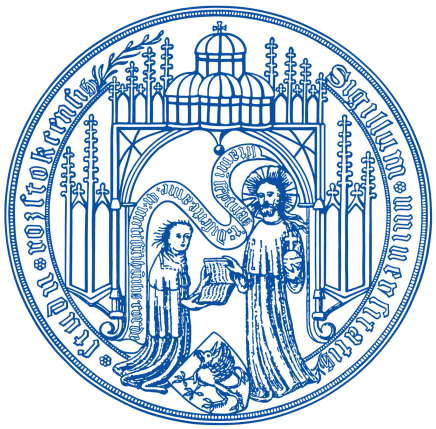
Systematische Untersuchung der Anfragekapazität verschiedener DBMS

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Maik Schmude
Matrikelnummer: 210207260
geboren am: 25.10.1989 in Rostock
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Dr.-Ing. Holger Meyer
Betreuer: M. Sc. Hannes Grunert

Abgabedatum: 13. September 2016

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 12. September 2016

Inhaltsverzeichnis

Abstract

Smarte Umgebungen besitzen meist eine vertikale Architektur, bei dem die Daten auf der untersten Ebene, den Sensoren erzeugt werden. Anschließend werden die Daten weiter nach oben in der Hierarchie gereicht, bis sie zumeist eine Cloud erreichen. Dort werden die Daten analysiert und ausgewertet um z.B. die Intentionen eines Anwenders zu erkennen. Bei diesem Vorgang werden sensible Informationen über den Nutzer gesammelt. Der Lehrstuhl Datenbank- und Informationssysteme der Universität Rostock hat sich diesem Problem zugewendet und hat mit dem Privacy-by-Design-Prinzip das Projekt PArADISE [1] gestartet. In diesem Projekt sollen die Anfragen, die an den eingegangen Daten der Nutzer gestellt werden, soweit wie möglich nach unten in der Hierarchie verteilt werden. Um dies zu erreichen muss die ursprüngliche Anfrage gesplittet werden, um anschließend an die unterschiedlichen Systemebnen der Smarten Umgebungen, verteilt zu werden. So das jede Ebene einen Teil der Anfrage bearbeiten kann. Dazu müssen zunächst jedoch, die Anfragekapazitäten der unterschiedlichen Datenbankmanagementsysteme der Ebenen der Hierarchie untersucht werden. In Rahmen dieser Arbeit werden sieben unterschiedliche Datenbankmanagementsysteme hinsichtlich ihrer Möglichkeit Daten zu erfragen untersucht.

[1] Privacy AwaRe Assistive Distributed Information System Environment

Kapitel 1

Einleitung

Ob in modernen Smart Home Systemen oder der Industrie 4.0, immer häufiger begegnen wir in unserem modernen Alltag Assistenzsystemen, die unser Verhalten messen und beurteilen. In vielen Bereichen unseres Lebens sind wir daher von immer mehr Sensoren umgeben, die uns dabei helfen sollen, unser Leben zu vereinfachen. Dazu beobachten und analysieren sie unser Verhalten und leiten ihre Erkenntnisse an andere Systemen weiter. Dabei fallen große Mengen Daten an, die teilweise sehr sensible Informationen über uns enthalten.

Der Lehrstuhl Datenbank- und Informationssysteme der Universität Rostock verfolgt hierbei das so genannte Privacy-by-Design-Prinzip. Bei diesen Ansatz soll schon bei der Entwicklung des Systems darauf geachtet werden, dass sensible Daten so früh wie möglich verschlüsselt oder anonymisiert werden um vor eventuellen Missbrauch zu schützen und das Recht auf Privatsphäre vor neugierigen Konzernen zu gewährleisten. Um dies zu erreichen sollen nur so wenig Daten wie möglich an Cloud Provider von Smart Home Systemen weitergeleitet werden, so dass ein großer Teil der Daten im Umfeld des Nutzers, in diesem Fall seiner eigenen vier Wände, bleibt. Für solch einen Ansatz ist es essenziell Anfragen an die Sensordatenbestände vertikal, also an alle Ebenen der vorliegende Systemstruktur, zu verteilen. Dazu müssen die Anfragen gesplittet werden. Es soll hierbei versucht werden möglichst große Teile der Anfrage an die jeweils weiter unten in der Hierarchie liegende Ebene zu geben und die Ergebnisse in Form von Daten nur sehr sparsam wieder in die Rückrichtung (nach oben) zu geben. Durch die Verteilung von Anfragen an die einzelnen Stufen des Assistenzsystems, siehe Abbildung 1.1, sollen sowohl die Privatheitsaspekte als auch die Leistung verbessert werden. Dabei sollen komplexe Anfragen möglichst nah an die einzelnen Sensoren des Systems geschoben werden. Denn weiter unten in der Hierarchie sind die Ressourcen, wie z.B. Arbeitsspeicher oder Rechenleistung, die für die Verarbeitung der Anfrage zur Verfügung stehen, begrenzter. Um hier effizient arbeiten zu können, muss die Anfragekapazität verschiedener Datenbankmanagementsysteme (ab hier nur noch als DBMS bezeichnet) unterschiedlicher Leistungsklassen untersucht werden.

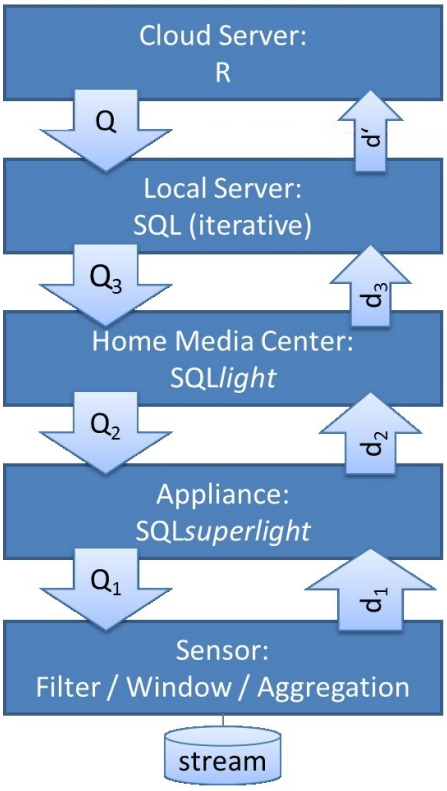


Abbildung 1.1: Stufen der Anfrageumschreibung [GH16]

Zu diesem Thema führe ich in dieser Bachelorarbeit eine ver-

gleichende Analyse ausgewählter DBMS durch. Dabei sind von eingebetteten DBMS wie BerkeleyDB, über Opensource-DBMS wie MySQL und PostgreSQL bis hin zu den großen DBMS wie DB2 und Oracle Database. Neben der Anfragekapazität, liegt ein besonderer Fokus auf den Statistik-Funktionen, die für die Analyse der Daten wichtig sind.
Die einzelnen DBMS wurden dabei in drei Kategorien eingeteilt: kleine Systeme Kapitel 2, mittlere Systeme Kapitel 3 und große DBMS Kapitel 4. Nachdem alle Systeme bearbeitet wurden, folgt eine Übersicht im Kapitel 5 über die gewonnen Erkenntnisse. Anschließend folgt eine Prototypische Implementierung, einer Erweiterung für eine bestehende Metadatenklasse eines JDBC-Treibers, im Kapitel 6. Letztlich wird im Kapitel 7 eine Zusammenfassung mit einen Ausblick dieser Arbeit gezeigt. Doch zunächst werfen wir ein Blick in den SQL Standard, um einen Überblick über mögliche Anfragetypen zu erhalten.

1.1 SQL Standard

SQL die Structured Query Language ist eine Datenbanksprache, die zum Verwalten von relationalen Datenbanken genutzt wird. Sie wurde von einem gemeinsames Gremium von ISO und IEC standardisiert. Ein Vorteil von relationalen Datenbanken ist, da sie auf Relationsalgebren und der Mengenlehre basieren, dass sie sehr anschaulich als Tabellen dargestellt werden können. So sind sie auch leicht für Anwender zu verstehen sind. Weitere Vorteile sind eine größere Auswahl von Anfragemechanismen, in Vergleich zu nicht SQL nutzenden Systemen und die leichtere Handhabung von Analysewerkzeugen. Diese müssen die Daten nicht erst in ein Arbeitsverzeichnis kopieren sondern können direkt auf der Datenbank angewendet werden. In dieser Arbeit beziehe ich mich auf den SQL-Standard von 2011 [Mel11].

Die allgemeine Anfrage in SQL hat folgende Struktur1.2:
Die drei wichtigsten Befehle sind SELECT, FROM und WHERE. Wie zuvor bereits erwähnt, werden

```
SELECT [DISTINCT|ALL] Auswahlliste [AS Spaltenalias]
FROM Quelle [ [AS] Tabellenalias]
 [WHERE Where-Klausel]
 [GROUP BY (Group-by-Attribut)+]
 [HAVING Having-Klausel]
 [ORDER BY (Sortierungsattribut [ASC|DESC])+];
```

Abbildung 1.2: Struktur einer allgemeinen Anfrage in SQL [Wika]

Relationale Datenbanken als Tabellen repräsentiert. Der SELECT-Befehl dient dazu auszuwählen welche Attribute(Spalten) genutzt werden sollen. Mit SELECT * wird die ganze Relation (Tabelle) ausgewählt. Das Ergebnis jeder SELECT-Anweisung ist eine Menge von Zeilen. Mit FROM wird angegeben welche Relationen bzw. Sichten genutzt werden sollen. Optional kann hier mit AS gefolgt von einer Zeichenkette eine Umbenennung der Spalten vorgenommen werden. Die WHERE-Klausel wird genutzt um Bedingungen zu formulieren. Diese filtern die auszugebenden Daten. Mit GROUP BY können einzelne Zeilen zu Gruppen zusammengefasst werden. Die Anweisung ORDER BY wird genutzt um die Spalten der Ergebniszeilen zu sortieren. Nach dem Befehl ORDER BY werden ein Attribut oder mehrere Attribute angegeben, nach denen sortiert werden soll. Anschließend wird angegeben in welcher Reihenfolge sortiert werden soll. Die Standardvoreinstellung ist eine aufsteigende Sortierung (ASC), das Gegenstück wäre DESC eine absteigende Sortierung, also mit dem größten Wert zuerst. Falls nach mehr als einem Attribut sortiert werden soll, wird zuerst die Spalte des zuerst angegebenen Attributes sortiert, falls dann doppelte Werte vorhanden sind, wird nach dem nachfolgenden Attribut geordnet.
Ein weiterer wichtiger Punkt sind die sogenannten Aggregatfunktionen. Sie führen eine Operation auf einer Wertemenge aus und geben anschließend einen einzelnen Wert zurück. Wenn zusätzlich ein GROUP BY gesetzt wird, wird für jede Gruppe innerhalb der Wertemenge ein Wert zurückgeliefert. Alle Aggregatfunktionen außer COUNT ignorieren NULL-Werte. NULL-Werte stehen für ein leeres Attributfeld, die nicht zu verwechseln sind mit dem Wert-Eintrag Null. Aggregatfunktionen werden in HAVING-Klauseln und SELECT-Anweisungen eingesetzt.
Die Funktionen (siehe Abbildung 1.3) möchte ich hier als die einfachen Aggregatfunktionen bezeichnen. Da sie wesentlich häufiger in DBMS implementiert wurden als die komplexeren Aggregatfunktionen (siehe Abbildung 1.2).
Beachtet werden muss, dass Argumente wie DISTINCT und ALL bei Varianz und Standardabweichung berechnenden Funktionen nicht gestattet sind. Alle redundanten Duplikate werden bei der Ausführung dieser Funktionen nicht entfernt.
Eine weitere Funktion die im SQL Standard beschrieben wird, ist die Fensterfunktion. Diese dient der Teilung einer Liste von Zeilen des Anfrageergebnisses, dem Resultset, in definierte Abschnitte in denen dann

Befehl	Bedeutung
AVG	Berechnet den arithmetischen Mittelwert der Werte einer Spalte
COUNT	Gibt die Anzahl aller Elemente einer Gruppe zurück
MIN	Berechnet das Minimum einer Spalte
MAX	Berechnet das Maximum einer Spalte
SUM	Berechnet die Summe aller Werte

Tabelle 1.1: Ausschnitt der "einfachen" Aggregatfunktionen in SQL 2011 [Mel11]

Befehl	Bedeutung
REGR_SLOPE	Bestimmt den Anstieg einer linearen Gleichung
REGR_INTERCEPT	Bestimmt den y-Achsenabschnitt der linearen Gleichung
REGR_R2	Berechnet das Bestimmtheitsmaßes
CORR	Berechnet den Korrelation Koeffizient
COVAR_POP	Bestimmt die Kovarianz der gegebenen Ausdrücke
STDDEV_POP	Berechnet die Standardabweichung der gegeben Gruppe
VAR_POP	Gibt die statistische Varianz aller Werte im angegebenen Ausdruck zurück

Tabelle 1.2: Ausschnitt der "komplexen" Aggregatfunktionen in SQL 2011 [Mel11]

z.B. Aggregatfunktionen ausgeführt werden können. Um die Fensterfunktion zu nutzen wird die OVER-Klausel verwendet; in Kombination mit einer Aggregatfunktionen wie SUM und der Klausel PARTITION BY könnte z.B. die Summe eines Datensets in Gruppen zerlegt werden. Das folgenden Beispiel zeigt eine mögliche Anwendung der Fensterfunktion 1.3.

```
SELECT Läufer, Laufzeit
SUM Laufzeit OVER (PARTITION BY Läufer ORDER BY Startzeit) AS Ges_Laufzeit
FROM Wettbewerb
```

Abbildung 1.3: Beispiel für die Fensterfunktion

Sind die angeforderten Informationen über mehrere Tabellen verteilt, so können Verbundoperationen (engl. joins) genutzt werden. Die Verbundoperation verbindet zwei Relationen über ein kartesisches Produkt miteinander. Dabei werden jedoch nur solche Tupel für die Ergebnismenge ausgewählt, die in einer gewissen Beziehung zueinander stehen, indem sie ein vorgegebenes Selektionsprädikat erfüllen[Lex12]. Es gibt vier verschiedene Typen von Verbundoperationen, dass sind der Theta-Join, der Equi-Join, der Natural-Join und der Outer-Join. In dieser Arbeit betrachten wir jedoch nur den Equi-Join und den Natural-Join, da alle nötigen Verbunde mit ihnen abgebildet werden können. Beim Equi-Join ist als Selektionsprädikat nur der Vergleichsoperator = zulässig. Eine Spezialform des Equi-Joins ist der Natural-Join und wird auch in SQL als Inner Join bezeichnet. Dieser verknüpft Tabellen über gleich benannte Spalten, indem er jeweils zwei Tupel verschmilzt, falls sie dort gleiche Werte aufweisen. Wird dabei für ein Tupel kein Gegenstück gefunden, so werden diese sogenannten dangling tuples rausgeschmissen.

Wenn in einer WHERE-Klausel eine weitere SELECT-Anweisung steht, so wird die Anfrage als verschachtelt bezeichnet. Damit lassen sich komplexere Anfragen stellen, bei denen auf Informationen einer Anfrage wieder eine Anfrage gestellt werden kann. Ein Nachteil von geschachtelten Anfragen ist, dass die Abfragetiefe fest vorgegeben werden muss. Abhilfe bei diesem Problem schaffen rekursive Abfragen. Diese werden mit Hilfe der WITH-Klausel eingeleitet und sehen beispielsweise wie in folgender Abbildung 1.4 aus:

```
WITH RECURSIVE Strecke (Startort, Zielort) AS (
        SELECT Startort, Zielort
        FROM Fahrplan
        WHERE Startort='Berlin'
                        UNION ALL
        SELECT A.Startort, B.Zielort
        FROM Strecke A, Fahrplan B
        WHERE A.Startort = B.Zielort)
SELECT DISTINCT * FROM STRECKE
```

Abbildung 1.4: Rekursive Abfrage einer Stecke von Berlin [Sch10]

Diese Anfrage wird im ersten Durchlauf alle Orte, die eine direkt Verbindung mit Berlin haben zurück geben, dann alle Orte die über einen Zwischenstopp mit Berlin verbunden sind und so weiter bis alle Möglichkeiten ausgeschöpft sind. Mit dem Befehl UNION können mehrere Abfragen miteinander verbunden werden.

Bei den großen Systemen, die OLAP nutzen sollen, sind weitere SQL Konstrukte nötig. Ab den SQL Standard von 1999 wurde angefangen OLAP-Operationen mit in den Standard aufzuführen, dazu gehören auch CUBE, ROLLUP und GROUPING SETS, diese sind Erweiterungen von der GROUP BY-Klausel. Der CUBE-Operator generiert von angegebenen Attributen alle möglichen Untermengen. Diese Anweisung wird genutzt um bei Aggregationen, wie z.B. der Summenbildung, alle möglichen Gruppierungskombinationen aus der angegebenen Menge der Gruppierungsattribute zu bilden. Die gleiche Funktionalität könnte auch mit mehreren GROUP BY-Anweisungen erreicht werden, doch bei bereits drei angegebenen Gruppierungsattribute wären acht einzelne SQL-Anfragen, die mit UNION ALL kombiniert wurden nötig. Damit erleichtert die CUBE-Anweisung dem Nutzer, bei der Nutzung von Aggregationen auf OLAP Strukturen, viel Schreibarbeit.
Eine weitere GROUP BY Erweiterung ist ROLLUP. ROLLUP wird genutzt um Daten zusammenzufassen. Das bedeutet, dass der ROLLUP-Operator ein Resultset generiert, das Summenzeilen oder Zwischensummenzeilen in die Ergebnismenge einer Anfrage, die eine GROUP BY-Klausel enthält, einfügt. Dieses Resultset enthält z.B. Aggregate für eine Hierarchie von Werten, die aus den ausgewählten Spalten (Dimensionsattributen) entstanden sind.
Die letzte GROUP BY Erweiterung, die wir hier betrachten ist GROUPING SETS. GROUPING SETS funktioniert ähnlich wie CUBE, mit dem Unterschied dass anstatt sämtliche Untermengen zu bilden, nur ein zuvor definierter Teil der Attributmengen aggregiert wird [Kle12].

CUBE Anweisung	**Äquivalente GROUPING SETS Anweisung**
GROUP BY CUBE(a,b,c)	GROUP BY GROUPING SETS ((a,b,c) (a,b) (a,c) (b,c) (a) (b) (c) ())

Tabelle 1.3: Aus den drei Gruppierungsattribute von CUBE wurden acht GROUPING SETS [Kle12]

Weitere Informationen über OLAP und Dimensionen sind im nachfolgenden Abschnitt 1.2 zu finden.

1.2 Weiterführende Analysemethoden

In der obersten Ebene der Anfrageverteilung haben wir große Mengen von Daten, die aus unterschiedlichen Quellen zusammen getragen wurden. Diese müssen in ein System integriert werden um sie anschließend analysieren zu können. Ein solches System wird als Data Warehouse bezeichnet [CU97]. Aufgaben eines Date Warehouses ist es daher die Daten zu extrahieren, anschließend die Daten zu transformieren um Fehler zu beheben und die Daten vereinheitlichen. Letztlich müssen dann Analysewerkzeuge geladen und angewendet werden. Eine Methode wie dabei vorgegangen werden kann nennt sich OLAP. OLAP steht für On-Line Analytical Processing und wird bei der Analyse großer Datenmengen eingesetzt um Entscheidungsprozessen zu unterstützen. Hierbei werden multidimensionale Datenmodelle genutzt. Als Beispiel für eine Visualisierung eines multidimensionales Datenmodell wird häufig ein Datenwürfel 1.5 verwendet.

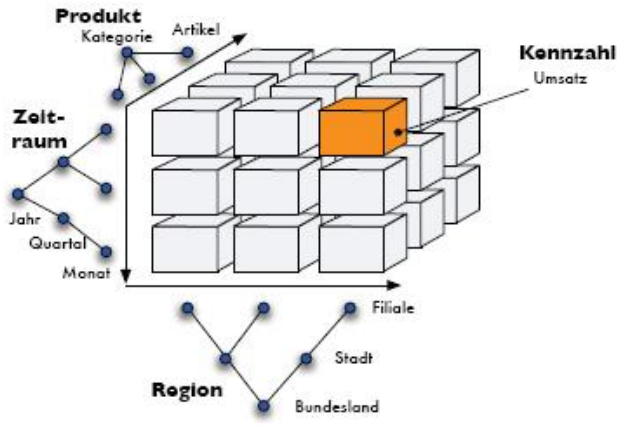


Abbildung 1.5: Modell eines Datenwürfels [SSH12]

Ein Datenwürfel besteht aus mindestens drei Dimension. Dimensionen können unterschiedliche Größen und Aufteilungen haben, oft werden sie wir hier im Bild zu sehen hierarchisch gegliedert. Der abgebildete Datenwürfel hat drei Dimensionen: Produkt, Zeitraum und Region. Jede Dimension besteht aus einer Menge aus Kategorienattributen, so besteht beispielsweise die Dimension Produkt, aus den Kategorienattributen Kategorie und Artikel. Das feinste Kategorienattribut wird als Primärattribut bezeichnet, in diesem Fall ist Artikel ein Primärattribut. Die Daten im Würfel werden als Kennzahl bezeichnet, sie sind nach Dimensionen sortiert

Kapitel 2

Kleine Systeme

Kleine Systeme sollen in der untersten Ebene, der Sensorebene, eingesetzt werden. Ihre Hauptaufgabe ist es die von den Sensoren produzierten Daten zu filtern und damit den Rechenaufwand für die in den höheren Ebene stattfinden Analysen und Operationen zu reduzieren. Daher wird bei den kleinen Systemen der Fokus auf die Projektion und die Selektion der Daten gelegt. Bei der Selektion sollen die Standardvergleichsoperatoren für die Äquivalenz und für die Ungleichheit vorhanden sein; weiterhin die Möglichkeit für die Vergleiche: kleiner, größer, größer gleich und kleiner gleich. Um komplexere Selektionsanfragen zu stellen werden logische Operatoren wie UND, ODER oder NICHT benötigt.
Da die Sensoren nur geringe Ressourcen für die Berechnung von Anfragen zur Verfügung stellen können, werden in dieser Ebene nur Systeme eingesetzt die selbst wenig Speicherplatz erfordern. Wir haben uns auf zwei Systeme beschränkt: TinyDB und Berkeley DB. Bei TinyDB ist zu beachten, dass es zwei DBMS gibt die diesen Namen benutzen, einmal vom Massachusetts Institute of Technology [HMFH04] und einmal von Markus Siemens[Sie16]. In dieser Arbeit beziehe ich mich auf das System von Markus Siemens.

2.1 TinyDB

TinyDB ist, wie der Name eventuell schon verrät, mit nur 1200 Zeilen Code, dass kleinste der hier vorgestellten Datenbankmanagementsysteme. Seit 20.7.2013 [Sie16] ist es im Internet frei verfügbar, es wurde in Python geschrieben und legt seinen Fokus auf die Verarbeitung von Dokumenten.
Zum Umwandeln der zu speichernden Daten nutzt TinyDB standardmäßig ein Modul namens Python JSON. Dieses Modul ist in der Lage einfache Datentypen zu einen für das DBMS schreibbaren Typen zu wandeln. Bei komplexeren Datentypen, wie z.B. selbsterstellte Klassen, stößt es aber schnell an seine Grenzen. Möchte der Nutzer dennoch komplexere Datentypen nutzen, ist er selbst gefragt seine eigene Speicherverwaltung zu schreiben und diese anschließend einzubinden.

Generell erlaubt uns TinyDB zwei verschiedene Möglichkeiten die Anfragen zu erstellen, zunächst einmal mit ORM (object-rational-mapping). Dies ist eine Technik aus der Softwareentwicklung, mit der ein Programm objektorientierte Konzepte an einer relationalen Datenbank nutzen kann. Dem Programm erscheint die Datenbank als objektorientierte Datenbank, was die Programmierung erleichtert [Wikc]. Soll diese Methode genutzt werden, muss zunächst ein neues Query-Objekt erstellt werden. Anschließend kann dieses genutzt werden um die Rückgabe zu spezifizieren. Wie hier im Beispiel zu sehen 2.1:

```
FROM tindb import Query
Teilnehmer = Query()
db.search(Teilnehmer.name=='Bob')
```

Abbildung 2.1: Anfrage mit ORM

Eine zweite Möglichkeit Anfragen, mit TinyDB zu konstruieren, ist die sogenannte klassische Methode, wie an folgender Selektion gezeigt:

```
FROM tindb import where
db.search(where('name')=='Bob')
```

Abbildung 2.2: Klassische Anfrage

Eine Projektion ist hier leider nicht möglich, es werden immer alle Attribute des Tupels mit ausgegeben. Anstatt des Operators == für die Äquivalenz, sind die Operatoren kleiner <, größer >, größer gleich >=, kleiner gleich <= oder der Operator != für die Ungleichheit möglich. Alternativ kann auch ein Negations-Operator vor die Anfrage setzen, der den gleichen Zweck erfüllt wie ein Ungleichheits-Operator. Auch sind Anfragen auf verschachtelte Strukturen leicht zu stellen, wie in der Abbildung 2.3 zu sehen ist.

```
FROM tindb import Query
Teilnehmer = Query()
db.search(Teilnehmer.name.vorname=='Bob')
```

Abbildung 2.3: Verschachtelte Anfrage

In TinyDB gibt es nicht nur den Negationsoperator ~, sondern auch noch die logischen Operatoren UND & und ODER | mit denen Anfragen kombiniert werden können.

```
FROM tindb import Query
Teilnehmer = Query()
db.search((Teilnehmer.name.=='Bob') & (Teilnehmer.alter>=18))
db.search((Teilnehmer.name.=='Bob') | (Teilnehmer.name==Alice))
```

Abbildung 2.4: Kombination von Anfragen

Zusätzlich gibt es noch eine ganze Liste von Befehlen die genutzt werden können (siehe Tabelle 2.1). Dabei fällt jedoch auf, dass wichtige Aggregatfunktionen, wie z.B. das Minimum oder Maximum eines Attributes, fehlen. Zusätzlich bietet TinyDB die Möglichkeit eigene Test-Funktionen schreiben.

Befehl	**Bedeutung**
Query().field.exists(c)	Prüft jedes Element, ob ein Attribut namens *field* existiert
Query().field.matches(regex)	Prüft, welche Elemente zum regulären Ausdruck *regex* passen
Query().field.search(regex)	Prüft jedes Element die mit einer Teilzeichenfolge zum regulären Ausdruck *regex* passen
Query().field.test(func,*args)	Prüft jedes Element, für die die angegebene Funktion *func* den Wert True zurückgibt
Query().field.all(query|list)	Falls eine Anfrage *query* hier angegeben wird, dann wird geprüft, ob alle Elemente in der Liste*field*, zu allen Elementen der Anfrage passen. Falls eine Liste angegeben wird, werden alle Elemente der Liste *field* überprüft, ob alle von ihnen Element der gegebenen Liste *list* sind.
Query().field.any(query|list)	Falls eine Anfrage *query* hier angegeben wird, dann wird geprüft ob mindestens ein Element in der Liste *field* existiert, dass zur Anfrage passt. Falls eine Liste angegeben wird, werden alle Elemente der Liste *field* überprüft, ob mindestens eins von ihnen Element der gegeben Liste *list* ist.

Tabelle 2.1: Weitere Anfragemöglichkeiten

Letztlich kann durch den Einsatz von Erweiterung der Funktionsumfang von TinyDB zu vergrößert werden. Wichtige Vertreter sind dabei:

- **Tinyindex** ermöglicht es ein Dokument mit einen Index zu versehen und sichert die Determiniertheit der Datenbank.
- **Tinymongo** ist eine Schnittstelle, die es Nutzern ermöglicht TinyDB als Flat-File-System zu nutzen.
- **Tinyrecord** ist eine Bibliothek, die TinyDB um die Umsetzung der Atomaritätseigenschaft von Transaktionen bei nicht SQL nutzenden Datenbanken erweitert. Es nutzt dabei ein Protokoll, dass zunächst die Transaktion aufzeichnet und sie erst anschließend ausführt. Dadurch sollen Sperrzeiten, sogenannte locks, reduziert werden.
- **Tinydb-serialization** Diese Erweiterung hilft TinyDB Objekte zu serialisieren mit welchen TinyDB sonst nichts anfangen könnte.
- **Tinydb-smartcache** stattet TinyDB mit einen intelligenten Cache aus. Dieser ist nützlich, wenn sehr viele Anfragen auf vornehment statischen Datenbanken stattfinden.

Tinyindex und Tinymongo sind noch in einer Testphase und sollten derzeit nur mit Bedacht eingesetzt werden.
TinyDB beherrscht weder Index-Strukturen noch Beziehungen (z.B. Joins) zwischen Tabellen. Auch wird der Zugriff von mehreren Prozessen oder mehreren Threads nicht unterstützt. Es ist daher für simplere Anfragen auf überschaubar großen Datensätzen gedacht.

2.2 Berkeley DB

Ein zweites kleines System welches, ich vorstellen möchte, ist Berkeley DB. Berkeley DB wurde an der University of California Berkeley entwickelt und dann im Jahre 1992 als Berkeley DB 1.85 veröffentlicht. Seit 2006 ist es im Besitzt von Oracle [Wikb]. Berkeley DB hat eine Open-Source-Lizenz und bezeichnet sich selbst als eingebettete Datenbank-Bibliothek mit anpassbarer High Performance [DB16]. Eingebettet bedeutet hier, dass das DBMS im selben Adressraum wie das Anwendungsprogramm läuft; dadurch ist keine zusätzliche Kommunikation zwischen einzelnen Prozessen des selben Rechners nötig. Des Weiteren besitzt es funktionsorientierte Programmierschnittstellen (API) für die gängigen Programmiersprachen wie z.B. PHP, JAVA oder C++. Die Datenbankbibliothek, in der alle Datenbankoperationen stattfinden, ist mit nur 300 KB klein genug um in eingebetteten System eingesetzt zu werden.
Berkeley DB unterstützt vier verschiedene Speicherstrukturen aus denen der Nutzer wählen kann: Hashtabellen, B-Bäume, Tupel und persistente Warteschlangen. Konzeptionell werden jedoch alle Daten als (Schlüssel, Wert)-Paare abgelegt. Es handelt sich bei Berkeley DB daher nicht um ein relationales DBMS, sondern um ein sogenanntes Key-Value-Stores nutzendes System. Diese sind eine einfachere Form von DBMS und liefern die Werte anhand eines Schlüssels wieder zurück. Um auf die gespeicherten Daten zuzugreifen, können unterschiedliche Zugriffsmethoden genutzt werden. Derzeit gibt es in Berkeley DB fünf verschiedene Zugriffsmethoden[DB16]:

- **B-Baum** Die Zugriffsmethode auf einen B-Baum nutzt einen sortieren und balancierten Baum um Einfüge-, Such- und Löschoperationen durchzuführen. Dabei braucht sie nur eine Zeit von $O(n)$, wobei n die Tiefe des Baumes ist. Berkeley DB nutzt ein optimiertes Einfüge-verfahren um Seiten gleichmäßig auszulasten.
- **Hashing** Diese Methode nutzt ein lineares Hashverfahren.
- **Heap** Die Heap-Zugriffsmethode speichert die Datensätze in Heaps. Ein Heap ist eine auf Bäumen basierende, abstrakte Datenstruktur. Die Datensätze werden referenziert allein durch die Seite und den Offset, in dem sie geschrieben worden sind.
- **Queue** Die Queue/Warteschlangen-Zugriffsmethode speichert Datensätze fester Größe, als Schlüssel werden dabei logische Adressen genutzt. Diese Methode wurde entwickelt um schnelles Einfügen von Datensätzen am Ende der Queue zu ermöglichen. Um Einträge am Anfang der Queue zu löschen oder zurückzugeben wird ein Cursor genutzt.
- **Recno** Der Zugriff läuft über eine Datensatznummer, damit können sowohl feste als auch variable Datensatzlängen gespeichert werden.

Aufgrund der Interaktion zwischen den Zugriffsmethoden und den Datenmengen der Anwendung führen die unterschiedlichen Methoden zu unterschiedlich schnellen Bearbeitungszeiten der Anfrage, daher sollte die Wahl der zugrundeliegenden Datenstruktur mit Bedacht erfolgen. Zusätzlich hat jeder Speicherstruktur seine eigenen Vor- bzw. Nachteile. Z.B. sind Heap Strukturen gut geeignet für eingebettete Systeme, da sie gut mit eingeschränkter Speichergröße zurecht kommt. Dafür muss bei einer Heap Struktur darauf geachtet werden, dass die Reihenfolge, in der Tupel zurückgegeben werden, nicht vorhersehbar ist. Daher beherrscht ein Heap im Vergleich zu einen B-Baum weniger Cursoroperationen.

Um Daten aus der Datenbank zurück zu holen gibt es einige (unabhängige) Standard-Zugriffsoperationen. Um diese zu nutzen, muss zunächst die Datenbank, mit dem Befehl $DB \rightarrow open()$, geöffnet werden. Bei diesem Befehl wird als Parameter festgelegt, welche Zugriffsmethode verwendet werden soll. Alternativ kann der Wert DB_UNKNOWN mitgeben werden, falls der Typ unbekannt ist. Nachdem die Datenbank geöffnet worden ist, können mit dem Befehl $DB \rightarrow get()$ Dateneinträge aus der Datenbank ausgelesen

werden. Dazu wird ein Schlüssel benutzt, der mit den gewünschten Datensatz assoziiert ist. Es gibt hierbei einige Optionen die genutzt werden können, um das Lesen der Daten anzupassen:

- **DB_GET_BOTH** sucht sowohl nach den passenden Schlüssel als auch den passenden Datensatz. Diese Operation ist nur dann erfolgreich, wenn beide mit den Datensätzen in der Datenbank übereinstimmt.
- **DB_RMW** RMW steht für Read-Modify-Write. Falls dieser Befehl gesetzt wurde, werden beim Lesen statt Lesesperren Schreibsperren gesetzt. Dadurch soll die Gefahr von Deadlocks reduziert werden.
- **DB_SET_RECNO** Falls eine B-Baum Speicherstruktur genutzt wurde, kann auf Datensätze auch über logical record numbers zugegriffen werden.

Bei Duplikaten wird $DB \rightarrow get()$ immer den ersten Eintrag der Duplikate zurückgeben, sofern bei der Konfiguration der Datenbank nichts anderes eingestellt worden ist.
Key-Value-Store Systeme bieten grundsätzlich weniger Anfragemöglichkeiten als ein SQL nutzendes Datenbanksystem und bieten meist nur exact-match-Anfragen, die zu einen bestimmten Schlüssel den zugehörigen Wert zurückliefern oder Bereichsanfragen. Diese liefern z.B. alle Werte, die sich zwischen zwei definierten Grenzen befinden, zurück. Solche Anfragen sind jedoch in nicht allen angebotenen Speicherstrukturen möglich. Für komplizierte Anfragen muss daher eine API herangezogen werden.

Mit Hilfe der Berkeley DB SQL-API lässt sich Berkeley DB wie eine relationale Datenbank bedienen und sich die daraus ergebenen Vorteile bei der Anfragekapazität nutzen. Die BerkeleyDB SQL API ist dabei nahezu funktionsgleich wie das DBMS SQLite. Es bietet die selben APIs, die gleiche Kommandozeilen-Umgebung, die gleichen PRAGMAs und, für uns am wichtigsten, es bietet die identischen SQL-Anweisungen wie in SQLite. PRAGMAs sind Anweisungen, die genutzt werden um die SQLite-Bibliothek zu verändern oder um Anfragen an interne Daten (Daten, die nicht in der Relation stehen) zu stellen. Es gibt jedoch auch Unterschiede zwischen BerkeleyDB SQL und SQLite, doch diese fallen hauptsächlich in den der Konfliktbewältigung von parallel ablaufenden Transitionen und sind daher für diese Arbeit nicht relevant. Eine Anweisung könnte mit der BerkeleyDB SQL wie folgt aussehen 2.5:

```
SELECT Angestellter, SUM(gehalt)
  FROM Firma
  WHERE gehalt >= 3500 AND gehalt <= 5500
  GROUP BY name ORDER BY name DESC;
```

Abbildung 2.5: Anfragebeispiel in SQLite oder BerkeleyDB SQL

Diese Anfrage wird eine Liste zurückgeben, wo alle Mitarbeiter und die Summe ihrer Gehälter angeben werden, gruppiert und sortiert nach ihren Namen mit der Bedingung, dass ihr Gehalt zwischen 3500 und 5500 Einheiten liegt. Bei dieser Anfrage würden Duplikate nicht beachtet werden, was je nach Situation ein Fehler sein kann. SQLite wird ausführlicher im Abschnitt 3.2 behandelt.

Kapitel 3

Mittelgroße Systeme

Die mittleren Systeme haben die Aufgabe, die Daten der vielen (bis zu 100) Sensoren der unteren Ebene zusammenzufassen und die Daten für die weitere Verarbeitung weiterzuleiten. Daher liegt in diesem Systemgröße der Fokus auf den einfachen Aggregatfunktionen, zum Generieren von z.B. Durchschnittswerten von Attributen und Joins von Tabellen, um Daten unterschiedlicher Herkunft zu verbinden. Um z.B. aus den Sensordaten den Durchschnittlichen Verbrauch an Nahrungsmitteln, in Abhängigkeit von der Zeit zu bilden, wünschen wir uns, dass die Windowfunktion vom DBMS unterstützt werden. Weiterhin legen wir in dieser Ebene mehr Wert auf die Organisation der Daten, daher soll hier die Sortierung und die Gruppierung der Daten eine wichtigere Rolle spielen.
Einsatzgebiet der mittelgroßen Systeme können z.B. Fernseher, Media Center bis hin zum lokalen Rechner sein. Wir haben uns hier entschieden die beiden Marktführer MySQL und SQLite zu untersuchen.

3.1 MySQL

MySQL ist ein relationales Datenbankmanagementsystem und wurde im Jahre 1994 von dem schwedischen Unternehmen MySQL AB entwickelt. Dieses wurde 2008 von Sun Microsystems übernommen, welches anschließend 2010 von Oracle gekauft wurde. MySQL ist das weltweit meist verbreitete Open-Source-Datenbanksystem [Wik16b].
Wie der Name verrät, ist MySQL an den SQL-Standard angelehnt und unterstützt die meisten Anfragestrukturen.

```
SELECT
  [ALL | DISTINCT |DISTINCTROW]
    [HIGH_PRIORITY]
    [MAX_STATEMENT_TIME = N]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr ...]
  [FROM table_references
  [PARTITION partition_list]
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC], ...]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  [PROCEDURE procedure_name(argument_list)]
  [INTO OUTFILE 'file_name'
      [CHARACTER SET charset_name]
      export_options
    | INTO DUMPFILE 'file_name'
    | INTO var_name [, var_name]]
  [FOR UPDATE | LOCK IN SHARE MODE]]
```

Abbildung 3.1: MySQL Select Syntax [MyS16, S. 1867]

Wie in Abbildung 3.1 zu sehen gehören dazu SELECT, FROM, WHERE, HAVING, GROUP BY und ORDER BY. MySQL bietet folgende Verbundmethoden: INNER JOIN, OUTER JOIN, CROSS JOIN und
STRAIGHT_JOIN. Als Optionen für die Verbundmethoden werden NATURAL, LEFT und RIGHT unterstützt. MySQL bietet kein FULL OUTER JOIN, falls dies benötigt wird, muss ein LEFT OUTER JOIN und ein RIGHT OUTER JOIN mit UNION verbunden werden. Statt INNER JOIN kann auch JOIN oder einfach nur ein Komma gesetzt werden, alle diese Varianten sind äquivalent. Eine weiter Besonderheit ist, dass bei MySQL CROSS JOIN die selbe Ergebnismenge ausgibt, wie ein JOIN ohne die Nutzung einer JOIN-Bedingung. Solche Bedingungen können ON oder USING sein. Zusätzlich können alle JOINs auch die Option PARTITION enthalten. Falls das der Fall ist, werden nur die Zeilen miteinander verschmolzen, die in der Partition aufgelistet sind. Zum Umbenennen von Tabellen als auch Spalten kann der Ausdruck AS genutzt werden [MyS16, S. 1881].
Eine weitere Anweisung die der Abbildung 3.1 zu sehen ist, ist LIMIT. Mit ihr kann die Anzahl der Zeilen in der Ergebnismenge angepasst werden. So lässt sich zum Beispiel eine Anfrage stellen, die die teuersten fünf Flugreisen aus den Flugplan ausgibt 3.2.

```
SELECT *
FROM Flugplan
ORDER BY Kosten DESC
LIMIT 5
```

Abbildung 3.2: MySQL Anfrage mit LIMIT

Mit der PROCEDURE-Klausel können, außerhalb von der UNION Anweisung Prozeduren aufgerufen werden. Diese können vordefiniert sein, wie z.B. ANALYSE, oder sie wurden mit CREATE PROCEDURE selbst erstellt. MySQL bietet dem Nutzer keine Window-Funktion an. Dafür existiert aber die Funktion GROUPCONCAT. Mit GROUPCONCAT können Strings innerhalb einer Gruppe verkettet werden, und sie kann genutzt werden um die fehlende Window-Funktion zu ersetzen. Sie kann zwar nicht alle, aber jedoch einen Teil der Funktionalität einer Window-Funktion erfüllen, wie z.B. die relativ häufige Abfrage nach den größten k Elementen einer Gruppe (siehe Abbildung 3.3)

```
SELECT Continent, Name, SurfaceArea, Population
FROM   world.Country,
   (
     SELECT  GROUP_CONCAT(top_codes_per_group) AS top_codes
     FROM (
          SELECT SUBSTRING_INDEX(GROUP_CONCAT(Code ORDER BY SurfaceArea DESC)
                                        , ',', 5)
                                       AS top_codes_per_group
          FROM world.Country
          GROUP BY Continent
            ) s_top_codes_per_group
   ) s_top_codes
WHERE FIND_IN_SET(Code, top_codes)
ORDER BY Continent, SurfaceArea DESC;
```

Abbildung 3.3: Top k Anfrage von Gruppen mit GROUPCONCAT [Noa12]

In diesem Beispiel ist auch zu sehen, dass MySQL mit verschachtelten Anfragen, also einen weiteren SELECT-FROM-WHERE-Block in einer WHERE-Klausel, keine Probleme hat.
Was jedoch in der Abbildung 3.1 fehlt, ist die WITH-Klausel, weswegen MySQL auch nicht in der Lage ist, rekursive Anfragen an Relationen zu stellen. Die Aggregatfunktionen COUNT, AVG, SUM, MAX und MIN werden von MySQL angeboten. Weiterführende Aggregatfunktionen zur Datenanalyse unterstützt MySQL nur teilweise. Es werden Funktionen für die Standardabweichung, hier als STD bezeichnet, anstatt der Bezeichnung STDDEV_POP aus dem SQL-Standard von 2011, und Varianz VAR_POP geboten. Jedoch werden nicht die komplexeren Aggregatfunktionen für die lineare- oder quadratische Regression angeboten. Weiterhin bietet MySQL eine Fülle an arithmetischen und geometrischen Funktionen, mit denen sich solche Funktionen selber definieren lassen.

MySQL teilt seine Datentypen in Kategorien ein: numerische Typen, Typen für Datum und Uhrzeit String-Typen, Typen für Volumina und JSON-Datentypen.
Für die numerischen Typen gibt es die Optionen *Zerofill*, *Unsigned*, *D* und *M*, wobei MySQL bei *Zerofill* automatisch *Unsigned* für vorzeichenfreie Werte hinzufügt wird. Mit *M* wird die Anzeigebreite definiert, die maximale ist 255 sein kann. Für z.B. Gleitkommazahlen und Festkommazahlen bedeutet das, dass die maximale Anzahl an Stellen die gespeichert werden können 255 ist. Für Gleitkommazahlen z.B. FLOAT oder DOUBLE kann die Option *D* angegeben werden. Diese gibt an wie viele Stellen nach dem Komma gespeichert werden sollen. Weitere Beispiele für numerische Typen sind z.B. INT, DECIMAL und BOOLEAN. Für Gleitkommazahlen und Festkommazahlen werden zusätzlich verschiedene Größen angeboten.
Die wichtigsten Datentypen für Datum und Uhrzeit sind TIME, DATETIME, und TIMESTAMP. Bei diesen Typen ist es möglich eine Option anzugeben, mit der die Präzision (fraction) der Werte mit maximal auf die Mikrosekunde genau, festgelegt werden kann. MySQL repräsentiert den Type DATE im Format 'YYYY-MM-DD' in einem Wertebereich von '1000-01-01' zu '9999-12-31' und DATETIME im

Format 'YYYY-MM-DD HH:MM:SS[.fraction]' und im Wertebereich '1000-01-01 00:00:00.00' bis '9999-12-31 23:59:59'. Mit TIMESTAMP wird angegeben wie viele Sekunden seit '1970-01-01 00:00:00' (UTC) vergangen sind. Beachtet werden muss, dass bei temporalen Werten die Aggregatfunktionen SUM und AVG nicht funktionieren. Um dieses Problem zu umgehen, müssen die Werte zunächst in numerische Werte umgewandelt werden, dann die Aggregation durchgeführt und anschließend das Ergebnis wieder in temporale Werte zurück gewandelt werden.

3.2 SQLite

Ein weiteres System, welches ich hier als mittel großes System betrachten möchte, ist SQLite. Dieses relationale Datenbanksystem wird vor allem in eingebetteten Systemen eingesetzt und benötigt keine eigene Server-Umgebung. SQLite ist in C geschrieben worden, bietet jedoch API's zu nahezu allen gängigen Programmiersprachen und kann ohne Probleme Datenbanken von 32-Bit zu 64-Bit oder von big-endian zu little-endian Architekturen kopieren, was wohl ein wichtiger Grund ist, dass viele bekannte Plattformen SQLite nutzen. So ist SQLite z.B. auf jedem Android- oder iOS-Gerät zu finden [ST16a].
Weiterhin ist die Bibliothek von SQLite relativ klein, teilweise unter 500 Kibibyte, und benötigt zum arbeiten nur einen Stapelspeicher von 4 KiB und einen Speicher für einen Heap von 100 KiB. Wie auch bei MySQL existiert für SQLite eine kostenlose Lizenz. SQLite ist angelehnt an denn SQL-Standard von 1992 und beherrscht daher viele bekannte SQL-Anweisungen 3.4.

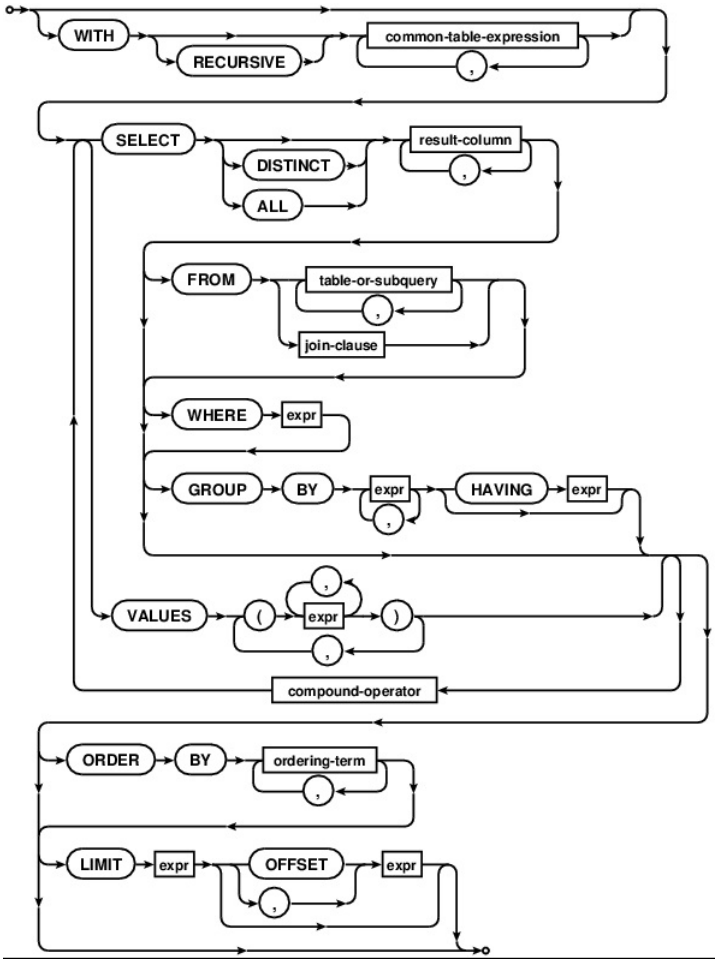


Abbildung 3.4: Select-Syntax von SQLite[ST16a]

Dazu gehören SELECT, FROM, WHERE, WITH, GROUP BY, HAVING, ORDER BY und RECURSIVE. SQLite unterstützt folgende JOIN Methoden: NATURAL JOIN, LEFT JOIN, LEFT OUTER JOIN, INNER JOIN und CROSS JOIN.

In der Abbildung 3.4 fällt auf, dass im Vergleich zu anderen Systemen (siehe MySQL 3.1 oder PostgreSQL 4.1) nach einem SELECT kein Ausdruck (=engl. expression) folgt, sondern ein result-column. Jedoch beinhaltet das result-column die übliche expression. Bei den Aggregatfunktionen werden nur die einfachen Aggregatfunktionen unterstützt, also AVG, COUNT, SUM, MAX und MIN. Auch SQLite bietet eine GROUP_CONCAT Funktion mit einen oder zwei Parametern. Eine Window-Funktion wird in SQLite nicht unterstützt. Hier muss der Nutzer versuchen sich mit den gegebenen Mitteln zu helfen.
SQLite nutzt ein dynamisches Typsystem, die Datentypen der Werte werden direkt mit einen Wert assoziiert und nicht erst auf einen Container, wie bei statischen Typisierungen üblich. Dazu werden Speicherklassen genutzt [ST16b]. Speicherklassen sind allgemeiner als Datentypen, so beinhaltet z.B. die Speicherklasse INTEGER sechs verschiedene Datentypen für unterschiedliche Größen. Jedoch werden alle Werte, wenn sie aus der Datenbank in den Arbeitsspeicher zur Bearbeitung geladen werden, in den allgemeinsten Typ umgewandelt. Alle Werte, die in SQLite Datenbanken gespeichert werden, gehören zu den folgenden fünf Speicherklassen:

- **NULL** Der Wert der NULL-Klasse ist NULL.
- **INTEGER** Der Wert der Klasse INTEGER ist ein Vorzeichen behafteter Integer-Wert.
- **REAL** Der Wert der Klasse REAL wird als 8 Byte Gleitkommaformat nach dem IEEE-Standard gespeichert.
- **TEXT** Der Wert der Speicherklasse TEXT ist ein String, der als UTF8, UTF16BE oder UTF16LE kodiert werden kann.
- **BLOB** Der Wert eines BLOB ist wiederum ein Blob von Daten, die genauso gespeichert wurden wie sie eingegeben worden sind. Blobs sind Binary large Objects und werden zum Speichern von z.B. Video- oder Audiodateien genutzt.

SQLite besitzt keine separate Boolean-Speicherklasse, stattdessen werden boolesche Werte als Integers gespeichert, 0 für Falsch und 1 für Wahr [ST16a].

Kapitel 4

Große Systeme

Große Systeme werden auf der obersten Ebene der Anfragebearbeitung eingesetzt und sollen in Cloud-Systemen verwendet werden. Auf dieser Ebene wird mit großen Datenmengen gearbeitet, die von unterschiedlichen Quellen zum Zweck der Analyse bzw. Entscheidungsunterstützung zusammengetragen werden. In diesem Zusammenhang wird oft von Data Warehouses gesprochen [CU97].Um diese großen Datenmengen zu analysieren, kann die OLAP-Methode genutzt werden. OLAP steht für On-Line Analytical Processing. Bei dieser hypothesengestützten Analysemethode muss vor der eigentlichen Analyse schon bekannt sein, welche Anfragen an das OLAP-Systeme gestellt werden soll. Das OLAP-Systeme gibt anschließend aus, ob die zuvor gestellte Hypothese mit den gegebenen Daten bestätigt oder widerlegt ist[Wik16c]. Um OLAP zu unterstützen, ist es für DBMS wichtig, dass sie die Möglichkeit bietet, mit dem multidimensionalen Datenmodellen, wie z.B. den Datenwürfel, arbeiten zu können. Um die Datenanalyse zu vereinfachen, sollen bestimmte Erweiterungen von bekannten SQL-Anweisungen verstanden werden. Dazu gehören die GROUP BY Erweiterungen CUBE, ROLLUP und GROUPING SET. Auch soll SELECT...OVER, besser bekannt als die Window-Funktion, vom DBMS unterstüzt werden.
Weiterhin sollen zur Datenanalyse und Dateninterpretation die komplexen Aggregatfunktionen 1.2 angewendet werden können, sowie die nötige Funktionalität zur Anwendung des Hidden Markov Modells, welches zur Intentionserkennung genutzt werden kann. Für das Hidden Markov Modell sollte das DBMS in der Lage sein, mit Matrizen und Vektoren zu arbeiten. Bei den großen Systemen betrachte ich drei verschiedene Systeme: PostgreSQL, Oracle DB und DB2. Wie auch bei den mittleren Systemen erwarten ich hier, dass die größeren Systeme alles können, was bereits die kleineren Systeme konnten.

4.1 PostgreSQL

PostgreSQL ist ein objekt-relationales DBMS, dass an der Universität von Kalifornien am Berkeley Computer Science Department entwickelt worden ist und ist der Open-Source Nachfolger eines ursprünglichen DBMS namens POSTGRES, welches wiederum auf dem System INGRES basiert.

```
    [WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ]
              [ NULLS { FIRST | LAST } ] [, ...]
          [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }
              [ OF table_name [,...] ] ]
```

Abbildung 4.1: Select Syntax von PostgreSQL [Gro]

Wie in Abbildung 4.1 zu erkennen, unterstützt PostgreSQL die bekannten SQL-Statements, als auch die gewünschte Window Funktion. In der FROM-Klausel können folgende Join-Operationen eingesetzt werden: CROSS JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN und FULL OUTER JOIN, jeweils mit der Option NATURAL. Als Verbindungsbedingungen stehen die Klauseln ON und USING bereit.
PostgreSQL bietet sowohl die einfachen als auch die komplexen Aggregatfunktionen, wie sie in 1.3 und 1.2 beschrieben worden sind [Pos, S.301], einschließlich mit der gleichen Syntax der Befehle. PostgreSQL bietet genau 20 verschiedene Datentypen an, dazu gehören z.B. numerische Typen, Typen für Datum und Uhrzeit, boolesche Typen und viele weitere. Zusätzlich bietet PostgreSQL die Möglichkeit, weitere Datentypen mit dem Befehl CREATE TYPE zu definieren[Pos, S.124ff].
Es gibt keine Datentypen für Matrizen oder Vektoren und daher auch keine Funktionen für Matrixoperationen wie z.B. der Matrixmultiplikation. Allgemein kann jedoch für jede Matrix eine Tabelle angelegt werden, anschließend können die gewünschten Operationen durchgeführt werden. Ein einfaches Beispiel könnte wie folgt aussehen 4.2:

```
SELECT A.i, A.j,
       (A.element_value + B.element_value) AS element_tot
  FROM MatrixA AS A, MatrixB AS B
 WHERE A.i = B.i AND A.j = B.j;
```

Abbildung 4.2: Addition zwei Matrizen gleicher Dimension [Cel12]

Eine andere Möglichkeit, eine Matrix zu speichern, können Arrays sein. PostgreSQL ermöglicht, dass die Spalten einer Tabelle als Längen veränderliche multidimensionale Arrays definiert werden können. Arrays können aus allen Standardtypen, benutzerdefinierten Basistypen, Aufzählungstypen oder Verbundtypen

erstellt werden. Arrays haben jedoch den Nachteil, dass wenn nach Werten gesucht werden soll, alle angefasst werden müssen; dies wird bereits bei moderaten Größen aufwendig.

Mit Hilfe eines zusätzlichen Moduls namens cube (nach der Dokumentation von PostgreSQL 9.5.4 im Anhang F.8 [Pos, 3061]) ist es möglich, einen Datentyp *cube* zu erstellen, der dazu dient multidimensionaler Würfel darzustellen. Ein multidimensionale Würfel wird mit folgenden Mittel dargestellt 4.1:

x	Ein ein-dimensionaler Punkt
$(x1, x2, ..., xn)$	Ein Punkt im n-dimensionalen Raum, dargestellt als leerer Würfel
$(x), (y)$	Ein ein-dimensionaler Intervall gestartet bei x und endet bei y
$(x1, ..., xn), (y1, ..., yn)$	Ein n-dimensionaler Würfel dargestellt durch diagonale Eckpaare

Tabelle 4.1: Möglichkeiten der externen Darstellung des Datentyps *cube* [Pos, 3062]

Bei dieser Repräsentierung spielt es keine Rolle in welcher Reihenfolge die entgegengesetzten Ecken eingegeben werden, bei der Verwendung von Funktionen auf den Würfel werden die Werte automatisch in eine interne Repräsentation umgewandelt. Alle Werte im n-dimensionalen Datenwürfel werden intern als 64 Bit Gleitkommazahl gespeichert, dies bedeutet, dass Zahlen mit mehr als etwa 16 signifikanten Ziffern abgeschnitten werden. Weiterhin im Modul F.8. cube enthalten sind die GiST Indexoperatoren 4.2:

Operator	**Beschreibung**
$a = b$	Die Würfel a und b sind identisch
$a \&\& b$	Die Würfel a und b überlappen
$a @ > b$	Der Würfel a beinhaltet Würfel b
$a < @ b$	Der Würfel a ist enthalten in Würfel b

Tabelle 4.2: GiST Operatoren des Datentyps *cube* [Gro, 3062]

GiST steht für Generalized Search Tree und ist eine Zugriffsmethode auf balancierte Bäume. Dabei wird GiST oft als Schablone für die Implementierung beliebiger Index-Strukturen genutzt. Weiterhin sind im Modul enthalten die Standard Operatoren kleiner $<$ und größer $>$ die z.B. bei der Sortierung eingesetzt werden können. Letztlich sind enthalten eine Reihe von Funktionen die mit dem Datentyp *cube* arbeiten können z.B.:

$cube(float8[], float8[])\, returns\, cube$
: Erzeugt einen Würfel mit den diagonalen Ecken, wie sie in den Arrays beschrieben worden sind, dabei müssen beide Arrays gleich groß sein.

$cube\_subset(cube, []int)\, returns\, cube$
: Erschafft aus einen bestehenden Würfel einen neuen Würfel. Diese Funktion wird auch genutzt um die Dimensionen eines Würfels zu reduzieren.

$cube\_union(cube, cube)\, returns\, cube$
: Fasst zwei Würfel zusammen.

$cube\_inter(cube, cube)\, returns\, cube$
: Erstellt die Schnittmenge von zwei Würfeln.

$cube\_enlarge(cube\, c, double\, r, int\, n) returns\, cube$
: Erweitert die Größe von einen Würfel um einen bestimmten Radius r für mindestens n Dimensionen. Falls der Radius negativ ist, verkleinert sich der Würfel. Diese Funktion ist nützlich bei der Suche nach benachbarten Punkten, durch die Bildung von Begrenzungsboxen.

Ein weiteres Modul ist Itree (nach der Dokumentation von PostgreSQL 9.5.4 im Anhang F.20 [Pos, 3127]), dieses Modul ermöglicht die Repräsentierung von Datenlabels in Hierarchisch aufgebauten Bäumen. Ein *label* ist ein Sequenz von alphanumerischen Zeichen. Weiterhin muss ein *label* immer kleiner als 256 Byte sein. Um den Speicherort innerhalb eines Baumes darzustellen, werden *labelpath* genutzt. Ein *labelpath* ist eine Folge von *labels*, die einen Pfad von der Wurzel bis zu einen Knoten darstellt. Demzufolge speichert ein Itree eine Menge von *labelpath*.
Um Daten aus dem Itree zu holen wird lquery genutzt. Mit lquery kann ein Ausdruck artiges Muster genutzt werden um Vergleiche durchzuführen 4.3.

```
foo          Sucht nach dem exakten label path namens foo
*.foo.*      Sucht jeden label path, der ein Label namens foo enthält
*.foo        Sucht jeden label path, wo das letzte Label foo ist
```

Abbildung 4.3: [Gro, S.3127]

Neben den Standard-Vergleichsoperatoren die im Modul angeboten werden, gibt es noch eine Reihe von Funktionen, die auf hierarchische Bäume (Itrees) angewendet werden können 4.3.

Funktion	**Beschreibung**
$subltree(ltree, int\,start, int\,end)b$	Gibt einen Teilbaum zurück auf Basis zweier Positionen
$subpath(ltree, int\,offset, int\,len)$	Erstellt einen Teilbaum, der an der Position des Offsets beginnt und eine vorgegebene Länge *len* hat.
$subpath(ltree, int\,offset)$	Bestimmt Teilbaum, der beginnt an der Position des Offsets und erweitert sich bis zum Ende des Pfades.
$index(ltree\,a, ltree\,b)$	Gibt die Position (integer) des ersten Auftretens von Itree b in Itree a zurück, die Suche beginnt am Offset.
$nlevel(ltree)$	Errechnet die Anzahl von *labels* im Pfad.
$text2ltree(text)$	Formt einen Text zu einen ltree um
$ltree2text(ltree)$	Formt einen Itree zu einen Text um
$lca(ltree[])$	Gibt eine Baum zurück auf Basis des kleinsten gemeinsamen Vorfahren

Tabelle 4.3: Auszug aus dem im Modul Itree vorgestellten Funktionen [Pos, S.3129]

Letztlich unterstützt das Modul Itree verschieden Indexstrukturen, die genutzt werden können um Operationen effizienter zu machen.
Ohne eine zusätzliches Modul kann PostgreSQL die für OLAP-Operationen wichtigen Anweisungen GROUPING SETS, CUBE, und ROLLUP anwenden. CUBE und ROLLUP können entweder direkt in der GROUP BY oder verschachtelt in einer GROUPING SETS-Anweisung verwendet werden. Falls eine GROUPING SETS Anweisung in einer anderen GROUPING SETS Anweisung steht, ist der Effekt gleich, als wenn alle Elemente der inneren Anweisung direkt an die äußere Anweisung gegeben worden sind [Gro].Falls eine Anfrage eine Window-Funktion enthält, werden diese Funktionen nach der Gruppierung, Aggregation, und HAVING-Filtern ausgeführt.

4.2 Oracle Database

Das zweite DBMS, welches ich bei den großen Systemen betrachten möchte, ist Oracle Database Version 12.1 von der Firma Oracle. Oracle Database ist ein objektrelationales Datenbanksystem welches schon seid 1979 unter dem Namen Oracle V2 auf dem Markt erschienen ist und seitdem ständig erweitert wurde. Eine objektrelationale Datenbank beinhaltet neben den Konzepte einer klassischen relationalen Datenbank auch Prinzipien, der objektorientierten Programmierung aus der Softwaretechnik. Dadurch bieten objektrelationale Datenbanken diverse Vorteile gegenüber reinen relationalen Datenbanken. Ein Vorteil ist, dass es möglich ist Objekten zu speichern, die wiederum aus anderen Objekten bestehen können. Zusätzlich gibt es die Möglichkeit dabei Konsistenzbedingungen zu setzen. Zum Beispiel, ein Schrank besteht aus diversen Brettern und Schrauben, wenn dann die Größe des Schrankes geändert wird, erhöht sich automatisch die Anzahl der Schrauben. Weiterhin ermöglichen objektrelationale Datenbanken, die Verwaltung verschiedener Repräsentationen desselben Objektes, die z.B. bei einen Update, alle simultan zu ändern sind. Auch bietet die Nutzung von Objekten, den Vorteil der Wiedernutzung von alten Objekten um neue Objekte zu erstellen, dadurch kann bei der Erstellung von neuen Datentypen Zeit gespart werden [TB10].
Die Abbildung zeigt den Aufbau einer Anfrage in Oracle Database 4.4 Hier ist schon ein Unterschied

```
[ with_clause ]
SELECT [ hint ] [ { { DISTINCT | UNIQUE } | ALL } ] select_list
  FROM { table_reference | join_clause | ( join_clause ) }
       [ , { table_reference | join_clause | (join_clause) } ] ...
  [ where_clause ]
  [ hierarchical_query_clause ]
  [ group_by_clause ]
  [ model_clause ]
```

Abbildung 4.4: Syntax eines Anfrage Blocks in Oracle Database [Ora16, S. 9-2]

zu relationalen Datenbanken zu erkennen. Zusätzlich zu den bekannten Klausel gibt es hier eine extra Klausel für die hierarchische Anfrage. Weiterhin in der Abbildung zu sehen ist die Option UNIQUE. Diese ist funktionsgleich zu DISTINCT und entfernt Duplikate aus dem Ergebnis der Anfrage.
Falls eine Tabelle hierarchisch aufgebaute Daten enthält, dann müssen hierarchische Anfragen genutzt werden um die gewünschten Zeilen zu erhalten 4.5.

```
CONNECT BY [ NOCYCLE ] condition [ START WITH condition ]
| START WITH condition CONNECT BY [ NOCYCLE ] condition
```

Abbildung 4.5: Syntax der hierarchical_query_clause [Ora16, S. 9-3]

Eine hierarchical_query_clause kann entweder mit START WITH oder direkt mit CONNECT BY beginnen. START WITH definiert die Wurzelzeilen der Hierarchie und CONNECT BY spezifiziert die Beziehung zwischen Eltern- und Kindzeile der Hierarchie. Die einfachste Form einer hierarchischen Anfrage benötigt nur die Information wie jedes Kind mit seinen Eltern in Verbindung steht. Dazu wird CONNECT BY ... PRIOR genutzt 4.6.

```
SELECT employee_id, last_name, manager_id
  FROM employees
  CONNECT BY PRIOR employee_id = manager_id;
```

Abbildung 4.6: Beispiel für eine hierarchische Anfrage [Ora16, S. 9-5]

Um Hierarchien effektiv zu benutzen, stehen den Nutzer folgende Funktionen bereit:

Funktion	Beschreibung
LEVEL	Bestimmt die Position der gewählten Zeile in Abhängigkeit zum Wurzelknoten.
CONNECT_BY_ROOT	Gibt die Elternknoten zurück, die in Beziehung stehen zur aktuellen Zeile.
SYS_CONNECT_BY_PATH	Gibt ein Pfad von der Wurzel zur aktuellen Zeile an.
CONNECT_BY_ISLEAF	Gibt an, ob die aktuelle Zeile ein Blatt ist.
ORDER_SIBLINGS_BY	Gibt Geschwisterknoten eine definierte Ordnung ohne die Grundlegende Struktur zu ändern.

Tabelle 4.4: Funktionen für Hierarchien [Ora16, S. 9-3]

In der FROM-Klausel werden die schon bekannten Joins geboten, dazu gehören zum einen Inner- und Outer-Join, sowie Joins zur Bildung des Kartesischen Produkts. Zusätzlich werden aber noch weitere Tabellenverbundoperationen angeboten wie z.B. Self Joins und Antijoins. Bei Self Joins wird eine Tabelle mit sich selbst verbunden und erscheint somit zweimal in der FROM-Klausel. Antijoins geben die Zeilen der linken Tabelle zurück, zu denen keine passenden Gegenstücke in der rechten Tabelle gefunden worden sind nicht. Für Anitjoins wird die Anweisung NOT IN benutzt.
Auch Oracle Database besitzt die Möglichkeit Datenwürfel zu verwenden. Dabei können folgende Datentypen im Datenwürfel verwendet werden: numerische Datentypen, Datentypen für Uhrzeit und Datum sowie Stringtypen [Ora14a, 1-2].
Auf Datenwürfel können die OLAP-Funktionen angewendet werden. Diese erweitern die analytischen Funktionen von SQL und können in folgende Kategorien eingeteilt werden:

Aggregatfunktionen Beinhaltet die bekannten einfachen Aggregatfunktionen1.2

Hierarchische Funktionen Enthält Funktionen, die Informationen über vorliegende Hierarchien zurückgeben, wie z.B. Tiefe einer Hierarchie (Ebenen), Vor- oder Nachfolger einer Hierarchie, usw.

Lag Funktionen Diese Funktionen beziehen sich auf einen vorher gegangen Zeitpunkt und geben z.B. Differenzen zwischen alten und neuen Werten zurück.

OLAP DML-Funktion Führt Ausdrücke in der OLAP DML-Sprache aus, das können z.B. Funktions- oder Programmaufrufe sein.

Rank Funktionen Sind Funktionen, die z.B. innerhalb einer Dimension die Werte ordnen .

Share Funktion Diese Funktion berechnet das Verhältnis zwischen einem Wert des angegebenen Ausdrucks eines Elements der aktuellen Dimension und dem Wert eines in Beziehung stehenden Elementes der selben Dimension.

Zusätzlich gibt es neben den OLAP-Funktionen auch OLAP-Zeilenfunktionen, diese erweitern die Syntax der SQL-Zeilenfunktionen auf die Anwendung auf multidimensionale Objekte [Ora14a, 3-1]. Um Data Warehouses besser zu unterstützen, bietet auch Orcale DataBase eine Reihe von Funktionalitäten.

Zunächst einmal die GROUP BY Erweiterungen ROLLUP ,CUBE und GROUPING SETS. Diese Ausdrücke berechnen eine einzige Ergebnismenge, ähnlich wie bei UNION ALL. Ein wichtiger Vorteil dieser Ausdrücke ist es, dass sie parallelisiert werden können, was bei großen Datenmengen, wie sie in Data Warehouses vorkommen, essentiell ist.
Der Ausdruck ROLLUP kann auch nur partiell durchgeführt werden, dann wird nur ein Teil aller möglichen Aggregationen durchgeführt. Solch ein Partial Rollup hat folgende Syntax4.7.

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

Abbildung 4.7: Syntax eines partiellen ROLLUP [Ora14b]

In diesem Fall würden drei Aggregationen durchlaufen werden. Zuerst würde (expr1, expr2, expr3) berechnet werden, anschließend (expr1, expr2) und letztlich würde (expr1) berechnet werden. Auch für den Ausdruck CUBE gibt es eine partielle Variante. Mit Partial CUBE werden die Dimension des Datenwürfels begrenzt und es ermöglicht damit die Menge der stattfindenden Operationen zu begrenzen. Die Syntax eines Partial CUBE sieht wie 4.8 aus:

```
GROUP BY expr1, CUBE(expr2, expr3)
```

Abbildung 4.8: Syntax eines partiellen CUBE [Ora14b]

Hierbei würden vier Berechnungen durchlaufen werden. Als erstes würde (expr1, expr2, expr3) berechnet werden anschließend (expr1, expr2) dann (expr1, expr3) und zuletzt (expr1). Sowohl ROLLUP als auch CUBE können durch SELECT-Anweisungen kombiniert mit UNION ALL dargestellt werden. Dabei würde jedoch für ein n-Dimensionalen Würfel 2^n SELECT Anweisungen nötig sein, d.h. für jede weitere Dimension die hinzugefügt wird verdoppelt sich die Anzahl der Anweisungen.
Um zu bestimmen, welche Gruppen in ROLLUP und CUBE berechnet werden sollen, gibt es die GROUPING Funktionen. Weiterhin ermöglicht GROUPING die Unterscheidung zwischen NULL als zuvor gespeicherten Wert und Null als Ergebnis von CUBE oder ROOLUP. Falls GROUPING auf einen Wert NULL trifft, entstanden durch die Verwendung von CUBE oder ROOLUP, gibt sie eine 1 zurück unter der Bedingung, dass die Zeile ein Zwischenergebnis ist. Bei jedem anderen Wert gibt sie eine Null zurück.

Bei den Aggregatfunktionen sind die einfachen Aggregatfunktionen MIN, MAX, AVG, SUM und COUNT vorhanden. Für die lineare Regression stehen insgesamt neun Funktionen bereit, darunter auch die geforderten REGR_SLOPE, REGR_INTERCEPT und REGR_R2. Auch die restlichen Funktionen die in der Tabelle 1.2 sind, sind mit der gleichen Benennung wie im SQL-Standard von 2011 vorhanden [Ora16, S. 7-11].

4.3 DB2

Ein weiteres relationales DBMS ist DB2, welches zur Zeit mit der Version 11.1.0 auf dem Markt ist. DB2 beruht auf den DBMS System R welches in den 1970er Jahren entwickelt wurde. System R war das erste relationale Datenbankmanagementsystem. Es nutzte außerdem die Abfragesprache SEQUEL (=Structured English Query Language). Beide diese Systeme waren grundsteinlegend für die SQL-Abfragesprache. DB2 und System R wurden beide von der Firma IBM entwickelt [Wik16a].
Bei diesem Hintergrund ist es daher nicht verwunderlich, dass DB2 alle zuvor bereits betrachteten SQL-Anweisungen als auch die Window-Funktion nutzen kann. Auch bei den Joins sind die bekannten INNER JOIN, OUTER JOIN und CROSS JOIN Anweisungen vorhanden, mit den Optionen LEFT, RIGHT und

FULL. In der Version 11.1.0 unterstützt DB2 die einfachen Aggregatfunktionen 1.3. Bei den komplexen Aggregatfunktionen kann für COVAR_POP auch COVAR oder COVARIANCE angegeben werden. Die Funktionen für die Varianz und Standardabweichung, als auch für die Bestimmung des Korrelationskoeffizient sind vorhanden. Letztlich sind auch die Aggregatfunktionen für die Regressionsanalyse REGR_SLOPE, REGR_INTERCEPT und REGR_R2 sind vorhanden.

Darüber hinaus ermöglicht DB2 rekursive Anfragen auf die Datenmenge, wobei den Nutzer die Möglichkeit eingeräumt wird, zu entscheiden, ob die Rekursion als Tiefen- oder Breitensuche durch geführt werden soll [IBMd]. Um zu entscheiden welche Methode genutzt werden soll, wird die SEARCH BY Klausel gesetzt. Wenn eine Breitensuche gewählte wurde, werden vom Elternknoten aus, zuerst alle Kinder durchlaufen und anschließend alle Kindeskinder. Dies wird solange fortgesetzt, bis der Suchbaum erschöpft ist. Bei der Tiefensuche werden zunächst alle Nachfahren eines Kindes durchlaufen. Die Wahl der Suchmethode ändert in der Ergebnismenge, die Reihenfolge der erhaltenen Zeilen. Abbildung 1.4 zeigt ein Beispiel mit einer Tiefensuche.

```
WITH RECURSIVE Strecke (Startort, Zielort) AS (
        SELECT Startort, Zielort
        FROM Fahrplan
        WHERE Startort='Berlin'
                        UNION ALL
          SELECT A.Startort, B.Zielort
          FROM Strecke A, Fahrplan B
          WHERE A.Startort = B.Zielort)
          SEARCH DEPTH FIRST BY Startort
SELECT DISTINCT * FROM STRECKE
```

Abbildung 4.9: Erweiterung des Beispiels 1.4 mit SEARCH BY

In DB2 gibt es keinen Datentyp für die Darstellung von Datenwürfeln, für mehrdimensionale Objekte werden sogenannte Multidimensional-clustering-tables genutzt. Diese wurden speziell für Data Warehouses entwickelt und werden auch bei OLAP genutzt.
Die GROUP BY Erweiterungen, die bei Data Warehouses häufig zum Einsatz kommen, ROLLUP und CUBE, werden bei DB2 auch als Super-Groups bezeichnet. Weiterführend bietet DB2 die Möglichkeit, dass statt der bekannten Anwendung von ROLLUP, es auch möglich ist, mit WITH ROLLUP zu arbeiten. Statt ROLLUP gefolgt von einer grouping-expression-list, würde dann zuerst die grouping-expression-list stehen gefolgt von der Anweisung WITH ROLLUP. Diese Methode ist aber nur dann möglich, wenn sie allein in der GROUP-BY-Klausel steht. Analog ist dies auch mit der CUBE-Anweisung möglich. DB2 erlaubt weiterhin die Anwendung von GROUPING SETS. Verschiedene Typen von GROUP BY Anweisungen können auch miteinander Verbunden werden 4.10. Dabei werden Duplikate aus jeden GROUPING SET entfernt.

```
GROUP BY a, ROLLUP(b,c)
```

Abbildung 4.10: Kombination von GROUP BY Anweisungen [IBMb]

Datentypen werden in DB2 in fünf Hauptkategorien eingeteilt, diese heißen: datetime, string, boolean, signed numeric, und extensible markup language (XML) [IBMa].

datetime
: datetime beinhaltet die Datentypen time, timestamp und date.

string
In der Kategorie string gibt es folgende Unterkategorien: charakter, graphic und binary. Für jeden dieser drei Typen gibt es je zwei Möglichkeiten: einmal eine feste Länge und einmal eine variable Länge des Datentyps. Bei charakter gibt es den Datentyp Char als Datentyp fester Länge und die Datentypen Varchar und Clob als Datentypen variabler Länge. In der Unterkategorie graphic gibt es den Datentyp fester Länge graphic, und bei den Datentypen variabler Länge gibt es Vargraphic und Dbclob. In der letzten Unterkategorie gibt es die Datentypen Binary,als Datentyp fester Länge, Varbinary und Blob, als Datentypen variabler Länge.

boolean
Beinhaltet den Boolean-Datentyp zur Darstellung von true und false.

signed numeric
Hier enthalten sind exakte Datentypen, sowie approximierte Datentypen, und dezimal Gleitkommazahlen. Exakte Datentypen können binäre Integer sein, wie Integer oder Bigint je nach Größe des Datentyps, oder aber Dezimal. Für Dezimal-Gleitkommazahlen wird ein Typ namens Decfloat angeboten. Approximierte Datentypen werden in zwei Detailgraden angeboten, in einfacher (Real) und in doppelter Präzision (Double).

extensible markup language
Enthält Typen zur Darstellung von XML-Dokumenten.

Alle Datentypen sind in der Lage den NULL-Wert zu repräsentieren, außer sie stehen in einer als NOT NULL definierten Spalte. Neben diese standardmäßig vorhanden Typen gibt es auch die Möglichkeit eigene Datentypen aus den vorhandenen zu erstellen. Es gibt sechs verschiedene benutzerdefinierbare Datentypen. Einer davon heißt structured type. Dieser Datentyp erlaubt es Strukturierte Typhierarchien aufzubauen. Damit ist es möglich, Objekte zu definieren, die sich aus mehreren verschiedenen Dateitypen zusammensetzen. Solche strukturierte Typen haben den Vorteil, dass das Prinzip der Vererbung genutzt werden kann. Dass heißt, dass andere Objekte als Untertypen fungieren können, die von einem Supertyp die Struktur erben können um diese dann zu erweitern. Mit diesem Datentyp kommen objektrelationale Datenbankeigenschaften nach DB2 [IBMc].

Kapitel 5

Übersicht

Nachdem die einzelnen Systeme untersucht worden sind, folgt eine Übersicht der gewonnen Erkenntnisse über die Anfragekapazitäten der unterschiedlichen Systeme. In dieser Tabelle steht ein ✓dafür, dass die angegebene Funktionalität unterstützt wird und das Zeichen X bedeutet, dass die Angegebene Funktionalität nicht unterstützt wird.

Übersicht zu den DBMS								
		Kleine Systeme		Mittler Systeme		Große Systeme		
Anweisungen	Art	TinyDB	BDB-SQL	MySQL	SQLite	PostgreSQL	Oracle DB	DB2
Selektion	5.1	✓	✓	✓	✓	✓	✓	✓
	5.2	✓	✓	✓	✓	✓	✓	✓
	5.3	✓	✓	✓	✓	✓	✓	✓
Projektion	5.4	X	✓	✓	✓	✓	✓	✓
Order BY	5.8	X	✓	✓	✓	✓	✓	✓
GROUP BY	5.8	✓	✓	✓	✓	✓	✓	✓
Joins	5.5	X	✓	✓	✓	✓	✓	✓
	5.6	X	✓	✓	✓	✓	✓	✓
	5.7	X	✓	✓	✓	✓	✓	✓
einfache Aggregate 5.9	AVG	X	✓	✓	✓	✓	✓	✓
	COUNT	X	✓	✓	✓	✓	✓	✓
	MIN	X	✓	✓	✓	✓	✓	✓
	MAX	X	✓	✓	✓	✓	✓	✓
	SUM	X	✓	✓	✓	✓	✓	✓
Komplexe Aggregate 5.10	VAR_POP	X	X	✓	X	✓	✓	✓
	STDDEV_POP	X	X	✓	X	✓	✓	✓
	REGR_SLOPE	X	X	X	X	✓	✓	✓
	REGR_INT	X	X	X	X	✓	✓	✓
	REGR_R2	X	X	X	X	✓	✓	✓
	CORR	X	X	X	X	✓	✓	✓
	COVAR_POP	X	X	X	✓	✓	✓	✓
Rekursion	5.11	X	✓	X	✓	✓	✓	✓
Window Funktion	5.12	X	X	X	X	✓	✓	✓
OLAP Erweiterungen	GROUPING SETS	X	X	X	X	✓	✓	✓
	ROLLUP	X	X	✓	X	✓	✓	✓
	CUBE	X	X	✓	X	✓	✓	✓

Tabelle 5.1: Übersicht der untersuchten Anfragekapazitäten

Für jeweils ein DBMS der gegebenen Systemgrößen werden die nachfolgenden Anfragen getestet. Bei den kleinen Systemen wird TinyDB getestet, bei den mittlere Systemen MySQL und bei den großen Systemen wird DB2 untersucht. Dabei soll die Syntax der Anfragen keine Rolle spielen, sonder nur mit der Funktionalität, der gezeigten Anfrage, übereinstimmen. Bei allen anderen Systemen wird angenommen, dass die Anfragekapazitäten mit den Herstellerangaben übereinstimmen.

Zunächst zu den drei Selektionen: bei der ersten Selektion 5.1 soll bei einer genauen Übereinstimmung eines Wertes des Attributes Kosten das zugehörige Tupel auswerfen.

```
SELECT *
FROM Tabelle
WHERE Kosten=150
```

Abbildung 5.1: Selektion 1

Die zweite Selektionsoperation soll alle Tupel ausgeben, bei denen die Kosten größer gleich 30 Einheiten sind.

```
SELECT *
FROM Tabelle
WHERE Kosten>=30
```

Abbildung 5.2: Selektion 2

In der dritten Selektion wird mit der Hilfe eines logischen Operators, zwei Selektionsbedingungen zusammengeführt. Es werden dabei nur Tupel ausgegeben, die beim Attribut Kosten einen Wert zwischen 30 und 149 haben. Andere mögliche logische Operatoren wären ODER bzw. NICHT.

```
SELECT *
FROM Tabelle
WHERE Kosten<150 and Kosten>=30 OR
```

Abbildung 5.3: Selektion 3

Abbildung 5.4 zeigt eine Projektion, d.h. aus den Tupeln der Ergebnismenge werden nicht mehr alle Attribute angezeigt.

```
SELECT Preis
FROM Flugplan
```

Abbildung 5.4: Projektion über der Datenmenge

Inner Joins Kombinieren zwei Tabellen miteinander. Dabei wird jede Zeile mit in die Ergebnismenge genommen, die die Join Bedingung erfüllen. Anstatt der Anweisung INNER JOIN wird häufig auch einfach nur ein Komma in der FROM-Klausel gesetzt. Im folgenden Beispiel 5.5 wird über den Standort der Hotels an den jeweiligen Ankunftsflughafen, in diesem Fall Berlin, die Tabellen verbunden.

```
SELECT Hotel.Name, Flugplan.Ankunft, Hotel.Preis
  FROM Hotels INNER JOIN Flugplan
  ON Hotel.Stadt = Flugplan.Ankunft
        WHERE Ankunft='Berlin';
```

Abbildung 5.5: Inner Join

Es gibt drei Arten von OUTER JOINs: LEFT OUTER JOIN, RIGHT OUTER JOIN und FULL OUTER JOIN. Folgende Abbildung 5.6 zeigt einen LEFT OUTER JOIN. Bei diesem Join behalten wir die Tupel der linken Tabelle in der Ergebnismenge, aus der rechten Tabelle werden nur diejenigen übernommen, die zur der Verknüpfungsbedingung passen.

```
SELECT Hotel.Name, Flugplan.Ankunft, Hotel.Preis
FROM Hotels LEFT OUTER JOIN Flugplan
            ON (Hotel.Stadt= Flugplan.Ankunft)
                WHERE Ankunft='Berlin';
```

Abbildung 5.6: Outer Join

Ein Cross-Join bestimmt das Kartesisches Produkt von zwei Tabellen.

```
SELECT*
FROM Flugplan CROSS JOIN Hotels
```

Abbildung 5.7: Cross Join

Das Beispiel für eine Sortierung und Gruppierung sieht wie folgt aus 5.8:

```
SELECT Stadt,
       COUNT(*) AS Anzahl
  FROM Hotel
  GROUP BY Stadt
  ORDER BY Stadt;
```

Abbildung 5.8: Group by und Order by

Aus den fünf Aggregatfunktionen wurde beispielhaft die COUNT-Funktion zum Zählen von Tupeln gewählt 5.9.

```
SELECT COUNT(*)
FROM Hotels
```

Abbildung 5.9: einfache Aggregation

Für die komplexen Aggregationen wurde beispielhaft die Funktion CORR zur Bestimmung des Korrelationskoeffizenten gewählt 5.10.

```
SELECT CORR(Preis, Buchungen)
FROM Hotels
```

Abbildung 5.10: komplexere Aggregation für die Datenanalyse

Als Beispiel für einen rekursive Anfrage, wurde die schon bekannte aus Kapitel 2 genommen 5.11.

```
WITH RECURSIVE Strecke (Startort, Zielort) AS (
        SELECT Startort, Zielort
        FROM Fahrplan
        WHERE Startort='Berlin'
                        UNION ALL
        SELECT A.Startort, B.Zielort
        FROM Strecke A, Fahrplan B
        WHERE A.Startort = B.Zielort)
SELECT DISTINCT * FROM STRECKE
```

Abbildung 5.11: Rekursive Abfrage einer Stecke von Berlin [Sch10]

Eine Test Window-Funktion für DB2 auf der Beispieldatenbank von IBM.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
  RANK() OVER(ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMP WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME;
```

Abbildung 5.12: Window-Funktion

Kapitel 6

Prototypische Umsetzung

Wie in der Einleitung erwähnt, arbeitet der Lehrstuhl Datenbank- und Informationssysteme der Universität Rostock an dem Privacy-by-Design-Prinzip. Ein Projekt, dass mit diesem Ansatz arbeitet, nennt sich PArADISE. PArADISE soll als Prototyp für Softwareentwickler im Bereich der Modellbildung eingesetzt werden, um ihnen bei der datenschutzgerechten Entwicklung von Assistenzsystemen für die Intentions- und Situationserkennung zu helfen [GH16]. Weiterhin wird im Rahmen dieses Projektes ein JDBC-Treiber genutzt, um mit den unterschiedlichen Datenbanken zu kommunizieren.

```
1  public class Info extends PrivacyDatabaseMetaData{
2   boolean TinyDB_GroupBy =false;
3   boolean TinyDB_OrderBy =false;
4   boolean TinyDB_Select=true;
5   boolean TinyDB_projection=false;
6   boolean TinyDB_InnerJoin=false;
7   boolean TinyDB_OuterJoin=false;
8   boolean TinyDB_CrossJoin=false;
9   boolean TinyDB_Avg=false;
10  boolean TinyDB_Max=false;
11  boolean TinyDB_Min=false;
12  boolean TinyDB_Count=false;
13  boolean TinyDB_Sum =false;
14  boolean TinyDB_VarPOP =false;
15  boolean TinyDB_RegrSlope =false;
16  boolean TinyDB_RegrIntercept =false;
17  boolean TinyDB_RegrR2 =false;
18  boolean TinyDB_Corr =false;
19  boolean TinyDB_CovarPop =false;
20  boolean TinyDB_Stddevpop =false;
21  boolean TinyDB_Window =false;
22  boolean TinyDB_Rekursive =false;
23  boolean TinyDB_GroupingSets =false;
24  boolean TinyDB_Rollup=false;
25  boolean TinyDB_Cube=false;
26 }
```

Abbildung 6.1: Erweiterung der Metadatenabfrage des JDBC-Treibers

Für diesen JDBC-Treiber sollen weitere Funktionen, zu den bereits vorhandene hinzugefügt werden. Diese sollen zur Abfrage der Anfragekapazität der unterschiedlichen DBMS eingesetzt werden. Dazu wurde zunächst eine neue Klasse Info erstellt, die die gesammelten Informationen über die DBMS enthält. In Abbildung 6.1 zeigt einen Ausschnitt dieser Klasse, der die Informationen über TinyDB enthält. Anschließend wurden neue Funktionen zu einer bestehenden Metadaten-Klasse hinzugefügt. Hier zu sehen ist die Anfrage ob das DBMS die CROSS JOIN-Operation unterstützt 6.2.

```
1  public class PrivacyDatabaseMetaData implements DatabaseMetaData{
2
3    public enum Datenbank {
4     TinyDB, BDB_SQL, MySQL, SQLite , DB2, OracleDB, PostgreSQL
5                   }
6
7    Info info;
8    Datenbank datenbank;
9
10
11   public boolean supportsCrossJoin() throws SQLException {
12           boolean test;
13                   switch (datenbank) {
14     case TinyDB: test=info.isTinyDB_CrossJoin() ;
15       break;
16     case BDB_SQL:  test = info.isBDBSQL_CrossJoin();
17       break;
18     case MySQL:  test = info.isMySQL_CrossJoin();
19       break;
20     case SQLite:  test = info.isSQLite_CrossJoin();
21       break;
22     case DB2:  test = info.isDB2_CrossJoin();
23       break;
24     case PostgreSQL: test = info.isPostgreSQL_CrossJoin();
25       break;
26     case OracleDB: test = info.isOracleDB_CrossJoin();
27       break;
28     default: test = false;
29       break;
30     }
31   System.out.println(test);
32
33   return false;
34         }
35 }
```

Abbildung 6.2: Erweiterung der Metadatenabfrage des JDBC-Treibers

Alle weiteren Supportfunktionen und sind auf der beiliegenden CD zu finden.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Anfragekapazität von sieben ausgewählten Datenbankmanagementsystemen zu untersuchen, mit dem Ziel, eine Übersicht über die Anfragemöglichkeiten dieser Systeme anzufertigen. Diese Übersicht soll später einmal dabei helfen, Anfragen an verschiedene Systemebenen von intelligenten Umgebungen zu verteilen.
Dazu wurde zunächst ein Überblick gegeben über die für diese Arbeit wichtigen SQL-Anweisungen und auch über zwei ausgewählte Verfahren zur Entscheidungsunterstützung. Anschließend wurden alle DBMS in Hinblick nach ihren Datenabfrageoperationen hin untersucht. Dabei wurden die DBMS eingeordnet nach kleinen Systemen wie TinyDB und Berkeley DB, nach mittleren Systemen wie SQLite und MySQL und nach großen Systemen wie PostgreSQL, Oracle DB und DB2. In Abhängigkeit von welcher Gruppe die einzelnen DBMS Systeme sind, wurden andere Anforderungen an das System gestellt. Als erstes wurden die kleinen Systeme untersucht, von diesen wurde erwartet, dass sie Selektion und Projektion auf einer Datenmenge durchführen können. TinyDB hat diese Anforderungen nur zum Teil erfüllt, zwar kann die Selektion der Daten ausreichend ausgeführt werden, jedoch ist eine Projektion mit dem Standardsystemumfang nicht möglich. Das zweite kleine System Berkeley DB hat die Besonderheit, dass es unterschiedliche Speicherstrukturen unterstützt und dass der Zugriff auf die gespeicherten Daten mit unterschiedlichen Zugriffsmethoden geschehen kann. Beide diese Besonderheiten haben Einfluss auf die Anfragekapazität und Anfragebearbeitungszeit. Um den Vergleich mit anderen Systemen zu ermöglichen, wurde entschieden die Berkeley DB SQL-API zu verwenden. Damit wird Berkeley DB zu einen relationalen DBMS mit der gleichen Anfragekapazität wie SQLite, welche alle Anforderungen an kleine System erfüllt.
Anschließend wurden die mittleren Systeme betrachtet. An diese wurden, zusätzlich zu den Anforderungen an den kleinen Systemen, die Anforderung gestellt, dass die als einfachen definierten Aggregatfunktionen möglich sind. Weiterhin sollten JOIN, GROUP BY, ORDER BY sowie die Window-Funktion einsetzbar sein. Hier wurde zunächst MySQL untersucht; dieses erfüllt fast alle geforderten Anforderungen außer der Window-Funktion und der WITH-Klausel. Ohne die WITH-Klausel ist MySQL nicht in der Lage, rekursive Anfragen zu stellen. Auch bei dem zweiten mittleren System SQLite sieht es nicht anders aus, es unterstützt die sowohl die einfachen Aggregatfunktionen als auch JOIN, GROUP BY und ORDER BY jedoch nicht die Window-Funktion. Allgemein konnte festgestellt werden, dass die Window-Funktion erst ab den großen Systemen unterstützt wird.
Die großen Systeme sollen zum einen in der Lage sein, multidimensional Datenmodelle abzubilden als auch mit Hilfe der definierten komplexen Aggregatfunktionen Analyseoperationen auszuführen. Um die Analyse auf großen Datenmengen zu vereinfachen, soll es möglich sein, die GROUP BY Erweiterungen ROLLUP, CUBE und GROUPING SETS anzuwenden. Bei den großen Systemen gab es keine Überraschungen, sowohl bei den Marktführern IBM und Oracle als auch bei den Entwicklern von PostgreSQL. Sie alle erfüllten die gestellten Anforderungen.

Nachdem alle Systeme betrachtet worden sind, ist eine Übersicht in Form einer Tabelle erstellt worden. Für jeweils drei Systeme, eins aus jeder Gruppe wurden einzelne Anfragestrukturen getestet.
In Rahmen dieser Arbeit wurde eine Übersicht der Anfragekapazität verschiedener DBMS erstellt. Diese, so wird gehofft, soll bei zukünftigen Projekten, die mit einer Aufteilung von Anfragen an unterschiedliche Systeme, helfen. Die zuvor festgelegte Aufteilung der Systeme hat sich nur bedingt bewährt, so kann das kleine DBMS Berkeley DB mit seiner SQL-API durchaus bei dem mittelgroßen Systemen eingetragen werden. Dabei muss jedoch immer beachtet werden, dass bei allen System nur die Anfragekapazität untersucht wurde, welche nur ein Teil des Systemumfangs eines DBMS ist. Um diese Übersicht zu erweitern ist es erforderlich, dass weitere Datenbankmanagementsysteme untersucht werden und, falls erforderlich, neue Anfrageoperationen mit hinein zu nehmen.

Literaturverzeichnis

[Cel12] Joe Celko. Matrix math in SQL, September 2012. (auf beiliegender CD).

[CU97] Surajit Chaudhuri and Dayal Umeshwar. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1), März 1997.

[DB16] Oracle Berkeley DB. *Programmer's Reference Guide*. Oracle, 12c release 1 edition, März 2016.

[GH16] Hannes Grunert and Andreas Heuer. Datenschutz im paradise. *Datenbank-Spektrum*, 2016.

[Gro] The PostgreSQL Global Development Group. Postgresql 9.5.4 documentation select html. https://www.postgresql.org/docs/current/static/sql-select.html. (auf beiliegender CD).

[HMFH04] Wei Hong, Samuel R. Madden, Michael J. Franklin, and Joseph M. Hellerstein. Tinydb: An acquisitional query processing system for sensor networks. Technical report, Massachusetts Institute of Technology, Intel Research Berkeley, 2004.

[IBMa] IBM. IBM Knowledge Center - data types. Stand 11. September 2016 (auf beiliegender CD).

[IBMb] IBM. IBM Knowledge Center - groupbyclause. Zugriff am 30.08.2016 (auf beliegender CD).

[IBMc] IBM. IBM Knowledge Center - user-defined types. Stand 11. September 2016 (auf beiliegender CD).

[IBMd] IBM. IBM Knowledge Center - using recursive queries. Stand 11. September 2016 (auf beiliegender CD).

[Kle12] Meike Klettke. Anfragen, Anfrageverarbeitung, Optimierung teil 1. Fakultät für Informatik und Elektrotechnik, 2012. (auf beiliegender CD).

[Lex12] Datenbanken Online Lexikon. Join, 2012. Zugriff am 28.5.2016 (auf beiliegender CD).

[Mel11] Jim Melton. Information technology database languages sql. Technical report, ISO/IEC, 2011.

[MyS16] *MySQL 5.7 Reference Manual*, 5.7 edition, Februar 2016.

[Noa12] Shlomi Noach. SQL: selecting top n records per group. *code.openark.org*, 2012. (auf beiliegender CD).

[Ora14a] Oracle. *OLAP Expression Syntax Reference*. Oracle, E23382-06, 12c release 1 (12.1) edition, Juni 2014. (auf beiliegender CD).

[Ora14b] Oracle. *Oracle Database Data Warehousing Guide*, 12c edition, November 2014. E41670-08 (auf beiliegender CD).

[Ora16] Oracle. *Oracle Database SQL Language Reference*, 12c edition, Januar 2016. E41329-20 (auf beiliegender CD).

[Pos] The PostgreSQL Global Development Group. *PostgreSQL Documentation*, 9.5.2 edition.

[Sch10] Rupert Schneeberger. Rekursive Anfragen in SQL. *http://www.infosys.tuwien.ac.at*, 2010. (auf beiliegender CD).

[Sie16] Markus Siemens. *TinyDB Documentation*, release 3.1.3 edition, April 2016.

[SSH12] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken Konzepte und Sprachen*. Verlagsgruppe Hüthig Jehle Rehm GmbH, Mai 2012.

[ST16a] Das SQLite-Team. Sqlite documents. https://www.sqlite.org/docs.html, August 2016. Release 3.14.1 (auf beiliegender CD).

[ST16b] Das SQLite-Team. Sqlite documents. https://www.sqlite.org/datatype3.html, August 2016. Release 3.14.1 (auf beiliegender CD).

[TB10] Erich Schubert Thomas Bernecker, Tobias Emrich. Objektrelationale Datenbanken, 2010. LMU München.

[Wika] Wikibooks. Einführung in SQL: Ausführliche SELECT-Struktur. abgerufen am 11. September 2016 `https://de.wikibooks.org/w/index.php?title=Einf%C3%BChrung_in_SQL:_Ausf%C3%BChrliche_SELECT-Struktur&oldid=779882` (auf beiliegender CD).

[Wikb] Wikipedia. Berkeley db. Stand 5. September 2016 `https://de.wikipedia.org/w/index.php?title=Berkeley_DB&oldid=152044351` (auf beiliegender CD).

[Wikc] Wikipedia. Objektrelationale abbildung. Stand 23. August 2016 `https://de.wikipedia.org/w/index.php?title=Objektrelationale_Abbildung&oldid=154903021` (auf beiliegender CD).

[Wik16a] Wikipedia. Db2, 2016. Stand 11. September 2016 `https://de.wikipedia.org/wiki/DB2` (auf beiliegender CD).

[Wik16b] Wikipedia. Mysql, 2016. Stand 11. September 2016 `https://de.wikipedia.org/wiki/MySQL` (auf beiliegender CD).

[Wik16c] Wikipedia. Online analytical processing, 2016. Stand 21. August 2016 `https://de.wikipedia.org/w/index.php?title=Online_Analytical_Processing&oldid=156692782` (auf beiliegender CD).