Fachpraktikum 1590
„Erweiterbare Datenbanksysteme“

Aufgaben Phase 2

Wintersemester 2007/2008

Ralf Hartmut Güting, Thomas Behr, Markus Spiekermann, Simone Jandt, Christian Düntgen

Lehrgebiet Datenbanksysteme für neue Anwendungen, FernUniversität in Hagen

58084 Hagen

1 Algebra für Vektoren, Mengen und Multimengen

1.1 Einleitung, Motivation

Oft ist es wünschenswert, mengenwertige Attribute in Relationen zu speichern, um die Vervielfachung anderer Attribute zu vermeiden. Zum Beispiel haben einige Ortschaften mehrere Postleitzahlen, so dass der Name des Ortes mehrfach in der Relation gespeichert werden muss. Durch die Verwendung von Mengen als Attributtypen lässt sich somit teilweise eine enorme Verkleinerung des gespeicherten Datenvolumens erreichen. Beispielsweise enthält die `plz`-Relation der Datenbank `opt` in SECONDO 41267 Einträge (Name, Postleitzahl). Allerdings sind in dieser Datenbank nur 16788 unterschiedliche Ortsnamen gespeichert. Somit könnte die Speicherung von 24479 Strings (innerhalb von SECONDO immerhin je 56 Byte, zusammen also etwa 1.3 MB) vermieden werden. Man kann sich leicht überlegen, dass die Einsparungen bei ernsthaften Daten wesentlich größer ausfallen werden. Unsere `plz` Relation würde man mit der unten angegebenen Anweisung wie folgt komprimieren können (verwendete Operationen der Mengenalgebra werden weiter unten erklärt):

```
let plz_k = plz feed sortby[Ort]
                groupby[Ort;
                        PLZs : group feed project[PLZ]
                                      transformstream collect_set]
                consume
```

Auch für die Realisierung von *n*:*m* Beziehungen können sich solche Datentypen als nützlich erweisen. In klassischen relationalen Datenbanksystemen müssen solche Beziehungen über eine zusätzliche Relation dargestellt werden. Das Zusammenführen der gewünschten Informationen erfordert so teure Joins. Auch in SECONDO gibt es noch keine Möglichkeit, Mengen von Objekten als Attribut in einer Relation zu verwenden. In dieser Aufgabe soll diese Lücke geschlossen werden.

Aber nicht nur zur Datenkompression und zur Beschleunigung von Datenbankabfragen lassen sich solche Datentypen einsetzen. So lässt sich beispielsweise die Ähnlichkeit zweier Texte (bis auf Permutationen der enthaltenen Wörter) wie folgt bestimmen:

```
let set1 = text1 keywords collect_multiset;
let set2 = text2 keywords collect_multiset;
query 2*(size(set1 intersection set2)) / (size(set1) + size(set2));
```

Auch Filterbedingungen lassen sich mitunter wesentlich einfacher formulieren, z.B. lässt sich die Anfrage:

```
query   rel feed filter [ (.No = 3) or (.No = 5) or (.No = 17)
              or (.No = 23)] consume
```

mit Hilfe der Mengenalgebra wie folgt formulieren:

```
query rel feed filter [.No in create_set(3, 5, 17, 23)] consume
```

1.2 Algebrabeschreibung

1.2.1 Typkonstruktoren

Die Algebra soll drei verschiedene Datentypen zur Verfügung stellen, die jeweils Objektmengen eines Objekttyps der Kind *DATA* darstellen. Im Einzelnen sind dies *vector*, *set* und *multiset*. Alle Datentypen sollen als Attribute innerhalb von Relationen verwendet werden können, bilden also von der Kind *DATA* in die Kind *DATA* ab.

Achtung

Die Implementierung der dazugehörigen Klassen ist keineswegs trivial. `DBArrays` können hier nicht verwendet werden, da es sich um einen generischen Ansatz handelt. Somit muss direkt auf der `FLOB`-Schnittstelle gearbeitet werden. Eine weitere Hürde ist die Einbettung von evtl. in den „Subklassen“ enthaltenen `FLOBs` in die klasseneigenen `FLOBs`. Studieren Sie daher vorher genau, welche Funktionen Ihnen die `FLOB`-Klasse sowie die Klasse `Attribute` zur Verfügung stellen.

1.2.2 Operationen

Natürlich soll die Algebra auch einige mehr oder weniger nützliche Operationen enthalten. Diese werden im folgenden näher beschrieben.

- **create_vector:** $(t)^*, t \in DATA \rightarrow vector(t)$
- **create_set:** $(t)^*, t \in DATA \rightarrow set(t)$
- **create_multiset:** $(t)^*, t \in DATA \rightarrow multiset(t)$

Durch diese Operationen ist es möglich, Mengen aus einer beliebigen Anzahl von Elementen zu erzeugen. Z.B. wird durch **create_set**(1, 4, 9, 3, 4, 4, 1) die Menge {1, 3, 4, 9} erzeugt.

- **insert:** $set(t) \times t \rightarrow set(t)$
  $multiset(t) \times t \rightarrow multiset(t)$
- **+:** $vector(t) \times t \rightarrow vector(t)$

Durch Anwendung dieser Operationen wird einer Menge ein Element hinzugefügt (**insert**) bzw. ein Element an einen Vektor angehängt (+).

- **delete:** $set(t) \times t \rightarrow set(t)$
  $multiset(t) \times t \rightarrow multiset(t)$

Mittels dieses Operators wird ein Element aus einer Menge entfernt. Ist das Element nicht in der (Multi-) Menge enthalten, so entspricht das Ergebnis dem ersten Parameter.

- **contains:** $set(t) \times t \rightarrow bool$
  $multiset(t) \times t \rightarrow int$
  $vector(t) \times t \rightarrow bool$
- **in:** $t \times set(t) \rightarrow bool$
  $t \times multiset(t) \rightarrow int$
  $t \times vector(t) \rightarrow bool$

Hiermit wird geprüft, ob ein Element in einer Menge enthalten ist oder nicht (**contains**). Beachten Sie hierbei, dass bei einer Multimenge die Anzahl der enthaltenen Objekte das Ergebnis dieser Operation ist. Der Operator **in** ist eine symmetrische Variante von **contains**.

- **union, intersection, difference:** $set(t) \times set(t) \rightarrow set(t)$
  $multiset(t) \times multiset(t) \rightarrow multiset(t)$

Hier handelt es sich um die gewöhnlichen Mengenoperationen. Stellen Sie für alle diese Operationen eine lineare Laufzeit sicher.

- **concat:** $vector(t) \times vector(t) \rightarrow vector(t)$

Der **concat** Operator verschmilzt zwei Vektoren durch Hintereinanderhängen zu einem neuen.

- **<, <=, >, >=, =:** $set(t) \times set(t) \rightarrow bool$
  $multiset(t) \times multiset(t) \rightarrow bool$

Diese fünf Operationen prüfen je zwei Mengen auf ihre Teilmengenbeziehungen (echte Teilmenge, Teilmenge usw.). Auch diese Operationen sollen eine lineare Laufzeit besitzen.

- **get:** $vector(t) \times int \rightarrow t$

Der **get** Operator erlaubt es, ein bestimmtes Element innerhalb eines Vektors auszulesen. Ist die angegebene Position außerhalb des erlaubten Bereichs, ist ein undefiniertes Objekt zurückzugeben.

- **components:** $set(t) \rightarrow stream(t)$
  $multiset(t) \rightarrow stream(t)$
  $vector(t) \rightarrow stream(t)$

Dieser Operator gibt alle Elemente, die in der entsprechenden Mengendarstellung gespeichert sind, in einen Strom ab. Ist ein Element mehrfach in einer Menge (*multiset*, *vector*) vorhanden, so wird es entsprechend auch mehrfach in den Strom gegeben.

- **collect_set:** $stream(t), t \in DATA \rightarrow set(t)$
- **collect_multiset:** $stream(t), t \in DATA \rightarrow multiset(t)$
- **collect_vector:** $stream(t), t \in DATA \rightarrow vector(t)$

Mit diesen Operatoren werden Stromelemente in die entsprechende Mengendarstellung eingesammelt. Prinzipiell also eine Umkehrfunktion zur **components** Operation.

- **size**: $set(t) \rightarrow int$
  $multiset(t) \rightarrow int$
  $vector(t) \rightarrow int$

Der Operator **size** liefert die Anzahl der innerhalb einer Menge enthaltenen Elemente.

- **is_defined**: *set*(*t*) → *bool*
  multiset(*t*) → *bool*
  vector(*t*) → *bool*

Prüft, ob das übergebene Argument definiert ist. Verschiedene Operationen (z.B. **extract** angewandt auf einen leeren Tupelstrom) erzeugen undefinierte Werte. Dieser Operator dient dazu, diese von definierten (auch leeren) Mengenbeschreibungen zu unterscheiden.

1.3 Aufgabenstellung

1.3.1 Algebra-Implementierung

Implementieren Sie die Typen sowie die Operationen entsprechend der oben angegebenen Algebrabeschreibung. Alle Datentypen müssen als Attributtyp innerhalb von Relationen verwendbar sein. Kommentieren Sie Ihren Quellcode ausreichend und verständlich. Die Dateien müssen durch das PD-System bearbeitet werden können.

1.3.2 Display-Klassen

Erweitern Sie die textbasierten Schnittstellen von SECONDO um Display-Klassen für die neuen Datentypen. Auch hier müssen alle Kommentare PD-fähig sein.

1.3.3 Erweiterung des HoeseViewers

Implementieren Sie Display-Klassen für Ihre drei Datentypen. Beachten Sie dabei, dass Sie nicht wissen, welches die „Untertypen“ Ihrer Darstellungen sind. Verwenden Sie daher einen entsprechenden generischen Ansatz. Achten Sie wiederum auf eine vernünftige, ausreichende Kommentierung Ihres Quellcodes. Das `javadoc` Tool soll ordentliche Schnittstellenbeschreibungen aus Ihrem Code generieren können. Denken Sie daran, dass möglichst viel der Funktionalität der Subtypen verwendet werden soll. Daher muss Ihre eigene Display-Klasse möglichst viele Interfaces, die für die Realisierung von Funktionen notwendig sind, implementieren. Da es sich um einen Attributtyp handelt, darf eine Menge nur einen einzigen Eintrag im `QueryResult` erzeugen. Implementieren Sie daher das `ExternDisplay` Interface. Zeigen Sie die einzelnen Elemente dann in einem separaten Fenster an. Falls Ihre Unterklasse ebenfalls das `ExternDisplay` Interface implementiert, reagieren Sie entsprechend auf einen (Doppel-) Klick auf das jeweilige Element.

1.3.4 Tests

Erstellen Sie für jede Signatur der Operatoren einen enstsprechenden Eintrag in Ihrer `example` Datei. Schreiben Sie `Testrunner` Dateien, die sicherstellen, dass jeder Programmzweig Ihrer Implementierung durchlaufen wird. Testen Sie Ihre Implementierung auch mit großen Datenmengen. Versuchen Sie dabei einerseits, innerhalb Ihrer Datentypen viele Werte abzulegen. Andererseits sollten Sie auch große Relationen mit solchen mengenwertigen Attributen erzeugen und mit diesen Tests ausführen. Achten Sie bei solchen Tests auf den Speicher, der von SECONDO benötigt wird (Kommando `top` in der `bash`).

2 Eine Algebra für Histogramme

2.1 Motivation

Sicherlich kennen Sie folgende Situation: In der abendlichen Talkshow behauptet Politiker A.: „Überall, wo die CDU gewählt wird, geht es den Leuten besser.“, worauf Politiker B. entgegenhält: „Deshalb haben wir ebenda auch die meisten Selbstmörder und Obdachlosen...“. Politikerin C. meint dazu nur: „Wo die SPD an der Macht ist, wird die NPD gewählt.“.

Glauben Sie den Herrschaften? Nun, Vertrauen ist gut, Kontrolle ist besser, vor allem in der Demokratie. Doch wie will man solcherlei Behauptungen überprüfen? Glücklicherweise hilft uns die amtliche Statistik: Unter der URL *http://www.landesdatenbank-nrw.de/* bereitet man uns eine große Freude, indem man uns eine große Menge an Daten über NRW einfach zugänglich macht. Und jetzt?

Natürlich kann man Methoden des Data Mining bequemen, um potentiell interessante Zusammenhänge zu finden. Für den Hausgebrauch ist das zu aufwändig. Eine bequeme Methode besteht in der Nutzung visueller Datenexploration. Wir wollen *Histogramme* dazu verwenden, mögliche Zusammenhänge aufzuspüren.

Diese eignen sich insbesondere dazu, die *Häufigkeitsverteilung stetiger Merkmale* (z.B. Fließkommawerte) zu visualisieren. Dazu klassifiziert man zunächst die Daten, d.h. man unterteilt die Merkmalsachsen in Intervalle und zerlegt damit den Merkmalsraum in *Klassen*. Dann betrachtet man jeweils die Klassenhäufigkeiten, d.h. wieviele Werte in jede Klasse fallen. Das Histogramm veranschaulicht die Klassen und die zugehörigen Häufigkeiten anhand von *Flächen*, die über der Werteachse abgetragen werden. Die Breite der Flächen entspricht dem Wertebereich der Klasse, die Höhe r_i einer Fläche i berechnet sich aus der Klassenbreite b_i und der relativen Klassenhäufigkeit h_i wie folgt: $h_i = b_i \cdot r_i$ bzw. $r_i = h_i / b_i$

Der Verlauf der oberen Rechteckseiten entspricht dann in etwa der *Verteilungsdichte* der betrachteten Daten – bei anderen Darstellungsformen, etwa Säulendiagrammen, werden dagegen häufig lediglich absolute Häufigkeiten dargestellt (vgl. Abb. 1).

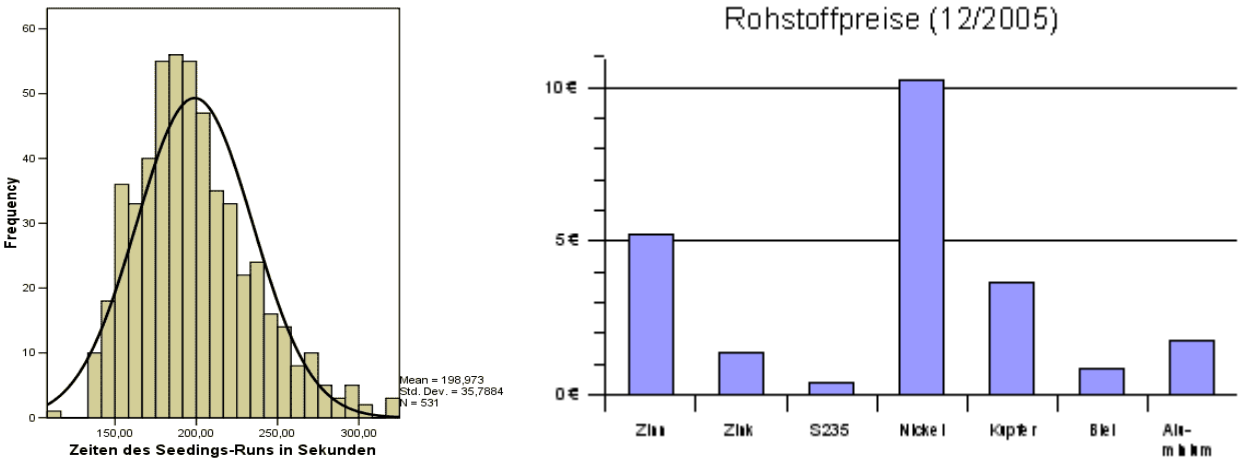


Abb. 1: Histogramm. *Links*: Ein Histogramm über normalverteilte Daten. Man sieht, dass sich Histogramm und Dichtefunktion (durchgehende Linie) entsprechen. *Rechts*: Nicht wirklich ein Histogramm, sondern nur ein Säulendiagramm: kein stetiges Merkmal, Balken berühren sich nicht, Häufigkeit wird durch Balkenhöhe dargestellt. Derartige Darstellungen werden fälschlicherweise oft als „Histogramme“ bezeichnet.

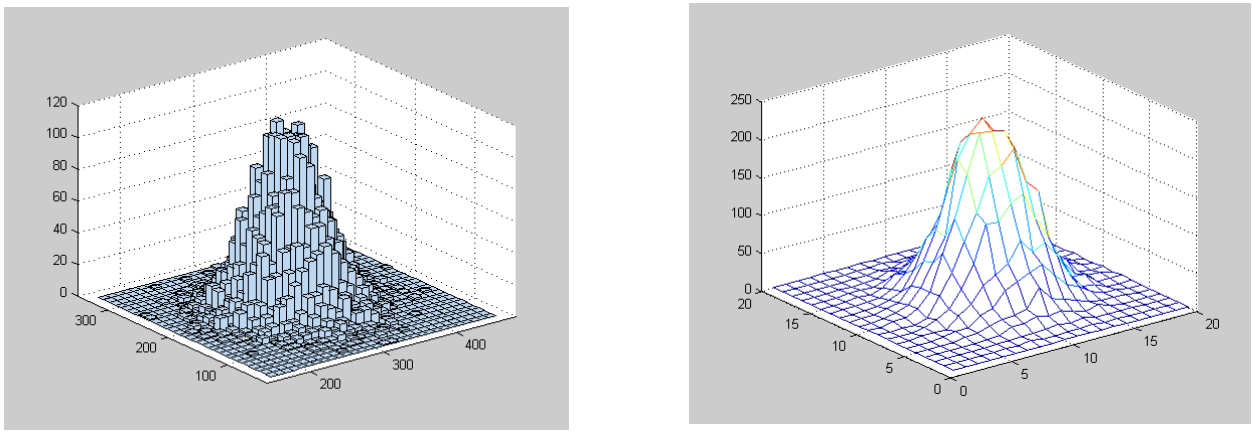


Abb. 2: 2D-Histogrammm. *Links*: Reguläre Darstellung. *Rechts*: Die entsprechende 2D-Häufigkeitsfläche als alternative Darstellungsform – die Mittelpunkte benachbarter Säulenstirnflächen werden miteinander verbunden.

Für Histogramme wählt man meist gleichbreite Klassen. Für besondere Anwendungen werden jedoch auch Histogramme erstellt, die eine möglichst gleiche relative Häufigkeit für alle Klassen anstreben und dazu die Klassenbreiten variieren.

2D-Histogramme erlauben es sogar, bivariate Verteilungen darzustellen, also gleichzeitige Abhängigkeiten von zwei Parametern aufzuzeigen (Abb. 2).

2.2 Datentypen

Um den Politikern auf den Zahn zu fühlen, sollen sie später einige interessante Daten durchforsten. Dazu sollen zwei neue Attributdatentypen in SECONDO verfügbar gemacht werden:

- *histogram1d*: Einfache Histogramme für ein stetiges Merkmal. Die Klassen werden über benachbarten Intervallen auf der Merkmalsachse gebildet.
- *histogram2d*: Histogramme für Produkte zweier stetiger Merkmale *X*,*Y*. Für beide Merkmale gibt es jeweils eine Menge benachbarter Intervalle auf der Merkmalsachse. Die Klassen werden über den den Produkten entsprechenden Teilflächen der *XY*-Ebene gebildet.

In der GNU Scientific Library (GSL) finden Sie eine C-Implementierung für diese Datentypen, die Sie gerne verwenden dürfen. Bedenken Sie jedoch, dass dort intern Zeiger benutzt werden, was in SECONDO Datentypen nicht erlaubt ist. Die GSL wird in SECONDO bereits von der GSL-Algebra verwendet, diese können Sie also als Beispiel für die Einbindung dieser Bibliothek in Ihre Algebra und SECONDO betrachten. Die Dokumentation zur GSL finden Sie u.a. im Internet: *http://www.gnu.org/software/gsl/manual.*

2.3 Operatoren

2.3.1 Operatoren zur Erzeugung von Histogrammen aus Daten

1. **set_histogram1d:** *stream*(*real*) → *histogram1d*
   set_histogram2d: *stream*(*real*) × *stream*(*real*) → *histogram2d*

Erzeugt ein Histogramm mit vorgegebenen Klassen-Intervallen und leeren Behältern.[1] Der Eingabestrom enthält alle Intervallgrenzen in aufsteigender Reihenfolge. Im Falle von **create_histogram2d** wird je ein Strom für die *X*- bzw. *Y*-Intervallgrenzen erwartet.

2. **create_histogram1d:** $stream(tuple(\mathrm{X})) \times a_i \times histogram1d \rightarrow histogram1d$

   create_histogram2d: $stream(tuple(\mathrm{X})) \times a_x \times a_y \times histogram2d \rightarrow histogram2d$

   Erzeugt ein Histogramm mit der vorgegebenen Klasseneinteilung des übergebenen Histogramms aus dem *real*-wertigen Attribut a_i (bzw. a_x und a_y) des Tupelstroms. Kann ein Element keiner Klasse zugeordnet werden, so wird es ignoriert (also nirgends zugeordnet).

3. **create_histogram1d_equiwidth:** $stream(tuple(\mathrm{X})) \times a_i \times int \rightarrow histogram1d$

   create_histogram2d_equiwidth: $stream(tuple(\mathrm{X})) \times a_x \times a_y \times int \times int \rightarrow histogram2d$

   Erzeugt adaptiv ein Histogramm mit maximal n (bzw. $n_x \times n_y$) Klassen aus dem Attribut a_i (der Kombination (a_x, a_y)) des Tupelstroms. Alle Klassen haben dieselbe Breite (bzw. dieselbe Grundfläche, wobei sich die Intervallbreite für beide Dimensionen unterscheiden darf). Alle definierten Stromelemente müssen einer Klasse zugeordnet werden.

4. **create_histogram1d_equicount:** $stream(tuple(\mathrm{X})) \times a_i \times int \rightarrow histogram1d$

   create_histogram2d_equicount: $stream(tuple(\mathrm{X})) \times a_x \times a_y \times int \times int \rightarrow histogram2d$

   Erzeugt adaptiv ein Histogramm mit maximal n (bzw. $n_x \times n_y$) Klassen aus dem Attribut a_i (der Kombination (a_x, a_y)) des Tupelstroms. Alle Behälter sollten möglichst denselben Füllstand erhalten. Dazu muss die optimale Intervall-Einteilung ermittelt werden. Alle definierten Stromelemente müssen einer Klasse zugeordnet werden.

2.3.2 Operatoren zur Auswertung von Histogrammen

5. **is_undefined**: $histogram1d \rightarrow bool$

   $histogram2d \rightarrow bool$

   Liefert TRUE, gdw. das Histogramm UNDEF ist.

6. **no_components**: $histogram1d \rightarrow int$

   binsX, **binsY**: $histogram2d \rightarrow int$

   Ermittelt die Anzahl der Klassen, bzw. die Anzahl der Intervalle auf der X-/Y-Achse.

7. **binrange_min, binrange_max**: $histogram1d \times int \rightarrow real$

   binrange_minX, binrange_maxX: $histogram2d \times int \rightarrow real$

   binrange_minY, binrange_maxY: $histogram2d \times int \rightarrow real$

---

1. Wir verwenden folgende Sprachregelung: Für jede Klasse i eines Histogramms A führen wir einen Behälter A_i ein. Die Behälter sind anfangs leer. Beim Aufbau des Histogramms wird für jeden Wert, der in eine bestimmte Klasse fällt, der Füllstand $|A_i|$ des zugeordneten Behälters um 1.0 erhöht.

Ermitteln die Intervallgrenzen für das *n*-te Intervall.

8. **getcount**: *histogram1d* × *int* → *real*
   histogram2d × *int* × *int* → *real*

   Ermittelt den Füllstand des Behälters einer Klasse.

9. **findbin**: *histogram1d* × *real* → *int*
   findbinX, findbinY: *histogram2d* × *real* → *int*

   Ermittelt den Index (bzw. den *X*-Index, *Y*-Index) der Klasse, in die das Element *z* fallen würde. Wird keine passende Klasse gefunden, so ist das Ergebnis UNDEF.

10. **is_refinement:** *histogram1d* × *histogram1d* → *bool*
    histogram2d × *histogram2d* → *bool*

    Testet ob Histogramm *A* bezüglich der Intervalgrenzen aller Klassen eine Verfeinerung von (oder identisch mit) Histogramm *B* ist. Die Intervallgrenzenverteilung ist feiner, wenn die Menge aller Intervallgrenzen von *A* in der von *B* vollständig enthalten ist.

11. <, =: *histogram1d* × *histogram1d* → *bool*
    histogram2d × *histogram2d* → *bool*

    Vergleichsrelationen für Histogramme. Es gilt $A < B$ gdw. *A* und *B* identische Klassen-Intervalle haben UND für jedes Paar von Behältern (A_i, B_i) gilt: $|A_i| < |B_i|$ bzw. $|A_{x,y}| < |B_{x,y}|$. Es gilt $A = B$ gdw. *A* und *B* identische Klassen-Intervalle haben UND für jedes Paar von Behältern (A_i, B_i) gilt: $|A_i| = |B_i|$ bzw. $|A_{x,y}| = |B_{x,y}|$.

12. **find_minbin, find_maxbin**: *histogram1d* → *stream*(*int*)
    histogram2d → *stream*(*tuple*((*X int*)(*Y int*)))

    Liefert einen Strom aller Klassen-Indizes *i* (bzw. Paare (*x, y*)), deren Behälter minimalen bzw. maximalen Füllstand haben.

13. **mean:** *histogram1d* → *real*
    histogram2d → *real*
    variance: *histogram1d* → *real*
    varianceX, **varianceY**, **covariance**: *histogram2d* → *real*

    Berechnen Mittelwerte, Varianzen und Kovarianz aus dem Histogramm.

14. **distance**: *histogram1d* × *histogram1d* → *real*
    histogram2d × *histogram2d* → *real*

    Berechnet den Abstand zwischen Histogrammen. Der Abstand entspricht der Summe der Quadrate der Flächendifferenzen (Volumendifferenzen) für jede Klasse. Falls eines der beiden Histogramme eine Verfeinerung des anderen ist, wird es zuerst passend „vergröbert“

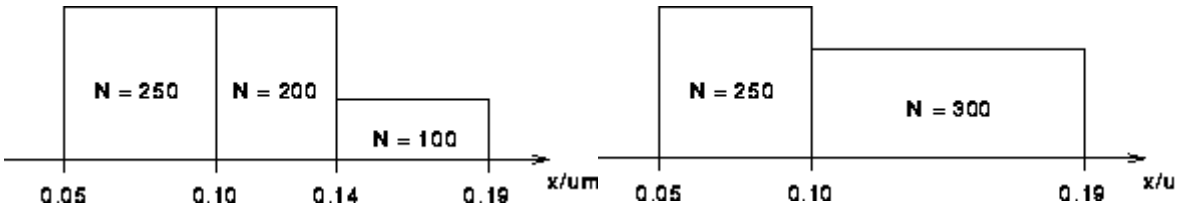
(vgl. Operator **translate**). Für gleiche Histogramme ergibt sich der Abstand 0. Sind beide Histogramme nicht kompatibel, so ist das Ergebnis UNDEF.

2.3.3 Operatoren zur Erzeugung von Histogrammen aus Histogrammen

15. **translate**: $histogram1d \times histogram1d \rightarrow histogram1d$

$histogram2d \times histogram2d \rightarrow histogram2d$

Vergröbere das Histogramm *A* zu einem Histogramm vom Typ *B*. Falls Histogramm *A* keine Verfeinerung des anderen ist, so ist das Ergebnis UNDEF.



Translate: Vergröberung eines Histogramms. *Links*: Die mittlere und die rechte Klasse sollen vereinigt werden. *Rechts*: Das Ergebnis.

16. **use**: $histogram1d \times (real \times T^* \rightarrow real) \rightarrow histogram1d$

$histogram2d \times (real \times T^* \rightarrow real) \rightarrow histogram2d$

use2: $histogram1d \times histogram1d \times (real \times real \times T^* \rightarrow real) \rightarrow histogram1d$

$histogram2d \times histogram2d \times (real \times real \times T^* \rightarrow real) \rightarrow histogram2d$

Die Parameterfunktion für **use** hat mindestens ein *real*-wertiges Argument, für das der Füllstand der Behälter eingesetzt wird. Weitere Parameter sind optional. Der Operator wendet diese Parameterfunktion auf den Füllstand jeder Klasse des ersten Arguments an. Der berechnete Wert wird in den Behälter derselben Klasse des Ergebnishistogramms abgelegt. So kann man etwa das Histogramm mittels (*****: $real \times real \rightarrow real$) skalieren. Falls eines der beiden Histogramme von **use2** eine Verfeinerung des anderen ist, wird es zuerst passend „vergröbert“ (vgl. Operator **translate**). Sind beide Histogramme nicht kompatibel, so ist das Ergebnis UNDEF. Beide Histogramme haben nun dieselbe Klasseneinteilung. Diesmal erhält die Parameterfunktion die Füllstände *beider* Histogramme als Eingaben an 1. bzw. 2. Stelle, ansonsten ändert sich nichts. Damit kann man z.B. mittels (+: $real \times real \rightarrow real$) die klassenweise Summe zweier Histogramme bilden.

17. **fold**: $histogram1d \times (T \times real \rightarrow T) \times T \rightarrow T$

$histogram2d \times (T \times real \rightarrow T) \times T \rightarrow T$

Faltet das Histogramm mit der Parameterfunktion. Beispiel: **fold** `(H,fun(R1:`*real*`, R2:`*real*`) R1 + R2), 0.0)` berechnet die Summe aller Füllstände im Histogramm *H*. Der 3. Parameter ist der Startwert, der beim ersten Aufruf der Parameterfunktion für *R1* eingesetzt wird. Für *R2* wird der Füllstand des Behälters zur 1. Klasse in *H* eingesetzt. Das Ergebnis wird in der 2. Iteration für *R1* eingesetzt, für *R2* wird dann der Füllstand des Behälters zur 2. Klasse von *H* verwendet. Nach dem Aufruf für die letzte Klasse wird das Ergebnis des letzten Auf-

rufs der Parameterfunktion als Ergebnis des Operatoraufrufs ausgegeben. Ist *H* UNDEF, so ist das Ergebnis UNDEF. Der Operator **aggregate** hat eine ähnliche Funktionalität.

18. **shrink_eager**: *histogram1d* × *real* × *real* → *histogram1d*
histogram2d × *real* × *real* × *real* × *real* → *histogram2d*
shrink_lazy: *histogram1d* × *real* × *real* → *histogram1d*
histogram2d × *real* × *real* × *real* × *real* → *histogram2d*

shrink_eager reduziert ein Histogramm auf den angegebenen Wertebereich. Nur Klassen, deren Intervalle vollständig im gegebenen Wertebereich liegen, werden in das Ergebnis übernommen. Bei **shrink_lazy** werden hingegen alle Klassen übernommen, deren Intervalle den gegebenen Wertebereich schneiden.

19. **insert**: *histogram1d* × *real* × *real* → *histogram1d*
histogram1d × *real* → *histogram1d*
histogram2d × *real* × *real* × *real* → *histogram1d*
histogram2d × *real* × *real* → *histogram1d*

Erhöht für einen gegebenen Wert *x* bzw. (*x, y*) den Füllstand des zugehörigen Klassenbehälters um einen gegebenen Wert *z* (letztes *real*-Argument). Wird *z* nicht angegeben, soll *z = 1.0* angenommen werden. Kann das Element keiner Klasse zugeordnet werden, so wird es ignoriert (also nirgends zugeordnet).

2.4 Erweiterung der Display-Klassen und der JavaGUI

Erweitern Sie die textbasierten Schnittstellen und den *HoeseViewer* um eine Darstellungsfunktionalität für Histogramme. Im HoeseViewer sollen mindestens 2 selektierte Histogrammen desselben Typs (*histogram1d* oder *histogram2d*) im direkten Vergleich gleichzeitig darstellbar sein. Auch Informationen wie Intervallgrenzen und relative Häufigkeiten sollen angezeigt werden. Für die 2D/3D-Darstellung können Sie auf bereits bestehende Komponenten (siehe Verzeichnis *secondo/Javagui/viewer3d/*) und das Interface `ExternDisplay` zurückgreifen. In der 3D-Darstellung soll die Perspektive frei veränderbar sein, um eine Betrachtung aus beliebigen Blickwinkeln zu ermöglichen. Neben der jeweiligen regulären Darstellung mit Flächen/Säulen können Sie gerne zusätzliche Darstellungsoptionen anbieten.

2.5 Tests

Bitte schreiben Sie Beispielanfragen in die `example` Datei und `TestRunner`-Testfälle für jeden Operator. Die Testmenge soll auch Parameter-Kombinationen mit „unsinnigen“, falschen und undefinierten Argumenten umfassen. Die Operatoren sollten innerhalb einer Query oft (mehrere 100.000 mal!) aufgerufen werden, um die Stabilität zu testen und „Speicherlöcher“, sowie Fehler in der Stromverarbeitung zu finden. Testen Sie jeden Zweig Ihres Codes. Ersinnen Sie bitte Testfälle, die tatsächlich die Ausführung jedes einzelnen Codeblockes erzwingen. Verifzieren Sie

die Ausführung der Codeblöcke mittels Testausgaben während der Abarbeitung („Debug-Ausgaben“). Code zum Testen (wie Testausgaben) können Sie z.B. mittels Makrodefinitionen und Precompiler-Anweisungen (`#ifdef ... #endif`) ein- und ausschalten. Überlegen Sie sich **vor** der Testausführung, welche Ergebnisse Sie erwarten und verifizieren oder falsifizieren Sie damit anschließend die tatsächlichen Ergebnisse.

2.6 Dokumentation

Kommentieren Sie bitte Ihren Quelltext. Dies erlaubt es *Ihnen*, ihren Kommilitoninnen und Kommilitonen, späteren Benutzern *und nicht zuletzt uns*, Ihre Gedankengänge nachzuvollziehen. Nutzen Sie das `PD-System`, um ihre C++ - Dateien zu dokumentieren. Mittels `javadoc` muss man aus ihrem Java-Quelltext ordentliche Schnittstellenbeschreibungen erzeugen können.

2.7 Anwendung

Nachdem Sie die erforderlichen Datentypen und Operatoren implementiert und getestet haben, sollen Sie nun die Früchte ihrer Arbeit ernten.

Unter der URL

```
http://www.informatik.fernuni-hagen.de/import/pi4/duentgen/lehre/
fapra0708.html
```

haben wir einige interessante Tabellen aus der Landesdatenbank NRW für Sie zusammengestellt:

1. Sozialversicherungspflichtig Beschäftigte am 30.06.2004 nach Arbeitsort (Gemeinden)
2. Bevölkerungsanzahl, Bodenfläche, Bevölkerungsdichte am 31.12.2004 (Gemeinden)
3. Bundestagswahl 2005 Wahlberechtigte, Wähler, Zweitstimmen nach Parteien (Gemeinden)
4. Gemeinderatswahlen 2004 -Wahlberechtigte, Wähler, Stimmen nach Parteien (Gemeinden)
5. Obdachlose 30.06.2004 (Gemeinden)
6. Todesursachenstatistik 2004 - Ausgewählte Todesursachen (Kreise u. kreisfreie Städte)

Mit Hilfe dieser Daten sollen Sie die Aussagen der Politiker (siehe Abschnitt 2.1, Motivation) überprüfen. Dazu müssen Sie die Daten zunächst nach SECONDO importieren. Dann können Sie mittels geeigneter Anfragen unter Verwendung von Joins, Gruppierung und anderen in SECONDO vorhandenen Operatoren *real*-Daten erzeugen, die Sie unter Zuhilfenahme der von ihnen implementierten Datentypen visualisieren.

Ist an den Behauptungen irgendetwas dran? Finden Sie vielleicht andere interessante Korrelationen?[1] Doch Vorsicht: Statistische Befunde sind nicht immer „Beweise“ für kausale Zusammenhänge! Dokumentieren Sie interessante Befunde und die zugehörigen Anfragen zur späteren Diskussion („Fakten, Fakten, Fakten!“, Herr Markwort lässt grüßen!).

---

1. Falls Sie sich nicht auf ihr Augenmaß verlassen wollen, so können Sie die empirische Korrelation $\rho(X,Y)$ für 2D-Histogramme mittels der Operatoren **varianceX**, **varianceY** und **covariance** und der folgenden Formel berechnen:

$$\rho(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X)} \cdot \sqrt{Var(Y)}}$$

2.8 Hinweise

Wenn Sie mehr zu Histogrammen und statistischen Methoden wissen wollen, finden sie umfangreiche Definitionen und Erläuterungen in allen Lehrbüchern zur Statistik, z.B.

> J. Hartung. Statistik. Lehr- und Handbuch der angewandten Statistik. 13. Auflage, München 2002.

Für am Thema Interessierte bietet das Lehrgebiet Datenbanksysteme für neue Anwendungen zum kommenden Sommersemester ein Seminar zum Thema „Data Mining“ an.

3 Erweiterung der Schachalgebra

Um diese Aufgabe bearbeiten zu können, müssen Sie die Grundregeln des Schachspiels kennen (lernen). Tiefergehende strategische oder taktische Kenntnisse sind nicht erforderlich.

In diesem Projekt soll die bereits im letzten Praktikum implementierte Schachalgebra um neue Operatoren erweitert und an einigen Stellen modifiziert werden. Bislang benötigen die Operationen kein Schachwissen. Es sollen nun u.a. Funktionen implementiert werden, die aus einer gegebenen Stellung die Vorausschau um einen Zug ermöglichen, d.h.: Für jede Figur einer gegebenen Stellung müssen alle von ihr gemäß der Schachregeln erreichbaren Felder berechnet werden können.

3.1 Status Quo der Implementierung

Die Algebra besitzt die Datentypen *chessgame*, *position*, *move* und *material*. Der Typ *chessgame* stellt eine komplette Partie dar, *position* eine Stellung und *move* einen Halbzug. Der Typ *material* stellt eine Menge von weißen und schwarzen Figuren dar; damit kann z.B. nach Stellungen gesucht werden, die dieses Figurenmaterial enthalten. Grundlegende Operationen über diesen Typen sind:

getkey	:	$chessgame \times string \rightarrow string$
positions	:	$chessgame \rightarrow stream(position)$
moves	:	$chessgame \rightarrow stream(move)$
getposition	:	$chessgame \times int \rightarrow position$
getmove	:	$chessgame \times int \rightarrow move$
pieces	:	$position \rightarrow material$
moveNo	:	$position \rightarrow int,\ move \rightarrow int$

Damit lassen sich aus einer Partie detailliertere Informationen gewinnen. Die Operation **getkey** akzeptiert die folgenden Schlüsselwörter: `name_{w,b}` (Name des Spielers, der die weißen bzw. schwarzen Steine spielt), `rating_{b,w}`, `event`, `site`, `date`, `result`, `eco_code`, `moves`, ggf. weitere Metadaten eines Spiels. Die Operationen **getposition** und **getmove** ermitteln die Stellung bzw. die Figurbewegung des angegebenen Halbzugs. Mittels **pieces** kann das Material einer Stellung berechnet werden und **moveNo** bestimmt zu einer Stellung oder einem Zug die Nummer des

Halbzugs der zugehörigen Partie. Charakteristika von Zügen lassen sich über die folgenden Operationen bestimmen:

agent : *move* → *string*
captured : *move* → *string*
startrow : *move* → *int*
endrow : *move* → *int*
startfile : *move* → *string*
endfile : *move* → *string*
check : *move* → *bool*
captures : *move* → *bool*

Die Operationen **agent** und **captured** ermitteln den engl. Figurennamen einer bewegten oder gefangenen (geschlagenen) Figur bzw. den Wert `"none"`. Dabei werden weiße Figuren durch einen großen, schwarze durch einen kleinen Anfangsbuchstaben gekennzeichnet, z.B.: weiße Dame `"Queen"`, schwarzer Springer `"knight"`. Linien werden durch die Stringwerte `"a"` bis `"h"` und Reihen durch die Integerwerte 1-8 identifiziert. Über die Prädikate **check** und **captures** kann man testen, ob in einem Zug Schach geboten bzw. eine Figur geschlagen wird. Tests auf Stellungen oder Teilstellungen können mittels der folgenden Operationen durchgeführt werden:

posrange : *position* × *string* × *int* × *string* × *int* → *position*
includes : *position* × *position* → *bool*

Durch Angabe des linken, unteren Feldes und des rechten, oberen Feldes liefert der *range* Operator eine Stellung, die nur die Figuren aus dem angegebenen Bereich enthält, z.B. definiert der Ausschnitt `"a",1,"a",8` die gesamte Linie *a*. Der Operator **includes** liefert `TRUE`, wenn die erste Stellung in der zweiten enthalten ist, d.h. jede Figur des ersten Arguments ist in der zweiten Stellung auf demselben Feld. Um die Suche nach ähnlichen Stellungen zu ermöglichen, gibt es die Operationen

piececount : *position* × *string* → *int*
piececount : *material* × *string* → *int*
=, <, ~ : *material* × *material* → *bool*
=, <, ~ : *position* × *position* → *bool*

die die Anzahl der Figuren eines bestimmten Typs bzw. die Anzahl aller Figuren ermitteln oder einen Vergleich des Materials erlauben. Die Operationen < und ~ berücksichtigen die gewichtete Summe der Figurwerte (pawn = 1, knight, bishop = 3, rook = 5, queen = 9) während der Operator = einen exakten Vergleich von Positionen bzw. Material durchführt.

3.2 Beispielanfragen

Die folgende Anfrage ermittelt die Spiele, in denen Weiß eine kurze Rochade und Schwarz eine lange Rochade durchgeführt hat.

```
query wjc2 feed filter[ (.Match moves
 filter[ (. startfile = "e") and (. endfile = "g")
         and (. agent = "King") ] ensure[1])]
 filter[ (.Match moves
   filter[ (. startfile = "e") and (. endfile = "c")
           and (. agent = "king") ] ensure[1])] consume
```

Das Objekt `wjc2` ist hier eine Relation, die ein Attribut `Match` vom Typ *chessgame* besitzt. In der Relation sind 359 Partien mit insgesamt 33948 Zügen gespeichert. Der Operator **filter** kann auf Ströme eines beliebigen Typs angewendet werden (hier *stream*(*move*)) ebenso der Operator **ensure**, der prüft, ob der Strom eine bestimmte Anzahl von Elementen enthält.

Alle Partien, in denen 4 Türme auf einer Linie zu finden sind, lassen sich etwa folgendermaßen berechnen:

```
query wjc2 feed filter[ fun(t: TUPLE) allFiles feed
 filter[ fun(s: TUPLE) attr(t, Match) positions
  filter[. posrange[attr(s,File), 1, attr(s,File), 8]
   piececount["Rr"] > 3 ] ensure[1]] ensure[1]] consume
```

Dabei ist `allFiles` eine Relation mit Attribut `File` vom Typ *string*, die 8 Tupel mit den Werten von „a“ bis „h“ enthält.

3.3 Erweiterungen und Modifikationen

Zunächst sollen zwei neue Datentypen

field - ein Feld eines Schachbretts, z.B. e7
piece - eine Figur bzw. eine Teilmenge aller möglichen Figuren, z.B. bishop

eingeführt werden. Diese werden häufig in Anfragen benötigt, um eine gesuchte Spielsituation zu beschreiben. Um zu vermeiden, dass dann immer eine Konstante wie z.B.

```
[const field value "e7"]
[const piece value "bishop"]
```

innerhalb der Anfragen notiert werden muss, soll es ein Skript “*initchessdb*” der Form

```
let a1 = [const field value "a1"]
...
let h8 = [const field value "h8"]

let Pawn = [const piece value "Pawn"]
...
let king = [const piece value "king"]
let KING = [const piece value "KING"]
```

geben. Dieses sollte im bin-Verzeichnis liegen und (manuell) beim Anlegen einer neuen Chess-Datenbank ausgeführt werden.

Wird eine Figur komplett in Großbuchstaben notiert, z.B. `KING`, bedeutet dies, dass es sich um einen König einer beliebigen Farbe handelt. Außerdem kann der Typ *piece* den besonderen Wert `NONE` annehmen. Damit kann man in Anfragen Bedingungen formulieren wie etwa

```
.Field = e7
.Piece = rook
```

Durch Einführung dieser Typen ist es außerdem möglich, einige der bestehenden Operationen typsicherer zu machen. Die Signaturen einiger Operationen sollen daher wie folgt geändert werden:

agent, captured	: *move* → *piece*	# (_)
posrange	: *position* × *field* × *field* → *position*	_ # [_ , _]

Um eine einheitliche Syntax innerhalb der Schachalgebra zu erzielen, geben wir in diesem Abschnitt für die Operationen auch Syntaxmuster an. Dabei steht "_" für ein Argument, "#" für den Operator. Klammern und Kommas sind zu setzen wie angezeigt. Solche Syntaxmuster werden in etwas anderem Format im spec-file definiert.

Die Operationen **startrow**, **endrow**, **startfile** und **endfile** sollen durch

startfield, endfield	: *move* → *field*	# (_)

ersetzt werden. Weiterhin sollen auch noch die folgenden Überladungen für den Operator **piececount** implementiert werden:

piececount	: *position* × *piece* → *int*	_ # [_]
piececount	: *material* × *piece* → *int*	_ # [_]

Da nun auch die Schachregeln implementiert werden, sind folgende Operationen möglich:

attackedby	: *position* × *piece* × *piece* → *bool*	_ _ # [_]
attackedby	: *position* × *field* × *piece* → *bool*	_ _ # [_]
attackedfrom	: *position* × *piece* × *field* → *bool*	_ _ # [_]
attackedfrom	: *position* × *field* × *field* → *bool*	_ _ # [_]
protectedby	: *position* × *piece* × *piece* → *bool*	_ _ # [_]
protectedby	: *position* × *field* × *piece* → *bool*	_ _ # [_]
protectedfrom	: *position* × *piece* × *field* → *bool*	_ _ # [_]
protectedfrom	: *position* × *field* × *field* → *bool*	_ _ # [_]
attackcount	: *position* × *field* → *int*	_ # [_]
protectcount	: *position* × *field* → *int*	_ # [_]
apply_move	: *position* × *field* × *field* → *position*	_ # [_ , _]
checkmate, stalemate	: *position* → *bool*	# (_)

Die Operation **attackedby** prüft, ob eine Figur oder ein Feld von einer bestimmten Figur angegriffen wird; alternativ prüft **attackedfrom**, ob dies von einem bestimmten Feld aus geschieht. Bei der ersten Version ist also der Ort, von dem aus angegriffen wird, unbestimmt und bei der zweiten Version ist offen, welche Figur angreift. Ob dem König Schach geboten wird, kann nun auch z.B. folgendermaßen geprüft werden:

```
.Pos KING attackedby[f1]
.Pos KING attackedby[PAWN]
```

Falls das zweite Argument ein Feld ist, machen diese Operationen nur Sinn, falls dort eine Figur steht, da sonst unklar ist, wer der Angreifer ist. Wenn dieses Feld leer ist, liefert die Operation daher `false` zurück.[1] Die Operationen **protectedby** und **protectedfrom** verhalten sich analog. Die Operationen **attackcount** und **protectcount** zählen, wie oft ein Feld, auf dem eine Figur steht, angegriffen bzw. gedeckt ist. Die Operation **apply_move** liefert die Ergebnisstellung nach Anwendung eines Zuges. Die Operationen **checkmate** und **stalemate** prüfen, ob eine gegebene Position ein Schachmatt (ein König steht im Schach und kein möglicher Zug kann dies ändern) bzw. ein Patt (die am Zug befindliche Partei kann keinen regulären Zug ausführen) ist.

1. Um Zugmöglichkeiten beider Spieler auf leere Felder zu überprüfen, werden unten noch allgemeinere Operationen angeboten.

Für Figuren sollen folgende neue Operationen hinzugefügt werden:

iswhite	: *piece* → *bool*	# (_)
=, is, samecolor	: *piece* × *piece* → *bool*	_ # _
piecevalue	: *piece* → *int*	# (_)
pieceof	: *piece* × *int* → *piece*	_ # [_]

Die Gleichheit "=" erfordert gleiche Art und gleiche Farbe; die **is** Operation erfordert nur gleiche Art, die **samecolor** Operation nur gleiche Farbe. Der Operator **piecevalue** liefert das Gewicht bzw. den Wert einer Figur und **pieceof** bestimmt zu einem farblosen Figurtyp, wie etwa `KING`, und z.B. einer Zugnummer den tatsächlich gezogenen Figurtyp, also `King` oder `king`. Dabei stellt jede ungerade Zahl den weißen, jede gerade Zahl den schwarzen Spieler dar.

Auf Feldern soll es die unten aufgeführten Operationen geben:

iswhite	: *field* → *bool*	# (_)
file	: *field* → *string*	# (_)
row	: *field* → *int*	# (_)
neighbors	: *field* → *stream*(*field*)	_ #

Ein bestimmtes Nachbarfeld zu einem gegebenen Feld läßt sich folgendermaßen ermitteln:

north, east, south, west, **northwest, southeast, northwest, southwest**	: *field* → *field*	# (_)

Die folgenden Prädikate überprüfen, ob und in welcher Art zwei Felder benachbart sind

is_neighbor, left, right, above, below	: *field* × *field* → *bool*	_ # _

Da manche Operationen von der Zugnummer abhängen und über die Zugnummer bestimmt werden kann, ob Weiß oder Schwarz am Zug ist, sind folgende Operationen hilfreich:

even, odd	: *int* → *bool*	# (_)
lastmove	: *chessgame* → *int*	# (_)

Um Bedingungen auf Strömen etwas einfacher formulieren zu können, sollen noch die Operationen

exists, forall	: *stream*(*ANY*) × (*ANY* → *bool*) → *bool*	_ # [_]

implementiert werden, die für einen beliebigen Eingabestrom prüfen, ob die in der Parameterfunktion formulierte Bedingung für mindestens ein bzw. alle einströmenden Elemente gilt. *ANY* can dabei natürlich auch ein Tupeltyp sein.

Darstellung interessanter Spielsituationen

Mit den oben aufgeführten Operationen lassen sich nun zwar schon einige elementare Spielsituationen ausdrücken, richtig ausdrucksstark wird die Algebra aber erst, wenn zu einer Stellung die daraus für Weiß und Schwarz möglichen Züge ermittelt werden können. Dies erlaubt die folgende Operation:

pos_moves: *position* → *stream*(*tuple*([*SPiece: piece, SField: field,*
EPiece: piece, EField: field])) _ #

Für eine gegebene Stellung kann man mit **pos_moves** eine Relation (als Tupelstrom) erzeugen, die für jede Figur *SPiece* (*S* für *start*) angibt, auf welchem Feld *SField* sie steht, auf welches andere Feld *EField* (*E* für *end*) sie ziehen kann und welche Figur *EPiece* dort steht. Falls auf dem Zielfeld keine Figur steht, enthält das Attribut *EPiece* den Wert NONE.

Die Semantik ist dabei, dass die Figur das Zielfeld erreichen kann aufgrund ihrer Bewegungsart, nicht notwendig, dass der Zug tatsächlich erlaubt ist (dies könnte z.B. durch Fesselung nicht der Fall sein). Auf dem Zielfeld kann eine Figur anderer Farbe oder auch der gleichen Farbe stehen; damit sind sowohl Angriffsmöglichkeiten dargestellt wie auch Deckungen.

piece_moves: *position* × *piece* → *stream*(*tuple*([*SPiece: piece,*
SField: field,
EPiece: piece,
EField: field])) _ # [_]

Diese Variante erzeugt nur die Tupel für eine gegebene Figurenart (auf dem Startfeld). Sie ist nur der Bequemlichkeit und Effizienz halber noch hinzugefügt. Mit diesen Operationen könnte man z.B. die Springergabel, die simultan König und Dame angreift, wie folgt ausdrücken:

```
query games feed
  filter[.Match positions
    exists[fun(pos: position)
      pos piece_moves[KNIGHT] {a}
      pos piece_moves[KNIGHT] {b}
      symmjoin[.SField_a = ..SField_b]
      filter[(.EPiece_a = KING) and (.EPiece_b = QUEEN) and
        (.EPiece_a samecolor .EPiece_b) and
        not (SPiece_a samecolor EPiece_a)
      ]
    ]
  ]
  consume
```

Damit werden Schachpartien gefunden, in denen es Positionen gibt, in denen ein Springer einer beliebigen Farbe *x* sowohl einen König als auch eine Dame einer anderen Farbe *y* angreift. Da durch den Join alle Paare möglicher Felder, die von Springern (beliebiger Farbe) erreichbar sind, gebildet werden, muss danach sichergestellt werden, dass nur Angriffe betrachtet werden, d.h., dass Springer und König/Dame verschiedene Farben haben.

Zu **pos_moves** und **piece_moves** gibt es noch Varianten:

pos_moves_blocked: *position* → *stream*(*tuple*([*SPiece: piece,*
SField: field,
BPiece: piece,
BField: field
EPiece: piece,
EField: field])) _ #

Dies bedeutet, dass die Figur *SPiece* auf Feld *SField* auf das Feld *EField* ziehen könnte (auf dem Figur *EPiece* steht - NONE ist möglich), wenn nicht die Figur *BPiece* auf Feld *BField* diesen Zug blockieren würde (*B* für *blocking*). Der Wert von *BPiece* ist hierbei nicht NONE.

Analog dazu gibt es wieder eine Variante, die dies von vornherein auf Figuren einer bestimmten Art (für das Startfeld *SField*) einschränkt.

piece_moves_blocked: *position* × *piece* → *stream*(*tuple*([*SPiece: piece, SField: field, BPiece: piece, BField: field EPiece: piece, EField: field*])) _ # [_]

Weiterhin gibt es eine Operation, die eine *position* in eine Relation verwandelt, die die Belegung aller Felder beschreibt:

pos_fields: *position* → *stream*(*tuple*([*Field: field, Piece: piece*])) _ #

Dabei werden auch leere Felder angegeben; die Relation hat also 64 Tupel.

Für bestimmte Anfragen ist es hilfreich, Zugriff sowohl auf den aktuellen *move* wie auch die *position* zu haben. Dies erspart eine umstandliche Berechnung mittel **moveNo** und **getmove** bzw. **getposition**. Daher liefert

history: *chessgame* → *stream*(*tuple*([*No: int, Pos: position, Move: move*])) _ #

die komplette Geschichte einer Partie, in der Figurbewegung und Position direkt in einem Tupel zusammengehalten werden. Zusätzlich wird die Zugnummer mitgeliefert.

Manchmal ist es nützlich, wenn aufeinanderfolgende Züge und Positionen in einem Tupel vorzufinden sind. Deshalb sollen noch folgende Operationen implementiert werden:

twotuples: *stream*(*tuple*(*X*)) → *stream*(*tuple*(*X'1 X'2*)) _ #
ntuples : *stream*(*tuple*(*X*)) × *int* → *stream*(*tuple*(*X'1 ... X'n*)) _ # [_]

Diese Operatoren fassen jeweils zwei bzw. *n* aufeinanderfolgende Tupel eines Stroms zu einem konkatenierten Tupel zusammen. Um dabei Namenskonflikte zu vermeiden, wird an die Attributnamen des ersten Tupels die Ziffer 1, die des zweiten Tupels die Ziffer 2, usw. angehängt. Dabei sei *n* begrenzt, etwa auf 10.

```
query match history twotuples
```

liefert also für ein Objekt `match` vom Typ *chessgame* einen Strom von Tupeln des Typs

```
tuple([ No1: int, Pos1: position, Move1: move,
        No2: int, Pos2: position, Move2: move])
```

3.4 Viewer

Der Viewer soll um ansprechende und übersichtliche Darstellungen für die Typen *move*, *position* und *material* erweitert werden. Wenn der Viewer eine Relation erhält, die neben einem Attribut vom Typ *chessgame* auch noch ein Attribut `MoveNo` vom Typ *int* erhält, so wird dieses als Positionsangabe interpretiert. Die Schachpartie soll dann direkt an dieser Stelle angezeigt werden.

Bei der Konzeption dieser Aufgabe wurden ca. 20 Queries mit interessanten Schachanfragen formuliert. Diese sollen nun im Viewer als Anfragemuster bereitgestellt werden, d.h. es gibt Anfragemasken, in denen genau beschrieben wird, was diese Abfragen tun und welche Parameter ggf. umdefiniert werden können, z.B. Relationenname, Figurnamen etc.

Zusätzlich soll es über einen Editiermodus möglich sein, selbst solche Anfrageschablonen zu erstellen, zu speichern und zu laden. Somit können ganze Abfragebibliotheken erstellt werden und verschiedene Benutzer können diese untereinander austauschen.

3.5 Komplexere Beispielanfragen

Zeige Partien, in denen ein möglicher en passant Schlag unterbleibt (für Weiß als Ziehenden, Schwarz als nicht Schlagenden):

```
query games feed
  filter[.Match history twotuples
    exists[fun(h:TUPLE)
      (agent(attr(h, Move1)) = Pawn)
        and (row(startfield(attr(h, Move1))) = 2)
        and (row(endfield(attr(h, Move1))) = 4)
        and (attr(h, Pos2) pos_fields
          exists[
            (.Field west endfield(attr(h, Move1))
              or (.Field east endfield(attr(h, Move1)))
            and (.Piece = pawn)
            and not
              ((startfield(attr(h, Move2)) = .Field)
                and (endfield(attr(h, Move2))
                  south endfield(attr(h, Move1))))
          ]
    ]
  ]
  consume
```

Zeige Partien, die durch ersticktes Matt (nur der Springer setzt Matt) gewonnen wurden:

```
query Games feed
  filter[checkmate(.Match getposition[lastmove(.Match)]) and
    (.Match getposition[lastmove(.Match) + 1]
      piece_moves[KING pieceof[lastmove(.Match) + 1]]
      forall[.EPiece samecolor .SPiece]) and
    (agent(.Match getmove[lastmove(.Match)]) = KNIGHT)
  ]
  consume
```