Nachfolgend werden Teilaspekte der Datensatzadressierung von roDBMS aus Gründen der Vollständigkeit beschrieben, auch wenn bei der Klassifizierung der coDBMS nicht betrachtet wurden. Die folgenden Unterkapitel sind Ergänzungen zu Kapitel 2.5.1.

A.1.1.1 Pinned Records (fixierte Sätze)

Pinned Records sind an ihre Position gebunden, wenn auf sie mit einem Zeiger verwiesen wird. Bei Verschieben des Datensatzes in eine andere Seite müssten alle Zeiger darauf gesucht und geändert werden, da sonst die Gefahr leerer Zeiger (engl. *Dangling Pointers*) besteht. *Pinned Records* sollten deshalb vermieden werden

A.1.1.2 Unpinned Records (unfixierte Sätze)

Auf unfixierte Sätze wird von anderer Stelle aus mit logischen Zeigern verwiesen. Der logische Zeiger könnte z.B. ein Primärschlüssel einer Relation. Dieser Primärschlüssel wird an zentraler Stelle (z.B. Indexdatei) auf derzeit aktuelle Adresse umgesetzt. Wenn der Datensatz verschoben wird, muss nur an dieser einen Stelle die Adresse geändert werden. Unfixierte Sätze können also beliebig verschoben werden, ohne dass mehrere darauf verweisenden Datensätze ihre Zieladresse ändern müssen. Nachteil ist, dass neben der Quell- und Zielseite auch noch die Index-Seite geladen werden muss. Behoben wird dieser Nachteil durch das Prinzip des Tupel-Identifikators (**TID**).

A.1.1.3 Lange Felder oder große unstrukturierte Sätze

Nachfolgend werden in Ergänzung zu Kapitel 2.5.1 die beiden Datentypen für große unstrukturierte Daten (Texte, Bilder, Videos) beschrieben. Da Unterschiede zwischen **roDBMS** und **coDBMS** in der Behandlung solcher Daten in der recherchierten Literatur nicht beschrieben waren, blieb die Thematik bei der Klassifizierung unberücksichtigt. Es wird jedoch darauf hingewiesen, dass zumindest ein Anbieter für diese Art von Daten ein Zusatzmodul für sein coDBMS anbietet (vgl. [SybaseIQLOB]). Es gibt zwei Datentypen für unstrukturierte Daten:

1. *Binary Large Objcts (**BLOB**)* nehmen beliebige Byte-Folgen auf (Bilder, Audio, Video) und
2. *Character Large Objects (**CLOB**)* nehmen eine Folge von ASCII-Zeichen auf.

Da die Daten in der Regel die Größe einer Seite überschreiten, müssen sie auf mehrere Seiten verteilt werden. Auf der Originalseite werden nur die Nicht-**BLOB**-Informationen und einer der beiden möglichen Werte aufgenommen:

1. Ein Zeiger auf eine *Seiten-/Blockliste*, die das **BLOB** aufnimmt, wird als Attributwert aufgenommen. Nachteil ist der nicht mögliche wahlfreiem Zugriff, z.B. auf einen bestimmten Teil eines Videos (siehe Abbildung 40) oder ein
2. ***BLOB**-Directory* als Attributwert: Statt des Zeigers wird Directory gespeichert, welches die BLOB-Größe und mehrere Zeiger aufnimmt, die auf die einzelnen Seiten / Blöcke der Platte verweisen (siehe Abbildung 41). Vorteil ist ein schneller Zugriff auf einen BLOB-Teilbereich, Nachteil ist eine begrenzte Maximalgröße des **BLOB**, da bei einem 1 GB-BLOB, einer 8-Byte-Adressierung und einer Seitengröße von 1KB 8MB für das BLOB-Directory anfallen, das deshalb wiederum als BLOB gespeichert werden muss. Auf die in [SHS05] erwähnte effiziente Speicherung von **BLOB** in Baumstrukturen wird in dieser Arbeit nicht weiter eingegangen

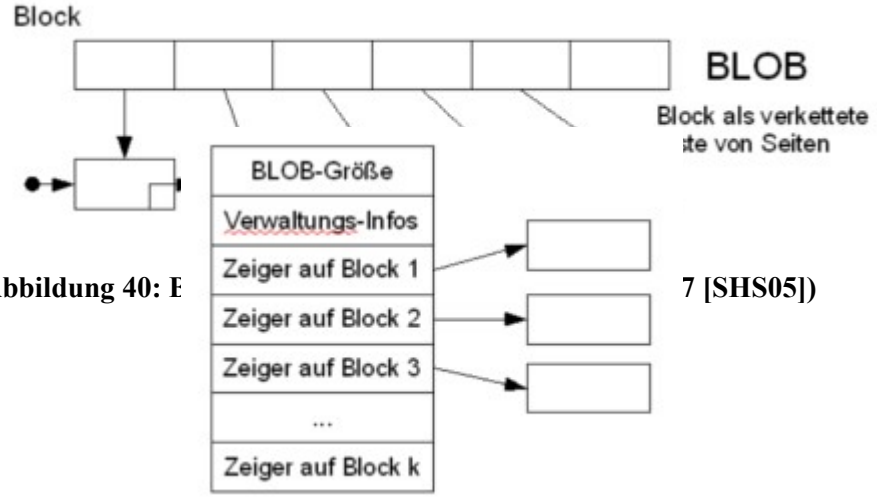

Abbildung 40: E **7 [SHS05])**

Abbildung 41: BLOB-Directory (Seite 109 [SHS05])

A.2 Adressierung von Datensätzen mittels Tupel-Identifier (TID)

Datensätze können mit Angabe der Seitennummer (bestehend aus Speichermedium, Zylinder, Spur und Blocknummer) und des Offsets (auf der Seite) adressiert werden, vgl. Kapitel 2.5 dieser Arbeit oder [Seite 108 [SHS05]]. Das Problem dabei ist, dass die Datensätze so an ihre Position in der Seite, auf der sie aktuell gespeichert sind, gebunden sind (*pinned record*). Wenn ein Datensatz auf eine andere Position, z.B. auf die hinter dem letzten Satz, verschoben werden muss (z.B. wegen geänderter Größe), müssten alle Zeiger auf diesen Datensatz (aufwändig) geändert werden. Lösung für dieses Problem ist das *Tupel-Identifikator-Konzept* (**TID**-Konzept**).** Synonyme für den Begriff „**TID**" sind (im Englischen) *Record Identifier* (**RID**) oder *Row-ID*.

Ein **TID** ist eine Datensatz-Adresse, bestehend aus Seitennummer und einem Offset. Dieser Offset *i* verweist aber nicht auf den Datensatz selbst, sondern auf eine Position *i* in einer Liste von Tupel-Zeigern, die am Anfang der Seite stehen. Jeder Tupel-Zeiger enthält den Offset des gesuchten Datensatzes zum Seitenanfang (siehe Abbildung 42).Vorteil des **TID**-Konzepts ist, dass Datensätze auf einer Seite ohne großen Aufwand und ohne die Gefahr von *dangling pointers* (vgl. Anhang A.1.1.1) verschoben werden können, da nur der Offset-Wert im Tupel-Zeiger am Seitenanfang geändert werden muss. Verweise von außen bleiben unverändert.

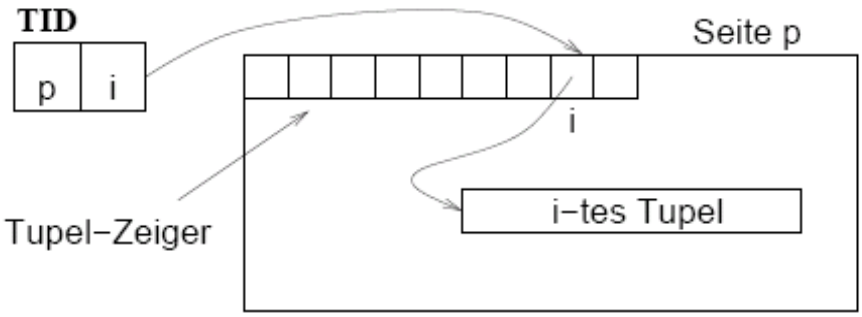

Abbildung 42: Einstufiger Tupel-Identifikator

Wenn ein Datensatz auf andere Seite verschoben wird (z.B. wegen Änderung seiner Größe), dann wird an die Position des alten Datensartzes einfach neuer **TID**-Zeiger aufgenommen, der auf den Tupel-Zeiger auf der neuen Seite verweist. Der Tupel-Identifikator wird aber auch bei mehrmaligem Verschieben eines Datensatzes auf andere Seiten maximal zweistufig, da bei Verschieben auf eine dritte Seite nur der Zeiger auf der ersten Seite geändert werden muss

Eine zweistufige Referenz kostet jedoch Performance (Laden von zwei Seiten notwendig), daher sollte eine regelmäßige Reorganisation erfolgen.

A.3 18 Codd'sche Regeln für Anforderungen an OLAP-Produkte

Nachfolgend aufgeführt sind die Regeln von Codd für OLAP-Produkte als Ergänzung zu Kapitel 2.10.2.

1. *Multidimensionale konzeptionelle Sichtweise* auf die Daten: Die konzeptionelle Sicht des Anwenders auf die Daten soll zur Analyse betriebswirtschaftlicher Zusammenhänge multidimensional sein. Kenngrößen wie Verkäufe, Einkäufe und Preise können so aus der Sicht von Dimensionen wie Produkt, Geografie und Zeit betrachtet werden.
2. *Transparenz*: Trennung zwischen Benutzerschnittstelle und Architektur. Aus Anwendersicht muss der Zugriff auf Daten unterschiedlicher Quellen transparent geschehen, es darf keine negative Beeinflussung z.B. von Funktionalität und Performanz des Analysewerkzeugs stattfinden. Das Wissen um die Herkunft der Daten kann in Form von Metadaten vorhanden sein, darf aber keinen Einfluss auf die Homogenität des Datenbestands nehmen.
3. *Zugriffsmöglichkeit*: Bezug der Basisdaten aus unternehmensexternen und unternehmensinternen (operationalen) Datenbeständen.
4. *Gleichbleibende Antwortzeit bei Berichterstellung*: Die Antwortzeit des Systems darf von der Anzahl der Dimensionen und der Menge der gespeicherten Daten nicht beeinflusst werden.
5. *Client/Server-Architektur*: Zur Trennung von Speicherung, Verarbeitung und Darstellung.
6. *Generische Dimensionalität:* Alle Dimensionen sind in Struktur und Funktionalität einheitlich. Keine Dimension (z.B. Zeit) soll vorbelegt sein (Hinweis: Lt. [BG09] ist die Anforderung mittlerweile überholt).
7. *Dynamische Behandlung dünn besetzter Matrizen:* Dynamische Speicherstrukturanpassung (physisches Schema) an gegebene Dimensionalität (vgl. Kapitel 6.2.2 [BG09]).

8. *Mehrbenutzerunterstützung*: Mehrere Benutzer müssen gleichzeitig dieselben Daten analysieren können.
9. *Uneingeschränkte kreuzdimensionale (dimensionsübergreifende) Operationen*: Anwender kann Berechnungen über beliebige Dimensionen hinweg definieren.
10. *Intuitive Datenbearbeitung*: Direkte Navigation über die Sicht der Dimensionen (in der Benutzeroberfläche).
11. *Flexible Berichterstellung*: Ergebnisse sollen im Report frei anordenbar sein.
12. *Unbegrenzte Anzahl von Dimensionen und Klassifikationsebenen*: Beliebig viele Dimensionen mit beliebig vielen Aggregationsstufen (in der Praxis sind zwischen fünf und acht Dimensionen ausreichend)
13. *Datenintegration*: Neben der Datenhaltung in der eigenen multidimensionalen Speicherstruktur sollen OLAP-Werkzeuge einen transparenten Zugriff auf darunterliegende Daten ermöglichen (**HOLAP**-Ansatz, vgl. Kapitel 6.2.5 [BG09]).
14. *Unterstützung verschiedener Analysemodelle*: Unterstützung des kategorischen, des exegetischen, des kontemplativen und des formelbasierten Datenmodells, wobei das
    1. *Kategorische Datenmodell* historische Daten mit aktuellen vergleicht und den aktuellen Zustand beschreibt,
    2. das *exegetische Datenmodell* die Ursachen des Zustands durch Nachvollziehen der Schritte ermittelt,
    3. das *kontemplative Datenmodell* die Ergebnisse für vorgegebene Werte in einer oder mehreren Dimensionen simuliert und
    4. das *formelbasierte Datenmodell* für vorgegebene Anfangs- und Endzustände ermittelt, welche Veränderungen für welche Kenngrößen bezüglich welcher Dimensionen vorgenommen werden müssen, um das angestrebte Ergebnis zu erreichen.
15. *Trennung analyseorientierter und operativer Daten*: Änderungen der Daten im Data Warehouse dürfen die Integrität der operativen Daten nicht berühren.
16. *Trennung der Speicherorte*: OLAP-Datenbestand darf nicht in den produktiven (operativen) DBMS gespeichert werden.

17. *Unterscheidung zwischen Null- und Fehlwerten:* Das OLAP-Werkzeug muss zwischen leeren Feldern und dem nummerischen Inhalt Null bzw. alphanummerischem Inhalt Leerzeichen unterscheiden können (vgl. Kapitel 5.2.3 [BG09])
18. *Behandlung von fehlenden Werten:* Die leeren Feldern sollen speicherplatzoptimiert verwaltet werden.

A.4 Klassifikation von Zugriffsstrukturen von Datensätzen

Unter die Zugriffsstrukturen fallen die *Dateiorganisationsformen* und die *Zugriffspfade* (Indizes). Die folgenden Unterkapitel klassifizieren die Zugriffsstrukturen gemäß Kapitel 4.1 [SHS05].

A.4.1 Primärschlüssel vs. Sekundärschlüssel

Unterscheidungsmerkmal für Zugriffsstrukturen ist z.B. die Art des/der unterstützten Attribute(s), hier Primär- oder Sekundärschlüssel.

Der *Primärschlüssel* bietet Duplikatfreiheit und ist identifizierende Attributmenge für ein Tupel, über ihn werden oft Verbunde (Joins) realisiert. Er kann durch die Zugriffsstrukturen *Primärindex* oder eine (ggf. geclusterte) Dateiorganisationsform unterstützt werden. Es ist ein dünn- oder dichtbesetzter Index möglich (sowohl der Begriff der Clusterung als auch des dünn- oder dichtbesetzten Index wird noch in diesem Kapitel des Anhangs erläutert).

Der *Sekundärschlüssel* ist eine beliebige andere Attributmenge, die durch eine Zugriffsstruktur unterstützt werden soll. Er besitzt meist keine Schlüsseleigenschaft bzw. kein identifizierendes Merkmal und dient der Unterstützung bestimmter Anfragen (z.B. Selektionen über Sekundärschlüssel). Der Sekundärschlüssel wird durch einen nicht geclusterten Sekundärindex (Zugriffspfad) unterstützt. Es ist nur ein dichtbesetzter Index möglich.

A.4.2 Primärindex vs. Sekundärindex

Eine weitere Klassifizierung der Zugriffsstrukturen kann über die Unterscheidung zwischen Primär- und Sekundärindex vorgenommen werden.

Der *Primärindex* ist ein *Zugriffspfad* auf eine interne Relation (Tabelle), der die Dateiorganisationsform der internen Relation ausnutzt (z.B. die Sortierung einer Tabelle). Im Normalfall ist er über dem Primärschlüsselattribut definiert (dann gibt es keine doppelten Attributwerte), könnte aber auch auf einem Sekundärschlüssel definiert werden.

Ein *Sekundärindex* ist jeder weitere Zugriffspfad auf eine interne Relation. Er kann die Dateiorganisationsform nicht mehr ausnutzen (da schon von Primärindex ausgenutzt) und ist anders aufgebaut als der Primärindex. Ei geclusterter oder dünnbesetzter Index ist nicht möglich, weil z.B. Sortierung der Relation nicht ausgenutzt werden kann, da der Sekundärindex nicht über dem Primärschlüssel, sondern über einem nichtsortierten Attribut gebildet wird. Pro Relation gibt es maximal einen Primärindex, es sind aber mehrere Sekundärindizes möglich.

A.4.3 Dateiorganisation vs. Zugriffspfad

Zur Unterscheidung zwischen Dateiorganisation und Zugriffspfad wird auf die Kapitel 2.11.2 und 2.11.3 verwiesen.

A.4.4 Dünnbesetzter vs. Dichtbesetzter Index

Bei einem *dünnbesetzten Index* existiert nicht für jeden Schlüssel (Attributwert) K ein Eintrag in der Indexdatei. Er wird eingesetzt, wenn die interne Relation nach dem Suchschlüssel K sortiert ist. Der Index verweist immer nur auf die Seitenanführer (Seite = Block mit mehreren Records bzw. Tupeln), denn wenn Schlüssel $K_?$ gesucht ist (K_1,K_2 = Seitenanführer von Seite 1 und 2) und $K_1 \leq K_? \leq K_2$, dann muss $K_?$ auf der Seite von $K_1\uparrow$ zu finden sein. Eine *indexsequenzielle Datei* ist eine sortierte Datei, deren Primärindex dünn besetzt sein kann. Ein Sekundärindex kann nicht dünnbesetzt sein, da Sortierreihenfolge auf dem Zugriffsattribut nicht ausgenutzt werden kann. Bei einem *dichtbesetzten Index* existiert für jeden Datensatz der internen Relation ein Eintrag in der Indexdatei, was nicht heißen muss, dass für n Datensätze n Einträge existieren, da der gleiche Suchschlüssel bei mehreren Datensätzen gleichzeitig vorkommen kann. Jeder Suchschlüssel (Attributwert) kommt als Eintrag in der Indexdatei vor. Ein Primärindex kann dichtbesetzt sein, wenn die Dateiorganisation der internen Relation ein (nichtsortierter) Heap ist.

A.4.5 Geclusterte vs. Nicht-geclusterte Indexe

Ein Index ist *geclustert* (Abbildung 43), wenn in gleicher Form sortiert wie die Relation. Dies gilt z.B. für den Primärindex. Jeder dünnbesetzte Index ist geclusterter Index, aber nicht umgekehrt

Ein *nicht geclusterter* Index (Abbildung 44) ist anders organisiert als die Relation, z.B. sind Studentendaten nach Matrikelnummern sortiert, der Sekundärindex aber über dem Attribut „Studienfach“ angelegt. Ein Sekundärindex kann nur dichtbesetzt und nichtgeclustert sein (sogenannte **invertierte Datei**), da er nicht die Sortierung interner Relationen berücksichtigen kann

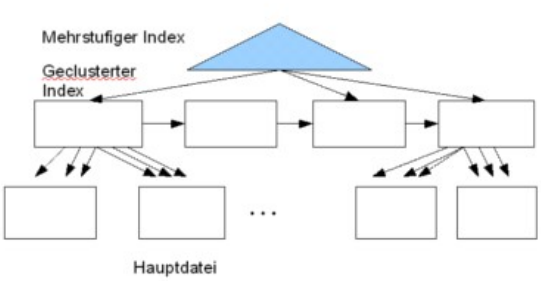

Abbildung 43: Geclusterter Index

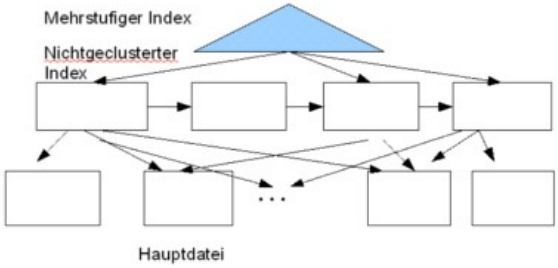

Abbildung 44: Nichtgeclusterter Index

A.4.6 Schlüsselzugriff vs. Schlüsseltransformation

Hier werden die Zugriffsverfahren nach der Umsetzung des Attributwert auf die Tupeladresse unterschieden.

- *Schlüsselzugriff*: Es erfolgt eine Zuordnung von Primär-/Sekundärschlüssel zu den Tupeladressen (z.B. in indexsequentieller Organisation, B-Baum, KdB-Baum)
- *Schlüsseltransformation*: Die Tupeladresse wird mit einer Formel aus dem Primär-/Sekundärschlüsselwert berechnet (z.B. beim Hashverfahren).

A.4.7 Ein-Attribut- vs. Mehr-Attribut-Index

- Ein-Attribut-Index (engl. Non-composite index): Zugriffspfad über einem einzigen Zugriffsattribut
- Mehr-Attribut-Index (composite index): Zugriffspfad über mehreren Attributen

A.4.8 Ein- vs. mehrdimensionale Zugriffsstruktur

Ein Ein-Attribut-Index ist immer eine eindimensionale Zugriffsstruktur. Eindimensional heißt, dass die Zugriffsattributwerte eine lineare Ordnung im eindimensionalen definieren.

Vorname	PLZ	Adresse
Andreas	18209	●
Andreas	18209	●
Antje	18059	●
Christa	69121	●
Gunter	39106	●
Michael	39104	●
Tamara	38106	●

Tabelle 31: Eindimensionaler Index über zwei Attributen

Ein *Mehr-Attribut-Index* ist ein- oder mehrdimensional. Ist er eindimensional, werden die Kombinationen der verschiedenen Zugriffsattributwerte konkateniert (zusammengefügt) und als ein einziger Attributwert betrachtet (siehe Tabelle 31). Ist er mehrdimensional, bilder die Menge der Zugriffsattributwerte keine lineare Ordnung, sondern spannt einen mehrdimensionalen Raum auf. Bei einer Exact-Match-Anfrage wird ein Punkt des Raums als Ergebnis festgelegt, bei einer Partial-Match-Anfrage eine horizontale oder vertikale Gerade im Raum.

A.4.9 Statische vs. Dynamische Zugriffsstruktur

Eine statische Zugriffsstruktur ist nur bei einer bestimmten (fester) Anzahl von verwaltenden Datensätzen optimal. Dynamische Zugriffsstrukturen Verwalten Datensätze unabhängig von ihrer Anzahl optimal.

A.5 Zugriffspfade

A.5.1 Hash-Index

Abbildung 45 illustriert das in Kapitel 2.11 beschriebene Hashverfahren.

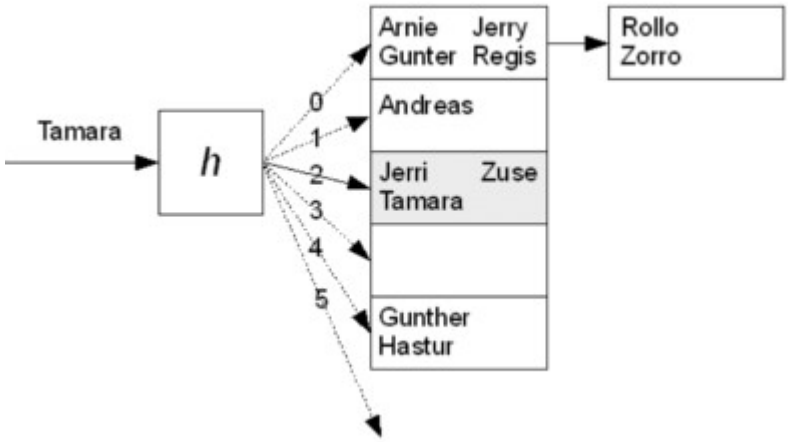

Abbildung 45: Hash-Implementierung [Seite 185 [SHS05]]

A.5.2 Bitmap-Index

Die Tabellen 32 und 33 zeigen die Bitmap-Indizierung für die Attribute position und branchNo. Bei der nachfolgenden Beispielanfrage kann durch AND-Verknüpfung der Bitmap-Indizes effizient das dritte Tupel ermittelt werden, welches beide Prädikate erfüllt:

```
SELECT staffNo,salary
FROM Staff
WHERE position='Supervisor' AND branchNo='B003'
```

staffNo	fName	lName	position	sex	DOB	salary	branch No
SL21	John	White	Manager	M	01.10.45	30.000	B005
SG37	Ann	Beech	Assistant	F	10.11.60	12.000	B003
SG14	David	Ford	Supervisor	M	24.03.58	18.000	B003
SA9	Mary	Howe	Assistant	F	19.2.70	9.000	B007
SG5	Susan	Brand	Manager	F	3.6.40	24.000	B003
SL41	Julie	Lee	Assistant	F	13.6.65	9.000	B005

Tabelle 32: Beispielrelation für Bitmap-Indizierung [Seite 1284 [CB05]]

Manager	Assistant	Supervisor
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0
0	1	0

B003	B005	B007
0	1	0
1	0	0
1	0	0
0	0	1
1	0	0
0	1	0

Tabelle 33: Bitmap-Indizes für Beispielrelation aus Tabelle 27 [Seite 1284 [CB05]]

A.5.3 Verbund-Index

Das folgende Beispiel soll den Verbund-Index erklären. Nimmt man das Beispiel der Hausverwaltung (engl. *Property Management*) aus [Seite 321 [CB05]] mit den Relationen *Branch* (Zweigbüros) und *PropertyForRent* (zu vermietende Objekte eines Zweigbüros, Beispiel aus [Seite 1285 [CB05]]), so wird der Verbundindex in Tabelle 36 für die (logischen) Tupel-ID mit identischen Werten gebildet. Der Verbundindex ist in diesem Fall auf dem Attribut branchRowID sortiert, kann aber nach einem beliebigen Attribut sortiert sein

rowID	branchNo	city
20001	B005	London
20002	B007	Aberdeen
20003	B003	Glasgow
20004	B004	Bristol
20005	B002	London

Tabelle 34: Relation 1 für Verbundindex [Seite 1285 [CB05]]

rowID	propertyNo	city
30001	PA14	Aberdeen
30002	PL94	London
30003	PG4	Glasgow
30004	PG36	Glasgow
30005	PG21	Glasgow
30006	PG16	Glasgow

Tabelle 35: Relation 2 für Verbundindex [Seite 1285 [CB05]]

branchRowID	propertyRowID	city
20001	30002	London
20002	30001	Aberdeen
20003	30003	Glasgow
20003	30004	Glasgow
20003	30005	Glasgow
20003	30006	Glasgow
20005	30002	London
20006	...	

Tabelle 36: Verbundindex für Tabellen 29 und 30

A.6 Kompressionstechniken in DBMS

Dieses Kapitel enthält Ergänzungen zu Kapitel 2.12.

A.6.1 Klassifizierung von Kompressionstechniken

Ziel von Datenkompression ist, Daten mit geringem Informationsgehalt zu verdichten bzw. die benötigte Informationsmenge zu minimieren, z.B. zwecks Einsparung von Übertragungsbandbreite in Netzwerken oder von Speicherplatz, vgl. u.a. [Seite 371 [Sed92]]. Die Informationen werden nach erfolgter Kompression in kürzerer Form dargestellt. Man unterscheidet dabei zwischen *verlustfreier* und *verlustbehafteter* Kompression, vgl. u.a. [WikiKompr]. Von einer verlustfreien Kompression spricht man, wenn die kodierten Daten nach Dekodierung exakt denen des Originals entsprechen. Bei verlustbehafteter Kompression ist dies nicht der Fall, hier werden Informationsverluste in Kauf genommen, wenn der Grad des Informationsverlustes für den jeweiligen Anwendungszweck akzeptabel ist. Insbesondere bei Bild-, Video- und Audiokompression (sogenannte Multimediadaten, die heutzutage selbstverständlicher Bestandteil von Webseiten sind) finden verlustbehaftete Kompressionsverfahren aufgrund der dort anfallenden großen Datenmengen Anwendung.
Gemäß [WikiKompr] spricht man bei verlustfreier und verlustbehafteter Kompression auch von *Redundanzreduktion* (weil bei verlustfreier Kompression redundante Informationen entfernt werden) und *Irrelevanzreduktion* (weil bei verlustbehafteter Kompression irrelevante Informationen eingespart werden). Verlustfreie Kompression ist eine bijektive Abbildung zwischen den originären Daten D und den komprimierten Daten D'.

Entscheidend ist auch die Zeit, die ein Kompressionsverfahren für Kodierung und Dekodierung benötigt. Ist die Zeit für beide Vorgänge in etwa gleich, spricht man von *symmetrischer* Kompression, ansonsten von *asymmetrischer* (vgl. [WikiKompr]).

A.6.2 Lauflängenkodierung

Tabelle 37 zeigt einen Auszug aus [Seite 373 [Sed92]], um den Ansatz zu verdeutlichen. Dieses Beispiel macht deutlich, dass Lauflängenkodierung nur effizient arbeitet, wenn lange Ketten sich wiederholender Zeichen vorhanden sind, da nur dann die kodierte Information weniger Speicherplatz verbraucht als die unkodierte.

00000000000000000000000000011111111111111000000000	28 14 9
00000000000000000000000000111111111111111110000000	26 18 7
…	…
01111111111111111111111111111111111111111111111111	1 50

Tabelle 37: Bitraster mit Lauflängenkodierung (Abbildung 22.1 [Sed92])

A.6.3 Huffman-Kodierung

Nachfolgend werden einige Beispiele für die Huffman-Kodierung (Kapitel 2.12.2.1) aufgeführt.

Nimmt man z.B. die Buchstaben des Alphabets als zulässige Symbole, so kann man diese 26 Symbole durch 5 Bits $(2^5 = 32)$ kodieren. Die Zeichenkette „ABRACADABRA“ könnte also mit 11*5 = 55 Bits kodiert werden, siehe [Seite 370 ff. [Sed92]]. Dabei würde jedoch der nur einmal auftretende Buchstabe „D“ genausoviel Bits reservieren wie der fünfmal vorkommende Buchstabe „A“. Ziel von Huffman ist es aber, für die häufigsten Zeichen die kürzeste Kodierung und für die am wenigsten auftretenden die längste zu verwenden.

Eine mögliche Kodierung für die Zeichenkette „ABRACADBRA“ nach diesem Prinzip ist in Tabelle 38 dargestellt.

Buchstabe	Anzahl	Gewichtung	Kodierung
A	5	0,46	0
B	2	0,18	1
R	2	0,18	01
C	1	0,09	10
D	1	0,09	11

Tabelle 38: Beispiel-Kodierung der Symbole für "ABRACADABRA"

Das Problem bei der Kodierung in Tabelle 38 ist die Nichteindeutigkeit. So kann die Kodierung „01“ entweder für den Buchstaben „R“ oder das Wort „AB“ stehen. Die Huffman-Kodierung muss also eine eindeutige Darstellung aller Symbole gewährleisten. Dies wird gemäß erreicht durch einen sogenannten Präfix-Code bzw. präfixfreie Kodierung, d.h. die Darstellung (Kodierung) eines Symbols darf niemals Präfix einer anderen Kodierung sein (vgl. u.a. [HuffWiki]). Dies wird erreicht durch Aufbau eines Binärbaums, siehe Abbildung 46 für das Beispiel aus Tabelle 38.

Der Baum wird dabei von rechts nach links aufgebaut, beginnend mit den beiden Symbolen mit der geringsten Gewichtung. Dies erhalten einen neuen Elternknoten, der die Summe der beiden Gewichte von Blatt C und D bildet. Der Algorithmus wird solange (mit aufsteigenden Gewichtungen) fortgesetzt, bis der Wurzelknoten erreicht ist. Nun wird der Baum vom Wurzelknoten aus traversiert, dabei werden anhand der (binären) Kantenbeschriftungen die Huffman-Kodierungen erzeugt. Tabelle 39 zeigt die Huffman-Kodierungen für das Beispiel aus Abbildung 46.

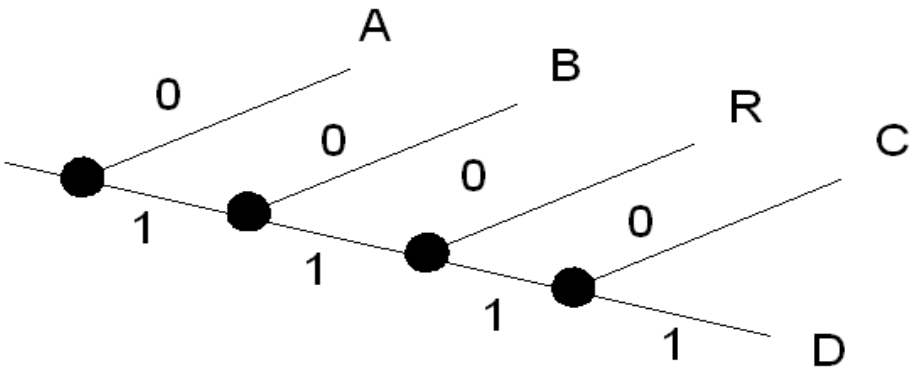

Abbildung 46: Huffman-Kodierung für Beispiel aus Tabelle 7

Buchstabe	Huffman-Kodierung
A	0
B	10
R	110
C	1110
D	1111

Tabelle 39: Huffman-Kodierung für Wort "ABRACADABRA"

A.6.4 Entropie

Aus der Häufigkeit des Auftretens der Symbole in den Eingabedaten kann man auch die Entropie der Eingabedaten berechnen. Sie ist ein Maß dafür, wie viel redundante Informationen in den Eingabedaten stecken bzw. ein Maß für den mittleren Informationsgehalt einer Nachricht. Häufig auftretende Zeichen haben einen geringeren Informationsgehalt als selten auftretende. Die Entropie gibt an, wieviel Bits notwendig sind, um ein Zeichen aus dem Alphabet identifizieren (und damit kodieren) zu können (vgl. auch [Seite 16 [Say06]]). Shannon zeigte, dass das Optimum eines verlustfreien Komprimierung die Kodierung mit einer durchschnittlichen Bitrate ist, die der Entropie entspricht.

Berechnet wird die Entropie mit $H=-\sum_{i=1}^{n} p_i * \log_2(p_i)$, wobei p_i die Häufigkeit des Auftretens des i-ten Zeichens des Alphabets in den Eingabedaten ist. Die durchschnittliche Bitrate ergibt sich dadurch, dass die Anzahl von Kodierbits, die für ein Symbol mit einer bestimmten Wahrscheinlichkeit notwendig sind $(\log_2(p_i))$ [37], mit der Wahrscheinlichkeit ihres Auftretens insgesamt (also im Eingabedatenstrom) multipliziert wird $(p_i * \log_2(p_i))$.

A.6.5 Arithmetische Kodierung

Ergänzend zu Kapitel 2.12.2.2 wird nachfolgend die arithmetische Kodierung anhand eines Beispiels beschrieben und die Schwäche der Huffman-Kodierung aufgezeigt.

Die Huffman-Kodierung erreicht die Entropie nicht immer, wie das [Seite 82 [Say06]] zeigt: Bei einem Alphabet $A=\{a_1, a_2, a_3\}$ mit der Verteilung P der Symbole $P(a_1)=0{,}95$, $P(a_2)=0{,}02$ und $P(a_3)=0{,}03$ ist die Entropie (benötigte Bits zur Kodierung) nach der oben genannten Formel 0,335 bits/Symbol. Ein Huffman-Code gemäß Kapitel 2.12.2.1 würde $a_1=0, a_2=11, a_3=10$ lauten. Damit werden bei der gegebenen Verteilungswahrscheinlichkeit für die Huffman-Kodierung einer Zeichenkette von 100 Zeichen 105 Bits benötigt, es sind also 1,05 bits/Symbol notwendig. 0,715 bits/Symbol sind redundante Information. Durch das Zusammenfügen von Buchstaben zu erweiterten Symbolen (z.B. $a_1 a_2$) und der damit verbundenen Erweiterung des Alphabets kann die Entropie auch bei Huffman-Kodierung gesenkt werden, jedoch wächst dadurch die Codetabelle an. Um für eine Sequenz der Länge m die Huffman-Kodierung zu finden, benötigt man m^n Codes (m=Anzahl der Einzelsymbole, n=Länge des zusammengesetzten Codeworts) für alle möglichen Sequenzen der Länge m, was die Kodierungstabelle stark wachsen lässt (vgl. Kapitel 3.2.4 [Say06]).

[37] Nach Shannon sind für selten auftretende Informationen mehr Bits notwendig

Bei einem Alphabet $A=(a_1,a_2,a_3)$ mit der Wahrscheinlichkeit $P(a_1)=0,8$, $P(a_2)=0,02$ und $P(a_3)=0,18$ beträgt die Entropie nach der oben genannten Formel 0,816 bits/Symbol. Bei einem Huffman-Code $a_1=0, a_2=11, a_3=10$ ist die benötigte Kodierlänge 1,2 bits/Symbol. Setzt man 2-stellige Codewörter zusammen, ergibt sich eine Codetabelle mit einem erweiterten Alphabet von 3^2 Symbolen, siehe Tabelle 40. Anzumerken ist hierbei., dass bei Erzeugung der Codetabelle für zweistellige Symbole von der gleichen Wahrscheinlichkeit des Auftretens (in denselben Eingabedaten) wie bei den Einzelsymbolen ausgegegangen wird.

Symbol	Wahrscheinlichkeit	Code
$a_1 a_1$	0,64[38]	0
$a_1 a_2$	0,016	10101
$a_1 a_3$	0,144	11
$a_2 a_1$	0,016	101000
$a_2 a_2$	0,0004	10100101
$a_2 a_3$	0,0036	1010011
$a_3 a_1$	0,1440	100
$a_3 a_2$	0,0036	10100100
$a_3 a_3$	0,0324	1011

Tabelle 40: Tabelle 3.11 aus [Say06]

Die Kodierlänge ist für diese Codetabelle 1,7228 bits/Symbol, wobei man zum Vergleich mit der ursprünglichen, einsymboligen Codetabelle diesen Wert durch zwei teilen muss, d.h. Die durchschnittliche Kodierlänge ist 0,8614 bits/Symbol, was nur 5,5% über der (optimalen) Entropie liegt, insofern ist in diesem Beispiel die Abweichung nicht sehr groß.

Wählt man den eben gezeigten Ansatz für das erste Beispiel mit $P(a_1)=0,95$, $P(a_2)=0,02$ und $P(a_3)=0,03$, so erhält man die Codetabelle in Tabelle 41.

[38] 0,8 * 0,8

Symbol	Wahrscheinlichkeit	Code
$a_1 a_1$	0,9025[39]	0
$a_1 a_2$	0,0190	111
$a_1 a_3$	0,0285	100
$a_2 a_1$	0,0190	1101
$a_2 a_2$	0,0004	110011
$a_2 a_3$	0,0006	110001
$a_3 a_1$	0,0285	101
$a_3 a_2$	0,0006	110010
$a_3 a_3$	0,0009	110000

Tabelle 41: Tabelle 4.2 aus [Say06]

Die durchschnittliche Bitrate ist hier 1.222 bits/Symbol bzw. durch zwei geteilt 0,611 bits/Symbol, was immer noch 72% über der Entropie liegt. Gemäß Kapitel 4.2 [Say06] entspricht die Bitrate erst bei einer Symbollänge von 8 Zeichen annähernd der Entropie, jedoch ist dann dafür eine Codetabelle mit 3^8 Einträgen notwendig, d.h. diese steigt exponentiell an.

Die Arithmetische Kodierung versucht, sich der Entropie auf andere Art anzunähern. Es wird iterativ eine reelle Zahl als Kodierung der gesamten Symbolfolge erzeugt (bei n Symbolen des Alphabets). Dazu wird ein Bereich zwischen 0 und 1 zunächst in Wahrscheinlichkeitsintervalle für die vorkommenden Symbole eingeteilt. Die Kodierung eines Symbols ist eine beliebige rationale Zahl, die zum korrespondierenden Intervall gehört. Anhand der in den Eingabedaten vorkommenden Symbole wird der Bereich (Intervall) für das Ergebnis (Fließkommazahl) immer weiter eingeschränkt. Das Intervall richtet sich dabei nach dem ersten auftretendem Symbol.

Nimmt man als Beispiel die Zeichenkette *baaaaaacaa* mit den Wahrscheinlichkeiten a=0,8 b=0,1 und c=0,1, so ergibt sich nach der obigen Formel eine Entropie von $-((0{,}8*-0{,}322)-2*(0{,}1*-3{,}322))=-(-0{,}2576-2*0{,}3322)=0{,}922\ Bits/Symbol$. Bei Huffman-Kodierung mit den oben genannten Codes für a, b und c wäre die Kodierungsrate 1,2 Bits/Symbol.

Die Abbildung 47 zeigt am genannten Beispiel die arithmetische Kodierung mit Intervallschachtelung (angelehnt an [WikiAK]). Die Zeichenkette kann durch eine reelle Zahl

[39] 0,95*0,95

zwischen 0,82359296 und 0,825690112 dargestellt werden, z.B. 0,824. Reelle Zahlen werden in einem Rechner als Fließkommazahl behandelt. Eine Fließkommazahl $fp = a * r^e$ besteht aus der Mantisse a, der Basis r und dem Exponen e, vgl. Kapitel 4.8.1 [Cle00]. Für die binäre Darstellung der Kodierung ist hier die Mantisse (also die Nachkommastellen) ausschlaggebend, die Kodierlänge für die Zeichenkette „baaaaaacaa“ ist also $\log_2(824) = 9{,}687$ Bits, das bedeutet die Abweichung von der Entropie ist hier mit $0{,}9687 - 0{,}922 = 0{,}0467$ Bits/Symbol niedriger als bei Huffman-Kodierung. Da ein Bit nicht teilbar ist, müssen natürlich im konkreten Fall statt 9,687 aufgerundet 10 Bits für die Kodierung verwendet werden, was gegenüber 12 Bits eine Ersparnis von aufgerundet 17% bedeutet.

Der Algorithmus für die Kodierung ist in Tabelle 42 dargestellt (übernommen aus [Bo09])

1. Lege [0,1] als aktuelles Intervall fest.
2. Teile das aktuelle Intervall in n Teilintervalle, wobei das i-te Teilintervall die p_i - fache Länge des aktuellen Intervalls hat.
3. Lese das nächste Zeichen von der zu kodierenden Zeichenkette ein und lege das hierzu korrespondierende Teilintervall als aktuelles Intervall fest.
4. Wenn noch Zeichen einzulesen, gehe zu Schritt 2.
5. Gib eine möglichst kurze Zahl x aus dem aktuellen Intervall aus.

Tabelle 42: Algorithmus der Arithmetischen Kodierung (aus [Bo09])

Abbildung 48 zeigt die Dekodierung der reellen Zahl 0,824 zum Wort „baaaaaacaa“. Dabei wird immer das Intervall des Buchstabens zum neuen Intervall, in dem der Wert 0,824 jeweils liegt.Tabelle 43 zeigt den Algorithmus für die Dekodierung, ebenfalls aus [Bo09].

1. Lege [0,1] als aktuelles Intervall fest.
2. Teile das aktuelle Intervall in n Teilintervalle, wobei das i-te Teilintervall die p_i -fache Länge des aktuellen Intervalls hat.
3. Gebe das Zeichen aus, in dessen korrespondierendem Teilintervall die kodierte Zahl x liegt. Lege dieses Teilintervall als aktuelles Intervall fest.
4. Wenn noch Zeichen einzulesen sind, gehe zu Schritt 2.

Tabelle 43: Algorithmus der Arithmetischen Dekodierung (aus [Bo09])

Ein Problem bei der Implementierung ist, dass die Mantisse mit der Menge der zu kodierenden Daten anwächst. Die maximale Genauigkeit (die durch die Mantisse festgelegt wird) von (Fließkomma-)Zahlen ist im allgemeinen aber fest vorgegeben (z.B. 32 Bits) und kann diesen Wert nicht überschreiten (vgl. [WikiAK]) . Gelöst wird dies durch regelmäßige Vergrößerung des Intervalls. Höherwertige Stellen, die feststehen, werden ausgegeben und aus den Zahlen entfernt. Im Beispiel von oben kann man also nach dem Kodieren des ersten a sicher sagen, dass die Ergebniszahl mit 0,8 beginnt, da das Intervall zwischen 8 und 8,8 (0,8) niemals durch eine weitere Multiplikation der Werte $(0{,}8 * 0{,}8)$ bis zum Endergebnis maximal um 0,64 eingeengt werden kann (da hier Zahlen mit einem Wert < 1 multipliziert werden) bzw. sich der rechte Rand des Intervalls um maximal 0,64 verkleinert. Man kann also bereits hier 0,8 ausgeben und von den Intervallgrenzen abziehen. Danach wird die Intervallgrenze mit 10 skaliert, und es wird mit diesem Wert weitergerechnet.

Abschließend wird kurz dargestellt, warum die arithmetische Kodierung optimal im Sinne der Entropie ist (vgl. [WikiAK]): Die Größe des Intervalls ist direkt verbunden mit der Anzahl der Nachkommastellen (bzw. Bits), die notwendig sind, um eine Zahl aus diesem Intervall darzustellen (dem so genannten Informationsgehalt). Je kleiner das Intervall (bzw. je mehr Nachkommastellen es hat), umso mehr Bits sind notwendig.

Ein Beispiel mit dezimalen Zahlen. In einem Intervall der Größe 10^{-3} lässt sich immer eine Zahl mit 3 Stellen nach dem Komma finden. Zwischen 0,11123 und 0,11223 (Intervall: 0,001) liegt beispielsweise 0,112. Ist die Größe des Intervalls *I*, so ist die Anzahl der notwendigen Dezimalstellen zur Darstellung einer (beliebigen) Zahl aus dem Intervall $-\log_{10} I$ (in diesem Fall also $-\log_{10} 0{,}001 = 3$). Um für binäre Zahlen die Anzahl der Bits zu berechnen, wird anstatt des Logarithmus zur Basis 10 der Logarithmus zur Basis 2 verwendet. Die Anzahl der notwendigen Bits ist also $-\log_2 I$. Nach jedem Kodierungsschritt wird das Intervall um einen Faktor kleiner, der der Auftrittswahrscheinlichkeit des eingelesenen Zeichens entspricht, da die Größe der Subintervalle diesen Wahrscheinlichkeiten entspricht.

Die neue Intervallgröße ist also $I' = I \cdot p(Zeichen)$. Die Anzahl der für I' benötigten Bits ist $\log_2 I' = -\log_2(I \cdot p(Zeichen)) = -\log_2 \cdot I - \log_2 \cdot p(Zeichen)$. Es kommen also genau so viele Bits zum Ausgabestrom hinzu, wie die Entropie verlangt. Somit ist die arithmetische Kodierung in Bezug auf den Informationsgehalt optimal.

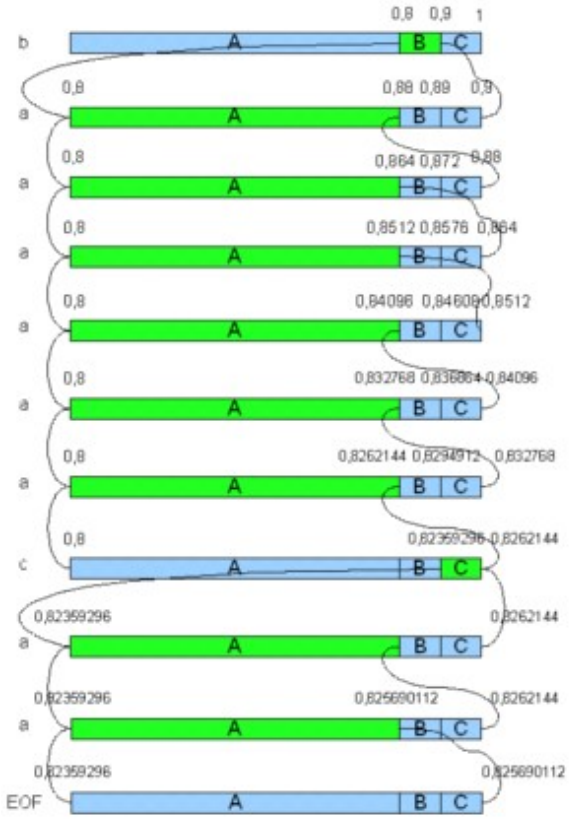

Abbildung 47: Arithmetische Kodierung für Zeichenkette "baaaaaacaa"

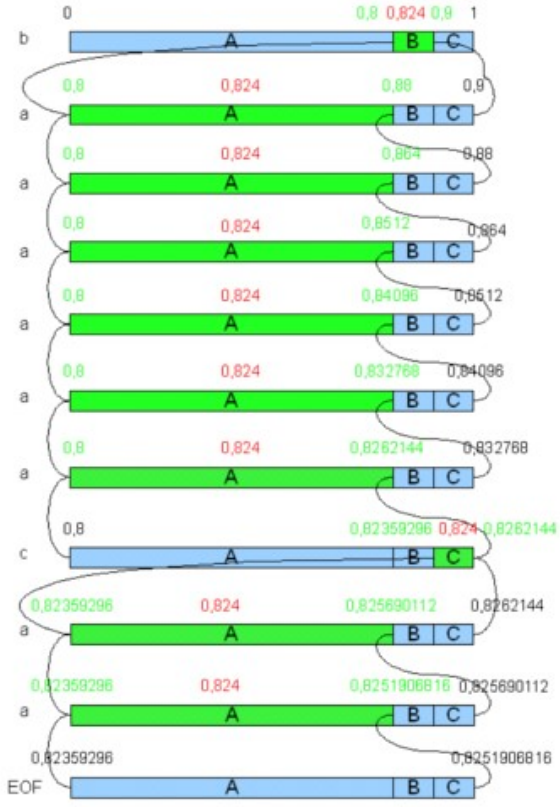

Abbildung 48: Arithmetische Dekodierung für Zeichenkette "baaaaaacaa"

A.6.6 Wörterbuchkompression

A.6.6.1 Übersicht über existierende Lempel-Ziv-Implementierungen

Abbildung 49 und Tabelle 44 geben einen Überblick über die Implementierungen der Lempel-Ziv-Familie.

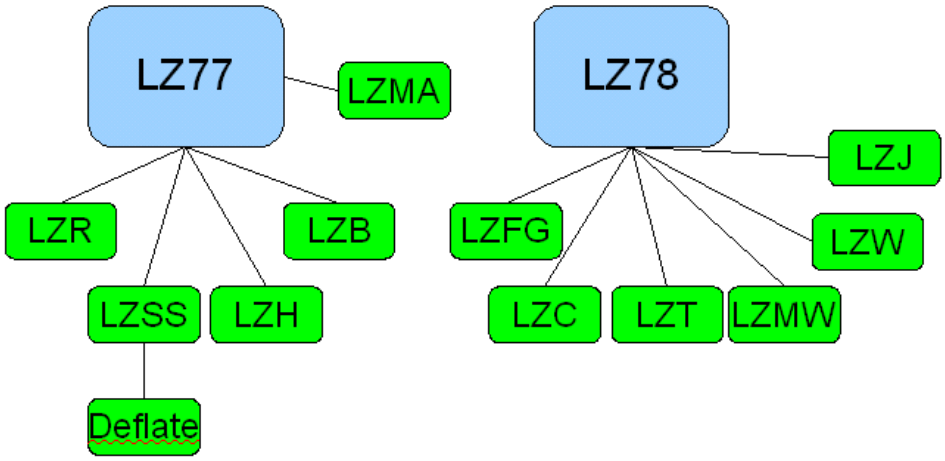

Abbildung 49: Lempel-Ziv-Algorithmenfamilie (aus [LZAlg1])

Abkürzung	Name des Lempel-Ziv-Algorithmus	Abkürzung	Name des Lempel-Ziv-Algorithmus
LZ77	Ursprünglicher Lempel-Ziv-Algorithmus	LZ78	Lempel-Ziv-Algorithmus von 1978
LZR	Lempel-Ziv-Renau	LZFG	Lempel-Ziv-Fiala-Green
LZB	Lempel-Ziv-Bell	LZC	Lempel-Ziv-Compress
LZSS	Lempel-Ziv-Storer-Szymanski (1982)	LZT	Lempel.Ziv-Tamayo
LZH	Lempel-Ziv-Hufman	LZMW	Lempel-Ziv-Miller-Wegmann (1985)
LZMA	Lempel-Ziv-Markov-Algorithmus (1998)	LZW	Lempel-Ziv-Welch (1984)
Deflate	Kombination aus LZSS und Huffman	LZJ	Lempel-Ziv-Jakobsson

Tabelle 44: Übersicht über Lempel-Ziv-Algorithmen (aus [LZAlg])

A.6.6.2 Beispiel für Lempel-Ziv 77

```
While (Vorschaupuffer nicht leer) {
      hole eine Referenz (Position, Länge) zur längsten
Übereinstimmung mit Suchpuffer;
      if (Länge > 0) {
            output (Position, Länge, nächstes Symbol);
            schiebe Fenster um Länge+1;
      } else {
            output (0,0,erstes Symbol im Vorschaupuffer);
            schiebe Fenster 1 Zeichen weiter;
      }
}
```

Tabelle 45: LZ77-Algorithmus (aus Kapitel 3.2 [LZAlg])

Tabelle 45 zeigt den Algorithmus für LZ77. Tabelle 46 zeigt ein aus [Wander05] übernommenes Beispiel der Funktionsweise von LZ77. Es arbeitet mit einem 8 Zeichen großen Suchpuffer und einem 4 Zeichen großen Vorschaupuffer. Tabelle 47 zeigt die Dekompression der Daten. Der Algorithmus geht bei jedem dekomprimiertem Tripel **P** Zeichen im bereits dekomprimiertem Text zurück, hängt dann ausgehend von dieser Position **L** Zeichen an das Wortende (blaue Textfarbe) und ergänzt es um Zeichen **N** (rote Textfarbe).

	Suchpuffer (Wörterbuch)								**Vorschaupuffer**			
	8	7	6	5	4	3	2	1	1	2	3	4
1									A	B	R	A
2								A	B	R	A	K
3							A	B	R	A	K	A
4						A	B	R	A	K	A	D
5				A	B	R	A	K	A	D	A	B
6		A	B	R	A	K	A	D	A	B	R	A
7	A	K	A	D	A	B	R	A	!			

Kodierung[40]		
P	**L**	**N**
0	0	A
0	0	B
0	0	R
3	1	K
2	1	D
7	3	A
0	0	!

Tabelle 46: Beispiel für eine Kompression nach LZ77 (aus [Wander05])

[40]Tripel bestehend aus Position, Länge und nächstem (nicht übereinstimmenden) Zeichen

	P	L	N	Dekodiertes Wort
1	0	0	A	A
2	0	0	B	AB
3	0	0	R	ABR
4	3	1	K	ABR|AK
5	2	1	D	ABRAK|AD
6	7	3	A	ABRAKAD|ABRA
7	0	0	!	ABRAKADABRA|!

Tabelle 47: Dekompression der Daten aus Tabelle 41

A.6.6.3 Beispiel für Lempel-Ziv 78 (LZ78)

Tabelle 48 zeigt den aus [LZAlg] entnommenen Algorithmus für LZ78.

```
w:= NIL; // Leerstring
While (NOT EOF) {
      N:= nächstes Eingabesymbol;
      // Konkatenation (Verlängerung der eingelesenen Zeichenkette)
      if (w+N existiert im Wörterbuch) {
            w:= w+N;
      }
      else {
            schreibe (i,N);
            füge w+N zu Wörterbuch hinzu;
            w:= NIL;
      }
}
```

Tabelle 48: LZ78-Algorithmus (aus [LZAlg])]

Tabelle 49 zeigt ein aus [BELZ78] entnommenes Beispiel für die Funktionsweise des LZ78-Algorithmus. Das Wörterbuch wird zur Laufzeit im Hauptspeicher aufgebaut, aber nicht persistent gesichert. Persistent gesichert werden nur die Tupel *(i,N)*, die die komprimierten Daten repräsentieren.

Schritt												
1	**A**	B	R	A	C	A	D	A	B	R	A	!
2	A	**B**	R	A	C	A	D	A	B	R	A	!
3	A	B	**R**	A	C	A	D	A	B	R	A	!
4	A	B	R	**A**	**C**	A	D	A	B	R	A	!
5	A	B	R	A	C	**A**	**D**	A	B	R	A	!
6	A	B	R	A	C	A	D	**A**	**B**	R	A	!
7	A	B	R	A	C	A	D	A	B	**R**	**A**	!
8	A	B	R	A	C	A	D	A	B	R	A	**!**

i	N
0	A
0	B
0	R
1	C
1	D
1	B
3	A
0	!

Wörterbuch	
i	Wort
1	A
2	B
3	R
4	AC
5	AD
6	AB
7	RA
8	!

Tabelle 49: Beispiel für LZ78-Kodierung (aus [BELZ78])

Bei der Dekodierung (siehe z.B. [DCRC97]) ist das Wörterbuch leer. Es wird beim Dekodieren anhand der Paare (i, N), bestehend aus Indexeinträgen und nicht übereinstimmenden Zeichen, rekonstruiert. Dabei ist bezüglich des Index i folgende Fallunterscheidung zu treffen:

1. $i > 0$: i referenziert einen Wörterbucheintrag: Verbinde Wörterbucheintrag mit N und schreibe das Ergebnis in den Ausgabestrom und ins Wörterbuch.
2. $i \neq 0$: Es wird kein existierender Wörterbucheintrag referenziert auf einen existierenden Eintrag im Wörterbuch. Schreibe N in Ausgabestrom und in Wörterbuch

Tabelle 50 zeigt die Dekodierung der Daten aus Tabelle 49.

i	N
0	A
0	B
0	R
1	C
1	D
1	B
3	A
0	!

Wörterbuch		Ausgabestrom
I	Wort	
1	A	**A**
2	B	A**B**
3	R	AB**R**
4	AC	ABR**AC**
5	AD	ABRAC**AD**
6	AB	ABRACADAB
7	RA	ABRACADABRA
8	!	ABRACADABRA!

Tabelle 50: Dekodierung des Beispiels aus Tabelle 14

A.6.7 Datenkompression bei Oracle DBMS

Mit Version 8i von Oracle wurden indexorganierte Tabellen (**IOT**) und die Schlüsselkomprimierung (engl. *Key Compression*) für Indizes und IOT eingeführt. Eine indexorganisierte Tabelle besteht nur aus einem Index-Segment, das Datensegment einer normalen Tabelle entfällt. Die Datensätze werden physikalisch nach dem Primärschlüssel sortiert. Eine IOT kann mit *Key Compression* komprimiert werden.

Abbildung 50 zeigt zunächst einmal den Unterschied zwischen einer Tabelle mit Index (unterteilt in Index- und Datensegment) und einer indexorganisierten Tabelle (nur aus Indexsegment bestehend). Bei einer „normalen“ Tabelle mit Index enthalten die Blattknoten zu allen Datensätzen je einen Indexeintrag mit der *Tupel-ID* (Kapitel 2.7) als Zeiger. In einer IOT gibt es kein Datensegment, die Daten sind in den Blättern enthalten, sortiert nach dem Primärschlüssel. Ein IOT bietet sich immer dann an, wenn die Spalten des Primärschlüssels einen wesentlichen Anteil am gesamten Datensatz ausmachen (vgl. [OraCompr6]). Wie eine IOT mit der Data Definition Language (**DDL**) angelegt wird, soll hier nicht erläutert werden.

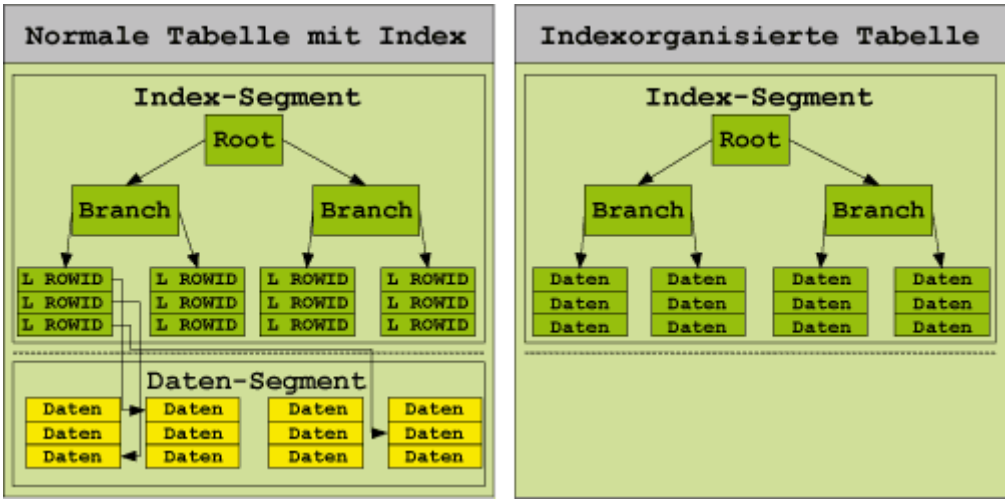

Abbildung 50: Unterschied zwischen Tabelle mit Index und IOT (aus [OraCompr6])

In Abbildung 51 wird die *Key Compression*, die sowohl für Indizes als auch für IOT eingesetzt werden kann, dargestellt. Sie wird ausschließlich in den Blattknoten eingesetzt und beruht auf der Eliminierung von sich wiederholenden Schlüsselwerten in zusammengesetzten Schlüsseln. Zusammengesetzte Schlüssel in einem Index werden in einen Präfix- und Suffix-Teil untergliedert. Der *Suffix* repräsentiert dabei den eindeutigen (sich nicht wiederholenden) Teil des Indexschlüssels (vgl. [M08]), der Präfix den nicht eindeutigen. Wiederholen sich Schlüsselwerte im Präfix-Teil des Index, werden sie nur einmal gespeichert und vom Suffix

referenziert. Es werden *alle* Suffix-Einträge, aber jeweils nur *einmal* jeder *unterschiedliche* Präfix-Eintrag (z.B. „Dortmund“, „Auf dem Toren“ in Abbildung 51) gespeichert.

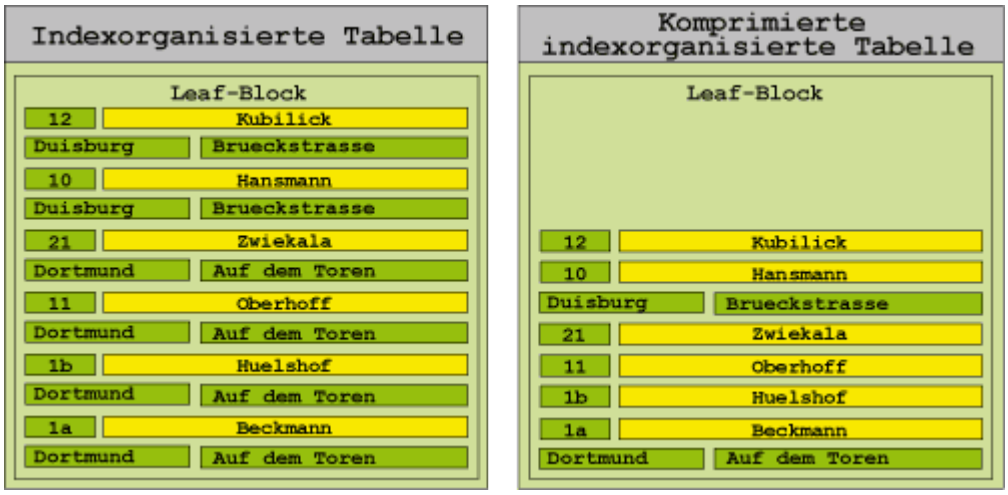

Abbildung 51: Kompression indexorganisierter Tabellen in Oracle 8i (aus [OraCompr6])

Die Index- und IOT-Komprimierung kann für B-Bäume und indexorganisierte Tabellen eingesetzt werden (vgl. [OraCompr5]). Bei Indizes ist die *key compression* bei non-unique Indizes effektiv.

Seit Version 9i Release 2 (veröffentlicht 2001) ist die Tabellenkomprimierung für Bulkload-Operationen in Oracle integriert (vgl. [OraCompr3]). Oracle selbst empfiehlt die Nutzung der Datenkompression für große Data Warehouses (vgl. [OraCompr2]).

Ziele der Tabellenkomprimierung sind lt. Oracle die bereits im vorigen Unterkapitel genannten, nämlich Einsparung von Speicherplatz im Sekundärspeicher und Primärspeicher (Puffer) sowie Performanceverbesserung bei Anfragen, welche große Datenmengen auswerten, sowie beim Backup und Recovery (weniger Speicherplatz für Backups und Recovery Logs). Die Tabellenkomprimierung wird auf Blockebene (bzw. Seitenebene) durchgeführt, vgl. [OraCompr5].

Gemäß [OraCompr2], das sich auf die Version 10g Release 2 bezieht, komprimiert Oracle Daten durch das Entfernen doppelter Werte in einem Block (bzw. einer Seite). Doppelte Werte aller Zeilen und Spalten in diesem Block (und *nur* in diesem) werden einmalig am Anfang des Blocks in einer sogenannten *Symboltabelle* gespeichert, auf die referenziert wird (siehe Abbildung 52).

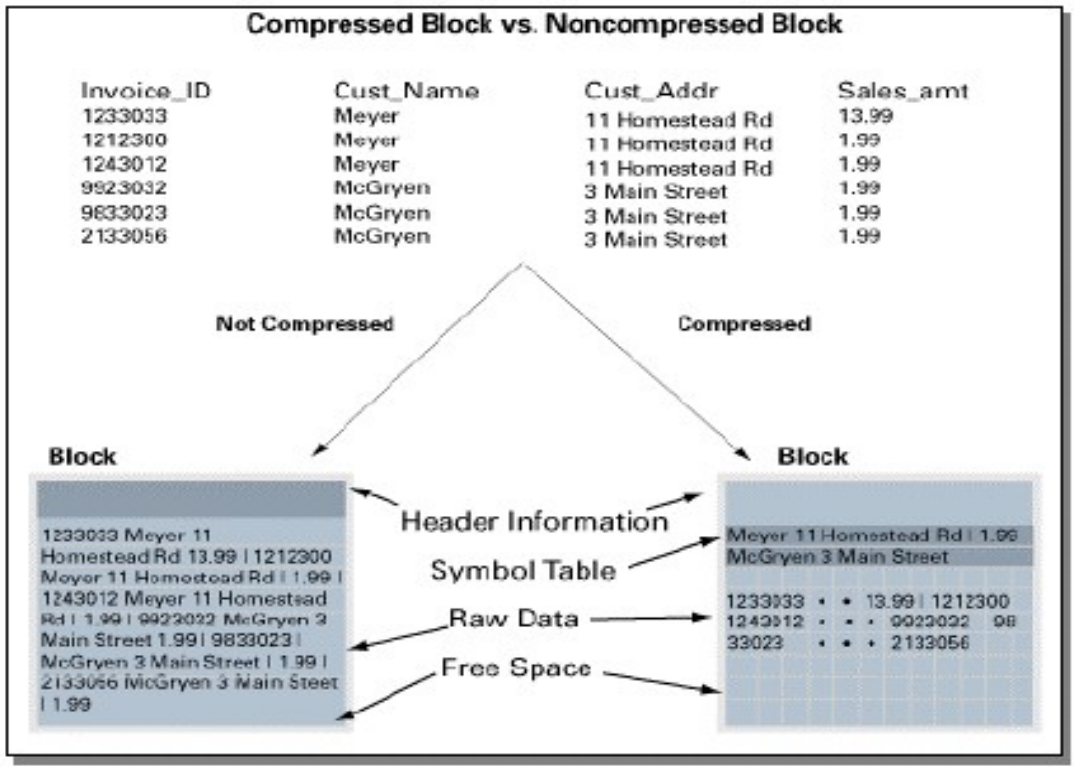

Abbildung 52: Kompression auf Blockebene mit Symboltabelle in Oracle ([OraCompr4])

Die Kompression wird bei Version 9i nur bei sogenannten *Bulk Loads* (*Direct Load* oder `CREATE TABLE … AS SELECT`-Statement (**CTAS**)) ausgeführt. Bestehende Daten werden durch den Befehl `ALTER TABLE … MOVE COMPRESS` in die komprimierte Form überführt, die einen exklusiven Lock auf der Tabelle setzt. Daten, die mit konventioneller Data Manipulation Language (**DML**) geändert werden, werden nicht komprimiert bzw. in Version 9i werden die einzelnen Sätze dann wieder dekomprimiert (vgl. [OraCompr5]).

Komprimiert werden können Datenbankobjekte inklusive Tabellen und Materialized Views. Bei partitionierten[41] Tabellen können einige oder alle Partitionen komprimiert werden. Auch die Komprimierung von Backups ist seit Version 10g mit dem Recovery Manager (RMAN) („physisches“ Backup der Blöcke der Datenbank) und Data Pump (logisches Backup von 1-n Tabellen in eine Datei, vgl. Seite 8 [OraCompr4]) möglich (vgl. u.a. Seite 31ff. [OraCompr3]), basierend auf ZLIB-Kompression. Das Kompressionsattribut (ja/nein) für eine Tabelle / einen Tablespace / eine Partition kann geändert werden, die Änderung wird jedoch nur für die neuen Daten des Datenbankobjekts wirksam., d.h. eine Tabelle kann sowohl

[41]Der Begriff der Partitionierung/Fragmentierung wird im Kapitel 2.13 erläutert

unkomprimierte als auch komprimierte Blöcke enthalten. Sollte die Kompression die Größe eines Blocks steigen lassen, wird sie nicht angewendet.

Bezüglich des Zeitpunkts der Dekompression ist zu sagen, dass Oracle die Blöcke zunächst einlesen kann, ohne sie zu dekomprimieren (vgl. Seite 4 [OraCompr4]).

Mit Oracle 11g wird die Tabellenkomprimierung durch die Einführung der *Advanced Compression (lizenzpflichtig)* erweitert, die Kompression auch für OLTP-Anwendungen ((also auch für konventionelle DML-Operationen wie INSERT, UPDATE und DELETE), vgl.Seite 6 [OraCompr3]) verfügbar macht.

Die Daten eines Blocks werden aber auch dabei im Batchmode (wie beim Bulkload) komprimiert (vgl. Seite 4 [OraCompr4]) und nicht bei jeder Schreiboperation. Ein neuer Block bleibt unkomprimiert, bis ein intern kontrollierter Grenzwert für die Datenmenge (im Block), in diesem Fall die PCTFREE[42]-Grenze (vgl. [OraCompr5]), überschritten wird. Dieses Prozedere wird auch bei einem bereits komprimiertem Block angewendet, zu dem neue (unkomprimierte) Daten hinzugefügt wurden. Durch das Batchprocessing gibt es bei OLTP lt. Aussage von Oracle keine großen Performancenachteile. Das Batchprocessing ist in Abbildung 53 illustriert.

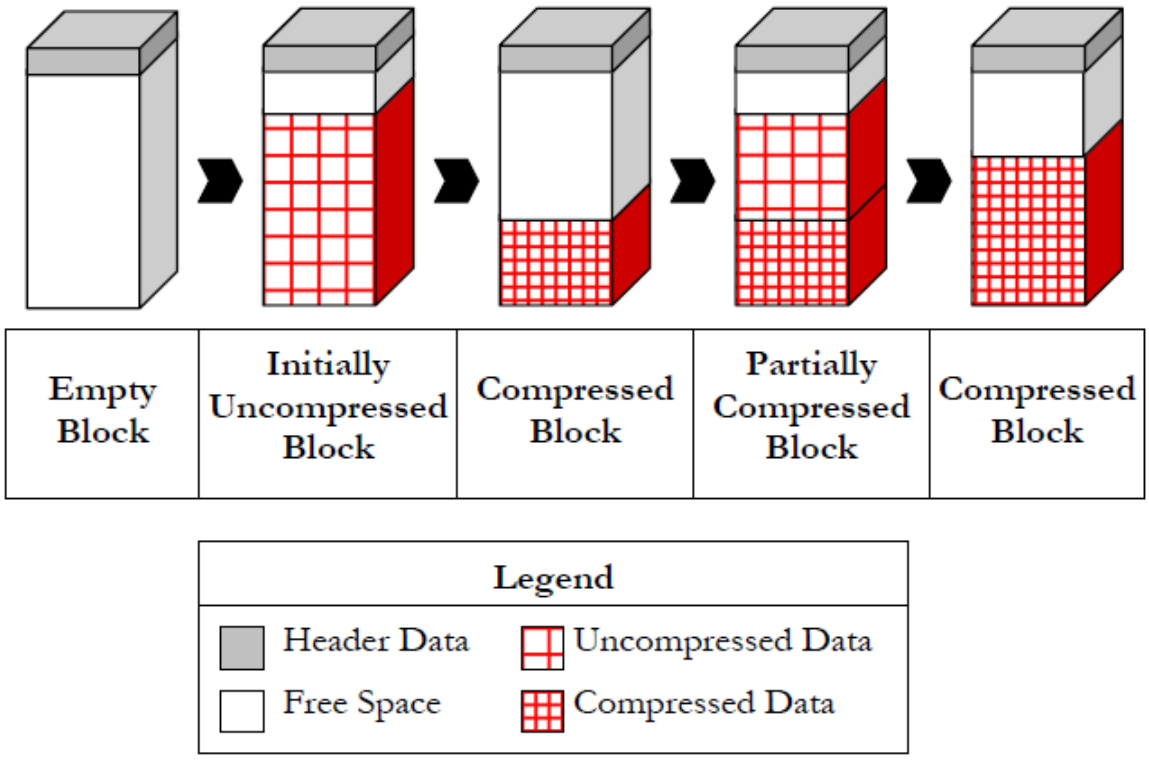

Abbildung 53: Batchprocessing bei OLTP-Blockkompression in Oracle 11g ([OraCompr4])

[42]PCTFREE ist in Oracle ein Parameter für die Verwaltung des Freibereichs eines Blocks ([SHS05])

Bei der von Oracle implementierten Blockkomprimierung mit dem Entfernen doppelter Werte sind sortierte Daten vorteilhaft. Ebenso von Vorteil ist das Verwenden größerer Blöcke, zum einen weil es die Wahrscheinlichkeit doppelter Werte erhöht, zum anderen werden weniger Symboltabellen für die gleiche Menge an Daten erzeugt. Oracle empfiehlt bei Einsatz des DBMS als Data Warehouse darüber hinaus einen niedrigen PCTFREE-Wert (am besten 0), da häufige Updates nicht zu erwarten sind. Ein hoher PCTFREE-Wert kann zu niedrigen Kompressionsraten führen.

Ab Version 11g sind auch Large Objects (**LOB**) mit Hilfe des SecureFiles-Feature (für unstruktierte Daten wie Dokumente, Tabellen und XML-Dateien, vgl. Seite 6 [OraCompr4]) und Seite 25 [OraCompr3]) und abgeleitete Typen (VARRAY) komprimierbar.

Die Tabellen 51 und 52 zeigen die Aktivierung von Bulkload- und OLTP-Kompression in Oracle-SQL-Syntax.

```
CREATE TABLE emp (
      emp_id NUMBER,
      first_name VARCHAR2(128),
      last_name VARCHAR2(128)
) COMPRESS FOR ALL OPERATIONS;
```

Tabelle 51: Oracle SQL-Syntax für OLTP-Kompression ([OraCompr4])

```
CREATE TABLE emp (
      emp_id NUMBER,
      first_name VARCHAR2(128),
      last_name VARCHAR2(128)
) COMPRESS FOR DIRECT_LOAD OPERATIONS;
```

Tabelle 52: Oracle SQL-Syntax für Bulkload-Kompression ([OraCompr4])

Einen Überblick, welche Relationen bereits komprimiert sind, kann man in Oracle per SQL-Statement erhalten (Abbildung 54).

In Planung ist ein Compression Advisor (Seite 16 [OraCompr3]), der Hinweise gibt, wie lohnend die Kompression bestimmter Relationen ist (Abbildung 55).

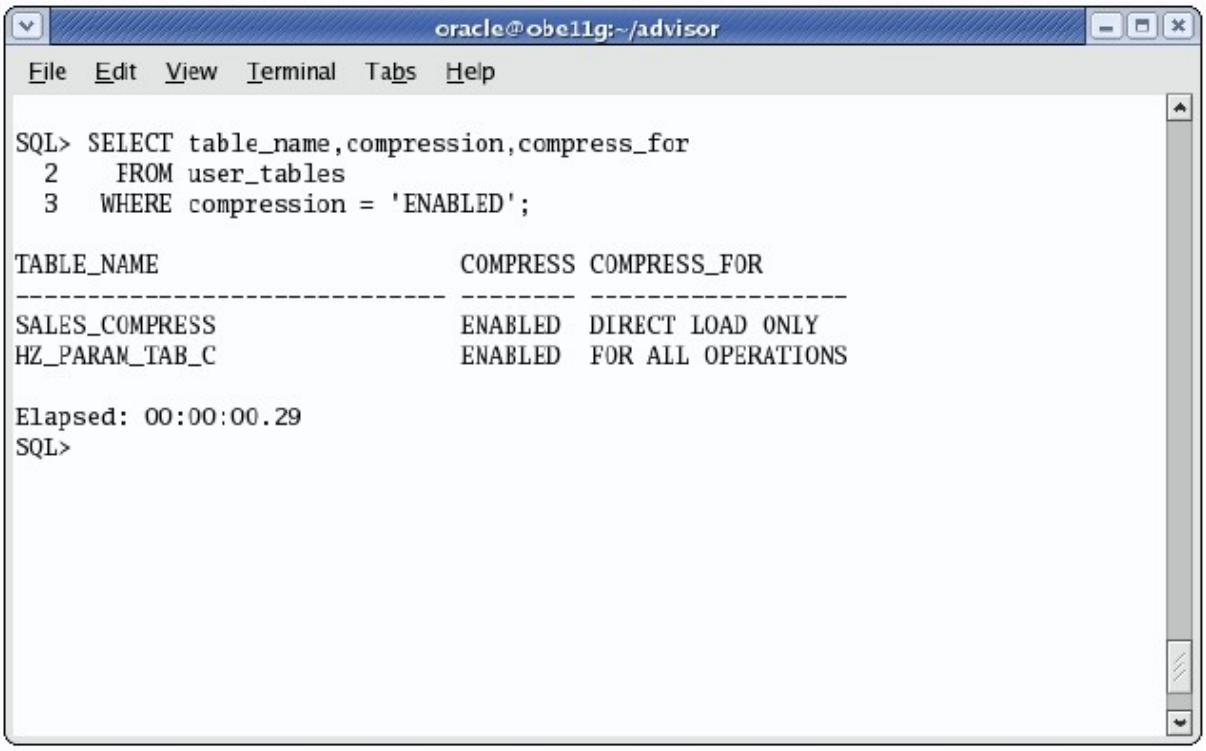

Abbildung 54: Abfrage der komprimierten Tabellen in Oracle 11g (Seite 17 [OraCompr3])

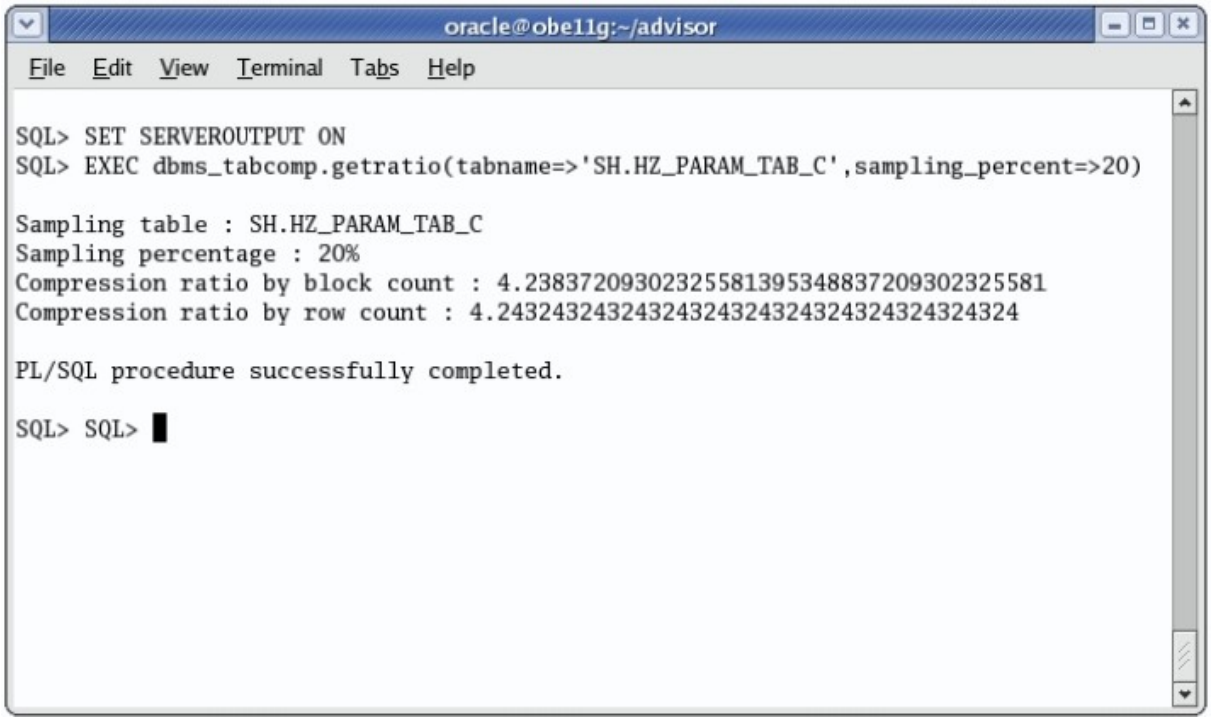

Abbildung 55: Compression Advisor in Oracle 11g (Seite 16 [OraCompr3])

A.6.8 Datenkompression in IBM DB2

Nachfolgend wird ein Beispiel für die Kompression in DB2 gezeigt. Abbildung 56 zeigt ein Beispiel aus [Ah06]mit zwei Tupeln. Nicht nur die sich wiederholenden Werte „500“ werden komprimiert, sondern auch das sich wiederholende zusammenhängende Muster „Plano“, „TX“ und „24355“, das als ein zusammenhängender Wert komprimiert wird.

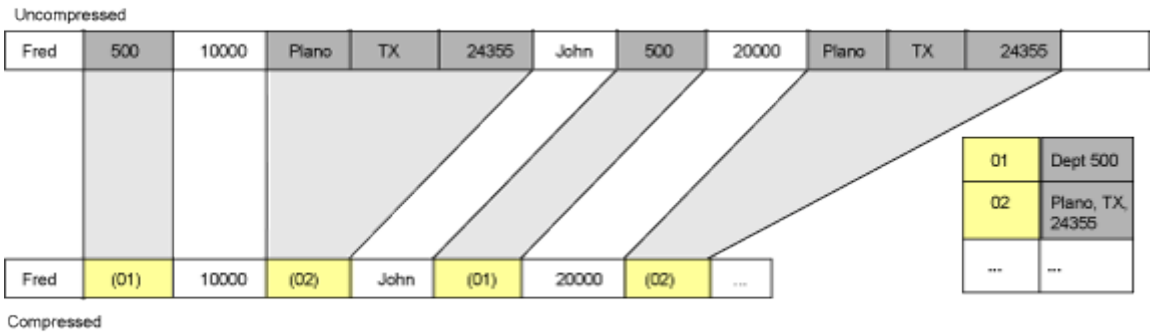

Abbildung 56: Kompression von zwei Tupeln in IBM DB 2 ([Ah06])

Das Wörterbuch zum Komprimieren und Dekomprimieren wird in versteckten Datenbankobjekten gespeichert und im Hauptspeicher gecached. Es wird also nicht wie bei Oracle blockbasiert (Symboltabelle) aufgebaut, sondern bezieht sich auf die gesamte Relation.

Die Kompression wird tabellenweise durch die SQL-Statements `CREATE TABLE name COMPRESS YES` oder `ALTER TABLE name COMPRESS YES` aktiviert. Die Kompression findet dabei beim Aufbau des Wörterbuchs, das ist bei DB2 während der Reorganisation, statt.

Nach Angaben von IBM ist das Wörterbuch selbst bei sehr großen Tabellen nur um die 100 KB groß (vgl. [Ah06]). Mit Hilfe des INSPECT-Tools kann DB2 die zu erwartende Kompressionsrate für eine Tabelle ermitteln. Auch DB2 speichert (wie Oracle) die Daten im Hauptspeicher (Puffer) in komprimierter Form. Anhang

Neben der eben beschriebenen Zeilenkomprimierung bietet DB2 9 außerdem Null- und-Defaultwert-Komprimierung, Backup-Kompression, XML-Tag-Ersetzung (ersetzt XML-Tags durch Integer-Werte) und multidimensionales Clustering (Block-Indizes).

Ebenso wird Partitionierung (auch bekannt als Fragmentierung) anhand eines Schlüsselbereichs ermöglicht und das Komprimieren einzelner Partitionen.

IBM DB2 implementiert Hardware (durch CPU)-und Software-Kompression, die gegenseitig dekodierbar sind (wodurch Load Balancing in Multiprozessor-Umgebungen möglich ist, wenn nur einige Prozessoren Hardwarekompression unterstützen).

A.6.9 Vergleich der Datenkompression in Oracle und DB2

Tabelle 53 (angelehnt an Seite 42 [OraCompr3]) stellt die Komprimierung in Oracle und DB2 zusammenfassend gegenüber.

Oracle 11g	**IBM DB2**
Blockkompression	Tabellen-/Partitionskomprimierung auf Tupelebene
Wörterbuch pro Block (Symboltabelle)	Wörterbuch pro Relation
Kein Zusammenfassen mehrerer Attribute bei Duplikateliminierung	Zusammenfassen mehrerer Attribute bei Duplikateliminierung
Kompression durch Batch-Processing (bei Bulkload oder OLTP)	Aufbau des Wörterbuchs nur bei Reorganisation, ansonsten unklar, ob Kompression bei jeder Änderung / jedem Insert oder auch nur bei Reorganisation
Indexkompression	Keine Indexkompression
LOB-Kompression	Keine LOB-Kompression

Tabelle 53: Vergleich der Kompression von Oracle und IBM DB2 (aus [OraCompr3])

A.6.10 Vergleich von zeilen- und spaltenorientierter Kompression

Anhand eines Beispiels aus [OraCompr6] sollen die unterschiedlichen Ansätze von Kompression (bzw. Kodierung) in roDBMS und coDBMS verdeutlicht werden. Angenommen wird dabei, dass Zahlenwerte mit 8 Bit (1 Byte) und bei Zeichenketten jeder Buchstabe mit 1 Byte kodiert werden. Tabelle 54 zeigt die zu komprimierenden Daten. Insgesamt werden 269 Bytes zur Kodierung benötigt

Tupel Nr.	1 Byte	1 Byte pro Zeichen	1 Byte	1 Byte pro Zeichen	Bytes insgesamt
9	2005	Jan ULLRICH	3	T-MOBILE TEAM	26
8	2005	Ivan Basso	2	TEAM CSC	20
7	2005	Lance ARMSTRONG	1	DISCOVERY CHANNEL TEAM	39
6	2004	Ivan Basso	3	TEAM CSC	20
5	2004	Andreas KLÖDEN	2	T-MOBILE-TEAM	29
4	2004	Lance ARMSTRONG	1	US POSTAL-BERRY FLOOR	38
3	2003	Alexandre VINOKOUROV	3	TEAM TELEKOM	34
2	2003	Jan ULLRICH	2	TEAM BIANCHI	25
1	2003	Lance ARMSTRONG	1	US POSTAL-BERRY FLOOR	38
Summe					269

Tabelle 54: Beispiel für zu komprimierende Daten ([OraCompr6])

Nach dem Ansatz von Oracle wird nun im Block die Symboltabelle (Tabelle 55) aufgebaut, und die Tupel des Blocks referenzieren die Einträge der Symboltabelle. Die Symboltabelle und die Datentabelle mit den Tupeln sind in den Tabelle dargestellt. Bei den Referenzierungen wird ein 4-Bit-Wert angenommen, da die insgesamt 12 Indexnummern mit 4 Bit kodiert werden können. Es werden nur die wirklich benötigten Bits für die Kodierung der Nutzdate als Grundlage verwendet, um einen Vergleich mit einem spaltenorientierten Kodierschemata durchzuführen. In Tabelle 56 werden die kodierten Daten der Relation dargestellt (rote Schrift stellt eine Referenz auf die Symboltabelle dar). Insgesamt werden 1.470 Bits für die Kodierung verwendet.

Index	Wert	Bits
1	2003	4+8 = 12
2	2004	4+8 = 12
3	2005	4+8=12
4	Lance ARMSTRONG	4+120 = 124
5	Jan ULLRICH	4+88=92
6	Ivan Basso	4+80=84
7	1	4+8=12
8	2	4+8=12
9	3	4+8=12
10	US POSTAL-BERRY FLOOR	4+168=172
11	T-MOBILE-TEAM	4+104=108
12	Team CSC	4+64=68
Summe		**710**

Tabelle 55: Symboltabelle für Beispiel aus Tabelle 28

Tupel Nr.					Bits
9	*3*	*5*	*9*	*11*	16
8	*3*	*6*	*8*	*12*	16
7	*3*	*4*	*7*	DISCOVERY CHANNEL TEAM	12+176 = 188
6	*2*	*6*	*9*	*12*	16
5	*2*	Andreas KLÖDEN	*8*	*11*	12+112 = 124
4	*2*	*4*	*7*	*10*	16
3	*1*	Alexandre VINOKOUROV	*9*	TEAM TELEKOM	8+32*8 = 260
2	*1*	*5*	*8*	TEAM BIANCHI	12+96 = 108
1	*1*	*4*	*7*	*10*	16
Summe					**760**

Tabelle 56: Relation kodiert mit Symboltabelle (Oracle-Ansatz)

Für die spaltenorientierte Kodierung wird das Schema der eindeutigen (unique) Werte pro Spalte zur minimalen Kodierung verwendet. Die Kodierungen für Spalte 1 ergeben sich aus Tabelle 57.

Spalte 1	Kodierung	Bits
2005	00	8+2=10
2004	01	8+2=10
2003	11	8+2=10
Summe		**30**

Tabelle 57: Kodierungen für Spalte 1 für Beispiel aus Tabelle 28

Für Spalte 1 sind also insgesamt 9*2+30 = 48 Bits zur Kodierung notwendig.Tabelle 58 zeigt die Kodierungen für Spalte 2.

Spalte 2	Kodierung	Bits
Jan ULLRICH	000	88+3 = 91
Ivan Basso	001	80+3=83
Lance Armstrong	010	120+3=123
Andreas Klöden	011	112+3=115
Alexandre VINOKOUROV	100	160+3=163
Summe		**575**

Tabelle 58: Kodierungen für Spalte 2 für Beispiel aus Tabelle 28

Für Spalte 2 sind 9*3+575 = 602 Bits zur Kodierung notwendig. Die Kodierungen für Spalte 3 ergeben sich aus Tabelle 59.

Spalte 3	Kodierung	Bits
1	00	10
2	01	10
3	11	10
Summe		**30**

Tabelle 59: Kodierungen für Spalte 3 für Beispiel aus Tabelle 28

Für Spalte 3 sind 9*2+30 = 48 Bits zur Kodierung notwendig.Die Kodierungen für Spalte 4 ergeben sich aus Tabelle .

Spalte 4	Kodierung	Bits
TEAM BIANCHI	000	96+3=99
US POSTAL-BERRY FLOOR	001	168+3=171
TEAM CSC	010	64+3=67
T-MOBILE-TEAM	011	104+3=107
TEAM TELEKOM	100	96+3=99
DISCOVERY CHANNEL TEAM	101	178+3=181
Summe		**724**

Tabelle 60: Kodierungen für Spalte 4 für Beispiel aus Tabelle 28

Für Spalte 4 sind 9*3+543 = 570 Bits zur Kodierung notwendig. Insgesamt werden bei diesem spaltenbasierten Kodierschemata also 48+602+48+724 = 1.422 Bits zur Kodierung benötigt. Es ist also gegenüber dem blockbasierten Kompressionsansatz, was die Rohdaten dieses Beispiels angeht, keine nennenswert bessere Komprimierung gegeben, wenn der gleiche Kodieransatz (Wörterbuchkompression) verwendet wird.

Bei Spalten mit sortierten Zahlen hingegen kann der spaltenorientierte Ansatz seinen Vorteil z.B. durch Einsatz der Lauflängenkodierung als Kompressionsverfahen ausschöpfen. Bei einer Spalte mit beispielsweise 100 unterschiedlichen Werten und 100.000 Records wäre die Kodierung dieser 100 Werte bei Blockkompression $\log_2 100$ = 7 Bits. Bei 100.000 Referenzierungen wären dies 100.000 * 7 Bits = 700.000 Bits. Bei Einsatz von Lauflängenkodierung würde man für 100 Paare $(Wert, Lauflänge)$ mit je 2*8 Bit Kodierung insgesamt 16*100 = 1.600 Bits benötigen, also wäre die Kompression hier 437,5 mal höher.

A.7 Verteilte und parallele Datenbanken

A.7.1 Verteilte Datenbanken

Tabelle 61 zeigt zusammenfassend eine Klassifikation von parallelen DBMS

Kriterium	Shared Memory	Shared Disk	Shared Nothing
Externspeicher-zuordnung	Zugriff auf alle Platten von allen Prozessoren aus	Zugriff auf alle Platten von allen Prozessoren aus	Zugriff nur auf lokale Platte
Fragmentierung der Datenbank zwingend	Nein, aber möglich	Nein, aber möglich	Ja, Zugriff auf Fragment nur über lokalen DBMS-Prozess
Hauptspeicher-zuordnung	Prozessoren teilen sich gemeinsamen Hauptspeicher	Jeder Prozessor hat eigenen Hauptspeicher	Jeder Prozessor hat eigenen Hauptspeicher
Räumliche Verteilung	Räumlich nahe Anordnung der Prozessoren	Räumlich nahe Anordnung der Prozessoren	Räumlich nahe oder verteilte Anordnung der Prozessoren (wie bei DDBMS)

Tabelle 61: Klassifikation von parallelen DBMS (ähnlich Kapitel 3 [Rahm94])

Tabelle 62 (aus [Gon09]) zeigt zusammenfassend die Unterschiede zwischen parallelen DBMS und verteilten DBMS.

	Parallele DBMS	Verteilte DBMS
Räumliche Verteilung	DBMS-Knoten sind in der Regel in einem Rechner oder in räumlicher Nähe	DBMS-Knoten sind geographisch weit verteilt
Performance	Parallele DBMS besitzen bessere Leistungs- und Verfügbarkeits-merkmale wegen paralleler Ausführung und schneller Kommnunikationsverbindung als zentralisierte Datenbanksysteme unter Beibehaltung einer zentralen Administrierbarkeit (Kapitel 1.1 [Rahm94]).	Performance gut bei Anfragen an lokalen Datenbestand, langsam bei Einbeziehung entfernter DBMS-Knoten (abhängig von der Kommunikationsverbindung)
Administration	In der Regel Beibehaltung zentraler Administration wegen räumlicher Nähe	Keine zentrale Administration. Die Verteilung erfolgt unter anderem wegen der Aufteilung der Administration.
Homogenität und Heterogenität (u.a. Kapitel 22.1.3 [CB05])	Homogenität der Software und Hardware der DBMS-Knoten (d.h. Auf allen Knoten läuft dasselbe DBMS)	Der Einsatz von heterogener Hard- und Software (Betriebssysteme und DBMS) ist nicht unüblich.
Datenautonomie	Lokale Autonomie über die Daten nur bei Shared Nothing	Lokale Autonomie über die Daten auf jedem Rechnerknoten

Tabelle 62: Unterschiede zwischen parallelen und verteilten DBMS (aus [Gon09])

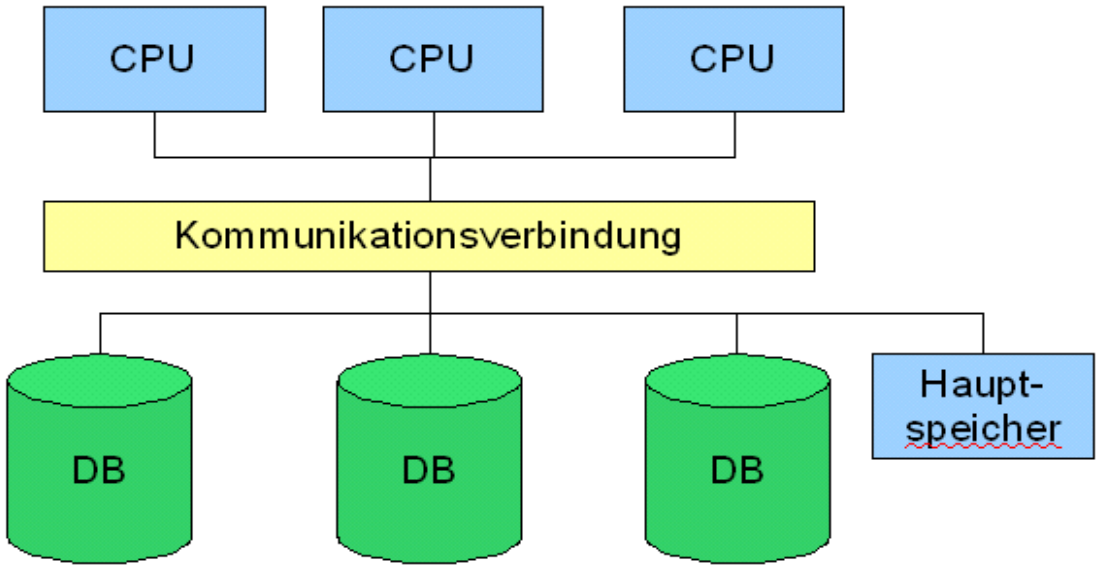

Abbildung 57: Shared-Memory-Umgebung (aus [CB05])

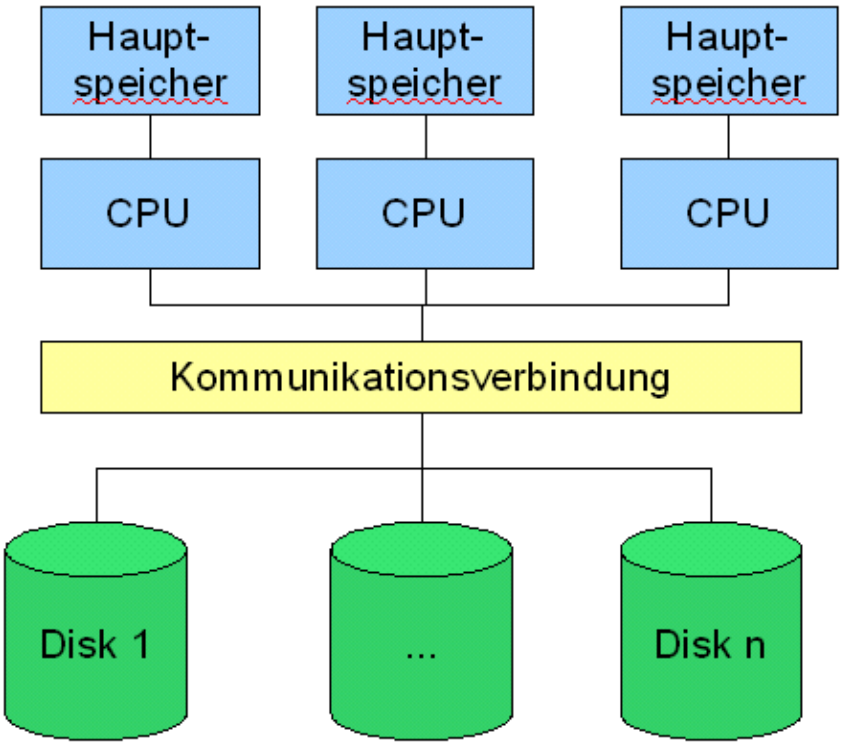

Abbildung 58: Shared-Disk-Umgebung (aus [CB05])

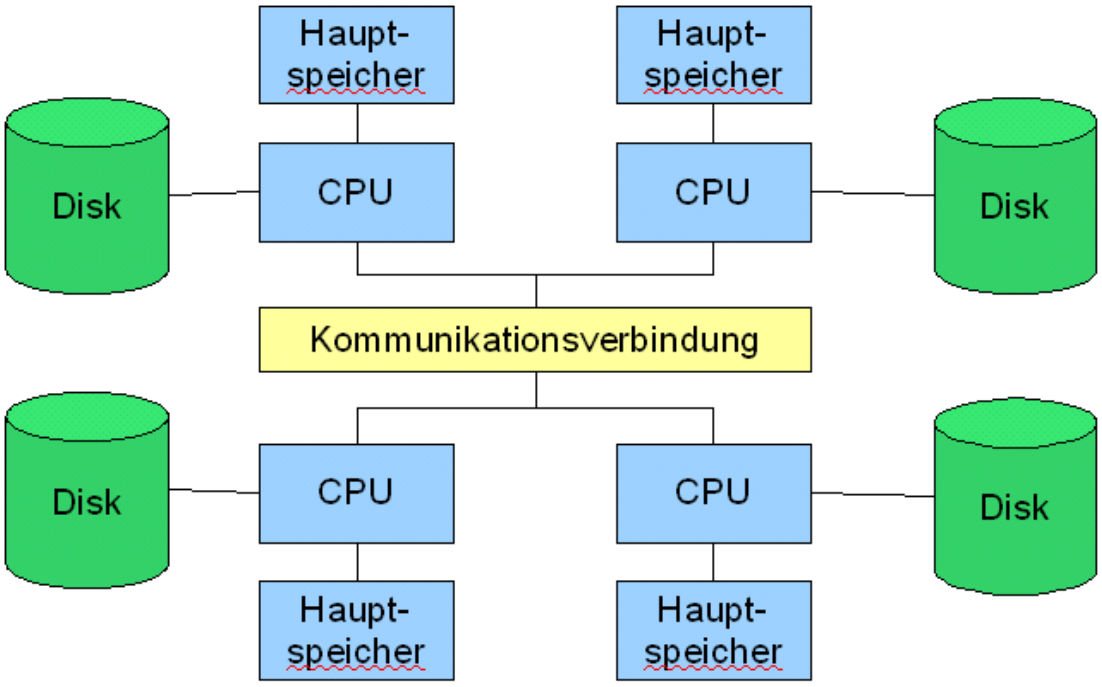

Abbildung 59: Shared-Nothing-Umgebung (aus [CB05])

A.7.2 Horizontale und vertikale Partitionierung

Die Abbildungen 60 und 61 zeigen eine vertikal partitionierte Relation, wie sie in **coDBMS** realisiert ist.

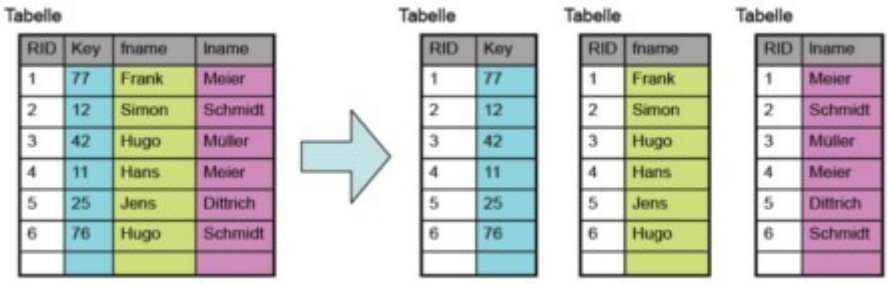

Abbildung 60: Logische Dekomposition einer Relation, [Seite 10 [ETH1]]

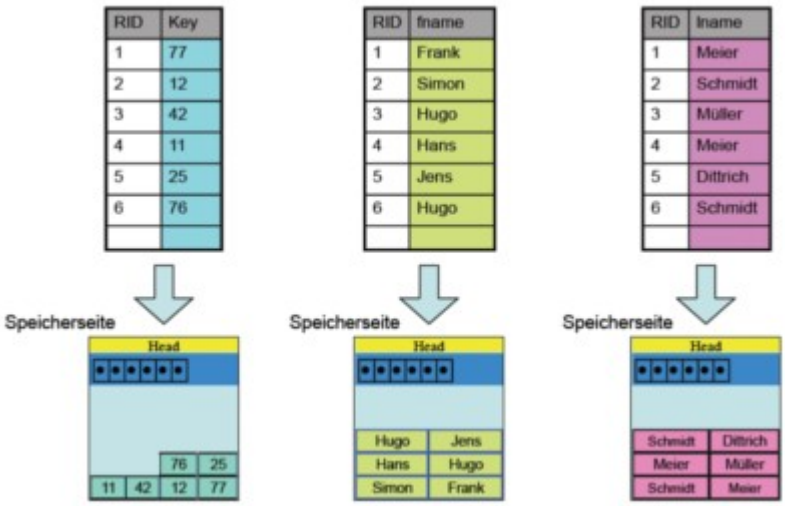

Abbildung 61: Physische Speicherung der Dekomposition aus Abbildung 53, [Seite 11 [ETH1]]

B Glossar

***ACID*:** Das ACID-Prinzip beschreibt die Eigenschaften, die beim Durchführen einer Transaktion eingehalten werden müssen, soll die Datenbank in einem konsistenten Zustand bleiben. Die durch ACID beschriebenen Eigenschaften sind

1. **Atomicity**, was bedeutet, dass eine Transaktion ganz oder gar nicht ausgeführt und bei Abbruch der Transaktion keine Zwischenergebnisse gespeichert werden,
2. **Consistency**, d.h. die Datenbank ist vor und nach der Transaktion jeweils in konsistentem Zustand,
3. **Isolation**, d.h. Benutzer soll das Gefühl haben, dass er alleine mit der DB arbeitet (was erreicht wird durch die Serialisierbarkeit von parallelen Transaktionen), sowie
4. **Durability**, d.h. Bei erfolgreicher Ausführung der Transaktion muss das Ergebnis dauerhaft gespeichert werden.

***Bulkload*:** Ein Bulkload ist das Einfügen und/oder Ändern sehr großer Datenmengen.

***Commodity Computing*:** Commodity Computing beschreibt den Einsatz von handelsüblichen Rechnersystemen (billige Massenware) auch in großen Rechnerverbunden (Computergrids). Die Komponenten dieser Rechnersysteme werden von vielen verschiedenen Herstellern angeboten und sind über offene Standards integrierbar. Es handelt sich nicht um proprietäre Systeme eines bestimmten Herstellers oder Herstellerverbunds. Dazu gehört auch der Umstand, dass Commodity-Rechner mit beliebigen handelsüblichen Betriebssystemen (Windows, Linux/Unix, Mac OS) betrieben werden können. Commodity Computing ist heutzutage der übliche Ansatz, da mit steigender Rechenleistung von handelsüblichen Rechnern auch komplexe, d.h. rechenintensive Anwendungen auf diesen bzw. auf Verbunden von vernetzten Commodity-Rechnern betrieben werden können. Ein typischer Commodity-Rechner besteht heutzutage aus einem ATX-Mainboard, einem Dual-/Quadcore-Prozessor mit einem einheitlichen Basisbefehlssatz, DDR2-RAM, DVD-Laufwerk, USB-Anschlüssen, Netzwerkanschlüssen, SATA-Anschlüssen für Festplatten, PCI-Slots für Erweiterungskomponenten sowie Audio- und Videoanschlüssen, siehe [ComComp].

***Dimension*:** Eine *Dimension* ist eine bestimmte Perspektive der (konsolidierten) Daten [Seite 6 [CoCS93]], aus der heraus diese Daten analysiert werden können.

Exact-match-query: Eine Art des Lookups (Suchen von Datensätzen), bei der Werte für alle Attribute einer Relation angegeben werden.

Equality-Join: Beim Equality- oder Equi-Join erfolgt die Verbindung von Tupeln zweier Relationen, wenn der Wert bestimmter (d.h. festgelegter) Spalten identisch ist. Beispiel für einen Equi-Join ist die Anfrage `SELECT * FROM R,S WHERE R.a=S.b;`

***Join*:** Ein Join (Verbund) ist ein kartesisches Produkt mit einer anschließend ausgeführten Selektion. Es gibt verschiedene Joins in der Relationenalgebra, und zwar den Equi Join, den Natural Join, den Semi Join, den (Left/Right/Full) Outer Join, vgl. [WikiRA].

***Multidimensionales Datenmodell*:** Das multidimensionale Datenmodell beschreibt die Perspektiven, aus der die Daten einer Organisation betrachtet werden können. Die simultane Analyse mehrerer *Dimensionen* wird multidimensionale Datenanalyse genannt.

Natural Join: Der Natural Join $R \bowtie S$ von zwei Relationen R,S ist ein Equi-Join (Prüfung auf Wertgleichheit) von je 1-n gleichbezeichneten Attributen zweier Relationen. Wenn die Attribute $A_1,A_2,...,A_n$ sowohl in R als auch in S vorkommen, dann werden zwei Tupel $r \in R$ und $s \in S$ bei Wertgleichheit von $r_{A1},r_{A2},...r_{An}$ und $s_{A1},s_{A2},...,s_{An}$ miteinander verbunden. Gibt es keine zwei gleichen Attribute, ist das Ergebnis des Natural Join das kartesische Produkt. Gleiche Attribute werden im Ergebnis durch eine Projektion ausgeblendet ([WikiRA], [Seite 43 [GMUW09]].

A	B	C	D
1	2	3	4
4	5	6	7
7	8	9	0

A	F	G
1	2	3
7	8	9

A	B	C	D	F	G
1	2	3	4	2	3
7	8	9	0	8	9

Tabelle 63: Beispiel für Natural Join aus [WikiRA]

***Selektivität*:** Die Selektivität beschreibt das Verhältnis der Anzahl der Ergebnistupel zu den Tupeln der Eingangsrelation. Eine Selektivität von 10% (bzw. 0,1) bedeutet, dass aus 100 Eingangstupeln 10 Ergebnistupel erzeugt werden (Seite 442 [SHS05])

Semi-Join: Ein Semi-Join R $\ltimes$ S bzw. R $\rtimes$ S berechnet den Anteil eines *Natural Joins*, der nach Reduktion auf die linke oder rechte Relation übrig bleibt. Ein Beispiel für zwei Relationen R $\ltimes$ S ist für die Verbundbedingung R.A=S.A in Tabelle 64 beschrieben.

A	B	C	D
1	2	3	4
4	5	6	7
7	8	9	0

A	F	G
1	2	3
7	8	9

A	B	C	D
1	2	3	4
7	8	9	0

Tabelle 64: Semi-Join für Relation R und S [WikiRA]

Star-Join: Ein Star-Join ist ein typisches Anfragemuster der relationalen Umsetzung von **OLAP**-Operationen [Seite 670 [SSH08]]. Aus einer großen Menge vorhandener Fakten (der Faktentabelle) wird ein bestimmter, in mehreren Dimensionen beschränkter (restriktiver) Datenbereich ausgewählt. Auf diesem werden Aggregationen berechnet [Seite 640 [SSH08]]. Das nachfolgende Beispiel aus [Seite 640 [SSH08]] beschreibt die Umsetzung der Anfrage: „Welche *Umsätze* sind in den Jahren 2005 und 2006 in den *Warengruppen* Sekt und Rotwein in den *Bundesländern* Sachsen-Anhalt und Thüringen pro Monat und Ort angefallen ?“. Die Umsätze sind hierbei die Fakten, die Einschränkung der Fakten erfolgt bezüglich der Dimensionen Jahr, Bundesland und Warengruppe. Über den Dimensionen werden Summen gebildet. Tabelle 65 zeigt die SQL-Anfrage für den Star-Join.

```
Select O.Stadt, Z.Monat, sum (Umsatz)
from VERKAUF V, ZEIT Z, PRODUKT P, ORT O
where V.Prod_ID = P.Prod_ID and
V.Zeit_ID = Zeit_ID and
V.Filial_ID = O.Filial_ID and
(P.Gruppe = 'Rotwein' or P.Gruppe = 'Sekt') and
(O.Bundesland  = 'Thüringen' or
 O.Bundesland = 'Sachsen-Anhalt') and
Z.Jahr between 2005 and 2006
group by O.Stadt, Z.Monat
```

Tabelle 65: SQL-Anfrage für einen Star-Join [Seite 640 [SSH08]]

***Transaktion*:** Eine Transaktion ist eine zu einer Funktionseinheit zusammengefasste Folge von Datenbankoperationen, die die Datenbank von einem konsistenten in einen konsistenten Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.

Literaturverzeichnis

[1010Arch] 1010data: *Data Warehousing: How Does 1010data Fit Into My Architecture?*, Zugriff 8. September 2009, http://www.1010data.com/solutions.dw.architecture.html

[Aba06] Daniel J. Abadi, Samuel R. Madden und Miguel C. Ferreira: *Integrating Compression and Execution in Column-Oriented Database Systems*, Proceedings of SIGMOD, Juni 2006, http://db.csail.mit.edu/projects/cstore/abadisigmod06.pdf

[Aba08] Daniel J. Abadi: *Query Execution in Column-Oriented Database Systems*, Dissertation, Massachussetts Institute of Technology, Februar 2008, http://cs-www.cs.yale.edu/homes/dna/papers/abadiphd.pdf

[AbaVP] Daniel Abadi: *Debunking Another Myth: Column-Stores vs. Vertical Partitioning*, Zugriff 10. September 2009, http://databasecolumn.vertica.com/2008/07/debunking-another-myth-columns.html

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill und Marios Skounakis: *Weaving Relations for Cache Performance*, Proceedings of VLDB, 2001, http://www.cs.wisc.edu/multifacet/papers/vldb01_pax.pdf

[Ah06] Rav Ahuja: *Introducing DB2 9, Part 1: Data compression in DB2 9*, Zugriff 12. August 2009, http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja/index.html

[Als75] P.A. Alsberg: *Space and Time Savings through large Database Compression and Dynamic Restructuring*, Proceedings of IEEE, August 1975, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1451833

[Ba85] M. Bassiouni: *Data Compression in Scientific and Statistical Databases*, IEEE Transaction on Software Engineering, Oktober 1985, http://portal.acm.org/citation.cfm?id=4509

[BELZ78] Binary Essence: *Lempel-Ziv-78 (LZ78)*, Zugriff 5. August 2009, http://www.binaryessence.com/dct/en000140.htm

[Bere95] Hal Berenson et. al: *A Critique of ANSI SQL IsolationLevels*, Proceedings of ACM SIGMOD, Juni 1995, ftp://ftp.research.microsoft.com/pub/tr/tr-95-51.pdf

[BG09] Andreas Bauer, Holger Günzel: *Data-Warehouse-Systeme*, dpunkt, 3. Auflage 2009

[Bloom] Burton H. Bloom: *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, Juli 1970, http://gnunet.org/papers/p422-bloom.pdf

[Bloor08] Philip Howard:: *What's cool about columns*, Zugriff März 2008, http://www.bloorresearch.com/research/Spotlight/939/Whats-cool-about-columns.html

[BM72] R. Bayer und E.M. McCreight: *Organization and maintenance of large ordered indices*, Acta Informatica Band 1, 1972, http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/Bayer_hist.pdf

[BNET1] BNET (CBS Interactive): *NCR demonstrates world's largest data warehouse. Record-breaking 11 Terabyte Data Warehouse Uses NCR WorldMark Enterprise Servers*, Zugriff 31. Juli 2009, http://findarticles.com/p/articles/mi_m0EIN/is_1996_Feb_27/ai_18030415/

[Bo09] Marcus Bombe: *Arithmetische Kodierung*, Zugriff Januar 2009, http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Lehre/WS0809/Datenkompression/arithmetischecodierung.pdf

[Bright08] Dominik Slezak, Jakub Wroblewski: *Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries*, Proceedings of VLDB, August 2008, www.vldb.org/pvldb/1/1454174.pdf

[BTWiki] Wikimedia e.V.: *BigTable*, Zugriff Juli 2009, http://en.wikipedia.org/wiki/BigTable

[Burl09] Oracle Tips by Burleson: *Using the Oracle 10g Bitmap Join Index*, Zugriff 29. August 2009, http://www.fast-track.cc/t_lessons_join_index.htm

[CasProj] Apache Software Foundation: *Cassandra Project Homepage*, Zugriff 2. September 2009, http://incubator.apache.org/cassandra/

[CB05] Thomas M. Connolly, Carolyn E. Begg: *Database Systems - A practical approach to Design, Implementation and Management*, Addison-Wesley, 4. Auflage 2005

[Chang06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: *Bigtable: A Distributed Storage System for Structured Data*, Proceedings of OCSI, 2006, http://labs.google.com/papers/bigtable.html

[Cle00] Alan Clements: *The Principles of Computer Hardware*, Oxford University Press, 3. Auflage 2000

[ClStore] Cloudstore: *Cloudstore Homepage*, Zugriff 12. September 2009, http://kosmosfs.sourceforge.net/

[CoCS93] E.F. Codd, S.B. Codd, C.T. Salley: *Providing OLAP (On-Line Analytical Processing) to User Analysts: An IT Mandate*, 1993, http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf

[Coding1] University of Saskatchewan: *Data Compression - Statistical Coding*, Zugriff 31. Juli 2009, http://www.cs.usask.ca/content/resources/tutorials/csconcepts/2001_1/tutorial.htm

[CoKho85] G.P. Copeland, S. Khoshafian: *A Decomposition Storage Model*, Proceedings of SIGMOD, 1985, http://portal.acm.org/citation.cfm?id=318898.318923

[ComComp] Wikimedia Foundation: *Commodity Computing*, Zugriff 12. September 2009, http://en.wikipedia.org/wiki/Commodity_computing

[CSHome] MIT: *C-Store Homepage*, Zugriff 17. Juli 2009, http://db.lcs.mit.edu/projects/cstore/

[CSWik] Wikimedia e.V.: *C-Store*, Zugriff 17. Juli 2009, http://en.wikipedia.org/wiki/C-Store

[ct00] Michael Tamm: *Packen wie noch nie*, c't 16 2000

[DB2Compr1] Toolbox for IT Blog: *Getting the Most out of DB2 for z/OS and System z*, Zugriff 9. August 2009, http://it.toolbox.com/blogs/db2zos/the-incredible-shrinking-data-db2s-compression-part-i-10568

[DBClasses] Universität Karlsruhe: *Welche Arten von Datenbankmanagementsystemen gibt es?*, Zugriff 9. September 2009, http://www.rz.uni-karlsruhe.de/rd/3633.php

[DBM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi und Samuel Madden: *Performance Tradeoffs in Read-Optimized DBMS*, Proceedings of VLDB, September 2006, http://db.csail.mit.edu/projects/cstore/VLDB06.pdf

[DCRC97] Data Compression Reference Center: *The LZ78 algorithm*, Zugriff 5. August 2009, http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/lz78.html

[DMS09] David J. DeWitt, Samuel Madden und Michael Stonebraker: *How To Build a High-Performance Data Warehouse*, Zugriff Juni 2009, http://db.csail.mit.edu/madden/high_perf.pdf

[DPRes] Thomson Reuters: *DataProbe - Research*, Zugriff 6. September 2009, http://home.thomsonhealthcare.com/Products/view/?id=215

[ETH1] Dr. Jens Dittrich: *Aktuelle Trends: Eine Reise von Hauptspeicherdatenbanken zu Dataspace Management Systemen*, Juli 2007, http://www.inf.ethz.ch/personal/jensdi/publications/ULDBkeynote.pdf

[GFS2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: *The Google File System*, Proceedings of SOSP, Oktober 2003, http://labs.google.com/papers/gfs.html

[GlusterFS] Gluster Inc.: *GlusterFS Homepage*, Zugriff 12. September 2009, http://www.gluster.com/community/documentation/index.php/GlusterFS

[GMUW09] Hector Garcia-Molina, Jeffrey D. Ullmann und Jennifer Widom: *Database Systems - The Complete Book*, Pearson, 2nd Edition 2009

[GoCo09] Google Code: *How Hypertable works*, Zugriff 3. September 2009, http://code.google.com/p/hypertable/wiki/HowHypertableWorks

[Gon09] Mani Kumar Gondi: *Introduction to Distributed Database Management Systems*, Zugriff 14. August 2009, http://www.cs.siue.edu/~behlman/Classes/cs534/Topic%20Papers/DistrDBMSs%20Introduction.doc

[GoogleMR] Jeffrey Dean und Sanjay Ghemawat: *MapReduce: Simplied Data Processing on Large Clusters*, Proceedings of OSDI, 2004, http://labs.google.com/papers/mapreduce-osdi04.pdf

[Gray08] J. Gray: *Hadoop/Hbase vs. RDBMS*, Zugriff 2. September 2009, http://www.docstoc.com/docs/2996433/Hadoop-and-HBase-vs-RDBMS

[GS91] Goetz Graefe, Leonard D. Shapiro: *Data Compression and Database Performance*, Proceedings of ACM/IEEE Symposium on Applied Computing, April 1991, http://web.cecs.pdx.edu/~len/compression.pdf

[HBAPI] Apache Software Foundation: *HBase 0.19.3 API* , Zugriff 2. September 2009, http://hadoop.apache.org/hbase/docs/current/api/overview-summary.html#overview_description

[HBArch] Apache Software Foundation: *HbaseArchitecture*, Zugriff 2. September 2009, http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture

[HBBloom] Jimbowj.com: *Understanding Hbase column-family performance options*, Zugriff 13. September 2009, http://jimbojw.com/wiki/index.php?title=Understanding_HBase_column-family_performance_options

[HBHome] Apache Software Foundation: *HBase Homepage*, Zugriff 2. September 2009 2009, http://hadoop.apache.org/hbase/

[HbShell] Apache Software Foundation: *Hbase/Shell*, Zugriff 12. September 2009, http://wiki.apache.org/hadoop/Hbase/Shell

[HBWiki] Wikimedia Foundation: *HBase*, Zugriff 2. September 2009, http://en.wikipedia.org/wiki/HBase

[Hen08] Doug Henschen: *Column-store databases and DW appliances: How to makethe right choice.*, Zugriff Juni 2009,

http://www.intelligententerprise.com/showArticle.jhtml;jsessionid=4XZ3AGGOMXKPQQSNDLPSKHSCJUNN2JVN?articleID=206901279&pgno=2

[Hit05] Andrew Hitchcock: *Google's BigTable*, Zugriff 31. Juli 2009, http://andrewhitchcock.org/?post=214

[HR99] Theo Härder und Erhard Rahm: *Datenbanksysteme - Konzepte und Techniken der Implementierung*, Springer, 1. Auflage 1999

[HS00] Andreas Heuer, Gunter Saake: *Datenbanken: Konzepte und Sprachen*, mitp, 2. Auflage 2000

[HT09] Hypertable: *Hypertable: An Open Source, High Performance, Scalable Database*, Zugriff 3. September 2009, http://www.hypertable.org/

[HTArch09] Hypertable: *ArchitecturalOverview*, Zugriff 3. September 2009, http://code.google.com/p/hypertable/wiki/ArchitecturalOverview

[HTHome] Hypertable: *Hypertable Homepage*, Zugriff 2. September 2009, http://www.hypertable.org/

[HTTable] Hypertable: *CREATE TABLE*, Zugriff 12. September 2009, http://www.hypertable.org/hql/create-table.html

[HTWiki] Wikimedia Foundation: *Hypertable*, Zugriff 2. September 2009, http://en.wikipedia.org/wiki/Hypertable

[Huff52] David A. Huffman: *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE, September 1952, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?&arnumber=4051119

[HuffWiki] Wikimedia Foundation: *Huffman Coding*, Zugriff 4. August 2009, http://en.wikipedia.org/wiki/Huffman_coding

[IBForum] Infobright Forum: *Fast Data Load and Comparison*, Zugriff 10. September 2009, http://www.infobright.org/Forums/viewthread/425/

[IBWP08] Infobright Inc.: *Infobright: Whitepaper zur analytischen Data-Warehouse-Technologie Enterprise Edition*, August 2008, http://www.infobright.com/?ACT=62&resource_id=25

[ICE31Specs] Infobright: *ICE 3.1 Product Specs*, Zugriff 17. August 2009, http://www.infobright.org/downloads/pdfs/ICE%203.1.1_Product-Specs.pdf

[IW94] Balakrishna R. Iyer, David Wilhite: *Data Compression Support in Databases*, Proceedings of the 20th VLDB Conference, 1994, http://www.vldb.org/conf/1994/P695.PDF

[Jimbojw09] Jimbojw.com: *Understanding HBase and BigTable*, Zugriff 2. September 2009, http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable

[Kel07] Jim Kellerman: *HBase: Structured Storage of sparse data for Hadoop*, Dezember 2007, http://wiki.apache.org/hadoop-data/attachments/HBase(2f)HBasePresentations/attachments/HBase.pdf

[Kho87] Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, und Patrick Valduriez: *A query processingstrategy for the decomposed storage model*, Proceedings of ICDE, 1987, A query processingstrategy for the decomposed storage model

[Kue09] Kersten Kühne: *Verwaltung von Statistiken zu Indexkonfigurationen für das Self-Tuning*, Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Februar 2009, http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/diplomarbeiten/Kue09.pdf

[KWiki09] Wikimedia Foundation: *K (programming language)*, Zugriff 23. August 2009, http://en.wikipedia.org/wiki/KDB_(database)

[Kx09] Kx Systems: *The kdb+ Database - A unified database for streaming and historical data*, 2009, http://kx.com/papers/Kdb+_Whitepaper_2009.pdf

[Lex09] Microsoft Press: *Computer-Fachlexikon mit Fachwörterbuch*, Zugriff 14. August 2009, http://www.hirzel.org/book/p.htm

[Los09] David Loshin (Sybase): *Gaining the Performance Edge using a Column-Oriented Database Management System)*, Zugriff 2009, http://www.sybase.com/files/White_Papers/Performance-Column-DBM-041509-WP.pdf

[LucArch] Lucid: *LucidDB Architecture*, Zugriff 25. August 2009, http://www.luciddb.org/arch.html

[Lucid] LucidEra Inc.: *LucidDB Home Page*, Zugriff 25. August 2009, http://www.luciddb.org/

[LucJoinOpt] The Eigenbase Project: *LucidDbJoinOptimization*, Zugriff 27. August 2009, http://pub.eigenbase.org/wiki/LucidDbJoinOptimization

[LucStor] Lucid: *LucidDBDataStorageAndAccess*, Zugriff 25. August 2009, http://pub.eigenbase.org/wiki/LucidDbDataStorageAndAccess

[Lue07] Andreas Lübcke: *Self-Tuning-Konzepte zur Verwaltung vonBitmap-Index-Konfigurationen*, Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, April 2007, http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/diplomarbeiten/Lue07.pdf

[LuFeat] Lucid Inc.: *LucidDB Features*, Zugriff Juli 2009, http://www.luciddb.org/features.html

[LZ] University of California: *Data Compression - Lempel-Ziv Codes*, Zugriff 4. August 2009, http://www.ics.uci.edu/~dan/pubs/DC-Sec5.html

[LZ77] Jacob Ziv and Abraham Lempel: *A universal algorithm for sequential data compression.*, IEEE Transactionson Information Theory, Mai 1977, http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

[LZ78] Jacob Ziv und Abraham Lempel: *Compression of individual sequences via variable-rate coding*, IEEE Transactions on Information Theory, September 1978, http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1978_variable-rate.pdf

[LZAlg] Christina Zeeh: *The Lempel-Ziv-Algorithm*, Zugriff 4. August 2009, http://tuxtina.de/files/seminar/LempelZivReport.pdf

[LZO] Marcus Oberhumer: *LZO Data Compression Library*, Zugriff 3. September 2009, http://www.oberhumer.com/opensource/lzo/

[LZODoc] Marcus Oberhumer: *LZO Documentation*, Zugriff 3. September 2009, http://www.oberhumer.com/opensource/lzo/lzodoc.php

[M08] Dr. Klaus Manhart: *SQL: Indexkompression und effiziente Speichertechnologie*, Zugriff 12. August 2009, http://www.tecchannel.de/server/sql/1767492/sql_indexkompression_und_effiziente_speichertechnologie/index2.html

[MapRed] Jeffrey Dean und Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, Sixth Symposium on Operating System Design and Implementation, Dezember 2004, http://labs.google.com/papers/mapreduce.html

[MArch09] The Mail Archiv: *Cassandra vs. Hbase*, Zugriff 2009, http://www.mail-archive.com/hbase-user@hadoop.apache.org/msg05908.html

[MFIndex] Kevin Burton's New Feed Blog: *Java an In-Memory Data Structures - A Flat File Map Proposal*, Zugriff 12, September 2009, http://feedblog.org/2008/03/23/java-and-in-memory-data-structures-a-flat-file-map-proposal/

[Mic05] Michael Stonebraker et. al: *C-Store: A column-oriented DBMS*, Proceedings of VLDB 2005, Januar 2005, http://db.lcs.mit.edu/projects/cstore/vldb.pdf

[Mo04] Dominik Morent (Uni Konstanz): *The Decompositional Storage Model*, 23. Februar 2004, http://www.inf.uni-konstanz.de/dbis/teaching/ws0304/datatypes/download/paper-decompositional-storage-model

[MySQLSE09] MySQL AB: *MySQL Pluggable Storage Engine Architecture*, Zugriff 11. September 2009, http://solutions.mysql.com/engines.html

[Nab09] Nabble Forum: *How is HBase column oriented ?*, Zugriff 2. September 2009, http://www.nabble.com/How-is-Hbase-column-oriented---td22841566.html

[NQ97] P. O'Neil und D. Quass: *Improved query performance with variant indexes*, Proceedings of SIGMOD, 1997, http://www.cs.umb.edu/~poneil/varindex.ps

[ONEI96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil,: *The Log-Structured Merge-Tree*, Acta Informatica 33, Juni 1996, http://www.cs.umb.edu/~poneil/lsmtree.ps

[OraCompr] Dr. Klaus Uebelgünn: *Datenkompression unter Oracle (Teil I)*, Zugriff 1. August 2009, http://www.ordix.de/ORDIXNews/3_2006/Datenbanken/oracle_datenkompression.html

[OraCompr2] Oracle: *Table Compression in Oracle Database 10g Release 2An Oracle White Paper*, Mai 2005, http://www.oracle.com/index.html

[OraCompr3] Bill Hodak: *Advanced Compression in Oracle Database 11g*, November2007, http://www.oracle.com/technology/deploy/performance/pdf/s291861.pdf

[OraCompr4] Oracle White Paper: *Oracle Advanced Compression (11g)*, April 2008, http://www.oracle.com/technology/products/database/oracle11g/pdf/advanced-compression-whitepaper.pdf

[OraCompr5] Klaus Reimers: *Compression, weniger ist mehr (Oracle 11g)*, Zugriff 10. August 2009, http://www.ordix.de/ORDIXNews/2_2009/Datenbanken/oracle_11g_teil_vi.html

[OraCompr6] Dr. Klaus Uebelgünn: *Komprimierung von Index Organized Tables*, Zugriff 12. August 2009, http://www.ordix.de/ORDIXNews/1_2007/Datenbanken/datenkompression_IOT.html

[ParAccel1] Merv Adrian's IT Market Strategy Blog: *ParAccel Rocks the TPC-H - Will See Added Momentum*, Zugriff 2. August 2009, http://www.paraccel.com/library/bitl/ParAccel_Rocks_the_TPC.pdf

[Paw07] Z. Pawlak, A. Skowron: *Rudiments of Rough Sets*, Information Sciences 2007

[Paw91] Z. Pawlak: *Rough Sets: Theoretical Aspects of reasoning about data*, Kluwer, 1. Auflage 1991

[Pen95] N. Pendse: *The FASMI Definition for OLAP*, Business Intelligence 1995

[Pen97] N. Pendse: *The OLAP Report: Market Share Analysis*, Business Intelligence 1997

[Rahm94] Erhard Rahm: *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung*, Addison-Wesley, 1. Auflage 1994

[RData] GNU R Project: *R Data Im/Export*, Zugriff 5. September 2009, http://cran.r-project.org/doc/manuals/R-data.html

[RDS02] Ravishankar Ramamurthy, David Dewitt, Qi Su: *A case for fractured mirrors*, Proceedings of VLDB, 2002, http://www.vldb.org/conf/2002/S12P03.pdf

[ROTH93] Mark A. Roth und Scott J. van Horn: *Database Compression*, Proceedings of SIGMOD Record 22(3), 1993, www.sigmod.org/record/issues/9309/compress.ps

[Say06] Khalid Sayood: *Introduction to Compression*, Morgan Kaufmann, 3. Auflage 2006

[ScanClass] Apache Software Foundation: *org.apache.hadoop.hbase.client.Scan - Klassendokumentation*, Zugriff 12. September 2009, http://jimbojw.com/wiki/index.php?title=Understanding_HBase_column-family_performance_options

[Sed92] Robert Sedgewick: *Algorithmen in C*, Addison-Wesley, 1. Auflage 1993

[SHS05] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler: *Datenbanken: Implementierungstechniken*, mitp, 2. Auflage 2005

[SK08] Knut Stolze, Marcel Karnstedt: *Datenbankimplementierungstechnikenzum Anfassen - Ein Erfahrungsbericht*, Datenbank-Spektrum 27 2008

[Sky451] the451group: *Impact Report: Skytide hones focus on analyzing online content*, Zugriff 9. Oktober 2009, http://www.skytide.com/

[SkyHome] Skytide: *Homepage*, Zugriff 6. September 2009, Skytide Insight for Digital Media

[SkyICDN] Skytide: *Skytide Insight for Content Delivery Networks - Product Sheet*, 2009, http://www.skytide.com/

[SkyIDM] Skytide: *Skytide Inside for Digital Media - Product Sheet*, 2009, http://www.skytide.com/

[SSH08] Gunter Saake, Kai-Uwe Sattler, Andreas Heuer: *Datenbanken: Konzepte und Sprachen*, mitp, 3. Auflage 2008

[Stack09] Michael Stack: *HBasics: An Introduction to Hadoop HBase*, ApacheCon, Amsterdam, 25. März2009, http://wiki.apache.org/hadoop-data/attachments/HBase(2f)HBasePresentations/attachments/HbaseApacheConEU03252009-2.pdf

[SWiki] Wikimedia Foundation: *S (programming language)*, Zugriff 6. September 2009, http://en.wikipedia.org/wiki/S_programming_language

[Sybase1] Sybase: *Petabyte Data Warehouse Using Solaris 10 OS, Sybase IQ and BMMsoft DataFusion*, Zugriff 2. August 2009, http://www.sybase.com/detail?id=1054011

[SybaseIQLOB] Sybase: *Sybase® IQ — Large Object Management Option*, Zugriff 2. August 2009, http://www.sybase.com/detail?id=1054301

[TBFeat] 1010data: *Tenbase Features*, Zugriff 8. September 2009, http://www.1010data.com/solutions.dw.tenbase.html

[TH07] Tom's Hardware: *: SATA-Festplatten-Benchmark 2007*, Zugriff 2. August 2009, http://www.tomshardware.com/de/festplatten-vergleichstest-samsung-western-digital-hitachi-seagate,testberichte-237711-17.html

[ThRHome] Thomson Reuters : *Homepage*, Zugriff 6. September 2009, http://www.thomsonreuters.com/products_services/healthcare/gov_federal/

[TPC-H1] Transaction Processing Performance Council (TPC): *Top-Ten TPC-H by Price/Results*, Zugriff 2. August 2009, http://www.tpc.org/tpch/results/tpch_price_perf_results.asp

[Val09] Paradigma Software Inc.: *Why Valentina is Better?*, Zugriff 27. August 2009, http://valentina-db.net/index.php/de/component/content/article/55-white-papers/94-why-valentina-is-better

[Val09-2] Paradigma Software Inc.: *Valentina Software - Articles*, Zugriff 25. August 2009, http://www.valentina-db.com/dokuwiki/doku.php?id=valentina:articles:articles

[VectorStarDoc] Vectornova: *VectorStar Documentation*, Zugriff 28. August 2009, http://www.vectornova.com/doc.htm

[VERT1] Vertica Inc.: *Vertica Analytic Database Benchmarks*, Zugriff 31. Juli 2009, http://www.vertica.com/benchmarks/

[VertAdmin] Vertica Systems Inc.: *Vertica Analytic Database 3.0 - Administrator's Guide*, 2009, http://www.vertica.com/

[VertCompr] Vertica Systems Inc.: *Squeezing More Value Out Of Data Warehouses through the Power of Database Compression*, August 2009, http://www.vertica.com/_pdf/VerticaDatabase-ThePowerOfCompressionInDataWarehousing.pdf

[VertConcepts] Vertica Systems Inc.: *Vertica Analytic Database 3.0 Concepts Guide*, Juni 2009, http://www.vertica.com/

[VertSQL] Vertica Systems Inc.: *Vertica Analytic Database 3.0 - SQL Reference Manual*, 2009, http://www.vertica.com/

[VertWC08] Winter Corporation: *The Vertica Analytic Database*, 2008, The Vertica Analytic Database

[VertWhitePaper] Vertica Systems: *The Vertica Analytic Database - Technical Overview White Paper*, Zugriff November 2008, http://www.vertica.com/_pdf/VerticaArchitectureWhitePaper.pdf

[VERTWIKI] Wikimedia e.V.: *Vertica*, Zugriff 29. Juli 2009, http://en.wikipedia.org/wiki/Vertica

[VertWP] Vertica Systems: *The Vertica Analytic Database - Technical Overview White Paper*, Zugriff November 2008, http://www.vertica.com/_pdf/VerticaArchitectureWhitePaper.pdf

[VSHome] Vectorstar Homepage: *VectorStar Features*, Zugriff 26. August 2009, http://www.vectornova.com/home.htm

[VSOverview] Vectornova: *VectorStar Overview*, Zugriff 27. August 2009, http://www.vectornova.com/vstar_overview.htm

[Wander05] Theo Wander: *Verlustfreie Datenkompression mittels der LZ-Algorithmen*, Zugriff 4. August 2009, http://jbg.swznet.de/uni/lzw_vortrag.pdf

[WBloom] Wikimedia e.V,: *Bloomfilter*, Zugriff 13. September 2009, http://de.wikipedia.org/wiki/Bloomfilter

[Wiki1010] Wikimedia Foundation: *1010data*, Zugriff 8. September 2009, http://en.wikipedia.org/wiki/1010data

[WikiAK] Wikimedia e.V.: *Arithmetisches Kodieren*, Zugriff 6. August 2009, http://de.wikipedia.org/wiki/Arithmetisches_Kodieren

[WikiCODBMS] Wikimedia Foundation: *Column-oriented DBMS*, Zugriff 2. August 2009, http://en.wikipedia.org/wiki/Column-oriented_DBMS

[WikiIB] Wikimedia Foundation: *Infobright*, Zugriff 16. August 2009, http://en.wikipedia.org/wiki/Infobright

[WikiIDE] Wikimedia e.V.: *ATA/ATAPI*, Zugriff 1. August 2009, http://de.wikipedia.org/wiki/Integrated_Drive_Electronics

[WikiKompr] Wikimedia e.V.: *Datenkompression*, Zugriff 2. August 2009, http://de.wikipedia.org/wiki/Datenkompression

[WikiMR] Wikimedia e.V,: *MapReduce*, Zugriff 12. September 2009, http://de.wikipedia.org/wiki/MapReduce

[WikiOLAP] Wikimedia e.V,: *Online Analytical Processing*, Zugriff 13. August 2009, http://de.wikipedia.org/wiki/Online_Analytical_Processing

[WikiOLTP] Wikimedia e.V.: *Online Transaction Processing*, Zugriff 12. August 2009, http://de.wikipedia.org/wiki/Online_Transaction_Processing

[WikiR] Wikimedia Foundation: *R (programming language)*, Zugriff 5. September 2009, http://en.wikipedia.org/wiki/GNU_R

[WikiRA] Wikimedia e.V.: *Relationale Algebra*, Zugriff 8. September 2009, http://de.wikipedia.org/wiki/Relationale_Algebra

[WinterCorp05] Kathy Auerbach: *Winter Corporation 2005 TopTen Program Summary*, Mai 2006, http://www.wintercorp.com/WhitePapers/WC_TopTenWP.pdf

[WKHM] Till Westmann, Donald Kossmann, Sven Helmer und Guido Moerkotte: *The Implementation and Performance of Compressed Databases*, ACM SIGMOD Record, September 2000, http://www.dbis.ethz.ch/research/publications/sigrec00.pdf

[Wu09] John Wu: *FastBit Tutorial*, Juni 2009, https://codeforge.lbl.gov/docman/view.php/44/30/0906-FastBit-tutorial.pdf

Selbstständigkeitserklärung

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

Magdeburg, den 14. September 2009

Vorname Name