ISA
Institut für
SoftwareArchitektur

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Sören Gutzeit

Relationale Algebra mit Clojure

4. Februar 2016

Technischer Bericht Nr. 6
des Instituts für SoftwareArchitektur

Inhaltsverzeichnis

1 Einführung

In diesem Kapitel wird das Thema des Entwicklungsprojektes, sowie der Aufbau dieses Berichts erklärt.

Das Entwicklungsprojekt befasste sich mit der Planung und Entwicklung von verschiedenen, alternativen Implementierungen der relationalen Algebra. Das aktuelle Ergebnis des Projektes ist im Git-Repository *more.relational*[10] auf GitHub[9] frei verfügbar.

Eine Erläuterung der wesentlichen Eigenschaften und Merkmale, welche für die nachfolgenden Implementierungen wichtig sind, wird im Unterkapitel 1.1 aufgeführt.

Die Implementierungen werden ausschließlich in der Programmiersprache Clojure entwickelt. Begründungen hierzu finden sich im Unterkapitel 1.1.1.

Als Vorlagen für die alternativen Implementierungen dienen drei verschiedene Ansätze aus verschiedenen Veröffentlichungen:

In Kapitel 2 wird eine zeilenorientierte Implementierung auf Basis von Clojure-HashMaps und der Bibliothek *clojure/set* beschrieben. Als Vorlagen hierfür wurden *core.relational* von Markus Bader und *Dataset* von *Incanter* verwendet.

Kapitel 3 beschreibt eine spaltenorientierte Implementierung durch Fragmentierung von mehrdimensionalen Relationen in binäre Relationen. Hierfür diente die *Binary Association Table* Algebra von Martin Kersten als Vorlage.

In Kapitel 4 wird eine spaltenorientierte Implementierung durch Trennen von Struktur und Werten der Relation aufgezeigt. Hierbei wurde das *Transrelationale Model* von Chris J. Date als Vorlage verwendet.

1.1 Einflüsse der relationalen Algebra

Für die nachfolgenden Implementierungen sind Richtlinien zu beachten. Diese gelten teils für die Eigenschaften einer Relation und teilweise für die Operationen beziehungsweise die Benutzerschnittstellen.

Eine Relation ist im Sinne der mathematischen Definition und nicht als Datenbanktabelle zu sehen. In Datenbankmanagementsystemen ist es gängig, dass Ergebnismengen von Operationen Duplikate beinhalten können. Die Datenhaltung in Relationen ist als Menge zu betrachten. Das bedeutet, ein Tupel ist in der Relation, oder nicht. Duplikate existieren nicht in einer Relation.[5, Kapitel 2.1]

Außerdem gilt in einer Relation keine vorgeschriebene Sortierung der Tupel. Die Reihenfolge obliegt daher der Implementierung des Datenschemata oder der Art des Erstellens (z.B. die erzeugende Operation).

1.1.1 Verwendung von Clojure

In der Definition einer Relation nach Chris Date wird zwischen Wert und Variable unterschieden.[4] Er erklärt den Unterschied anhand eines Vergleichs zu einer Integer-Variable in einer Programmiersprache.

```
1 int qty = 5;
```

Listing 1.1: Pseudocode für die Deklaration einer Integer-Variable

Der Pseudocode in Listing 1.1 deklariert *qty*. *qty* wird als Integer-*Variable* bezeichnet, und nicht als Integer-*Wert*. Der Wert in dieser Deklaration lautet *5*, kann jedoch innerhalb der Variable jederzeit durch einen anderen Wert ausgetauscht werden.

Relvar sind Variablen, die einen Relation-Wert als Wert beinhalten können. [6, Seite 19-20] Im weiteren Verlauf dieses Berichts wird für den Begriff *ein Relation-Wert* vereinfacht *eine Relation* verwendet. Mit anderen Worten ist jede Relation als Wert zu betrachten.

Weiter werden Werte nicht durch Operationen verändert. Eine mathematische Addition zweier Werte, beziehungsweise Zahlen, verändert diese Werte nicht, sondern ergibt einen anderen Wert.

Durch Operationen der relationalen Algebra werden somit ebenfalls keine bestehenden Relationen verändert. Es wird durch die gegebenen Parameter mit Hilfe der Operationsdefinition eine neue Relation erstellt.

Für die Implementierung von relationaler Algebra ist somit eine Programmiersprache vorteilhaft, welche eine Datenstruktur unterstützt, die unveränderliche Daten vorsieht. Aufgrund dieser Datenstruktur sollten Funktionen frei von Seiteneffekten sein. Daten wären somit als Wert zu sehen.

Die funktionale Programmiersprache Clojure ist im Stil eines modernen Lisp-Dialekts. In Clojure wird eine persistente Datenstruktur verwendet. Datentypen innerhalb von Clojure enthalten somit unveränderliche Werte, welche komplett frei von Seiteneffekten sind. [11, Abschnitt „Immutable Data Structures“]

Dadurch bietet sich diese Programmiersprache für die Entwicklung sehr an.

1.1.2 Relationale Algebra

Die relationale Algebra definiert Operationen, die sich auf eine Menge Relationen anwenden lassen.[4, Kapitel 7] Die Implementierungen sollen sich an der Menge von gängigen Operationen orientieren:

- Mengenoperationen für kompatible Relationen: Vereinigung, Schnittmenge, Differenz
- Umbenennung von Attributen
- Projektion
- Selektion / Restriktion
- mindestens eine Verbundsoperationen: bevorzugterweise natürlicher oder Gleichverbund
- Gruppierung

- Aggregatsfunktionen: Anzahl, Summierung, Minimum, Maximum und ähnliches

Diese Menge stammt überwiegend aus der mathematischen Definition der Algebra. Grundlegend definiert sich jede Operation dadurch, dass die Ergebnismenge eine neue Relation erzeugt. Ausnahmen dabei sind die Aggregatsfunktionen. Diese fassen oft die Relation anhand ihrer Beschaffenheit auf einen Wert zusammen.

Einige Operationen, beispielsweise das kartesische Produkt, wurden innerhalb der Orientierung weggelassen. Gründe dafür sind zum einen die Eingrenzung des Aufwandes und zum anderen die alternativen Abbildungsmöglichkeiten der fehlenden Operationen durch bereits bestehende.

1.2 Variablen

Die Relationen sollen in den Implementierungen in *relvar* verwendet werden können. Für diese Verwendung sollen die Implementierungen Unterstützung in Form von Benutzerschnittstellen liefern. Über diese Funktionen sollen *relvar* definiert werden, sowie der Inhalt dieser Variablen zugewiesen und verändert werden können.

Außerdem bedürfen diese Variablen auch Unterstützung für das Lesen und Schreiben der Relationen in Dateien. Da die Verwendung der relationalen Algebra Hauptspeicherorientiert ist, wird diese Funktionalität benötigt, um Datenbestände langfristig verwenden zu können.

Zusätzlich ist das Zusammenfassen mehrerer Relationsvariablen zu einer Datenbank eine weitere wünschenswerte Funktion. Somit können sich auch komplexere Datenstrukturen in den Relationen abbilden lassen.

1.3 Aufbau der Implementierung

Jede Implementierung befinden sich in einem eigenen Clojure-Namenspace. Für die Implementierung aus Kapitel 2 wird der Namespace *relation.hashRel* zur Verfügung gestellt. Die Umsetzungen aus Kapitel 3 und 4 befinden sich in den Namensräumen *relation.bat* und *relation.transrelational*. Jeder Namespace umfasst die Funktionalität zum Erstellen der Relationen, sowie für das Manipulieren dieser Relationen und das Erstellen und Manipulieren der *relvar*.

Die Bibliothek *Potemkin*[20] wird dabei verwendet, dass nach außen hin nur ein Namespace gebraucht wird. Die Implementierungen selbst sind in Unter-Namespaces aufgeteilt.

2 Geradlinige Implementierung auf Basis von Clojure Maps

In diesem Kapitel wird die Entwicklung an einer Implementierung auf Basis von Clojure Maps dargestellt. In den folgenden Unterkapiteln werden Informationen zur Gestaltung dieser Implementierung, sowie besondere Details zum Code und zu Mechanismen geliefert.

Diese Implementierung der relationalen Algebra basiert grundlegend auf dem Abbilden der Relationen tupelorientiert als Clojure-HashMaps.

Der Code wurde nach den Schnittstellen von *core.relational* entworfen. Für Ideen der Neuentwicklung wirken *incanter.core*, sowie die *Transducer* aus Clojure Version 1.7. mit.

2.1 Basis der Idee

Zu Beginn des Projektes bestand die Aufgabe darin, die Bibliothek *core.relational* von Markus Bader[1] mit alternativen Implementierungen zu vergleichen. Als Alternative diente der Namespace *incanter.core* der Clojure-Bibliothek Incanter[17].

2.1.1 core.relational

core.relational sieht eine Trennung der Daten zwei Teile: einen Tabellenkopf und eine Menge von Daten.

Der Kopf ist ein Vektor (*vec*) und repräsentiert die Reihenfolge der Attribute. Diese Reihenfolge ist bindend für die folgende Datenmenge.

Die Daten sind eine Menge(*set*) von Vektoren. Durch das Verwenden von dem Clojure-Datentyp *set* wird hierbei gewährleistet, dass keine doppelten Datensätze existieren. Die Vektoren sind Datensätze der Relation, deren Attribute anhand der Reihenfolge der Kopfzeile geordnet sind (siehe [1, S. 18-19]). In Listing 2.1 ist ein Beispiel für eine zweizeilige Relation nach Bader zu sehen.

```
1 #rel [:id :name :status :city]
2 #{["S1" "Smith" 20 "London"]
3   ["S2" "Jones" 10 "Paris"]}
```

Listing 2.1: Darstellung einer Relation in core.relational

Für Funktionen, bei denen die verwendende Applikation auf Ebene der Datensätze operiert (beispielsweise *restrict* oder *project+*), werden die Prädikate beziehungsweise Funktionen als *relfn* dargestellt. Zum Zeitpunkt dieser Arbeit ist die *relfn* lediglich ein Makro zum Aufgreifen einer Funktion sowohl als Wert als auch als unausgewertete Liste. Dies soll für künftige Erweiterungen der Bibliothek als Möglichkeit dienen, die benutzerdefinierten *relfn* zu optimieren.

In *core.relational* wurden das Konzept der *relvar* durch eine gleichnamige Darstellung von Clojure-Referenzen implementiert. Die *relvar* bieten innerhalb der verwendenden Applikation mehrere Vorteile. Zum einen können global definierte *relvar* über spezielle Funktionen manipuliert werden. Diese Manipulationen, wie etwa das Einfügen oder Löschen von Datensätzen, gelten ebenfalls für die ganze Applikation.

Relvar können Constraints beinhalten. Darüber lassen sich dauerhaft Regeln für die Relationen formulieren. Regeln können Bedingungen über die Inhalte von einzelner Datensätze sein, oder über den Zusammenhang unterschiedlicher Datensätze in verschiedenen Relationen (zum Beispiel Fremdschlüssel). Das Konzept der Anwendung von Contraints auf relvar wurde bereits von Chris Date aufgegriffen.[4, Kapitel 9.4]

2.1.2 Dataset

In der *Dataset*-Implementierung von *Incanter* wurden die Daten aus einer Sammlung von *Maps* aus dem Clojure-Standarddatentypen abgebildet. Eine beispielhafte Abbildung von Daten ist in Listing 2.2 zu sehen.

```
#dataset [:id :name :status :city]
  [{:id "S1" :name "Smith" :status 20 :city "London"}
   {:id "S2" :name "Jones" :status 10 :city "Paris"}]
```

Listing 2.2: Beispielhafte Darstellung einer Relation in incanter.core

Damit unterscheidet sich die Incanter-Implementierung grundlegend in zwei Punkten von *core.relational*.

Zum einen wird die Menge der Datensätze in *core.relational* als tatsächliche Menge (*set*) abgebildet. Incanter hingegen verwendet Listen (*seqences*) und Vektoren zum Sammeln der Datensätze innerhalb einer Relation. Dies resultiert darin, dass in Incanters *Datasets* Duplikate vorkommen können. Dies ist jedoch für das Ziel dieser Arbeit nicht vorgesehen.

Zum anderen werden die einzelnen Datensätze in Incanter als HashMaps abgebildet. In der Bader-Implementierung wird der Zugriff auf einzelne Attribute der Datenzeilen auf Vektoren abgewickelt. Um gezielt ein Element aufzugreifen, wird in der Kopfzeile über die Funktion *.indexOf* die Position des Attributes ermittelt. Mit der ermittelten Position wird im Vektor bzw. in den Vektoren über die *nth*-Funktion auf das Element zugegriffen.

Das Konzept der *relvar* ist in Incanter nicht vorgesehen.

2.1.3 Vergleich

In Listing 2.3 ist zu sehen, dass sich die Effizienz der Zugriffsarten nicht grundlegend unterscheidet. Auch bei Tests mit größeren Datenmengen zeigte sich in den Zugriffen auf vereinzelte Elemente kein Unterschied. Die Bader-Implementierung wird dann weniger effizient, wenn sich die Attribute im Laufe größerer Operationen unterscheiden oder verschieben. Bei mehrfachen Neuberechnungen der Indizes erhöht sich die Zugriffszeit schon bei kleinen Datenmengen (siehe Listing 2.4).

```
(def head [:sno :sname :status :scity])
(def rows-as-vector  [["S1" "Smith" 20 "London"]
                      ["S2" "Jones" 10 "Paris"]
                      ["S3" "Blake" 30 "Paris"]
                      ["S4" "Clark" 10 "London"]
                      ["S5" "Adams" 30 "Athen"]])

(time (let [index (.indexOf head :status)]
        (doall (map #(nth % index) rows-as-vector)))) ; 0.112416 msecs
```

```
10
11 (def rows-as-map  [{:sno "S1" :sname "Smith" :status 20 :scity "London"}
12                   {:sno "S2" :sname "Jones" :status 10 :scity "Paris"}
13                   {:sno "S3" :sname "Blake" :status 30 :scity "Paris"}
14                   {:sno "S4" :sname "Clark" :status 10 :scity "London"}
15                   {:sno "S5" :sname "Adams" :status 30 :scity "Athen"}])
16
17 (time (doall (map #(get % :status) rows-as-map))) ; 0.10627 msecs
```

Listing 2.3: Vergleich von Zugriffen auf Elementen in Vektoren und Hash-Maps)

```
1 (def heads-and-rows  [[[:sno :sname :status :scity] ["S1" "Smith" 20 "London"]]
2                       [[:sname :status :scity :sno] ["Jones" 10 "Paris" "S2"]]
3                       [[:status :scity :sno :sname] [30 "Paris" "S3" "Blake"]]
4                       [[:scity :sno :sname :status] ["London" "S4" "Clark" 10]]
5                       [[:sno :sname :status :scity] ["S5" "Adams" 30 "Athen"]]])
6
7 (time (doall (map (fn[[head row]] (nth row (.indexOf head :status)))
8           heads-and-rows))) ; 0.186422 msecs
```

Listing 2.4: Zugriffe auf Elementen in Vektoren mit wechselnder Position)

Aufgrund der Abbildung von Datensätzen in HashMaps greifen Operationen von Incanter auf in Clojure abgebildete Mengenoperationen zurück. Dies sorgt dafür, dass äquivalente Funktionen zu Baders Implementierung deutlich effizienter sind.

Für die Mengenoperationen wurde die Clojure Bibliothek *clojure.set* verwendet. Diese liefert Funktionen wie *union*, *intersection* oder *difference*, um *Hash-Maps* miteinander zu kombinieren und neue Mengen zu erzeugen. Beispiele für die genannten Funktionen sind in Listing 2.5 zu sehen.

Andere Operationen wurden dem Datentyp entsprechend implementiert.

```
1 (use 'clojure.set)
2 (union #{1 2} #{2 3}) ; #{1 2 3}
3 (intersection #{1 2} #{2 3}) ; #{2}
4 (difference #{1 2} #{2 3}) ; #{1}
```

Listing 2.5: clojure.set Beispieloperationen

Ein Vergleich der Verbund-Operationen (*join*) beider Implementierungen zeigte deutliche Unterschiede, wobei Incanters Implementierung deutlich effizientere Zeiten aufweist. Dies beruht ebenfalls auf die Verwendung von HashMaps in Kombination mit Mengen-Operationen aus *clojure.set*. Die Funktion *clojure.set/index* gruppiert eine Sammlung von HashMaps anhand einer Liste von übergebenen Schlüsseln (siehe Listing 2.15). Als Resultat liefert die Funkion eine neue HashMap mit den übergebenen Schlüsseln, inklusive der vorhandenen Werte, als Schlüssel. Als Wert haben die Werte-Paare alle Datensätze mit dem gleichen Wert des Schlüssels als Menge zusammengefasst. Die Datensätze der Werte haben die Schlüssel-Paare nicht weiter in der HashMap. Incanter verwendet in ihrer Implementierung diese Mechanik, um so Tupel anhand ihrer gleichen Attribute einander zuzuordnen. Der eigentliche Vergleich der Attributswerte ist damit ein Aufruf in der HashMap anhand eines Schlüsselwertes.

```
1 (use '[clojure.set :only (index)])
2
3 (def weights #{ {:name 'betsy :weight 1000}
4   {:name 'jake :weight 756}
5   {:name 'shyq :weight 1000} })
6
7 (index weights [:weight])
8 ; {{:weight 1000} #{{:name shyq, :weight 1000} {:name betsy, :weight 1000}},
       {:weight 756} #{{:name jake, :weight 756}}}
```

Listing 2.6: Beispiel clojure.set/index (Vgl. [16])

In *core.relational* wurde der die Verbund-Operation durch den bekannteren *Nested loop* Algorithmus implementiert. Innerhalb dieses Algorithmus werden zwei ineinander geschachtelte Schleifen verwendet. Dabei geht die äußere Schleife durch die Tupel der einen Relation, die innere Schleife durch die der anderen. Im Körper der inneren Schleife werden alle Tupel beider Relationen miteinander verglichen.

Um die Effektivität beider Varianten zu vergleichen, wurden anhand von Beispieldaten beide Implementierungen miteinander verglichen. Tests mit Datenmengen im vier- bis fünf-stelligen Bereich zeigten, dass die Variante von Incanter, als ohne *Nested loop join*, deutlich weniger Laufzeit kostet (siehe Listing 2.7).

```
1 (def employees-data ...) ; 10000 rows
2 (def salaries-data ...) ; 5000 rows
3
4 (def employees-rel (rel employees-data))
5 (def salaries-rel (rel salaries-data))
6
7 (time (join employees-rel salaries-rel)) ; 11981.83025 msecs
8
9
10 (def employees-dataset (dataset employees-data))
11 (def salaries-dataset (dataset salaries-data))
12
13 (time  ($join [:emp_no] employees-dataset salaries-dataset)) ; 36.575089 msecs
```

Listing 2.7: Vergleich der Effizienz von Verbünden in core.relational und Incanter

2.2 Umsetzung

Beide genannten Implementierungen haben ihre Vorteile, beziehungsweise ihre Alleinstellungsmerkmale. Für die neue Zeilen-basierte Implementierung der relationalen Algebra werden die Vorteile beider Implementierungen verwendet.

Die Verwendung von HashMaps und die Benutzung von *clojure.set*-Funktionen zum Abbilden der relationalen Operationen aus *Dataset* sind deutlich effizienter als die Implementierungen von *core.relational*. Darum wurde dies für die Darstellung von einzelnen Datensätze gewählt.

Die Tupel der Relation werden in eine *set* gespeichert. Diese Entscheidung wurde getroffen, um die Clojure-Mechaniken auszunutzen, dass eine Relation keine Duplikate beinhalten kann.

Schnittstellen, sowie die *relvar*-Mechaniken, aus *core.relational* werden für die neue Implementierung übernommen. Somit soll die Neuimplementierung von außen kaum Unterschiede aufweisen.

2.2.1 Darstellung der Relation

Für die Darstellung wird mit *deftype* ein neuer Datentyp für unsere Relationen definiert. *deftype* wurde gewählt, um dem Datentyp eigens definierte Funktionalität für Vergleiche, Zähloperationen und Ähnliches zu geben. Auch das Umwandeln der Relationen in Listen über *clojure.lang.Seqable* zählt zu den Funktionen.

Die Daten einer Relation teilen sich in zwei Bestandteile:

- Die Kopfzeile, beziehungsweise das Schema der Relation als *Vector* von Attributenamen (siehe Listing 2.8 Zeile 8).
- Der Körper, beziehungsweise die Datensätze werden jeweils als *HashMaps* mit den Attributenamen als Schlüssel abgebildet. Zusammengefasst werden die Datensätze in einem *Set* (siehe Listing 2.8 Zeile 8).

```
1 (rel [:sno :sname :status :scity]
2   [["S1" "Smith" 20 "London"]
3    ["S2" "Jones" 10 "Paris"]
4    ["S3" "Blake" 30 "Paris"]
5    ["S4" "Clark" 10 "London"]
6    ["S5" "Adams" 30 "Athen"]])
7
8 ; #rel [:sno :sname :status :scity]
      #{{:sno "S2", :sname "Jones", :status 10, :scity "Paris"}
      {:sno "S4", :sname "Clark", :status 10, :scity "London"}
      {:sno "S3", :sname "Blake", :status 30, :scity "Paris"}
      {:sno "S5", :sname "Adams", :status 30, :scity "Athen"}
      {:sno "S1", :sname "Smith", :status 20, :scity "London"}}
```

Listing 2.8: Darstellung der Daten in neuer Implementierung

Während des Erstellens der Relation wird die Kopfzeile anhand des ersten Datensatzes erstellt. Der Kopf dient als Zwischenspeicher des Schemas der Relation. Das Schema soll auch dann in der Relation bestehen, wenn sie eine leere Menge ist. Daher kann für die Menge der Attribute nicht immer auf die Tupel Bezug genommen werden.

Zudem bietet die Kopfzeile als Vektor einen weiteren Nebeneffekt. Im Laufe der Verwendungszeit können Operationen die Reihenfolge der Attribute in einzelnen, beziehungsweise jedem Datensatz ändern. Um weiterhin die vom Benutzer zuletzt definierte Reihenfolge bewahren zu können, wird sie in der Kopfzeile gespeichert.

Außerdem kann für Operationen die Kopfzeile als Abbild aller Attribute in der Relation verwendet werden. Für Vergleichsoperationen müssen die vorhandenen Attribute nicht neu gesucht werden.

Die Funktion *rel* wird zur Erstellung von Relationen verwendet. Als Parameter können sowohl eine Liste von Attributenamen und eine Sammlung von Datensätzen (als Vektor oder Liste) (siehe Listing 2.8 Zeile 1 bis 6), als auch eine Sammlung von HashMap übergeben werden (siehe Listing 2.9).

```
1 (rel [{:sno "S1" :sname "Smith" :status 20 :scity "London"}
2       {:sno "S2" :sname "Jones" :status 10 :scity "Paris"}
3       {:sno "S3" :sname "Blake" :status 30 :scity "Paris"}
4       {:sno "S4" :sname "Clark" :status 10 :scity "London"}
5       {:sno "S5" :sname "Adams" :status 30 :scity "Athen"}])
```

Listing 2.9: Aufruf von rel mit Hash-Map als Parameter

2.2.2 Operationen

In Tabelle 2.1 werden die Operationen aufgelistet, welche für die Neuimplementierung entwickelt wurden. Orientiert wurde sich an den Schnittstellen von *core.relational*. Durch die Neugestaltung des Datentypen könnten die Implementierungen nicht übernommen werden, könnten jedoch als Vorlage verwendet werden.

Ein Makro *relfn* wird für Benutzer-definierte Funktionen vorausgesetzt. Diese finden ihren Gebrauch in den Operationen *restrict*, *project*, *project+* und *summarize* (siehe Tabelle 2.1). *relfn* dienen zum Aufbewahren von nicht ausgewerteten Clojure-Funktionen in dieser Implementierung. In möglichen Weiterentwicklungen dieser Implementierung können *relfn* für Optimierungen der Operationen verwendet werden. Derzeit wird lediglich der Code der Funktion als Liste aufbewahrt (siehe Listing 2.10).

```
1 (defmacro relfn
2   [args body]
3   (with-meta (list 'fn args body)
4         {:body (list 'quote body)}))
```

Listing 2.10: Implementierung: relfn

Operation	Beschreibung
(rename r smap)	Benennt in Relation r Attributename k in v um für alle k, v in *smap*.
(rename match replace)*	Benennt in Relation r in allen Attributenamen alle Sub-Strings, welche auf den regulären Ausdruck *match* passten, in *replace* um.
(restrict r pred)	Filtert Relation r auf alle Tupel, für die das Prädikat *pred* in Form einer *relfn* wahr ist.
(project r attrs)	Reduziert die Attribut-Menge in Relation r auf die Attribute in der Menge *attrs*. *attrs* kann auch in Form einer Map übergeben werden, wobei das Resultat aus Attributen k mit den Wert v für alle k, v in *attrs* besteht. v kann dabei als Attributename oder als *relfn* übergeben werden.
(project+ r attrs)	Ergänzt die Attribut-Menge in Relation r um Attribute k mit dem Resultat der *relvar* v für alle k, v in *attrs*.
(project- r attrs)	Reduziert die Attribut-Menge in Relation r um die Attribute in der Sammlung *attrs*.
(join r1 r2)	Bildet einen natürlichen Verbund als Relation aus den Relationen *r1* und *r2*. Als Schlüsselattribute dienen alle gleichnamigen Attribute in beiden Relationen.
(compose r1 r2)	Bildet einen natürlichen Verbund als Relation aus den Relationen *r1* und *r2*. Als Schlüsselattribute dienen alle gleichnamigen Attribute in beiden Relationen. Diese sind in der Ergebnisrelation nicht vorhanden.
(union r1 r2)	Bildet eine Vereinsmenge aus den Relationen *r1* und *r2*. Die Attribute von *r1* und *r2* müssen gleichnamig ein.
(intersect r1 r2)	Bildet die Schnittmenge aus den Relationen *r1* und *r2*. Die Attribute von *r1* und *r2* müssen gleichnamig ein.
(difference r1 r2)	Bildet die Differenzmenge aus den Relationen *r1* und *r2*. Die Attribute von *r1* und *r2* müssen gleichnamig ein.
(divide r1 r2)	Bildet die Divisionsmenge aus den Relationen *r1* und *r2*.
(tclose r)	Bildet für die binäre Relation r eine neue Relation mit zusätzlich allen transitiven Abschlüssen für die binären Tupel in r.
(group r gmap)	Gruppiert die Attribute der Menge v in der Relation r als Attribut k für alle k, v in *gmap*. Gruppierte Attribute werden als Unterrelation dargestellt.
(ungroup r attrs)	Wandelt in Relation r die Gruppierung in Attribute a in die Attribute der Gruppe um für alle a in Sammlung *attrs*.
(wrap r wmap)	Fügt in Relation r alle Attribute der Menge v zu einer *Clojure-Map* im Attribut k zusammen für alle k, v in *wmap*.
(unwrap r attrs)	Wandelt in Relation r die Clojure-Map a in die enthaltende Attribute um für alle a in *attrs*.
(summarize r group smap)	Gruppiert Relation r anhand der Attribute in Sammlung *group* und ergänzt sie anhand des Attributes k mit dem Resultat der *relfn* v für alle k, v in *smap*.

Tabelle 2.1: Übersicht der Operationen in der Neuimplementierung

2.2.2.1 Verwendung von clojure.set

In dieser Implementierung bietet sich aufgrund der Datenstruktur die Verwendung von *clojure.set* an. Dieser *Namespace* bietet hochperformante Mengenoperation für *Sets* (siehe [15]). Innerhalb vom *clojure.set* wird auch in verschiedenen Operationen (beispielsweise *index*) der Begriff *xrel* für Parameter verwendet. Dies bezeichnet eine Menge von HashMaps innerhalb eines Sets. Dies ist identisch zu dem Körper des neuen Datentyps und kann deswegen effizient verwendet werden.

Für *union* in Bezug auf die relationale Algebra wurde beispielsweise die *clojure.set*-Variante dieser Operation verwendet (siehe Listing 2.11). Zur Verarbeitung des Relation-Körpers ist diese Operation sehr effizient.

```
1 (defn union [relation1 relation2]
2   (when-not (tools/same-type? relation1 relation2)
3     (throw (IllegalArgumentException. "The two relations have different types.")))
4   (rel (.head relation1) (clojure.set/union (.body relation1) (.body relation2))))
```

Listing 2.11: Implementierung: union

Da die Implementierung in *core.relational* sich den gleichen Bibliotheken bedient, ist ein direkter Vergleich der Neuimplementierung schwierig. Jedoch besteht aufgrund des neuen Datentypen der Vorteil, dass es keine Probleme bei der Reihenfolge der Attribute besteht. In *core.relational* werden Vektoren für das Darstellen von Tupeln verwendet. Da in der *union*-Operation unterschiedliche Reihenfolgen der Attribute der Operanden vorliegen können, muss die Operation gegebenenfalls einen Operanden anpassen (siehe Listing 2.12).

```
1 (union [relation1 relation2]
2   (when-not (same-type? relation1 relation2)
3     (throw (IllegalArgumentException. "The two relations have different types.")))
4
5   (let [rel2-body (if (same-attr-order? relation1 relation2)
6                       ; same order: nothing todo
7                       (.body relation2)
8
```

```
9                     ; different order: sort the second relation like the first one
10                       (set (let [sorter (sort-vec relation1 relation2)]
11                                  (map (fn [tuple]
12                                           (vec (map (fn [pos]
13                                                       (nth tuple pos))
14                                                     sorter)))
15                                       (.body relation2)))))]
16          (rel (.head relation1) (clj-set/union (.body relation1) rel2-body))))
```

Listing 2.12: Implementierung: union in core.relational

Für den Vergleich von HashMaps ist die interne Reihenfolge von Attributen nicht relevant (siehe Listing 2.13). Daher ist in der Neuimplementierung (siehe Listing 2.11) keine Sortierung im Vorfeld notwendig.

```
1 (clojure.set/union #{{:a :b, :c :d, :e :f}}
      #{{:e :f, :c :d, :a :b}}
      #{{:a :b, :e :f, :c :d}}) ; #{{:a :b, :e :f, :c :d}}
```

Listing 2.13: Beispiel für das Verhalten von Hash-Maps in clojure.set/union

Die Operation *union* wurde exemplarisch für ähnliche Operationen erläutert. Für die Operationen *intersection*, *difference* und aufgrund dessen Verwendung *divide* gilt das gleiche Sortierungsproblem.

2.2.2.2 Verbund

Der Verbund wurde der Implementierung von *Incanter* nachempfunden, da diese (auch aufgrund der Verwendung von *clojure.set*) deutlich effizienter im Vergleich zu der in *core.relational* ist.

Augenmerk in der Implementierung liegt auf der Verwendung von *clojure.set/index* auf die kleinere der übergebenen Relationen (siehe Listing 2.14, Zeile 8). Mit *kleinere* ist die Relation gemeint, die die geringere Anzahl an Tupeln beinhaltet.

```
1  (defn join [relation1 relation2]
2    (if (and (seq relation1) (seq relation2))
3        (let [ks (clojure.set/intersection (set (.head relation1))
4                                           (set (.head relation2)))
5              [r s] (if (<= (count relation1) (count relation2))
6                        [relation1 relation2]
7                        [relation2 relation1])
8              idx (clojure.set/index r ks)]
9          (rel (reduce (fn [ret x]
10                        (let [found (idx (select-keys x ks))]
11                           (if found
12                               (reduce #(conj %1 (merge %2 x)) ret found)
13                               ret)))
14                       #{} s)))
15        (if (seq relation1)
16            relation1
17            (if (seq relation2)
18                relation2
19                (rel [] #{})))))
```

Listing 2.14: Implementierung: join

In Listing 2.15 wird exemplarisch die Funktionsweise von *index* aufgeführt. *index* erstellt aus einer *xrel* anhand einer Liste von Attributenamen eine neue HashMap. Diese HashMap hat als Schlüsselwerte alle in der *xrel* existierenden Kombinationen der übergebenen Attribute. Als Werte zu den Schlüsseln ein Set aller Tupel aus der *xrel*, die diese Kombination beinhalten.

```
1  (def sp #{{:pid 1, :aname "Hammer",   :qty 1}
2            {:pid 2, :aname "Hammer",   :qty 1}
3            {:pid 1, :aname "Nagel",    :qty 10}
4            {:pid 3, :aname "Schraube", :qty 1}})
5
6  (clojure.set/index sp [:pid])
7  ; {{:pid 3} #{{:pid 3, :aname "Schraube", :qty 1}},
8  ;  {:pid 1} #{{:pid 1, :aname "Nagel", :qty 10}
9  ;             {:pid 1, :aname "Hammer", :qty 1}},
10 ;  {:pid 2} #{{:pid 2, :aname "Hammer", :qty 1}}}
```

Listing 2.15: Beispiel für die Verwendung von clojure.set/index

Durch die Verwendung von dieser Funktion kann das klassische Verwenden des *Nested loop* Algorithmus zum Zuordnen der Tupeln umgangen werden. Das Resultat von *index* wird auf die jeweils andere Relation zugeordnet. Diese Zuordnung der Schlüsselattribute in der Ergebnis-Map von *index* ist sehr effizient.

Der Vergleich zu der Implementierung von *core.relational*, welche die klassische Variante von Verbünden verwendet, zeigt einen deutlichen Unterschied in der Performanz. Listing 2.16 zeigt, die neue Implementierung, welche grundlegend von Incanter stammt, ist deutlich effizienter.

```
1 (def employees ...) ; 10000 Tupel, fuer das Beispiel Datentyp-unabhaengig
2 (def salaries ...) ; 5000 Tupel, fuer das Beispiel Datentyp-unabhaengig
3
4 (time (core.relational/join employees salaries)) ; 13321.541299 msecs
5 (time (new/join employees salaries)) ; 25.66578 msecs
```

Listing 2.16: Vergleich von join in core.relational und Neuimplementierung

2.2.2.3 Gruppierung

Die Gruppierung ist in dieser Implementierung nach der Vorlage von *core.relational* entwickelt. Hierbei werden in den Ergebnismengen die Untergruppen als Unterrelation in den zugehörigen Tupeln gespeichert. In Listing 2.17 wird anhand einer Beispielrelation (Zeile 1-5) eine Gruppierung (Zeile 7) abgebildet. Die Attribute *:sid*, *:qty* und *:description* wenden unter den Namen *:NameAndQty* gruppiert. Tupel mit gleichen Gruppen und sonstigen Attributen werden zusammengefasst, sodass zwei Tupel mit je zwei Untertupeln (in den Gruppen) entstehen (Zeile 9-17).

```
1  (def rs (rel
2  #{{:sid 2, :description "Hammer", :id 1, :name "Arthur",  :qty 2}
3   {:sid 3, :description "Nail", :id 2, :name "Betty",  :qty 100}
4   {:sid 1, :description "Scrows", :id 1, :name "Arthur",  :qty 200}
5   {:sid 2, :description "Hammer", :id 2, :name "Betty",  :qty 1}}))
6
7  (group rs {:NameAndQty #{:sid :quantity :description}})
8
9  ;#rel [:id :name :NameAndQty]
10 ;#{{:id 1, :name "Arthur",
11 ; :NameAndQty   #rel [:description :qty :sid]
12 ;                #{{:description "Hammer", :qty 2, :sid 2}
13 ;                  {:description "Scrows", :qty 200, :sid 1}}}
14 ;{:id 2, :name "Betty",
15 ; :NameAndQty   #rel [:description :qty :sid]
16 ;                #{{:description "Hammer", :qty 1, :sid 2}
17 ;                  {:description "Nail", :qty 100, :sid 3}}}}
```

Listing 2.17: Beispiel: group

Durch die Operation *ungroup* lassen sich diese Untergruppen wieder auflösen und das ursprüngliche Schema wieder herstellen.

2.2.3 Variablen

Für die Verwendung der Relationen als Variablen (siehe auch [13]) wurden die Schnittstellen von *core.relational* übernommen und teilweise neu implementiert. In Tabelle 2.2 sind die Funktionen und deren Funktionsweise dargestellt.

Operation	**Beschreibung**
(relvar [r] [r cons])	Erstellt anhand einer Relation *r* eine *relvar*. Optional können eine Menge an Constraints *cons* zusätzlich übergeben werden.
(assign! rvar r)	Ersetzt die beinhaltende Relation von *relvar rvar* durch die Relation *r*.
(insert! rvar t)	Fügt der Relation in *relvar rvar* den Datensatz/das Tupel *t* hinzu.
(delete! rvar pred)	Löscht in der Relation in *relvar rvar* alle Datensätze, welche für das Prädikat *pred* wahr sind.
(update! rvar pred a v)	Setzt in der Relation in *relvar rvar* in allen Datensätzen, welche für das Prädikat *pred* wahr sind, das Attribut *a* auf den Wert *v*.
(constraint-reset! rvar cons)	Entfernt alle vorhandenen Constraints in *relvar rvar* und ersetzt sie durch die Menge an übergebenen Constraints *cons*.
(add-constraint! rvar c)	Fügt der Menge an Constraints in *relvar rvar* den Constraint *c* hinzu.
(save-relvar rvar f)	Schreibt den Inhalt (Relation und Constraints) von *relvar rvar* in eine Datei mit dem Pfad *f*.
(load-relvar f)	Liest den Inhalt von einer Datei mit dem Pfad *f* und erstellt eine *relvar* anhand des Inhalts.
(save-db db f)	Schreibt den Inhalt (Relation und Constraints) von *relvar*-Datenbank *db* in eine Datei mit dem Pfad *f*.
(load-db f)	Liest den Inhalt von einer Datei mit dem Pfad *f* und erstellt eine *relvar*-Datenbank anhand des Inhalts.

Tabelle 2.2: Übersicht der Operationen für Referenzen in der Neuimplementierung

3 Binary associated table Algebra

In diesem Kapitel wird die Entwicklung einer Implementierung der *Binary Association Table* (BAT) Algebra in Clojure dargestellt.

Im folgenden Unterkapitel 3.1 werden die Grundlagen dieser Algebra, wie etwa mathematische Darstellung und die erdachte Funktionsweise, erläutert.

In dem Unterkapitel 3.2 wird auf die Umsetzung der Algebra in Clojure eingegangen. Hierbei wird auch auf die Verwendung und Erweiterungsmöglichkeiten eingegangen.

3.1 Basis der Idee

Die zweite relationale Algebra dieser Arbeit unterscheidet sich grundlegend von der in Kapitel 2. Als Vorlage für diese Implementierung dient das Datenbankmanagementsystem *MonetDB* vom *Centrum Wiskunde & Informatica* aus Amsterdam.[8]

Innerhalb deren Datenstruktur werden Relationen vertikal fragmentiert. Vertikale Fragmentierung innerhalb von MonetDB bedeutet, Relationen werden spaltenweise separiert und mithilfe von Objekt-Identifizierer (*OIDs*) zum zugehörigen Tupel zugeordnet.[18]

Operationen werden innerhalb von MonetDB durch die *Monet interpreter language* (*MIL*) von gängigen Zugriffsarten auf die BAT-Algebra abgebildet (siehe Abbildung 3.1). Somit kann von außen eine Relation wie gewohnt gesehen und manipuliert werden. Im Inneren werden Operationen und Anfragen in die BAT-Algebra übersetzt.

3.1.1 BAT Algebra

Die *Binary Association Table* Algebra beschreibt Operationen zwischen oder mit binären Relationen. Als Ergebnisse dieser Operationen wird grundlegend eine neue binäre Relation mit der Ergebnismenge erstellt.

Die Definition dieser Algebra stammt aus einer Veröffentlichung von Martin Kersten in *THE VLDB JOURNAL* von 1970. [2]

Binäre Relationen sind Relationen, deren Schema nur zwei Spalten besitzt. Die Typen der Spalten werden in mathematischer Notation als $bat[T_1, T_2]$ definiert. Als Typ stehen *head* und *tail* laut Definition von Kersten alle Datentypen zur Verfügung. Auch BATs selbst können als Typ für eine Spalte verwendet werden. Als *nil* werden leere Zellen ohne spezifischen Typ definiert.

Die Tupel innerhalb einer BAT werden als *BUNs* (*Binary Units*) bezeichnet, wobei der linke Wert auch als *head* und der rechte *tail* bezeichnet wird. Laut Definition von Kersten wird eine BAT als Multimenge definiert.

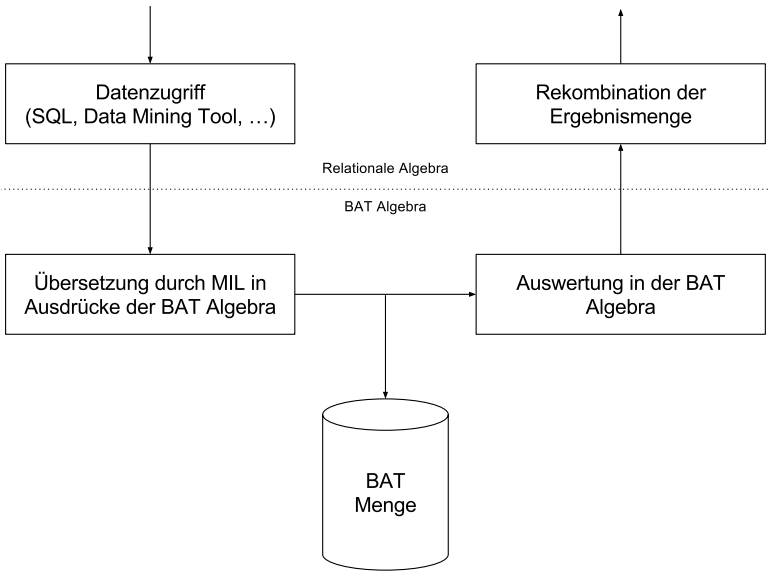

Abbildung 3.1: Verarbeitung bei Zugriffen in MonetDB

3.1.1.1 Vertikale Fragmentierung

Relationen, wie beispielsweise die *Person Table* in Abbildung 3.2, können in der Regel beliebig viele Attribute beziehungsweise Spalten besitzen. Für die Verwendung der BAT Operatoren werden daraus binäre Relationen für jedes Attribut erstellt. Für die Zuordnung zu ihren ursprünglichen Tupel werden die *heads*, die linke Spalte, in den BATs als Objekt-Identifizierer (*OID*) verwendet. Die Werte der separierten Spalten werden in den *tails*, der rechten Spalte, gespeichert.

In Abbildung 3.2 wird eine vertikale Fragmentierung anhand eines Beispiels aufgezeigt.

Anhand der eindeutigen *oid* kann ein Datensatz der ursprünglichen Relation zusammengesetzt werden. Um somit die gesamte ursprüngliche Relation zu erstellen, müssen die Tupel aller fragmentierten Unterrelationen anhand ihrer *oid* zusammengefasst werden. Abbildung 3.3 zeigt weiter am Bespiel, wie die Tabelle wiederhergestellt wird. Dabei ist zu beachten, dass die BAT von sich aus keine Informationen über die ursprünglichen Namen der Attribute oder deren Reihenfolge enthalten. Diese müssten als Meta-Informationen beigefügt werden oder in übergeordneten Datenstrukturen zugeordnet werden.

3.1.1.2 Operationen

Die BAT Algebra verfügt über eine Grundmenge an Operationen. Diese unterteilen sich in die *(rein-)algebraischen* Operationen und die *update*-Operationen.

Unter den algebraischen Operationen bezeichnen Funktionen für das Manipulieren und Verbinden von bestehenden BATs. Darunter finden sich unter anderem einige Funktionen der relationalen Algebra wie die Gruppierung, der Verbund oder Mengen-Operatoren. Grundlegend sind jedoch Funktionen zum Spiegeln oder Kippen von BUN-Werten. Nachfolgend werden die algebraischen Operationen aus dem Paper von Kersten [2] aufgeführt, welche für die Implementierung in Clojure in Frage kommen.

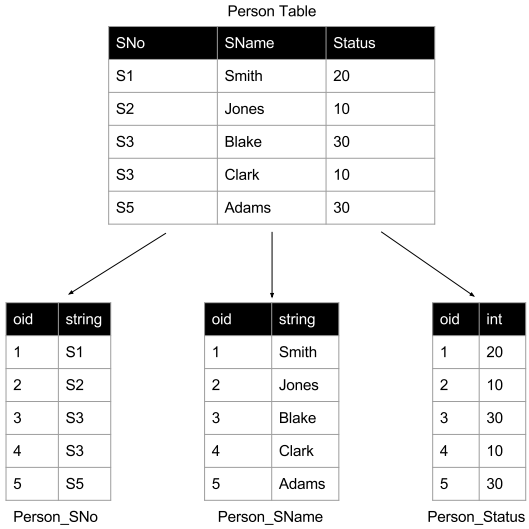

Abbildung 3.2: Fragmentierung einer relationalen Tabelle in BAT Tabellen (Vgl. [**Seite 104**, **Abbildung 3**, 2])

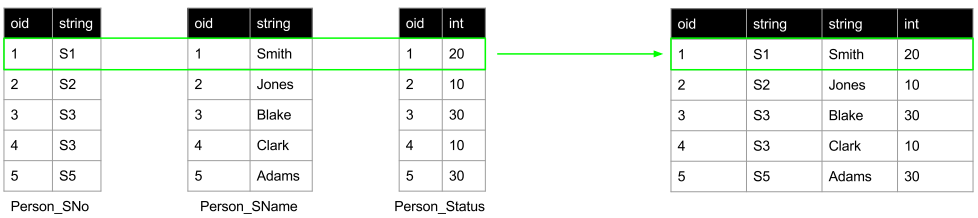

Abbildung 3.3: Wiederherstellung einer relationalen Tabelle aus BAT Tabellen

select (Definition 3.1) liefert alle *heads* einer BAT, deren *tails* für ein Prädikat *pred* (mit der Arität $n+1$, wobei n der Anzahl an optionalen Zusatzparametern p entsprechen sollte) wahr sind.

$$select(bat[H,T]AB, pred_{n+1}, p_1, ..., p_n) : bat[H, oid] \equiv \langle [a, nil] | [a,b] \in AB \wedge pred(b, p_1, ..., p_n) \rangle \tag{3.1}$$

Die **find** Operation (Definition 3.2) bedarf als Parameter eine *BAT* und einen Wert, dem Typ entsprechend des *heads* der *BAT*. Die Operation liefert den Wert der *tail* aus einer *BUN*, dessen *head* den übergebenem Wert entspricht.

$$find(bat[H,T]AB, Ha) : T \equiv b \, if \exists [a,b] \in AB, else \, t(nil) \tag{3.2}$$

join (Definition 3.3) verbindet zwei BATs anhand ihrer Werte und einer übergebenen booleschen Funktion.

$$join(bat[H_1,T_1]AB, bat[T_1,T_2]CD, pred_{n+2}, p_1, ..., p_n) : bat[H_1,T_2] \equiv \langle [a,d] | [a,b] \in AB \wedge [c,d] \in CD \wedge pred(b,c,p_1,...,p_n) \rangle \tag{3.3}$$

Die Operationen **reverse** (Definition 3.4) und **mirror** (Definition 3.5) strukturieren die zu übergebene BAT um. **reverse** vertauscht *head* und *tail* für alle BUNs einer BAT. **mirror** ersetzt in allen BUN in der übergebenen BAT den Wert aus *tail* durch den Wert aus *head*.

$$reverse(bat[H,T]AB) : bat[T,H] \equiv \langle [b,a] | [a,b] \in AB \rangle \tag{3.4}$$

$$mirror(bat[H,T]AB) : bat[H,H] \equiv \langle [a,a] | [a,b] \in AB \rangle \tag{3.5}$$

mark (Definition 3.6) ersetzt alle *tails* einer übergebenen BAT durch eine ansteigende Nummerierung, beginnend in der ersten Zeile mit o.

$$mark(bat[H,T]AB, oid\, o) : bat[H, oid] \equiv \{[a_0, o], ..., [a_n, o+n]\} \; if \; AB = \cup_{i=0}^{n} \langle [a_i, b_i] \rangle \tag{3.6}$$

Mit **count** (Definition 3.7), **sum** (Definition 3.8), **max** (Definition 3.9) und **min** (Definition 3.10) sind Aggregatfunktionen in der BAT Algebra definiert:

- **count** liefert die Anzahl an BUNs einer BAT.
- **sum** summiert die Zahlenwerte aller *tails* in einer BAT.
- **max** liefert den höchsten Zahlenwert aus allen *tails*.
- **min** liefert den niedrigsten Zahlenwert aus allen *tails*.

$$count(bat[H,T]AB) : int \equiv |AB| \tag{3.7}$$
$$sum(bat[H,T]AB) : T \equiv \Sigma_{[a,b] \in AB} b \tag{3.8}$$
$$max(bat[H,T]AB) : T \equiv b : [a,b] \in AB \wedge \nexists\, y > b, [x,y] \in AB \tag{3.9}$$
$$min(bat[H,T]AB) : T \equiv b : [a,b] \in AB \wedge \nexists\, y < b, [x,y] \in AB \tag{3.10}$$

Aus dem Bereich der **Mengenlehre** sind folgende Operationen definiert:

- **unique** (Definition 3.11) stellt aus einer Multimenge eine Menge her. Sie filtert Duplikate aus einer BAT heraus.
- **diff** (Definition 3.12) liefert die Differenzmenge zweier BATs.
- **union** (Definition 3.13) bildet die Vereinsmenge aus zwei BATs.
- **intersect** (Definition 3.14) liefert die Schnittmenge zwischen zwei BATs.

$$unique(bat[H,T]AB) : bat[H,T] \equiv \{[a,b] | [a,b] \in AB\} \tag{3.11}$$

$$\begin{aligned} diff(bat[H,T]AB, bat[H,T]CD) : bat[H,T] \equiv \{[c,d] | [c,d] \in CD \\ \wedge \nexists\, [c,d] \in AB\} \end{aligned} \tag{3.12}$$

$$\begin{aligned} union(bat[H,T]AB, bat[H,T]CD) : bat[H,T] \equiv \{[a,b] | [a,b] \in AB \\ \vee [a,b] \in CD\} \end{aligned} \tag{3.13}$$

$$\begin{aligned} intersect(bat[H,T]AB, bat[H,T]CD) : bat[H,T] \equiv \{[a,b] | [a,b] \in AB \\ \wedge [a,b] \in CD\} \end{aligned} \tag{3.14}$$

Die **group**-Operationen liefern Gruppen für BAT-Inhalte in Form von Gruppierungs-BATs. Sie sind definiert mit einer oder zwei BATs als Parametern.

Erstere (Definition 3.15) bildet die Gruppen anhand der Werte in den *tails*. Die Funktion $id_{AB}(...)$ in der Definition 3.15 liefert eine künstliche Gruppen-ID für die Gruppen.

Die Operation für zwei BATs (Definition 3.16) liefert eine Gruppierungs-BAT mit den Gruppen der ersten übergebenen BAT unter Berücksichtigung der weiterer Unterteilungen der zweiten BAT. Die Gruppen werden anhand der Kombinationen aus den *tails* zugeordnet. Logisch verbunden werden die BATs anhand ihrer *heads*. Die Funktion $id_{CD}(...)$ in der Definition 3.16 liefert eine künstliche Gruppen-ID für die Gruppen anhand der unterschiedlichen Kombinationen der *tails*.

$$group(bat[oid,T]AB) : bat[oid,oid] \equiv \{[a,o] | o = id_{AB}(b) \wedge [a,b] \in AB\} \tag{3.15}$$

$$group(bat[oid, oid]AB, bat[oid, T]CD) : bat[oid, oid] \equiv \{[a, o]| \\ o = id_{CD}([b, d]) \wedge [a, b], [o, b] \in AB \wedge [a, d] \in CD\} \tag{3.16}$$

fragment (Definition 3.17) liefert eine Fragmentierung einer BAT anhand der Bereiche übergeben durch eine zweite. *Heads* und *tails* in *CD* wirken als Unter- beziehungsweise Obergrenzen für die Fragmente in der Ergebnismenge.

$$fragment(bat[H, T]AB, bat[H, H]CD) : bat[H, bat[H, T]] \equiv \\ \{[h, select(AB, between, l, h)]|[l, h] \in CD\} \tag{3.17}$$

split (Definition 3.18) teilt eine BAT in bis zu n Bereiche und liefert deren niedrigsten und höchsten *head* als *head* und *tail* als Ergebnisse.

$$split(bat[H, T]AB, int\, n) : bat[H, H] \equiv \{[l, h]|l \leq h \wedge \exists[l, x], [h, y] \in AB\} \\ \wedge \forall [a, b] \in AB : \exists unique[l, h] : l \leq a \leq h \tag{3.18}$$

Die Operation in Definition 3.19 wird als **Multijoin** beschrieben.[2, Seite 107] Mehrere BATs werden anhand ihrer *heads* verbunden und eine übergebene Funktion wird mit allen zugehörigen *tails* als Parameter ausgewertet.

$$[f](bat[H, T_1]AB_1, ..., bat[H, T_n]AB_n) : bat[H, T_r] \equiv \langle [a, f(b_1, ..., b_n)]| \\ \forall 1 \leq i \leq n : [a, b_i] \in AB_i \rangle \tag{3.19}$$

Die Operation in Definition 3.20 wird als **pump** beschrieben.[2, Seite 107] Sie wird verwendet, um eine Operation, welche Gruppierung voraussetzt, auf alle Gruppenwerte anzuwenden. Die Gruppen von der Ziel-BAT wird anhand einer zweiten zugeordnet.

$$\{f\}(bat[H, T]AB, bat[H, I]CD) : bat[H, R] \equiv \langle [a, f(S_a)]| \\ [a, d] \in CD, S_a = \langle [b, b]|[a, b] \in AB \rangle \rangle \tag{3.20}$$

Als *update*-Operationen zählen Funktionen, die BATs modifizieren oder erweitern. Dazu gehören bekannte Operationen wie *insert*, *delete*, oder *update*. Diese fügen neue BUNs zu einer BAT hinzu, beziehungsweise entfernt oder verändert bestehende.

Weiter werden im Paper von Kersten [2] auf Funktionen für Transaktionsmanagement, sowie zum Verwalten von entfernten Zugriff auf Datenbanken aufgelistet. Diese werden jedoch nicht für die neue Implementierung in Clojure berücksichtigt, da hier nur Augenmerk auf die Realisierung der Algebra gelegt wird.

3.1.2 Abbildung der Algebra

MonetDB unterstützt als Benutzer-, beziehungsweise Frontend-Schnittstellen gängige Zugriffsmöglichkeiten auf die internen Daten wie SQL oder ODMG (*Object Database Management Group*). Innerhalb des Datenbanksystems werden die darüber gesendeten Abfragen in die MIL-Syntax umgewandelt. Die MIL (*Monet Interpreter Language*) ist eine für die MonetDB entwickelte Implementierung der BAT-Algebra. Somit übt die MIL ihre Operationen auf binäre Tabellen aus. Besagte Operatoren aus Unterkapitel 3.1.1.2 sind in dieser Sprache so implementiert, wie Funktionen zur Transaktionssteuerung und das Steuern von Festplattenzugriffen.

Eine Umwandlung beispielsweise einer SQL-Abfrage wird demnach in Teilschritte aus der BAT-Algebra übersetzt. Diese Teilschritte agieren auf BATs, die die Tabellen aus der SQL-Abfrage abbilden. Dabei können nicht in der SQL-Abfrage verwendete Attribute außer acht gelassen werden.

In Listing 3.1 wird eine SQL-Abfrage in die MIL übersetzt wird. Innerhalb der SQL-Abfrage werden die Datensätze gefiltert und zwei Tabellen miteinander verbunden. Dies spiegelt sich im MIL-Code durch den *select* (Zeile 8) und durch die *join*-Operationen (Zeilen 10, 11, 16 und 18). Die Sortierung aus der SQL-Abfrage wird im MIL-Code erst nach dem Verlassen der Form der Relationen als BATs getätigt. In Zeile 22 werden alle BATs, die zur Ergebnismenge gehören, zu einer Relation höherer Algebra zusammengefasst. Innerhalb dieser Operation wird durch die Angabe der Attribut-Positionen als numerische Werte die Sortierung vorgegeben.

```
SELECT item.id AS id,
       item.price*item.tax AS total
WHERE order.id = item.order AND
      order.discount BETWEEN 0.00 AND 0.06
ORDER BY total,id


ORD_NIL := select(order_discount,"between",0.0,0.06)
# a bat[oid,oid], head column with selected order-oids, nils in tail
IDS_NIL := join(order_id.reverse,ORD_NIL,"=")
# creates a bat[oid,oid] with selected order-IDs in head, nil tail
ITM_NIL := join(item_order,IDS_NIL,"=")
# creates a bat[oid,oid] with selected item-IDs in head, nil tail
UNQ_ITM := mark(ITM_NIL,oid(0)).reverse
# creates a bat[oid,oid] fresh oids in head, selected item-IDs in tail
UNQ_PRI := join(UNQ_ITM,item_price,"=")
# creates a bat[oid,flt] with selected item-IDs and their prices
UNQ_TAX := join(UNQ_ITM,item_tax,"=")
# creates a bat[oid,flt] with selected item-IDs and their taxes
UNQ_TOT := [*](UNQ_PRI,UNQ_TAX)
# creates a bat[oid,flt] with selected item-IDs and totals
table("2,1",UNQ_ITM,UNQ_TOT)
# prints a 2-column table with item IDs and totals, with major ordering on the second column
    , and secondary ordering on the first
```

Listing 3.1: Einfache SQL-Abfrage und MIL-Übersetzung (Vgl. [**Abbildung 4**, 2])

3.2 Umsetzung

Die Operationen der BAT-Algebra sind, wie einleitend erwähnt, Abbildungen von relationaler Operationen in mehrere Unterschritte. Ein direkter Vergleich von der BAT-Algebra mit beispielsweise der Algebra in Kapitel 2 ist schwierig. Für einen direkten Vergleich wäre eine automatisierte Abbildung, wie etwa in der MIL von Kersten, notwendig. Zu Beginn dieses Teilprojekts wurde sich gegen die Implementierung eines Übersetzers entschieden. Gründe dafür waren zum einen der erhöhte Aufwand für die vielseitige Abbildung. Zum anderen sollte im Projekt zunächst der Fokus auf die BAT-Implementierung gesetzt werden. Eine mögliche Weiterentwicklung kann zu einem späteren Zeitpunkt geschehen.

Die BAT-Algebra weicht in ihren Eigenschaften an gewissen Stellen von den in Kapitel 1.1 genannten Richtlinien ab. Zur Umsetzung der BAT-Datentypen, sowie der Algebra, müssen mehrere Änderungen in der Definition geschehen.

3.2.1 Relation statt Multimenge

Der Datentyp der binären Relationen ist in der BAT-Algebra als Multimenge definiert. Duplikate in einer BAT können auftreten, auch ohne dass in der ursprünglichen Relation ein Duplikat vorhanden ist. Gründe dafür können Kombinationen von Operationen und Datensätzen sein. Unter bestimmten Umständen lassen sich diese Duplikate sogar nicht umgehen.

Im folgenden Beispiel wird anhand einer Relation von Verkaufsdaten dieser Fall geschildert. Die Attribute dieser Relation sind die Rechnungsnummer, Name des Käufers sowie des Artikels und die Menge. In Abbildung 3.4 wird die Relation, sowie das Fragmentieren in die jeweiligen BATs geschildert.

Für das Beispiel wird nun erfragt, welcher Kunde wie viele Artikel Nägel gekauft hat. Eine SQL-Abfrage dazu ist in Listing 3.2 aufgeführt.

```
SELECT Name, sum(Menge) As Menge
FROM Bestellungen
WHERE Artikel = "Nagel"
GROUP BY Name
```

Listing 3.2: SQL-Anfrage für die summierte Menge aller gekaufter Nägel pro Person

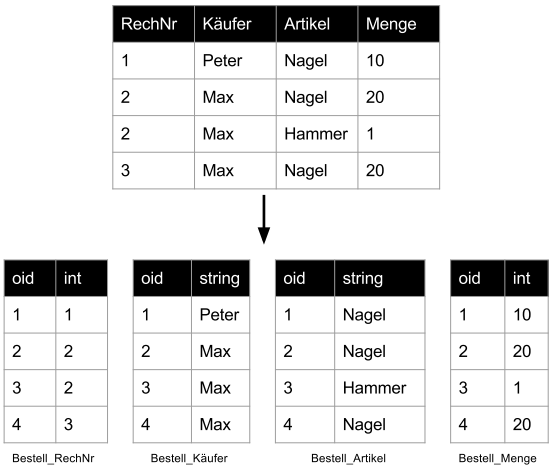

Abbildung 3.4: Beispielrelation "Bestellungen"(oben) und dessen BAT-Fragmentierung (unten)

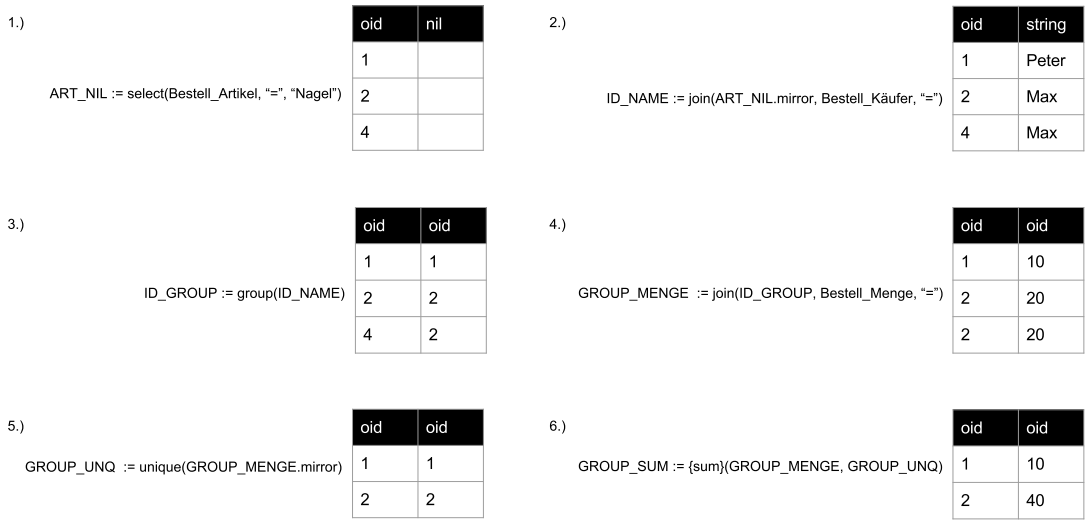

Abbildung 3.5: Summierung der Mengen in Beispiel zu "Bestellungen"

Die Abfolge inklusive der Zwischenergebnisse sind in Abbildung 3.5 zu sehen. Die Zwischenergebnisse zeigen eine mögliche Ergebnismenge. Durch die *ID*-Funktionen in den Gruppierungsfunktionen (siehe auch Definition 3.15 und 3.16 in Kapitel 3.1.1.2) kann in Schritt 3 der Abbildung eine andere Id für die Gruppe mit den *oids* 2 und 4 gewählt werden.

1. In Schritt 1 werden zunächst die *oids* herausgefiltert, die zu einer Bestellung von dem entsprechenden Artikel gehören.

2. Im 2. Schritt wird das Ergebnis mit den Namen der Käufer verbunden.

3. Schritt 3 ersetzt den Namen durch eine Gruppen-ID.

4. In Schritt 4 werden zu den Gruppen anhand der *oids* die Mengen verbunden.

5. Der 5. Schritt erstellt eine gespiegelte BAT mit allen vorhandenen Gruppen. Diese BAT wird im nächsten Schritt für das angeben aller Gruppen verwendet. Außerdem filtert er doppelte Gruppen heraus.

6. Im 6. Schritt werden dann die Mengen innerhalb der Gruppen aus Schritt 4 summiert.

Ab Schritt 4 ist in Abbildung 3.5 zu sehen, dass unter Verwendung der BAT-Algebra nach den Definitionen in Kapitel 3.1.1.2 eine Multimenge als Datentyp zwingend notwendig ist. Die Gruppierung mit der ID *2* werden identische Werte verbunden. Diese waren in vorherigen Schritten noch unter anderen *oids*, gehören aber zur selben Gruppe. Würden diese Tupel in einer Menge dargestellt werden, würde eines der Tupel als Duplikat verschwinden. Das würde in Schritt 6 das Ergebnis der Summenfunktion verfälschen.

Nach den definierten Richtlinien aus Kapitel 1.1 hat eine Relationen keine Duplikate. Für diese Umsetzung der BAT-Algebra wird somit innerhalb einer BAT eine Menge ohne möglichen Duplikaten verwendet. Die Operation *unique* wird dementsprechend nicht implementiert, da das Datenmodell selbst diese Funktion enthält.

Eine Lösung für das oben beschriebene Problem wird weiter in Kapitel 3.2.3.3 beschrieben.

3.2.2 Das Datenmodell

Ein BAT wird in Clojure-Syntax durch *deftype* erstellt. Definiert wird der Datatype mit den BUNs als Argument. Für den *type* wurden die *specs* für die Funktionen *seq*, *count* und *hashCode* angegeben (s. Listing 3.3). Deftype bietet neben möglichen anderen *Specs*, die Möglichkeit, direkten Zugriff auf die BUNs ohne Zwischenschritt zu liefern. Da keine anderen Informationen innerhalb der BAT bisher dargestellt sind, ist dies in der Entwicklung von Operationen von praktischen Nutzen.

```
1  (deftype BAT [buns]
2
3    clojure.lang.Seqable
4    (seq [this]
5      (seq (.buns this)))
6
7    clojure.lang.Counted
8    (count [this]
9      (count (.buns this)))
10
11   Object
12   (hashCode [this]
13     (let [hashs (conj (map hash (.buns this)) 17)]
14       (apply hashCalcHelper  hashs))))
```

Listing 3.3: Definiton des Datatype BAT

Die BUNs beziehungsweise die Tupel innerhalb der BAT werden in einem Set gesammelt. Jede BUN besteht aus einer HashMap mit jeweils zwei Wertepaaren. Die Wertepaare haben *:head* und *:tail* als Schlüsselattribute. In Listing 3.4 die Darstellung einer BAT beispielhaft mit drei BUNs gezeigt. Die Werte innerhalb einer BUN sind nicht auf bestimmte Typen beschränkt. Möglich ist es auch, eine BAT als Wert in einer BUN zu speichern.

```
1  #{{:head 1, :tail "A"}
       {:head 2, :tail "B"}
       {:head 3, :tail "C"}}
```

Listing 3.4: Darstellung der BUNs in einer BAT

Metadaten, wie der Name der BAT, beziehungsweise des Attributes in der ursprünglichen Relation, sind zunächst nicht angedacht im Datenmodell. Möglich wäre auch, ein Log für den temporären Zustand der Situation innerhalb des Datenmodells zu speichern. Jedoch wurde in dieser Implementierung der Augenmerk rein auf die Daten gelegt.

Das Zusammenfassen mehrerer BATs als Sammlung einer Fragmentierung wird als HashMap dargestellt. Somit wird anhand des ursprünglichen Attributnamen der Schlüssel für die jeweilige BAT verwendet (siehe Listing 3.5).

```
#{"SNo" Person_SNo_BAT,
      "SName" Person_SName_BAT,
      "Status" Person_Status_BAT }
```

Listing 3.5: Darstellung einer BAT-Map zu Abbildung 3.2

Um eine BAT zu erzeugen wird die Funktion *bat* (siehe Listing 3.6) bereitgestellt. Sie kann mit einem oder mit mehreren Parametern aufgerufen werden. Der Datentyp BAT soll nicht direkt vom Endbenutzer verwendet werden, da Dateninkonsistenz oder ähnliche Probleme beim falschen Verwenden entstehen könnten. Diese können schleichend erst nach mehreren Operation zu Fehlern führen. Die *bat*-Funktion soll die Daten für die BAT aufbereiten und in das korrekte Format überführen.

```
(defn bat
  ([tuple-vec]
    (if (coll? tuple-vec)
      (let [data (into #{} (comp
                    (filter map?)
                    (filter #(= 2 (count %)))
                    (filter #(contains? (set (keys %)) :head))
                    (filter #(contains? (set (keys %)) :tail)))
              (seq tuple-vec))]
        (BAT. data))
    (BAT. #{})))
  ([one & more]
    (let [tails (map #(assoc {} :tail %) (conj (seq more) one))
        heads (map #(assoc {} :head %) (rest (take (inc (count tails)) (range))))
        both  (set (map (fn [pair] (apply merge pair)) (map vector heads tails)))]
      (BAT. both))))
```

Listing 3.6: Implementierung: bat

Der Aufruf der Funktion mit einem Parameter setzt eine Kollektion als Übergabewert voraus, ansonsten wird eine leere BAT als Rückgabewert geliefert. Die übergebene Kollektion wird nach HashMaps gefiltert, dessen Inhalt aus zwei Wertepaaren besteht. Die Schlüssel der Wertepaare müssen als *:head*, beziehungsweise *:tail* definiert sein. Alle HashMaps, die diese Voraussetzung besitzen, werden in die Ergebnisrelation übertragen (siehe Listing 3.7).

```
(bat [{:head 1 :tail 1}
{:head 2 :tail 2}
{:head 3 :tail 3}])
      ; #BAT
       3 - 3
       1 - 1
       2 - 2
```

Listing 3.7: *bat*-Aufruf mit einem Parameter

Der Aufruf der *bat*-Funktion liefert eine BAT mit generierten *oids* in den *heads* und den übergebenen Werten in den *tails* (siehe Listing 3.8).

```
(bat \a \b \c)
      ; #BAT
       1 - a
       3 - c
       2 - b
```

Listing 3.8: *bat*-Aufruf mit mehreren Parameter

Für diese Fragmentierung liefert die Clojure-Implementierung eine Funktion *convertToBats* (siehe Listing 3.9). Als Parameter nimmt die Funktion eine Tabelle im Clojure-Format in Form von einer Sammlung von HashMaps. Als Resultat liefert die Funktion eine HashMap von BATs. Mit Hilfe dieser Funktion soll es Benutzern erleichtert werden, gängige Tabellen-Daten in die BAT-Datenstruktur zu übersetzen. Für die Verwendung der Operationen werden jedoch nur die einzelnen BATs übergeben.

```
(defn convertToBats
  [table]
  (let [heads (reduce (fn [heads entry](apply conj heads (keys entry))) #{} table)
      table (vec table)]
    (reduce (fn [m attr]
          (assoc m attr (bat
                  (filter #(not (nil? (:tail %)))
                      (map (fn [entry]
                          {:head (inc (.indexOf table entry)),
                          :tail (get entry attr)}) table)))))
      {} heads)))
```

Listing 3.9: Implementierung: convertToBats

3.2.3 Operationen

Das Repertoire an Operationen für die *batrel*-Implementierung besteht grundlegend aus den in Kapitel 3.1.1.2 mathematisch definierten Operationen. Wie in Kapitel 3.2.1 beschrieben, zählt jedoch der *unique*-Operator nicht zu dieser Implementierung.

Details zu den Implementierungen, sowie Beispiele für die Verwendung und Änderungen der Definition finden sich in den folgenden Unterkapiteln.

3.2.3.1 Find und Select

Zu den einfacheren Operationen in der BAT-Algebra zählen *find* (s. Definition 3.2) und *select* (s. Definition 3.1).

Durch *find* wird zu einer BAT ein *tail* geliefert, sofern der *head* in dieser Relation dem übergebenen Wert entspricht. Hierbei spielt die Reihenfolge der Daten keine vordefinierte Rolle. Somit lässt sich diese Operation in Clojure durch ein simples Filtern der BUNs und Auswählen des ersten Fundes implementieren. Durch den Code in Listing 3.10 ist diese Funktion in *relation.bat* definiert. Diese Funktion ist, abgesehen von den Aggregatsfunktionen, die einzige Funktion, dessen Rückgabewertetyp keine Relation ist.

```
(defn find
[batObj head]
  (:tail (first (filter #(= head (:head %)) (buns batObj)))))
```

Listing 3.10: Implementierung: find

Die Operation *select* liefert anhand einer BAT, einer booleschen Funktion und gegebenenfalls zusätzlicher Parameter für diese Funktion eine BAT als Ergebnis. Das Ergebnis beinhaltet alle *heads*, dessen *tail* bei Anwendung der übergebenen Funktion *true* ergibt. Die Rückgabe bezieht sich nur auf die *heads*. Die *tails* der Rückgabe sind einheitlich *nil*. Listing 3.11 zeigt die Implementierung.

```
(defn select
[batObj f & more]
  (let [newBat (into [] (comp
            (filter #(let [b (:tail %)]
                (apply f b more)))
              (map #(assoc  % :tail nil)))
            (buns batObj))]
  (bat newBat)))
```

Listing 3.11: Implementierung: select

3.2.3.2 Der Verbund

Der Verbund (siehe *join* in Definition 3.3) in der BAT-Algebra unterscheidet sich dem in der Implementierung aus Kapitel 2.2.2.2 in mehreren Punkten. Zum einen kommen die Verbundattribute innerhalb der Ergebnis-Relation nicht selbst vor. Lediglich die äußeren Attribute der Operanden (*head* vom ersten und *tail* vom zweiten) sind in der Ergebnis-BAT vorhanden.

Ein weiterer Unterschied ist die Zuordnung der Verbundattribute. Die Implementierung aus Kapitel 2.2.2.2 ordnet die Verbundattribute anhand ihrer Gleichheit zu. Dies wird in der relationalen Algebra als *natural join* oder auch als natürlicher Verbund bezeichnet. Dies ist jedoch ein Spezialfall des allgemeinen Verbundes. Der allgemeine Verbund in der relationalen Algebra (auch θ-*join* oder θ-Verbund) ist durch keine spezielle Vergleichsoperation der Verbundattribute definiert. Der Vergleichsoperator ist für diesen Verbund frei wählbar.

Die BAT-Algebra nimmt den allgemeinen Verbund als Vorlage für die *join*-Operation. Der Vergleichsoperator ist dadurch frei wählbar und wird als Parameter übergeben. Optional können weitere Parameter für den Operator mit übergeben werden.

Aus der Implementierung der *join*-Operation und dessen Effizienztests in Kapitel 2.2.2.2 wurde eine Lehre gezogen. Gleichheitstests von Verbundattributen lassen sich ohne Doppelschleife in Clojure deutlich performanter darstellen. Aufgrund der verwendeten Funktionen (siehe Listing 2.14 und 2.15) ist die Herangehensweise jedoch auch nur für den Vergleich der Gleichheit anwendbar. Da die Verbundoperation in der BAT-Algebra eine beliebige Vergleichsoperation verwendet, ist die Herangehensweise keine vollständige Lösung. Jedoch sind die häufige Verwendung der Gleichheitsprüfung, sowie der deutliche Performanzunterschied Argumente für das Verwenden einer Zwischenlösung.

Für die Clojure-Implementierung wurden daher sowohl die *index*-Variante für Gleichheitsprüfungen als auch doppelte Schleife für restliche Funktionen verwendet. In Listing 3.12 ist die Definition der Clojurefunktion für den Verbund. Clojure ermöglicht es, Funktionen, wie Werte, miteinander zu vergleichen. In Zeile 7 wird geprüft, ob die übergebene Funktion f der Gleichheitsoperation entspricht. In diesem Fall werden die Relationen über die Indizierung verbunden.

Da laut Definition 3.3 die Vergleichsoperation zusätzliche Parameter enthalten kann, wird in Zeile 8 überprüft, ob zwei unterschiedliche Parameterwerte in der Gleichheitsprüfung enthalten sind. In diesem Fall wird eine leere BAT als Ergebnis geliefert, da eine Gleichheit zu zwei verschiedenen Werten nicht vorkommen kann.

Ist der Vergleichsoperator keine Gleichheitsprüfung, wird in Doppelschleife-Methode der Verbund vollzogen. Für die Verringerung der Vergleiche wird auf die Relationen die *index*-Funktion von Clojure (siehe dazu auch Kapitel 2.2.2.2) angewendet (Zeile 6 und 30). Dadurch *gruppieren* sich die BUNS anhand des Verbundattributs. Somit wird jedes Wertpaar von Verbundattributen genau einmal verglichen. Bei Übereinstimmung werden alle BUNs mit dem entsprechenden Werten verbunden und Teil der Ergebnisrelation.

```
1  (defn join
2    [batAB batCD f & params]
3      (let [ AB (map (fn[bun] (clojure.set/rename-keys bun {:tail :key})) (buns batAB))
4          CD (map (fn[bun] (clojure.set/rename-keys bun {:head :key})) (buns batCD))
5          ks #{:key}
6          idx (clojure.set/index AB ks)]
7      (if (= f =)
8        (if (and (>= (count params) 2) (not (apply = params)))
9          (bat [])
10         (bat (reduce (fn [ret x]
```

```
11          (let [filtered-idx  (if (empty? params)
12                                idx
13                                (select-keys idx [{:key (first params)}]))
14                found       (filtered-idx (select-keys x ks))]
15            (if found
16              (reduce #(conj %1 (dissoc  (merge %2 x) :key)) ret found)
17              ret)))
18          [] CD)))
19
20      (bat (reduce (fn [ret x]
21        (let [found  (reduce (fn [m [k v]]
22                       (if (apply f (:key k) (:key (first x)) params)
23                         (apply conj m v)
24                         m)) #{} idx)] (println x  idx found )
25          (if (empty? found)
26            ret
27            (reduce #(apply conj %1
28                  (map (fn[y] (dissoc (merge %2 y) :key))
29                       (second x))) ret found))))
30       [] (clojure.set/index CD ks))))))
```

Listing 3.12: Implementierung: join

Durch die freie Wahl der Vergleichsoperation lassen sich einige spezialisierte Verbünde ohne eigener Implementierung darstellen. Die Operation muss nur der Definition des jeweiligen Spezialfalls entsprechen. Das kartesische Produkt (auch Kreuzprodukt oder *cross join* genannt) ist eine Operation, in dem die Tupel der einen Relation mit jedem Tupel der zweiten Relation verbunden werden. Dies lässt sich mit unserer definierten *join*-Funktion durch eine Vergleichsoperation darstellen, die in jedem Fall *true* als Wert liefert (siehe Listing 3.13).

```
1 (def A (bat [{:head 1 :tail 1}
       {:head 2 :tail 2}
       {:head 3 :tail 3}]))
2
3 (def B (bat [{:head 4 :tail 4}
       {:head 5 :tail 5}
       {:head 6 :tail 6}]))
4
5 (join A B (fn[& _] true))
       ;#BAT
       3 - 6
       3 - 5
       3 - 4
       2 - 5
       2 - 4
       1 - 6
       1 - 4
       1 - 5
       2 - 6
```

Listing 3.13: Beispiel: Kartesisches Produkt

3.2.3.3 Neudefinition der Gruppierung

Da die ursprüngliche Gruppierung zu Duplikaten und Inkonsistenz führen würde (siehe dazu Kapitel 3.2.1) sind Neudefinitionen der Gruppierungsoperationen notwendig. Grundlegend sollen die gruppierten BUNs in Unter-BATs dargestellt werden. Das Konzept von BATs in BATs widerspricht der Algebra nicht, da bereits in der Definition der BAT dieser Datentyp als möglicher Inhalt vorgegeben ist.

Das Zusammenfinden, sowie das Indizieren der Gruppen soll von der ursprünglichen Definition übernommen werden. Die zusammengefassten BUNs sollen unverändert als einzelne BAT in dem *tails* der Ergebnisrelation dargestellt werden. In den *heads* befinden sich die Gruppen-IDs, welche in der ursprünglichen Definition in den *tails* waren. In Abbildung 3.6 ist eine beispielhafte Darstellung, wie sich die neue Gruppierung im Vergleich zur alten verhalten soll.

Auf Basis der Definitionen 3.15 und 3.16 wurden die nun folgenden mathematischen Definition für die neuen Gruppierungen mit einem (3.21) und mit zwei Parametern (3.22) erstellt. Die id-Funktionen verhalten sich wie in den den ursprünglichen Definitionen. Zu beachten ist, dass die Gruppierung mit zwei Parametern eine *vor*-gruppierte BAT als ersten Parameter voraussetzt. Dies war auch schon indirekt in der ursprünglichen Definition so, da beide Typen von AB *oid* sind. Dies führt dazu, dass die besagte

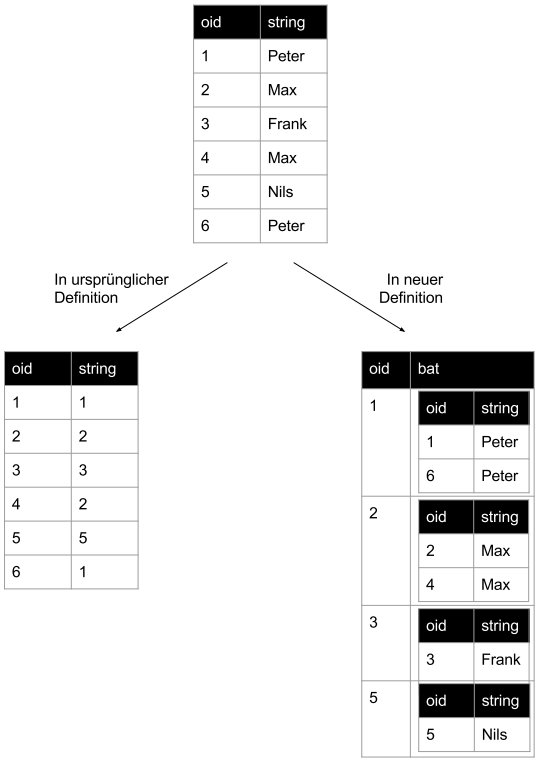

Abbildung 3.6: Darstellung gewünschter Ergebnismenge nach Gruppierung

Gruppierung nur als Verfeinerung einer bereits bestehenden Gruppen-BAT verwendet wird.

$$group(bat[oid, T]AB) : bat[oid, bat[oid, T]] \equiv \{[o, XY] | \langle o = id_{AB}(b) \\ \wedge [o, b] \in AB \wedge \forall [x, y] \in XY : y = b \rangle\} \tag{3.21}$$

$$group(bat[oid, bat[oid, T]]AB, bat[oid, T]CD) : bat[oid, bat[oid, T]] \equiv \\ \{[o, XY] | \langle [a, b] \in AB : \exists b : XY \subset b \\ \wedge [x, y] \in XY, [c, d] \in CD : \\ \langle \forall [x_i, y_i], [x_j, y_j] \exists [c_i, d_i], [c_j, d_j] : \\ (x_i = c_i) \wedge (x_j = c_j) \wedge (d_i = d_j) \rangle \\ \wedge o = id_{AB}(XY) \rangle\} \tag{3.22}$$

Die dazugehörige Implementierung für *batrel* ist in Listing 3.14 zu sehen. Die Definition *group* hat sowohl eine Funktion für die ein-Parameter-Variante, als auch für die mit zwei Parametern.

```
1  (defn group
2    ([batObj]
3      (let [tails (distinct (map (fn[tuple](:tail tuple)) batObj))
4          tails' (map (fn[tail] (bat (filter #(= (:tail %) tail) batObj))) tails)]
5        (bat (map (fn[tuple] {:head (:head (first tuple)) :tail tuple}) tails'))))
6    ([AB CD]
7      (let [B (distinct (map (fn[tuple] (:tail tuple)) AB))
8          groups (vals (reduce (fn[m bun] (let [k (:head bun)
9                                v (:tail bun)]
10                           (if (contains? m v)
11                             (assoc m v (conj (get m v) k))
12                             (assoc m v #{k})))) {} (buns CD)))
13         tails' (reduce (fn[m x](apply conj m x)) []
14                  (map (fn[subBat] (filter (fn[x] (not (empty? x)))
```

```
15                                  (map (fn[group]
16                                    (filter #(contains? group (:head %))
17                                        subBat))
18                                    groups)))
19                          B))]
20      (bat (map (fn[tuple] {:head (:head (first tuple)) :tail (bat tuple)}) tails')))))
```

Listing 3.14: Implementierung: group

Die Weiterverwendung der verschachtelten BATs kann grundlegend auf 2 verschiedene Arten geschehen. Die Verwendung jeder einzelner BAT im *tail* kann beispielsweise durch direktes Selektieren des Wertes geschehen. *find* (siehe Definition 3.2), welcher der Operator zum Erlangen von Werten in BATs ist, kann dafür verwendet werden, zu einer bestimmten Gruppen-ID die Gruppen-BAT zu liefern. Auch durch die Sequenzierbarkeit des BAT-Datentyps lässt sich jede Unter-BAT einzeln erreichen und verwenden.

Soll hingegen die Struktur der Ober-BAT nicht verändert werden, während Operationen an den Gruppen-BATs ausgeführt werden, kann die *mulitjoin*-Operation (siehe Definition 3.19) dafür verwendet werden. Der *multijoin* wendet eine Funktion auf jedes einzelne *tail* einer BAT an. Diese Funktion kann benutzerdefiniert sein, kann aber auch aus der BAT-Algebra stammen. Die Struktur der oberen BAT wird dabei nicht verändert.

In Listing 3.15 ist hierzu ein Beispiel. In den Befehlen von Zeile 2 wird eine BAT-Gruppierung von allen gleichen Zahlen erzeugt. Dadurch entstehen vier Untergruppen. Durch die Verwendung der Summierung mit Hilfe des *multijoins* (Zeile 1) werden die Gruppen innerhalb der BAT summiert. Als Ergebnis bleibt die Struktur der vier Untergruppen vorhanden und statt der Unter-BATs sind die Ergebnisse der Operation vorhanden.

```
1 (multijoin sum
2       (group (bat 10 10 20 20 30 40 40)))
            ;#BAT
              1 - 20
              6 - 80
              2 - 40
              5 - 30
```

Listing 3.15: Beispiel für das Verwenden von Multijoin an einer Gruppierung

3.2.4 BATvar

Als Äquivalent zu der RelVar aus Kapitel 2.2.3 wird für die BAT-Implementierung eine ähnliche Funktionalität geliefert. Der Funktionsumfang aus Tabelle 2.2 ist dabei nahezu identisch realisiert. Der große Unterschied liegt dabei, dass es sich bei den BATVars um eine Menge von BAT-Relationen handelt, welche attributweise als Map aufgeführt sind. Dies ist angelehnt an die vertikale Fragmentierung einer Tabelle. Da diese einzelnen BATs trotz ihrer Aufspaltung noch eine gewisse Verbindung besitzen, werden sie zusammen als eine BATVar verwaltet. In Tabelle 3.1 ist eine Übersicht der Funktionen für die Verwendung von BATVars zu finden.

Die Verwendung und Verwaltung von Constraints wurde in der Implementierung zunächst weggelassen. Dieser Punkt ist jedoch für eine Weiterentwicklung der Bibliothek offen.

Operation	**Beschreibung**
(batvar [batMap] [batMap cons])	Erstellt anhand einer Map von Attributnamen und BATs *batMap* eine *BATVar*. Optional können eine Menge an Constraints *cons* zusätzlich übergeben werden.
(assign! batVar batMap)	Ersetzt die beinhaltende Relation von *BATVar batVar* durch die Map von Attributnamen und BATs *batMap*.
(insert! [batVar attr head tail] [batVar tuple])	Fügt einer BAT in *BATVar batVar* mit dem Attribut *attr* einen Eintrag mit *head* und *tail* hinzu. Alternativ kann allen BATs in *batVar* das jeweilige Attribut aus einem übergebenen Tupel *tuple* eingefügt werden.
(delete! [batVar attr head tail] [batVar tuple])	Fügt in einer BAT in *BATVar batVar* mit dem Attribut *attr* einen Eintrag mit *head* und *tail*. Alternativ kann allen BATs in *batVar* das jeweilige Attribut aus einem übergebenen Tupel *tuple* entfernt werden.
(update! [batVar attr head oldTail newTail] [batVar attr old new])	Setzt in einer BAT in *BATVar batVar* mit dem Attribut *attr* einen Eintrag mit *head* und *oldTail* den Tail-Wert auf *newTail*. Optional kann *head* weggelassen werden um alle Werte *old* in *new* zu verändern.
(makeTable! [orderseq batVar] [batVar])	Erstellt aus der *BATVar batvar* eine Tabelle im xrel-Format. Optional kann eine Reihenfolge der Attribute *orderseq* vorgegeben werden.
(save-batvar batVar file)	Schreibt den Inhalt (Relation und Constraints) von *BATVar batVar* in eine Datei mit dem Pfad *file*.
(load-batvar file)	Liest den Inhalt einer Datei mit dem Pfad *f* und erstellt eine *BATVar* anhand des Inhalts.
(save-db db file)	Schreibt den Inhalt (Relation und Constraints) von *BATVar*-Datenbank *db* in eine Datei mit dem Pfad *file*.
(load-db file)	Liest den Inhalt einer Datei mit dem Pfad *file* und erstellt eine *BATVar*-Datenbank anhand des Inhalts.

Tabelle 3.1: Übersicht der Operationen für BATVar

4 Transrelational Model

Innerhalb dieses Kapitels wird auf die Entwicklung des *Transrelational Model* (TR) eingegangen. Dieses findet seinen Ursprung in einem US Patent aus dem Jahr 1999 von dem New Yorker Stephen A. Tarin [19]. Chris Date griff die Idee in seinem Buch Go Faster![5] auf. Er verwendete TR-Technologie zum Abbilden der relationalen Algebra. Die TR-Technologie beschreibt dabei die Mechanismen innerhalb des Transrelational Model.

Im folgenden Unterkapitel 4.1 werden die Grundkenntnisse des Transrelational Model vermittelt. Darunter zählen die Erklärung der Relationsdarstellung und die grundlegenden Definitionen für das Verhalten von Operationen.

Im Unterkapitel 4.2 wird auf die Umsetzung der Definition in Clojure eingegangen. Dazu gehören auch Vergleiche von verschiedenen Ansätzen der Operationen.

4.1 Basis der Idee

Innerhalb des TR wird grundlegend in zwei Kategorien unterschieden: Werte und Struktur. In gewöhnlichen Relationen gibt es einen festen Zusammenhand zwischen diesen Kategorien. Zur genaueren Erklärung betrachten wir die Relation in Tabelle 4.1. Hier sind Werte und Struktur vereint. Werte sind die Inhalte der einzelnen Zellen wie beispielsweise *S2*, *30* oder *Jones*.

Struktur bedeutet, welche Werte zusammen gehören. In der vorgegebenen Struktur ergeben zum Beispiel die Werte *S4*, *Clark*, *20* und *London* ein Tupel. Man erkennt es daran, dass sie alle in der selben Zeile dargestellt sind. Das Gleiche zählt auch für jede andere Zeile, beziehungsweise jedes andere Tupel.

4.1.1 Abbildung der Daten

Betrachten wir ein anderes Beispiel. Die Relation in Tabelle 4.2 enthält die gleichen Werte wie die in Tabelle 4.1. Die Struktur ist jedoch anders. Innerhalb der neuen Relation sind alle Werte in ihrer Spalte aufsteigend sortiert. Gehen wir davon aus, dass die Struktur der alten Relation immer noch gelten soll, kann sie nicht aus der

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Tabelle 4.1: Supplier-Beispiel Relation nach Date

S#	SNAME	STATUS	CITY
S1	Adams	10	Athens
S2	Blake	20	London
S3	Clark	20	London
S4	Jones	30	Paris
S5	Smith	30	Paris

Tabelle 4.2: Supplier-Beispiel Relation nach Date (Spalten-orientiert)

neuen Relation erkannt werden. Die Zugehörigkeit durch die selbe Zeile ist nicht mehr gegeben. Das bereits genannte Tupel mit *S4* enthält nun *Jones*, *30* und *Paris* als weitere Werte.

4.1.1.1 Field Value Table

Der Vorteil der neuen Relation in Tabelle 4.2 ist jedoch, dass bestimmte Werte in den Spalten sehr schnell gefunden werden können. Durch ihre Sortierung kann beispielsweise eine Binärsuche Bereiche und Werte in der Spalte ausfindig machen.

An dieser Stelle kommt das TR ins Spiel. Das TR speichert die Werte einer Relation auf den Weg, wie die Relation in Tabelle 4.2 dargestellt sind. Dass heißt, jede Spalte wird für sich sortiert. Diese Ansammlung an sortierten Spalten wird *Field Value Table* (FVT) bezeichnet. Man kann also auch die Tabelle 4.2 als eine FVT von Tabelle 4.1 bezeichnen.

Dabei ist drauf zu achten, dass diese und auch die folgende Tabelle nicht für den Benutzer bestimmt sind. Die FVT ist möglicherweise inhaltlich verständlich für Benutzer. Es ist jedoch nicht gedacht, dass der Benutzer direkt mit ihr interagiert. Sie ist ausschließlich durch Verwendung von vorgegebenen Operationen zugreifbar und veränderbar.

4.1.1.2 Record Reconstruction Table

Die FVT verwaltet wie bereits erklärt die Werte einer Relation im TR. Was sie jedoch nicht abbildet ist die Struktur. Nach der Struktur der FVT würden Tupel mit fortlaufend aufsteigenden Werten wiedergegeben werden, welche mit der ursprünglichen Struktur möglicherweise nichts weiter zu tun haben.

Um dies zu verhindern, wird die *Record Reconstuction Table* (RRT) verwendet. Durch sie sollen die Werte in der FVT wieder in die vorgegebene Struktur zurückgeführt werden können. Die RRT ist eine weitere Tabelle, welche zusammen mit der FVT die Grundlage einer Relation im TR darstellt. In Tabelle 4.3 ist eine mögliche RRT für die FVT in Tabelle 4.2 zu sehen.

RRTs sind grundsätzlich von der Anzahl der Zeilen und Spalten gleich der FVT aufgebaut. Jedoch befinden sich keine Werte in den Zellen, sondern Verlinkungen, beziehungsweise Wegweiser, in Form von Zahlen. Diese Zahlen sind Zeilennummern für die jeweils nachfolgende Spalte. Werden FVT und RRT gleichzeitig betrachtet, dann steht in der FVT in einer beliebigen Zelle ein Wert und in der RRT in der *selben* Zelle die Zeilennummer für den nächsten zugehörigen Wert in der nächsten Spalte in der FVT. Die Zahlen der letzten Spalte in der RRT verweisen jeweils auf die Zeilen der ersten Spalte, sodass insgesamt ein Rundlauf für die Zuordnung eines Tupels entsteht. Dieses Vorgehen wird auch als *Zigzag*-Algorithmus (dt. *ZickZack*) bezeichnet.

S#	SNAME	STATUS	CITY
5	4	4	5
4	5	2	4
2	2	3	1
3	1	1	2
1	3	5	3

Tabelle 4.3: Beispiel einer Record Reconstruction Table

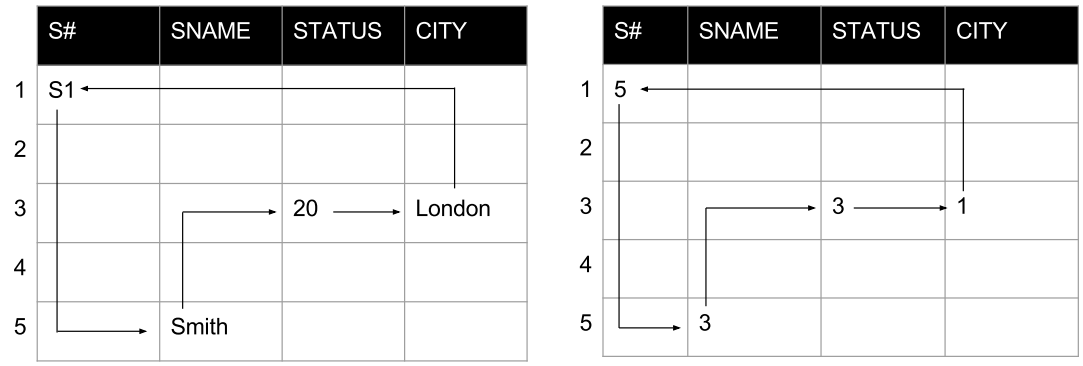

Abbildung 4.1: Rundlauf für Tupel (Vgl. [**Abbildung 4.4**, 5])

In der Abbildung 4.1 ist ein kompletter Rundlauf eines Tupels in der Relation aus der FVT und RRT aus den Tabellen 4.2 und 4.3.

Aufgrund der geschlossenen Verbindung der Tupel in der RRT können alle Werte eines Tupels von einem beliebigen Punkt aus erreicht werden. Das bedeutet, Werte, beziehungsweise ihre Position(en) können in der FVT durch effiziente Suchalgorithmen bestimmt werden. Durch ihre Positionen können wiederum anhand des Traversierens durch die RRT die zugehörigen Tupel rekonstruiert werden. Somit lassen sich effektiv zusammenhängende Daten lesen.

Anmerkung: Es existiert nicht zwingend nur eine RRT für das korrekte Rekonstruieren der gesamten Relation. In einer FVT kann eine Spalte einen Wert mehrfach enthalten. Somit kann eines der enthaltenden Tupel auf mehreren verschiedenen Wegen korrekt rekonstruiert werden.

In der Relation in Tabelle 4.1 sind Städte wie *Paris* oder *London* oder Status wie *20* oder *30* mehrfach vorhanden. Für das Rekonstruieren des Tupels $\{S1, Smith, 20, London\}$ aus Tabelle 4.2 ist es jedoch nicht wichtig, durch *welche* Spalten mit den Werten 20 oder *London* das Traversieren führt. Wichtig ist nur, dass alle Tupel für sich korrekt rekonstruierbar sind.

Mit steigender Anzahl solcher gleicher Werte steigt auch die Anzahl an möglichen RRTs für eine FVT.

4.1.1.3 Adaptionen der Datenabbildung

In den vorherigen Unterkapiteln wurde die Grundmechanik innerhalb des TR erklärt. Grundlegend wird gezeigt, wie Werte und Struktur getrennt verwaltet werden, und wie sie wieder zusammengeführt werden.

S#	SNAME	STATUS	CITY
S1	Adams	10	Athens
S2	Blake	20	London
S3	Clark	30	Paris
S4	Jones		
S5	Smith		

Tabelle 4.4: Supplier-Beispiel (kondensiert)

Für eine Optimierung der bisherigen Mechanismen stellt Chris Date in Go Faster[5] drei Anpassungen vor. Diese Anpassungen sollen vorwiegend Redundanz von Werten vermeiden. Außerdem wird eine Vorsortierung von RRTs von bestimmten Reihenfolgen der Spaltensortierungen vorgestellt. Von den Adaptionen wurde in dieser Arbeit nur Augenmerk auf die *kondensierten Spalten* gelegt.

Die *vermischten Spalten* werden auch als Adaption genannt. Dieses Konzept sieht vor, Attribute, welche gleiche Werte beinhalten können, zusammen in einer Spalte zu verwalten. Dies wurde jedoch aufgrund der mangelnden automatisierten Entscheidungsmöglichkeit, welche Spalten zu vermischen sind, nicht für die Implementierung berücksichtigt.

Auch nicht berücksichtigt für die Implementierung ist das Konzept der *Major-to-Minor Orderings*. Grob erklärt greift es auf die Anmerkung in Kapitel 4.1.1.2 zurück, dass es mehrere mögliche RRT für eine FVT geben kann. Diese Eigenschaft wird verwendet, um Vorsortierungen für die Wertesortierung nach mehreren Attributen vorzubereiten. Dies bedarf aber für die jeweilige Vorsortierung (*ORDER BY A, B, C...*) jeweils eine eigene RRT. Aufgrund des Aufwands wurde sich auch hiergegen entschieden.

Das Konzept der *kondensierten Spalten* ist eine Anpassung der Wertespeicherung. In der FVT in Tabelle 4.1 ist zu sehen, dass einige Werte, wie etwa London oder 30, mehrfach auftreten. In weitaus größeren Relationen kann dies bei mehr Attributen in weit größeren Zahlen entstehen. In Tabelle 4.4 ist eine abgeänderte FVT, in welcher keine Werte mehrfach auftreten. In diesen Beispiel werden dadurch vier Stellen eingespart, was 20% der Daten entspricht. Jedoch lässt sich die Anzahl an Verwendungen der Daten in der rekonstruierten Form nicht verändern, da die Daten somit inkonsistent werden.

Die Eigenschaft, dass FVT und RRT den gleichen Tabellen-Aufbau besitzen, ist mit dieser Erweiterung nicht mehr gegeben. Um die TR-Technologie weiter anzuwenden zu können, müssen FVT und RRT Zellen nun eine zusätzliche Verbindung besitzen. Zum einen muss aus den Zellen der FVT heraus erschlossen werden können, welche die repräsentative Zelle im der RRT ist. Zum anderen muss auch aus der RRT heraus erkannt werden können, welche Zelle in der FVT die Korrekte ist.

Innerhalb der FVT wird für jeden Wert der *Gültigkeitsbereich* angegeben, von welcher bis zu welcher Spalte dieser Wert in der RRT gilt. Gilt der Wert beispielsweise in der RRT in den ersten beiden Spalten, wäre es mit $[1 : 2]$ dargestellt. In Tabelle 4.5 sind die entsprechenden Bereiche in die kondensierte FVT dargestellt. Für nicht-kondensierte Spalten, also welche ursprünglich keine mehrfachen Werte enthielten, gibt es zwei Möglichkeiten für diese Bereiche. Zum einen können die Bereich wie in den kondensierten Spalten angegeben werden. Dies bedeutet, jeder Bereich ist entsprechend eine Spalte.

S#	SNAME	STATUS	CITY
S1	Adams [1:1]	10 [1:1]	Athens [1:1]
S2	Blake [2:2]	20 [2:3]	London [2:3]
S3	Clark [3:3]	30 [4:5]	Paris [4:5]
S4	Jones [4:4]		
S5	Smith [5:5]		

Tabelle 4.5: Supplier-Beispiel (kondensiert mit Bereichen)

S#	SNAME	STATUS	CITY
5	1 ■ 4	1 ■ 4	1 ■ 5
4	2 ■ 5	2 ■ 2	2 ■ 4
2	3 ■ 2	2 ■ 3	2 ■ 1
3	4 ■ 1	3 ■ 1	3 ■ 2
1	5 ■ 3	3 ■ 5	3 ■ 3

Tabelle 4.6: Beispiel einer Record Reconstruction Table (kondensiert mit Bereichen)

Die andere Möglichkeit ist, solche Spalten mit keinen Bereichen auszustatten. Somit ist es an der Implementierung, beziehungsweise den Operationen den Sonderfall zu erkennen. Dann muss wie im Grundkonzept die Spaltennummer des Wertes nehmen, um die Spalte für die RRT zu finden.

In der FVT in Tabelle 4.5 sind beide Varianten in Attribut S# und SNAME zu sehen.

Um auch von der RRT in die richtige Spalte der FVT schließen zu können, bedarf auch sie Zusatzinformationen. Anstatt einer Zahl, sind nun zwei in den einzelnen Zellen. Die erste Zahl (*Value-Linker*) dient zum Verweis auf die richtige Spalte in der FVT. Auch mehrere der gleichen Verweise können in einer Spalte auftreten. Dies ist ein Merkmal für mehrfache Werte in den ursprünglichen Daten. Die zweite Zahl (*Attribute-Linker*) ist die schon vorher bereits verwendete Verlinkung in die nächste Spalte.

Auch hier gibt es wieder die zwei Möglichkeiten für nicht-kondensierte Spalten in der FVT. Zum einen können einfach die FVT-Verlinkungen spaltenweise aufgelistet werden wie auch bei den kondensierten Spalten. Es kann aber auch nur die Zahl der ursprünglichen Mechanik vorliegen und die Operationen müssen den Unterschied erkennen. In der RRT in Tabelle 4.6 ist die Kondensierung entsprechend der FVT in Tabelle 4.5 dargestellt. Auch hier werden in den Attributen S# und SNAME beide Varianten für nicht-kondensierte Spalten dargestellt.

4.1.2 Operationen

Nach Date sind für die TR-Grundoperationen vorgesehen. Als grundlegende Operationen zählen das Lesen, sowie das Einfügen, Löschen oder Bearbeiten eines Tupels im TR. Die relationale Algebra selbst ist getrennt zu betrachten, kann aber auf die Basis-Operatoren zurückgreifen.

4.1.2.1 Basis-Operationen

Grundlegend werden Tupel innerhalb einer TR daran identifiziert, indem eine Position eines der Werte in der FVT beziehungsweise die jeweilige Position in der RRT vorgege-

ben wird. Anhand dieser Position lässt sich durch den *Zigzag*-Algorithmus der gesamte Tupel rekonstruieren.

Der *Zigzag*-Algorithmus ist die Beschreibung für das Traversieren durch RRT und FVT um ein Tupel wieder zu rekonstruieren. Dabei gibt die RRT durch ihre Zahlenwerte eine Wegstrecke durch die Tabelle vor. Diese liefert, in der FVT an exakt den selben Zellen verwendet, das jeweilige Tupel. Der Startpunkt, beziehungsweise die Startzelle, kann beliebig gewählt werden, da die Wegstrecke ein geschlossener Kreis ist (siehe dazu auch Abbildung 4.1).

Das *Lesen* von Tupeln ist eine Grundfunktion in der TR. Ein Tupel soll anhand der Positionsdaten einer der Zellen in der FVT beziehungsweise RRT geliefert werden. Dabei bestehen die Positionsdaten aus der Zeilen- und Spaltennummer der jeweiligen Zeile. Das Rückgabeformat ist grundlegend nicht definiert, sollte jedoch die Werte sowie die Zuordnungen zu den Attributen besitzen.

Das *Löschen* eines Tupels basiert auf einer ähnlichen Technik wie das Lesen. Anhand eines der Tupel-Zellen werden die zu löschenden Werte identifiziert. In der Relation werden nun alle diese Werte entfernt und eine neue Relation geliefert. Die neue Relation muss in ihrer RRT konsistent sein. Das bedeutet, die Verlinkungen müssen nach dem Löschen aller Zellen angepasst werden und weiterhin die bestehenden Tupel korrekt rekonstruierbar machen.

Das *Einfügen* eines neuen Tupels bedarf keine Positionsangabe für die Relation. Es werden jedoch die FVT und RRT überarbeitet, damit die bestehenden Tupel zusammen mit dem neuen konsistent bleiben.

Das *Bearbeiten* eines Tupels soll anhand einer Zelle die Werte identifizieren und einen oder mehrere Werte in diesem Tupel verändern. Die kann als ein Schritt innerhalb der FVT und RRT erfolgen. Es kann jedoch auch in Teilschritte durch das Löschen des alten Tupels und Einfügen des überarbeiteten Tupels erfolgen.

4.1.2.2 Relationale Algebra

Für die Anwendung der relationalen Algebra auf die TR-Technologie erklärt Chris Date die grundlegende Operationen. Dazu zählen die Restriktion, Projektion, Erweiterung, Mengenoperationen und der Verbund.

Die Restriktion gilt als großer Gewinner in der TR-Technologie. Da alle Daten sortiert gespeichert sind, können sie sehr schnell auf einzelne Bedingungen geprüft werden. Das Vergleichen von mehreren Bedingungen in einer Restriktion ist jedoch schwieriger. Dazu wird mehr in der Umsetzung dieser Operation erläutert.

Die Projektion wird grundlegend durch das Austauschen von Verlinkungen in der RRT umgesetzt. Das Ändern der Reihenfolge wird ebenfalls in der Umsetzung der Implementierung genauer beschrieben. Bei der Erweiterung-Operation wird eine Spalte zur Relation hinzugefügt. Dies ähnelt der Projektion in gewissen Punkten. Deswegen werden auch diese Details in der Implementierung erwähnt.

Die Mengenoperationen für die Vereinsmenge, Schnittmenge und Differenzmenge werden von Chris Date in der Aufzählung der relationalen Operationen betont. Aufgrund der Verwendung dieser Operationen in beispielsweise der Restriktion wird in der Umsetzung verdeutlicht.

4.2 Umsetzung

In diesem Unterkapitel wird über die Umsetzung der TR in Clojure berichtet. Diese Umsetzung erwies sich als besonders aufwendig im Vergleich zu den anderen Implementierungen. Dies begründet sich auf Details, die in der Beschreibung von Chris Date in Go-Faster[5] wenig behandelt wurden. Beschreibungen dieser Probleme finden sich vorwiegend in Kapitel 4.2.2 bei den Erklärungen einzelner Operationen.

4.2.1 Der Datentyp

Für die Abbildung der TR-Relation wurde in Clojure ein Datentyp angelegt. Durch *deftype* wurde er im Namespace *relation.transrelational.table* definiert. Aufgrund von *deftype* lassen sich Clojure-Interfaces und somit Funktionen an den Typ binden. Derzeit ist nur *clojure.lang.Counted* im Datentyp implementiert, geplant sind jedoch weitere wie beispielsweise die Sequenzierbarkeit.

Der für die TR angelegte Datentyp besteht grundlegend aus drei Komponenten: Keyorder, FVT und RRT. Die Keyorder ist ein Vektor aller in der Relation existierenden Attributnamen. Die Reihenfolge der Attribute wird zum Zeitpunkt der Erstellung der Relation anhand des ersten übergebenen Tupels entnommen. Diese Reihenfolge spielt für die interne Verwaltung der Daten insofern eine Rolle, dass sie in der RRT die Spalten beschreibt. Außerdem wird sie verwendet, wen ein oder mehrere Tupel der Relation wieder in Clojure-HashMaps umgewandelt werden.

Die FVT wird im Datentyp als PersistentArrayMap dargestellt. Jeder Eintrag repräsentiert eine Spalte, beziehungsweise ein Attribut der FVT. Schlüssel eines Eintrags ist immer der jeweilige Attributname. Der Wert eines Eintrags ist eine Sequenz von weiteren PersistentArrayMap. Aufgrund der Sortierung können bisher nur Werte gespeichert werden, dessen Datentyp das Interface *java.lang.Comparable* unterstützt. Funktionen als Wert zählen nicht dazu und führen derzeit zu Fehlern.

Innerhalb dieser Maps ist der Wert, sowie der Gültigkeitsbereich mit Ober- und Untergrenze. Das Konzept des Gültigkeitsbereich ist in Kapitel 4.1.1.3 als *kondensierte Spalten* erläutert. Eine mögliche FVT ist in Listing 4.1 abgebildet. In diesem Beispiel sind lediglich *:name* und *:city* in der FVT. Jedoch sind beide Fälle für die Gültigkeitsbereiche zu erkennen. In *:name* hat jeder Wert genau eine Zelle in der Relation. Das bedeutet, kein Name kommt mehrfach vor. Dies ist dann an einem Wert zu erkennen, wenn die Untergrenze (*:from*) und die Obergrenze (*:to*) die selbe Zeilennummer beinhalten. In der Spalte *:city* ist wiederum zu sehen, dass London und Paris jeweils zweimal in der Relation vorkommen. Dies erkennt man an die Differenz zwischen Ober- und Untergrenze.

```
1 {:name ({:value "Adams", :from 0, :to 0}
2         {:value "Blake", :from 1, :to 1}
3         {:value "Clark", :from 2, :to 2}
4         {:value "Jones", :from 3, :to 3}
5         {:value "Smith", :from 4, :to 4}),
6
7  :city ({:value "Athens", :from 0, :to 0}
8         {:value "London", :from 1, :to 2}
9         {:value "Paris",  :from 3, :to 4})}
```

Listing 4.1: Beispiel: FVT

Die RRT wird als ein Vektor von Sequenzen dargestellt. Jede Sequenz steht für eine Spalte der RRT und enthält Wertpaare in Form von Vektoren. Diese Wertepaare sind nach dem Konzept der *kondensierten Spalten* und stehen für den Verweis auf die Zeile in der FVT (erste Zahl) und den Verweis der Zeile in der nachfolgenden RRT-Spalte

(zweite Zahl). Die Reihenfolge dieser Spalten entspricht der in der Keyorder festgehaltenen Reihenfolge. Ist beispielsweise die Keyorder *[:city :name]*, so verweist die erste Sequenz auf die Spalte für das Attribut *:city* und das zweite auf *:name*. In Listing 4.2 ist dieses Beispiel für die FVT in Listing 4.1 abgebildet. Auch hier erkennt man, dass die erste Spalte teilweise auf gleiche Werte verweist. Die erste Zeile, repräsentativ für *:city* vereist doppelt auf die *1* (London) und auf die 2 (Paris).

```
[([0 0] [1 4] [1 2] [2 3] [2 1])
 ([0 0] [1 4] [2 2] [3 3] [4 1])]
```

Listing 4.2: Beispiel: RRT

Der komplette Datentyp ist entsprechend der Beispiele in Listing 4.3 zu sehen.

```
#TR Key order: [:city :name]

Field Value Table:
{:name ({:value "Adams", :from 0, :to 0}
        {:value "Blake", :from 1, :to 1}
        {:value "Clark", :from 2, :to 2}
        {:value "Jones", :from 3, :to 3}
        {:value "Smith", :from 4, :to 4}),

:city ({:value "Athens", :from 0, :to 0}
       {:value "London", :from 1, :to 2}
       {:value "Paris",  :from 3, :to 4})}

Record Reconstruction Table:
[([0 0] [1 4] [1 2] [2 3] [2 1])
 ([0 0] [1 4] [2 2] [3 3] [4 1])]
```

Listing 4.3: Beispiel: Komplette TR-Relation

Aufgrund der Beschaffenheit der Daten ist keine einfache Mechanik vorgegeben, die Duplikate innerhalb der Relation herausfiltert. Dementsprechend ist ein wichtiges Vorgehen beim Erstellen sowie Manipulieren der Relationen, mehrfach auftretende Tupel zu vermeiden. Während der Erstellung kann dies durch Voroperationen getätigt werden, insofern die Daten noch nicht im Relationsschema dargestellt werden. Operationen müssen sich durch ihre Implementation meist selbst darum kümmern, dass diese Vorgabe erfüllt bleibt.

4.2.2 Operationen

Die Implementierung der TR-Operationen erwies sich als deutlich aufwendiger im Vergleich zu denen in Kapitel 2 und 3. Die entsprechenden Implementierungen sind daher auch deutlich umfangreicher als die der bisher erwähnten Operationen.

Mehrere Details sind zu beachten, was die Implementierungen betrifft. Zum einen wurde in Kapitel 1.1.2 vorgegeben, dass eine Operation grundlegend eine Relation als Ergebnismenge liefern. Diese Vorgabe führt zu Effizienzproblemen in der Implementierung.

Als Grundvoraussetzung liefern Operationen eine neue Relation. Innerhalb der TR ist eine Relation durch FVT und RRT definiert. Das bedeutet, nach jeder Operation muss gewährleistet sein, dass RRT und FVT entsprechend der Operation geändert und konsistent sind. Das Erstellen, sowie das Anpassen und damit synchron halten der RRT und FVT kann kostenaufwendig sein. Somit entstehen aufgrund dieser Anforderung Aufwände.

Für die Implementierung der meisten Operationen gibt es zwei mögliche Herangehensweisen. Davor ist zu erwähnen, dass nicht alle Operationen von der Struktur des Datentyp profitieren. Operationen, wie beispielsweise das Lesen von einzelnen oder allen Tupeln basieren auf die TR-Technologie. Operationen, welche innerhalb der RRT und FVT durchgeführt werden, werden im Laufe dieser Arbeit auch als Aktionen innerhalb

des Formates der internen Repräsentation der Relation (kurz *innerhalb des Formats*) bezeichnet.

Einfügen und Löschen von Tupeln, sowie das Erkennen von Duplikaten sind aufwendig in der Relation durchzuführen. Daher bietet es sich in entsprechenden Implementierungen an, die Tupel zunächst alle zu rekonstruieren und die Operationen in einem anderen Datenformat durchzuführen. Nach Manipulation der Daten können die Tupel als ganzes wieder in eine TR-Relation umgewandelt werden. Solche Aktionen werden im Laufe der Arbeit auch als *außerhalb des Formats* beschrieben.

4.2.2.1 Tupel lesen

Das Lesen eines Tupels basiert auf das Traversieren durch die RRT. Dieses Traversieren wird aufgrund des Musters des entstehendes Weges *Zigzag*-Algorithmus genannt. In Listing 4.4 wird die Implementierung dazu gezeigt. Sie liefert relativ zur übergebenen Position in der Tabelle alle Verlinkungen zur FVT. Relativ bedeutet, dass die erste Verlinkung in der Ergebnismenge die aus der übergebenen Position ist.

```
1  (defn zigzag
2    [trans-table row column]
3    (let [rrt (recordReconst trans-table)
4          rrt (apply conj (vec (drop column rrt)) (drop-last  (- (count rrt) column ) rrt))]
5      (loop [rrt rrt
6             row row
7             result []]
8        (if (empty? rrt)
9          result
10         (let [[value-link next-row] (nth (first rrt) row)]
11           (recur (rest rrt) next-row (conj result value-link)))))))
```

Listing 4.4: Implementierung: zigzag

Die eigentliche Lese-Funktion ist durch *retrieve* implementiert. Diese Funktion verwendet die zigzag-Funktion, um die entsprechenden Linker aus der RRT zu erlangen. Anschließenden wird mit diesen Linkern die FVT gelesen, um die entsprechenden Werte des Tupels zusammenzutragen. Als Ergebnismenge wird eine PersistentArrayMap mit Attributnamen als Schlüsseln und den entsprechenden Werten geliefert.

```
1  (defn retrieve
2    [trans-table row column]
3    (let [orig-attrs  (keyorder trans-table)
4          attrs (apply conj (vec (drop column orig-attrs)) (drop-last (- (count orig-attrs)
               column ) orig-attrs))
5          indizes (zigzag trans-table row column)]
6      (conj {} (select-keys (into {} (map (fn [attr index][attr (fieldValueOf trans-table
           index attr)] ) attrs indizes)) orig-attrs))))
```

Listing 4.5: Implementierung: retrieve

4.2.2.2 Punkt- und Bereichssuche

Um in einer TR-Relation nach Werten in einer Spalte zu suchen, werden grundlegend in Punktsuchen und Bereichssuchen unterschieden. Jede Suchfunktion liefert alle Tupel, die den entsprechenden Voraussetzungen entsprechen, im *xrel*-Format. Dieses Format wird im weiteren Verlauf aus Performancegründen weiterverwendet. Dies wird bei der Restriktion genauer erläutert.

Die Punktsuche liefert für eine übergebene Relation, einem übergebenen Attribut und einem übergebenen Wert alle Tupel aus der Relation, welche im jeweiligen Attribut diesen Wert besitzen. Dies können je nach Attribut mehrere Tupel sein, der Wert ist jedoch in der FVT einmal (siehe Kapitel 4.2.1). Da die Werte in jeder Situation spaltenweise aufsteigend sortiert sind, eignet sich eine binäre Suchoperation, um schnell Positionen zu bestimmen. In Listing 4.6 ist die Implementierung der Funktion zu sehen.

```
(defn- point-search
  [trans-table attr value]
  (let [binary-search (fn [column value]
                (java.util.Collections/binarySearch column value #(compare (:value %1) %2)))
        found (binary-search (get (fieldValues trans-table) attr) value)]
    (if (neg? found)
      #{}
      (let [entry (nth  (get (fieldValues trans-table) attr) found)]
        (set (map #(retrieve trans-table % (.indexOf (keyorder trans-table) attr)) (range (:
            from entry) (inc (:to entry)))))))))
```

Listing 4.6: Implementierung: point-search

Für Suchoperationen, dessen Ergebnismenge aus mehreren verschiedenen Werten bestehen kann, wird die Bereichssuche verwendet. Die Bereichssuche teilt sich innerhalb der Implementierung in drei Unterkategorien. Für logische Vergleichsoperationen wie *kleiner-als* und *kleiner-gleich*, sind Suchoperationen von Vorteil, welche die Spalten von *oben nach unten* durchsucht, bis der erste Wert auftritt, welcher dem Vergleich nicht mehr entspricht. Für *größer-als* und *größer-gleich* gilt das gleiche mit einer Suche von *unten nach oben*. Für diese Anwendungsfälle wurden die Operationen *down-to-up-scan* (siehe Listing 4.7) und *up-to-down-scan* (siehe Listing 4.8) erstellt. Sie liefern alle Einträge in der Spalte der FVT, welche den jeweiligen Vergleich entsprechen. Auf höherer Instanz werden aus diesen Einträgen die Tupel erstellt.

```
(defn- down-to-up-scan
  [column f right-value]
  (loop [c column
         result []]
    (if (and (not-empty c) (f (:value (last c)) right-value))
      (recur (drop-last c) (conj result (last c)))
      result)))
```

Listing 4.7: Implementierung: down-to-up-scan

```
(defn- up-to-down-scan
  [column f right-value]
  (loop [c column
         result []]
    (if (and (not-empty c) (f (:value (first c)) right-value))
      (recur (rest c) (conj result (first c)))
      result)))
```

Listing 4.8: Implementierung: up-to-down-scan

Für eine Ungleichung werden beide genannten Bereichssuchen verwendet, um die Bereiche *vor* und *nach* dem nicht gültigen Wert zu erhalten. Dies ist in der Funktion *not=-scan* implementiert. Die Definition dieser Operation ist in Listing 4.9 einzusehen.

```
(defn- not=-scan
  [column value]
  (let [ops (if (string? value)
              [#(neg? (compare %1 %2)) #(pos? (compare %1 %2))]
              [< >])]
    (apply conj (up-to-down-scan column (first ops) value)
           (down-to-up-scan column (second ops) value))))
```

Listing 4.9: Implementierung: not=-scan

Um die richtige Bereichssuche zur übergebenen Funktion zuzuordnen, wird die Funktion *area-search* verwendet (siehe Listing 4.10). Diese Funktion ruft anhand des übergebenen Vergleichsoperators die benötigte Bereichssuche auf. Anschließend werden die Spalteneinträge der Ergebnismenge in eine xrel mit den rekonstruierten Tupeln umgewandelt. Unbekannte Vergleichsoperatoren werden durch das Liefern einer leeren Ergebnismenge verarbeitet.

```
(defn- area-search
  [trans-table attr f right-value]
  (let [up-to-down #{<= <}
        down-to-up #{>= >}
        column (get (fieldValues trans-table) attr)]
    (mapv #(retrieve trans-table % (.indexOf (keyorder trans-table) attr))
        (flatten (map (fn [n] (range (:from n) (inc (:to n))))
          (cond
```

```
9             (contains? up-to-down f) (up-to-down-scan column f right-value)
10            (contains? down-to-up f) (down-to-up-scan column f right-value)
11            (= not= f) (not=-scan column right-value)
12            :else []))))))
```

Listing 4.10: Implementierung: area-search

4.2.2.3 Duplikate eliminieren

Es werden in der Implementierung grundlegend zwei Strategien verfolgt, um Duplikate in bestehenden Relationen zu vermeiden. Dabei wird in eine *Nachuntersuchung* und eine *Voruntersuchung* unterschieden. Operationen, welche den Inhalt von Relationen beeinflussen, benötigen eine dieser Untersuchungen. Ansonsten können sich Duplikate innerhalb der Relation bilden, welche im weiteren Verlauf von Benutzung bestehen bleiben. Grundlegend wird zu jeder Operation davon ausgegangen, dass die übergebenen Relationen keine doppelten Einträge beinhalten. Als *Gegenleistung* liegt es in der Verantwortung jeder Operation, dafür zu sorgen, dass keine Duplikate in ihrer Ergebnismenge entstehen. Ausgenommen sind Operationen wie das Löschen eines Tupels. Diese Operation kann keine neuen Duplikate in einer Relation schaffen.

Für bestehende Relationen, oder welche, die als Ergebnis aus einer Operation kommen, wird die *Nachuntersuchung* durchgeführt. Das bedeutet, die Relation wird nach Duplikaten untersucht und gegebenenfalls werden diese herausgefiltert. Dieses Vorgehen eignet sich vor allem für Operationen die Projektion, da dabei die gesamte Relation verändert wird.

Das TR-Format ist nicht gut geeignet, Duplikate effektiv zu erkennen. Die Funktion zum Erkennen von bestehenden Duplikaten muss daher gegebenenfalls Tupel mindestens zum Teil wiederherstellen, um sie miteinander vergleichen zu können.

Der Algorithmus dieser Funktion sieht zunächst vor, dass die Spalten nacheinander kontrolliert werden, ob diese mehrfach vorkommende Werte besitzt. Die Implementierung der kondensierten Spalten ist für diese Suche sehr gut geeignet, da in der FVT einfach die Bereiche jedes Wertes überprüft werden müssen. Sollte es eine Spalte geben, in der dies nicht der Fall ist, kann es in der gesamten Relation keine zwei Tupel mit exakt den selben Werten geben. Gängig wäre diese Spalte ein Primärschlüssel oder etwas ähnliches. Sollte es kein Spalte mit nur einzigartigen Werten geben, wird diejenige herausgesucht, welche die *wenigsten* doppelten Werte besitzt. Alle Tupel dieser doppelten Werte werden über den Zigzag-Algorithmus zum Teil rekonstruiert und miteinander verglichen. Zum Teil bedeutet hierbei, dass die nicht die Werte der FVT gebraucht werden, sondern nur die Linker aus der RVT auf die Werte. Im direkten Vergleich werden anschießend alle Linker-Tupel miteinander verglichen, um Duplikate ausfindig zu machen. Diese Duplikate werden in der Ergebnismenge als Zeilennummern zu der zugehörigen Spalte geliefert. Das Löschen dieser Duplikate kann daraufhin an andere Stelle geschehen. Die Implementierung dieser Funktion *find-duplicates* ist in Listing 4.11 einzusehen.

```
1  (defn find-duplicates
2    [trans-table]
3    (let [column-duplicates (loop [fvt-columns (fieldValues trans-table)
4                                   result []]
5                              (if (empty? fvt-columns)
6                                result
7                                (let [column-duplicates  (filter #(not= (:from %) (:to %)) (
                                      second (first fvt-columns)))]
8                                  (if (empty? column-duplicates)
9                                  '()
10                                 (recur (rest fvt-columns)
11                                   (conj result
12                                     (map (fn [entry] (range (:from entry) (inc (:to entry)))
                                           ) column-duplicates)))))))]
13     (if (some empty? column-duplicates)
```

```
14          [0 '()]
15          (let [column-info  (map (fn [x] [ (.indexOf column-duplicates x)
16                                            (apply + (map count  x)) x])
17                                  column-duplicates)
18                few-duplicates (first (sort-by second column-info))]
19            [(first few-duplicates)
20              (flatten (map (fn [rows ] (map #(map first (rest (second %)))
21                            (group-by second
22                              (map (fn[row][row (zigzag trans-table row (first few-duplicates))])
                                    rows)))) (last few-duplicates)))])))))
```

Listing 4.11: Implementierung: find-duplicates

Um die Duplikate nun aus der Relation zu entfernen, wird die Funktion *distinct-tr* verwendet. Diese ruft von sich aus *find-duplicates* auf, um eine Liste aller Duplikate zu bekommen. Im Laufe der Entwicklung zeigte sich, dass das Löschen von Duplikaten oftmals teurer ist, als die Unikate aus der Relation herauszulesen und daraus eine neue Tabelle zu erstellen. Daher wird dies auch in der Implementierung (siehe Listing 4.12) berücksichtigt.

Diese Funktion ist aufgrund des Vorsondierens der Spalten in einigen Situationen effizienter als ein einfaches Hin- und Rückumwandeln der Relation in ein xrel-Format. Beispielsweise, wenn eine Spalte keine mehrfachen Werte beinhaltet. Für Relationen, wo dies nicht gilt, liegt der Aufwand beider Herangehensweisen bei einer ähnlichen Laufzeit.

```
1 (defn distinct-tr [trans-table]
2   (let [[column rows] (find-duplicates trans-table)]
3     (if (empty? rows)
4     trans-table
5     (let [ row-set (set rows)
6           unique-tuples (filter #(not (contains? row-set %))  (range (count trans-table)))]
7       (tr (map #(retrieve trans-table % column) unique-tuples))))))
```

Listing 4.12: Implementierung: distinct-tr

Jedoch ist die Benutzung dieser Funktion besonders bei Relation mit vielen Tupeln sehr aufwendig. Deshalb sollte sie nur sehr sparsam verwendet werden. Um für einige Operationen eine Alternative zu der Nachuntersuchung zu liefern, wurde die Voruntersuchung eingeführt.

Der Auslöser für die Implementierung der Voruntersuchung waren Performance-Tests der nachfolgenden Insert-Funktion. Durch die Nachuntersuchung wurde zu Beginn die manipulierte Relation im letzten Schritt in ein xrel-ähnliches Format umgewandelt. Gerade dieser Schritt sollte innerhalb des Einfügens nicht geschehen, da die Implementierung innerhalb des Formates effizienter ist, als außerhalb. Eine detailliertere Erklärung findet sich im nachfolgenden Kapitel. Um in solchen Funktionen Duplikate auszuschließen, wird im Vorfeld geprüft, ob die Relation bereits das einzufügende Tupel enthält. Beim Einfügen von Tupeln in eine Duplikate-freie Relation kann nur ein Duplikat entstehen, wenn das einzufügende Tupel bereits in der Relation vorhanden ist.

Um herauszufinden, ob das Tupel vorhanden ist, wurden mehrere Ansätze ausprobiert. Jeder Ansatz unterscheidet sich in der Effizienz. Da zu Beginn der Implementierung die Nachuntersuchung verwendet wurde, war auch der Aufwand der gesamten Insert-Funktion sehr hoch. Dies führte zu Optimierungsansätzen in diesem Bereich der Funktion.

Zunächst wurde eine rekursive Variante entwickelt, welche für jedes Attribut eine Punktsuche in der Relation macht. Von zwei Ergebnismengen werden in jedem Durchlauf die Schnittmenge gebildet. Sollte diese Schnittmenge leer sein, ist gewiss, dass das Tupel nicht in der Relation sein kann. Die Implementierung kann in Listing 4.13 eingesehen werden.

```
(defn- tuple-in-tr-rec
  [trans-table tuple]
  (loop [x (apply point-search trans-table (first tuple))
         tuple (rest tuple)]
    (cond
      (empty? x) false
      (empty? tuple) true
      :else (recur  (clojure.set/intersection x (apply point-search trans-table (first tuple
          )))
          (rest tuple)))))
```

Listing 4.13: Implementierung: tuple-in-tr-rec

Diese Implementierung ist besonders effektiv, wenn keines der im Tupel befindlichen Werte innerhalb der Relation vorhanden ist. Die verwendete Funktion *point-search* führt eine binäre Suche auf die übergebene Spalte mit dem dazu übergebenen Wert und liefert jedes Tupel, welches den Wert beinhaltet, innerhalb einer xrel zurück. Da die Spalten nach ihren Werten sortiert sind, ist eine binäre Suche auf die Werte sehr effizient. Die Implementierung von *point-search* ist in Listing 4.6 zu sehen.

In Listing 4.14 wird die Effizienz von der in Listing 4.13 vorgestellten *tuple-in-tr-rec* auf zehntausend Angestelltendaten aus den Beispieldatenbank von MySQL[7].

```
(def employees ...) ; TR of 1000 employees

(time (tuple-in-tr-rec employees {:emp_no 0, :birth_date "", :first_name "", :last_name "",
    :gender "", :hire_date ""})) ; 0.644795 msecs
(time (tuple-in-tr-rec employees {:emp_no 438441, :birth_date "1952-07-16", :first_name "
    Kiyokazu", :last_name "Mahmud", :gender "M", :hire_date "1985-12-15"})) ; 2522.655397
    msecs
```

Listing 4.14: Performance-Test: tuple-in-tr-rec

In Zeile 3 wird die Relation gegen ein Tupel geprüft, welches keine übereinstimmenden Werte besitzt. Dementsprechend wird die Rekursion im ersten Durchlauf abgebrochen und *false* wird zurückgegeben. Anhand der Laufzeit ist zu erkennen, dass dieser Fall sehr performant ist. In Zeile 4 wird die Relation gegen ein Tupel getestet, welches existiert. Hierbei erkennt man den Nachteil dieser Implementierung: Punktsuchen für jedes Attribut, sofern das Tupel existiert. Sozusagen ist für diese Funktion das Finden eines in der Relation existierenden Tupels der ineffizienteste Fall. Es werden zudem nicht nur für jede Spalte eine Punktsuche geführt. Die Punktsuche kann zudem mehrere Tupel liefern, welche zunächst erstellt werden müssen.

tuple-in-tr-rec wurde weiter als Vorlage für eine Funktion genutzt, dessen effiziente und ineffiziente Fälle nicht so weit auseinander klaffen. Hierzu werden nicht auf alle Attribute eine Punktsuche gemacht. Lediglich das erste Attribut des Tupels wird in der Relation gesucht. Anschließend wird durch Clojure-Mechanismen geprüft, ob das (vollständige) Tupel in der Ergebnismenge der Punktsuche vorhanden ist. In Listing 4.15 ist die Implementierung dieser Funktion zu sehen.

```
(defn- tuple-in-tr-not-rec
  [trans-table tuple]
  (let [x (apply point-search trans-table (first tuple))]
    (contains? x tuple)))

(time (tuple-in-tr-not-rec employees
      {:emp_no 0,
      :birth_date "",
      :first_name "",
      :last_name "",
      :gender "",
      :hire_date ""})) ; 0.651453 msecs

(time (tuple-in-tr-not-rec employees
      {:emp_no 438441,
      :birth_date "1952-07-16",
      :first_name "Kiyokazu",
      :last_name "Mahmud",
      :gender "M",
      :hire_date "1985-12-15"})) ; 1.332866 msecs

(time (tuple-in-tr-not-rec employees
      {:gender "M",
      :hire_date "1985-12-15",
      :emp_no 438441,
      :birth_date "1952-07-16",
```

```
    :first_name "Kiyokazu",
    :last_name "Mahmud",})) ; 2398.635586 msecs
```

Listing 4.15: Implementierung und Performance-Test: tuple-in-tr-not-rec

In Listing 4.15 sind außerdem die Laufzeiten für die gleichen Abfragen wie in Listing 4.13 zu sehen (Zeile 6 und 8). Es ist zu erkennen, dass ein existierendes Tupel deutlich schneller erkannt wird, als in der vorherigen Implementierung. Die Zeiten liegen in einem ähnlichem Rahmen.

Jedoch war auch in dieser Implementierung ein Kniff zu finden. Da die Funktion nach dem erstem Attribut im Tupel sucht, beeinflusst die Reihenfolge der Tupel auch die Laufzeit der Funktion. In Zeile 10 wurde das Tupel als Zeile 8 so angeordnet, dass *:gender* das erste Attribut ist. *:gender* enthält in verwendeten Testdaten lediglich zwei verschiedene Werte. Der übergebene Wert *M* kommt in ca. 60% der Tupel in der Relation vor (6019 der 10000 Einträge). Was bedeutet, dass die Punktsuche auch jedes dieser Tupel rekonstruiert. In der Punktsuche aus Zeile 8 wird nach *:emp_no* gesucht. Diese sind in der Relation eindeutig und somit wird an dieser Stelle nur *ein* Tupel rekonstruiert, was den zeitlichen Unterschied beider Suchen erklärt.

Um auch diese Ineffizienz zu eliminieren, wird eine Hilfsfunktion eingeführt, welche in der Relation die Spalte mit der höchsten Werteanzahl identifiziert. Je höher die Anzahl an verschiedenen Werten in einer Spalte liegt (im Vergleich zu den anderen Spalten), desto geringer ist Wahrscheinlichkeit, dass mehrere Tupel diesen Wert besitzen. Demnach können Punktsuchen auf diese Spalten, beziehungsweise Attribute gezielt verwendet werden, um die Laufzeit der Tupelsuche zu minimieren. Die Hilfsfunktion *get-most-present-attr* sowie die Anpassung an *tuple-in-tr-not-rec* ist in Listing 4.16 zu sehen.

```
1   (defn- get-most-present-attr
2     [tr]
3     (let [counts (map #(first (last %)) (recordReconst tr))]
4       (get (keyorder tr) (.indexOf counts (apply max counts)))))
5
6   (defn- tuple-in-tr-not-rec
7     [trans-table tuple]
8     (let [mpa (get-most-present-attr trans-table)
9           x (point-search trans-table mpa (get tuple mpa))]
10      (contains? x tuple)))
11
12
13  (time (tuple-in-tr-not-rec employees
        {:emp_no 0,
        :birth_date "",
        :first_name "",
        :last_name "",
        :gender "",
        :hire_date ""})) ; 2.571497 msecs
14
15  (time (tuple-in-tr-not-rec employees
        {:emp_no 438441,
        :birth_date "1952-07-16",
        :first_name "Kiyokazu",
        :last_name "Mahmud",
        :gender "M",
        :hire_date "1985-12-15"})) ; 4.339434 msecs
16
17  (time (tuple-in-tr-not-rec employees
        { :gender "M",
        :hire_date "1985-12-15",
        :emp_no 438441,
        :birth_date "1952-07-16",
        :first_name "Kiyokazu",
        :last_name "Mahmud",})) ; 3.04165 msecs
```

Listing 4.16: Implementierung und Performance-Test: get-most-present-attr und tuple-in-tr-not-rec

Anhand der Laufzeit-Beispielen in den Zeilen 13-17 ist zu erkennen, dass die Laufzeit der vorherigen Implementierungen in den effizienten Fällen nicht mehr erreicht werden. Aufgrund der Vorsuche nach dem MPA (*most present attribute*) werden die Suchen nach dem gleichen Attribute nun um diese Laufzeit erweitert. Jedoch ist die Differenz der Laufzeiten für die effiziente Fälle in einem vertretbaren Rahmen und können vernachlässigt werden. Die Laufzeit des ineffizienten Falls ist dafür nun im gleichen

zeitlichem Rahmen. Somit klaffen die verschiedenen Fälle nicht mehr auseinander. Die Reihenfolge der Attribute in dem Tupel beeinflussen nicht weiter die Laufzeit. Die Funktionen in Listing 4.16 wurden als Voruntersuchung für die TR-Implementierung, also für die nachfolgenden Operation, übernommen.

4.2.2.4 Einfügen

Das Einfügen von neuen Tupeln in die bestehende Relation kann auf zwei Arten angegangen werden. Wichtig ist bei beiden Herangehensweisen, dass nach der Operation keine Duplikate entstehen.

Zum einen besteht die Möglichkeit in unserer Implementierung, dass das Einfügen *außerhalb* des Formats erfolgt. In einem einfacheren Format kann das einzufügende Tupel einfacher in die bestehende Menge eingeschlossen werden. Beispielsweise das *xrel* Format, wie in der Implementierung aus Kapitel 2. Da dieses Format bei der Erstellung einer TR-Relation als Übergabewert unterstützt wird, und das Einfügen von Tupeln nicht aufwendig ist, ist dieser Vorgang durchaus anwendbar. Ein Vorteil dieser Herangehensweise ist, dass die *xrel* von ihrer Implementierung her Duplikate herausfiltert. Eine mögliche Implementierung ist in Listing 4.17 einzusehen.

```
1 (defn alter-insert
2   [table tuple]
3   (let [converted   (convert table)
4         manipulated (conj converted tuple)]
5     (tr manipulated)))
```

Listing 4.17: Implementierung: alter-insert

Als Alternative kann das Einfügen *innerhalb* des TR-Formats erfolgen. Hierbei ist zu beachten, dass FVT und RRT bearbeitet werden müssen. Ein möglicher Algorithmus zum Einfügen eines Datensatzes wird im folgenden erklärt:

1. Für jedes Attribut a_i im einzufügenden Tupel t die erst mögliche, richtige Position p_i in der zugehörigen Spalte i der FVT finden. Die richtige Position setzt eine bestehende Sortierung voraus.
2. Wert für Attribut a_i in Spalte i in der FVT einfügen. Nachfolgende Werte wandern eine Position weiter.
3. In der RRT für alle Spalten i an Position p_i den Zahlenwert $p_{(i+1)modn}$ einfügen. Nachfolgende Werte wandern eine Position weiter.
4. In der RRT für alle Spalten i jeden Zahlenwert w_{ij} um 1 erhöhen, wenn er nicht an Position p_i steht und der Zahlenwert von w_{ij} größer oder gleich dem Zahlenwert in p_i ist.

Der genannte Algorithmus wurde für TR-Relation ohne Optimierung erstellt. In der nachfolgenden Implementierung sind Abweichungen vorhanden, da die Optimierung für kondensierte Spalten im Datenschema mit integriert wurde. Die Abweichung liegt dabei, dass nicht einfach der Wert in die FVT eingetragen wird. Sollte der Wert bereits in der FVT in der jeweiligen Spalte vorhanden sein, muss nur der *to*-Zähler des Wertes erhöht werden.

Um Duplikate in der Operation zu vermeiden, wird anhand einer Voruntersuchung geprüft, ob das einzufügende Tupel bereits in der Relation vorhanden ist (siehe Kapitel 4.2.2.3). Sollte dies der Fall sein, wird das Einfügen ausgelassen und die übergebene Relation wird als Ergebnismenge zurückgegeben.

Sollte das Tupel nicht dem Schema der Relation entsprechen, wird von der Operation eine entsprechende Fehlermeldung geworfen.

Wichtig war es, dass die Implementierung keinerlei Teilschritt besitzt, indem eine TR-Relation erstellt wird, die nicht das Endresultat ist. Innerhalb der Entwicklung wurden andere Methoden genutzt, die diese Mechanik verwendeten. Dies führt zu einem immensen Aufwand der Einfüge-Operation. Weiter gilt dies auch für andere Implementierungen in der TR.

Zum Vergleich wurden beide Varianten der Implementierung bei Relationen mit kleinen und großen Datenmengen verwendet. Als *insert* ist der oben genannte Algorithmus definiert. Die direkten Vergleiche, und ihre Laufzeiten, sind in Listing 4.18 einzusehen. Die für die Tests verwendeten Relationen beinhalten 10 (*some-tuples*, Zeilen 1 und 4) und 10000 (*many-tuples*, Zeilen 2 und 5) Tupel.

```
1 (time (insert some-tuples tuple)) ; 2.136222 msecs
2 (time (alter-insert some-tuples tuple)) ; 1.613731 msecs
3
4 (time (insert many-tuples tuple)) ; 20.15577 msecs
5 (time (alter-insert many-tuples tuple)) ; 71262.238534 msecs
```

Listing 4.18: Vergleich: insert und alter-insert

Wie zu sehen, unterscheiden sich die Laufzeiten bei kleinen Tupelmengen nicht besonders von einander. Das liegt größtenteils daran, dass bei so weniger Datenmenge die einzelnen Schritte beider Operationen keinen großen Aufwand besitzen, weil kaum Daten zu lesen und zu verändern sind.

Bei größeren Datenmengen driften die Zeiten deutlich von einander ab. Die *insert* Funktion, welche *innerhalb* des Formats operiert, zeigt dabei, dass sie die performantere Variante ist. Da alter-insert zunächst alle 10000 Tupel rekonstruiert und sie anschließend in eine TR umwandelt, liegt der Aufwand in der Verarbeitung der gesamten Daten. *insert* manipuliert lediglich betroffene Daten, welche in der Tabelle verschoben sind, ohne in Zwischenschritten eine TR-Relation zu erstellen. Somit erklärt sich der deutliche Unterschied in den Laufzeiten.

4.2.2.5 Löschen

Die Operation zum Entfernen eines Tupels tut dies anhand der Position eines der Werte in der Relation. Dieses und alle zugehörigen Werte werden aus der Relation entfernt.

Auch hier in dieser Implementierung lassen sich Herangehensweisen sowohl innerhalb als auch außerhalb des Formats anwenden. Jedoch wurde aufgrund der Erfahrung aus der Implementierung der Einfüge-Operation eine Variante außerhalb des Formats vernachlässigt.

Für die Operation innerhalb des Formats wird, ähnlich wie bei der Einfüge-Operation, ein Algorithmus für die Manipulation der FVT und RRT vordefiniert. Der folgende Algorithmus ist für eine TR-Relation ohne Optimierungen definiert. Für die Verwendung der kondensierten Spalten müssen Veränderungen beim Löschen der Daten in der FVT vorgenommen werden.

1. Anhand der übergebenen Position eines Wertes des Tupels für alle Spalten i die Position p_i des Linkers in der RRT ausfindig machen.

2. Für alle Spalten i in der FVT den Wert an der Position p_i entfernen.

3. Für alle Spalten i in der RRT für alle Werte w_{ij} den Wert dekrementieren, sofern gilt: $w_{ij} > w_{p_i}$, wobei w_{p_i} für den Wert an Position p_i steht.
4. Für alle Spalten i in der RRT den Wert w_{p_i} an Position p_i entfernen.

Für die Verwendung der Optimierung *kondensierte Spalten* muss in der Modifizierung der FVT in Schritt 2 eine Änderung vorgenommen werden. Da sich mehrere Tupel den Wert-Eintrag teilen könnten, wird in zwei Fälle unterschieden: Der Wert wird nur vom zu löschenden Tupel verwendet, oder der Wert wird von weiteren Tupeln verwendet. Dies kann an den Zahlenwerten des Gültigkeitsbereiches erkannt werden. Sind die Ober- und Untergrenze des Bereichs identisch, so wird der Wert nur von einem Tupel verwendet. In diesem Fall wird der Eintrag ganz entfernt. Ansonsten wird nur der Zähler für die Untergrenze dekrementiert. Außerdem müssen in beiden Fällen die Zahlenwerte aller nachfolgenden Einträge dekrementiert werden, da sich alle Werte um eine Position nach oben verschieben.

4.2.2.6 Editieren

Das Editieren bedarf in der TR die Position eines Wertes des jeweiligen Tupels. Zusätzlich werden ein oder optional mehrere Attribute gepaart mit den neuen Werten benötigt, um die Änderungen vorzunehmen.

In einigen Datenbankmanagementsystemen wird die Mechanik des Editierens durch das Löschen der ursprünglichen Zeile und Einfügen des editierten Datensatzes abgebildet. Diese Mechanik wird in dieser Implementierung ebenfalls durchgeführt. In Listing 4.20 ist die Definition der Funktion *update* einzusehen.

```
1 (defn update
2   [trans-table  row column update-map]
3   (when (not-any? #(contains? (set (keyorder trans-table)) %) (keys update-map))
4     (throw (IllegalArgumentException. "Update map contains illegal attribute.")))
5   (let [old-row (retrieve trans-table row column)
6         new-row (merge old-row update-map)]
7     (insert (delete trans-table row column) new-row) ))
```

Listing 4.19: Implementierung: update

4.2.2.7 Projektion

Projektionen in der relationalen Algebra wird als Abbildung einer Relation auf weniger Attribute verwendet. Innerhalb der TR-Technologie bedeutet dies, dass eine oder mehrere Spalten innerhalb einer TR-Relation entfernt werden. Innerhalb der Keyorder, sowie der FVT ist diese Operation kein besonderer Aufwand, da die Spalten, beziehungsweise die Attribute einfach weggelassen werden können.

Die Hauptaufgabe für diese Operation ist das Umwandeln der RRT, sodass die Tupel weiterhin konsistent bleiben. Dafür erden alle Spalten, welche *vor* einer zu entfernenden liegen, angepasst. In Abbildung 4.2 werden aus dem bekannten Beispiel die Attribute SNAME und STATUS entfernt. S# liegt *vor* diesen beiden Attributen. Da sich zwischen den beiden zu entfernenden Attributen kein weiteres befindet, übernimmt S# die Änderung für beide Attribute.

Grundlegend gilt folgende Regel für das Bearbeiten einer Spalte in der RRT: Für ein Attribut, welches sich vor einer zu löschenden Menge von Attributen befindet, muss jeder Attribut-Linker durch denjenigen ersetzt werden, zu den er in der letzten zu löschenden Spalte traversiert. In Abbildung 4.2 wird dies im linken Bild an einer Zeile

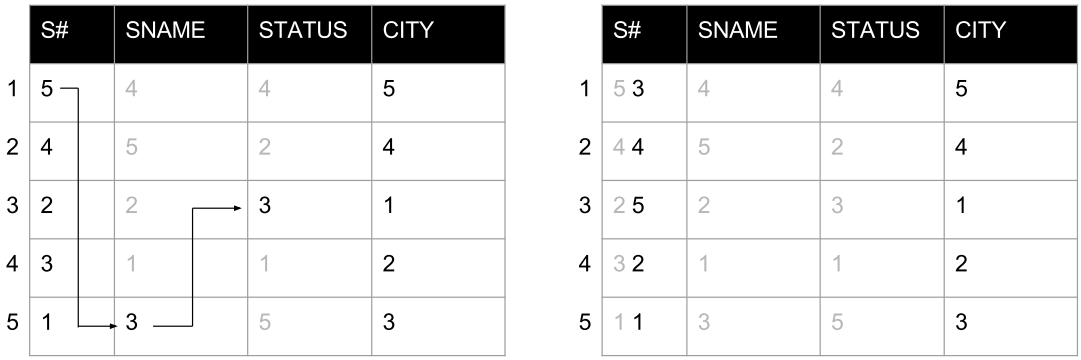

Abbildung 4.2: Beispiel für das Ersetzen der Einträge in der Projektion

aufgezeigt. Die erste Zeile in S# traversiert für diese zu löschende Spalte (SNAME und STATUS) einen Schritt weiter in der RRT. Der Wert in der letzten zu löschenden Spalte wird als neuer Wert in S# übernommen, damit das Tupel instand bleibt.

Dieser Vorgang wird zunächst für alle Zeilen in der zu bearbeitenden Spalte übernommen. Es kann vorkommen, dass durch eine Projektion mehrere Spalten auf diese Art bearbeitet werden müssen. Beispielweise wenn zwei nicht aufeinanderfolgende Spalten entfernt werden.

Sobald die RRT und auch die anderen Komponenten der Relation bearbeitet wurden, ist eine Nachuntersuchung auf Duplikate notwendig. Projektionen können dazu neigen, viele Duplikate aus der Operation heraus zu erstellen. Da vor oder während der Operation keine effiziente Kontrolle möglich ist, wird die Nachuntersuchung zum Schluss verwendet.

Das Erweitern einer Relation (*extend*), also das Hinzufügen einer Spalte, wird durch das Umwandeln in das xrel-Format gelöst. Das zu übergebene Prädikat, welches die Werte für die neue Spalte vorgibt, kann sich auf andere Werte des Tupels beziehen und somit nur mit rekonstruierten Tupel ausgewertet werden. Da dies für jedes Tupel gilt, wird die Relation als ganzes für das Einfügen von neuen Spalten umgewandelt.

4.2.2.8 Mengenoperation

Die Mengenoperationen für die Vereinigungsmenge, Schnittmenge und Differenzmenge wurden durch das Umwandeln in xrel und größtenteils durch das Verwenden ihrer *clojure.set*-Äqivalente implementiert.

```
1  (defn union
2    [tr1 & more]
3    (tr (flatten (apply conj  (convert tr1) (map convert more)))))
4
5  (defn intersection
6    [tr1 & more]
7    (tr (apply clojure.set/intersection (set (convert tr1)) (map #(set (convert %)) more))))
8
9  (defn difference
10   [tr1 & more]
11   (tr (clojure.set/difference (set (convert tr1))  (map #(set (convert %)) more))))
```

Listing 4.20: Implementierung: union, intersection und difference

4.2.2.9 Restriktion

Die Restriktion gilt als der Vorteil im TR-Model. Aufgrund der sortierten Spalten sollen einzelne Werte und ihre Tupel schnell auffindbar und erreichbar sein. Für die Verwendung von Prädikaten repräsentativ für Vergleichsoperationen und ähnliches bedarf es jedoch einer weit gedehnten Herangehensweise. Die Spalten-orientierte Darstellung der Daten eignet sich nicht für eine verschachtelte Bedingung, welche mehrere Attribute mit einschließt.

In Listing 4.21 ist ein mehrfach verschachteltes Prädikat als Beispiel einzusehen. Zu erkennen ist, dass das Sondieren von *einer* Spalte nicht ausreicht, um auf die korrekte Ergebnismenge zu kommen. Jedoch lassen sich die Teilschritte innerhalb des Prädikats ablesen: Zeile 1 beinhaltet eine Bereichssuche für das Attribut :status, Zeile 2-3 je eine Punktsuche auf das Attribut :city. Die Funktionalität findet sich bereits in Kapitel 4.2.2.2.

```
1 (fn [t] (and (>= 30 (:status t))
2               (or (= (:city t) "Paris")
3                   (= (:city t) "Athens"))
```

Listing 4.21: Beispiel-Prädikat

In Kapitel 2.2.2 wurde bereits auf das Verwenden von Clojure-Makros zum Speichern und gegebenenfalls Optimieren von benutzerdefinierten Prädikaten erwähnt. In der Implementierung von Kapitel 2 wurde bisher keine Optimierung für den Prädikat-Code vorgenommen, jedoch findet sich diese Idee in dieser Implementierung.

Die benutzerdefinierten Prädikate für die Restriktion werden in dieser Implementierung in dem Moment verändert, in dem sie erzeugt werden. Die Veränderung des abstrakten Syntaxbaumes sollen bewirken, dass das Prädikat direkt auf die Relation angewendet werden kann, ohne dass verschachtelte Bedingungen ein Hindernis in der Ausführung darstellen.

Wertevergleiche, wie die bereits angesprochenen Teilschritte in Listing 4.21, werden durch entsprechende Suchoperationen ausgetauscht. Dies führt dazu, dass aus in diesen Stellen im Prädikat von einer xrel als Ergebnismenge ausgegangen werden kann. Weiter kann davon ausgegangen werden, dass alle Tupel in dieser xrel die Bedingung erfüllen, die ursprünglich an der jeweiligen Stelle im Prädikat war.

Das Ersetzen der Wertevergleiche geschieht bisher nach folgendem Schema:

- **and**-Operationen werden in **clojure.set/intersection** übersetzt. Somit bildet sich eine Schnittmenge aus den (in xrel-Format abgebildeten) Operanden (siehe auch Abbildung 4.3).
- **or**-Operationen werden in *clojure.set/union* übersetzt. Damit werden die Operanden in eine Vereinsmenge zusammengefasst.
- =-Operatoren, also **Gleichheitsprüfungen**, werden auf die Punktsuche aus Kapitel 4.2.2.2. Rückgabe dieser Operation ist eine Menge von Tupeln, die den übergebenen Wert enthalten, im xrel-Format.
- Ungleichheitsprüfungen werden als Bereichssuchen abgebildet (siehe auch Abbildung 4.3). Hierbei wird abhängig von der Art der Operation eine bestimmte Variante der Bereichssuche verwendet. Die Zuordnung geschieht folgendermaßen:
  - < und <=, beziehungsweise die **kleiner-** und **kleiner-gleich**-Operationen, werden durch die *up-to-down-scan*-Funktion abgebildet.

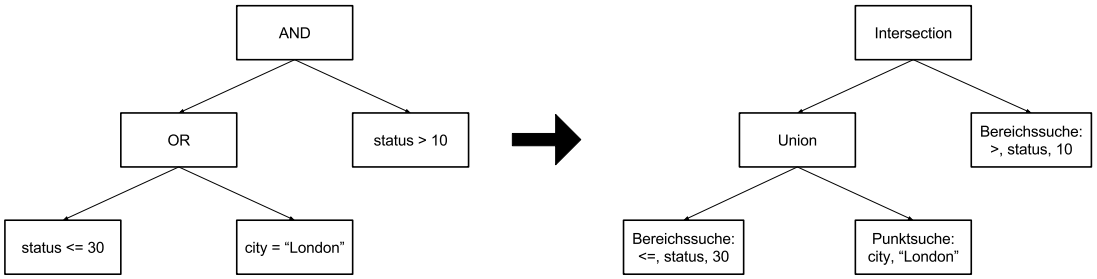

Abbildung 4.3: Beispiel für Umformen von Vergleichen und Verbunde im abstrakten Syntaxbaumes

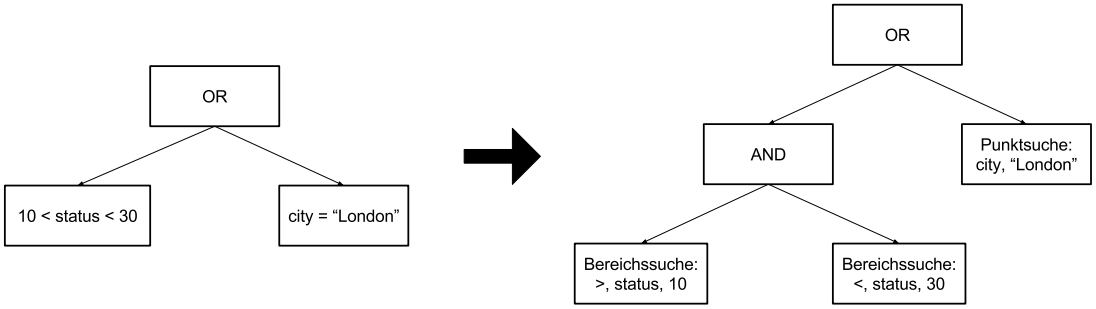

Abbildung 4.4: Beispiel für Umformen vom Operationen mit drei Operanden im abstrakten Syntaxbaumes

  - > und >=, beziehungsweise die **größer-** und **größer-gleich**-Operationen, werden durch die *down-to-up-scan*-Funktion abgebildet.

  - **not=**, also die direkte **Ungleichheit**, wird durch die Vereinigung von einer *kleiner-* und *größer*-Vergleichsoperation abgebildet. Hierzu werden die Vergleiche von String- und Zahlenwerten verschieden behandelt.

- Vergleiche, welche mehr als zwei Operanden besitzen, werden in mehrere Unteroperationen aufgeteilt und die Ergebnismengen zu einer Vereinsmenge zusammengefasst (siehe auch Abbildung 4.4).

- Zur Vereinheitlichung und Vereinfachung der Optimierung werden die Relationswerte als erster Operand und der Operand als zweiter Wert erwartet. Ist dies nicht gegeben, so werden sowohl die Reihenfolge der Operanden, als auch die Relation, *gedreht*. Beispielsweise wird eine **größer**-Operationen in eine **kleiner**-Operation übersetzt (siehe auch Abbildung 4.4).

- Zudem werden Vergleichsoperationen, dessen Operanden mehr als ein Attribut des Tupels enthalten, durch einen *inneren Vergleich* (inner-Compare) abgebildet.

Der zuletzt erwähnte Vergleich für mehrere Attribute wird durch die Methode *inner-compare* umgesetzt. Da das TR-Model durch seine Struktur nicht geeignet für eine solche Operation ist, ist diese sehr aufwendig und benötigt möglichst viele Vereinfachungen.

Die Priorität liegt dabei, dass diese Vergleiche mit möglichst wenig Rekonstruktion durchgeführt werden. Zu Beginn wird für beide Operanden geprüft, ob in der jeweils anderen Attribute-Spalte ein Wert existiert, auf den der Vergleich zutrifft. Alle Werte von beiden Attributen, wo kein mögliches Ziel gefunden wurde, werden für die nächsten

Schritte übergangen. Im nächsten Schritt werden die Distanzen der beiden Attribute verglichen. Dies dient dazu, dass beim Traversieren der Weg mit den wenigsten Schritten verwendet wird. Anschließend werden für jedes noch vorhandene Tupel auf kürzesten Weg die Attribute durch Traversieren verbunden und die Vergleichsoperation wird angewandt. Alle Tupel, auf die dieser Vergleich ebenfalls zutrifft, werden in der Ergebnismenge aufgeführt.

5 Vergleichende Betrachtung und Ausblick

In diesem Kapitel wird zusammenfassend der bisherige Entwicklungsstand betrachtet. Dazu gehören (bisher mögliche) Vergleiche zwischen den drei Konzepten der relationalen Algebra, sowie ihre Implementierung in Clojure. Außerdem werden Fazite zu den Implementierungen gezogen, sowie Ausblicke auf Erweiterungen und Optimierungen dargestellt.

5.1 Vergleich

Um alle drei Implementierungen direkt miteinander vergleichen zu können, müssen äquivalente Funktionen gegenübergestellt werden. Aufgrund der unvollständigen Implementierung vom Transrelational Model beschränkt sich dies auf die Erstellung von Relationen, die direkte Manipulation von Tupeln und die Restriktionen. Die in Tabelle 5.1 aufgeführten Laufzeiten wurden an Relationen mit den gleichen Tupeln ausgeführt. Die Anzahl der in den Relationen enthaltenden Tupeln liegt bei 10000. Jedes Tupel besteht aus sechs Attributen. Für das Erstellen der jeweiligen Relation wurde immer die selbe Vorlage im xrel-Format verwendet. Somit kann vorausgesetzt werden, dass Ergebnismengen (abgesehen von Reihenfolgen) identisch sind.

Für jede Operation, abgesehen von dem Erstellen einer Relation (in Tabelle unter *Relation erstellen*) wurde die gleiche Relation als Ausgangswert verwendet. Die Bereichssuchen wurden gekennzeichnet (in Klammern aufgeführte Zahl), wie groß die Ergebnismenge der jeweiligen Suche war.

Anhand der Laufzeiten in Tabelle 5.1 ist deutlich zu erkennen, dass die HashRel-Implementierung aus Kapitel 2 sich stark von den anderen abhebt. Die Einfachheit der Implementierung mit den Verwenden von Clojure-Mechaniken bildet eine deutlich performantere Implementierung.

Die BAT-Algebra bildet das Mittelfeld in diesen Vergleichen. Deutlich sind die längeren Zeiten für das Löschen und Einfügen im Vergleich zu den anderen Implementierungen. Die Suchoperationen sind zeitlich jedoch weit unter denen des TR-Models.

Die Implementierung des TR-Models aus Kapitel 4 schneidet im Vergleich zu den anderen als deutlich ineffizienter ab. Das Einfügen, sowie das Löschen eines Tupels schneiden als schnellste Aktion in den Tests ab. Als große Stärke des TR-Model gilt jedoch eine schnelle Suche wegen sortierter Spalten. Das Auffinden der jeweiligen Werte in den Spalten ist aufgrund der binären Suche und der Einmaligkeit der Werte auch vergleichsweise schnell. In Tabelle 5.2 wird aufgeführt, wie sich die Laufzeiten bei einer Bereichssuche verhalten. Die Suche entspricht der ersten Bereichssuche aus Tabelle 5.1. Da in dieser Suche auf die Gleichheit eines Wertes geprüft wird (Geschlecht), verhält sich die Suche in der TR wie eine Punktsuche. Zu sehen ist, dass das Rekonstruieren der Tupel (Anzahl: 3981) den Großteil der gesamten Laufzeit einer Suche ausmacht.

	HashRel	BAT-Algebra	TransRelational Model
Relation erstellen	0.12 ms	9844.35 ms	14487.01 ms
Einfügen	0.79 ms	200.16 ms	9.07 ms
Löschen	0.23 ms	242.90 ms	4.2 ms
Punktsuche	1.81 ms	109.19 ms	16756.49 ms
Bereichssuche (einfache Bedingung) (3981)	7.36 ms	205.93 ms	20223.07 ms
Bereichssuche (Verbund zweier Bedingungen) (1)	5.14 ms	141.26 ms	29755.04 ms
Bereichssuche (Verbund vier Bedingungen) (2)	3.82 ms	101.49 ms	11921.54 ms

Tabelle 5.1: Übersicht von Laufzeiten

Funktion	Zeiten
Reine binäre Suche	2.80 ms
Suche und Rekonstruktion	19370.60 ms
Suche, Rekonstruktion und Relation erstellen	20223.07 ms

Tabelle 5.2: Übersicht von Laufzeiten einer Bereichssuche in der TR-Implementierung

Das Löschen und Einfügen basiert grundlegend auf das binäre Suchen, jedoch weniger auf das Rekonstruieren. Daher weichen die Laufzeiten der Manipulationen und der Suchen (Punktsuche und Bereichssuche) voneinander ab.

5.1.1 HashRel

Aufgrund der Vorlagen von *core.relational* und *Incanter* war die Implementierung in Kapitel 2 weniger aufwendig im Vergleich zu den anderen Implementierungen. Grundlegende Züge und Mechanismen konnten aus einer der beiden Quellen übernommen und an dem Datentyp angepasst werden. Somit wurde auch der umfangreichste Funktionssatz für die Verwendung der Relationen erstellt. Die Implementierungen anderen Relationen waren aufgrund der rein theoretischen Vorlage komplizierter zu erstellen.

Die HashRel-Relation ist aufgrund ihrer Einfachheit und der vielfachen Verwendung von Clojure-Mechaniken im Vergleich die performanteste Implementierung. Die Suchen, beziehungsweise Restriktionen, sind als Prädikate und Filteroperationen in Clojure abgebildet. Die Laufzeiten in Tabelle 5.1 befinden sich bei 10000 Tupeln, auch unter Berücksichtigung von *Lazy Sequences*, im ein-stelligen Millisekundenbereich. Die Laufzeiten der anderen Implementierungen fangen bei gleichen Operationen erst im drei-stelligen Bereich an.

Jedoch sind die Laufzeiten nicht repräsentativ für andere Operationen. Verbünde und Gruppierungen zählen zu den teureren Operationen. Aufgrund der unvollständigen Implementierung des TR-Models kann dort bisher kein Vergleich gezogen werden.

Funktion	Zeiten
Punktsuche	3.75 ms
Punktsuche und Ergebnismenge mit 2 Attributen	22.85 ms
Punktsuche und Ergebnismenge mit 4 Attributen	43.91 ms
Punktsuche und gesamte Ergebnismenge (6 Attribute)	109.19 ms

Tabelle 5.3: Übersicht von Laufzeiten einer Punktsuche in der BAT Implementierung

5.1.2 Binary associated table Algebra

Die *binary associated table* Algebra operiert in kleinen, einzelnen Schritten. Operationen der relationalen Algebra müssen daher auf diese Operationen abgebildet werden. Die Ergebnisse dieser Operationen werden weiterverwendet oder zum Endergebnis verbunden.

Für diese Operationen bedarf es einer Fragmentierung der Eingangsdaten. Das Erstellen dieser Menge an Relationen ist deutlich zeitaufwendiger, als die der HashRel (siehe Tabelle 5.1). Grund dafür ist, dass die HashRel lediglich die Eingangsdaten im xrel-Format kopiert. Die Fragmentierung für das BAT-Format bedarf einer Bearbeitung und ist daher deutlich aufwendiger.

Die in Tabelle 5.1 aufgezeigten Suchen bezeichnen bei den BAT-Varianten das Manipulieren der entsprechenden BATs und das anschließende Verbinden zu einer Ergebnismenge im xrel-Format. Dieser letzte Schritt ist dabei der aufwendigste. In Tabelle 5.3 wird aufgezeigt, dass die Gesamtlaufzeit von der Anzahl der darzustellenden Attribute abhängt. Grund dafür ist, dass die Suchoperation selbst nur ein minimaler Aufwand ist, für das Darstellen der Tupel aber mit jedem Attribut zusätzlicher Aufwand im Verbinden der fragmentierten Spalten besteht.

Die weiteren Laufzeiten für Suchoperationen in Tabelle 5.1 sind dadurch ebenfalls. In der einfachen Bereichssuche wird eine größere Anzahl an Tupel zurückgegeben. Da die Ergebnismenge für jedes Attribut einen Verbund durchführt, werden diese durch die große Menge an Zeilen sehr aufwendig.

Für die Weiterentwicklung dieser Implementierung drängt sich eine Übersetzung von relationalen Operationen in die BAT Algebra förmlich auf. Somit wäre es einfacher, diese Implementierung als Benutzer zu bedienen. Weiter können in dieser Übersetzung Optimierungen im Verlauf von größeren Operationen getätigt werden. Auch Caching-Verfahren bieten sich in dieser Implementierung an, da Zwischenergebnisse der BAT-Operationen wiederverwendbar sind.

5.1.3 Transrelational Model

Das Implementieren des Transrelational Models stellte sich als der größte Aufwand in diesem Projekt heraus. Die Beschaffenheit der Datenhaltung und das Operieren mit dieser ist eine Detailarbeit. Die Operationen sind aufwendig zu entwickeln und enthalten ein großes Maß an Zwischenergebnissen, welche in Clojure aufgrund der *Immutable Data Structure* alle (zumindest zeitweise) erstellt werden und im Hauptspeicher bestehen. Insbesondere in größeren Relationen, für welche das TR-Model von Vorteil sein

soll, führt dies zu einem großen Speicheraufwand sowie einer schlechten Laufzeit der Operationen.

Die derzeitige Implementierung ist aus zeitlichen Gründen nicht vollständig. Zu dem noch offenen Punkten der Implementierung zählen unvollständige Stellen in der Restriktionsoptimierung, sowie ganze Operationsbereiche wie Verbünde, Gruppierung und RelVars. Zudem wäre ein Vergleich insbesondere dieser Funktionen zu den anderen Implementierungen notwendig, und vor allem aufschlussreich für die Verwendbarkeit des TR-Models in Clojure.

Zur Optimierung der Laufzeit der bisherigen Funktionalität könnten Teile der Grundfunktionen, sowie der Datentyp selbst, überarbeitet werden. Möglicherweise wurden einzelne Operationen in der Entwicklung bereits zu komplex abgebildet, sodass die für den Benutzer freigegebenen Operationen unter der Ineffizienz leiden.

5.2 Fazit

Nach der bisherigen Projektzeit wurde der in diesem Bericht dargestellte Stand erstellt. Es zeigte sich im Vergleich, dass eine einfache Implementierung, welche mehr auf Clojure-Mechaniken und weniger auf komplexere theoretische Modelle basieren, deutlich effizientere Laufzeiten aufweist. Der Quellcode von der HashRel-Implementierung ist daher wesentlich geringer als beispielsweise der des Transrelational Models. Deswegen, und auch wegen der bereits in Clojure-Code existierenden Vorlagen, war die Entwicklung dieser Variante weniger zeitintensiv.

Für detailliertere Vergleiche und für das Herausarbeiten von idealen Einsatzbereichen dieser drei Ansätze ist eine vollständige Implementierung aller Varianten von Nöten. Der Abschluss dieses Projektes legte zunächst eine grobe Verwendbarkeit aller Implementierungen an den Tag. Auch eine Ausfallsicherheit und ein effizienter Speicheraufwand während der Verwendung sind bisher nicht vollständig gewährleistet. Dies liegt vor allem daran, dass bisher keine automatisierte Testumgebung für das Projekt verwendet wurde. Bei bisherigen Tests wurden für jede Funktion mehrere Testfälle angelegt, jedoch nur händisch ausgewertet.

Beide Punkte, die Vervollständigung und die Testumgebung, sollten bei der Weiterentwicklung des Projektes berücksichtigt werden. Insbesondere eine Clojure-Testumgebung für automatisierte Funktions- und Performanztests ist für eine Weiterentwicklung von Vorteil. Grade Performanztests sind für Optimierung und für den Vergleich der Implementierungen untereinander von großem Vorteil.

Abschließend bleibt noch zu sagen, dass trotz der Vergleichswerte in den Laufzeittests keines der Ansätze aus diesem Projekt über den Haufen geworfen wird. Die Weiterentwicklung aller drei Implementierungen kann durchaus starke Optimierungen und Erkenntnisse erbringen. Diese Erkenntnisse können Aufschluss über ideale Anwendungsbereiche der Implementierungen liefern. Sie können außerdem Fehler in theoretischen Konzepten aufdecken und damit zu weitläufigeren Verbesserungen der Theorie führen.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] Markus Bader. „Relational Algebra in Clojure“. Master Thesis. Technische Hochschule Mittelhessen, Dez. 2014.

[2] Peter A. Boncz und Martin Kersten. „MIL Primitives For Querying A Fragmented World“. In: *The VLDB Journal (VLDB J)* (Feb. 1970).

[3] Chris J. Date. *An introduction to database systems*. 6. ed. The systems programming series. Reading, Mass. [u.a.]: Addison-Wesley, 1995. ISBN: 020154329X.

[4] Chris J. Date. *An introduction to database systems*. 8. ed. Addison-Wesley, 2004. ISBN: 0-321-19784-4.

[5] Chris. J. Date. *Go Faster! The TransRelational ™Approach to DBMS Implementation*. English. ISBN: 978-87-7681-905-7.

[6] Chris J. Date und Hugh Darwen. *Foundation for object/relational databases : the third manifesto*. 1. print. Reading, Mass. [u.a.]: Addison-Wesley, 1998. ISBN: 0201309785.

[7] *Employees Sample Database*. Oracle Corporation. 2016. URL: https://dev.mysql.com/doc/employee/en/.

[8] *FrontPage — MonetDB*. MonetDB B.V. URL: https://www.monetdb.org/.

[9] *GitHub*. GitHub, Inc. 2016. URL: https://github.com/.

[10] Sören Gutzeit. *more.relational*. URL: https://github.com/seegy/more.relational.

[11] Rich Hickey. *Clojure - Functional Programming*. Rich Hickey. 2014. URL: http://clojure.org/functional_programming.

[12] Rich Hickey. *Clojure - home*. Rich Hickey. 2014. URL: http://clojure.org (besucht am 20. 01. 2015).

[13] Rich Hickey. *Clojure - refs. Refs and Transactions*. Rich Hickey. URL: http://clojure.org/refs.

[14] Rich Hickey. *Clojure - transducers*. Rich Hickey. 2014. URL: http://clojure.org/transducers.

[15] Rich Hickey. *clojure.set namespace — ClojureDocs - Community-Powered Clojure Documentation and Examples*. Rich Hickey. URL: https://clojuredocs.org/clojure.set.

[16] Rich Hickey. *index - clojure.set — ClojureDocs - Community-Powered Clojure Documentation and Examples*. Rich Hickey. URL: https://clojuredocs.org/clojure.set/index#example-542692d0c026201cdc326ef1.

[17] David Edgar Liebke. *Incanter: Statistical Computing and Graphics Environment for Clojure*. 2010. URL: http://incanter.org.

[18] *Storage model — MonetDB*. MonetDB B.V. URL: https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture/Storagemodel.

[19] Stephen A. Tarin. „Value-instance-connectivity computer-implemented database“. 6009432. URL: http://www.patentgenius.com/patent/6009432.html.

[20] Zachary Tellman. *ztellman/potemkin. GitHub.* URL: https://github.com/ztellman/potemkin.