U2R2 – Wie alles funktioniert

Timo Weithöner
im Juli 2008

Architektur

- Kern: Ein relationales DBMS, dass sehr spezialisiert ist.
  - Alle Relationen haben genau drei nutzbare Spalten
  - Nach Außen enthalten alle Spalten Strings (von maximal 100 Zeichen).
  - Intern enthalten die Relationen Long Ids, die die Strings repräsentieren.
  - Eine Hashtabelle besorgt die Verknüpfung zwischen Ids und Strings (Konsequenz: Bereichsanfragen sind nicht performant möglich und werden daher nicht unterstützt).

Architektur II

- Auf dem DB-Kern sitzt ein Regelprozessor (RuleTrigger)
  - Prädikate in den Regeln entsprechen Relationen
  - Konjunktionen werden auf Joins abgebildet
  - Dann erfolgt eine Projektion auf den Regelkopf.

  - Die Regeln werden solange immer wieder abgearbeitet wie dabei neue Tupel entstehen.
  - Die neuen Tupel werden in den Relationen persistent gespeichert.
  - Also: „Total forward chaining and materialization“

Architektur III

- Kern und Regelinterpreter werden durch eine Implementierung der OWLReasoner-Klasse aus der OWLAPI gekapselt (sind aber dennoch auch direkt zugänglich).
  - Bei Laden von OWL-Axiomen werden diese automatisch in die entsprechenden Tupel der verschiedenen Relationen übersetzt und
  - Der Regelprozessor gestartet.
  - Danach liegen alle ableitbaren Fakten persistent in der Datenbank vor. Für Abfragen ist dann keine aufwendige Berechnung mehr nötig.

  - Achtung: Die OWLAPI lädt die komplette Ontologie in den Speicher. Bei sehr grossen Ontologien kann das scheitern. In diesem Fall sollte man versuchen die Axiome häppchenweise in den Reasoner schieben.

Der RuleGenerator

- Relativ unabhängig vom Reasoner steht der RuleGenerator.
- Er compiliert Regelsätze in ihre für den RuleTrigger auswertbare Form (Java-Klassen). Dies geschieht in mehreren Schritten:
  - Logic Program (LP) => Relational Algebra Program (RAP)
  - RAP => Java-Code
  - Java-Code => Java-Byte-Code (Klassen) verpackt in einem JAR.
- Der RuleGenerator erstellt außerdem eine Java-Factory-Klasse, die der RuleTrigger lädt (Das Regel-JAR muss im Classpath sein; die Factory-Klasse muss in der RuleTrigger.ini-Datei benannt sein). Mit dieser Factory-Klasse kann der RuleTrigger dann alle Regeln laden.
- Der RuleGenerator wird über ein eigenes ini-File parametrisiert.

Die Grammatik des LP

- Programm = {Regel}
- Regel = [Ruletag {, Ruletag}":"]
  Praedikat ":-" Praedikat {"," Praedikat} "."
- Praedikat = ID "(" VarLit ["," VarLit ["," VarLit]] ")"
- Ruletag = ID
- ID = Kleinbuchstabe {GrossOderKleinbuchstabe}
- VarLit = Variable | Literal | "_"
- Variable = Grossbuchstabe {Grossbuchstabe}
- Literal = """ {BeliebigesZeichenOhneAnführungszeichen} """

- Kommentare in "/*" und "*/" sind an allen Stellen erlaubt.

Was hat es mit den Ruletags auf sich?

- Vor jeder Regel kann eine Menge von Ruletags stehen.
- Diese Tags dienen dazu aus einem LP nur bestimmte Regelmengen zu verwenden.
- So können z.B. bestimmte Regeln als „Experimentell“ gekennzeichnet werden und dann gezielt ein oder ausgeschaltet werden.
- Es ist so auch möglich in einem LP-File Regeln für verschiedene Sprachfragmente (OWL-R DL, DL-lite) unterzubringen und zwischen diesen Fragmenten umzuschalten ohne den ganzen Regelsatz auszutauschen.
- Details hierzu finden sich in den Kommentaren in **RuleTrigger.ini**, wo die nötigen Einstellungen auch vorgenommen werden.

Wozu das RAP

- Hat „historische“ Gründe. „Früher“ habe ich den Regelsatz so programmiert.
- Das RAP zeigt genau wie eine Regel abgearbeiten wird. Dabei ist es noch übersichtlicher als der Java-Code. Es ist also zu Debug-Zwecken durchaus interessant.
- Ausserdem kann man hier evtl. noch Optimierungen vornehmen, wenn der RuleGenerator ein unvorteilhaftes RAP erstellt hat.
- Eine Regel und das RAP-Statement
  - subC(X, Y) :- subC(X, T), subC(T, Y) wird als RAP:
  - subC :- (JOIN subC subC L1=R0 (L0, R1, -))

^Operation

^Beteiligte Relation (kann auch das Erg. einer anderen Operation sein

^Join-Bedingung zweite Spalte der linken = erste Spalte der rechten Relation

^Projektion (welche Spalten bilden die Erg.-Relation)

Noch eine Anmerkungen

- Man sollte immer bedenken, dass U2R2 bottom-up abeitet.
- Das folgende Programm ist für Prolog kein großes Problem:
  - gross(X, Y) :- type(X, _), type(Y, _).
  - test("OK") :- gross("hund", "katze").
- Angenommen „type“ enthält 10.000 Fakten so müsste Prolog im Wesentlichen zweimal in diesen 10.000 Fakten suchen.
- U2R2 hingegen würde die Relation „gross“ komplett berechnen und dabei 10.000x10.000 = 100.000.000 Tupel generieren und auf Platte schreiben.
- Man sollte also nicht allen Quatsch in das LP-File schreiben. Besonders, wenn eine Regel zu einem Kreuzprodukt führt (wie im Beispiel) ist Vorsicht geboten.

RuleGenerator - Verwendung

- Der RuleGenerator ist in net.weithoener.xu2r2.RuleGenerator implementiert. Alle für den RuleGenerator nötigen Klassen befinden sich in „libs/u2r2.jar“. Ausserdem wird „tools.jar“ aus dem JDK benötigt (s.u.).
- Die main-Methode erwartet als einziges Argument einen String, der den Pfad zum toplevel ini-Files (s.u.) angibt.
- Der RuleGenerator erwartet im ini-Verzeichnis (im toplevel ini-File definiert) eine eigene ini-Datei (RuleGenerator.ini), die alle weiteren Konfigurationen enthält.
- RuleGenerator benötigt Java >= 1.6.0, um die den erstellten Java-Code selbst zu compilieren und zu einem JAR zu packen. Dann muss auch „tools.jar“ aus dem JDK im CLASSPATH sein. Sollte eine entsprechende Java-Version nicht verfügbar sein, müssen die entsprechenden Operationen im ini-File deaktiviert werden und im Anschluss an den RuleGenerator-Aufruf „von Hand“ durchgeführt werden.
- Fußnote: „net.weithoener.**x**u2r2“ ist kein Tippfehler sondern der Tatsach geschuldet, dass \u, unicode und JavaCC eine tödliche Kombination ist. jjtree erzeugt in einem Kommentar einen Pfadnamen der Datei mit „\“ als Separator. Das dabei entstehende „...ner\u2r2\...“ wird dann von javacc als irgendein unicode-escape-code „\u“ interpretiert, was zu einem Syntaxfehler führt. Andere haben ähnliche Probleme: googlen!

Die Relationen

- Letztlich ist es U2R2 relativ egal wie die Regeln, die ausgewertet werden aussehen und wie die Prädikate/Relationen heißen.
- Allerdings muss U2R2 die OWL-Axiome ja in irgendwelche Relationen schreiben. Und die Relationennamen in den Regel sollten dann natürlich zu diesen Relationen passen. Hier wird also das Schema festgelegt.
- Im Folgenden zunächst kurz wie OWL-Axiome in Tupel übersetzt werden, dann die Liste der verwendeten Relationen.

OWL => Tupel

- Gegeben das Axiom
  - `subClassOf(A, hasValue(inverseOf(P), I))`
- Dieses Axiom wird zunächst in mehrere Axiome zerlegt.
  - `subClassOf(A, _anon1)`
  - `equivalentClass(_anon1, hasValue(_anon2, I)`
  - `equivalentProperty(_anon2, inverseOf(P)).`
- Dabei werden allerdings keine echten EquivalentXY Axiome erzeugt ...

OWL => Tupel

- ... vielmehr geht der Regelsatz von Prädikaten aus, die die Äquivalenz implizit enthalten.
  - `equivalentClass(_anon1, hasValue(_anon2, I) equivalentProperty(_anon2, inverseOf(P)).`
- Wird also zu
  - `hasValue(_anon1, _anon2, I). inverseOf(_anon2, P).`
- Oder vielmehr (tatsächlicher Name der Relation):
  - `hvR(_anon1, _anon2, I). invP(_anon2, P).`
- Hinweis:Es kann nicht immer die volle Semantik der Äquivalenz ausgewertet werden (Bsp. AllValuesFrom oder SomeValuesFrom)

OWL => Tupel

- Zusammengefasst wird also:

```
subClassOf(A, hasValue(inverseOf(P), I))
```

- Übersetzte in die folgenden Tupel:

```
subC(_anon2, P).
hvR(_anon1, _anon2, I).
invP(_anon2, P).
```

OWL => Tupel

- Unions und Intersections werden zu Head-Tail-Listen
  - `subClassOf(A, unionOf(B, C, D)) wird zu`
  - ```
    subClassOf(A, _anon3)
    equivalentClass(_anon3, unionOf(B, _anon4))
    equivalentClass(_anon4, unionOf(C, D))
    ```
- Wobei im Regelsatz wiederum Prädikate mit „eingebauter Equivalenz“ verwendet werden
  - ```
    subC(A, _anon3)
    unionOf(_anon3, B, _anon4))
    unionOf(_anon4, C, D))
    ```
-

OWL => Tupel

- Nominals werden zu einer Union von neuen Konzepten, die jeweils exakt ein Individuum beschreiben.
  - `subClassOf(N, oneOf(I1, I2, I3)).`
- Wird also zu
  - ```
    subC(N, _anon5).
    unionof(_anon5, _anon6, _anon7).
    unionof(_anon7, _anon8, _anon9).
    exactly(_anon6, I1).
    exactly(_anon8, I2).
    exactly(_anon9, I3).
    ```
- Es kann für exactly(C, I) nicht type(I, C) verwendet werden, da für die Regelauswertung festgehalten werden muss, dass C aus genau einem Individuum gebildet ist.
    ```

Welche Relationen gibt es also?

- **subC(A,B), subP(A,B)**
  A ist Subklasse/-Property von B
- **eqC(A,B), eqP(A,B), disjointC(A,B), disjointP(A,B)**
  A und B sind equivalente/disjunkte Klassen/Properties
- **compl(A,B)**
  A und B sind komplementär
- **type(I,C)**
  I ist Instanz der Klasse C
- **pI(P, I, F)**
  Individuen I und F sind via Property P miteinander verbunden.
- **same(A,B), different(A,B)**
  A und B sind dasselbe/unterschiedliche Individuen

Mehr Relationen

- **union(A, B, C), intersection(A, B, C)**
  A ist die Vereinigung/der Schnitt von B und C
- **invP(A, B)**
  Properties A und B sind inverse Properties.
- **symP(P), asymP(P), reflP(P), irreflP(P), funcP(P), invFuncP(P), trans(P)**
  Property P ist symmetrisch, asymmetrisch, reflexiv, ...
- **avfR(R, P, C), svfR(R, P, C), hvR(R, P, I)**
  R ist AllValuesFrom-/SomeValuesFrom-/HasValueRestriction bzgl. Property P und Klasse C/Individuum I.
- **Inconsistent** und **unsatisfiable**
  Werden gefüllt, wenn bei der Regelauswertung Inkosistenzen oder unerfüllbare Konzepte erkannt werden. Dies erledigen übrigens auch Regeln!

Relationennamen? Brauch ich die jemals?

- Ja, um Regeln zu schreiben.
- Hin und wieder hilft es (zur Fehlersuche) auch die Relationen auszugeben. Das ist übrigens ziemlich einfach:
  - U2R2 u2r2 = new U2R2(...);
    ...
    System.out.println(u2r2.getRelation("subC"));
- Die Relationennamen gefallen mit nicht. Kann ich sie ändern?
  - Ja, das geht. Allerdings muss man dazu in den Java-Code. Die verwendeten Relationennamen sind als Konstanten in net.weithoener.u2r2.Settings definiert. Dort können sie geändert werden.
  - Und man muss dann natürlich auch den Regelsatz „übersetzen“.

Starten des Reasoners

- Wie gesagt, U2R2 implementiert einen OWLReasoner. Man muss also eigentlich nur die OWLAPI kennen und kann mit dem System arbeiten. Einzig die Parameter des Konstruktors müssen dazu bekannt sein.
- `public OWLReasoner(OWLOntologyManager ontologyManager, String propertiesFile, boolean clean)`
  - `OntologyManager: Der verwendete OntologyManager`
  - `PropertiesFile: Der Pfad zum toplevel ini-File`
  - `Clean: Falls „true“ startet der Reasoner „clean“. D.h. Alle Fakten, die in vorherigen Läufen des Systems persistent gespeichert wurden werden gelöscht. Falls „false“ sind diese Fakten weiterhin zugänglich.`
- Und wenn ich den Reasoner ohne zu programmieren einfach von der Kommandozeile starten will?
  - Das geht nicht! U2R2 spricht nur OWLAPI und muss daher in ein anderes Programm eingebunden werden.

Konfiguration

- Zum Starten von U2R2 müssen alle JARs aus dem „libs“-Verzeichnis der Installation im Java-Classpath enthalten sein.
- Außerdem muss das JAR, das den zu verwendenden Regelsatz enthält, im Classpath sein. Ein mitgelieferter Regelsatz befindet sich in „rules/mini/mini.jar“. Andere Regelsätze können mit dem RuleGenerator (s.o.) compiliert werden.

- Wie auf der vorigen Seite gezeigt erhält U2R2 eigentlich nur einen Konfigurationsparameter: Nämlich die Lokation des sog. toplevel ini-Files.

Das toplevel ini-File

- Das toplevel ini-File enthält genau drei Werte:
  - AP_RPATH_BASE: Das Verzeichnis zu dem alle anderen Parameter in diesem und in anderen ini-Files, die mit AP_RPATH beginnen und damit einen Pfad enthalten, relativ sind. Wenn also
    - AP_RPATH_BASE = f:/u2r2/ und
      AP_RPATH_CLEAN_INI = ini/activeSet
    - Verweist AP_RPATH_CLEAN_INI auf das Verzeichnis f:/u2r2/ini/activeSet
  - AP_RPATH_INI_DIR: Das Verzeichnis in dem die vom Reasoner verwendeten ini-Files gespeichert werden sollen. Achtung dieser Satz von ini-Files sollte nie geändert werden. Es kann sonst passieren, dass das System bei einem Neustart seine Datenbank nicht mehr findet ...
  - AP_RPATH_CLEAN_INI: Dieses Verzeichnis muss beim ersten Start von U2R2 die gültigen ini-Files enthalten. Diese Files können geändert werden. Eine Änderung der ini-Dateien muss von einem Neustart mit clean=true gefolgt sein, damit die Änderung aktiv werden. Dies bedeutet aber auch, dass die Datenbank dann gelöscht wird.

Die anderen ini-Files

- Alle weiteren ini-Files und deren Parameter sind in den, in der Installation enthaltenen Vorlagen, beschrieben.
- Folgende weitere ini-Files gibt es:
  - Logger.ini: Einstellungen bzgl. des Loggings (nicht Locking!)
  - PageManager.ini: Einige wichtige Einstellungen: Z.B. die Größe der Datenbank und die Menge des maximal zu verwendenden Hauptspeichers (jeweils in Seiten).
  - RelationManager.ini: Hier besonders wichtig, die Anzahl der Relationen und URIs (Strings in den Relationen), die verwaltet werden können.
  - RuleTrigger.ini: Angaben zum zu verwenden Regelsatz
- Darüber hinaus gibt es ein separates ini-File für den RuleGenerator (RuleGenerator.ini). Achtung: Die Einstellungen hierin werden nicht vom RuleTrigger übernommen. Die Factory-Klasse muss also in beiden ini-Files angegeben werden.

Persistenz

- U2R2 sorgt automatisch für Persistenz aller Daten im System!
- Persistenz wird erreicht indem alle Daten in so genannten Pages (net.weithoener.u2r2.storage.pagemgr.Page) abgelegt werden.
- Die Page Klasse ist abstrakt und wird z.B. von den Knoten der BBäume implementiert.
- Seiten werden in der Regel über PageRef-Objecte (net.weithoener.u2r2.storage.pagemgr.PageRef) angesprochen. Die PageRef enthält die Information, ob eine Seite im Hauptspeicher und/oder auf der Platte liegt, ihre Adresse auf dem jeweiligen Medium und eine eindeutige ID.
- Ein Verweis von einem Bbaum-Knoten auf seine Kinder ist also über PageRefs gelöst. Dies stellt sicher, dass der Baum auch dann wieder aufgebaut werden kann, wenn Seiten ausgelagert wurden oder der Reasoner inzwischen neu gestartet wurde.

Der PageManager

- Für das Auffinden und Laden von Seiten ist der PageManager (net.weithoener.u2r2.storage.pagemgr.PageManager) zuständig.
- Der PageManager verfügt über eine Anzahl von Methoden, die zu einer PageRef das entsprechende Page-Object liefern.
- Zu beachten ist, dass die Seiten dabei immer mit einem Lock versehen werden, der auch wieder entfernt werden muss, da andere Routinen sonst nicht auf diese Seiten zugreifen können.
- Ein Page-Objekt sollte nach der Rückgabe des Locks nicht mehr verwendet werden, da nicht sichergestellt werden kann, dass die darin enthaltenen Daten dann noch aktuell sind (Die Seite könnte inzwischen ausgelagert und wieder geladen worden sein. Es wäre dann ein neues Object für dieselbe Seite entstanden dass inzwischen auch andere Daten enthalten kann).

Speicherplatz und Swapping

- Vom PageManager wird beim Start ein Array von Pages angelegt. Die Größe dieses Arrays legt fest wieviele Seiten maximal im Hauptspeicher gehalten werden können.
- Werden mehr Seiten benötigt, werden Seiten, die im Moment nicht so wichtig sind, persistent auf Platte gespeichert (swapping) und aus dem Hauptspeicher entfernt so das Platz für weitere Seiten entsteht.

- U2R2 vermerkt während vieler Operationen, wie wichtig ihm bestimmte Seiten zur Zeit sind. So wird versucht zu vermeiden, dass z.B. bei einem BBaum Knoten, die nah an der Wurzel liegen, aus dem Hauptspeicher verdrängt werden. Diese Knoten werden periodisch gebraucht würden von einem LRU-Ansatz aber zu schnell verdrängt.

Fußnote zur Persistenz

- Wird U2R2 an der OWLAPI vorbei verwendet, muss vor dem Beenden des Systems U2R2.save() aufgerufen werden, um sicherzustellen, dass alle nötigen Daten persistent auch Platte geschrieben sind.
- Wird U2R2 nur über die in OWLReasoner definierten Methoden angesprochen erfolgt dies automatisch.

Das Datenmodell der Relationen

- Relationen werden in BBäumen (net.weithoener.u2r2.storage.btree.BTree) abgelegt.
- Bbäume sind eigentlich Indexstrukturen und Indizes werden ja auf einzelne Spalten (oder Spaltengruppen) angelegt. Sie dienen aber in der Regel nicht dazu die Nutzdaten der Relationen zu speichern.
- In U2R2 sind die Tupel aber sehr kurz (5 Longs), so dass eigentlich nichts dagegen spricht diese direkt im Index zu speichern. Der Bbaum enthält in seinen Knoten also nicht nur die Schlüsselattribute sondern, dass komplette Tupel.
- Ein wenig verschwenderisch wird dieser Ansatz, wenn man berücksichtigt, dass bis zu drei Bbäume nötig sind, um eine Relation voll zu indizieren. Dann sind die Nutzdaten dreimal im Speicher. Dies erhöht die Performance aber dermaßen, dass dies in kauf genommen wird.

Der RelationManager

- ... verwaltet in einer Hashtabelle die Zuordnung von Relationennamen auf die eindeutige ID der Seite, die den Wurzelknoten des entsprechenden BBaumes enthält.
- Ausserdem wird eine Hashtabelle verwaltete, die die Stringliterale (die Werte in den Tupeln; z.B. URIs) den Longs zuordnet, die dann tatsächlich in den Tupeln gespeichert werden.
- Diese Hashtabellen (net.weithoener.u2r2.storage.pagemgr.hash.Table) werden natürlich auch Persistent gespeichert (natürlich ebenfalls mittels Pages).
- Die maximale Größe der Tabellen wird im ini-File angegeben.
- Die maximale Länge der HashEntries kann nur im Sourcecode geändert werde!

Regelauswertung

- Regeln werden Schrittweise sooft ausgeführt bis keine neuen Fakten mehr generiert werden.
- Dabei werden in Schritt N nur diejenigen Regeln ausgeführt, die in Ihrem Rumpf Prädikate haben, die sich in Schritt N-1 verändert haben.
- Dabei wird die sog. Delta-Iteration verwendet. Das heißt für ein Prädikat P in Schritt N berechneten Tupel werden zunächst nur in einer Hilfsrelation AUX^P_N gespeichert. An den Stellen an denen P verwendet werden müsste wird $Delta^P_N$ verwendet.

  Nach dem Auswertungsschritt wird:

  - $Delta^P_{N+1} = AUX^P_N \setminus P$
  - $P = P \cup AUX^P_N$
  - $AUX^P_{N+1} = \emptyset$

Regelauswertung II

- Dabei gilt es zu beachten, dass einzelne Regeln in einem Auswertungsschritt evtl. mehrfach ausgeführt werden müssen. Dies ist der Fall, wenn ...
- Eine Regel in ihrem Rumpf mehrere Prädikate hat, die im vorherigen Schritt M-1 verändert wurden (gilt auch, wenn mehrfach das selbe Prädikat auftritt). Im Schritt M gilt dann:
  - $H :- C_1, ..., C_N, U_1, ..., U_K.$
    Mit $Delta^{C1}_M, ..., Delta^{CN}_M \neq \emptyset$ und $Delta^{U1}_M, ..., Delta^{UK}_M = \emptyset$
  - Die Regel muss N-mal ausgeführt werden wobei für Ausführung I (0<I<=N) folgende Regel ausgewertet wird:
  - $AUX^H_M :- C_1, ..., C_{I-1}, Delta^{CI}_M, C_{I+1}, ..., C_N, U_1, ..., U_K.$

Warum so kompliziert?

- Ganz einfach: Die Joins bleiben erheblich kleiner.
  - Angenommen type enthält 1.000.000 Tuple subC 10.000. Im letzten Schritt sind zu type 1.000 und zu subC 100 neue Tupel hinzugekommen.
  - type(I, C) :- type(I, SUB), subC(SUB, C).
  - Klassisch wird ein Join von 1.000.000 mit 10.000 Tupeln berechnet. Maximal 10.000.000.000 neue Tupel.
  - Mit Delta-Iteration erhalten wir zwei Joins
    - 1,000,000 mit 100 Tupeln (max. 100.000.000 neue Tupel) und
    - 1.000 mit 10.000 Tupeln (max. 10.000.000 neue Tupel)
  - Maximal 10 Milliarden neue Tupel stehen also Maximal 110 Millionen neue Tupel gegenüber. Faktor 100 weniger Speicherbedarf und ggf. weniger Platten-IO.
  - Außerdem können tatsächlich nur 110 Millionen neue Tupel entstehen. Das heißt im klassischen Ansatz ist höchstens jedes hundertste Tupel neu. Alle anderen müssen beim Einfügen in die Relation verworfen werden.

Noch Fragen?

- Dann Fragen:
  - t.weithoener@gmx.net
  - 0731 1658226