Otto-von-Guericke Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Das Grid-File als zugriffsbalancierte mehrdimensionale Indexstruktur

Verfasser:
Jan Mensing
18. März 2011

Betreuer:
Dr.-Ing. Eike Schallehn
Otto-von-Guericke Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39016 Magdeburg
Germany

Jan Mensing:
Das Grid-File als zugriffsbalancierte mehrdimensionale Indexstruktur
Otto-von-Guericke Universität
Magdeburg, 2011

Inhaltsverzeichnis

INHALTSVERZEICHNIS I

ABBILDUNGSVERZEICHNIS III

TABELLENVERZEICHNIS V

1 EINLEITUNG 1

2 INDEXSTRUKTUREN 3

2.1 ÜBERBLICK INDEXSTRUKTUREN 3
2.2 DAS GRID-FILE 5
2.2.1 Unterteilung des Datenraumes 6
2.2.2 Aufbau des Grid-Files 8
2.2.3 Funktionen des Grid-Files 10
2.2.4 Verhalten des Grid-Files 15

3 TUNING UND SELF-TUNING 18

3.1 TUNING 18
3.2 INDEXTUNING 18
3.3 SELF-TUNING 20
3.4 SELF-TUNING VON INDEXEN 21
3.5 PARTIELLE INDEXE 22
3.6 LAST - BALANCIERTE INDEXE 24

4 PROBLEMSTELLUNG UND LÖSUNGSKONZEPT 27

4.1 BETRACHTUNG DES PROBLEMFELDES 27
4.2 GRÖßENBESCHRÄNKUNG DES GRID-FILES 28
4.3 TRENNWERTERMITTLUNG MIT DEM MEDIAN 30
4.4 PARTIELLES GRID-FILE 32
4.5 ENTWURF EINES LAST-BALANCIERTEN GRID-FILES 33
4.5.1 Bucketlisten 33
4.5.2 Trennwertverlegung 35
4.5.3 Self-Tuning Konzept 38
4.5.4 Theoretisches Verhalten des last-balancierten Grid-Files 41
4.5.5 Mögliche Weiterentwicklungen 42

5 PROTOTYPISCHE IMPLEMENTIERUNG UND TESTUMGEBUNG 44

5.1 EINSCHRÄNKUNGEN DES PROTOTYPEN 44
5.1.1 Spezifikation der Daten 44
5.1.2 Parametrisierung des Grid-Files 45
5.2 AUFBAU DES PROTOTYPEN UND DER TESTUMGEBUNG 46
5.2.1 Datensatz hinzufügen 47
5.2.2 Datensatz suchen 48
5.2.3 Trennwertverlegung 49
5.2.4 Grid-Region trennen 50
5.3 ZUFALLSVERTEILUNGEN 51

6 EVALUATION 54
6.1 VERGLEICH MEDIAN UND MITTELWERT 54
6.2 LAST-BALANCIERTES GRID-FILE 60
6.2.1 Vorbereitungen 61
6.2.2 Evaluierungsergebnisse 62
7 ZUSAMMENFASSUNG UND AUSBLICK 70

LITERATURVERZEICHNIS 72
ANHANG 75
A.1 AUFBAU DER AUSWERTUNGSDATEI 75
A.2 AUSGANGSSITUATION NACH EINFÜGEN 76
A.3 VERTEILUNG KONSTANT IN DEN ANFRAGEN ALLER SKALEN, 25. ZYKLUS 77
A.4 VERTEILUNG KONSTANT IN DEN ANFRAGEN ALLER SKALEN, 100. ZYKLUS 78
A.5 VERTEILUNG SÄULE IN DEN ANFRAGEN ALLER SKALEN, 25. ZYKLUS 79
A.6 VERTEILUNG SÄULE IN DEN ANFRAGEN ALLER SKALEN, 100. ZYKLUS 80
A.7 VERTEILUNG GAUSS IN DEN ANFRAGEN ALLER SKALEN, 25. ZYKLUS 81
A.8 VERTEILUNG GAUSS IN DEN ANFRAGEN ALLER SKALEN, 100. ZYKLUS 82
A.9 GEMISCHTE VERTEILUNGEN IN DEN ANFRAGEN ALLER SKALEN, 25. ZYKLUS 83
A.10 GEMISCHTE VERTEILUNGEN IN DEN ANFRAGEN ALLER SKALEN, 100. ZYKLUS 84
A.11 VERTEILUNG KONSTANT IN DEN ANFRAGEN ALLER SKALEN, 25. ZYKLUS 85

Abbildungsverzeichnis

ABB. 2-1: VEREINFACHTES SCHICHTENMODELL EINES DBMS [HARA99] S.15 3
ABB. 2-2: DREIDIMENSIONALER DATENRAUM 7
ABB. 2-3: SCHEMATISCHE DARSTELLUNG DES GRID-FILES (ANLEHNT AN [AP04]) 8
ABB. 2-4: GRID-FILE IMPLEMENTIERUNG NACH [HARA99] S.271 10
ABB. 2-5: ABFRAGE EINES DATENSATZES ÜBER DAS GRID-FILE [NIHI84] 11
ABB. 2-6: TEILUNGSOPERATION 1/4 12
ABB. 2-7: TEILUNGSOPERATION 2/4 12
ABB. 2-8: TEILUNGSOPERATION 3/4 13
ABB. 2-9: TEILUNGSOPERATION 4/4 14
ABB. 2-10: ÄNDERUNGEN AM GRID-DIRECTORY IM BUDDY UND NEIGHBOR SYSTEM [NIHI84] 15
ABB. 3-1: REGELKREIS SELF-TUNING 21
ABB. 3-2: LAST-BALANCIERTER BINÄRBAUM [SASC05] 25
ABB. 4-1: MEDIAN DER DATEN, MITTELWERT DES INTERVALLS 31
ABB. 4-2: PARTIELLES GRID-FILE IM DATENRAUM 32
ABB. 4-3: GRID-FILE MIT VERKETTETEN BUCKETS 34
ABB. 4-4: TRENNWERTLÖSCHUNG 37
ABB. 4-5: BESTIMMUNG DES VERBESSERUNGSWERTES 40
ABB. 5-1: KLASSENDIAGRAMM 46
ABB. 5-2: ZUFALLSVERTEILUNG KONSTANT 51
ABB. 5-3: ZUFALLSVERTEILUNG SÄULE 52
ABB. 5-4: ZUFALLSVERTEILUNG GAUSS 53
ABB. 6-1: VERGLEICH MEDIAN MITTELWERT BIS 100.000 DATENSÄTZE 59
ABB. 6-2: VERGLEICH MEDIAN MITTELWERT BIS 1.000.000 DATENSÄTZE 59
ABB. 6-3: VERGLEICH MEDIAN MITTELWERT BIS 10.000.000 DATENSÄTZE 60
ABB. 6-4: SKALA ZEITSTEMPEL, 0. ZYKLUS, 20 INTERVALLE 63
ABB. 6-5: SKALA ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE 64
ABB. 6-6: VERTEILUNG KONSTANT IN ALLEN SKALEN 64
ABB. 6-7: SKALA ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE 65
ABB. 6-8: VERTEILUNG SÄULE IN ALLEN SKALEN 65
ABB. 6-9: SKALA ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE 66
ABB. 6-10: VERTEILUNG GAUSS IN ALLEN SKALEN 66
ABB. 6-11: VERTEILUNG GEMISCHT 67
ABB. 6-12: VERTEILUNG VARIABEL 68
ABB. A-1: ZEITSTEMPEL, 0. ZYKLUS, 20 INTERVALLE 76
ABB. A-2: ZAHL, 0. ZYKLUS, 20 INTERVALLE 76
ABB. A-3: TEXT, 0. ZYKLUS, 20 INTERVALLE 76
ABB. A-4: ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE, KONSTANT 77
ABB. A-5: ZAHL, 25. ZYKLUS, 20 INTERVALLE, KONSTANT 77
ABB. A-6: TEXT, 25. ZYKLUS, 20 INTERVALLE, KONSTANT 77
ABB. A-7: ZEITSTEMPEL, 100. ZYKLUS, 20 INTERVALLE, KONSTANT 78
ABB. A-8: ZAHL, 100. ZYKLUS, 20 INTERVALLE, KONSTANT 78
ABB. A-9: TEXT, 100. ZYKLUS, 20 INTERVALLE, KONSTANT 78
ABB. A-10: ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE, SÄULE 79
ABB. A-11: ZAHL, 25. ZYKLUS, 20 INTERVALLE, SÄULE 79
ABB. A-12: TEXT, 25. ZYKLUS, 20 INTERVALLE, SÄULE 79
ABB. A-13: ZEITSTEMPEL, 100. ZYKLUS, 20 INTERVALLE, SÄULE 80
ABB. A-14: ZAHL, 100. ZYKLUS, 20 INTERVALLE, SÄULE 80

ABB. A-15: TEXT, 100. ZYKLUS, 20 INTERVALLE, SÄULE 80
ABB. A-16: ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE, GAUSS 81
ABB. A-17: ZAHL, 25. ZYKLUS, 20 INTERVALLE, GAUSS 81
ABB. A-18: TEXT, 25. ZYKLUS, 20 INTERVALLE, GAUSS 81
ABB. A-19: ZEITSTEMPEL, 100. ZYKLUS, 20 INTERVALLE, GAUSS 82
ABB. A-20: ZAHL, 100. ZYKLUS, 20 INTERVALLE, GAUSS 82
ABB. A-21: TEXT, 100. ZYKLUS, 20 INTERVALLE, GAUSS 82
ABB. A-22: ZEITSTEMPEL, 25. ZYKLUS, 20 INTERVALLE, KONSTANT 83
ABB. A-23: ZAHL, 25. ZYKLUS, 20 INTERVALLE, SÄULE 83
ABB. A-24: TEXT, 25. ZYKLUS, 20 INTERVALLE, GAUSS 83
ABB. A-25: ZEITSTEMPEL, 100. ZYKLUS, 20 INTERVALLE, KONSTANT 84
ABB. A-26: ZAHL, 100. ZYKLUS, 20 INTERVALLE, SÄULE 84
ABB. A-27: TEXT, 100. ZYKLUS, 20 INTERVALLE, GAUSS 84
ABB. A-28: ZEITSTEMPEL, 25. ZYKLUS, 16 INTERVALLE, KONSTANT 85
ABB. A-29: ZAHL, 25. ZYKLUS, 16 INTERVALLE, KONSTANT 85
ABB. A-30: TEXT, 25. ZYKLUS, 16 INTERVALLE, KONSTANT 85

Tabellenverzeichnis

TAB. 2-1: KLASSIFIKATION VON ZUGRIFFSVERFAHREN [SAHE05] S.151 4
TAB. 6-1: ERGEBNISSE DES STANDARD GRID-FILES 54
TAB. 6-2: ERGEBNISSE DES STANDARD GRID-FILES BEI OFFENEM DATENRAUM 55
TAB. 6-3: ERGEBNISSE BEI ZUFÄLLIGER EINFÜGEREIHENFOLGE 56
TAB. 6-4: ERGEBNISSE BEI ZUFÄLLIGER EINFÜGEREIHENFOLGE UND OFFENEM DATENRAUM ... 57

1 Einleitung

Datenbanksysteme (DBS) werden in unserer Zeit immer bedeutender. Durch verbesserte Erfassungsmethoden und Digitalisierung steigt die zu verwaltende Datenmenge stetig an. Neue Aufgaben, wie die Analyse der gespeicherten Daten oder die Informationsgewinnung aus diesen im Rahmen des Information Retrieval oder des Data Minings, lassen die Anforderungen an die Datenbanksysteme steigen. Neue Einsatzgebiete wie Web-Datenbanken oder das Cloud-Computing, die eine erhöhte Nutzeranzahl mit sich bringen, führen zu neuen Herausforderungen. Um diesen Aufgaben gewachsen zu sein, widmen sich zahlreiche Wissenschaftler der Erforschung neuer Möglichkeiten zur Verbesserung oder Erweiterung der heutigen Datenbanksysteme.

Der immer größer werdende Funktionsumfang von DBS ermöglicht es weitreichendere Aufgaben zu bewältigen. Die Kehrseite dieses Sachverhaltes ist ein erhöhter Verwaltungsaufwand des Systems, womit die Optimierung durch den Datenbankadministrators schwieriger wird. Die bisherige manuelle Durchführung des Tunings stößt dadurch an seine Grenzen. Neben den bereits genutzten Hilfsprogrammen zur Unterstützung des Datenbankadministrators, muss die Entwicklung in Richtung einer vollständig autonomen Anpassung des Systems gehen. Einige theoretischen Arbeiten sowie die Funktionsweise des Self-Tunings werden in Kapitel 3 näher erläutert.

Ein zentrales Thema des Tunings stellt das Index-Tuning dar. Da Indexe die Verwaltung großer Datenmengen erst ermöglichen, ist ihre Optimierung und Weiterentwicklung eine Notwendigkeit. Ein für umfangreiche mehrdimensionale Daten entwickelter Index ist das Grid-File, welches jedoch den Nachteil eines stark ansteigenden Speicherplatzbedarfes besitzt. Um diesen eingrenzen zu können, ist es notwendig die Gleichbehandlung aller Daten zu beenden. Da Indexe zur Beschleunigung der Nutzeranfragen dienen, erscheint es sinnvoll, die Wichtigkeit der Daten nach diesen auszurichten. Die Schwierigkeit hierin liegt in den ständig wechselnden Anfragemustern, die ein manuelles Index-Tuning unmöglich machen. Die vorliegende Arbeit widmet sich deshalb der Weiterentwicklung des Grid-Files, mit dem Ziel der selbstständigen Anpassung an die jeweilige Anfragesituation.

Um dieses Ziel zu erreichen folgt nach dieser kurzen Einführung ein Überblick über das Grid-File in Kapitel 2. Es werden die einzelnen Komponenten, die Funktionsweise, sowie das Verhalten eingehend erläutert. Kapitel 3 geht im Anschluss auf das Tuning und Self-Tuning ein, wobei sich der Fokus auf das Tuning von Indexstrukturen richtet. In den Unterkapiteln

3.5 und 3.6 werden mit den partiellen Indexen und den last-balancierten Indexen bestehende Konzepte zur Umsetzung eines Index-Self-Tunings beschrieben. Die Anwendung und Umsetzung dieser Konzepte auf das Grid-File werden in Kapitel 4 erläutert. Das Grid-File wird hier weiterentwickelt, um es an das Konzept der Last-Balancierung anzupassen. In Kapitel 5 erfolgen die Beschreibung der prototypischen Implementierung dieser Anpassung sowie die Spezifikation einer Testumgebung. Diese wird in Kapitel 6 genutzt, um das Konzept des last-balancierten Grid-Files zu evaluieren. Eine Zusammenfassung der Arbeit sowie ein Ausblick der weiteren Entwicklung werden in Kapitel 7 dargestellt.

2 Indexstrukturen

In diesem Kapitel werden die Grundlagen sowie Aufbau und Funktionsweise des Grid-Files dargestellt. Eine allgemeine Einführung von Indexstrukturen erfolgt in Kapitel 2.1. Hierbei wird ein grober Überblick gegeben, sowie eine Einordnung des Grid-Files bezüglich anderer Indexe vorgenommen. Anschließend werden in Kapitel 2.2 die Eigenschaften, der Aufbau und die Funktionsweise erläutert. Das letzte Unterkapitel gibt einen Einblick in das Verhalten des Grid-Files.

2.1 Überblick Indexstrukturen

Als Datenbankmanagementsystems (DBMS) bezeichnet man Software zur Verwaltung von einer oder mehreren Datenbanken. Hierbei stellt jedes DBMS mindestens die Grundfunktionalität zur Verfügung, die in den neun Codd‘schen Regeln[1] beschrieben sind. Härder beschreibt den Aufbau eines DBMS in einem Schichtenmodell. In [Ha87] stellt er die fünf modellierten Schichten vor. Jede dieser Schichten stellt einen Teil der Transformation von Anfragen der Anwenderebene zur physischen Speicherebene dar. Abb. 2-1 zeigt ein vereinfachtes Modell, bei dem die Puffer- und Externspeicherverwaltung im Speichersystem integriert wurden.

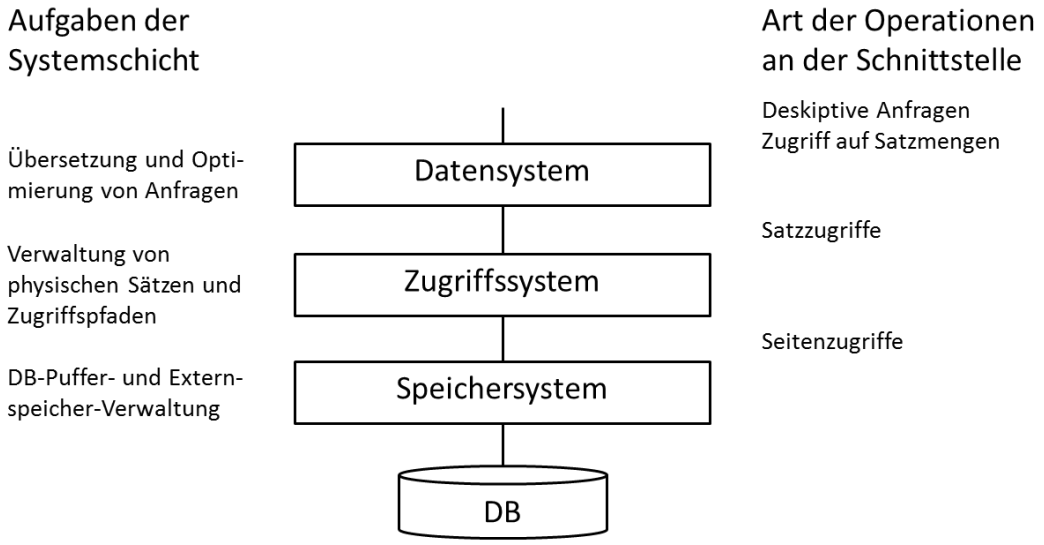

Abb. 2-1: vereinfachtes Schichtenmodell eines DBMS [HaRa99] S.15

[1] Vgl. [SaHe05]

Durch die Aufteilung der Aufgaben eines DBMS in einzelne Schichten ist es möglich diese separat zu untersuchen und zu realisieren. Einen wichtigen Teil des Zugriffssystems jedes DBMS stellen Indexstrukturen oder kurz Indexe dar. Sie ermöglichen eine effiziente Verwaltung und Zugriff auf größere Datenbestände. Hierzu besteht ein Index aus Verweisen auf die Daten. Für die Strukturierung dieser Verweise existieren zahlreiche Methoden, von denen einige im Anschluss angeführt werden. Da der Index zusätzlich zu den Daten angelegt wird, stellt er eine redundante Information dar und muss bei jeder Aktualisierung der Daten ebenfalls aktualisiert werden.

Die Einteilung der verschiedenen Indexstrukturen kann nach mehreren Kriterien erfolgen, z.B. in Primär- und Sekundärschlüsselindexe oder in dünn besetzter und dicht besetzter Index[2]. Die folgende Tabelle aus [SaHe05] zeigt die Klassifikation einer Auswahl an Indexen.

Indexart	**Dimensionalität**	**Verfahrensart**	**Verfahren**
kein Index	eindimensional	direkt	Heap sequentiell
Primärindex	eindimensional	Schlüsselzugriff	indexsequentiell mehrstufig indexsequentiell B-Baum, B^+-Baum
		Schlüsseltransf.	Hashverfahren
	mehrdimensional	Schlüsselzugriff	KdB-Baum Grid-File
		Schlüsseltransf.	Mehrdimensionales Hashen
Sekundärindex	eindimensional	Schlüsselzugriff	Indexiert-nichtsequentiell Mehrstufig indexiert-nichtseq. B-Baum, B^+-Baum-Varianten
		Schlüsseltransf.	Hashverfahren
	mehrdimensional	Schlüsselzugriff	KdB-Baum Grid-File
		Schlüsseltransf.	Mehrdimensionales Hashen

Tab. 2-1: Klassifikation von Zugriffsverfahren [SaHe05] S.151

Wie in der obigen Tabelle zu sehen ist, ist ein weiteres Unterscheidungsmerkmal die Dimensionalität der Indexe. Hierbei wird nach der Anzahl der indizierten Attribute zwischen

[2] Eine eingehende Erläuterung verschiedener Unterscheidungskriterien findet sich in [SaHe05] S.140ff.

eindimensional und mehrdimensional unterschieden. Eindimensionale Indexe sind als die Standardindexe anzusehen, da sie sowohl gut erforscht als auch leicht handhabbar sind. Bekannte Vertreter dieser Gruppe sind der B-Baum und das Hashverfahren[3]. Die Suche nach mehreren Attributen gleichzeitig ist bei eindimensionalen Indexen aufwändig. Die Ergebnisse der einzelnen Indexzugriffe müssen durch Verknüpfung zusammengefasst werden, wodurch ein Overhead an abgefragten Daten entsteht. Mehrdimensionale Indexe vermeiden diesen.

Einen guten Überblick über die Verfahren der mehrdimensionalen Indexstrukturen geben Gaede und Günther in [GaGu98]. Sie zeigen die Entwicklungsgeschichte sowie Ähnlichkeiten und Unterschiede in den Verfahren auf. Das Grid-File ist neben dem KdB Baum und dem mehrdimensionalen Hashen einer der bekanntesten mehrdimensionalen Indexe.

Im Folgenden wird das ursprüngliche Grid-File nach Nievergelt et al. genauer betrachtet. Weiterentwicklungen wie das Two-Level-Grid-File [Hi85] oder das Twin-Grid-File [HuSi88] werden nicht berücksichtigt. Hierauf aufbauend werden in Kapitel 4 einige Veränderungen vorgenommen, um es zu einem last-balancierten Grid-File weiter zu entwickeln.

2.2 Das Grid-File

Das Grid-File wurde das erste Mal in [NiHi84] durch J. Nievergelt, H.Hinterberger und K. C. Sevcik vorgestellt. K. Hinrichs liefert in [Hi85] eine genauere Beschreibung der Implementation. Das Grid-File ist ein anpassbarer, symmetrischer und mehrdimensionaler Index. Anpassbar bedeutet, dass sich die Form des Indexes mit seinem Inhalt ändert. Dadurch sind die durchschnittliche Anfragezeit sowie die durchschnittliche Speicherauslastung gleichverteilt. Dies ist auch gewährleistet, wenn die zu indizierenden Daten höchst uneinheitlich über den Datenraum verteilt sind.

Unter Symmetrie, der zweiten Eigenschaft des Grid-Files, versteht man bei Indexen die Gleichbehandlung aller Merkmale. Diese werden demnach alle wie Primärschlüssel behandelt. Ein anschauliches Gegenbeispiel ist das Telefonbuch. Die Suche nach einem bestimmten Namen geht sehr schnell, da alle Werte nach diesem sortiert sind. Hat man aber eine Telefonnummer und möchte den Anschlussinhaber ermitteln, müssen im schlimmsten Fall alle Einträge durchsuchen werden, da keine Ordnung für dieses Attribut existiert. Das Telefonbuch ist asymmetrisch.

[3] Vgl. [SaHe05] S.163 ff. und [ElNa02]

Die letzte Eigenschaft des Grid-Files ist die Mehrdimensionalität. Wie im vorherigen Abschnitt erläutert, wird hierunter der Zugriff über mehrere Attribute gleichzeitig verstanden. In einem Index mit Name, Alter und Adresse wäre diese Eigenschaft gegeben, wenn sowohl Anfragen über alle Attribute als auch über eine Kombination aus diesen vom Index unterstützt wird.

Bei der Entwicklung des Grid-Files wurden zwei Prinzipien beachtet: Das ‚Two-Disk-Access' Prinzip und das Prinzip ‚Efficient Range Queries with Respect to all Attributes'. Nach dem 'Two-Disk-Access' Prinzip muss ein gesuchter Wert spätestens nach zwei Sekundärspeicherzugriffen gefunden werden. Das Prinzip 'Efficient Range Queries with Respect to all Attributes' sagt aus, das Datensätze, deren Werte ähnlich sind, auch auf dem physischen Speicher eng beieinander liegen sollen. Auf diese Weise können Bereichsanfragen schneller bearbeitet werden.

2.2.1 Unterteilung des Datenraumes

Alle Indexstrukturen lassen sich in zwei große Kategorien einordnen. Indexe, die die zu speichernden Daten organisieren und Indexe, die den umgebenen Datenraum strukturieren. Im Gegensatz zu Binärbäumen, die zur Speicherung der Grenzen des Datenraumes die zu indizierenden Daten nutzen[4], gehört das Grid-File zu den Indexstrukturen, die den Datenraum organisieren und unterteilen. Hierbei wird durch feststehende Werte, die im Laufe des Lebenszyklus der Daten nicht ständig geändert werden müssen, der Datenraum unterteilt.

Um dies zu verdeutlichen, sei ein dreidimensionalen Datenraum D mit D = X * Y * Z gegeben, indem eine Gridpartition G gelegt wird. Die Gridpartition G wird aus den Intervallen U, V und W gebildet wobei gilt:

$$G = U * V * W$$

mit

$$U = \{u_0, u_1, \ldots, u_l\}$$

$$V = \{v_0, v_1, \ldots, v_m\}$$

$$W = \{w_0, w_1, \ldots, w_n\}$$

[4] Zu genaueren Erläuterungen von Binärbäumen siehe [Se06] S. 57 ff.

Jeder Unterabschnitt von U, V und W wird durch einen Trennwert gebildet, der die gesamte Gridpartition in einer Dimension teilt. Durch diese Unterteilung entstehen zahlreiche Grid-Blöcke. Das wird in Abbildung 2-2 veranschaulicht.

Im Laufe der Zeit sind Veränderungen an der Aufteilung der Gridpartiton notwendig, da sich durch einfügen und löschen von Datensätzen die Struktur der zugrunde liegenden Daten ändert. Hierfür gibt es Misch- und Trennoperationen, die neue Trennwerte einfügen oder zwei benachbarte Ebenen zusammenfügen und so einen Trennwert löschen können. Im unteren Bild wird der Bereich v_1 durch einen neuen Trennwert geteilt. Dadurch entstehen die Bereiche v_1 und v_2. Zu beachten ist, dass alle Indizes oberhalb des Trennwertes um eins erhöht werden. Im gegenteiligen Fall, dem Zusammenlegen der Bereiche v_1 und v_2, entsteht ein größerer Bereich v_1 und alle Indizes von V oberhalb des Trennwertes werden um eins vermindert.

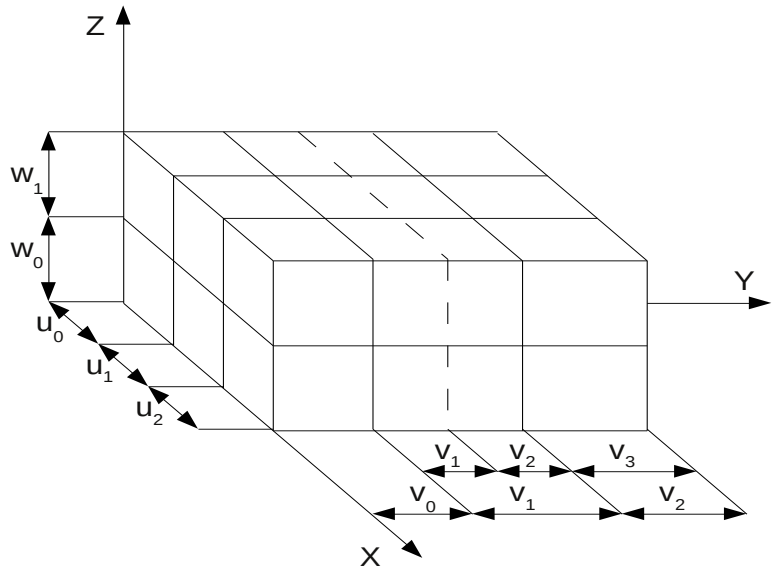

Abb. 2-2: dreidimensionaler Datenraum

Ein Problem stellen Attribute dar, die voneinander abhängig, das heißt korreliert, sind. Es entstehen entartete Strukturen, da die Daten zum Beispiel diagonal im Datenraum liegen. Somit wird eine Teilung des Datenraumes erzeugt, die nicht optimal ist. Bei Auswahl solcher Attribute entstehen zahlreiche leere Blöcke. Hier ist eine geeignete Auswahl an Attributen anzuraten oder es sind Verfahren der Datenanalyse zu verwenden, die diese Attribute z.B. durch Datentransformation eliminieren (Hauptkomponentenanalyse, Faktorenanalyse[5]).

[5] Zur genaueren Beschreibung der Verfahren vgl. [FaHa96]

2.2.2 Aufbau des Grid-Files

Das Grid-File besteht aus drei Komponenten, dem Grid-Array, den Skalen und den Buckets. Das Grid-Array bildet zusammen mit den Skalen das Grid-Directory. Im Folgenden werden die Komponenten einzeln betrachtet und ihr Zusammenwirken beschrieben. Abb. 2-3 verdeutlicht den Aufbau des Grid-Files, sowie die einzelnen Komponenten.

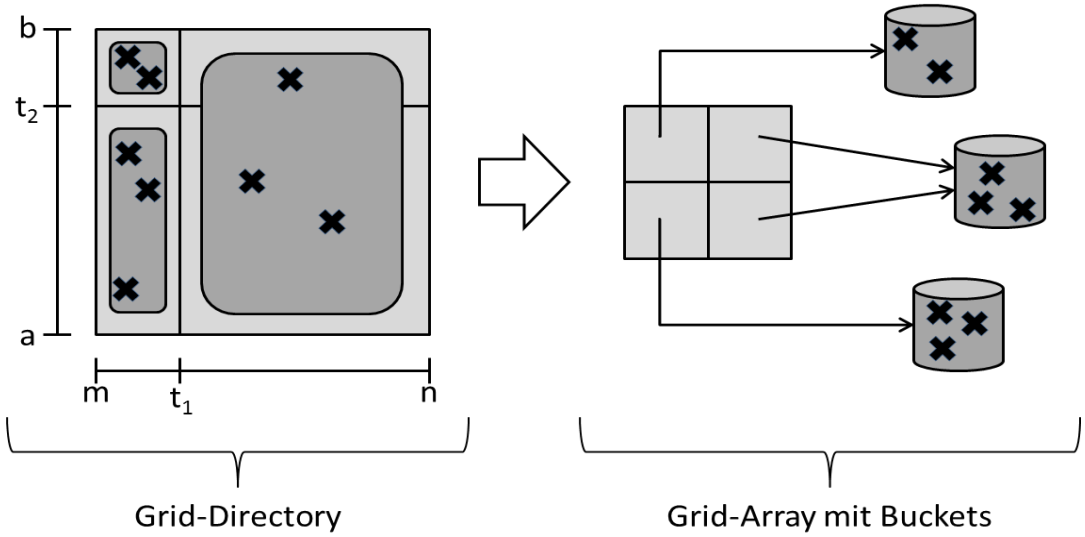

Abb. 2-3: Schematische Darstellung des Grid-Files (anlehnt an [Ap04])

Buckets

Die Buckets dienen der Speicherung der eigentlichen Daten oder Verweisen auf diese. Sie werden im oberen Bild als Zylinder dargestellt. Die Speicherung erfolgt in einer ungeordneten Liste, die eine Größenbeschränkung c besitzt. Es ist sinnvoll eine Größe zu wählen, die ein komplettes Durchsuchen des Buckets in kürzester Zeit ermöglicht. Nievergelt et al. haben in ihren Ausführungen hierfür eine Größe zwischen 10 und 1000 Datensätzen genannt. Aufgrund der Tatsache, dass das Bucket physisch gesehen eine Speicherseite darstellt und die Kapazitäten von Rechnern in den letzten 20 Jahren stetig zugenommen haben, wäre eine Überprüfung dieser Annahme sinnvoll, da besonders in den letzten Jahren eine Portierung auf 64 Bit Systeme stattgefunden hat, die die Handhabung größerer Speicherbereiche erleichtert.

Bei direkt vorliegenden Daten in den Buckets, ist es schwer möglich einen weiteren Index, z.B. über ein nicht berücksichtigtes Attribut zu erzeugen. Speichern die Buckets Verweise auf die Datensätze, muss für jeden Verweis auf die eigentliche Datenbank zugegriffen werden. Alternativ lässt sich eine Mischform bilden, bei der zusätzlich zum Verweis auf die

Datenbank, die indizierten Werte hinterlegt werden. Dies führt zwar zu einer Redundanz[6], beschleunigt aber die Anfragen.

Grid-Array

Das Grid-Array ist ein k-dimensionales Array. Hierbei besitzt jedes Attribut eine eigene Dimension. Die einzelnen Zellen verweisen auf die Buckets. Bei zwei oder mehr Grid-Zellen, die auf das selbe Bucket verweisen, werden diese logisch zusammengefasst und als Grid-Region bezeichnet. Hierbei wird die Forderung erhoben, dass die Regionen k-dimensionale Quader sind, d.h. dass sie konvex[7] sein müssen.

Skalen

Die Skalen sind eindimensionale geordnete Listen, die die Trennwerte enthalten. Sie bilden die Zuordnung der Werte zum Grid-Array. Für jedes indizierte Attribut wird eine eigene Skala angelegt, d.h. bei zwei Attributen existieren zwei Skalen, bei drei, drei Skalen usw. Da sie kaum Speicherplatz verbrauchen und bei jedem Zugriff auf das Grid-File benötigt werden, ist es sinnvoll sie im Hauptspeicher zu halten. In Abb. 2-3 sind im linken Teilbild zwei Skalen dargestellt, wobei die erste Skala die Werte von m bis n annehmen kann und die zweite Skala Werte von a bis b beinhaltet. t_1 und t_2 sind Trennwerte, wobei gilt $a < t_2 < b$ und $m < t_1 < n$. Ein Trennwert muss demnach immer innerhalb des Intervalls der Skala liegen.

Grid Directory

Das Grid-Directory ist eine logische Struktur, die sich aus den Skalen und dem Grid-Array zusammensetzt. Die Anzahl der indizierten Attribute bestimmt sowohl die Dimension des Arrays, als auch die Anzahl der Skalen. Ein Grid-File mit k Attributen besitzt demnach ein k-dimensionales Grid-Array mit k Skalen.

Eine Alternative zur direkten Implementierung als sortiertes Array wird in [HaRa99] beschrieben. Hierbei ist des Grid Directory ein Array welches an den Rändern wächst und schrumpft. Dadurch entfallen die Verschiebungsoperationen innerhalb des Directorys. Dies führt zu einer Anpassung der Skalen, deren Implementierung nicht mehr als sortierte Listen

[6] Die Redundanz führt bei einem Index zu keinem Fehler in der Datenhaltung, da bei jeder Änderung der eigentlichen Daten der Index ebenfalls angepasst werden muss.
[7] Konvex bedeutet, dass die Strecke zwischen zwei Punkten des Körpers innerhalb des Körpers liegt.

möglich ist. Stattdessen werden doppelt verkettete Listen verwendet, die einen Verweis auf den entsprechenden Eintrag im Directory enthalten. Abb. 2-4 verdeutlicht dies. Diese Implementierung ist effizienter in Bezug auf Erweiterungen und Verkleinerungen des Grid-Arrays, da die Verschiebeoperationen innerhalb des Arrays entfallen. Für Bereichsanfragen, bei denen der Zugriff auf die nächste Nachbarzelle erfolgen soll, ist sie von Nachteil. Für jede Grid-Zelle ist eine separate Anfrage an das Grid-Directory notwendig.

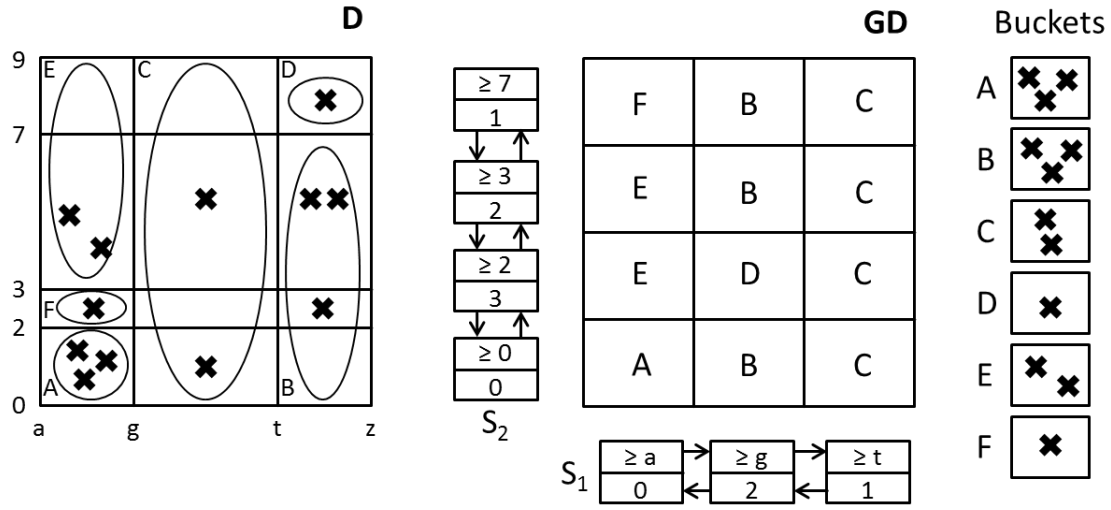

Abb. 2-4: Grid-File Implementierung nach [HaRa99] S.271

2.2.3 Funktionen des Grid-Files

Für das Grid-File sind die folgenden Funktionen definiert:

- Zugriff auf eine Zelle des Grid-Directorys
- Zugriff auf die nächste Zelle in irgendeine Richtung
- Eine Teilungsstrategie
- Eine Mischstrategie

Der Zugriff auf eine Zelle des Grid-Directorys ist in Abb. 2-5 dargestellt. Hierbei wird eine Punktanfrage für den Datensatz [1980, w, …] gestellt. Zunächst werden die Werte in die Skalen eingeordnet. Da es sich um geordnete Listen handelt, ist hierbei nur solange zu suchen, bis der Wert der Skala größer als der Anfragewert wird. Der Index des zuvor gefundenen Wertes stellt ebenfalls den Index in der entsprechenden Dimension des Arrays dar. Somit wird nach dem einordnen in alle Skalen direkt auf die entsprechende Grid-Zelle zugegriffen. Die Grid-Zelle verweist auf ein Bucket, in dem der gesuchte Wert entweder enthalten oder

nicht enthalten ist. Das Bucket, als unsortierte Liste, muss in diesem Fall komplett durchsucht werden.

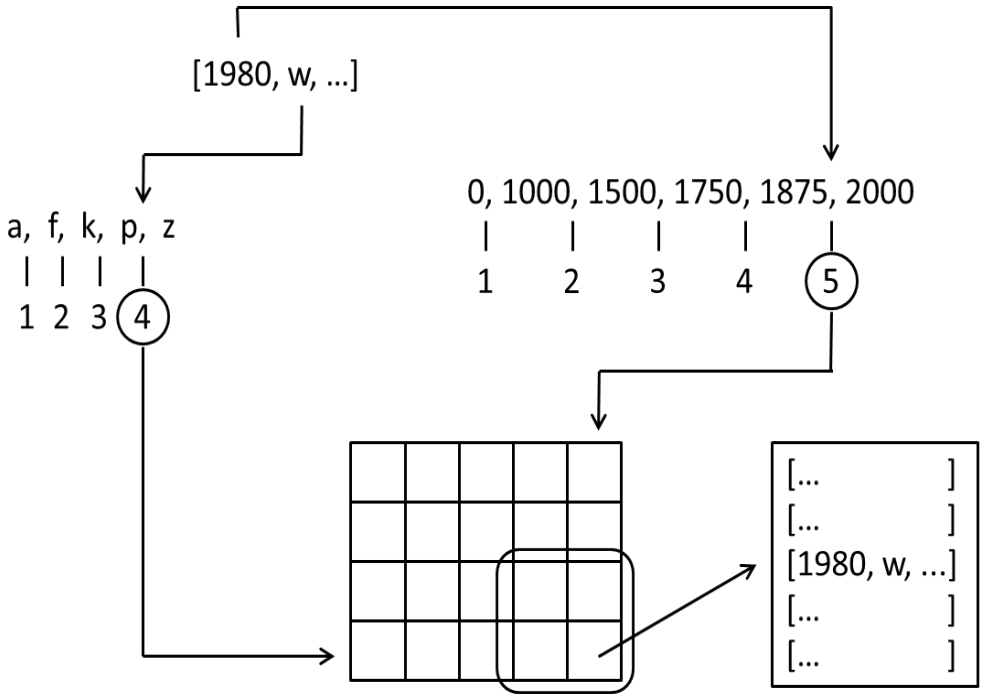

Abb. 2-5: Abfrage eines Datensatzes über das Grid-File [NiHi84]

Die Forderung nach einem Zugriff mit höchstens zwei Sekundärspeicherzugriffen ist mit diesem Aufbau erfüllt. Da die Skalen im Hauptspeicher gehalten werden, erfolgt der erste Zugriff, um die gesuchte Zelle des Grid-Arrays zu erhalten und der zweite, um auf das entsprechende Bucket zuzugreifen.

Das Auffinden der nächsten Zelle ist in einem geordneten Grid-Array einfach. In diesem Fall wird der entsprechende Index der Grid-Zelle um eins erhöht oder vermindert. Bei einer abweichenden Implementierung muss dies berücksichtigt werden. Im Beispiel der alternativen Implementierung des Grid-Directorys (Abb. 2-4) muss erneut über die Skalen auf das Grid-Array zugegriffen werden, um die Nachbarzelle zu ermitteln.

Teilungsstrategie

Die Teilungs- und Mischoperationen sind sehr stark von der jeweiligen Implementierung abhängig. Aus diesem Grund werden im Folgenden nur ein schematischer Ablauf der Teilungsoperation sowie die Möglichkeiten für die Mischoperationen erläutert.

Die Teilung des Datenraumes wird anhand mehrerer Schaubilder erläutert. Abb. 2-6 zeigt einen Datenraum mit drei Datensätzen. Die Bucketgröße beträgt ebenfalls drei, so dass für alle Daten ein Bucket ausreicht. Das Grid-Array besteht somit nur aus einer Zelle.

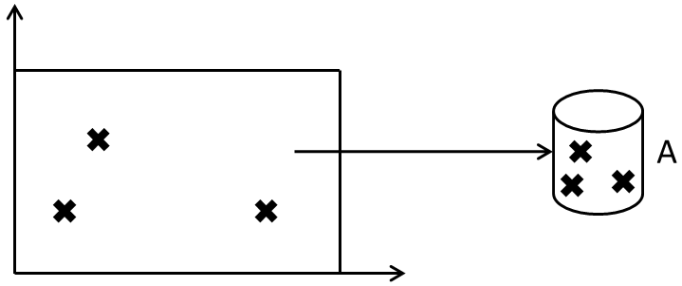

Abb. 2-6: Teilungsoperation 1/4

Wird ein Datensatz hinzugefügt, führt das zu einem Bucketüberlauf. Das Grid-Directory muss durch einen neuen Trennwert in zwei Zellen getrennt werden. Als Standard für die Festlegung eines Trennwertes gilt der Mittelwert des zu trennenden Intervalls. Dieser Wert ist leicht zu berechnen, da die benötigten Intervallgrenzen in den Skalen vorhanden sind. Die Teilung in der Mitte des Intervalls kann allerdings zu mehreren Teilungen hintereinander führen. Dies geschieht, wenn alle Datensätze auf einer Seite des neuen Trennwertes liegen. In diesem Fall wird solange geteilt bis genug Platz für den neuen Datensatz vorhanden ist. Um den Trennwert gezielter ermitteln zu können, wird in Kapitel 4.3 die Möglichkeit untersucht den Trennwert anhand des Medians der Daten festzulegen. Nachdem das Grid-Directory getrennt worden ist, wird ein neuer Bucket erzeugt und die Daten bezüglich des Trennwertes auf beide aufgeteilt. Abb. 2-7 zeigt das Endergebnis dieser Operation.

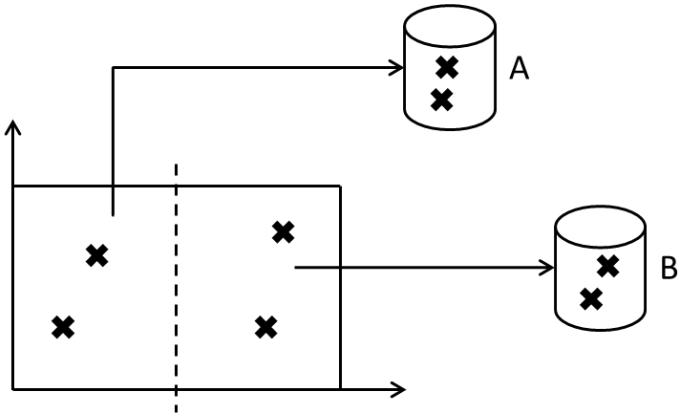

Abb. 2-7: Teilungsoperation 2/4

In Abb. 2-8 wurden zwei weitere Datensätze eingefügt, wodurch erneut Bucket A überläuft. Hierdurch wird eine weitere Trennung ausgelöst. Aufgrund der Teilungsstrategie jedoch dieses Mal in der anderen Dimension. Die Teilungsstrategie sieht vor, alle Dimensionen gleichmäßig zu trennen, d.h. die Größe der entsprechenden Skalen wächst gleichmäßig. Die Bevorzugung eines Attributes kann bei einer höheren Granularität desselben sinnvoll sein.

Durch das Einfügen des neuen Trennwertes wurde ebenfalls die Grid-Zelle getrennt, die auf Bucket B verweist. Da dieses Bucket aber ausreichend Platz für die Datensätze beider Zellen bietet, wird keine Trennung vorgenommen. Die Zellen bilden somit eine Grid-Region.

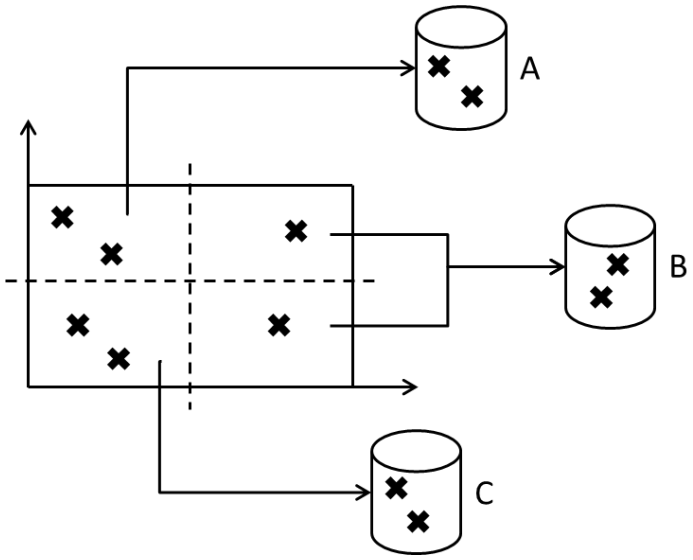

Abb. 2-8: Teilungsoperation 3/4

Nach dem Einfügen weiterer Daten ist Bucket C übergelaufen, so dass hier eine erneute Teilung des Grid-Directorys stattgefunden hat. Es zeigen nun zwei Zellen auf Bucket A. Des Weiteren führten neue Datensätze in Bucket B zu einem Überlauf. Da für dieses Bucket eine Grid-Region existiert, die aus mehreren Grid-Zellen besteht, wird keine Teilung des Grid-Directorys eingeleitet. Stattdessen wird ein vorhandener Trennwert bestimmt, der die Grid-Region des Buckets B in zwei Regionen aufteilt. Eine der neuen Regionen verweist auf einen neu erzeugten Bucket E und die Daten werden entsprechend aufgeteilt. In Abb. 2-9 ist das Endergebnis dargestellt.

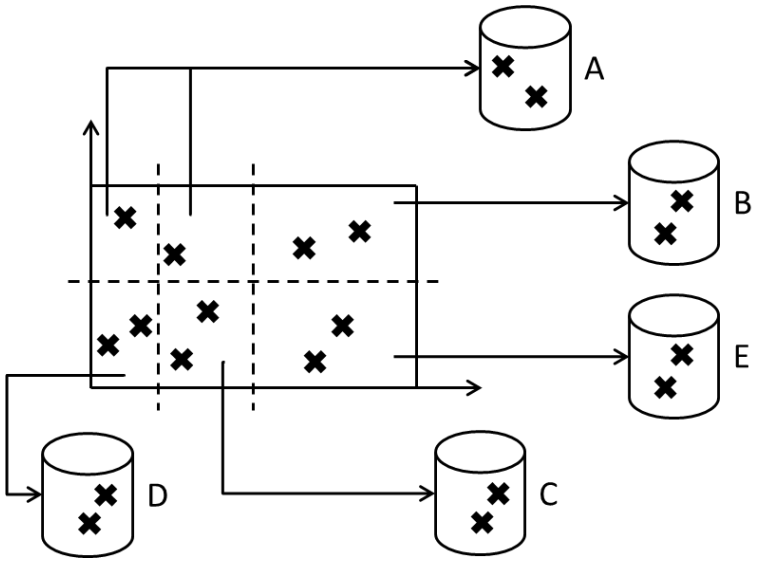

Abb. 2-9: Teilungsoperation 4/4

Zusammenfassend gibt es bei der Teilungsoperation demnach zwei Möglichkeiten. Erstens: Die Grid-Region, dessen Bucket überläuft, besteht aus nur einer Grid-Zelle. In diesem Fall muss ein neuer Trennwert gebildet und das Grid-Directory erweitert werden. Zweitens: Die Grid-Region besteht aus mehreren Grid-Zellen. Hier wird ein bestehender Trennwert zur Teilung der Grid-Region genutzt.

Mischstrategie

Die Mischstrategie muss zwei unterschiedliche Bereiche abdecken. Zum einen das Mischen von zwei benachbarten Ebenen des Grid-Directorys und zum anderen das Zusammenlegen von zwei Regionen bzw. Buckets zu einer. Die erste Variante ist nach Nievergelt et al. in den meisten Fällen unnötig, da sich dieses nur bei einem schrumpfenden Grid-File lohnen würde. Bei einem stetigen oder gar wachsenden Grid-File ist die Wahrscheinlichkeit, dass zeitnah ein neuer Trennwert eingefügt werden muss, sehr groß. Um zwei Ebenen des Grid-Directorys mischen zu können, müssen diese denselben Inhalt haben. Dies bedeutet, dass die Grid-Zellen, die durch den Trennwert geteilt sind zu einer Grid-Region gehören müssen.

Die zweite Variante, das Mischen von zwei Grid-Regionen, ist wesentlich sinnvoller. Für diesen Fall gibt es zwei Systeme, das buddy system und das neighbor system. Beim buddy system kann eine Grid-Region nur mit einer anderen Grid-Region verbunden werden, die vorher direkt von dieser getrennt wurde. Dies hat den Vorteil, dass immer konvexe Regionen entstehen. Beim neighbor system sind alle Nachbarregionen zulässig, solange eine konvexe Region gebildet wird. Dies führt dazu, dass das neighbor system die Mischoptionen des buddy

systems enthält, aber einen höheren Verwaltungsaufwand verursacht. Abb. 2-10 stellt beide Systeme nebeneinander dar.

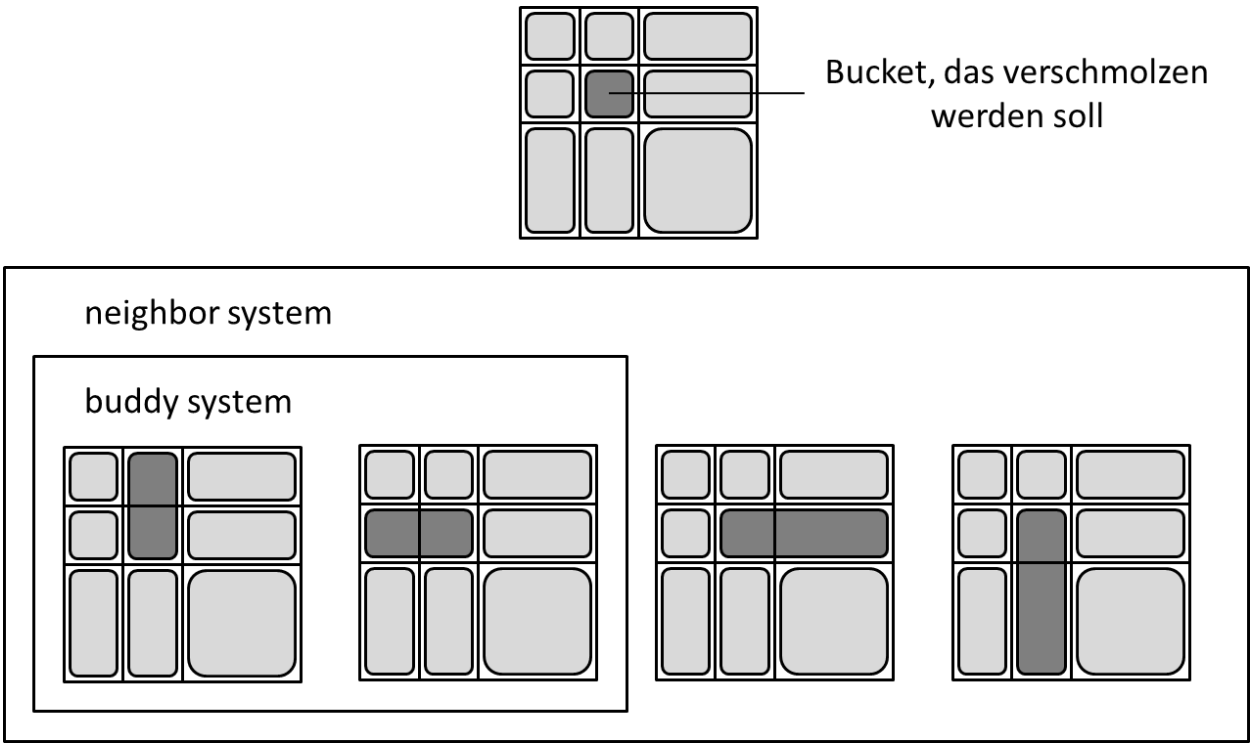

Abb. 2-10: Änderungen am Grid-Directory im buddy und neighbor system [NiHi84]

2.2.4 Verhalten des Grid-Files

Nievergelt et al. haben neben dem Aufbau des Grid-Files, auch dessen Verhalten analysiert und ebenfalls in [NiHi84] beschrieben. Zur Ermittlung des Verhaltens wurden mehrere Simulationen durchgeführt. Dabei wurde zwischen drei Fällen unterschieden: einem wachsenden, einem gleich groß bleibenden und einem schrumpfenden Grid-File. In allen drei Fällen wurden die Bucketauslastung und die Größe des Grid-Directorys gemessen, sowie eine Darstellung der Verteilung der Grid-Regionen ermittelt. Des Weiteren wurden die Teilungs- und Mischstrategien ausgewertet.

Das wachsende Grid-File wurde simuliert, indem 10000 Datensätze in ein leeres Grid-File eingefügt und zwei Bucketgrößen c = 50 und c = 100 getestet wurden. Hierbei stellte sich heraus, dass sobald die Anzahl der Grid-Zellen n ein Mehrfaches der Bucketgröße erreicht hat, die Bucketauslastung bei einem Wert von circa 69% (ln 2) schwankt. Eine Alternative zur Trennung in zwei Hälften bildet die Drittelung der Grid-Zelle. Dies ist allerdings nicht empfehlenswert, da die Bucketauslastung auf 39% absinkt.

Die konstante Bucketauslastung sagt jedoch nichts über die Anzahl der Grid-Zellen und somit über das Wachstum des Grid-Directorys aus, da ein Bucket durch mehrere Grid-Zellen repräsentiert werden kann. Besonders korrelierte Attribute haben einen vergrößernden Effekt auf das Grid-Directory, wogegen die Bucketauslastung nicht betroffen ist. Ebenfalls ist das Grid-Directory nicht immun dagegen, dass sich alle Datensätze in einem kleinen Areal befinden. Hierbei entstehen durch die Teilung des gesamten Grid-Directorys durch Hyperebenen zahlreiche Grid-Zellen, die auf bereits bestehende Buckets zeigen und zur Trennung der Daten nicht benötigt werden.

Die Größe des Grid-Directorys wächst in der Simulation wie erwartet stärker als die Anzahl der Buckets. Während sich die Bucketanzahl allerdings nahezu linear erhöht, weist die Anzahl der Grid-Zellen ein lineares Wachstum mit wesentlich höherer Amplitude und Periode auf. Dieses „stufenweise Wachstum“ lässt sich anschaulich damit erklären, dass zu bestimmten Zeitpunkten die Grid-Zellen ungefähr die gleiche Größe aufweisen und mit einem eigenen Bucket belegt sind. Unter der Bedingung gleichverteilter Daten kommt es nun zu einer Trennung vieler Grid-Zellen in kurzer Zeit. Danach können etliche Datensätze eingefügt werden, bis erneut die obige Situation eintritt, wobei das Grid-Directory nun doppelt so groß ist.

Es bleibt festzuhalten, dass das Wachstum des Grid-Directorys direkt von der Bucketkapazität c abhängt. Hierbei gilt, dass je kleiner die Bucketgröße, umso größer wird das Grid-Directory. Extreme Werte ergeben sich bei niedrigen Bucketgrößen von $c = 1$ bzw. $c = 2$, weshalb eine Größe von mindestens $c = 10$ als sinnvoll zu erachten ist.

Während das wachsende Grid-File ein guter Test für die Teilungsstrategie ist, ist beim gleichgroß bleibenden Grid-File die Interaktion zwischen Teilungs- und Mischstrategie gut zu beobachten. In der Simulation wurden 5000 Datensätze in ein leeres Grid-File mit einer Bucketgröße von $c = 16$ eingefügt. Anschließend wurden 5000 Datensätze je zur Hälfte angelegt oder gelöscht. Der Schwellwert zum Mischen[8] zweier Buckets wurde variiert und das buddy system verwendet.

Aus einem Schwellwert von 100% resultiert eine durchschnittliche Bucketauslastung von 70%, wohingegen sich bei einem Schwellwert von 50% die durchschnittliche Auslastung auf 60% verringert. Dem gegenüber steht die Anzahl der Teilungs- und Mischoperationen, die bei

[8] Der Schwellwert beim Mischen ist ein prozentualer Wert, der die Bucketauslastung des resultierenden Buckets angibt. Dieser Wert darf nicht unterschritten werden.

hohen Schwellwerten über 80% stark ansteigt, da die zusammengefügten Buckets zu voll sind und so zeitnah wieder getrennt werden müssen. Daher wird ein Schwellwert zum Mischen von 70% empfohlen.

Das schrumpfende Grid-File testet allein die Mischstrategie. Die Simulation mit 5000 zu löschenden Datensätzen mit dem buddy system ergab, dass ein Schwellwert von 100% die Bucketauslastung anfänglich stabil hält. Danach fällt dieser aber kontinuierlich. Im neighbor system bleibt die Bucketauslastung dagegen konstant.

3 Tuning und Self-Tuning

Nach einer kurzen Einführung in das Tuning (Kapitel 3.1) wird in Kapitel 3.2 auf das Indextuning eingegangen. Besonders das Problem der Indexselektion spielt hier eine Rolle. In Kapitel 3.3 und 3.4 werden das Self-Tuning sowie das Index Self-Tuning näher behandelt. Im Anschluss wird in Kapitel 3.5 das Konzept der partiellen Indexe vorgestellt. Eine weitere Verknüpfung zwischen dem Self-Tuning und Indexen erfolgt in Kapitel 3.6, in dem die lastbalancierten Indexstrukturen vorgestellt werden.

3.1 Tuning

Datenbanktuning ist seit geraumer Zeit Bestandteil der Forschung. DBMS müssen mit ihrer enormen Vielfalt an Einstellungsmöglichkeiten an die verschiedenen Situationen in der Praxis angepasst werden. Hierbei ist das Tuning als Verbesserung und Anpassung an die jeweilige Arbeitsumgebung mit dem Ziel eines Gewinnes an Performance nötig.

Um eine Lösung dieses Optimierungsproblems zu ermöglichen, wäre ein mathematisches Modell nötig. Dieses ist allerdings aufgrund der Komplexität und der damit einhergehenden Vielfalt an Parametern nicht verfügbar. Es bleibt nur die Annäherung an ein möglichst optimales Verhalten. Ein Datenbanktuning ist somit stark von der Erfahrung und dem Wissen des Administrators abhängig. Deshalb haben Shasha und Bonnet für das Tuning von DBMS in [ShBo03] fünf Grundprinzipien aufgestellt:

- Think globally, fix locally
- Partitioning breaks bottlenecks
- Start-up costs are high, running costs are low
- Render unto server what is due unto server
- Be prepared for trade-offs

Eine Orientierung an diesen Prinzipien führt nicht immer zum gewünschten Ergebnis. Es erhöht allerdings die Wahrscheinlichkeit dieses zu erreichen. Im Folgenden wird ein Teilbereich des Tunings behandelt, das Indextuning.

3.2 Indextuning

Indexe sind in der Lage den Zugriff auf die Daten der Datenbank stark zu beschleunigen. Bei einer ungünstigen Auswahl haben sie keinen oder sogar einen negativen Effekt auf die

Ausführungszeit. In dieser Situation verbrauchen sie Speicherplatz und Rechenkapazität ohne einen Nutzen zu bringen. Daher ist das Indextuning von enormer Wichtigkeit für die Performance eines Datenbanksystems.

Das Ziel des Indextunings ist es, eine Auswahl von Indexen zu bestimmen, die mit den vorhandenen Ressourcen[9] eine größtmögliche Steigerung der Performance bewirkt. Die Konfiguration der Indexe wird auch als Index-Selection-Problem (ISP) bezeichnet. Hierbei müssen mehrere Faktoren berücksichtigt werden:

- Die Art des Indexes
- Welche Attribute indiziert werden sollen
- Die Abhängigkeit von anderen Indexen[10]

Die Art des Indexes (z.B. B-Bäume, Hash Tabellen) bestimmt den Einsatzzweck, denn nicht alle Indexe können alle Datenbankoperationen unterstützen. So sind z.B. Hash Funktionen gut für Punktanfragen geeignet, für Bereichsanfragen sind sie dagegen nutzlos.

Um eine möglichst optimale Konfiguration an Indexen zu erhalten, werden sogenannte Workloads benutzt. Dies sind Sammlungen von Aufgaben und Anfragen an das DBMS, die das Nutzungsverhalten repräsentieren. Sie können während des laufenden Betriebes aufgezeichnet oder künstlich erzeugt werden. Die Annahme hinter dieser Analyse ist, dass der Workload nicht nur das vergangene Verhalten darstellt, sondern auch das zukünftige Nutzungsverhalten abbildet. In [LüSa07] ist das ISP mathematisch wie folgt dargestellt:

Anfragemenge: $Q_1, \ldots, Q_m$

Indexkandidaten: $I_1, \ldots, I_n$

mit Verwaltungskosten des Indexes: $mcost\ (I_i)$

und Größe des Indexes: $size\ (I_i)$

Der Gewinn ergibt sich aus der Differenz der Anfrage mit und ohne Index:

$$profit\ (Q_k, I_i) = max\ \{0,\ cost\ (Q_k) - cost\ (Q_k, I_i)\}$$

Es wird eine Konfiguration von Indexen $C = \{I_1, \ldots, I_n\}$ gesucht, die für die gegebene Anfragemenge den Gewinn unter Berücksichtigung der Verwaltungskosten der Indexe maximiert:

[9] Diese können durch Restriktionen, wie z.B. einer festgelegten Menge an Speicherplatz, begrenzt sein
[10] Indexe können sich gegenseitig unterstützen oder sogar behindern

$$\sum_{i=1}^{m} max\{profit(Q_i, I_j) : I_j \in C\} - \sum_{I_j \in C} mcost(Ij)$$

Weiterhin muss die Nebenbedingung für die Größenbeschränkung S beachtet werden:

$$\sum_{I_j \in C} size(I_j) \leq S$$

Hierbei handelt es sich um eine Variante des Rucksackproblems, bzw. der ganzzahligen linearen Optimierung. Dieses ist durch ein Greedy Verfahren oder genetische Algorithmen approximativ lösbar. Aktuelle DBMS bieten Programme an, die den Workload automatisch auswerten und eine Indexkonfiguration vorschlagen. Diese „Advisor“ finden sich sowohl im SQL-Server von Microsoft (Database Engine Tuning Advisor), in DB2 von IBM (Design Advisor) als auch in Oracle (SQL Advisor). Die Umsetzung des Vorschlages obliegt ebenso wie die Reaktion auf Veränderungen im Nutzungsverhalten weiterhin dem Datenbankadministrator.

3.3 Self-Tuning

Das Tuning, wie es zu Beginn des Kapitels beschrieben wurde, stößt zusehends an seine Grenzen. Neben den hohen Personalkosten für gut ausgebildete Spezialisten, lassen die Vielzahl an zu berücksichtigen Parametern und die Mehrschichtigkeit der DBMS die Komplexität immer weiter ansteigen. Aus diesen Gründen führen Chaughuri und Weikum in [ChWe06] an, dass die Systeme autonom funktionieren sollten. Zu diesen wünschenswerten Eigenschaften gehört neben dem Self-Management, dem Self-Monitoring und Self-Healing auch das Self-Tuning.

Unter Self-Tuning wird die automatische Anpassung an äußere Einflüsse mit dem Ziel der Leistungsverbesserung des Systems verstanden. Einen Überblick über die Entwicklung des Self-Tunings von Datenbanken im letzten Jahrzehnt geben Chaughuri und Narasayya in [ChNa07].

Allgemein kann das Self-Tuning in drei Abschnitte unterteilt werden, die kontinuierlich nacheinander ablaufen. Dieser Regelkreis wurde im Rahmen des COMPFORT Projektes von Weikum et al. entwickelt. Zu Beginn wird das System beobachtet (Observation). In dieser Phase werden Statistiken über das Verhalten des Systems angelegt. Hieraus werden in der

zweiten Phase (Prediction) Änderungen des Systemzustandes abgeleitet, die eine Optimierung ermöglichen. Die dritte Phase (Reaction) führt die Veränderungen am System zu einem geeigneten Zeitpunkt aus[11]. Abbildung 3-1 verdeutlicht dies.

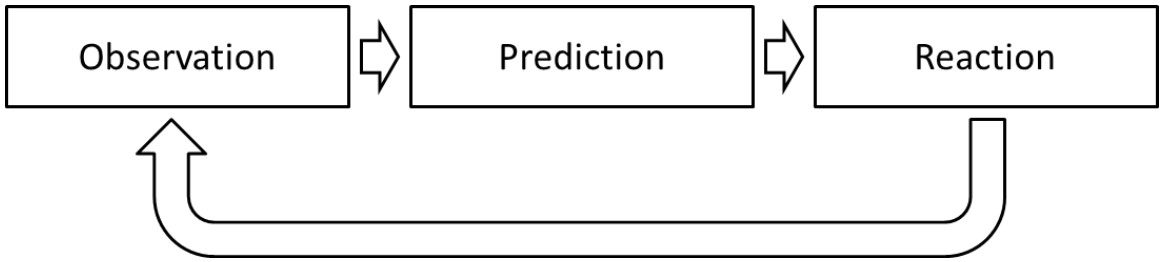

Abb. 3-1: Regelkreis Self-Tuning

Eine Alternative zum Regelkreis stellt das MAPE System von IBM dar. MAPE steht hierbei für Monitor, Analyse, Plan und Execute. Diese vier Phasen werden nacheinander durchlaufen. Die erste Phase Monitor entspricht der Observation Phase im Regelkreis. Die gewonnenen Daten werden analysiert (Phase 2) und es wird entschieden, ob eine Änderung der Parameter nötig ist. Ist dies der Fall, wird in Phase 3 (Plan) ein Aktionsplan erstellt, der autonom in Phase 4 durchgeführt wird. Alle Phasen von MAPE haben Zugriff auf eine Wissensdatenbank, die Informationen und Lösungskonzepte bereitstellt [Ib06].

3.4 Self-Tuning von Indexen

Eine Umsetzung des Self-Tunings für Indexe wurde mit dem QUIET System erstellt . Dieses wurde von K. Sattler, I. Geist und E. Schallehn in [SaGe03] und [SaSc04] vorgestellt und als Erweiterung von DB2 konzipiert. QUIET beobachtet das System, entscheidet selbständig aufgrund eines Kostenmodells welche Indexe anzulegen bzw. zu löschen sind und führt dies aus.

Das Kostenmodell von QUIET basiert auf dem obig beschriebenen Kostenmodell des ISP. Bei diesem wird die Indexkonfiguration allerdings nur einmal erzeugt und für den weiteren Betrieb unverändert zur Verfügung gestellt. Im Gegensatz dazu ist beim Self-Tuning eine Veränderung der Konfiguration erwünscht. Daher müssen neben den Kosten für die

[11] Vgl. [WeHa94], [WeMö02]

Verwaltung der Indexe auch die Kosten für die Erstellung der neuen Indexe berücksichtigt werden[12].

Da das QUIET System nur eine Erweiterung von DB2 und nicht integriert ist, erzeugt es einen gewissen Overhead. Eine weiterführende Integration des Self-Tunings von Indexen in ein DBMS wurde in [LüSa07] vorgestellt. Hierfür entwickelten Lühring et al. das Konzept der Soft-Indexe und implementierten dies in PostgreSQL. Der Begriff Soft-Indexe wurde für die selbstverwalteten Indexe gewählt, um sie gegen die Hard-Indexe, welche vom DBA erstellt werden, abzugrenzen. Die Integration ermöglichte eine Nutzung des Kostenmodells von PostgreSQL und verringerte den Overhead im Vergleich zu QUIET.

Des Weiteren können Indexe leer angelegt werden, um sie später während eines Table Scans durch eine Anfrage „on the fly“ erzeugen zu können. Dadurch verringert sich die Systemlast durch Einsparung eines Table Scans zur Indexerzeugung. Die Indexe des Systems verfügen somit über drei unterschiedliche Zustände: Realisiert, virtuell und leer angelegt (deferred).

Da die Anpassung der Indexe nicht kontinuierlich erfolgt, wird für jede Anfrage an die Datenbank folgender Ablauf durchgeführt:

1. Für eine Anfrage *Q* werden die nützlichen Indexe bestimmt.
2. Ein Kostenplan wird für *Q* bestimmt, der die realisierten Indexe nutzt.
3. Ein weiterer Kostenplan unter Berücksichtigung der virtuellen Indexe wird für *Q* erstellt.
4. Die Differenz zwischen 2 und 3 bildet den Profit, bei Nutzung der virtuellen Indexe. Dieser Wert wird zur Aktualisierung der globalen Indexkonfiguration genutzt, auf dessen Grundlage später entschieden wird, welche Indexe realisiert werden.

3.5 Partielle Indexe

Das Konzept der Partiellen Indexe stellte M. Stonebraker in [St89] vor. Bei dieser Form von Indexen werden nicht die gesamten Daten eines Attributes indiziert sondern nur ein Teil. Dies geschieht in einer zusätzlichen Bedingung, die beim Anlegen des Indexes spezifiziert werden muss. In der SQL Syntax wird die CREATE INDEX Anweisung durch eine zusätzliche WHERE-Klausel erweitert:

[12] Für weitere Details zum Kostenmodell von QUIET siehe [SaSc04]

```
CREATE INDEX <Index-name>
ON <Tabellen-name> (<Attribut-name>)
WHERE <Bedingung des Indexes>
```

Bei Anfragen entscheidet der Optimierer des DBMS ob der partielle Index zur Beantwortung der Anfrage genutzt werden kann. Hierzu muss die gesamte Anfrage im Bereich des Indexes liegen. Ist dies nicht der Fall, bleibt der Index ungenutzt.

Für einen partiellen Index auf eine Datenbank mit Preisen sieht dies folgendermaßen aus:

```
CREATE INDEX PreisIndex ON Preistabelle (Preis)
WHERE Preis > 10 AND Preis < 100
```

Der partielle Index wird für alle Datensätze erzeugt, deren Attribut „Preis“ größer als 10 und kleiner als 100 ist. Eine Anfrage die diesen Index nutzen kann, muss demnach in diesem Intervall liegen. Eine zulässige Anfrage wäre z.B.:

```
SELECT * FROM Preistabelle
WHERE Preis >= 50 AND Preis <= 60
```

Anfragen wie

```
SELECT * FROM Preistabelle
WHERE Preis = 120
```

oder

```
SELECT * FROM Preistabelle
WHERE Preis > 80
```

werden dagegen nicht durch den Index unterstützt. Die erste Anfrage liegt außerhalb des indizierten Bereiches. Die zweite Anfrage liegt mit einem Preisanfrage von über 80 teilweise im Intervall. Aufgrund der fehlenden oberen Schranke wird allerdings der Index nicht benutzt.

Partielle Indexe sind für verschiedenste Anwendungen sinnvoll einsetzbar. Es lassen sich mit ihrer Hilfe benutzerspezifische Indexe erstellen. Durch die Indexierung eines Teiles der Daten wird Speicherplatz gespart sowie die Anfragegeschwindigkeit für diesen Bereich erhöht. Des Weiteren sind partielle Indexe, die genau auf einzelne Anfragen, die z.B. täglich wiederholt werden, zugeschnitten sind, in der Lage das System zu entlasten. Dies ist möglich, indem die

Indexe bei geringer Systemlast erzeugt werden. Wenn die Nutzer nun in Zeiten hoher Systemlast ihre Anfragen an die Datenbank stellen, ist das DBMS in der Lage mit Hilfe der bereits erzeugten Indexe diese schnell zu beantworten.

Besonders hilfreich sind partielle Indexe bei der Erstellung von vollständigen Indexen, wenn diese nicht auf einmal erstellt werden können oder sollen. Hier kann der partielle Index schrittweise vergrößert werden, bis er die gesamte Datenmenge erfasst und somit zum vollständigen Index wird. Sinnvoll ist es ebenfalls Nutzeranfragen zu verwenden, bei denen entweder ein partieller Index genutzt, ein neuer angelegt oder ein bestehender Index erweitert wird. Der Nachteil von partiellen Indexen liegt dagegen in der fehlenden Unterstützung von Datenbankoperationen, die die gesamte Tabelle betreffen, wie z.B. Joins.

Im Rahmen des Self-Tunings wird die Auswahl verschiedener Indexkonfigurationen betrachtet und die beste Variante bezüglich der Beschleunigung der Anfragen umgesetzt. Hierbei können Indexe nur komplett realisiert werden. Mit Hilfe der partiellen Indexe ist es möglich einen Teilbereich zu indizieren. Dadurch lässt sich die Indexkonfiguration feinteiliger steuern und besser an die jeweilige Anfragesituation anpassen. Der Nachteil bei diesem Vorgehen liegt in der erhöhten Komplexität des ISP.

3.6 Last - balancierte Indexe

In den Kapiteln 3.1 bis 3.4 wurden die Grundlagen des Tunings und Self-Tunings vorgestellt, wobei genauer auf das Indextuning eingegangen worden ist. Hierbei wurden die Indexe, die am nützlichsten sind, d.h. die die Anfragen an die Datenbank am besten unterstützten, entweder vom Administrator ausgewählt, oder durch eine automatische Auswahl bestimmt und angelegt. Im folgenden Unterkapitel wird ein weitergehender Ansatz beschrieben, der das Problem des Indextunings von der Ebene der Indexselektion auf die Ebene der Indexe selbst verlagert.

Bisher waren der Aufbau und die Größe eines Indexes von seiner Art und von den zugrundeliegenden Daten abhängig. In [SaSc05] beschreiben K. Sattler, E. Schallehn und I. Geist einen Forschungsansatz, bei dem Indexe sich selbst verwalten und mit der Anzahl der Zugriffe wachsen bzw. schrumpfen. Größe und Aufbau werden demnach nicht länger von den Daten bestimmt sondern von der Nutzungshäufigkeit.

Für die Änderung dieses Konzeptes sprechen mehrere Gründe. Zum ersten sind die bisherigen Indexe darauf optimiert einen bestimmten Datensatz in möglichst kurzer Zeit zu finden. Dabei spielt es keine Rolle, ob dieser Datensatz häufig, selten oder nie abgefragt wird. In allen Fällen verursacht der Index die gleichen Kosten. Zum zweiten können die Standard Indexstrukturen nur ganz oder gar nicht angelegt werden. Es existieren zwar Konzepte hierfür, wie die oben beschriebenen partiellen Indexe, aber im Rahmen des Self-Tunings sollten Indexe je nach Bedarf wachsen oder schrumpfen können. Zum dritten ist beim Anlegen eines Indexes bekannt, welche Systemressourcen er benötigt. Dies ändert sich z.B. durch Einfügeoperationen, die ein Anwachsen des Indexes verursachen. Es ist nicht möglich dieses Wachstum zu beschränken und die Systemressourcen zur Laufzeit zu verteilen.

Um das Konzept an einem Beispiel zu demonstrieren, wurde ein Last-balancierter Binärbaum implementiert. Dieser hat eine steuerbare Größe, die Anzahl der Knoten. Weiterhin besitzt er als Blattknoten Container, die bei häufigen Zugriffen weniger Daten enthalten, um die Seitenzugriffe zu minimieren. Die Suche innerhalb des Baumes ändert sich nicht, innerhalb eines Containers benötigt sie einen sequentiellen Suchlauf. Abb. 3-2 stellt den Baum dar.

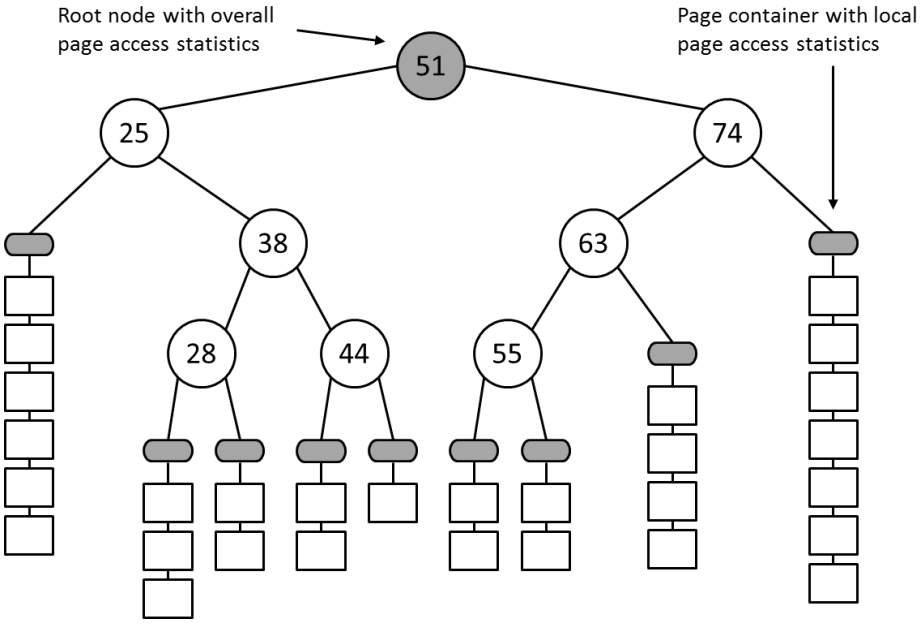

Abb. 3-2: Last-balancierter Binärbaum [SaSc05]

Die Reorganisation des Baumes erfolgt nach zwei einfachen Regeln. Während der Wurzelknoten die Gesamtzahl der Zugriffe und die aktuelle Größe des Baumes enthält, speichern die Container die Zugriffe auf sich selbst. Zunächst wird ein *balance* Wert ermittelt:

$$balance = \frac{all\ page\ accesses}{number\ of\ page\ containers}$$

Bei Knoten, die auf zwei Container verweisen, muss die Summe der Zugriffe auf beide Container größer sein als der *balance* Wert. Andernfalls wird der Knoten entfernt und die beiden Container zu einem verbunden. Andersherum wird ein Container auf zwei aufgeteilt, wenn der Zugriffszähler doppelt so hoch ist, wie der *balance* Wert. Außerdem muss ein freier Knoten verfügbar sein.

Eine Umsetzung des Konzeptes für einen Last-balancierter B-Baum erfolgte in [Mu08]. Es bleiben allerdings weiterhin Fragestellungen offen. So ist bisher ungeklärt, wie ein Self-Tuningkonzept auf dieser Basis für alle Indexe eines Systems aussehen könnte. Weiterhin ist eine Integration in kommerzielle DBMS bisher nicht vorgenommen wurden, weshalb auch keine Untersuchungen unter realen Bedingungen erfolgten. Ebenfalls fehlen Umsetzungen des Konzeptes auf multidimensionale Indexstrukturen. Einen Beitrag hierzu stellt die Übertragung des Konzeptes auf das Grid-File im nächsten Kapitel dar.

4 Problemstellung und Lösungskonzept

In den vorangegangenen Kapiteln wurden die Grundlagen und das Self-Tuning von Indexstrukturen beschrieben. Einen Schwerpunkt stellten hierbei der Aufbau und die Funktionsweise des Grid-Files dar. Weiterhin wurden die Konzepte der partiellen sowie der Last-balancierten Indexe vorgestellt. Von diesem Wissen ausgehend wird im Folgenden das Problemfeld dieser Diplomarbeit näher beschrieben und ein Lösungsansatz entwickelt. Eine experimentelle Umsetzung wird in Kapitel 5 beschrieben und die Evaluation des Konzeptes in Kapitel 6 präsentiert.

4.1 Betrachtung des Problemfeldes

Ausgehend von der Prämisse das Speicher und die Rechenkapazitäten, die zum Erstellen und Verwalten eines Indexes notwendig sind, unbegrenzt und kostenlos zur Verfügung stehen, ist es immer günstiger einen Index zu erstellen und zu benutzen. In dieser Situation sind auch Tuningmaßnahmen nicht notwendig, da diese eine optimale Nutzung begrenzter Ressourcen wie Anfragezeit oder Speicherplatz ermöglichen sollen. Dass die Annahme in der Realität nicht zutrifft, ist offenkundig. Dieses Beispiel zeigt allerdings, dass bei Indexen generell eine Abwägung zwischen dem verfügbaren Speicherplatz, der Rechenkapazität und der Beschleunigung der Datenbankoperationen erfolgt. Die Verbesserung eines Wertes ist nur auf Kosten der anderen möglich.

Auch wenn die oben genannte Prämisse nicht zutrifft, so nehmen der Speicherplatz sowie die Leistung von Computern bei gleichbleibenden Preisen stetig zu. Dem gegenüber steht die immer größer werdende Anzahl an zu verwaltenden Daten. M. Hilbert und P. Lopez haben in [HiLo11] ermittelt, wie hoch die aktuelle Datenmenge der Menschheit ist und die Entwicklung seit 1986 aufgezeigt. Dabei geben sie ein jährliches Wachstum der global gespeicherten Datenmenge mit 23% an. Auch wenn der Großteil dieses Zuwachses nicht in Datenbanken gespeichert wird, so steigert sich hier die Datenmenge ebenfalls. Um eine effiziente Verwaltung zu gewährleisten, ist das DBMS auf leistungsfähige Indexe angewiesen.

Zugriffsstrukturen wie z.B. Hashfunktionen können nur eine begrenzte Anzahl an Daten verwalten, da es sonst zu häufigeren Kollisionen kommt, die eine effiziente Suche verhindern. B-Bäume dagegen wachsen mit größeren Datenmengen mit, indem sie sich immer weiter

verzweigen. Neben dem benötigten Speicherplatz wächst auch die Höhe des Baumes. Diese nimmt logarithmisch mit der Anzahl der Daten zu. Bei einer Anfrage sind somit mehr Knoten zu durchsuchen. Der Suchaufwand vergrößert sich. Der Aufwand, der bei einer Restrukturierung des Indexes notwendig ist, nimmt ebenfalls zu.

Das Grid-File verhält sich ähnlich. Hierbei nimmt mit steigender Datenanzahl die Zahl der Buckets und somit die Anzahl der Zellen im Grid-Directory zu. Durch das ‚two-disk-access' Prinzip ist gesichert, dass ausschließlich zwei Sekundärspeicherzugriffe erfolgen und die Anfragegeschwindigkeit für alle indizierten Daten relativ gleich ist. Der Nutzen einer hohen Anfragegeschwindigkeit wird mit dem Anstieg des Speicherplatzverbrauches erkauft.

Es ist dabei unerheblich, ob ein Datensatz häufig, selten oder nie angefragt wird. Für Datenbereiche, die nie angefragt werden, ist die Anfragegeschwindigkeit belanglos. Ebenfalls ist es denkbar, dass eine Diskriminierung bezüglich der Anfragehäufigkeit stattfindet. So würden häufig angefragte Daten eine höhere Anfragegeschwindigkeit erhalten. Auf diese Weise steigt die durchschnittliche Anfragezeit nur geringfügig. Das Wachstum des Grid-Files lässt sich aber beschränken.

Im Rahmen der weiteren Ausführungen wird untersucht, wie eine Beschränkung der Größe des Grid-Files dynamisch auf Basis der Zugriffsverteilung umgesetzt werden kann. Hierzu wird im Folgenden genauer beschrieben, was unter „Größe des Grid-Files" zu verstehen ist.

4.2 Größenbeschränkung des Grid-Files

Um das Wachstum begrenzen zu können, muss zuerst definiert werden, was unter der Größe des Grid-Files zu verstehen ist. Das Grid-File besteht aus zwei Komponenten, die in ihrem Speicherverbrauch variieren: Das Grid-Directory und die Buckets. Beide hängen direkt voneinander ab. Vorausgesetzt sei eine konstante Anzahl an Datensätzen im Grid-File. Bei einer größeren Anzahl an Buckets (die Bucketgröße c wird verkleinert) steigt der Speicherplatzbedarf des Grid-Directorys, da mehr Grid-Zellen zur Verfügung gestellt werden müssen. Andersherum sinkt bei Verkleinerung des Grid-Directorys die Anzahl der Buckets, wodurch ihre Größe steigen muss um alle Daten zu verwalten.

Als Größe des Grid-Files wird die Anzahl der Grid-Zellen definiert[13]. Diese lässt sich aus der Anzahl der in den Skalen gespeicherten Intervalle berechnen. Bei Skalen mit der Intervallanzahl von 3, 3 und 4 ergibt sich eine Größe des Grid-Directorys von 3 * 3 * 4 = 36. Die Anzahl der Grid-Zellen lässt sich wählen, indem ein vorgegebener Speicherplatzverbrauch beachtet wird. Vorstellbar ist somit neben der Speicherung der Skalen im Hauptspeicher auch ein vorhalten des gesamten Grid-Directorys, ohne Gefahr Teile davon in den Sekundärspeicher auslagern zu müssen.

Mit einer festgelegten Größe des Grid-Directorys ergibt sich eine Maximalanzahl an zu verwaltenden Buckets. Da diese eine festgelegte Größe besitzen, ist es bei einer wachsenden Anzahl an Datensätzen unmöglich alle Daten zu verwalten. Als Problemlösung existieren zwei Möglichkeiten: Die Begrenzung der Datenmenge und die Aufhebung der Bucketgröße.

Zur Begrenzung der indizierten Datenmenge bietet sich das Konzept der partiellen Indexe an, das in Kapitel 3.5 vorgestellt wurde. Mit Hilfe des last-balancierten Ansatzes aus Kapitel 3.6 kann die Beschränkung der Bucketgröße aufgehoben werden. Um eine Verwechslung zwischen den verschiedenen Grid-File Varianten auszuschließen, wird das Grid-File nach Nievergelt et al. in dieser Arbeit als Standard Grid-File bezeichnet. Aus der Kombination beider Konzepte mit dem Grid-File ergeben sich folgende Abwandelungen.

- ein partielles Grid-File:

  Das Standard Grid-File wird auf einen Teil des Datenraumes erzeugt, so dass die Größenbeschränkung erfüllt ist. Um ein Überlaufen beim Einfügen neuer Daten zu verhindern und den größten Nutzen aus dem Index zu ziehen, ist eine Verwaltung aufgrund der Häufigkeit der Datennutzung im Rahmen des Self-Tunings notwendig.

- ein last-balanciertes Grid-File:

  Das Standard Grid-File wird modifiziert, so dass weiterhin alle Daten über den Index erreichbar sind, aber eine unterschiedliche Beschleunigung des Zugriffes auf diese erfolgt. Durch die Last-Balancierung ist es notwendig, dass sich das Grid-File intern selbst umstrukturieren und verwalten kann.

[13] Als Größe des Grid-Files wird somit nur die Größe des Grid-Directorys berücksichtig

Bevor diese zwei Varianten eingehender betrachtet werden, erfolgt ein Exkurs zur Bestimmung des Trennwertes. Bislang wurde der Mittelwert des zu trennenden Intervalls als neuer Trennwert festgelegt. Da der Mittelwert des Intervalls unabhängig von den indizierten Daten ist, kann er bei ungünstig verteilten Daten zu einer Vergrößerung des Grid-Directorys führen. Dies würde dem Ziel der Größenbeschränkung zu wieder laufen. Daher wird im Folgenden Abschnitt die Trennwertbestimmung aufgrund des Medians der Daten diskutiert.

4.3 Trennwertermittlung mit dem Median

Beim Standard Grid-File wird ein Trennwert durch den Mittelwert der Grenzen des zu teilenden Intervalls bestimmt. Wenn die Daten des zu trennenden Buckets auf einer Seite des Trennwertes liegen, kommt es zu keiner Aufteilung. In diesem Fall wird erneut ein Trennwert in dem Intervall gebildet, in welchem die Datensätze liegen. Solange ein überlaufenes Bucket nicht geteilt wurde, werden weitere Trennwerte erzeugt. Dieses Verfahren hat den Nachteil, dass eine Reihe von nicht benötigten Grid-Zellen entsteht. Um dies zu vermeiden, wäre es sinnvoll ein Verfahren zu benutzen, bei dem die Trennwerte gezielter gesetzt werden. Hier bietet sich der Median der Daten an.

Der Median $\tilde{x}$ einer Datenmenge *n* wird berechnet, indem vorweg die Daten aufsteigend sortiert werden$(x_1, x_2, \dots, x_n)$. Er ergibt sich aus dem mittleren Element bzw. Elementen wie folgt:

$$\tilde{x} = \begin{cases} x_{\frac{n+1}{2}} & n\ ungerade \\ \frac{1}{2} * \left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) & n\ gerade \end{cases}$$

Der Median des Intervalls ist gleich dem Mittelwert des Intervalls, da das Intervall nur aus den beiden Begrenzungswerten besteht und somit für den Fall n=2 dieselbe Formel für Median und Mittelwert ergibt. Daher wird im Folgenden der Mittelwert des Intervalls mit dem Median der Daten verglichen, wobei ausschließlich von Mittelwert und Median gesprochen wird.

Die Bestimmung des Mittelwertes ist in der Umsetzung des Grid-Files einfach, da die benötigten Intervallgrenzen in den Skalen im Hauptspeicher gehalten werden. Der Median benötigt dagegen alle Daten des zu trennenden Buckets, so dass ein Overhead entsteht. Da in der anschließenden Teilung des Buckets in beiden Verfahren die Daten sortiert werden

müssen, lässt sich durch eine intelligente Implementierung dieser verringern. Hierbei werden die sortierten Werte der Trennwertermittlung für die Bucket-Trennung genutzt, sodass sich der Aufwand auf das Einordnen des Trennwertes reduziert.

Bei der Bestimmung des Trennwertes mit dem Mittelwert werden die Daten zufällig getrennt. Dagegen erzeugt der Median eine gleichmäßige Aufteilung der Daten auf die beiden neu entstandenen Buckets. Bei einer geraden Anzahl an Datensätzen im Ursprungsbucket entstehen Buckets mit dem gleichen Auslastungsgrad. Ansonsten ist die Anzahl in einem Bucket um eins gegenüber dem anderen erhöht.

Ein Nachteil des Mittelwertes ist die Notwendigkeit eindeutiger Intervallgrenzen. Sie müssen bei Anlage des Indexes Apriori bekannt sein, um den Datenraum zu definieren. Bei Erstellung eines Indexes wird z.B. ein Attribut vom Typ Integer mit den Grenzen des Wertebereiches (−2.147.483.648; 2.147.483.647) angelegt. Werden nun Daten eingefügt, die nur zwischen 1.000.000 und 2.000.000 liegen, ist der Rest des Grid-Files leer. Dieser leere Raum würde durch den Mittelwert geteilt werden, wodurch ein übergroßes Grid-Directory entsteht. Eine Ermittlung des Datenraumes aus den bestehenden Daten berücksichtigt nicht zukünftige zulässige Datensätze, die außerhalb des Indexes liegen. Dies würde eine Neuerstellung des Grid-Files notwendig machen, da bei einer Erweiterung die vorhandenen Trennwerte nicht länger die Mittelwerte der Intervalle abbilden.

Ein Ähnliches Problem tritt bei Daten auf, die größere Lücken beinhalten. Bei diesen entstehen ebenfalls nicht benötigte Grid-Zellen. Abb. 4-1 stellt diesen Sachverhalt graphisch dar. Während durch den Median vier Grid-Zellen entstehen, erzeugt der Mittelwert 12. Allgemein lässt sich zusammenfassen, dass der Mittelwert bei ungleich im Datenraum verteilten Daten ein zu großes Grid-Directory erzeugt.

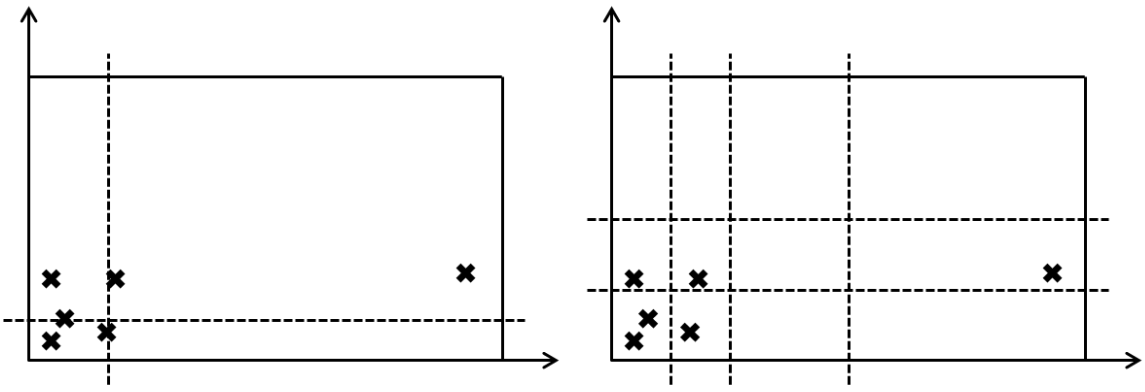

Abb. 4-1: Median der Daten, Mittelwert des Intervalls

Bei gleichverteilten Daten mit bekanntem Wertebereich sollten beide Methoden zu ähnlichen Ergebnissen führen. Dieses wird sowohl für die Anzahl der Grid-Zellen, als auch für die Bucketauslastung in Kapitel 6.1 experimentell überprüft.

4.4 Partielles Grid-File

Das partielle Grid-File ist aufbauend auf den Grundlagen, die in Kapitel 3.5 zu den partiellen Indexen erörtert wurden, ein Standard Grid-File, das einen Ausschnitt des Datenraumes indiziert. Die Schwierigkeit hierbei bildet die Ermittlung des Ausschnittes, der mit einer festgelegten Speichergröße den meisten Nutzen bringt. Als Kriterien für die Auswahl bietet sich die Zugriffshäufigkeit an. Da es sich beim Grid-File um ein mehrdimensionales Verfahren handelt, muss diese Entscheidung ebenfalls mehrdimensional getroffen werden.

Bei der Erstellung eines partiellen Indexes im eindimensionalen Fall steht eine große Menge an möglichen Indexen zur Auswahl. Bei mehreren Dimensionen erhöht sich diese Anzahl entsprechend. Die Komplexität des ISP potenziert sich mit der Anzahl der Dimensionen. Abb. 4-2 stellt ein partielles Grid-File in einem zweidimensionalen Datenraum dar.

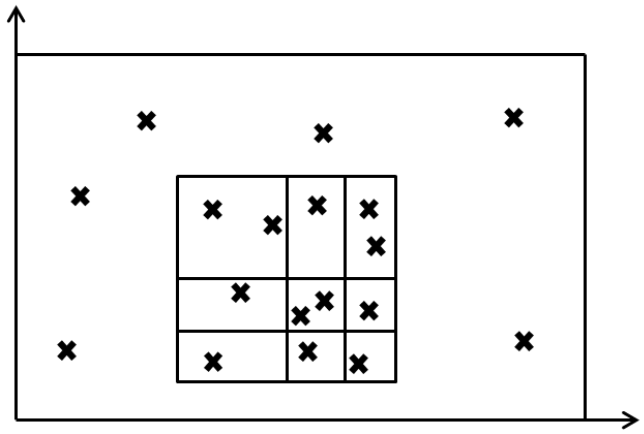

Abb. 4-2: partielles Grid-File im Datenraum

Um ein partielles Grid-File zu ermöglichen, müssten zuerst die Attribute in Klassen eingeteilt und die Anfragen auf diese ermittelt werden. Im Anschluss werden diese Bereiche mit der entsprechenden Datenanzahl kombiniert und so ein Datenraum zur Indizierung ermittelt. Die Datenanzahl ist notwendig, um die Größenbeschränkung beachten zu können.

Zur Bestimmung von partiellen Indexen hat M. Giard bereits in [Gi08] ein Testframework erarbeitet, um verschiedene Algorithmen zur Ermittlung von partiellen Indexen zu testen. Bisher wurden ausschließlich B-Baum Indexe als partielle Indexe erstellt. Dieses Framework

müsste erweitert werden, um eine mehrdimensionale Betrachtung des Problems zu ermöglichen.

Da der Rahmen dieser Arbeit nicht ausreichend ist sowohl das partielle als auch das last-balancierte Grid-File genauer zu untersuchen, wird auf eine eingehendere Diskussion dieses Ansatzes verzichtet.

4.5 Entwurf eines last-balancierten Grid-Files

Im Gegensatz zum partiellen Index ist bei einem last-balancierten Index der gesamte Datenraum zu indizieren. Um eine Größenbeschränkung für den Index zu gewährleisten, müssen Teilbereiche der Daten eine schlechtere Anfragebeschleunigung durch den Index erhalten. Analog zum Verfahren des last-balancierten Binärbaums[14] wird ein Größenmaximum festgelegt und weitere Daten in Datencontainern zusammengefasst. Je mehr Daten in einem Container vorliegen, umso länger dauert das Finden eines einzelnen Wertes. Die Verteilung der Datensätze auf die Container wird durch die Anfragen an die Datenbank gesteuert[15]. Hierbei gilt, dass häufig genutzte Daten schnell erreichbar, d.h. in kleinen Containern vorliegen sollen. In diesem Unterkapitel wird dieses Konzept auf das Grid-File angewendet, um es schrittweise zu einem last-balancierten Grid-File umzubauen.

4.5.1 Bucketlisten

Das Grid-File besitzt bereits in der Standardversion Datencontainer, die Buckets. Sie können eine fixe Menge c an Datensätzen aufnehmen und haben somit eine feste Größe. Um diese Beschränkung aufzuheben ist es notwendig, entweder die Buckets ohne Beschränkung wachsen zu lassen oder Buckets in einer Liste aneinander zu hängen. Im Rahmen der anschließenden Betrachtungen wird ausschließlich die Möglichkeit verketteter Listen von Buckets verwendet. Hierbei ist zu beachten, dass eine Begrenzung der Anzahl der Buckets nicht existiert.

Beide Lösungsmöglichkeiten haben gemeinsam, dass für den Zugriff auf alle Daten des Buckets bzw. der Bucketliste mehrere Speicherseiten aufgerufen werden können. Dies

[14] Vgl. Kap. 3.4 lastbasierte Indexstrukturen und [SaSc05]
[15] Kapitel 4.5.3 erläutert die formale Umsetzung des Konzeptes

verletzt das ‚two-access Prinzip' des Grid-Files. Für die weiteren Überlegungen ist es daher notwendig dieses Prinzip abzuschwächen und umzuformulieren:

> **Two-access-Ziel**: Für häufig angefragte Datenbereiche des Grid-Directorys sollen die Grid-Zellen, wenn möglich, auf ein einzelnes Bucket verweisen. Je geringer die Anzahl der Nutzeranfragen ist, desto länger darf die Bucketliste sein. Die selteneren Zugriffe machen eine längere Zugriffszeit zumutbar.

Durch die Änderung des Prinzips besteht die Möglichkeit, die Größe des Grid-Directorys auf Kosten der Anfragegeschwindigkeit zu begrenzen. Zur Veranschaulichung ist in Abb. 4-3 ein zweidimensionales Grid-File mit einer Bucketgröße von drei dargestellt. Die Größe des Grid-Directorys wurde auf vier Zellen beschränkt. Da in der unteren linken Ecke vier Datensätze eingefügt wurden, verweist die Grid-Zelle 2-1 auf eine Bucketliste, bestehend aus den Buckets C und D. Es ist zu beachten, dass das Bucket C vollständig gefüllt ist und Bucket D die restlichen Datensätze enthält. Um einen Datensatz in Grid-Zelle 2-1 zu suchen, sind nicht mehr zwei sondern drei Sekundärspeicherzugriffe nötig: Einer zur Einsicht des Grid-Directorys und zwei Zugriffe auf die Buckets.

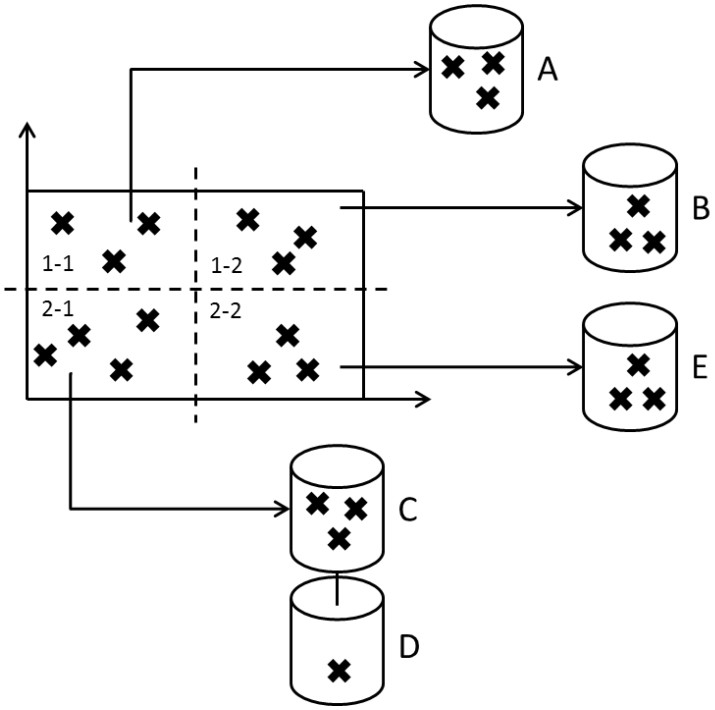

Abb. 4-3: Grid-File mit verketteten Buckets

Im Standard Grid-File können mehrere Grid-Zellen als Region auf ein Bucket zeigen. Es ist denkbar, dass dies auch für eine Bucketliste gilt. Eine Aufteilung der Liste ist ohne zusätzliche Kosten bezüglich der Größe des Grid-Directorys möglich. Daher wird definiert,

dass nur von einer einzelnen Grid-Zelle auf eine Bucketliste verwiesen werden darf. Sollte eine Grid-Region auf eine Bucketliste verweisen, so ist die Bucketliste entsprechend der Region zu teilen.

Durch diese Regel, welche keinerlei Ausnahmen zulässt, können bei ungünstiger Datenverteilung in einem Grid-File leere Buckets entstehen. Sollten in der Umgebung ausschließlich Bucketlisten existieren, ist eine Verschmelzung mit einer anderen Zelle ebenfalls nicht möglich. Eine Verringerung der durchschnittlichen Bucketauslastung ist das Ergebnis. Andererseits wird diese durch die erhöhte Auslastung in den Bucketlisten mehr als kompensiert, da alle bis auf das letzte Bucket zu 100% gefüllt sind.

Ein last-balanciertes Grid-File wächst nach diesen Regeln zunächst wie im Standardfall bis das Grid-Directory die festgelegte maximale Größe erreicht hat. Anschließend werden weitere Datensätze nur noch eingefügt. Ist im betreffenden Bucket kein Platz mehr vorhanden, wird ein neues Bucket an dieses angehängt. Bei gleichverteilten Daten und gleichverteilten Trennwerten entstehen auf diese Weise gleichlange Bucketlisten. Einfüge- und Löschoperationen weiterer Daten erfolgen nach dem obig beschriebenen Schema. Ebenfalls wurde die Merge-Strategie des Standard-Grid-Files nicht geändert. Diese wird verwendet, solange die betreffenden Zellen auf einzelne Buckets verweisen.

4.5.2 Trennwertverlegung

Die Betrachtung der Nutzeranfragen blieb bisher außen vor. Nach dem 'two-access-Ziel' ist eine Verkleinerung der Bucketlisten in häufig angefragten Datenbereichen wünschenswert. Um dies zu ermöglichen, muss die entsprechende Grid-Zelle durch einen neuen Trennwert geteilt werden. Da die Größenbeschränkung des Grid-Directorys keine neuen Trennwerte zulässt, ist ein bereits bestehender Trennwert zu löschen. Hieraus ergibt sich die Notwendigkeit einer neuen Funktion: Der Trennwertverlegung.

Diese lässt sich, wie beschrieben, in zwei Einzelschritte teilen:

- Löschung eines Trennwertes
- Einfügen eines neuen Trennwertes

Während das Einfügen eines neuen Trennwertes bereits im Standard Grid-File implementiert wurde[16], ist die Löschung von Trennwerten bisher nicht berücksichtigt worden.

Um einen Trennwert löschen zu können, müssen die Grid-Zellen, die von diesem Trennwert geteilt werden auf dasselbe Bucket zeigen, d.h. zu einer Grid-Region gehören (in Abb. 4-4 mit Pfeilen gekennzeichnet). Vereinfacht dargestellt existieren eine linke und eine rechte Grid-Zelle, die zu einer Region zusammengefasst werden müssen. Hierbei bleibt die Bucketauslastung in den zugehörigen Buckets bzw. Bucketlisten unberücksichtigt[17]. Die bisher genutzten Verfahren zur Zusammenlegung von Grid-Zellen, das buddy system und das neighbor system, wurden konstruiert, um einzelne Grid-Zellen bei zu geringer Belegung zu vereinen. Das buddy system hat den Nachteil, dass es nur die vorher getrennten Grid-Zellen wieder zusammenfügen kann. Da aber ein wahlfreier Löschvorgang nötig ist, bei dem der Trennwert und somit die zu verbindenden Zellen ausgesucht werden, ist das buddy system für diese Aufgabe ungeeignet.

Das neighbor system ermöglicht eine Verschmelzung von frei gewählten Nachbar Grid-Zellen. Es hat aber die Einschränkung, dass die neu gebildete Grid-Region konvex ist. Diese Bedingung kann bei einer Vereinigung der Grid-Zellen eines Trennwertes nicht gewährleistet werden, wie in Abb. 4-4 verdeutlicht. Wenn der Trennwert zwischen den Zellen 3 und 4 entfernt werden soll, müssen die grau schraffierten Grid-Regionen zusammengelegt werden. Es entsteht eine große nicht konvexe Grid-Region. Daher ist das neighbor system kein geeignetes Mittel, zeigt aber eine Lösung auf.

Um alle Paare von Grid-Zellen zusammenfassen zu können, wird das neighbor system ohne die Einschränkung, dass konvexe Regionen entstehen müssen, durchgeführt. Um nach der Trennwertlöschung wieder konvexe Regionen zu erhalten, ist eine Teilung der entstandenen Region durchzuführen. Spätestens eine Region bestehend aus zwei Grid-Zellen ist konvex. Als Alternative stellt sich die Zulassung von konkaven Regionen dar, so dass die entstandene Region wie erstellt weiter existieren kann. Zu beachten ist, dass die Bedingung aus Kapitel 4.4.1, das nur eine einzelne Zelle auf eine Bucketliste verweisen darf, weiterhin gilt. Sollte die Region auf eine Bucketliste verweisen, muss diese getrennt werden.

[16] Vgl. Kap 2.2.3 und [NiHi84]
[17] Die Daten werden in einer Bucketliste aneinander gehängt, so dass beliebige Grid-Regionen zusammengefasst werden können. Die Bucketliste wird im späteren Verlauf der Trennwertverlegung wieder geteilt.

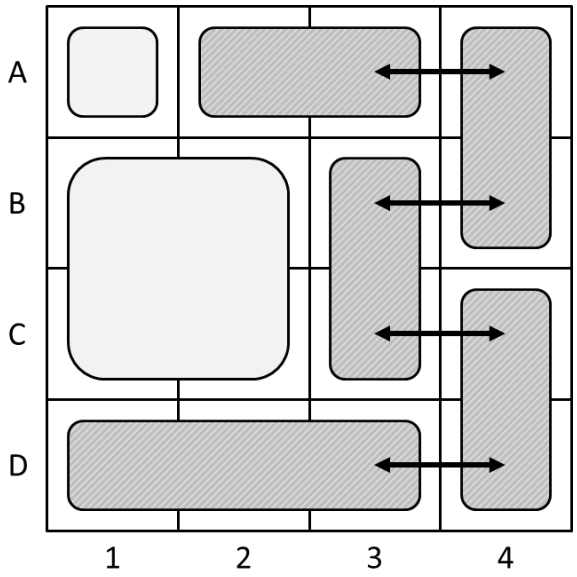

Abb. 4-4: Trennwertlöschung

Der Unterschied zwischen konkav und konvex zeigt sich nicht bei der Anfrage von Daten, sondern bei der Verwaltung der Regionen. Bei der Anfrage eines Datums wird auf eine Grid-Zelle zugegriffen, wodurch konkave und konvexe Regionen keinen Unterschied darstellen. Bei Bereichsanfragen wird zuerst eine Liste aller relevanten Grid-Zellen, bzw. Buckets erstellt, so dass hier ebenfalls kaum Unterschiede auftreten. Während für Grid-Files mit konvexen Regionen allerdings Weiterentwicklungen existieren[18], sind diese bei konkaven Regionen nicht anwendbar. Bei einer Implementation, wie sie in Kap. 5 durchgeführt wird, sind beide Varianten möglich. Der Rechenaufwand bei konvexen Regionen liegt in der Unterteilung der bei Trennwertverlegung fusionierten Regionen. Bei Zulassung konkaver Regionen muss bei jeder Trennung sichergestellt werden, dass die entstehenden Regionen zusammenhängend sind. Am Beispiel in Abb. 4-4, lässt sich dies zeigen. Eine Aufteilung der Bucketliste des grau hinterlegten Bereiches ist für den Trennwert zwischen Spalte 2 und 3 nicht möglich, da sonst die Grid-Zellen A-2, D-1 und D-2 eine Grid-Region darstellen würden, diese aber nicht verbunden ist. Dies wäre ein Wiederspruch gegen das Prinzip, das ähnliche Werte ebenfalls im Speicher eng beieinander liegen.

Nach dem Zusammenfassen der Buckets kann der Trennwert entfernt werden. In der entsprechenden Skala wird der Wert, sowie im Grid-Array eine der zwei zuvor zusammengefassten Ebenen gelöscht. Die Trennung von vorhandenen Regionen wird erst nach diesem Schritt durchgeführt, da hierdurch die Komplexität verringert wird.

[18] Vgl. Two-Level Grid-File [Hi85]

Das Einfügen eines neuen Trennwertes kann jetzt durchgeführt werden. Wie schon oben beschrieben, ist das Verfahren das gleiche wie im Standard Fall[19]. Einzig die Festlegung des Trennwertes unterscheidet sich für den Fall, das der Median der Daten verwendet wird. Da es hierbei keinen überlaufenden Bucket gibt, der zur Bestimmung des Medians herangezogen werden kann, muss ein anderes Maß gefunden werden. Der Sinn für das Einfügen eines neuen Trennwertes stellt die Tatsache dar, dass die vorhandenen Bucketlisten verkleinert werden sollen. Es bietet sich an, die größte Bucketliste in der entsprechenden Spalte des Grid-Directorys als Entscheidungskriterium heranzuziehen, um eine möglichst große Reduktion zu erreichen. Eine Berücksichtigung aller Daten der Spalte ist schon aus Gründen der Geschwindigkeit nicht empfehlenswert. Des Weiteren fehlt in diesem Fall ein Maß für die Verringerung der Bucketlistengröße. Für die Umsetzung und Evaluierung in Kapitel 5 und 6 wird auf diese Möglichkeit vorerst verzichtet und der Mittelwert des Intervalls als Trennwert genutzt.

4.5.3 Self-Tuning Konzept

Nachdem die Möglichkeit besteht einen Trennwert gezielt zu verlegen, ist es notwendig auf den Self-Tuning Aspekt zurückzukehren. Das Konzept eines last-balancierten Indexes beinhaltet, dass sich der Index selbstständig an Anfragemuster anpasst. Hierzu müssen die Anfragen gezählt werden. Um dies zu realisieren, werden diese in Kategorien eingeteilt, die den Abschnitten auf den Skalen entsprechen. So wird für jedes Attribut eine separate Anfragestatistik geführt. Dies ermöglicht eine unabhängige Anpassung der Attribute des Grid-Files. Der Anfragezähler wird in den Skalen implementiert, da bei einer Anfrage diese abgefragt werden müssen. Im Gegensatz zur Speicherung der Anfragen in den einzelnen Grid-Zellen wird so wesentlich weniger Speicherplatz benötigt.

Es wird definiert, dass eine Anfrage den Anfragezähler aller Skalenintervalle um eins erhöht, die zum Ergebnis der Anfrage zählen. Eine Punktanfrage erhöht somit den Anfragezähler in einem Intervall pro Skala. Hingegen kann eine Bereichsanfrage alle Anfragezähler einer Skala erhöhen, wenn für diese keine Einschränkung stattgefunden hat.

Allein auf Grundlage der Anfragen kann ein Self-Tuning im Grid-File nicht erfolgen. Im Gegensatz zum last-balancierten Binärbaum dienen beim Grid-File die Buckets als kleinste Speichereinheiten und nicht die einzelnen Datensätze. Dies hat Auswirkungen auf die

[19] Vgl. Kap. 2.2.3

Anpassungsstrategie. Wenn eine Grid-Zelle auf ein einzelnes Bucket verweist, ist es nicht sinnvoll diese durch einen neuen Trennwert zu teilen. Die beiden neuen Grid-Zellen nach der Trennung wären eine Grid-Region und würden auf das selbe Bucket verweisen.

Des Weiteren ist nicht nur eine Grid-Zelle von einem neuen Trennwert betroffen, sondern entsprechend der Größe der anderen Skalen des Grid-Directorys mehrere. Dies führt dazu, dass ein Maß gefunden werden muss, wie sinnvoll ein Trennwert in einem Skalenintervall ist. Hierfür wird der Verbesserungswert eingeführt. Er wird wie der Anfragezähler innerhalb der Skala pro Intervall verwaltet. Die Berechnung erfolgt, indem alle Grid-Zellen der entsprechenden Spalte durchlaufen werden und die Anzahl der Buckets minus eins zum Verbesserungswert hinzuaddiert wird. Für ein zweidimensionales Grid-File ergeben sich folgende Formeln:

n = Anzahl der Zeilen
m = Anzahl der Spalten
V = Verbesserungswert

$$V_n = \sum_{i=1}^{m} (Anzahl\ der\ Buckets\ der\ Gridzelle\ (n,i) - 1)$$

$$V_m = \sum_{i=1}^{n} (Anzahl\ der\ Buckets\ der\ Gridzelle\ (i,m) - 1)$$

Abb. 4-5 verdeutlicht dies anhand eines Beispiels. Die Zahlen innerhalb des Arrays stellen die Anzahl der Buckets der Region dar. Die Zahlen an den Rändern sind die errechneten Verbesserungswerte. Da Grid-Regionen mit mehreren Grid-Zellen auf ein einzelnes Bucket verweisen, haben sie keinen Effekt auf den Verbesserungswert. Weiterhin kann daher ausgeschlossen werden, dass Buckets mehrfach zum Verbesserungswert innerhalb einer Skala beitragen.

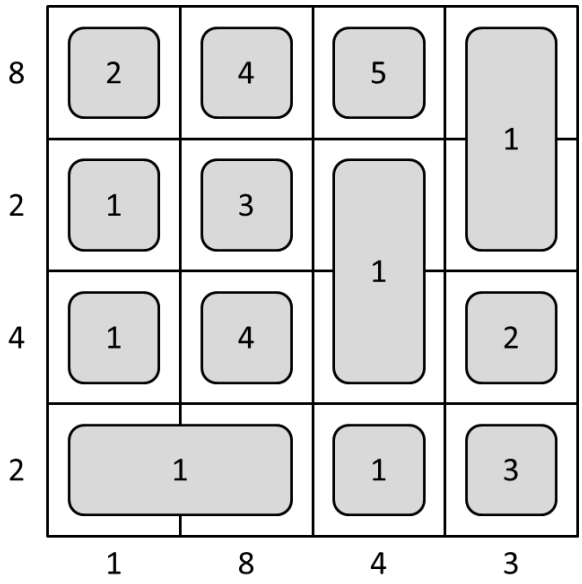

Abb. 4-5: Bestimmung des Verbesserungswertes

Mit dem Anfragezähler und dem Verbesserungswert wurden Parameter erstellt, aufgrund derer eine Entscheidung zur Änderung eines Trennwertes getroffen werden kann. Drei Werte sind hierfür zu bestimmen:

- die Skala, auf der ein Trennwert verlegt wird
- das Intervall der Skala, in dem ein neuer Trennwert erstellt wird
- der zu löschende Trennwert

Die Bestimmung der Skala und des zu trennenden Intervalls erfolgt in einem Schritt. Hierzu wird für jedes Intervall ein Wert *E* errechnet, indem der Anfragezähler durch die Summe aller Anfragezähler der zugehörigen Skala geteilt und mit dem Verbesserungswert multipliziert wird. *E* ist somit ein Maß für die Wichtigkeit eines neuen Trennwertes.

$$E_i = \frac{Anfragezähler_i}{\sum_{k=1}^{n} Anfragezähler_k} * Verbesserungswert_i$$

Das Intervall, dessen Wert *E* am größten ist, wird der Trennwertverlegung als Einfügeintervall übergeben. Somit ist auch eine Skala ermittelt. Um Auszuschließen, dass für optimal verteilte Trennwerte eine Trennwertverlegung erfolgt, muss der Wert *E* größer als der Abbruchwert *A* sein. *A* ist ein frei wählbarer Parameter, so dass dieser individuell einstellbar ist.

Nachdem ein Einfügeintervall und somit auch eine Skala ermittelt worden sind, muss ein Trennwert der Skala bestimmt werden, der gelöscht wird. Hierzu werden die Anfragezähler

der beiden angrenzenden Intervalle addiert. Der Trennwert, der die wenigsten Anfragen auf sich vereint, wird gelöscht.

$$L_i = Anfragezähler_i + Anfragezähler_{i+1}$$

Nachdem die Bestimmung des Einfügeintervalles und des zu löschenden Trennwertes beendet ist, erfolgt die zuvor beschriebene Trennwertverlegung. Nach Abschluss dieser werden alle Anfragezähler auf null gesetzt, da für das getrennte Intervall nicht ermittelbar ist, wie viele Anfragen auf die nun entstandenen Teilintervalle entfallen sind.

Um die Bestimmung des Einfüge- und des Löschwertes sowie die Trennwertverlegung zu veranlassen, wird für das weitere Vorgehen ein globaler Anfragezähler definiert, der nach einer zuvor eingestellten Anzahl an Anfragen diese auslöst. Die Alternative, die Trennwertverlegung zu bestimmten Zeiten zu starten, ist mit dem bisherigen Konzept nicht vereinbar, da nur ein Trennwert pro Zyklus verlegt wird. Für einen produktiven Einsatz wäre dies wünschenswert, macht aber eine Anpassung des Self-Tunings und der Trennwertverlegung notwendig[20].

4.5.4 Theoretisches Verhalten des last-balancierten Grid-Files

Nachdem mit den Kapiteln 4.5.1 bis 4.5.3 die Entwicklung des Standard Grid-Files zu einem last-balancierten Grid-File vollzogen ist, wird sich dieser Abschnitt mit den daraus resultierenden Veränderungen im Verhalten beschäftigen. Ob dieses Verhalten auch zutrifft, wird in Kapitel 6 anhand einer prototypischen Implementierung untersucht werden.

Das Einfügen eines neuen Datensatzes funktioniert wie im normalen Grid-File, bis die Größenbeschränkung erreicht ist. Nachdem diese erreicht ist, wird bei einem Bucketüberlauf ein neuer Bucket angehängt und die Verbesserungswerte erhöht. Ein neuer Trennwert wird auf diese Weise nicht benötigt und die Einfügeoperationen beschleunigen sich.

Das Löschen eines Datensatzes erfolgt auf ähnliche Weise. Bei maximaler Größe des Grid-Directorys wird dieser ohne Anpassung der Trennwerte erfolgen. Die Verbesserungswerte werden bei der Löschung eines Buckets angepasst. Die Löschung größerer Datenmengen ist bisher nicht berücksichtigt. In dieser Situation müsste eine Überprüfung stattfinden, ob ein

[20] Vgl. Kapitel 4.5.5

Trennwert nach der Löschung von Daten überflüssig ist. Hier ist weiterer Forschungsbedarf vorhanden, der allerdings in dieser Arbeit nicht geklärt werden kann.

Die Abfrage eines Datensatzes erfolgt nach dem Schema, das für das Standard Grid-File in Kapitel 2.2.3 erläutert wurde. Durch die Bucketlisten sind mehrere Buckets zu durchsuchen. Der Suchaufwand wird demnach zumindest für einen Teil der Anfragen ansteigen. Im Laufe der Anpassung des Grid-Files an die Anfrageverteilung sollte dieser Suchaufwand sinken. Dies gilt ebenso für Bereichsanfragen.

Die Anpassung des Grid-Files an die Anfrageverteilung wird durch den globalen Anfragezähler nach x Anfragen ausgelöst. Dies führt zu einer regelmäßigen Durchführung der Trennwertverlegung, bis der Abbruchwert erreicht ist. Durch die Verlegung eines einzelnen Trennwertes pro Zyklus, ändert sich die Verteilung nur langsam. Wie viele Zyklen für eine Anpassung notwendig sind, wird im Rahmen der Evaluierung zu bestimmen sein.

Zusammenfassend lässt sich sagen, dass die Geschwindigkeit bei Abfragen verringert wird, da mehr Buckets durchsucht werden müssen. Dies ist der Preis, der für die Beschränkung des Größenwachstums des Grid-Directorys gezahlt werden muss. Auch die regelmäßigen Trennwertverlegungen bei geänderten Anfrageverhalten werden zu Geschwindigkeitseinbußen führen.

4.5.5 Mögliche Weiterentwicklungen

Um das last-balancierte Grid-File flexibler anpassbar an die Anfragesituation zu gestalten, besteht die Möglichkeit die Beschränkung der Trennwertverlegung auf eine Skala aufzuheben. Zur Realisation müsste zuerst eine Skala gewählt werden, bei der ein Trennwert zu löschen ist. Das Grid-Array muss bis zum Rand umkopiert und die überzählige Spalte gelöscht werden. In einer anderen Skala wird eine Spalte hinzugefügt und das Grid-Array bis zur Einfügeposition umkopiert. Anschließend kann der neue Trennwert eingefügt werden.

Im Gegensatz zur Verlegung eines Wertes innerhalb der Skala, muss bei jeder dieser Trennwertverlegungen auf die Größenbeschränkung geachtet werden. Angenommen die Skalen A und B haben 4 und 8 Intervalle, so dass sich eine Größe des Grid-Arrays von 32 ergibt. Dies entspricht der Größenbeschränkung. Wenn nun in Skala B ein Wert gelöscht und in Skala A eingefügt wird, ergibt sich eine Größe von 5 * 7 = 35, was eine Verletzung der

Größenrestriktion darstellt. Somit müssten zwei Trennwerte in der Skala B gelöscht werden, um einen Wert in Skala A einfügen zu können (5 * 6 = 30).

Die Umsetzung einer getrennten Lösch- und Einfügeoperation für die Trennwerte ermöglicht gleichzeitig die Löschung bzw. das Einfügen eines Trennwertes unabhängig voneinander. Dies erscheint allerdings nur sinnvoll, wenn die Größe des Grid-Files unter der Größenbeschränkung liegt. Im gegenteiligen Fall führt ein neuer Trennwert immer zu einer Verbesserung.

Weiterhin wäre es möglich, die Beschränkung der Trennwertverlegung auf eine Skala aufzuheben, so dass alle Skalen während eines Zyklus geändert werden können. Dies könnte allerdings zu Nebeneffekten führen, da jede Skalenänderung auch Auswirkungen auf die anderen Skalen hat.

Einen weiteren Ansatzpunkt zur Entwicklung bilden die Regeln des Self-Tunings aus Kapitel 4.5.3. Die Berechnungsregel des Verbesserungswertes könnte geändert werden, so das längere Bucketlisten einen höheren Wert besitzen. Dies würde eine Bevorzugung dieser bewirken, so dass sie bei einer relativ gleichen Anzahl an Anfragen eher getrennt werden.

Der Anfragenzähler könnte ebenfalls geändert werden. So besteht die Möglichkeit diesen bei der Trennwertverlegung nicht auf null zu setzen sondern ihn beispielsweise zu halbieren. So altert der Anfragezähler, so dass bei zeitlich begrenzten Ausreißern keine komplette Umstrukturierung des Grid-Files stattfindet.

5 prototypische Implementierung und Testumgebung

Das folgende Kapitel beschreibt die Implementierung des in Kapitel 4 entwickelten last-balancierten Grid-Files. Des Weiteren wurde eine Testumgebung geschaffen, um mit zufallsgenerierten Werten das Verhalten zu untersuchen. Da die Umstrukturierung des last-balancierten Grid-Files nach der Streuung der Anfragen erfolgt, werden verschiedene Wahrscheinlichkeitsverteilungen im zweiten Teil dieses Kapitels erläutert.

5.1 Einschränkungen des Prototypen

Die Implementation des last-balancierten Grid-Files erfolgte nach den in Kapitel 4.5.1 bis 4.5.3 gemachten Ausführungen. Da es sich um einen Prototypen handelt, der zur Evaluation des vorgestellten Konzeptes dient, ist es nicht nötig eine allgemeine Implementierung vorzunehmen. Im Vorfeld wurden daher einige Festlegungen getroffen, die im Folgenden vorgestellt werden.

5.1.1 Spezifikation der Daten

Den Ausgangspunkt der Evaluierung bildet eine statische Datenbank. Folglich werden während der Laufzeit keine Einfüge- und Löschoperationen durchgeführt. Daher wird auf eine Mischstrategie für unterbelegte Buckets verzichtet. Die Anzahl der Datensätze in der Datenbank ist beliebig. Es wird eine Struktur für die Datensätze definiert, die aus drei Werten unterschiedlichen Datentyps besteht:

- long Zeitstempel
- int Zahl
- string Text

Der Zeitstempel ist die Sekundenanzahl für die Zeitspanne seit dem 01.01.1970. Er dient in der Datenbank als Primärschlüssel und ist damit eindeutig. Um dies zu gewährleisten wird bei der Erzeugung einer Datenbank einem neuen Zeitstempel der Wert des vorherigen Zeitstempels plus 60 zugewiesen, beginnend mit null. Die Datensätze sind somit innerhalb der Datenbank nach den Zeitstempeln sortiert. Als Wertebereich dieses Merkmals ergibt sich:

$$W_{Zeitstempel} = \{0, 60, 120, \dots, (Anzahl\ der\ Daten - 1) * 60\}$$

Die Zahl ist ein numerischer Wert zwischen 0 und 99.999. Durch diesen geringen Wertebereich kommt es innerhalb des Merkmales zu Wiederholungen, da während der Evaluation bis zu 10 Millionen Datensätze erzeugt werden. Hierdurch wird untersucht, ob eine Abweichung des Verhaltens gegenüber den anderen Skalen besteht.

$$W_{Zahl} = \{0, 1, 2, \dots, 99999\}$$

Das Merkmal Text ist ein alphanumerischer Wert, bestehend aus fünf Zeichen. Als Alphabet[21] stehen die Ziffern von 0 bis 9 sowie die Buchstaben von a bis z zur Verfügung. Eine Unterscheidung in Groß und Kleinschreibung erfolgt hierbei nicht. Durch Kombination entstehen insgesamt 36^5 mögliche Zeichenketten. Der Wertebereich ist somit definiert durch:

$$W_{Text} = \{00000, 00001, 00002, \dots, zzzzz\}$$

Innerhalb der Datenbank sollen alle Merkmale gleichverteilt vorliegen. Beim Zeitstempel ist dies bereits per Definition festgelegt. Die Attribute Zahl und Text werden mit Hilfe eines Zufallszahlengenerators erzeugt. Durch die Indizierung alle Merkmale erfolgt die Festlegung auf ein dreidimensionales Grid-File. Des Weiteren ermöglicht die Definition der Wertebereiche der Attribute eine eindeutige Festlegung der Grenzen des Datenraumes.

5.1.2 Parametrisierung des Grid-Files

Der Abbruchwert *A*, der beim Self-Tuning des Grid-Files überschritten werden muss, wird auf 0,1 festgelegt. Dies verhindert einen neuen Trennwert in einem Intervall, das einen Anfragezähler oder einen Verbesserungswert von Null besitzt. Ob dies ausreicht, um zyklische Trennwertverlegungen zu verhindern, müssen die Experimente in Kapitel 6 zeigen.

Des Weiteren wurde der globale Anfragezähler, nach dem eine Trennwertverlegung geprüft wird, auf 1000 fixiert. Bei einer Bucketgröße von 100, einer Bucketbelegung von 70% und einer Million gleichverteilter Daten ergibt sich rechnerisch im Standard Grid-File eine Skalengröße von 24,26. Bei Zehn Millionen Datensätze ergeben sich 52,28. Die Anzahl der Anfragen scheint somit ausreichend, um eine Anfrageverteilung durch die Anfragezähler in den Skalen abbilden zu können.

[21] Ein Alphabet ist eine endliche Menge eindeutig identifizierbarer Zeichen. Durch Kombination der Zeichen untereinander entstehen Zeichenketten. [RePo06]

Da bereits bei der Umsetzung des Standard Grid-Files durch Nievergelt et al. keine Messung der Anfragezeit erfolgte, wird bei der späteren Evaluierung in Kapitel 6 ebenfalls darauf verzichtet. Dies ermöglicht es die Implementation ohne Beachtung zeitkritischer Operationen durchzuführen.

5.2 Aufbau des Prototypen und der Testumgebung

Die Implementation sowohl des Prototypen als auch der Testumgebung erfolgte mit C/C++ unter openSuse 11.2. Zur Erzeugung der Datenbanken wurde der Standardzufallszahlengenerator des Systems verwendet. Um die verschiedenen Anfragen zu erstellen, kam das Paket Gnu Scientific Library zur Verwendung[22].

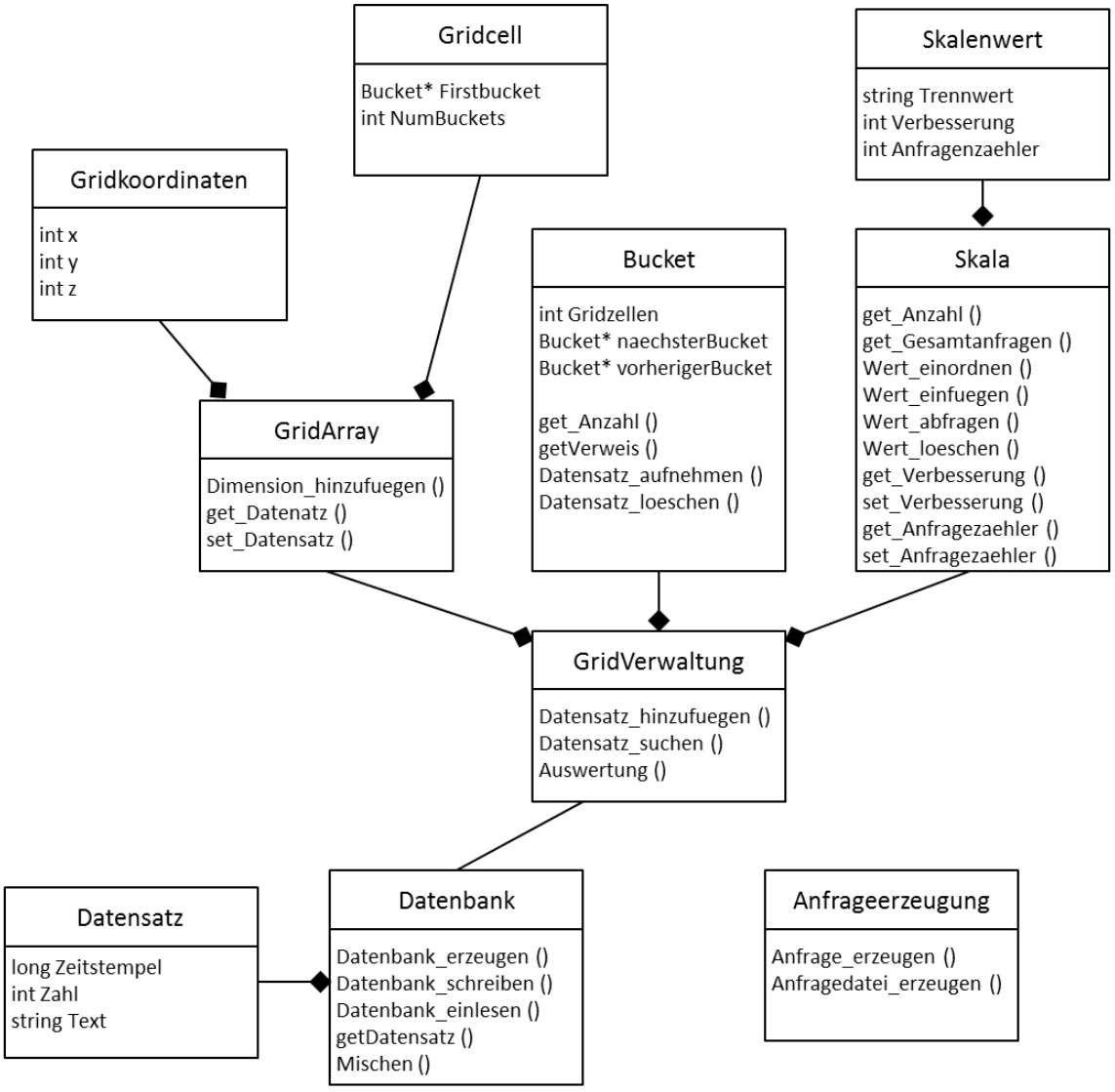

Abb. 5-1: Klassendiagramm

[22] Die vollständige Implementation ist auf dem beigefügten Datenträger enthalten

Das obige Klassendiagramm (Abb. 5-1) beschreibt sowohl den Prototypen, als auch das Testsystem. Hierbei stellen die Klassen Datenbank und Datensatz die Speicherstruktur der zu verwaltenden Daten zur Verfügung. Die Anfrageerzeugung dient der Generierung der in Kapitel 5.3 beschriebenen Zufallsverteilungen. Die Klasse GridVerwaltung sowie die mit ihr durch Komposition verbundenen Klassen bilden den Prototypen des last-balancierten Grid-Files.

Die Klassenstruktur des Prototyps ist in drei Ebenen unterteilt. Die erste Ebene, bestehend aus den Klassen Gridkoordinaten, Gridcell und Skalenwert, dient dazu eigene Datentypen zu definieren. Die zweite Ebene besteht aus den Klassen GridArray, Bucket und Skala. Diese stellen die Komponenten zur Speicherung und Verwaltung der einzelnen Elemente des Grid-Files dar[23]. Die dritte Ebene besteht aus der Klasse GridVerwaltung. Hier werden alle Entscheidungen getroffen und das Grid-File als Ganzes nach außen repräsentiert.

Im Folgenden werden einige wichtige Funktionen im Detail erläutert. Der Pseudocode gibt einen Überblick über den Ablauf der einzelnen Funktionen. Zuerst werden die beiden Hauptfunktionen der Grid-Verwaltung vorgestellt. Danach folgt die Betrachtung der Trennwertverlegung, die etwas abgewandelt zur Beschreibung in Kapitel 4.5.2 erfolgte. Die Funktion zur Trennung einer Grid-Region, bildet den Abschluss

5.2.1 Datensatz hinzufügen

Die Funktion dient dazu einen neuen Datensatz in das Grid-File einzufügen. Hierzu wird die entsprechende Grid-Zelle durch Einordnung der Werte in den Skalen bestimmt. Anschließend wird unterschieden, ob die entsprechende Grid-Zelle auf ein Bucket oder eine Bucketliste verweist. Sollte der einzelne Bucket nicht voll sein, oder eine Bucketliste existieren, wird der Datensatz angefügt. Im Falle des einzelnen Buckets muss überprüft werden, ob eine Grid-Region vorhanden ist, die getrennt werden kann, oder ob die maximale Größe des Grid-Directorys erreicht wurde. Sollten beide Bedingungen nicht zutreffen, kann ein neuer Trennwert erzeugt werden. Anschließend wird die Funktion erneut aufgerufen. Der Pseudocode der Funktion stellt sich wie folgt dar:

[23] Vgl. Kapitel 2.2.2

```
Datensatz hinzufügen (Datensatz D) {
    Grid-Zelle ermitteln ()
    If (Anzahl der Buckets == 1) {
        If (Bucket nicht voll) {
            Datensatz in Bucket einfügen ()
        }
        Else If (Grid-Zellen, die auf Bucket zeigen > 1) {
            Grid-Region trennen ()
            Datensatz hinzufügen (Datensatz D)
        }
        Else If (Größe des Grid-Arrays erreicht){
            Neuen Bucket an Liste anhängen ()
            Datensatz in Bucket einfügen ()
        }
        Else {
            Neuen Trennwert einfügen ()
            Datensatz hinzufügen (Datensatz D)
        }
    }
    Else {
        If (letzter Bucket nicht voll) {
            Datensatz in Bucket einfügen ()
        }
        Else {
            Neuen Bucket an Liste anhängen ()
            Datensatz in Bucket einfügen ()
        }
    }
}
```

5.2.2 Datensatz suchen

Die Suche nach einem Datensatz gestaltet sich wie in Kapitel 2.2.3 beschrieben. Unterschiede bestehen nur in dem Zugriff auf den Anfragezähler in der Skala und dem durchsuchen

mehrerer Buckets, wenn die fragliche Grid-Zelle auf eine Bucketliste verweist. Des Weiteren wird, wenn der globale Anfragezähler den definierten Schwellwert erreicht hat, die Trennwertverlegung gestartet. Der Pseudocode gestaltet sich wie folgt:

```
Datensatz suchen (Datensatz D) {
        Grid-Zelle ermitteln ()
        Anfragezähler erhöhen ()
        Buckets durchsuchen ()
        If (Anfragezähler > Schwellwert) {
                Trennwertverlegung ()
        }
}
```

5.2.3 Trennwertverlegung

Die Trennwertverlegung wurde bereits theoretisch in Kapitel 4.3 erläutert. Dabei wurde ein Trennwert gelöscht und danach ein neuer Trennwert eingefügt. Im Rahmen der Umsetzung sind beide Teilschritte zu einer Funktion verknüpft wurden, so dass die Verschiebung von Grid-Zellen innerhalb des Grid-Arrays reduziert wird. Da den Buckets die Anzahl der auf sie zeigenden Grid-Zellen bekannt ist[24], muss dieser beim Löschen eines Trennwertes um eins reduziert werden (aus zwei Grid-Zellen wird eine), sowie beim Einfügen um eins erhöht werden (aus einer Grid-Zelle werden zwei). Die anschließende Trennung aller Grid-Regionen stellt sicher, dass auf eine Bucketliste nur von einer einzelnen Grid-Zelle aus verwiesen wird. Der Pseudocode der Trennwertverlegung lautet entsprechend:

```
Trennwertverlegung () {
        For (alle Grid-Koordinaten links vom Lösch-Trennwert) {
                If (Grid-Zelle rechts vom Lösch-Trennwert zeigt nicht auf dasselbe
                Bucket) {
                        Grid-Zellen zusammenfassen ()
                }
                Zähler des Buckets für Anzahl der Grid-Zellen --
        }
```

[24] Dieser Zähler wird benötigt, um nicht bei jeder Operation die Grid-Region bestimmen zu müssen. Ebenso ist er ein Garant für die Konsistenz des Grid-Files, da der Zähler und die Anzahl der Grid-Zellen bei Bestimmung der Grid-Region übereinstimmen müssen.

```
        Löschung des alten Trennwert in der Skala ()
        Einfügen des neuen Trennwertes in die Skala ()
        Umkopieren der Grid-Zellen im Grid-Array ()
        For (alle Grid-Koordinaten links vom Einfüge-Trennwert) {
                Zähler des Buckets für Anzahl der Grid-Zellen ++
        }
        Trennung aller Grid-Regionen ()
        Neuberechnung der Verbesserungswerte ()
        Anfragezähler auf 0 setzen ()
}
```

5.2.4 Grid-Region trennen

Während der Trennwertverlegung entstehen zahlreiche Grid-Regionen, die im Anschluss getrennt werden müssen, um die Integrität des Grid-Directorys wieder herzustellen. Die Grid-Region als logisches Konstrukt ist dabei eine Ansammlung von benachbarten Grid-Zellen, die auf dasselbe Bucket verweisen. Zu Beginn ist zu unterscheiden, ob Grid-Regionen konvex sein müssen oder nicht. Im ersten Fall muss die Entscheidung, an welcher Stelle die Grid-Region getrennt wird, die Konvexität der entstehenden Regionen sicherstellen. Da bei der Implementation des Prototyps auf konvexe Regionen verzichtet wurde, muss garantiert werden, dass die entstehenden Grid-Regionen zusammenhängend sind[25]. Im Anschluss wird der beste Trennwert ermittelt, indem der Median der Daten der Region ermittelt wird. Hierzu wird der nächstgelegene Trennwert bestimmt und dieser zur Trennung der Region benutzt. Der Pseudocode lautet:

```
Grid-Region trennen () {
        Bestimmung der Grid-Region ()
        Alle zulässigen Trennwerte ermitteln ()
        Besten Trennwert bestimmen ()
        Grid-Region aufteilen ()
}
```

[25] Vgl. Kap. 4.5.2

5.3 Zufallsverteilungen

Das Ziel des last-balancierten Grid-Files ist es, sich an das Anfragemuster der Nutzung anzupassen. Um dies experimentell überprüfen zu können, wurden drei verschiedene Zufallsverteilungen erzeugt, die als Konstant, Säule und Gauss bezeichnet sind. Diese sind auf alle drei Attribute der Daten anwendbar, so dass eine Kombination zu insgesamt 27 mehrdimensionalen Verteilungen führt. Hiervon wird allerdings nur ein kleiner Teil für die Evaluierung genutzt.

Zur Veranschaulichung der Verteilungen wurden jeweils 1.000.000 Werte zwischen 0 und 9.999 generiert. Die Diagramme zeigen die Häufigkeitsverteilung der einzelnen Zahlen.

Konstant

Die Zufallsverteilung Konstant (Abb. 5-2) ist eine gleichverteilte[26] Wahrscheinlichkeitsverteilung. Dies bedeutet, dass alle Werte mit derselben Wahrscheinlichkeit erzeugt werden. Im idealen Fall ergibt sich eine horizontale Linie. Zur Erzeugung werden ein minimaler und ein maximaler Wert benötigt, die nicht unter- bzw. überschritten werden. Zur Umsetzung der Verteilung auf eines der Attribute der Datenbank werden die Grenzen des jeweiligen Wertebereiches als Minimum und Maximum definiert.

$$Min = W_{min} \qquad\qquad Max = W_{max}$$

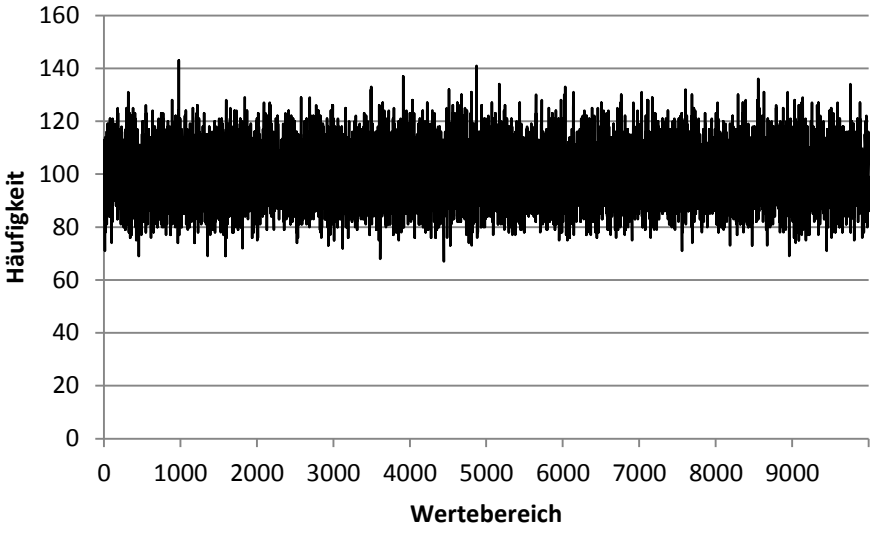

Abb. 5-2: Zufallsverteilung Konstant

[26] Zur Gleichverteilung siehe [Hü96]

Säule

Abb. 5-3 zeigt die Zufallsverteilung Säule. Diese ist ebenso wie die Verteilung Konstant gleichverteilt. Der Unterschied besteht in der Definition des Minimums und des Maximums. Das Minimum ist definiert als 10% des Wertbereiches unterhalb des Mittelwertes. Entsprechend liegt das Maximum bei 10% des Wertebereiches über dem Mittelwert. Es werden dementsprechend 20% des Wertebereiches des Attributes angefragt.

$$Min = 0{,}8 * \frac{(W_{min} + W_{max})}{2}$$

$$Max = 1{,}2 * \frac{(W_{min} + W_{max})}{2}$$

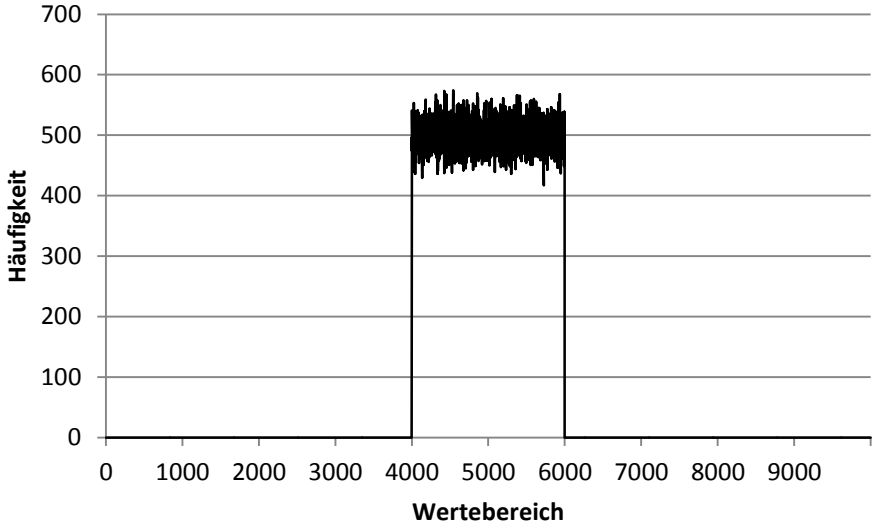

Abb. 5-3: Zufallsverteilung Säule

Gauss

Gauss (Abb. 5-4) ist eine Normalverteilung[27], deren Mittelwert gleich dem Maximum des Wertebereiches entspricht. Bei der Erzeugung der Zufallsverteilung wird hierzu jeder Wert verworfen, der größer als der Mittelwert ist. So werden die Anfragen auf aktuelle Daten bevorzugt. Die Standardabweichung wird mit 10% des Wertebereiches definiert.

$$\mu = W_{max}$$

$$\sigma = 0{,}1 * (W_{max} - W_{min})$$

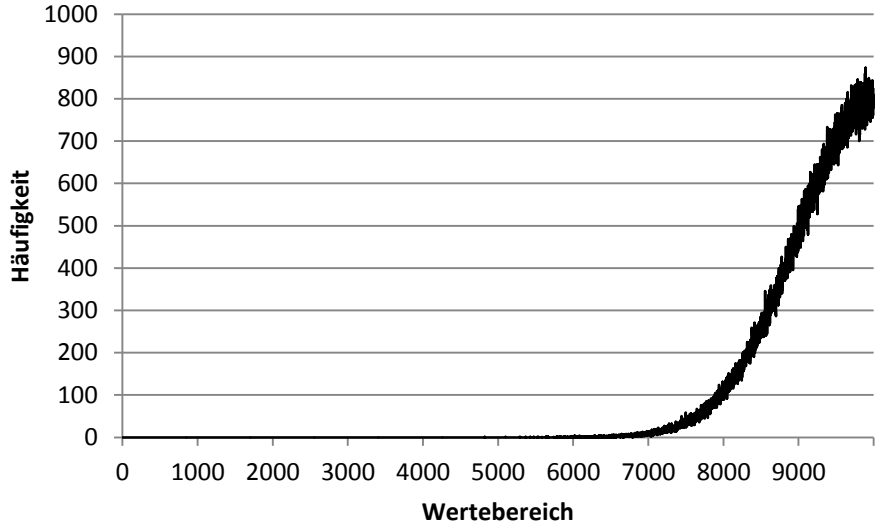

Abb. 5-4: Zufallsverteilung Gauss

[27] Zur Erläuterung der Normalverteilung vgl. [Hü96]

6 Evaluation

In Kapitel 5 wurden der entwickelte Prototyp und die Zufallsverteilungen erläutert. Auf dieser Grundlage werden im Folgenden die theoretischen Ausführungen aus Kapitel 4 geprüft. Zunächst wird in Abschnitt 6.1 der Median mit dem Mittelwert im implementierten System verglichen. Die Evaluation des last-balancierten Grid-Files findet in Kapitel 6.2 statt. Eine Auswertung sowie daraus gezogene Schlussfolgerungen werden direkt im Anschluss an das jeweilige Resultat diskutiert. In allen Experimenten, die durchgeführt wurden, beträgt die Bucketgröße 100. Als Datenmenge für die zugrunde liegende Datenbank wurden 1.000.000 Daten gewählt. Dies ist, mit mindestens 10.000 Buckets, eine ausreichende Größe, um Effekte bei der Veränderung der Trennwerte beobachten zu können. In den Fällen bei denen von diesen Werten abgewichen wird, ist dies explizit kenntlich gemacht.

6.1 Vergleich Median und Mittelwert

Um die Auswirkungen der beiden Trennverfahren auf die Trennwertermittlung des Grid-Files zu untersuchen, wurden 10 Datenbanken erzeugt, die mit A bis K bezeichnet sind. Für alle Datenbanken wurde ein Grid-File ohne Größenbeschränkung generiert. Das reine Einfügen von Daten stellt, wie in Kapitel 2.2.4 beschrieben, einen Test der Teilungsstrategie dar. Um die Verfahren vergleichen zu können, werden die Anzahl der Grid-Zellen und die Bucketauslastung ermittelt. Da die Trennwerte gleichmäßig auf die Skalen verteilt sind, ist die Anzahl der Trennwerte je Skala gleich der dritten Wurzel der Größe des Grid-Directorys. Die Ergebnisse stellen sich wie folgt dar:

Normal	Median		Mittelwert	
	Größe Grid-Directory	Bucketauslastung	Größe Grid-Directory	Bucketauslastung
A	122500	49,50 %	60840	60,22 %
B	120050	49,64 %	60840	60,17 %
C	127500	49,63 %	60840	60,24 %
D	120050	49,57 %	60840	60,38 %
E	127500	49,55 %	60840	60,35 %
F	120050	49,78 %	60840	60,35 %
G	127500	49,43 %	60840	60,28 %
H	125000	49,64 %	60840	60,16 %
I	122500	49,93 %	60840	60,23 %
K	120050	49,37 %	60840	60,26 %
Durchschnitt	123270	49,60 %	60840	60,26 %

Tab. 6-1: Ergebnisse des Standard Grid-Files

Der Median erzeugt im Durchschnitt ein mehr als doppelt so großes Grid-Directory wie der Mittelwert. Ebenso liegt die Bucketauslastung um mehr als 10% unter der des Mittelwertes. Neben den schlechten Ergebnissen des Medians verwundert das Ergebnis des Mittelwertes in Bezug auf die Bucketauslastung. In der Evaluierung von Nievergelt et al. wurde eine Bucketauslastung von 69% gemessen, die auch hier erwartet wurden war. Dies wird im späteren Verlauf näher untersucht.

Bei der Einführung des Medians wurde das Problem des Mittelwertes besprochen, mit unbekannten Intervallgrenzen umzugehen. Zur Untersuchung dieser Vermutung wurde die maximale Intervallgrenze des Attributes Zahl aufgehoben und durch den maximalen Integer Wert des Systems[28] ersetzt. Der Datenraum ist somit in einer Dimension über die Daten hinaus erweitert. Die Ergebnisse des Experiments stellen sich wie folgt dar:

Offenes Intervall	Median		Mittelwert	
	Größe Grid-Directory	Bucketauslastung	Größe Grid-Directory	Bucketauslastung
A	122500	49,50 %	135252	60,73 %
B	120050	49,64 %	146068	57,79 %
C	127500	49,63 %	127500	61,64 %
D	120050	49,57 %	137904	58,53 %
E	127500	49,55 %	143312	59,40 %
F	120050	49,78 %	148877	58,53 %
G	127500	49,43 %	137904	58,54 %
H	125000	49,64 %	151686	57,53 %
I	122500	49,93 %	148877	58,69 %
K	120050	49,37 %	127500	61,36 %
Durchschnitt	123270	49,60 %	140488	59,27 %

Tab. 6-2: Ergebnisse des Standard Grid-Files bei offenem Datenraum

Während der Median wie erwartet die exakt gleichen Ergebnisse liefert, steigert sich die Anzahl der Grid-Zellen durch den Mittelwert erheblich. Durchschnittlich führt der Wegfall der Intervallgrenze in diesem Experiment zu einer Steigerung um 132%. Dieser hohe Wert lässt sich mit der Größe des hinzugewonnenen Datenraumes erklären. Um den Bereich der Daten bis 100.000 mit den Trennwerten überhaupt zu erreichen, sind bereits 15 Trennwerte in der Dimension zu setzen. Anders verhält es sich mit der Bucketauslastung. Diese ist annähernd identisch mit dem vorhergehenden Experiment. Daraus folgt, dass das Fehlen von Intervallgrenzen durch die Größe des Grid-Directorys kompensiert wird.

[28] Beim verwendete Betriebssystem openSuse 11.2 ist dies 2.147.483.647

Trotz der Unempfindlichkeit des Medians gegenüber den Intervallgrenzen, ist sowohl die Bucketauslastung als auch die Größe des Grid-Directorys unerwartet schlecht. Die Vermutung besteht, dass die Sortierung der Daten bezüglich des Zeitstempels hier eine Rolle spielt. So wird der erste Bucket mit 100 Daten gefüllt und in der Mitte geteilt. Die beiden entstandenen Buckets haben eine Auslastung von 50%. Weitere Daten werden nun oberhalb der bisherigen Daten hinzugefügt. Die Auslastung des Buckets unterhalb des Trennwertes wird nicht mehr erhöht. Dieser Wert ist im obigen Experiment zu beobachten.

Um diese These zu überprüfen wurden die Daten in eine zufällige Reihenfolge gebracht. Hierzu dient eine Mischfunktion, die die gesamte Datenbank durchläuft und das jeweils aktuelle Element mit einem zufällig gewählten Element vertauscht. Dieser Vorgang wurde zehn Mal wiederholt, so dass die Datensätze innerhalb der Datenbank zufällig verteilt sind.

Die obigen Experimente sind mit den gemischten Datenbanken wiederholt wurden. Die Änderungen in den Ergebnissen lauten wie folgt:

Random Normal	Median		Mittelwert	
	Größe Grid-Directory	Bucketauslastung	Größe Grid-Directory	Bucketauslastung
A	32768	68,93 %	31744	61,68 %
B	31744	68,62 %	31744	61,81 %
C	35937	69,09 %	31744	61,66 %
D	31744	68,98 %	31744	61,67 %
E	30752	69,00 %	31744	61,63 %
F	33792	68,18 %	31744	61,64 %
G	33792	68,84 %	31744	61,64 %
H	34848	68,50 %	31744	61,72 %
I	32768	68,23 %	31744	61,78 %
K	34848	68,86 %	31744	61,69 %
Durchschnitt	33299,3	68,72 %	31744	61,69 %

Tab. 6-3: Ergebnisse bei zufälliger Einfügereihenfolge

Random Offenes Intervall	Median		Mittelwert	
	Größe Grid-Directory	Bucketauslastung	Größe Grid-Directory	Bucketauslastung
A	32768	68,93 %	57798	77,07 %
B	31744	68,62 %	57798	77,45 %
C	35937	69,09 %	56316	77,24 %
D	31744	68,98 %	57798	77,15 %
E	30752	69,00 %	57798	77,10 %
F	33792	68,18 %	57798	77,36 %
G	33792	68,84 %	57798	77,42 %
H	34848	68,50 %	57798	77,20 %
I	32768	68,23 %	57798	77,02 %
K	34848	68,86 %	53428	77,26 %
Durchschnitt	33299,3	68,72 %	57212,8	77,23 %

Tab. 6-4: Ergebnisse bei zufälliger Einfügereihenfolge und offenem Datenraum

Wie die obigen Ergebnisse zeigen, ist der Median in der Tat abhängig von der Reihenfolge der eingefügten Daten. Die Größe des Grid-Directorys ist auf beinahe ¼ des vorher ermittelten Wertes gesunken, was in diesem Fall einer Reduktion von ca. 17 Trennwerten je Skala entspricht. Ebenfalls verbessert hat sich die Bucketauslastung, die mit rund 69% sehr nah an den von Nievergelt et al. ermittelten Wert liegt. Auffällig sind die Ergebnisse des Mittelwertes. Die Größe des Grid-Directorys schrumpft durch die zufällige Reihenfolge der Daten ebenfalls. Während Median und Mittelwert bei bekannten Intervallgrenzen relativ gleiche Ergebnisse bezüglich des Grid-Directorys erzielen, ist die Bucketauslastung beim Mittelwert erneut mit 62% mangelhaft. Bei der Veränderung der Intervallgrenze erhöht sich die Größe des Grid-Directorys erwartungsgemäß. Überraschend ist die plötzliche Zunahme der Bucketauslastung auf rund 77%.

Zusammenfassend betrachtet, ist die zufällige Verteilung der einzufügenden Daten im Standard Grid-File vorzuziehen, da sich sowohl beim Median als auch beim Mittelwert eine starke Verringerung der Größe des Grid-Directorys beobachten lässt. Eine steigende Bucketauslastung kommt beim Median hinzu. Bei Indizierung eines Primärschlüssels ist die Trennwertermittlung mit Hilfe des Mittelwertes anzuraten, wohingegen bei nicht definierten Intervallgrenzen der Median Vorteile bietet. Die stark voneinander abweichenden Bucketauslastungen des Mittelwertes sind durch die bisherigen Experimente nicht mit den Ergebnissen von Nievergelt et al. in Einklang zu bringen. Daher wird im Folgenden das in Kapitel 2.2.4 beschriebene Experiment zur Evaluierung der Teilungsstrategie des Grid-Files aus [NiHi84] sowohl für den Median als auch den Mittelwert durchgeführt.

Um die obig beobachteten Auswirkungen eines Primärschlüssel-Attributes oder offener Intervalle zu vermeiden, wurde eine neue Datenbank mit 10 Millionen Datensätzen erzeugt. Alle drei Attribute wurden durch den Zufallsgenerator des Systems gleichverteilt generiert. Da das Attribut Text fix in den Prototypen implementiert ist, wurde die Dimension der Attribute Zeitstempel und Zahl entsprechend angepasst. Somit beträgt der Wertebereich aller Attribute:

$$W = \{0,1,2,\dots,60.466.175\}$$

Um die Auswirkungen des Medians und des Mittelwertes sowohl bei geringen, als auch großen Datenmengen beobachten zu können, wurde drei Experimentdurchläufe absolviert:

- 100.000 Datensätze, Messung im Abstand von 100 Einfügeoperationen
- 1.000.000 Datensätze, Messung im Abstand von 1.000 Einfügeoperationen
- 10.000.000 Datensätze, Messung im Abstand von 10.000 Einfügeoperationen

Dabei wurde die momentane Bucketauslastung, sowie die Anzahl der Buckets und der Grid-Zellen ermittelt. Die Abbildungen 6-1 bis 6-3 stellen die Ergebnisse graphisch dar, wobei auf der linken Seite die Trennwertermittlung mit dem Median, auf der rechten Seite mit dem Mittelwert dargestellt ist. Auf den x-Achsen ist die Anzahl der Datensätze, die eingefügt wurden, abgebildet, auf den y-Achsen die Bucketauslastung in Prozent, bzw. die Anzahl der Buckets/Grid-Zellen.

Die Unterschiede im Wachstum des Grid-Files sind in den Grafiken gut zu erkennen. Während die Bucketauslastung beim Mittelwert mit immer länger werdenden Perioden zwischen 60% und 81% schwankt, stabilisiert sich der Median nach anfänglichen Veränderungen bei ca. 69%. Der Median weist neben einer linear wachsenden Anzahl an Buckets nur ein leicht stufenförmiges Wachstum bezüglich der Größe des Grid-Directorys auf. Dagegen ist die Anzahl der Buckets beim Mittelwert einem stufenweisen Wachstum unterworfen, aus dem die hohe Schwankungsbreite in der Bucketauslastung resultiert. Die Größe des Grid-Directorys steigt innerhalb eines kurzen Zeitabstandes auf das 6 – 7 fache des ursprünglichen Wertes, bleibt anschließend allerdings lange Zeit stabil.

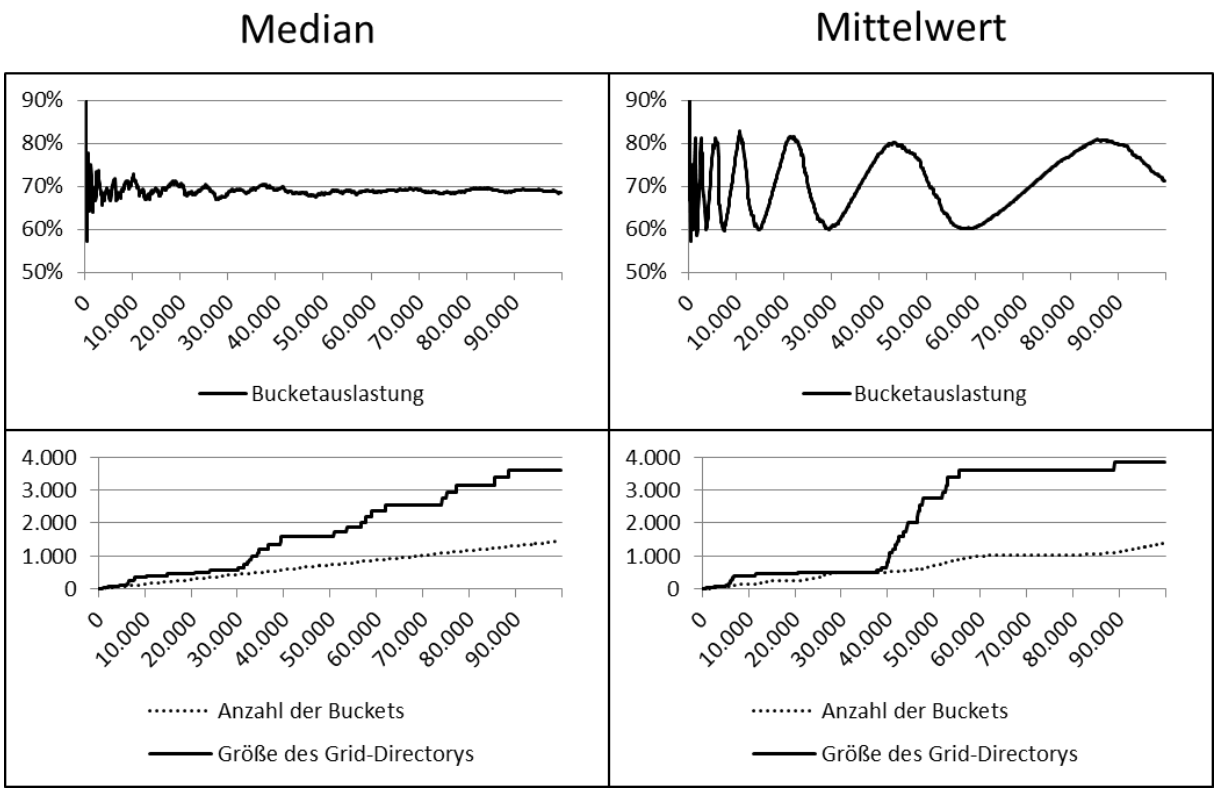

Abb. 6-1: Vergleich Median Mittelwert bis 100.000 Datensätze

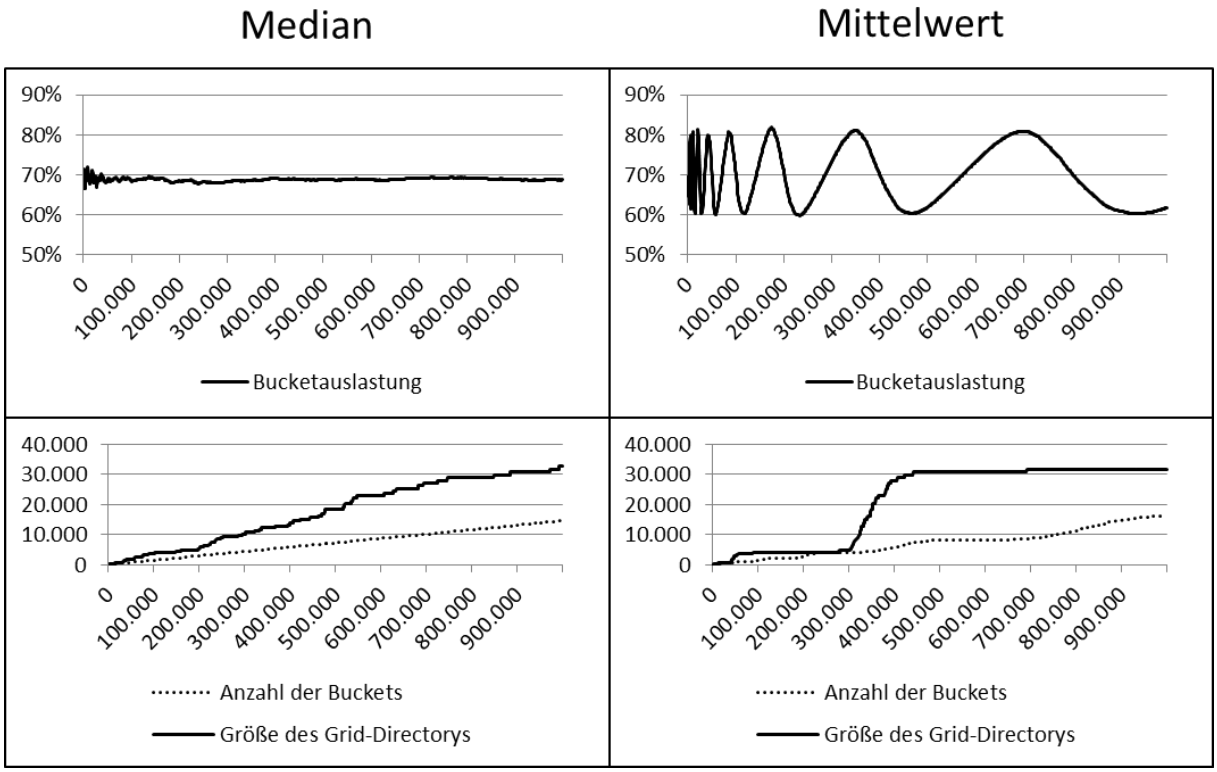

Abb. 6-2: Vergleich Median Mittelwert bis 1.000.000 Datensätze

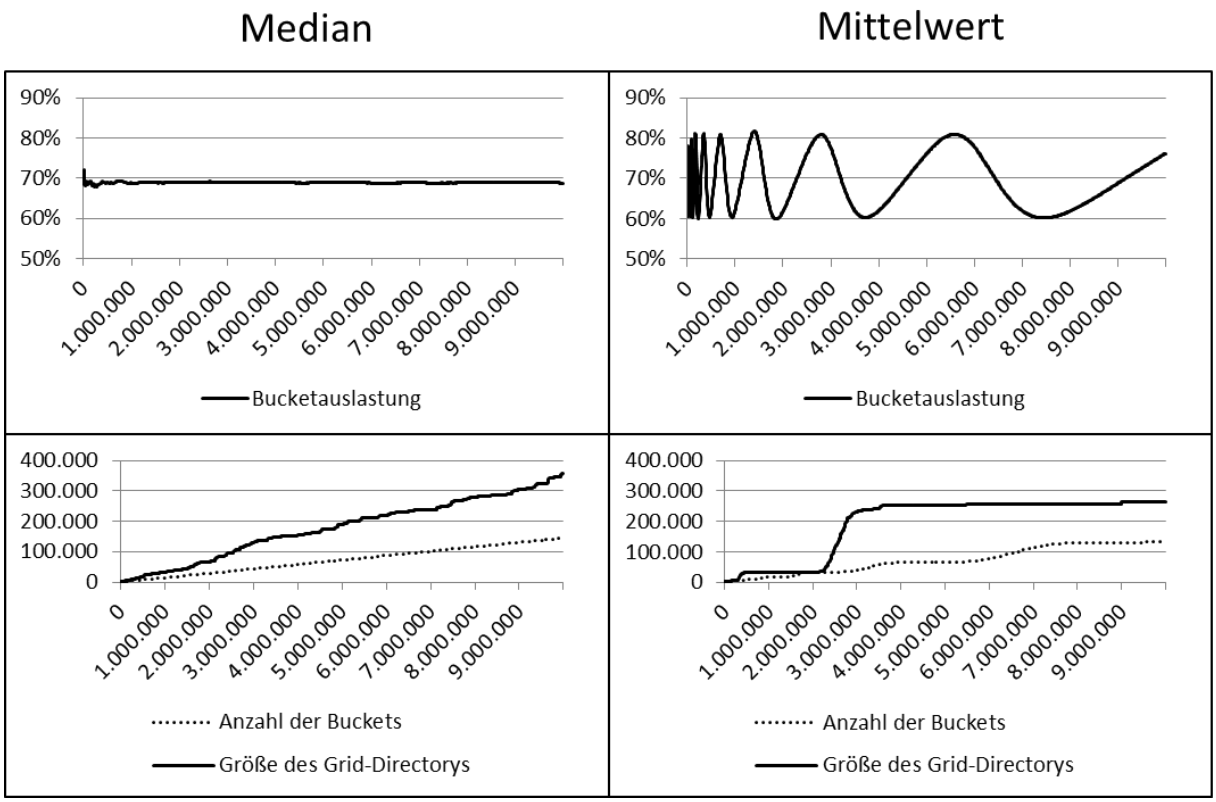

Abb. 6-3: Vergleich Median Mittelwert bis 10.000.000 Datensätze

Die hier beobachteten Ergebnisse stimmen nur zum Teil mit den Ergebnissen aus [NiHi84] überein. Während die durchschnittliche Bucketauslastung bei 69% liegt, ist die regelmäßige Schwankung nicht beschrieben. Ebenso der starke Anstieg der Größe des Grid-Directorys wurde nicht dokumentiert.

Zu vermuten ist, dass der von Nievergelt et al. gemessene Mittelwert in den durchgeführten Experimenten eine sehr gute Approximation des Medians darstellt. Dies könnte aufgrund des gewählten Datenraums geschehen sein. Da in [NiHi84] hierüber keine Angaben zu finden sind, kann dies nur anhand der Abbildungen zur Struktur des Grid-Files geschätzt werden. Es ist zu vermuten, dass der Wertebereich der einzelnen Attribute wesentlich kleiner als die Datenmenge gewählt wurde und es somit zu einer Annäherung des Mittelwertes an die Werte des Medians kam.

6.2 last-balanciertes Grid-File

Im Folgenden wird die Evaluation des last-balancierten Grid-Files durchgeführt. Aufgrund der obigen Ergebnisse wäre es sinnvoll den Median zur Trennwertbestimmung zu verwenden. Da allerdings aus den in Kapitel 4.3 genannten Gründen die Bestimmung eines neuen

Trennwertes bei Größenbeschränkung des Grid-Files auf den Mittelwert beschränkt wurde, wird dieser zur Ermittlung der neuen Trennwerte verwendet.

6.2.1 Vorbereitungen

Zur Evaluation des last-balancierten Grid-Files wird die Datenbank A mit 1.000.000 Datensätzen verwendet. Wie in Tab. 6-1 zu sehen wird daraus ein Standard Grid-File mit der Größe von 60840 erzeugt. Dies ergibt die Größe der Skalen von 40, 39 und 39. Als Größenbeschränkung für das Grid-Directory wurden daher folgende Werte gewählt:

- Skalengröße 10 → Größe des Grid-Directorys: 1000
- Skalengröße 15 → Größe des Grid-Directorys: 3375
- Skalengröße 20 → Größe des Grid-Directorys: 8000

Aufgrund der im weiteren Verlauf zu erläuternden Testergebnisse wurden ebenfalls Skalengrößen, die ein Vielfaches von 2 darstellen, experimentell ausgewertet:

- Skalengröße 8 → Größe des Grid-Directorys: 512
- Skalengröße 16 → Größe des Grid-Directorys: 4096
- Skalengröße 32 → Größe des Grid-Directorys: 32768

Aus den Zufallsverteilungen aus Kapitel 5.3 wurden verschiedene Anfragemuster erstellt. Zunächst sind alle Skalen mit der gleichen Anfrageverteilung untersucht wurden. Anschließend folgte ein gemischtes Anfragemuster, bei dem der Zeitstempel mit der Verteilung Konstant, das Attribut Zahl mit der Verteilung Säule und das Attribut Text mit der Verteilung Gauss angefragt wurden. Für jede dieser Anfragemuster wurde eine Anfragedatei mit einer Millionen Anfragen erstellt, so dass insgesamt 1.000 Trennwertverlegungen durchgeführt werden können. Als letztes ist ein variables Anfragemuster erstellt wurden, das alle 100.000 Anfragen das Anfragenmuster wechselt. Folgende Anfragemuster werden der Reihe nach ausgeführt (jeweils für Zeitstempel, Zahl, Text):

- Konstant, Konstant, Konstant
- Konstant, Säule, Konstant
- Gauss, Säule, Konstant
- Gauss, Gauss, Gauss
- Säule, Konstant, Säule

Zur Bestimmung der Veränderungen des Grid-Files wurden die Skalen und die Bucketbelegung nach jedem Zyklus ermittelt. Ein Zyklus ist hierbei die Spanne zwischen zwei Trennwertverlegungen, also genau 1000 Anfragen lang. Des Weiteren wurden die Trennwertverlegung sowie die Anzahl der aufgerufenen Buckets je Zyklus protokolliert. Der Aufbau der Auswertungsdatei ist in Anhang A aufgeführt.

6.2.2 Evaluierungsergebnisse

Aufgrund der Datenfülle sind nicht alle Ergebnisse darstellbar. Die vollständigen Testprotokolle sowie die Implementierung, die erzeugten Datenbanken und Anfrageverteilungen befinden sich auf dem beigelegten Datenträger. Zur Erläuterung der Evaluation wird im Folgenden ein kleiner Teil graphisch aufgearbeitet präsentiert. Hierbei wird zunächst die Skalengröße von 20 betrachtet und im Anschluss ein Vergleich mit den weiteren Größen gezogen.

Ausgangssituation

Die in Anhang A.2 dargestellte Ausgangssituation entsteht nach einfügen der Werte aus der Datenbank A in das Grid-File. Dies ist für alle weiteren Experimente der Startpunkt und wird mit Zyklus 0 bezeichnet. Abb. 6-4 zeigt die Skala Zeitstempel. Die hier abgetragenen Intervalle stellen die Skaleneinteilung in Bezug zum Verbesserungswert dar. Zu beachten ist die Beschriftung der x-Achse der Diagramme. Während die Skaleneinteilung bei Zeitstempel und Text in Tausendstel angegeben ist, wird beim Attribut Zahl durch den geringen Wertebereich die normale Einteilung genutzt. Weiterhin ist die x-Achse des Attributes Text mit den entsprechenden Zahlenwerten der zugehörigen fünfstelligen Zeichenketten beschriftet.

In Abb. 6-4 ist die Auswirkung der Primärschlüsseleigenschaft auf die Skala gut zu erkennen. Während die Attribute Zahl und Text eine Gleichverteilung der Trennwerte aufweisen, sind beim Zeitstempel zuerst die kleineren Skalenwerte eingefügt wurden. Diese wurden dann durch neue Skalenwerte getrennt. Für die höheren Werte waren somit keine neuen Trennwerte vorhanden, so dass eine lange Bucketliste in einem großen Intervall entsteht.

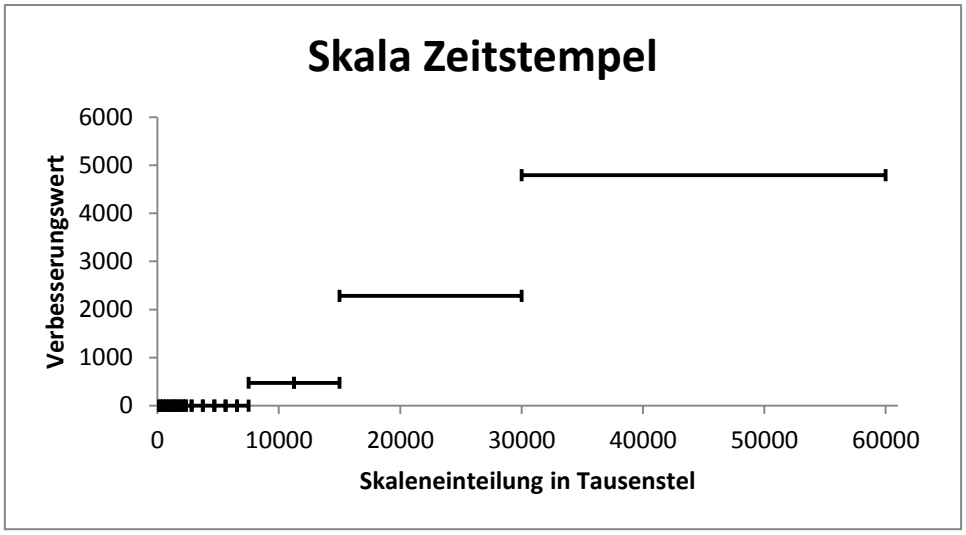

Abb. 6-4: Skala Zeitstempel, 0. Zyklus, 20 Intervalle

Anfrageverteilung Konstant in allen Skalen

Bei dieser Anfrageverteilung wird getestet, ob sich die Trennwerte so anordnen, dass alle Grid-Zellen auf die gleiche Anzahl an Buckets verweisen. Somit wäre die durchschnittliche Zugriffszeit gleichverteilt. Abb. 6-5 zeigt die Skala Zeitstempel nach 25 Zyklen. Grau hinterlegt ist die jeweilige Anfrageverteilung an das Diagramm. Hierbei gilt, dass ein hoher Anfragewert zu einem niedrigen Verbesserungswert führen sollte. Die grau hinterlegte Verteilung ist dabei ausschließlich eine Visualisierung und keine Darstellung der tatsächlichen Anfrageverteilung. Im unteren Bild sollte sich daher mit einer konstanten Anfrageverteilung somit eine Linie bilden, d.h. alle Skalenintervalle sollten den gleichen Verbesserungswert besitzen. Tatsächlich bilden sich zwei Linien heraus, die erste bei einem Verbesserungswert von ca. 500, die andere bei ca. 150. Anhang A.3 und A.4 stellen die Situation aller Skalen in Zyklus 25 und 100 dar. Zu erkennen ist, dass diese Zweiteilung der Trennwertintervalle in allen drei Skalen auftaucht.

Dies lässt sich durch die Verwendung des Mittelwertes erklären. Die gleichmäßige Aufteilung des Datenraumes benötigt beim Mittelwert eine Trennwertanzahl, die eine Potenz von zwei ist. Dadurch entsteht im Beispiel die obere Linie bei einem Verbesserungswert von ca. 500. Jeder zusätzliche Trennwert teilt ein Intervall. Bei der Anfrageverteilung Konstant wird bei einer Trennwertverlegung ein Trennwert zum Löschen ausgewählt, der Intervalle mit geringerem Verbesserungswert trennt[29]. Ein Intervall mit höherem Verbesserungswert wird im Gegenzug getrennt, wodurch neue Intervalle auf der unteren Linie entstehen. Dieser

[29] Dies geschieht zwangsläufig, da die Intervalle kleiner sind und somit weniger Anfragen auf sich vereinen.

Kreislauf setzt sich immer weiter fort. Um dies zu untermauern wurden die obig erwähnten Skalengrößen 8, 16 und 32 eingeführt. Es sei auf Anhang A.11 verwiesen, wo der 25. Zyklus bei einer Skalengröße von 16 darstellt ist. Alle Intervalle liegen auf einer Linie.

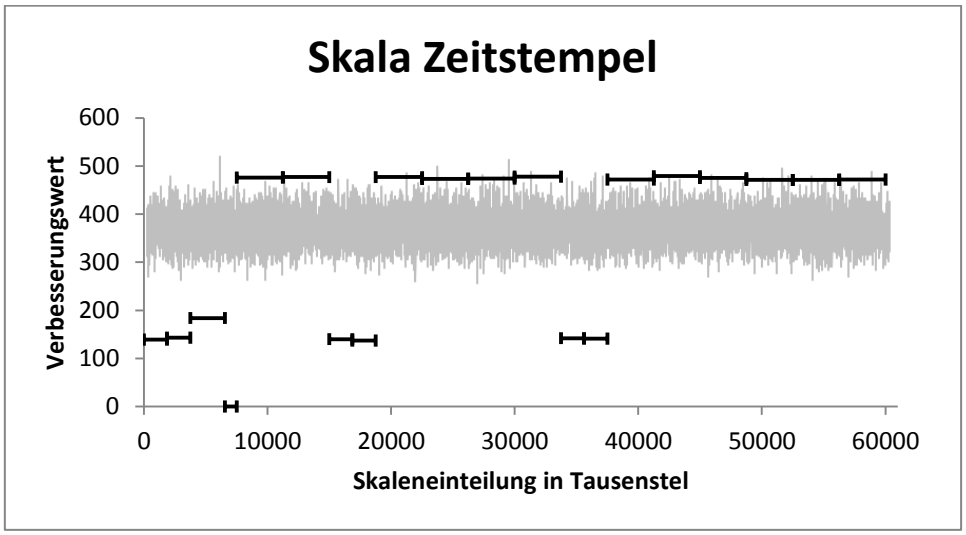

Abb. 6-5: Skala Zeitstempel, 25. Zyklus, 20 Intervalle

In Abb. 6-6 ist die Anzahl der Bucketzugriffe je Zyklus dargestellt. Dies ist ein Maß für die durchschnittliche Anfragezeit. Innerhalb der ersten 10 Zyklen reduziert sich die Anzahl der durchsuchten Buckets auf ca. 2100. Es werden demnach mehr als doppelt so viele Buckets durchsucht wie im Standard Grid-File. Dagegen ist allerdings die Größe des Grid-Directorys von 60840 auf 8000 gesunken was 13% der Größe entspricht.

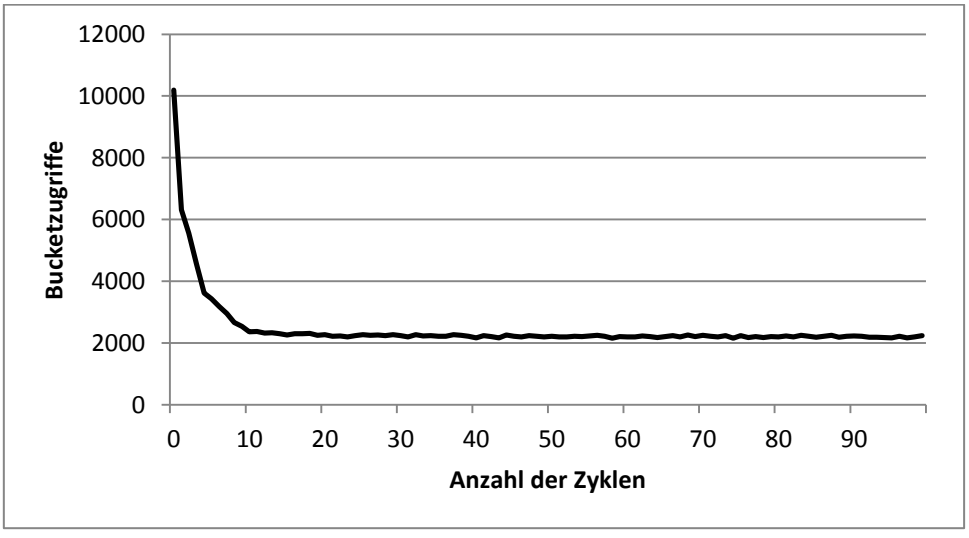

Abb. 6-6: Verteilung Konstant in allen Skalen

Anfrageverteilung Säule in allen Skalen

Bei der Anfrageverteilung Säule ist zu erwarten, das sich an den Rändern zwei große Intervalle bilden, die den nicht angefragten Datenbereich entsprechen. Die Datenbereiche, auf denen zugegriffen wird sollten dagegen gut strukturiert werden. Abb. 6-7 zeigt den 25. Zyklus für die Skala Zeitstempel. Der komplette Skalenüberblick für die Zyklen 25 und 100 ist in Anhang A.5 und A.6 dargestellt. Wie zu erkennen ist, werden die Trennwerte entsprechend der Verteilung verlegt.

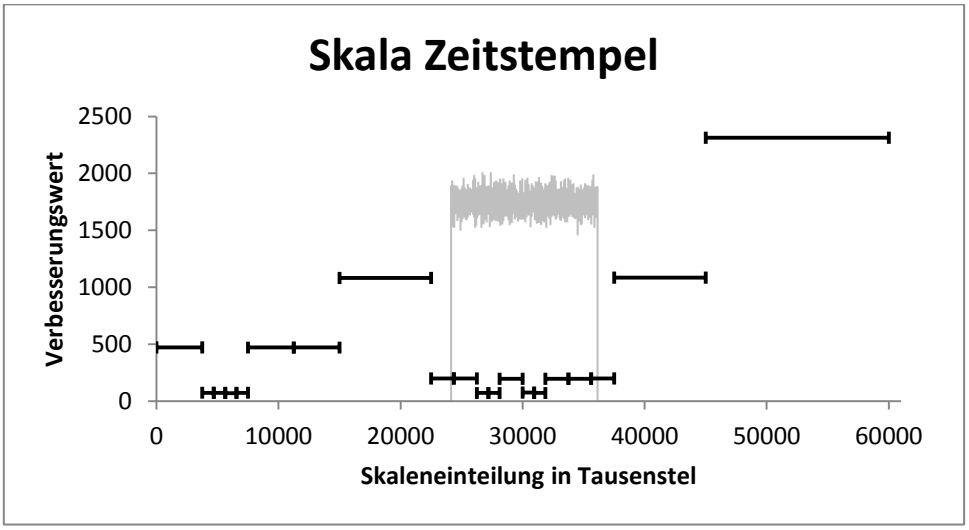

Abb. 6-7: Skala Zeitstempel, 25. Zyklus, 20 Intervalle

Abb. 6-8 stellt die Anzahl der Bucketzugriffe dar. Wie bereits bei der Anfrageverteilung Konstant sinkt die Anzahl der Bucketzugriffe rapide innerhalb der ersten 10 Zyklen. Allerdings wird hier ein Wert von knapp über eintausend Bucketzugriffen erreicht. Die Anfragezeit ist demnach entsprechend der im Standard Grid-File.

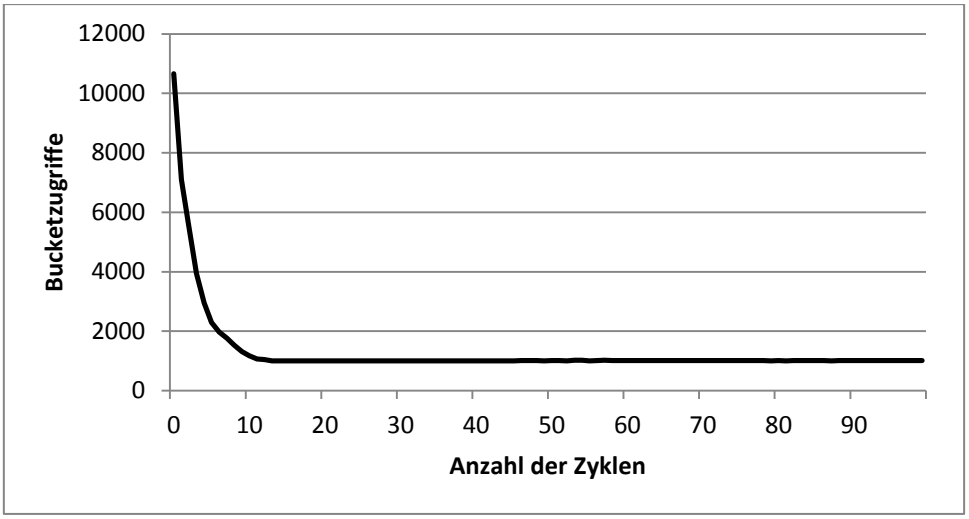

Abb. 6-8: Verteilung Säule in allen Skalen

Anfrageverteilung Gauss in allen Skalen

Für Anfrageverteilung Gauss ist eine Häufung der Trennwerte im oberen Drittel der Skaleneinteilung zu erwarten, da hauptsächlich hier Anfragen stattfinden. Abb. 6-9 stellt die Skala Zeitstempel nach dem 25. Zyklus dar. Der komplette Skalenüberblick für die Zyklen 25 und 100 ist in Anhang A.7 und A.8 dargestellt. Gleichfalls ist in diesem Fall eine Anpassung an die Anfrageverteilung zu erkennen.

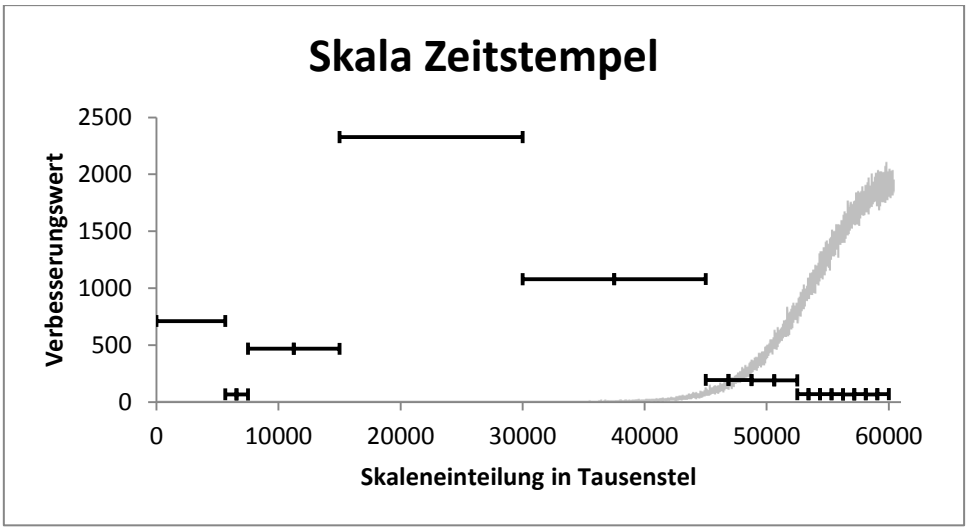

Abb. 6-9: Skala Zeitstempel, 25. Zyklus, 20 Intervalle

In Abb. 6-10 sinkt die Bucketauslastung wie in den vorherigen Experimenten innerhalb der ersten 10 Zyklen. Wie bei der Verteilung Säule wird hier ein Wert von ca. eintausend Bucketzugriffen je Zyklus erreicht.

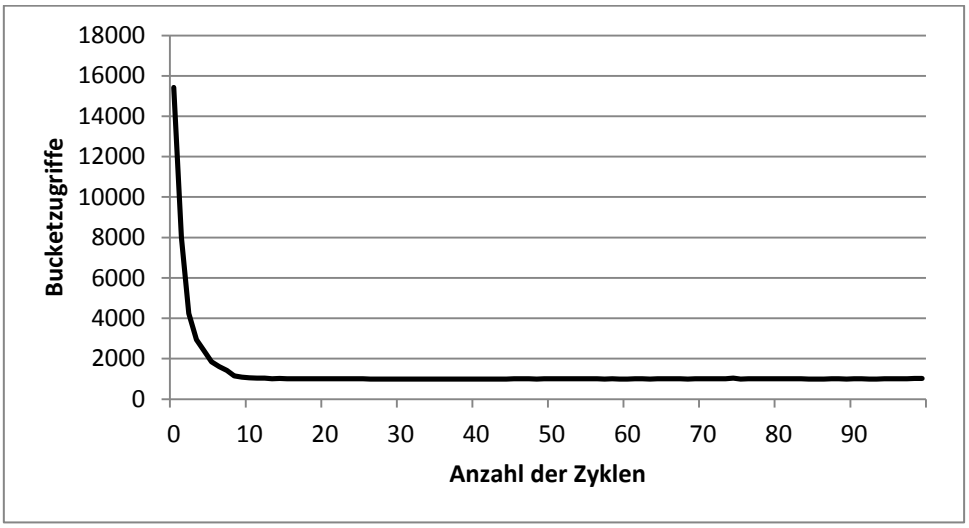

Abb. 6-10: Verteilung Gauss in allen Skalen

Anfrageverteilung Gemischt

Die Gemischte Anfrageverteilung dient zur Ermittlung in wie weit sich verschiedene Anfrageverteilungen auf verschiedene Attribute auswirken. Anhang A.9 und A.10 stellen die Einteilung der Skalen im 25. und 100. Zyklus dar. Die Skalen werden an ihre jeweilige Anfrageverteilung angepasst. Abb. 6-11 zeigt die Anzahl der Bucketzugriffe. Hierbei sinkt die Anzahl ebenfalls innerhalb der ersten 10 Zyklen, hat aber noch nicht das Optimum erreicht. Bei gemischter Anfrageverteilung liegen die zu verfeinernden Stellen nicht in den gleichen Skalenabschnitten. Die Trennwertverlegung in einer Skala unterstützt somit nicht die Verbesserung der anderen Skalen. Somit dauert bei Verlegung von ausschließlich einem Trennwert das Erreichen des Optimums länger. In diesem Fall 15 Zyklen.

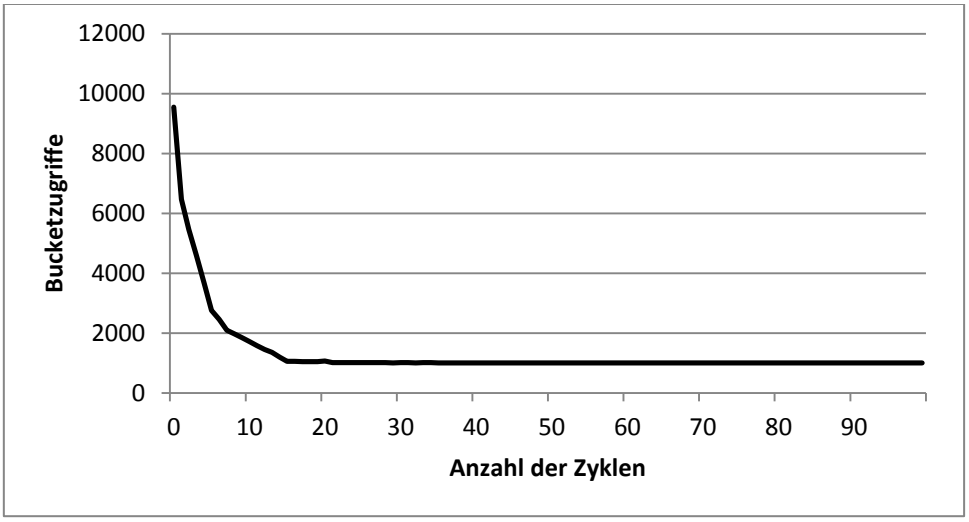

Abb. 6-11: Verteilung Gemischt

Anfrageverteilung Variabel

Die Anfrageverteilung Variabel soll zeigen, wie sich die Anzahl der Bucketzugriffe unter wechselnden Anfragesituationen verhält. Abb. 6-12 zeigt das Ergebnis für eine Skalengröße von 20. Jeweils zum 100., 200., 300. und 400. Zyklus ändert sich die Anfrageverteilung entsprechend der obig gemachten Angaben. Die Reaktionen auf die ersten drei Wechsel fallen relativ gering aus. Erst der vierte Wechsel, erzeugt einen gewaltigen Anstieg der Zugriffszahl auf über 170.000. Dieser sinkt ebenso rapide wieder ab. Zu erklären ist das Verhalten mit der Anfrageverteilung vor und nach dem Anstieg. Vorher waren alle drei Verteilungen Gauss, was zu langen Bucketlisten am unteren Ende der Skalen führt. Bei der Veränderung der Anfrageverteilung auf Konstant, bzw. Säule wird auf diese Bucketlisten zugegriffen, wodurch sich die hohe Anzahl an Bucketzugriffen erklärt.

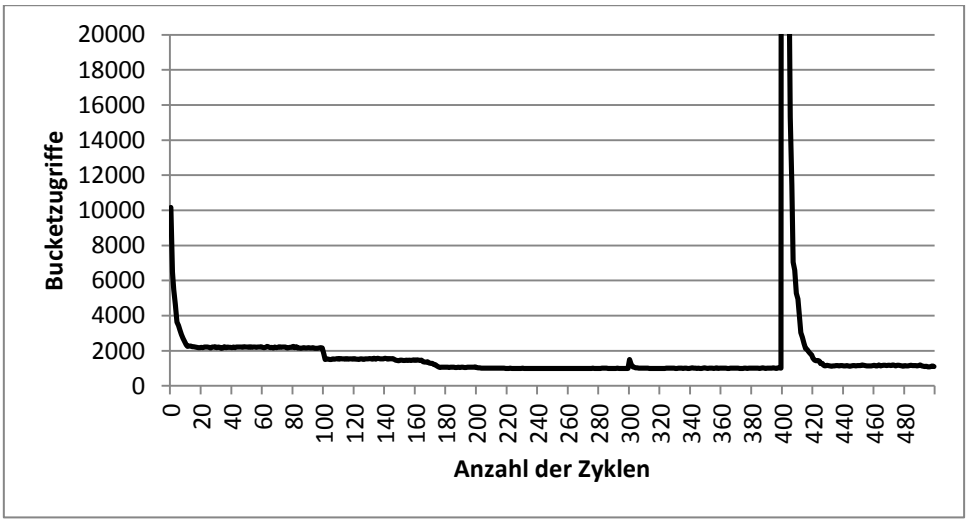

Abb. 6-12: Verteilung variabel

Vergleich der verschiedenen Skalengrößen

Neben den Ergebnissen für die Skalengröße von 20 Intervallen wurden die gleichen Experimente ebenfalls mit den anderen Skalengrößen durchgeführt. Das Verhalten änderte sich mit keinem der Fälle. Einzig die Anzahl der Bucketzugriffe nahm für kleinere Skalengrößen zu, da eine Aufteilung auf weniger Intervalle durchgeführt wurde. Es zeigte sich, dass bereits mit einer Skalengröße von 15 annähernd eine Anzahl an Bucketzugriffen von eintausend erreicht wird (ausgenommen bei der Verteilung Konstant). Allerdings sind mehr Zyklen als bei der Skalengröße von 20 notwendig, um diesen Wert zu erreichen.

Zusammenfassend lässt sich sagen, dass mit Erhöhung der Anzahl der Skalenintervalle die Länge der Bucketlisten sinkt, da diese besser aufgeteilt werden können. Gleichzeitig werden

durch das Self-Tuning die Trennwerte so verlegt, dass eine möglichst geringe Anzahl an Buckets angefragt wird. Dies führt dazu, dass ab einer bestimmten Skalengröße die Anfragegeschwindigkeit des Standard Grid-Files erreicht wird. Dies gilt allerdings nicht für die Anfrageverteilung Konstant, da für diesen Fall das Standard Grid-File das Optimum darstellt.

Weitere Ergebnisse

Neben den guten Ergebnissen, die bisher präsentiert wurden, existiert auch ein Nachteil. Dieser liegt im Abbruchwert begründet, der mit 0,1 zu gering gewählt wurde. So ist in allen Fällen (außer Anfrageverteilung Konstant, Gemischt und Variabel bei Skalengröße 32) eine fortlaufende Verlegung von Trennwerten zu beobachten ist, die teilweise zu einem erneuten Anstieg der Anzahl der Bucketzugriffe führt. Der Abbruchwert ist zu gering gewählt, um zyklische Trennwertverlegungen zu verhindern. Eine pauschale Erhöhung ist ebenfalls nicht sinnvoll, da wie in den Diagrammen dargestellt die Verbesserungswerte in den einzelnen Experimenten stark variierten. Es muss somit ein neues Maß gefunden werden, um die Trennwertverlegung bei guten Ergebnissen verhindern zu können.

Wie obig erwähnt kommt es im Fall der Skalengröße von 32 zu einem Abbruch. Dies rührt daher, dass die Trennwerte hier so verlegt werden, dass ein Standard Grid-File entsteht. Wie in Tab. 6-2 gezeigt, hat das Standard Grid-File für die Datenbank A bei gemischten Werten die Skalengrößen 32, 32 und 31. Somit ist die Größenbeschränkung unterschritten. Es ist demnach möglich, die negativen Auswirkungen der Indizierung eines Primärschlüssels auf die Größe des Grid-Directorys mit Hilfe einer konstanten Anfrageverteilung zu beseitigen.

Als weitere Beobachtung kann die Annahme bestätigt werden, dass beim last-balancierten Grid-File die Bucketauslastung zunimmt. So steigt sie bei einer Skalengröße von 8 bis auf 97 %. Dies ist auf die großen Bucketlisten zurückzuführen, die in diesem Fall auftreten. Die Bucketauslastung geht entsprechend bei einer Zunahme von Skalenintervallen und schrumpfenden Bucketlisten zurück.

7 Zusammenfassung und Ausblick

In der Arbeit wurde untersucht, wie das Grid-File nach Nievergelt et al. weiterentwickelt werden kann, um das Ziel einer Größenbeschränkung des Indexes zu ermöglichen. Hierzu wurde als Maß für die Wichtigkeit der Daten die Anfragehäufigkeit genutzt. Zur Umsetzung kamen zwei Konzepte in Frage, partielle Indexe und last-balancierte Indexe. Nachdem die Entscheidung für ein last-balanciertes Grid-File getroffen wurde, ist dieses theoretisch umgesetzt und später als Prototyp realisiert wurden. Im Anschluss ist das Konzept mit generierten Zufallsverteilungen getestet wurden.

Als Ergebnis dieser Evaluierung kann zusammenfassend gesagt werden, das das grundsätzliche Konzept eines last-balancierten Grid-Files zu guten Ergebnissen bezüglich der Anzahl der Bucketzugriffe führt. Diese nähern sich für stark angefragte Daten dem Optimum des Standard Grid-Files an, bei dem nur ein Bucket bei Punktanfragen aufgerufen wird. Gleichzeitig kann die Größe des Grid-Directorys sehr stark eingeschränkt werden. Ebenso ist das last-balancierte Grid-File robust gegenüber wechselnden Anfrageverteilungen, bzw. unterschiedlichen Anfrageverteilungen auf verschiedene Attribute. Das last-balancierte Grid-File hat somit entscheidende Vorteile gegenüber dem Standard Grid-File. Einzig die Definition eines Abbruchwertes, der theoretisch eine Trennwertverlegung bei einer günstigen Verteilung verhindern sollte, schlug fehl. Dies führte zu zyklischen Trennwertverlegungen, die keine Verbesserung und zum Teil gar eine Verschlechterung der Ergebnisse bewirkten. Dieses Problem muss in der Zukunft gelöst werden, um das Konzept effektiv einsetzen zu können.

Als Nebenuntersuchung wurde ein Vergleich zwischen der Trennwertermittlung mit dem Mittelwert des Intervalls und dem Median der Daten vorgenommen. Dies war notwendig um zu klären, ob mit einer neuen Methode zur Trennwertbestimmung das Wachstum des Grid-Files verlangsamt werden kann. Dabei stellte sich heraus, dass der Mittelwert eine wesentlich größere Schwankungsbreite in der Bucketauslastung, sowie eine ausgeprägte Stufenfunktion bezüglich der Größe des Grid-Directorys besitzt. Des Weiteren sind die negativen Auswirkungen eines Primärschlüssels sowie eines offenen Datenintervalls gezeigt wurden. Somit ist die Verwendung des Medians gegenüber der des Mittelwertes vorzuziehen.

Als offene Frage ergibt sich daraus, in wie weit die Ergebnisse variieren, wenn anstatt des Mittelwertes der Median der Daten zur Trennwertbestimmung im last-balancierten Grid-File

genutzt wird. Dabei ist zu bedenken, dass der Aufwand zur Ermittlung des Medians einer Bucketliste entsprechend mit der Länge der Liste wächst.

Es wurde bisher ausschließlich mit statischen Datenbanken gearbeitet. Neben dem Problem des Abbruchwertes wird deshalb zu klären sein, ob das Konzept mit Einfüge- und Löschoperationen von Datensätzen ebenfalls funktioniert. Des Weiteren wäre eine Umsetzung in ein DBMS interessant, um die Eigenschaften des last-balancierten Grid-Files unter realen Bedingungen untersuchen zu können.

Als weitere offene Aufgabe ergibt sich das Problem der Größe des Grid-Files. Es wurde festgestellt, dass die Bucketzugriffe mit dem Konzept des last-balancierten Grid-Files bei ungleichmäßigen Anfrageverteilungen auf das gleiche Niveau wie beim Standard Grid-File gesenkt werden können. Daher ergibt sich die Frage, bei welcher Größe dies erreicht wird, bzw. ob diese Größenbestimmung automatisch erfolgen kann und somit nur Grenzwerte vorgegeben werden müssen.

Abschließend kann darüber nachgedacht werden, das last-balancierte Grid-File durch das Konzept der partiellen Indexe zu ergänzen. Dies könnte erfolgen, indem nicht angefragte Datenbereiche aus dem Index gelöscht werden. Weiterhin müsste eine Bewertung der nicht indizierten Bereiche erfolgen, so dass bei Änderungen der Anfrageverteilung diese wieder hinzugefügt werden können. Es wäre so zur Laufzeit aufgrund der Anfragen möglich, Datenbereiche kontinuierlich oder in Intervallen zu archivieren.

Literaturverzeichnis

- [Ap04] M.Apell; The Grid File; studentische Ausarbeitung; Universität Konstanz; http://www.inf.uni-konstanz.de/dbis/teaching/ws0304/datatypes/download/paper-grid-file.pdf (letzter Aufruf am 13.03.2011)
- [ChNa07] S. Chaudhuri, V. Narasayya; Self-tuning database systems: A decade of progress; 2007
- [ChWe06] S. Chaudhuri, G. Weikum; Foundations of Automated Database Tuning; Tutorial; 2006; http://icde06.ewi.utwente.nl/cwicde06.pdf; (letzter Abruf am 13.03.2011)
- [ElNa02] R.A. Elmasri, S. B. Navathe; Grundlagen von Datenbanksystemen; 3. Edition Pearson Studium; 2002
- [FaHa96] L. Fahrmeir, A. Hamerle, G. Tutz; Multivariate Statistische Verfahren; Walter de Grypter & Co; 2. Erweiterte Auflage; 1996
- [GaGu98] V. Gaede, O. Günther; Multidimensional access methods, ACM Computing Surveys, 30 (2); S. 170 – 231, 1998
- [Gi08] M. Giard; Konzepte zum Index-Self-Tuning auf der Basis partieller Indexe; Diplomarbeit Otto-von-Guericke-Universität Magdeburg; Fakultät für Informatik; Institut für technische und betriebliche Informationssysteme; 2008
- [Ha87] T. Härder; Realisierung von operationalen Schnittstellen; In: P. C. Lockemann; Schmidt, J. W. (Hrsg); Datenbank-Handbuch, S. 163-335, Springer Verlag, Berlin 1987
- [HaRa99] T. Härder, E. Rahm; Datenbanksysteme – Konzepte und Techniken der Implementierung; Springer Verlag Berlin 1999
- [Hi85] K. Hinrichs, Implementation of the Grid File: Design Concepts and Experience; BIT 25, 1985, S. 569-592

- [HiLo11] M. Hilbert, P. Lopez; The World's Technological Capacity to Store, Communicate and Compute Information; *Science DOI:* 10.1126/science.1200970; Feb. 2011

- [HuSi88] A. Hutflesz, H.-W. Six, P. Widmayer; Globally order preserving multidimensional linear hashing; in Proceeding of the Fourth IEEE International Conference on Data Engineering; S. 572-579; 1988

- [Hü96] G. Hübner; Stochastik – Eine anwendungsorientierte Einführung für Informatiker, Ingenieure und Mathematiker; Vieweg Verlag; 1996

- [Ib06] IBM; An architectural blueprint for autonomic computing; White Paper, June 2006; Fourth Edition.

- [LüSa07] M. Lühring, K.-U. Sattler, E. Schallehn, K. Schmidt; Autonomes Index Tuning – DBMS integrierte Verwaltung von Soft Indexen; BTW 2007 S. 152-171

- [Mu08] S. Mundt; Last-balancierte Indexstrukturen; Diplomarbeit Otto-von-Guericke-Universität Magdeburg; Fakultät für Informatik; Institut für technische und betriebliche Informationssysteme; 2008

- [NiHi84] J.Nievergelt, H. Hinterberger; The Grid File: An Adaptable, Symmetric, Multikey File Structure; ACM Transactions on Database Systems, Vol. 9, No. 1, März 1984, S. 38-71

- [RePo06] P. Rechenberg, G. Pomberger; Informatikhandbuch; Carl Hanser Verlag München; 4. Auflage 2010

- [SaGe03] K. Sattler, I. Geist, E.Schallehn, QUIET: Continuous Query-driven Index Tuning, Proceedings of the 29th VLDB Conference Berlin S. 1129-1132, 2003

- [SaHe05] G.Saake, A. Heuer, K.-U. Sattler; Datenbanken: Implementierungstechniken; mitp-Verlag Bonn 2005; 2. Auflage

- [SaSc04] K. Sattler, E. Schallehn, I. Geist; Autonomous Query-driven Index Tuning; Proc. Int. Database Engineering and Applications Symposium (IDEAS 2004) in Coimbra, Portugal 2004, S. 439–448

- [SaSc05] K. Sattler, E. Schallehn, I. Geist; Towards Indexing Schemes for Self-Tuning DBMS; ICDE Workshops 2005, S. 1216

- [Se06] R. Sedgewick; Algorithmen in C; Pearsum Studium; 2005
- [ShBo03] D. Shasha, P. Bonnet; Database Tuning – Principles, Experiments and Troubleshooting techniques; Morgan Kaufmann Publishers; 2003
- [St89] M. Stonebraker; The case for partial indexes; SIGMOD Record, Band 18, Nr. 4, Dezember 1989
- [WeHa94] G. Weikum, C. Hasse, A. Mönkeberg, P. Zabback; The COMFORT Automatic Tuning Project, Invited Projec Review; Informaton Systems; 19(5) S. 381 – 432; 1994
- [WeMö02] G. Weikum, A. Mönkeberg, C. Hasse, P. Zabback, Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering, Proceedings of the 28th VLDB Conference, 2002

Anhang

A.1 Aufbau der Auswertungsdatei

LongSkala: <*Anzahl der Trennwerte in der Skala Zeitstempel*>
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>
…
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>

IntSkala: <*Anzahl der Trennwerte in der Skala Zahl*>
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>
…
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>

StringSkala: <*Anzahl der Trennwerte in der Skala Text*>
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>
…
<*Trennwert*>; <*Verbesserungswert*>; <*Anfragezähler*>

<*Gesamtanzahl der Buckets*> <*Bucketauslastung*> <*Größe des Grid-Directorys*>

<*Skala*> Trennwert löschen: <*zu löschender Trennwert*>
<*Skala*> Trennwert einfügen: <*einzufügender Trennwert*>
Anzahl der durchsuchten Buckets: <*Bucketanzahl auf die zugegriffen wurde*>

A.2 Ausgangssituation nach Einfügen

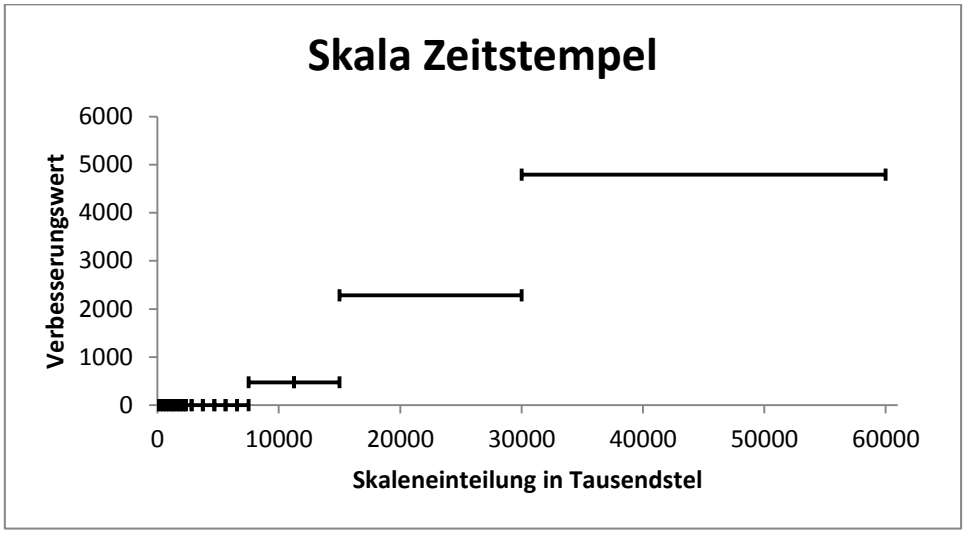

Abb. A-1: Zeitstempel, 0. Zyklus, 20 Intervalle

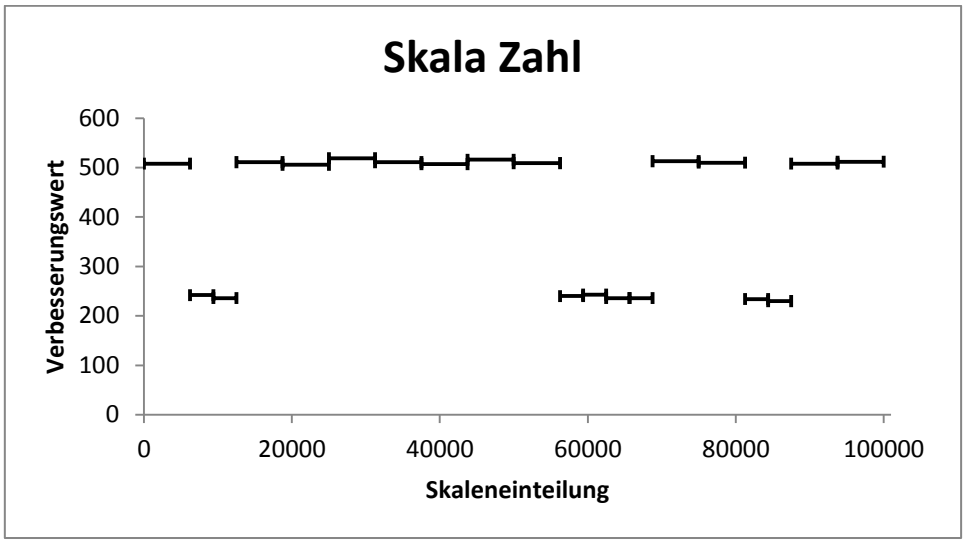

Abb. A-2: Zahl, 0. Zyklus, 20 Intervalle

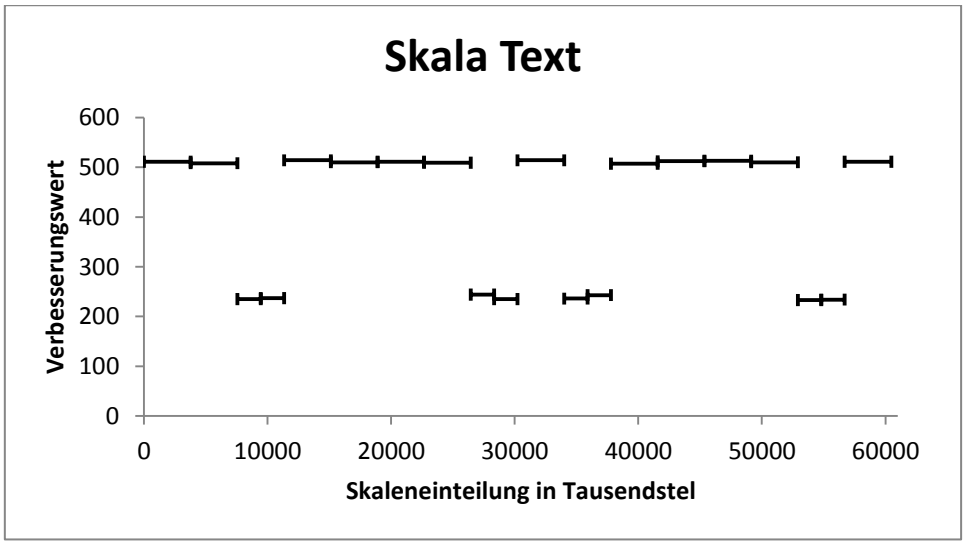

Abb. A-3: Text, 0. Zyklus, 20 Intervalle

A.3 Verteilung Konstant in den Anfragen aller Skalen, 25. Zyklus

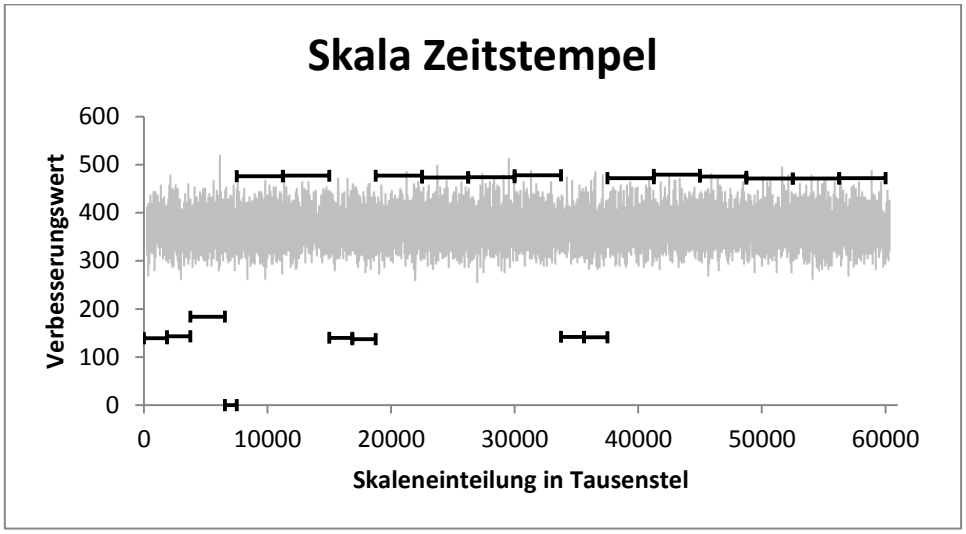

Abb. A-4: Zeitstempel, 25. Zyklus, 20 Intervalle, Konstant

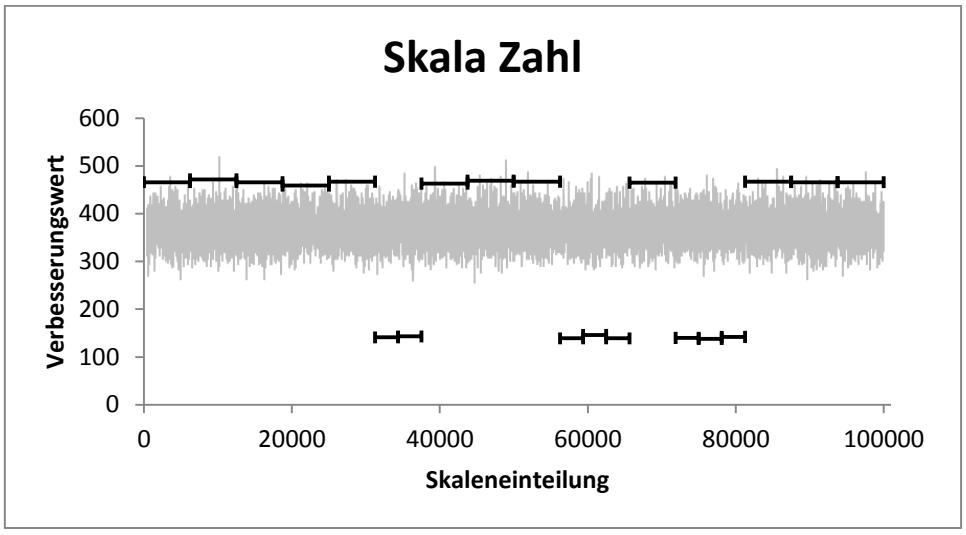

Abb. A-5: Zahl, 25. Zyklus, 20 Intervalle, Konstant

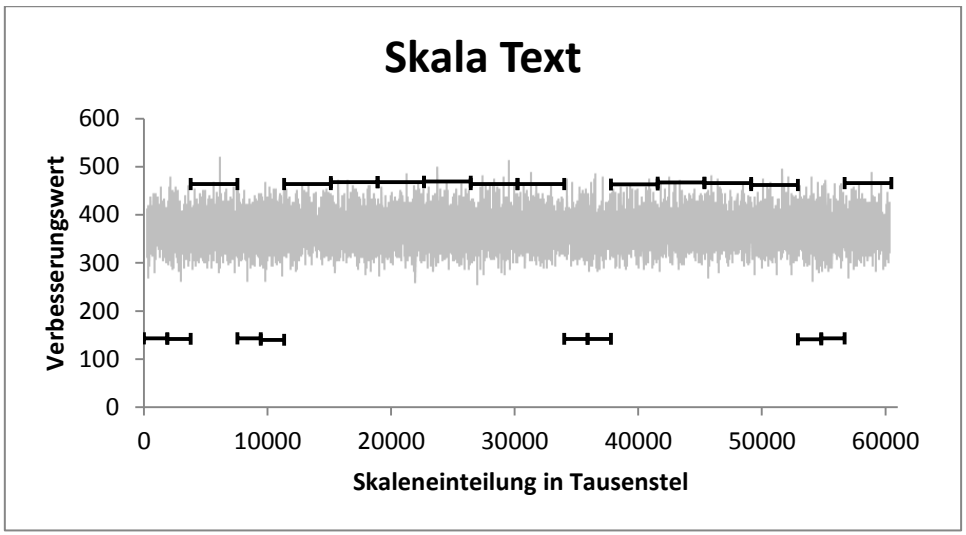

Abb. A-6: Text, 25. Zyklus, 20 Intervalle, Konstant

A.4 Verteilung Konstant in den Anfragen aller Skalen, 100. Zyklus

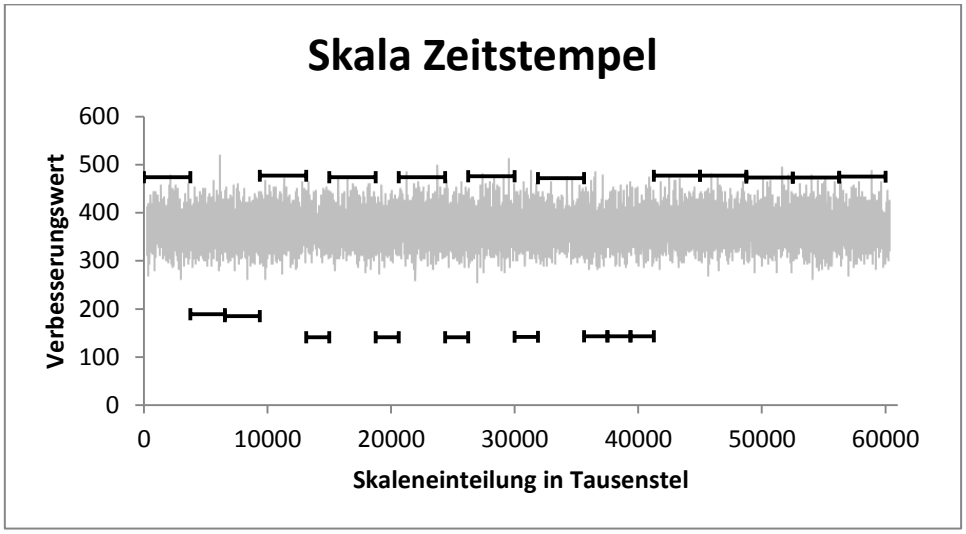

Abb. A-7: Zeitstempel, 100. Zyklus, 20 Intervalle, Konstant

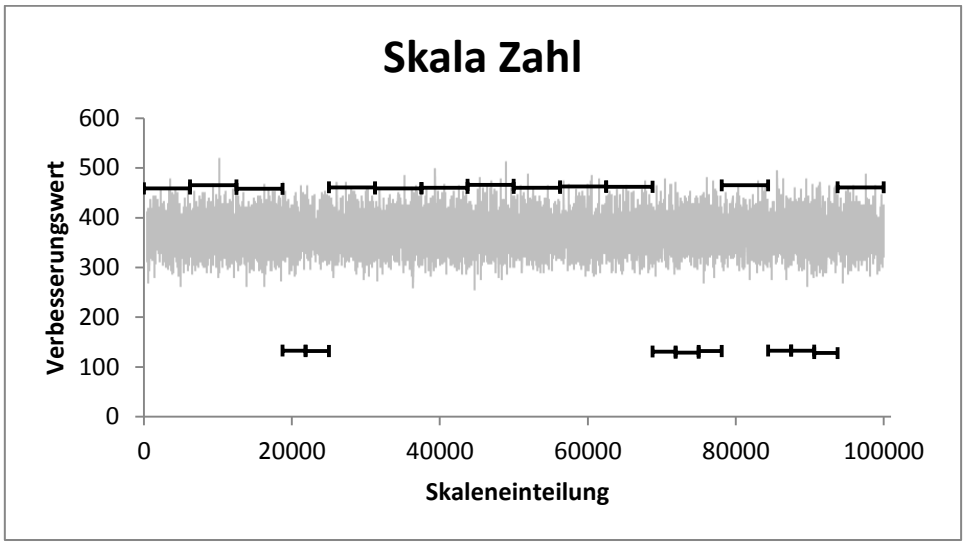

Abb. A-8: Zahl, 100. Zyklus, 20 Intervalle, Konstant

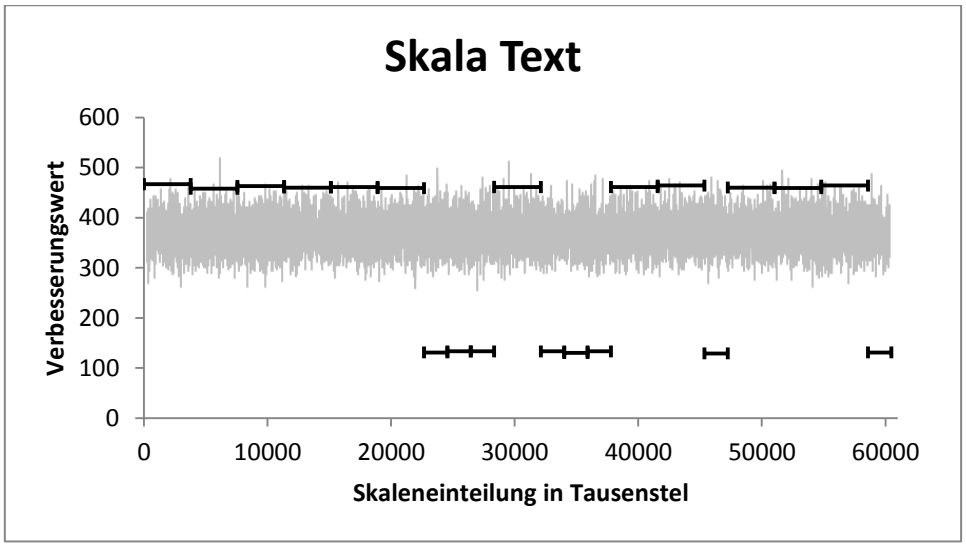

Abb. A-9: Text, 100. Zyklus, 20 Intervalle, Konstant

A.5 Verteilung Säule in den Anfragen aller Skalen, 25. Zyklus

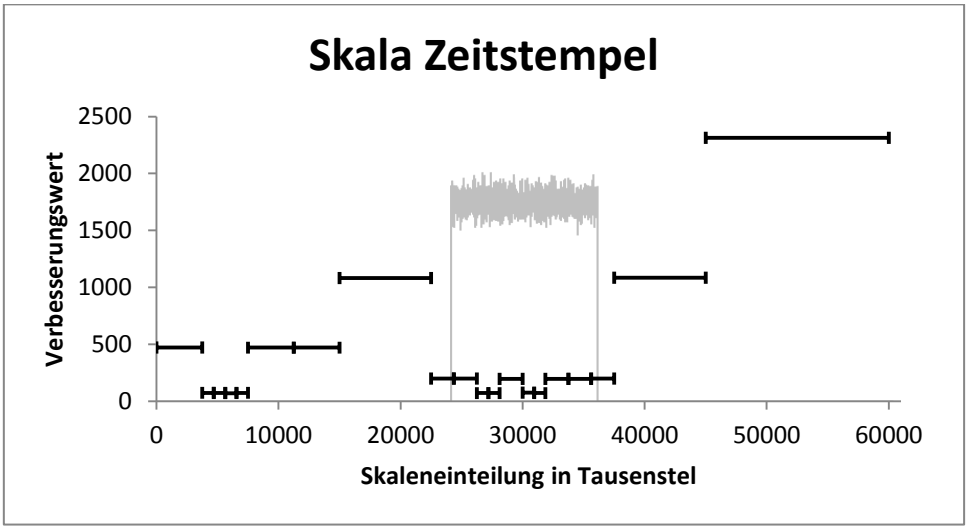

Abb. A-10: Zeitstempel, 25. Zyklus, 20 Intervalle, Säule

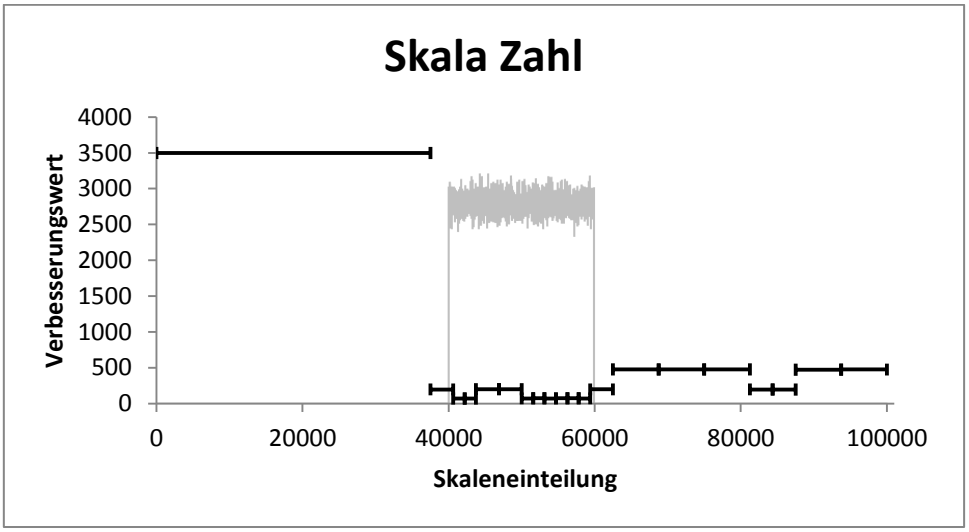

Abb. A-11: Zahl, 25. Zyklus, 20 Intervalle, Säule

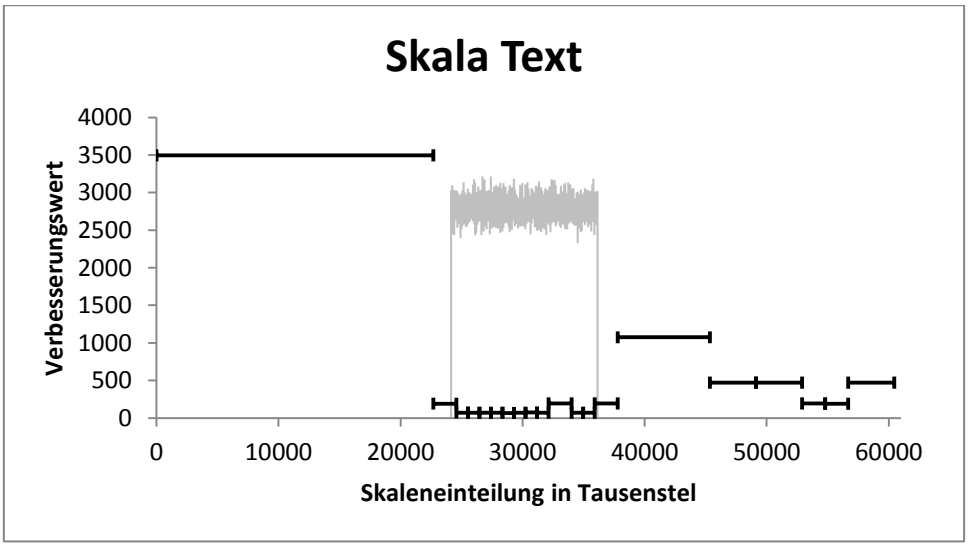

Abb. A-12: Text, 25. Zyklus, 20 Intervalle, Säule

A.6 Verteilung Säule in den Anfragen aller Skalen, 100. Zyklus

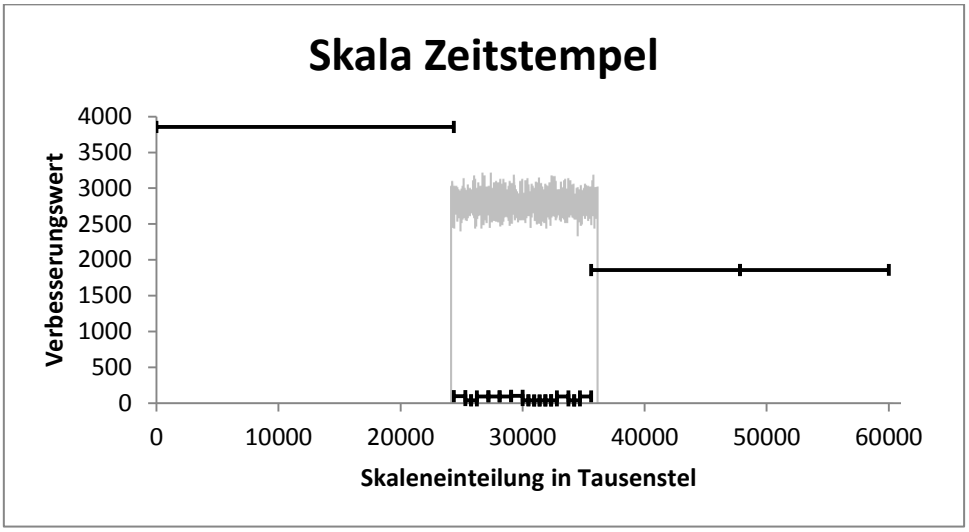

Abb. A-13: Zeitstempel, 100. Zyklus, 20 Intervalle, Säule

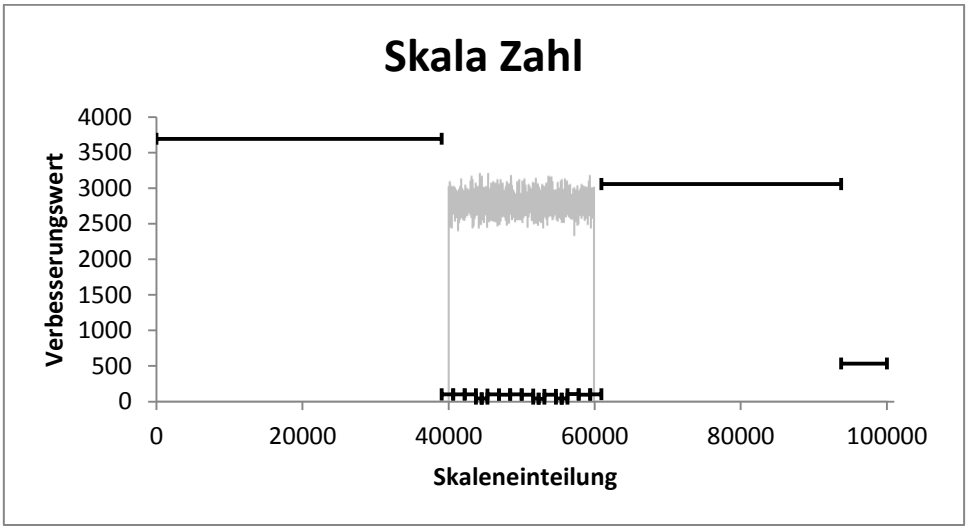

Abb. A-14: Zahl, 100. Zyklus, 20 Intervalle, Säule

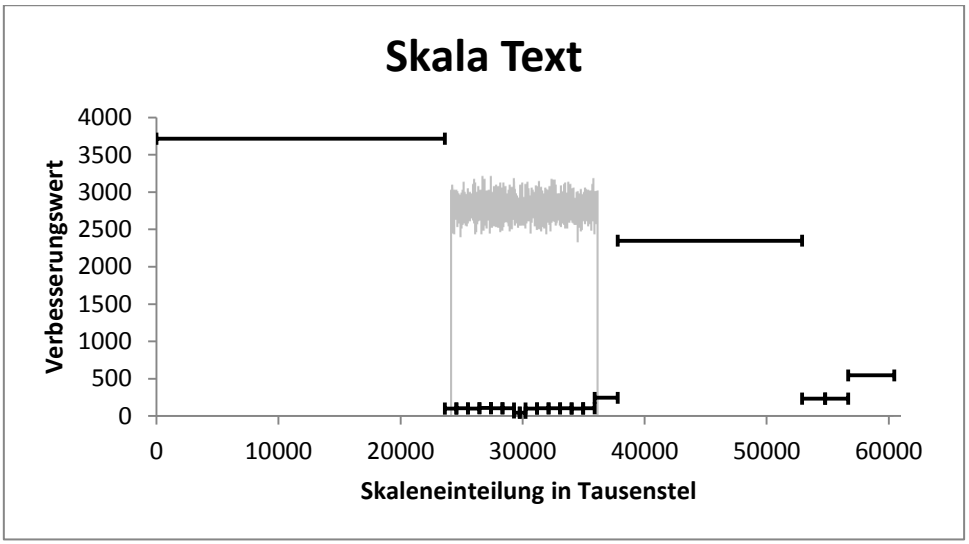

Abb. A-15: Text, 100. Zyklus, 20 Intervalle, Säule

A.7 Verteilung Gauss in den Anfragen aller Skalen, 25. Zyklus

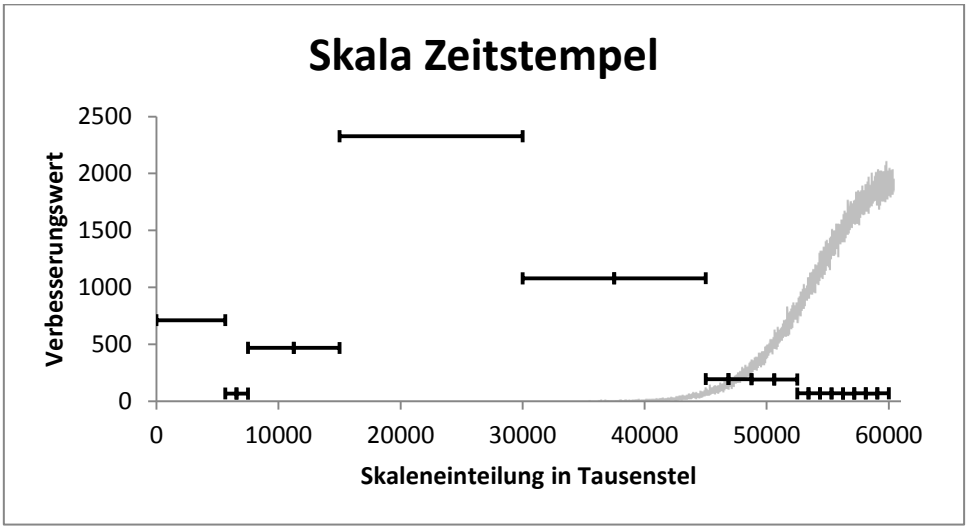

Abb. A-16: Zeitstempel, 25. Zyklus, 20 Intervalle, Gauss

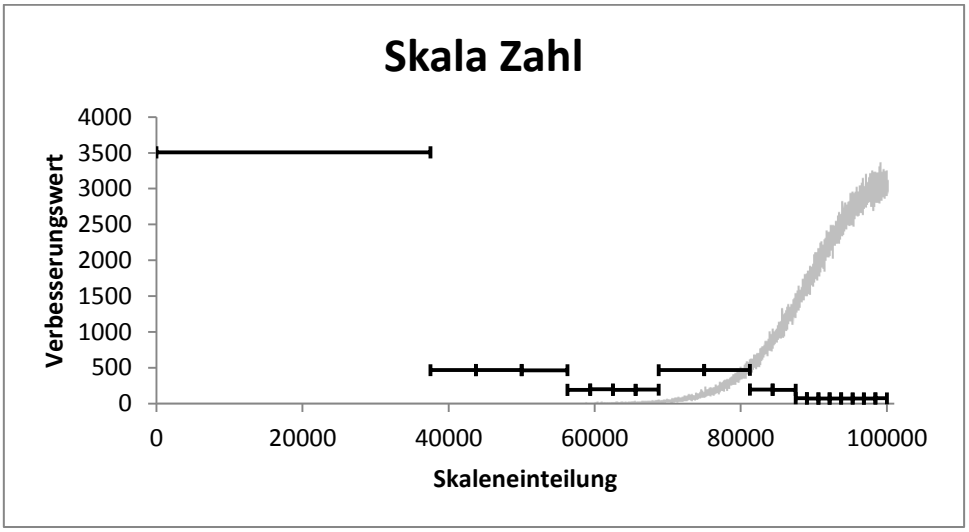

Abb. A-17: Zahl, 25. Zyklus, 20 Intervalle, Gauss

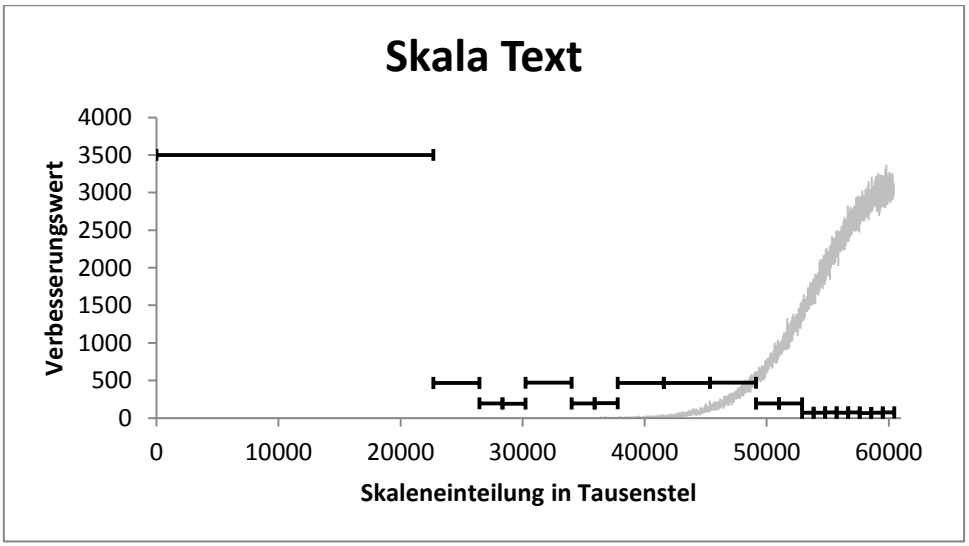

Abb. A-18: Text, 25. Zyklus, 20 Intervalle, Gauss

A.8 Verteilung Gauss in den Anfragen aller Skalen, 100. Zyklus

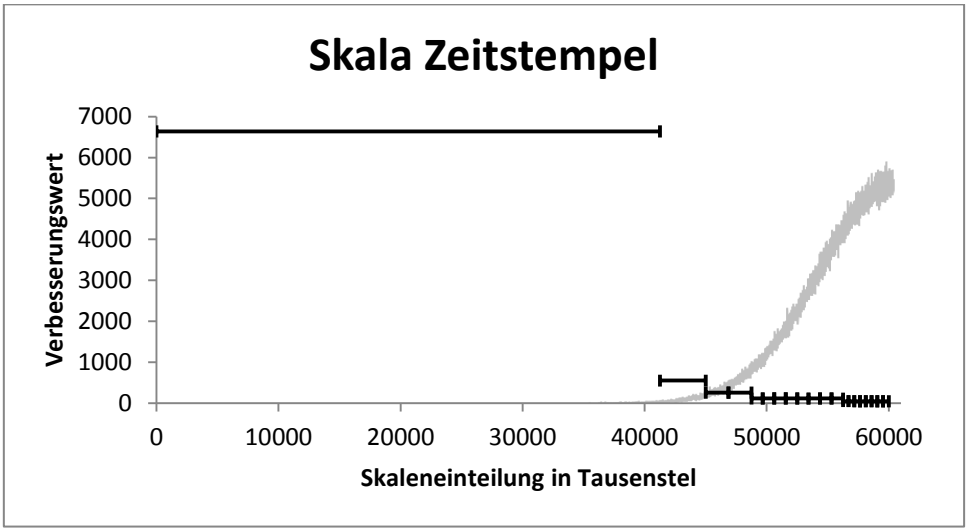

Abb. A-19: Zeitstempel, 100. Zyklus, 20 Intervalle, Gauss

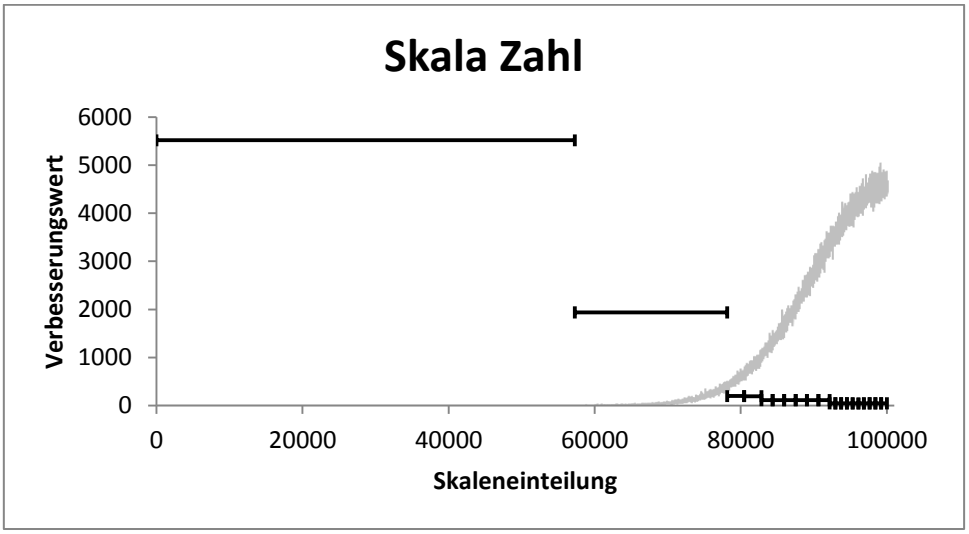

Abb. A-20: Zahl, 100. Zyklus, 20 Intervalle, Gauss

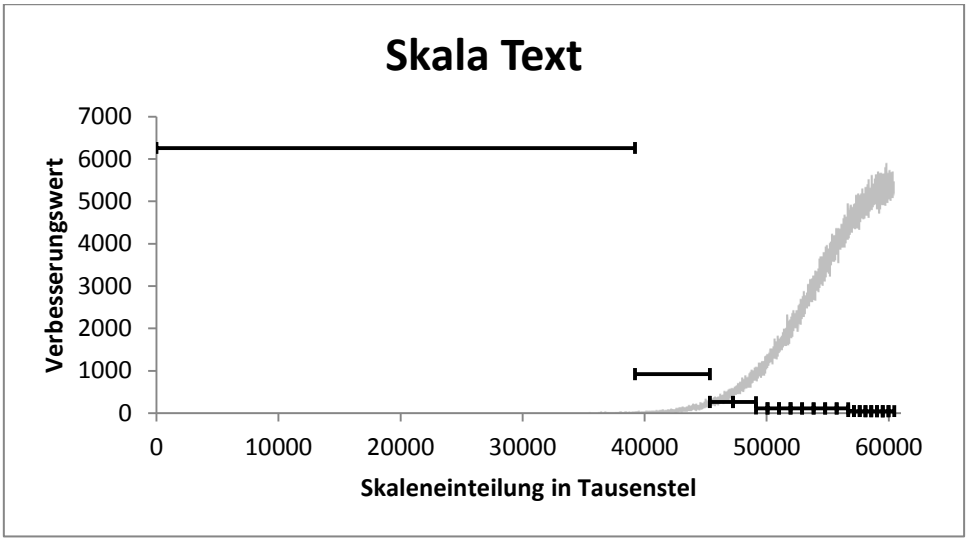

Abb. A-21: Text, 100. Zyklus, 20 Intervalle, Gauss

A.9 Gemischte Verteilungen in den Anfragen aller Skalen, 25. Zyklus

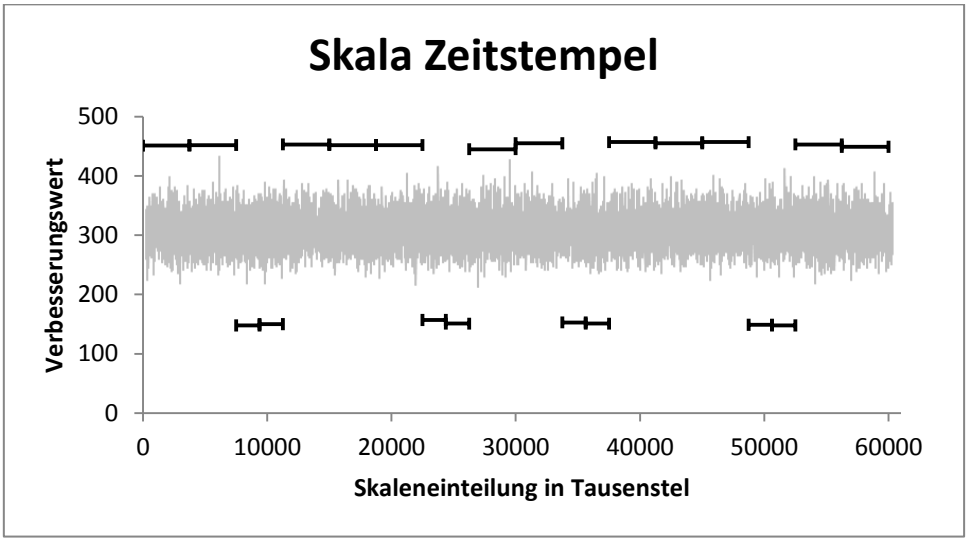

Abb. A-22: Zeitstempel, 25. Zyklus, 20 Intervalle, Konstant

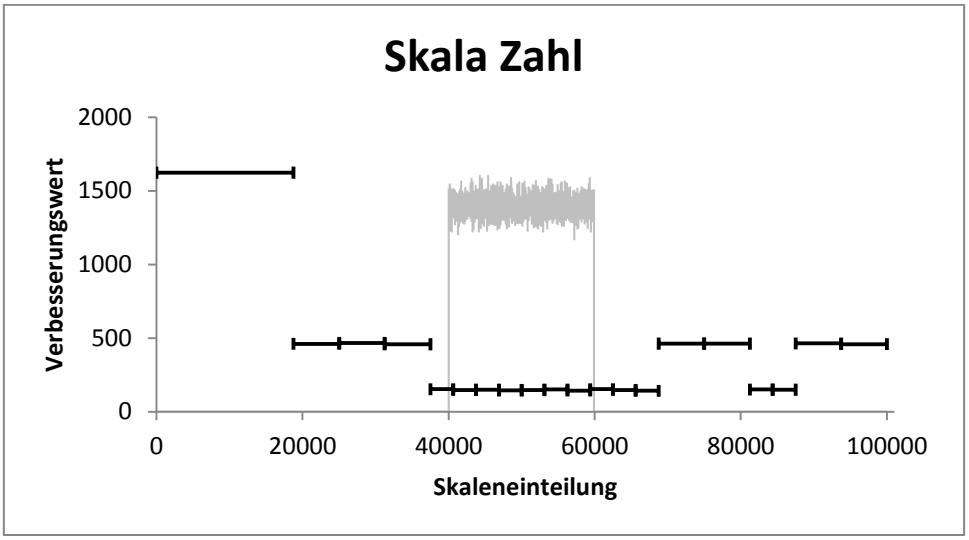

Abb. A-23: Zahl, 25. Zyklus, 20 Intervalle, Säule

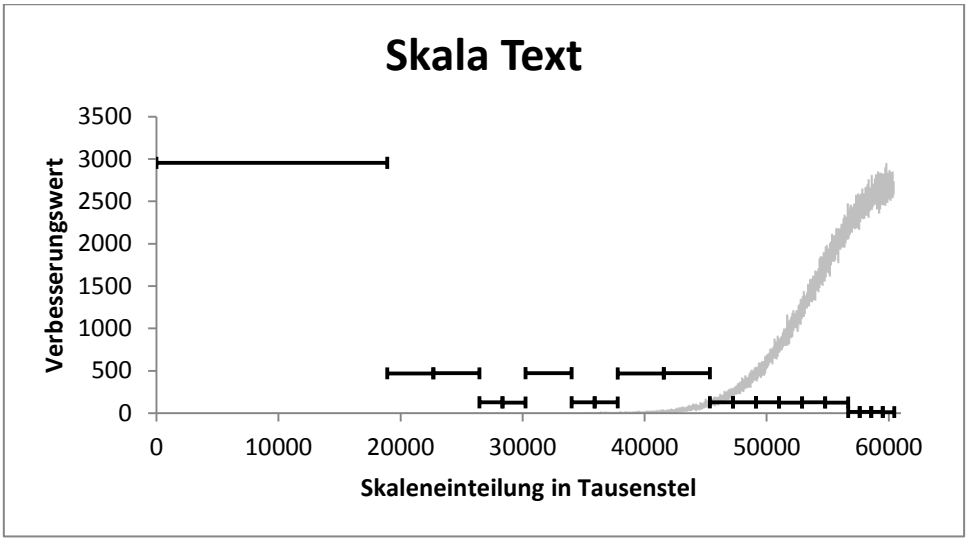

Abb. A-24: Text, 25. Zyklus, 20 Intervalle, Gauss

A.10 Gemischte Verteilungen in den Anfragen aller Skalen, 100. Zyklus

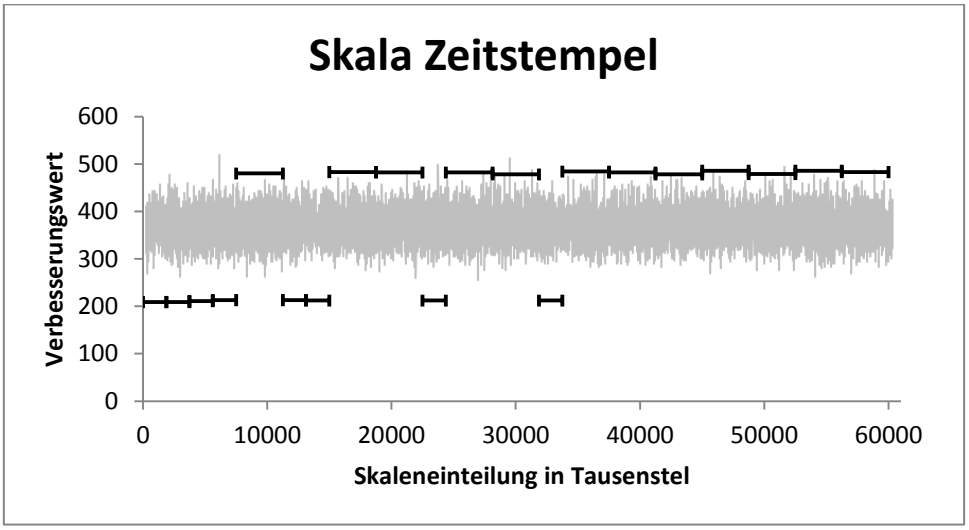

Abb. A-25: Zeitstempel, 100. Zyklus, 20 Intervalle, Konstant

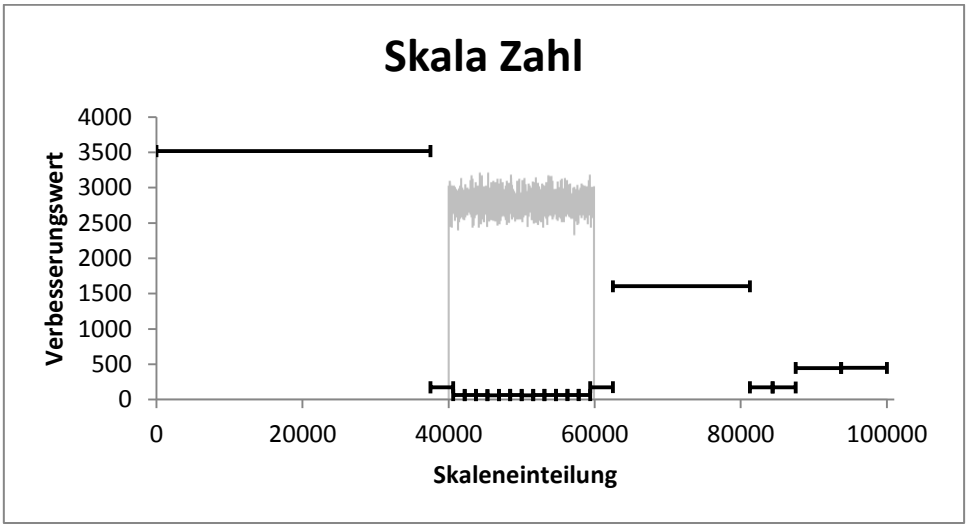

Abb. A-26: Zahl, 100. Zyklus, 20 Intervalle, Säule

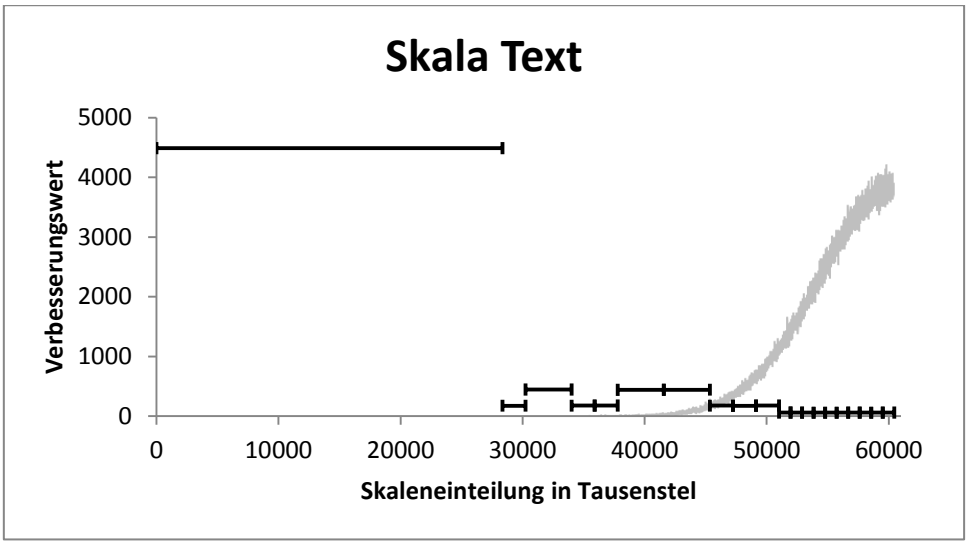

Abb. A-27: Text, 100. Zyklus, 20 Intervalle, Gauss

A.11 Verteilung Konstant in den Anfragen aller Skalen, 25. Zyklus

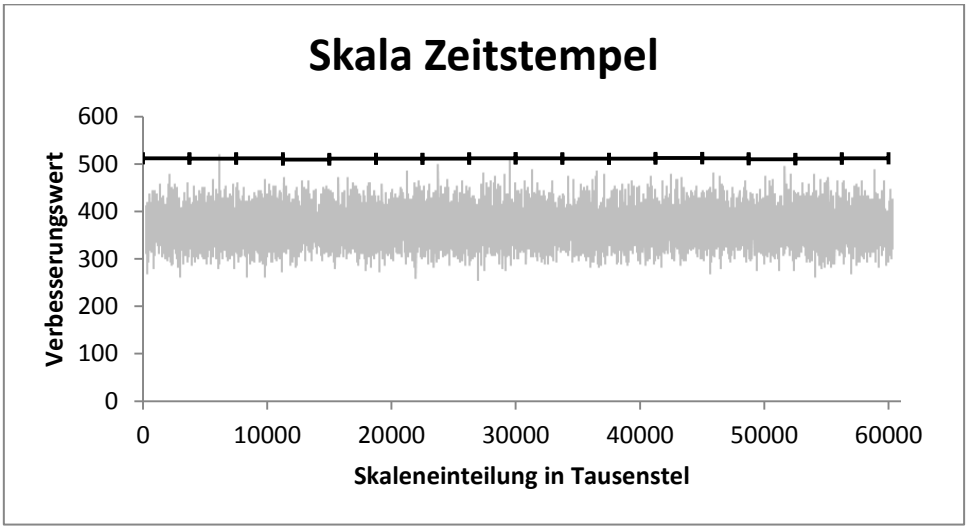

Abb. A-28: Zeitstempel, 25. Zyklus, 16 Intervalle, Konstant

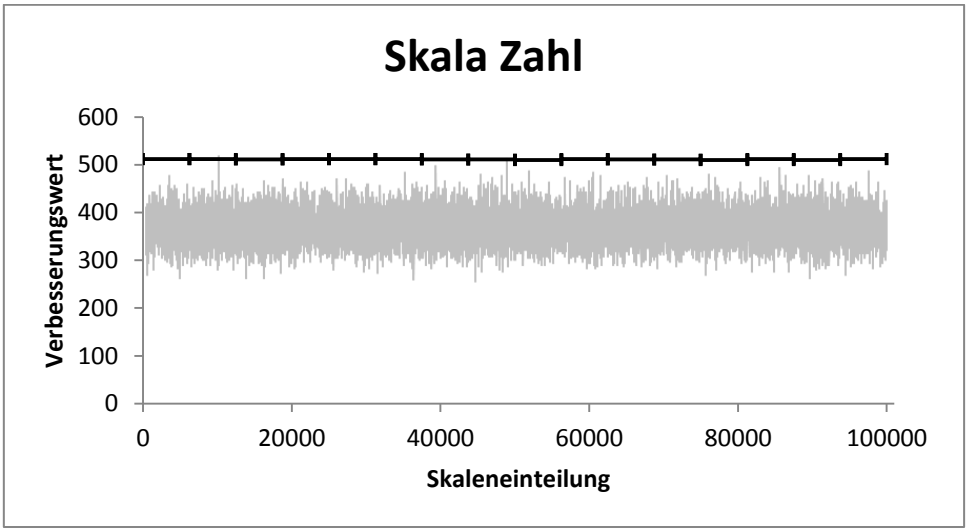

Abb. A-29: Zahl, 25. Zyklus, 16 Intervalle, Konstant

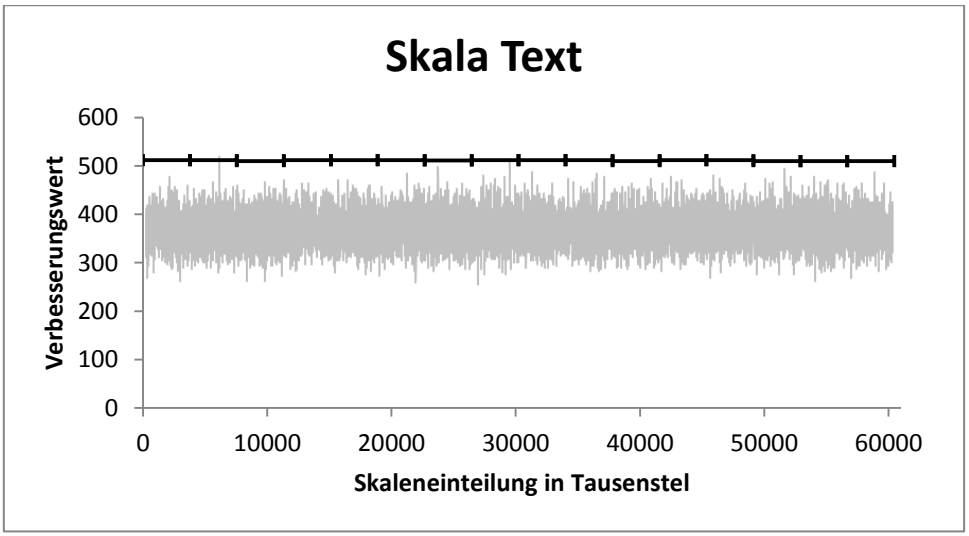

Abb. A-30: Text, 25. Zyklus, 16 Intervalle, Konstant

Selbständigkeitserklärung

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

Magdeburg, den 18. März 2011

Jan Mensing