Alternative Architekturen

Die Architektur eines Datenbankmanagementsystems, wie wir sie bisher besprochen haben, geht von zwei grundlegenden Annahmen aus:

1. Die Daten sind sowohl auf der Platte als auch für das Zugriffssystem im Prinzip als Haufendatei (natürlich mit unterstützenden Indexstrukturen) organisiert, also in einer datensatzorientierten, zeilenweisen Struktur.
2. Die Daten liegen auf der Festplatte und die gesamte Konstruktion des DBMS ist darauf ausgelegt, den I/O zu minimieren. Dies schlägt sich in der Pufferverwaltung und in den Algorithmen für die relationalen Operatoren deutlich nieder.

Wenn man von diesen beiden Annahmen Abstand nimmt, stellt sich die Frage der Architektur eines DBMS neu.

1. Eine Alternative zur zeilenweise Organisation der Daten ist die so genannte spaltenorientierte Organisation. Sie wird mit Erfolg insbesondere in Datenbanken für die Datenanalyse (OLAP – Online Analytic Processing) eingesetzt und hat sich dort in der Geschwindigkeit den traditionellen Architekturen als überlegen erwiesen.
2. Heute ist es durchaus möglich, dass ein Datenbank-Server 1TB Hauptspeicher hat – und viele Datenbanken sind deutlich kleiner. D.h. man kann *alle* Daten im Hauptspeicher halten – dadurch ist keine Pufferverwaltung mehr notwendig und die Frage, welche Algorithmen für die relationalen Operatoren verwendet werden sollten, stellt sich neu.

Aus diesen Überlegungen haben sich in den letzten Jahren neben dem NoSQL-Ansatz Konzepte für alternative Architekturen von Datenbankmanagementsystemen entwickelt, die auch als „NewSQL“ bezeichnet werden. Wir wollen einige dieser Konzepte anreißen.

Kritik an der gängigen Architektur

In Episode 199 des Software Engineering Radio http://www.se-radio.net/2013/12/episode-199-michael-stonebraker/ berichtet Michael Stonebraker http://www.csail.mit.edu/user/1547 von seinen Untersuchungen zur Architektur von Datenbankmanagementsystemen und

von neuen Entwicklungen auf diesem Gebiet. Im Folgenden ein Kurzreferat seiner Auffassungen in meinen Worten.

Der Datenbankmarkt hat drei große Bereiche:

- OLTP (Online Transaction Processing) – hierfür wurden die „klassischen“ DBMS entwickelt mit der Architektur, deren Konzepte Thema der Vorlesung waren. In diesen Bereich gehören sowohl die klassischen Geschäftsanwendungen, aber auch transaktionale Systeme im Internet wie etwa Spieleportale mit vielen Spielern, deren Zustand im Spiel konsistent gespeichert werden muss; oder das Verarbeiten von Daten in Echtzeit, die von Sensoren anfallen (Stichwort „Internet der Dinge“).
- Data Warehouse (OLAP – Online Analytic Processing, heute schicker: Business Intelligence) – in diesem Bereich werden neue Architekturen entwickelt: MonetDB, Hana etc.
- Potpourri von anderen Anwendungsfällen, für die sich NoSQL-Systeme eignen – wie Graphdatenbanken oder Anwendungen von MapReduce etc.

Grundzüge der gängigen Architektur

Die Grundzüge der heute noch gängigen Architektur stammt von System R von IBM (siehe [2]) und wurde von dort in IBM DB2 sowie alle anderen gängigen Datenbankmanagementsysteme übernommen.

- Organisation der Daten auf der Platte als Datensätze in einer Haufendatei, B-Bäume als Indexstrukturen.
- Im Hauptspeicher befindet sich ein Cache dieser Daten mit demselben Aufbau wie die Blöcke auf der Platte.
- In der Regel werden Sperren auf Datensätzen eingesetzt, um die Synchronisation konkurrierender Transaktionen zu erreichen. Teilweise werden auch Mechanismen der Multiversionierung mit Sperrmechanismen gemischt.
- Für die Wiederherstellung wird ein Write-Ahead-Log-Verfahren eingesetzt.
- Der Zugriff erfolgt via SQL, was von einem Anfrageoptimierer aufbereitet und mit einem Zugriffssystem ausgeführt wird, das datensatzorientiert ist.

Untersuchung dieser Architektur

Stonebraker hat untersucht, in welchen Komponenten eines DBMS die meiste Zeit verbraucht wird. Die Untersuchung wurde an einem akademischen DBMS, Shore DBMS, durchgeführt. Shore hat die gängige Architektur und Stonebraker geht davon aus, dass seine Ergebnisse bei kommerziellen DBMS ähnlich aussehen würden.

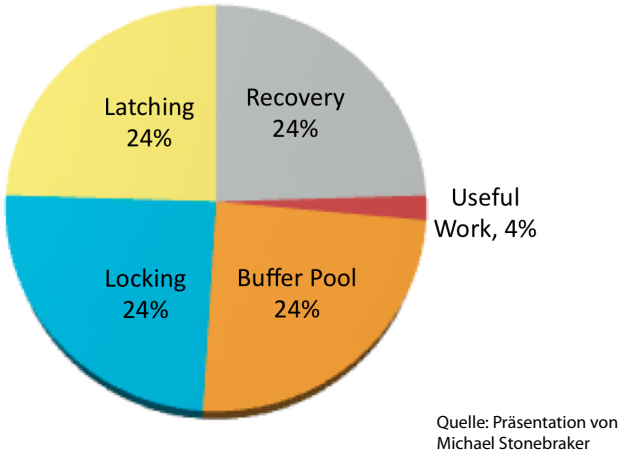


Abbildung 1: Zeitverbrauch im Shore DBMS

Der zeitliche Overhead ergibt sich durch:

Pufferverwaltung

Die Seiten in der Pufferverwaltung haben das Format der Blöcke auf der Festplatte. Für das Zugriffssystem müssen sie aber in das geeignete Format gebracht werden, wie tatsächlich dann auf die Daten zugegriffen wird.

Wenn man sich vorstellt, dass die gesamten Daten sich im Hauptspeicher befinden – was bei heutigen Größen des Hauptspeichers in vielen Fällen problemlos möglich ist –, dann ist diese Struktur der Pufferverwaltung obsolet.

Sperren

Die Sperrverfahren sind sehr teuer, denn bei jedem Zugriff müssen Sperren gesetzt bzw. freigegeben werden.

Write-Ahead-Log

Die Vorkehrungen für das Recovery, die typischerweise durch WAL-Verfahren gemacht werden, kosten etwa ein weiteres Viertel der Overhead-Zeit.

Multithreading – Latching

In den gängigen Architekturen wird Multithreading eingesetzt. Ein wesentlicher Grund für das Multithreading war, dass Plattenzugriffe langsam sind und deshalb während eines Plattenzugriffs ein anderer Thread sinnvolle Arbeit verrichten kann. Dieses Argument ist bei einer Hauptspeicherdatenbank hinfällig.

Die verschiedenen Threads verwenden die gleichen Daten, wie etwa B-Bäume zum Zugriff über Index. Also sind Vorkehrungen für die Synchronisation dieser Threads nötig, so genannte „Latches“. Deren Verwaltung kostet ein weiteres Viertel der Overhead-Zeit.

Fazit

Wenn man eine Hauptspeicherdatenbank hat, hat man zwar die Pufferverwaltung los. Was ist aber mit den anderen Teilen?

In den letzten Jahren gab es viel Forschung und Prototypen, die neue Konzepte entwickelten:

1. Keine Sperrverfahren mehr, stattdessen MVCC oder Timestamp-Ordnung.
2. Kein physisches Logging mehr, sondern logisches Logging. Das bedeutet natürlich, dass Recovery aufwändiger ist – aber der Fall ist viel seltener! Außerdem haben moderne Systeme Replikation für hohe Verfügbarkeit.
3. Was das Multithreading angeht:
   (1) kein Multithreading mehr, stattdessen Ausnutzen der Mehrkerntechnik. Auf einer Maschine mit 4 Kernen und 32 GB Speicher bekommt jeder Kern 8 GB Speicher fest zugeordnet und arbeitet darauf single-threaded (Beispiel VoltDB) – oder
   (2) Verwenden von gemeinsam genutzten Datenstrukturen, die jedoch keine Latches brauchen – in diesem Bereich gibt es aktuelle Forschungen.

Spaltenorientierte Speicherung

MonetDB

MonetDB [5] ist ein spaltenorientiertes Datenbankmanagementsystem, das von der Database Architectures group des Centrum Wiskunde & Informatica (CWI) in Amsterdam entwickelt wurde und wird. Die Architektur wurde in erster Linie für Data Warehouse-Anwendungen konzipiert.

> „MonetDB achieves significant speed up compared to more traditional designs by innovations at all layers of a DBMS, e.g., a storage model based on vertical fragmentation (column-store), a modern CPU-tuned query execution architecture, adaptive indexing, run-time query optimization, and a modular software architecture.“ [5]

Wir wollen uns hier insbesondere das Konzept der spaltenorientierten Organisation der Daten ansehen.

MonetDB fragmentiert Relationen vertikal, indem die Werte eines Attributs, eine Spalte in einer eigenen Tabelle bestehend aus einem Surrogat-Schlüssel sowie dem Wert gespeichert wird. Diese Tabelle wird *Binary Association Table* (BAT) genannt. Die linke Spalte einer BAT, die den Surrogat-Schlüssel enthält wird *head* genannt, die rechte Spalte, die den Attributwert enthält, wird als *tail* bezeichnet. Den Surrogat-Schlüssel nennen die Entwickler von MonetDB auch *object identifier* (OID). Zu einer Relation mit n Attributen gibt es also n BATs, die die jeweiligen Werte als `(OID, value)`-Paare enthalten.

In der Implementierung von gespeicherten BATs müssen die OIDs gar nicht gespeichert werden, sie ergeben sich einfach aus dem Index des jeweiligen Werts, man kann ein solches BAT als ein Array oder einen Vektor sehen. Die Werte eines Tupels der ursprünglichen Relationen werden an denselben Positionen der zugehörigen BATs gespeichert.

Haben die Attribute Werte fester Länge, werden die Werte selbst im BAT gespeichert; bei variabler Länge werden die Werte in einer Art Dictionary gespeichert und das BAT enthält Referenzen auf dieses Dictionary.

Am Beispiel der (bekannten) Relation `Suppliers` lässt sich das Konzept wie in Abb. 2 illustrieren.

Um die BATs in den Hauptspeicher zu laden, verwendet MonetDB speichereingeblendete Dateien (*memory mapped files*) des Betriebssystems,

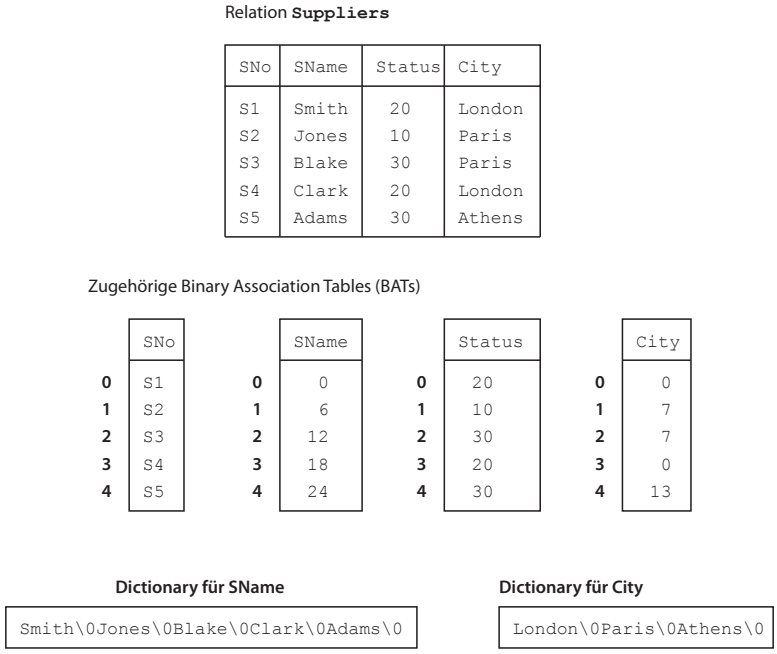


Abbildung 2: Vertikale Fragmentierung einer Relation in MonetDB

d.h. die Datenstrukturen im Speicher sind identisch mit denen auf der Platte.

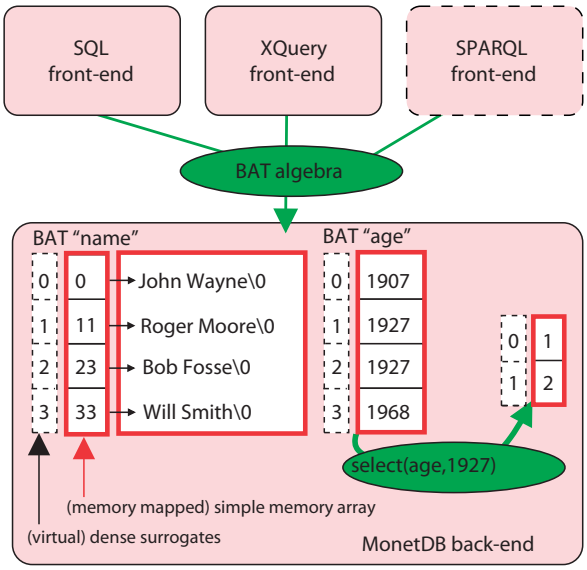
Für die Zugriffe auf die Daten hat MonetDB eine abstrakte Maschine, die eine relationale Algebra binärer Relationen, eben der BATs implementiert. D.h. Anfragen in (z.B.) SQL werden in eine Anfrage der *BAT-Algebra* übersetzt und in dieser Algebra ausgeführt. Dabei sind alle dabei entstehenden Zwischenergebnisse wieder BATs, die Ergebnis-Relation wird erst ganz am Ende aus den BATs rekonstruiert.

Die Implementierung der Operatoren der BAT-Algebra ist hoch optimiert in Bezug auf die heutigen Rechnerarchitekturen – siehe [1]

Die Abb. 3 aus [1] illustriert diese Architektur.

Das TransRelational Model

C.J. Date hat in der 8. Ausgabe seines Klassikers über Datenbanksysteme [4] einen Abschnitt über eine grundlegend andere Implementierung von Relationen. Eine ausführliche Beschreibung findet man in [3]. Er nennt dieses Konzept das „TransRelational Model“, wobei der Begriff „trans“ hierbei nicht von *transzendent* kommt, sondern von *transform*.



Quelle: Peter A. Boncz, Martin L. Kersten, and Stefan Manegold
Breaking the Memory Wall in MonetDB, Comm. ACM 12/2008

Abbildung 3: Architektur MonetDB

Zu diesem Konzept gibt es ein Patent und Date ist 2002 davon ausgegangen, dass demnächst eine kommerzielle Implementierung des Konzepts vorliegt. Dies ist meines Wissens bis heute nicht der Fall. Gleichwohl ist es ein interessanter Ansatz.

Im relationalen Datenmodell besteht eine Relation aus einem Relationsschema und einer Tupelmenge. Das Relationsschema definiert eine Menge von Attributen mit zugehörigen Datentypen, auf die die Werte in den Tupeln der Tupelmenge bezogen werden.

Weder bei den Attributen noch bei der Tupelmenge spielt die Reihenfolge eine Rolle. Fixiert man eine Reihenfolge der Attribute und der Tupel erhält man einen Repräsentanten der Äquivalenzklasse der Relationen. Date nennt dies ein *File* – ein Beispiel wird in Abb. 4 dargestellt.

Zu einer solchen fixierten Relation, einem File, gehören im TransRelational Model zwei Datenstrukturen[1]:

Die *Fields Value Table* enthält die Werte des Files und die *Record Reconstruction Table* enthält die Information, wie aus diesen Werten die

[1] Es handelt sich dabei um das Konzept, d.h. man kann sich durchaus verschiedene konkrete Implementierungsstrategien vorstellen.

File

SNo	SName	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Abbildung 4: File im TR-Modell

Datensätze der Datei rekonstruiert werden können.

Die Field Value Table (FVT) enthält die Werte der Attribute in jeweils sortierter Reihenfolge[2] Es handelt sich also nicht um eine tupelweise (zeilenweise), sondern um eine attributweise (spaltenweise) Strukturierung.

Die Record Reconstruction Table (RRT) hat dieselbe Zeilen- und Spaltenzahl wie die zugehörige FVT, sie enthält Zeilennummern mit deren Hilfe die Tupel zusammengehörender Werte der FVT zusammengebaut, rekonstruiert werden können. Diese Tabelle enthält in der Zelle $[i, j]$ die Zeilennummer des Werts von Attribut $i+1$ in einem Datensatz, der den Wert in Zelle $[i, j]$ der FVT enthält.

Für das obige Beispiel sind die beiden Datenstrukturen in Abb. 5 abgebildet.

Startet man bei der Rekonstruktion bei Zelle $[i, 1]$ erhält man die Folge:

$$[i, 1] \to [i + 1, [i, 1]] \to [i + 2, [i + 1, [i, 1]] \to \ldots$$

Im Ergebnis landet man wieder bei $[i, 1]$ und erhält einen Datensatz, indem man die zugehörigen Werte aus der FVT nimmt. Die RRT ist so konstruiert, dass man bei einer beliebigen Zelle starten kann, um einen Datensatz zu rekonstruieren, der den entsprechenden Wert an diesem Attribut hat.

Dadurch entsteht ein „Zickzack“, weshalb der Algorithmus zur Rekonstruktion auch Zickzack-Algorithmus genannt wird. Unser Beispiel demonstriert das Vorgehen in Abb. 6.

[2] Konzeptionell stellen wir uns *eine* Tabelle vor, es kann aber duchaus sein, dass die Werte jedes einzelnen Attributs in einer konkreten Implementierung gesondert gespeichert werden.

Field Values Table

	1	2	3	4
	SNo	SName	Status	City
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

Record Reconstruction Table

	1	2	3	4
	SNo	SName	Status	City
1	5	5	4	5
2	4	4	2	1
3	2	3	3	4
4	3	1	5	2
5	1	2	1	3

Abbildung 5: Fields Value Table und Record Reconstruction Table

Abbildung 6: Rekonstruktion der Datensätze (Zickzack-Algorithmus)

Die Grundidee des TransRelational Models besteht darin, dass die Werte zu einem Attribut in den internen Datenstrukturen (sowohl im Haupt-

speicher als im sekundären Speicher) sortiert abgelegt sind. Dadurch sind offensichtlich alle Operationen, bei denen auf einen bestimmten Wert oder einen fortfolgenden Wertebereich zugegriffen werden muss, zu *jedem* Attribut durch binäre Suche zu realisieren. Ebenso kann man Join-Algorithmen verwenden, die auf der Sortierung von Attributwerten basieren, ohne dass erst sortiert werden muss.

Andererseits ist offensichtlich, dass ändernde Aktionen mit den Daten im TR-Modell aufwändiger sind. Date diskutiert in [3] im Detail, wie solche Aktionen implementiert werden können.

Außerdem diskutiert Date Verfeinerungen des TR-Modells, z.B. dadurch, dass man in der FVT Duplikate nicht mehrfach speichert, sowie verschiedene Konzepte der Implementierung des Modells.

M.E. ist die Analyse und Evaluierung (und Weiterentwicklung) dieses Modells sowohl für Hauptspeicherdatenbanken als auch für die Speicherung auf sekundärem Speicher ein interessantes Forschungsthema.

Hauptspeicherdatenbanken

VoltDB

VoltDB ist eine spaltenorientierte Hauptspeicherdatenbank, die von Michael Stonebraker, Sam Madden und Daniel Abadi entwickelt wurde.

Folgende Punkte aus einem Whitepaper auf `http://voltdb.com/` charakterisieren die Architektur von VoltDB:

- VoltDB nutzt Mehrkern-Prozessoren so aus, dass auf jedem Kern eine Zugriffsmaschine in einem einzelnen Thread auf Daten zugreift, die ihr exklusiv zur Verfügung stehen. Man nennt dies eine *Shared-Nothing-Architektur*, eine Zugriffsmaschine auf einem Kern wird bei VoltDB *virtual node* genannt.
- Die Daten sind partitioniert über diese virtuellen Knoten, manche, kleinere Tabellen werden auch auf jeden Knoten repliziert. Alle Daten sind im Hauptspeicher.
- Alle Transaktionen in VoltDB werden als Stored Procedures definiert, d.h. alle Transaktionen sind kurz (ohne Benutzerinteraktion) und können deshalb seriell abgewickelt werden.
- Das Datenbankdesign und die Fragmentierung sollten von den Anwendungsentwicklern so konzipiert werden, dass Zugriffe möglichst auf einem Knoten stattfinden. Ist dies nicht der Fall, findet ein verteilter Zugriff statt.

- VoltDB ist hochverfügbar durch automatische Replikation. Dabei wird synchrone Replikation (ROWA) eingesetzt innerhalb eines Clusters, asynchrone Replikation zwischen geografisch getrennten Clustern.
- VoltDB arbeitet in einer so genannten *active-active*-Konfiguration, d.h. es findet ein automatisches Umschalten auf einen Replika-Knoten statt, wenn ein Problem mit einem Knoten besteht.
- Um die Dauerhaftigkeit abgeschlossener Transaktionen zu garantieren, verwendet VoltDB logisches Logging (*command logging*).

Das genannte Whitepaper enthält die Abb. 7 mit einem Überblick über die Architektur von VoltDB.

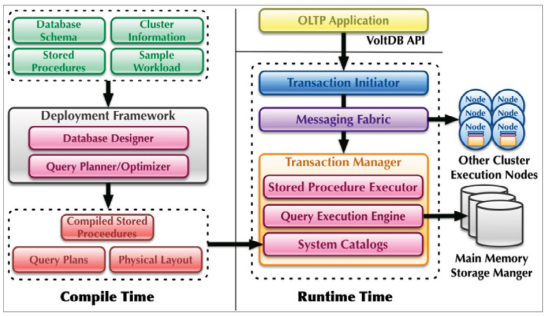


Abbildung 7: Überblick Architektur von VoltDB

HANA

SAP HANA ist eine spaltenorientierte Hauptspeicherdatenbank, deren Prototypen unter dem Namen SanssouciDB am Hasso-Plattner-Institut Potsdam entwickelt wurden.

In [6] stellt Hasso Plattner die grundlegenden Eigenschaften von SanssouciDB dar:

- Alle Daten liegen im Hauptspeicher und alle Operatoren werden im Hauptspeicher ausgeführt.
- Die Strukturierung der Daten ist spaltenorientiert. Dadurch kann die Zahl der Indexe reduziert werden.

- Das DBMS unterscheidet zwischen *aktiven* und *passiven* Daten. Aktiv sind Daten, die in aktuellen, noch nicht abgeschlossenen Geschäftsprozessen benötigt werden, passive Daten sind Daten abgeschlossener Geschäftsprozesse, sie werden nicht mehr verändert. Passive Daten können eventuell auf langsameren Speicher ausgelagert werden.

Abb. 8 aus [6, S.32] gibt einen Überblick über die Architektur von SanssouciDB.

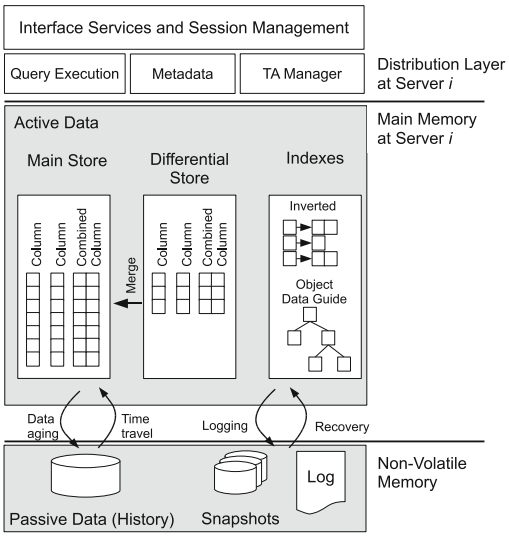


Abbildung 8: Überblick Architektur von HANA

Literaturverzeichnis

[1] Peter A. Boncz, Martin L. Kersten, Stefan Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12), Dezember 2008.

[2] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al. A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646, Oktober 1981.

[3] C. J. Date. *Go Faster! The TransRelational Approach to DBMS Implementation.*

[4] C. J. Date. *An Introduction to Database Systems*. Reading, MA, 8. Auflage, 2004.

[5] S. Idreos, F. E. Groffen, N. J. Nes, et al. MonetDB: Two Decades Of Research In Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40 – 45, March 2012.

[6] Hasso Plattner. *A Course in In-Memory Data Management*. Heidelberg, 2013.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 1.4 – 20. Mai 2014