nRQL Tutorial

Racer

Systems & Co. KG

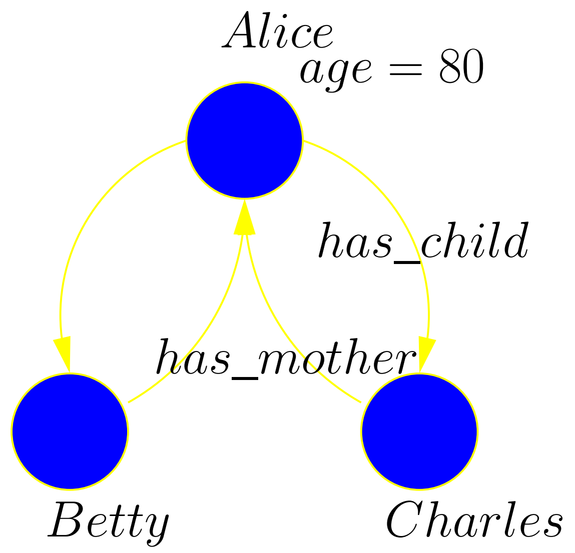
Blumenau 50
22089 Hamburg Germany

www.racer-systems.com

Overview of Tutorial

- The nRQL Query Language
  - Introductory examples - why nRQL?
  - Syntax (query atoms, variables, . . .)
  - Querying OWL Documents
  - Semantics, . . .
- The nRQL Query Processing Engine
  - Incremental Query Processing
  - Configurable Completeness
  - Simple Rules
  - Query Reasoning, Optimization, . . .
- Querying Large OWL KBs
  - The Lehigh University Benchmark (LUBM)

Motivating Simple Example



$mother(alice), age(alice) = 80,$

$has\_mother(betty, alice),$

$has\_mother(charles, alice),$

$mother(betty), mother(betty)$

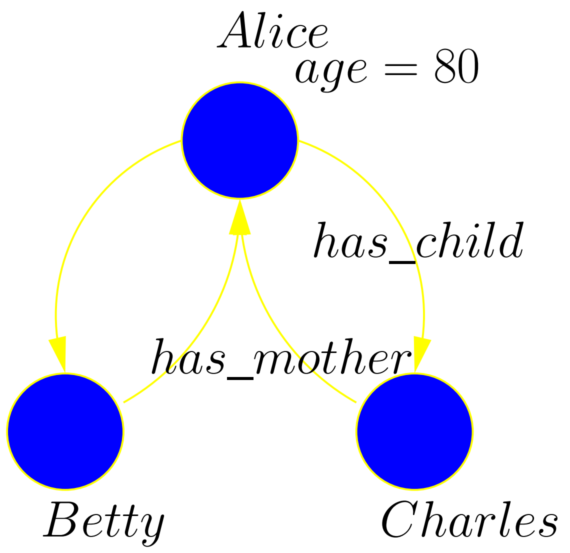
- How do we retrieve pairs of siblings (e.g., $\{(betty, charles)\}$)?
- Not expressible as DL instance retrieval query!

⊖ Solution 1: write a search program

⊕ Solution 2: use nRQL:

```
> (retrieve (?x ?y)
            (and (has-child ?z ?x)
                 (has-child ?z ?y)))
```

Motivating Simple Example (2)



$mother(alice),\ age(alice) = 80,$

$has\_mother(betty, alice),$

$has\_mother(charles, alice),$

$mother(betty),\ mother(betty)$

- How do we retrieve <u>pairs</u> of siblings (e.g., $\{(betty, charles)\}$)?
- Not expressible as DL instance retrieval query!

⊖ Solution 1: write a search program

⊕ Solution 2: use nRQL:

```
> (((?X CHARLES) (?Y BETTY))
   ((?X BETTY) (?Y CHARLES)))
```

nRQL Language – Overview

- Compositional syntax and semantics
- Compound/complex queries build from query atoms, using boolean connectors
- Query atoms contain objects = variables and/or individuals
- Syntax for variables: `?x`, `$?x`
- Syntax for individuals: `betty`, …
- Queries have a head and a body `(retrieve <head> <body>)`
- `<head>` might also contain special projection operators …
- … or generalized ABox assertions (nRQL rules)

Example KB

```
;;; Role Declarations

(define-primitive-role has-ancestor
              :inverse has-descendant :transitive t)

(define-primitive-role has-parent
              :inverse has-child
              :domain person :range person)

(define-primitive-attribute has-mother
              :parent has-parent)

(define-primitive-attribute has-father
              :parent has-parent)

(define-concrete-domain-attribute age :type cardinal)
```

Example KB (2)

```
;;; Concept Axioms

(disjoint man woman)
(implies man person)
(implies woman person)
(implies person (or man woman))
(implies person (and human (an age)))

(equivalent mother
            (and woman
                 (some has-child person)))

(equivalent grandmother
             (and mother
                  (some has-child mother)))
```

Example KB (3)

```
;;; The ABox

(instance alice mother)
(instance betty mother)

(related betty   alice has-mother)
(related charles alice has-mother)

(constrained alice alice-age age)
(constraints (= alice-age 80))

(instance charles man)
(instance james (not woman))
```

nRQL Query Atoms

- Concept Query Atoms
    - With variables in head and body

      ```
      ? (retrieve (?x) (?x woman))
      > ( ((?X ALICE)) ((?X BETTY)) )
      ```
    - With variables in body and empty head

      ```
      ? (retrieve () (?x woman))
      > T
      ```
    - With individuals in body

      ```
      ? (retrieve () (betty woman))
      > T
      ```

nRQL Query Atoms (2)

- Role Query Atoms
  - With variables

```
? (retrieve (?x ?y)
            (?x ?y has-child))
> (((?X ALICE) (?Y BETTY))
   ((?X ALICE) (?Y CHARLES)))
```

  - With variables and individuals

```
? (retrieve (?x)
            (alice ?x has-child))
> (((?X BETTY)) (?X CHARLES)))
```

nRQL Query Atoms (3)

- Constraint Query Atoms
  - Querying the Concrete Domain part of an ABox
  - Who has parents of same age?
  - We add `(related charles james has-father)`, `(constrained james james-age age)`, `(constraints (= james-age 80))`

```
? (retrieve (?x)
      (?x ?x (constraint
            (has-father age)
            (has-mother age) = ))
> (((?X CHARLES)))
```

nRQL Query Atoms (4)

- Constraint Query Atoms (2)
  - Who has a father that is at least 40 years older?
  - We add `(constrained charles charles-age age)`, `(constraints (= charles-age age 38))`

```
? (retrieve (?x)
      (?x ?x (constraint
               (has-father age) age
               (< (+ 40 age-2) age-1))))
> (((?X CHARLES)))
```

  - nRQL also supports role chains ended by attributes, e.g.
    - `(has-child has-father age)`

Head Projection Operators

- How to retrieve values from the CD, e.g. the age of Alice?
- Projection operators to fillers of CD attributes or OWL datatype properties

```
? (retrieve (?x (AGE ?x))
       (?x (and grandmother (an age))))
> (((?X ALICE) ((AGE ?X) (ALICE-AGE))))
```

- Projection to told values of the CD

```
? (retrieve (?x (TOLD-VALUE (AGE ?x)))
       (?x (and grandmother (an age))))
> (((?X ALICE) ((:TOLD-VALUE (AGE ?X))
  (80))))
```

nRQL Variables

- nRQL employs the active domain assumption (ADA): variables must always be bound to ABox individuals
- nRQL offers two kinds of variables

```
? (retrieve (?x ?y) (and (?x man) (?y man)))
> (((?X JAMES) (?Y CHARLES))
   ((?X CHARLES) (?Y JAMES)))
```

  - RQL employs the unique name assumption (UNA) for variables
  - The UNA can be turned off:

```
? (retrieve ($?x $?y)
            (and ($?x man) ($?y man)))
> (...(($?X JAMES) ($?Y JAMES))
```

nRQL Query Atoms (5)

- Other atoms
  - Bind a variable to an individual with `same-as`

```
? (retrieve (?x)
       (same-as ?x betty))
> (((?X BETTY)))
```

  - Check for the existence of role successors:

```
? (retrieve (?x)
       (?x (has-known-successor has-father)))
> (((?X CHARLES)))
```

nRQL Peculiarities

- The meaning of queries with individuals in head

```
? (retrieve (betty)
            (betty woman))
> ((($?BETTY BETTY)))
```

  - Explanation: query is rewritten into

```
(AND (SAME-AS $?BETTY BETTY)
     ($?BETTY WOMAN))
```

  - `$?BETTY` is a variable that does not obey the <u>unique name assumption for variables</u>
  - `(SAME-AS $?BETTY BETTY)` enforces binding of `$?BETTY` to `BETTY`

Complex nRQL Queries

- Compound nRQL queries are defined inductively
  - Every query atom `ai` is a body.
  - If `a1 ...an` are query bodies, then the following expressions are also bodies
    - `(neg ai)`
    - `(and a1 ...an)`
    - `(union a1 ...an)`
    - `(project-to (objects-in-ai) ai)`
- Each variable creates a new axis in an n-dimensional tuple space

$\Rightarrow$ Variable names matter

Complex nRQL Queries (2)

- More on union queries:

```
? (retrieve (?x)
        (union (?x woman) (?y man))) =
  (retrieve (?x)
        (union (and (?x woman) (?y top))
                    (and (?x top) (?y man)))) =
  (retrieve (?x) (union (?x woman)
                        (?x top))) =
  (retrieve (?x) (?x top))
> (((?X BETTY)) ((?X ALICE))
   ((?X CHARLES)) ((?X JAMES)))
```

  - Can be tricky!

nRQL Peculiarities (2)

- “Negation as Failure (NAF)” with `neg`:

```
? (retrieve (?x) (neg (?x woman)))
> ((?X CHARLES) (?X JAMES))
```

  - In this example,
    `(retrieve (?x) (?x (not woman)))`
    yields the same answer, but not in general

```
? (retrieve (?x) (neg (?x foobar)))
> (((?X CHARLES)) ((?X JAMES))
   ((?X BETTY)) ((?X ALICE)))
? (retrieve (?x) (?x (not foobar)))
> NIL
```

  - Also for role query atoms

```
(retrieve (?x ?y) (neg (?x ?y has-child)))
```

nRQL Peculiarities (3)

- “Negation as Failure (NAF)” (2)
  - “NAF” for atoms with individuals can be tricky

```
(retrieve (betty)
          (neg (betty woman)))
=
(retrieve ($?betty)
    (neg (and ($?betty woman)
              (same-as $?betty betty))))
=
(retrieve ($?betty)
    (UNION (neg ($?betty woman))
           (neg (same-as $?betty betty))))
```

(due to DeMorgan’s Law)

nRQL Peculiarities (4)

- DL-like “true” negation:
  - Different from “NAF” negation
  - For which individuals can Racer prove they are not woman?

```
? (retrieve (?x) (?x (not woman)))
> (((?X CHARLES)) ((?X JAMES)))
```

  - For which pairs of individuals can Racer prove they are not in the has-father relationship?

```
? (retrieve (?x ?y)
      (?x ?y (not has-father)))
> (((?X CHARLES) (?Y ALICE))
   ((?X CHARLES) (?Y BETTY)))
```

nRQL Peculiarities (5)

- “Pseudo-nominals” for concept query atoms:
  - treat atomic concept BETTYNOM as pseudo-nominal, referring to individual betty

  -
    ```
    (retrieve (?x)
        (?x (some has-child BETTYNOM)))
    ```
- Defined queries:

  -
    ```
    (defquery mother
              (?x ?y)
              (and (?x woman)
                   (?x ?y has-child)))
    (retrieve (?x ?child)
              (?x ?child mother))
    ```

nRQL Peculiarities (6)

- The projection operator `project-to` for query bodies - why is it needed?
  - Use the previously defined query `mother` to retrieve woman with known children:

```
? (retrieve (?x ?x) (?x ?y mother))
> (((?X ALICE) (?Y CHARLES))
  (((?X ALICE) (?Y BETTY))))
```

  - If we are just interested in the bindings of `?X`, we can also supply `NIL` as a parameter to the defined query `mother`:

```
? (retrieve (?x ) (?x nil mother))
> (((?X ALICE)))
```

nRQL Peculiarities (7)

- The projection operator `project-to` for query bodies - why is it needed? (2)
  - How do we retrieve the complement of the previous query?
  - For atoms, this is what NAF achieves!
  - Thus we try:

```
? (retrieve (?x )
        (NEG (?x nil mother)))
> (((?X CHARLES)) ((?X JAMES))
   ((?X ALICE)) ((?X BETTY)))
```

  - What went wrong?

nRQL Peculiarities (8)

- The projection operator `project-to` for query bodies - why is it needed? (3)

  -
    ```
    (retrieve (?x )
        (neg (?x nil mother)))
    =   (neg (and (?x woman)
                  (?x ?y-ano has-child)))
    =   (union (neg (?x woman))
               (neg (?x ?y-ano has-child)))
    =   (union (and (neg (?x woman))
               (?y-ano top))
               (neg (?x ?y-ano has-child)))
    ≈   (?x top)
    ```

nRQL Peculiarities (9)

- The projection operator `project-to` for query bodies - why is it needed? (4)
  - The “problem” is that `neg` preserves the arity: the complement set returned by `neg` has the same arity as the argument set
  - Solution: first project to `?x`, and then build the complement:

```
? (retrieve (?x )
      (neg (project-to (?x)
                 (?x nil mother))))
> (((?X CHARLES)) ((?X JAMES))
   ((?X BETTY)))
```

Querying OWL KBs

- Support for querying OWL datatype properties:

```
<owl:Class rdf:ID="Person">
  <rdfs:label>person</rdfs:label>
</owl:Class>

<owl:DatatypeProperty rdf:ID="age">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range rdf:resource=
    "http://www.w3.org/2001/XMLSchema#integer" />
</owl:DatatypeProperty>

<Person rdf:about="http://www.test.com/michael">
   <age>34</age>
<Person>
```

Querying OWL KBs (2)

- Support for querying OWL datatype properties (2):

```
? (retrieve
    (?x
      (datatype-fillers
         (|http://www.test.com/test.owl#age| ?x)))
    (?x (some |http://www.test.com/test.owl#age|
                (and (min 30) (max 35)))))

> (((?X |http://www.test.com/michael|)
   ((:TOLD-VALUE
    (|http://www.test.com/test.owl#age| ?X)) (34))))
```

- Extended Racer concept syntax (expressions like `(and (min 30) (max 35))` only recognized by nRQL)

Querying OWL KBs (3)

- Support for retrieval of values of OWL annotation properties from OWL documents

```
<owl:DatatypeProperty rdf:ID="annot1">
  <rdfs:range rdf:resource=
     "http://www.w3.org/2001/XMLSchema#string"/>
  <rdf:type rdf:resource=
     "http://www.w3.org/2002/07/owl#
                          AnnotationProperty"/>
</owl:DatatypeProperty>
```

- A special head projection operator `annotations` is provided by nRQL
- Similar to querying for datatype properties

nRQL - Syntax

Let $a, b \in \mathcal{O}$; C be an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ concept expression, R a nRQL role expression (a nRQL role expression is either a $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ role expression, or a negated $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ role expression); P one of the concrete domain expressions offered by Racer; and f, g be so-called attributes (whose range is defi ned to be one of the available concrete domains offered by Racer). Then, the set of nRQL atoms is given as follows:

- Unary concept query atoms: $C(a)$
- Binary role query atoms: $R(a, b)$
- Binary constraint query atoms: $P(f(a), g(b))$
- Binary same-as atoms: $same\_as(a, i)$
- Unary has-known-successor atoms: $has\_known\_successor(a, R)$
- Negated atoms: If A is a nRQL atom, then so is $\backslash(A)$, a so-called negation as failure atom or simply negated atom.

nRQL - Syntax (2)

A nRQL Query has a head and a body. Query bodies are defined inductively as follows:

- Each nRQL atom A is a body; and
- If $b_1 \ldots b_n$ are bodies, then the following are also bodies:
  - $b_1 \wedge \cdots \wedge b_n$, $b_1 \vee \cdots \vee b_n$, $\backslash(b_i)$

We use the syntax $body(a_1, \ldots, a_n)$ to indicate that $a_1, \ldots, a_n$ are all the object names ($a_i \in \mathcal{O}$) mentioned in $body$. A nRQL Query is then an expression of the form

$$ans(a_{i_1}, \ldots, a_{i_m}) \leftarrow body(a_1, \ldots, a_n),$$

The expression $ans(a_{i_1}, \ldots, a_{i_m})$ is also called the head, and $(i_1, \ldots, i_m)$ is an index vector with $i_j \in 1 \ldots n$. A conjunctive nRQL query is a query which does not contain any $\vee$ and $\backslash$ operators.

nRQL - Semantics

Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ knowledge base.
A positive ground query atom A is logically entailed (or implied) by $\mathcal{K}$ iff every model $\mathcal{I}$ of $\mathcal{K}$ is also a model of A. In this case we write $\mathcal{K} \models A$. Moreover, if $\mathcal{I}$ is a model of $\mathcal{K}$ (A) we write $\mathcal{I} \models \mathcal{K}$ ($\mathcal{I} \models A$). We therefore have to specify when $\mathcal{I} \models A$ holds. In the following, if the atom A contains individuals i, j, it will always be the case that $i, j \in \mathsf{inds}(\mathcal{A})$. From this it follows that $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and $j^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, for any $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $\mathcal{I} \models \mathcal{K}$:

- If $A = C(i)$, then $\mathcal{I} \models A$ iff $i^{\mathcal{I}} \in C^{\mathcal{I}}$.
- If $A = R(i, j)$, then $\mathcal{I} \models A$ iff $(i^{\mathcal{I}}, j^{\mathcal{I}}) \in R^{\mathcal{I}}$.
- If $A = P(f(i), g(j))$, then $\mathcal{I} \models A$ iff $(f^{\mathcal{I}}(i^{\mathcal{I}}), g^{\mathcal{I}}(j^{\mathcal{I}})) \in P^{\mathcal{I}}$.
- If $A = same\_as(i, i)$, then $\mathcal{I} \models A$.
- If $A = same\_as(i, j)$, then $\mathcal{I} \not\models A$.
- If $A = has\_known\_successor(i, R)$, then $\mathcal{I} \models A$ iff for some $j \in \mathsf{inds}(\mathcal{A})$: $\mathcal{I} \models R(i, j)$.

nRQL - Semantics (2)

Let $ans(a_{i_1}, \ldots, a_{i_m}) \leftarrow body(a_1, \ldots, a_n)$ be a nRQL query q such that $body$ is in NNF. Let $\beta(a_i) =_{def} x_{a_i}$ if $a_i \in \mathcal{I}$, and a_i otherwise; i.e., if a_i is an individual we replace it with its representative unique variable which we denote by x_{a_i}. Let $\mathcal{K}$ be the knowledge base to be queried, and $\mathcal{A}$ be its ABox. The answer set of the query q is then the following set of tuples:

$$\{\, (j_{i_1}, \ldots, j_{i_m}) \mid \exists j_1, \ldots, j_n \in \mathsf{inds}(\mathcal{A}), \forall m, n, m \neq n : j_m \neq j_n, \\ \mathcal{K} \models_{NF} \alpha(body)_{[\beta(a_1) \leftarrow j_1, \ldots, \beta(a_n) \leftarrow j_n]} \,\}$$

Finally, we state that $\{()\} =_{def}$ TRUE and $\{\} =_{def}$ FALSE.

Features of the nRQL Engine

- Cost-based optimizer
- Compilation of queries possible (similar to LISP implementations of Prolog)
- Different query processing modes
  - Set-at-a-time mode ("Get all tuples")
  - Tuple-at-a-time mode ("Get next tuple")
    - Uses LISP processes

    $\Rightarrow$ more than one active (running) query possible
    - Lazy: compute next tuple if requested
    - Eager: precompute next tuple(s)

Features of the nRQL Engine (2)

- Cost-based optimizer
- Compilation of queries possible (similar to LISP implementations of Prolog)
- Different query processing modes
- Degree of completeness configurable (next slide)
- Non-recursive defined queries (macro queries)
- Simple rule engine

Features of the nRQL Engine (3)

- Degree of completeness configurable
  - Told information (very incomplete)
  - Told information + exploited TBox information (much more complete)
  - Complete Racer ABox Retrieval (expensive!)
- $3 \times \#\{set\_at\_a\_time, tuple\_at\_a\_time\} = 6$
- Variations: realize ABox / classify TBox (or not)
- 7th tuple-at-a-time mode: “two-phase processing”
  - Phase 1: deliver cheap tuples (incomplete)
  - Warn user; then, if next tuple requested, start
  - Phase 2: use full ABox reasoning to deliver remaining tuples (complete)

Incremental Query Answering

TBox:			**ABox :**
$person$	$\sqsubseteq$	$\top$	$spouse(doris)$
man	$\sqsubseteq$	$person$	$spouse(betty)$
$woman$	$\sqsubseteq$	$person$	$man(adam)$
$spouse$	$\doteq$	$woman \sqcap$	$woman(eve)$
		$(\exists married\_to.man)$	$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-next-tuple :query-1)`

$\Rightarrow$ `((?X DORIS))`

Incremental Query Answering

TBox:			ABox :
$person$	$\sqsubseteq$	$\top$	$spouse(doris)$
man	$\sqsubseteq$	$person$	$spouse(betty)$
$woman$	$\sqsubseteq$	$person$	$man(adam)$
$spouse$	$\doteq$	$woman \sqcap$	$woman(eve)$
		$(\exists married\_to.man)$	$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-next-tuple :query-1)`

$\Rightarrow$ `((?X BETTY))`

Incremental Query Answering

TBox:			$\mathrm{ABox}:$
$person$	$\sqsubseteq$	$\top$	$spouse(doris)$
man	$\sqsubseteq$	$person$	$spouse(betty)$
$woman$	$\sqsubseteq$	$person$	$man(adam)$
$spouse$	$\doteq$	$woman \sqcap$	$woman(eve)$
		$(\exists married\_to.man)$	$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-next-tuple :query-1)`

$\Rightarrow$ `:WARNING-EXPENSIVE-PHASE-TWO-STARTS`

Incremental Query Answering

TBox:			ABox :
$person$	$\sqsubseteq$	$\top$	$spouse(doris)$
man	$\sqsubseteq$	$person$	$spouse(betty)$
$woman$	$\sqsubseteq$	$person$	$man(adam)$
$spouse$	$\doteq$	$woman \sqcap$	$woman(eve)$
		$(\exists married\_to.man)$	$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-next-tuple :query-1)`

$\Rightarrow$ `((?X EVE))`

Incremental Query Answering

TBox:			ABox :
$person$	$\sqsubseteq$	$\top$	$spouse(doris)$
man	$\sqsubseteq$	$person$	$spouse(betty)$
$woman$	$\sqsubseteq$	$person$	$man(adam)$
$spouse$	$\doteq$	$woman \sqcap$	$woman(eve)$
		$(\exists married\_to.man)$	$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-next-tuple :query-1)`

$\Rightarrow$ `:EXHAUSTED`

Incremental Query Answering

TBox:		
$person$	$\sqsubseteq$	$\top$
man	$\sqsubseteq$	$person$
$woman$	$\sqsubseteq$	$person$
$spouse$	$\doteq$	$woman \sqcap$
		$(\exists married\_to.man)$

ABox :

$spouse(doris)$

$spouse(betty)$

$man(adam)$

$woman(eve)$

$maried\_to(eve, adam)$

- `(retrieve (?x) (?x spouse))`

$\Rightarrow$ `(:QUERY-1 :RUNNING)`

- `(get-answer :query-1)`

$\Rightarrow$ `(((?X DORIS)) ((?X BETTY)) ((?X EVE)))`

Features of the nRQL Engine (5)

- Reasoning with Queries
  - Incomplete for full nRQL, but still useful
  - Complete for restricted nRQL
  - Query consistency check
  - Query entailment check (subsumption)

  ⇒ maintenance of a “Query repository” lattice (similar to a TBox)

  ⇒ use cached tuples of queries in repository for optimization purposes (“materialized views”)
  - Semantic optimization: query “realization” (similar to ABox realization)

    ⇒ add implied conjuncts to enhance informdness of backtracking search

Features of the nRQL Engine (6)

- Defined queries (simple Macro-mechanism)

  -
    ```
    (defquery mother
              (?x ?y)
              (and (?x woman)
                   (?x ?y has-child)))

    (defquery mother-with-male-child
              (?x ?child)
              (and (:substitute
                     (mother ?x ?child))
                   (?child man)))
    ```

Features of the nRQL Engine (7)

- Simple rule mechanism

  -
    ```
    (defrule
       (and (?x woman) (?y man) (?x ?y married))
            (neg (?x (:has-known-successor has-child)))
       ((instance (new-ind child-of ?x ?y) human)
        (instance ?x mother)
        (instance ?y father)
        (related (new-ind child-of ?x ?y) ?x
                 has-mother)
        (related (new-ind child-of ?x ?y) ?y
                 has-father)))
    ```

⇒ Rule antecedence is a query body; consequence is a list of generalized ABox assertions (also operators like `new-ind`)

Features of the nRQL Engine (8)

- Complex TBox queries
  - What are the child concepts of the concept `woman`?

```
> (tbox-retrieve (?x ?y)
       (and (?x woman)
            (?x ?y has-child)))
> (((?X WOMAN) (?Y SISTER))
   ((?X WOMAN) (?Y AUNT))
   ((?X WOMAN) (?Y *BOTTOM*))
   ((?X WOMAN) (?Y MOTHER))
   ((?X WOMAN) (?Y GRANDMOTHER)))
```

- …most of the present nRQL features have been requested by users
- …for more nRQL peculiarities: see manual

Querying Large OWL KBs

- Lehigh University Benchmark for benchmarking semantic web repositories
- See `http://www.lehigh.edu/˜yug2/Research/` `SemanticWeb/LUBM/LUBM.htm`
- Modeling of a university
  - OWL (DAML+OIL) classes for `departments`, various kinds of `professors`, `students`, . . .
  - roles like `worksFor`, `subOrganization` (transitive),
  - Datatype properties `telephone, age,` . . .
- Benchmark generator generates “ABoxes”
- 14 benchmarking queries

The LUBM (2)

```
Query 9: (retrieve
            (?x ?y ?z)
            (and (?x Student)
                 (?y Faculty)
                 (?z Course)
                 (?x ?y advisor)
                 (?x ?z takesCourse)
                 (?y ?z teacherOf)))
```

“Retrieve all triples `<?x,?y,?z>` such that `?x` is (bound to) a student undertaking a course `?z` whose teacher `?y` (from the faculty) happens to be his/her advisor”

The LUBM (3)

```
Query 12: (retrieve
                (?x ?y www.University0.edu)
                (and (?x chair)
                     (?y Department)
                     (?x ?y memberOf)
                     (?y www.University0.edu
                         subOrganizationOf)))
```

Cite LUBM: “The benchmark data do not produce any instances of class Chair. Instead, each Department individual is linked to the chair professor of that department by property headOf. Hence this query requires realization, i.e., inference that that professor is an instance of class Chair because he or she is the head of a department.”

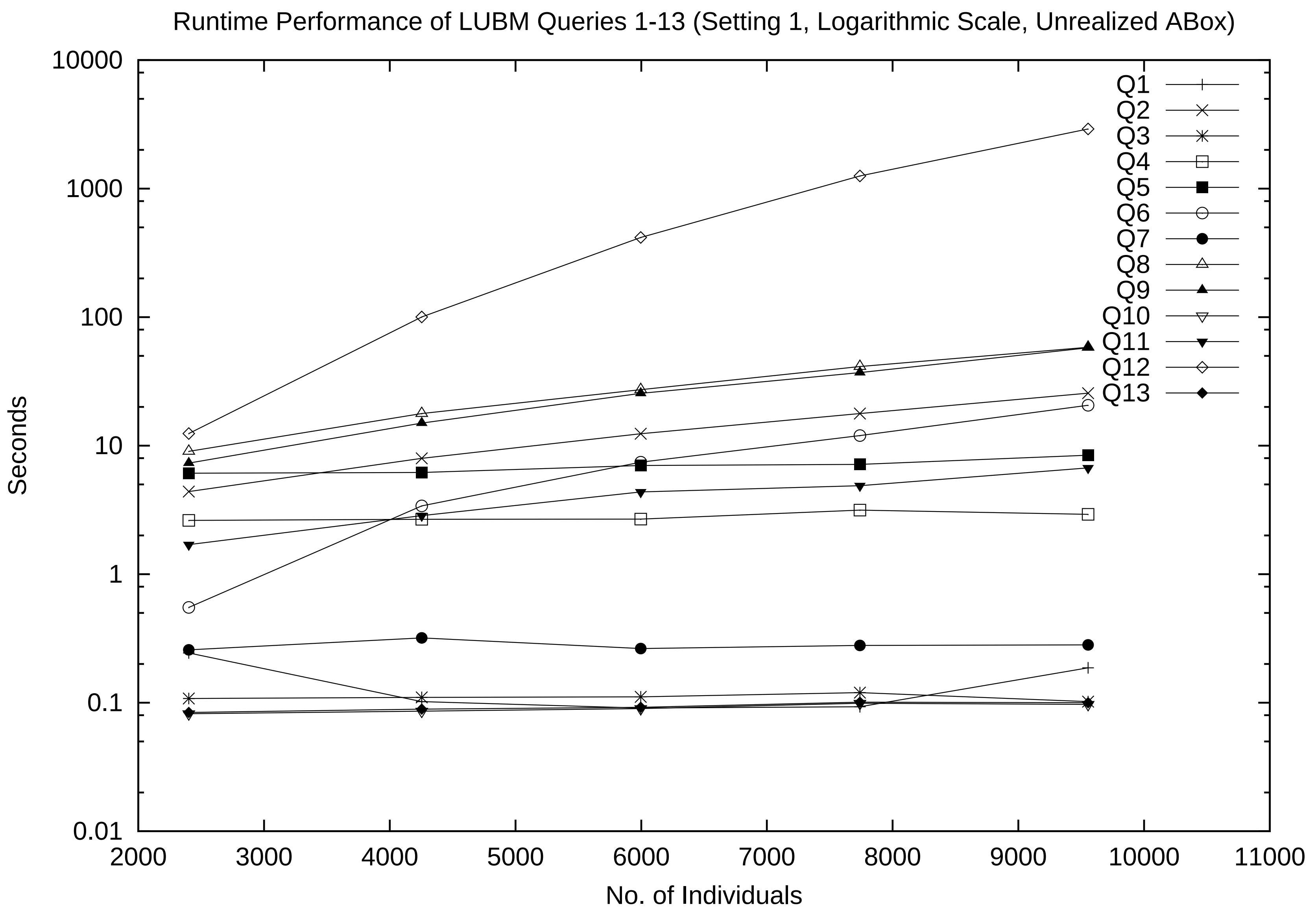
Benchmarking Racer + nRQL

- We ran LUBM queries in 3 settings:
- Setting 1: complete ABox querying using an unrealized ABox
- Setting 2: complete ABox reasoning using a realized ABox
- Setting 3: "told information querying" enhanced with TBox information – "upward saturation":

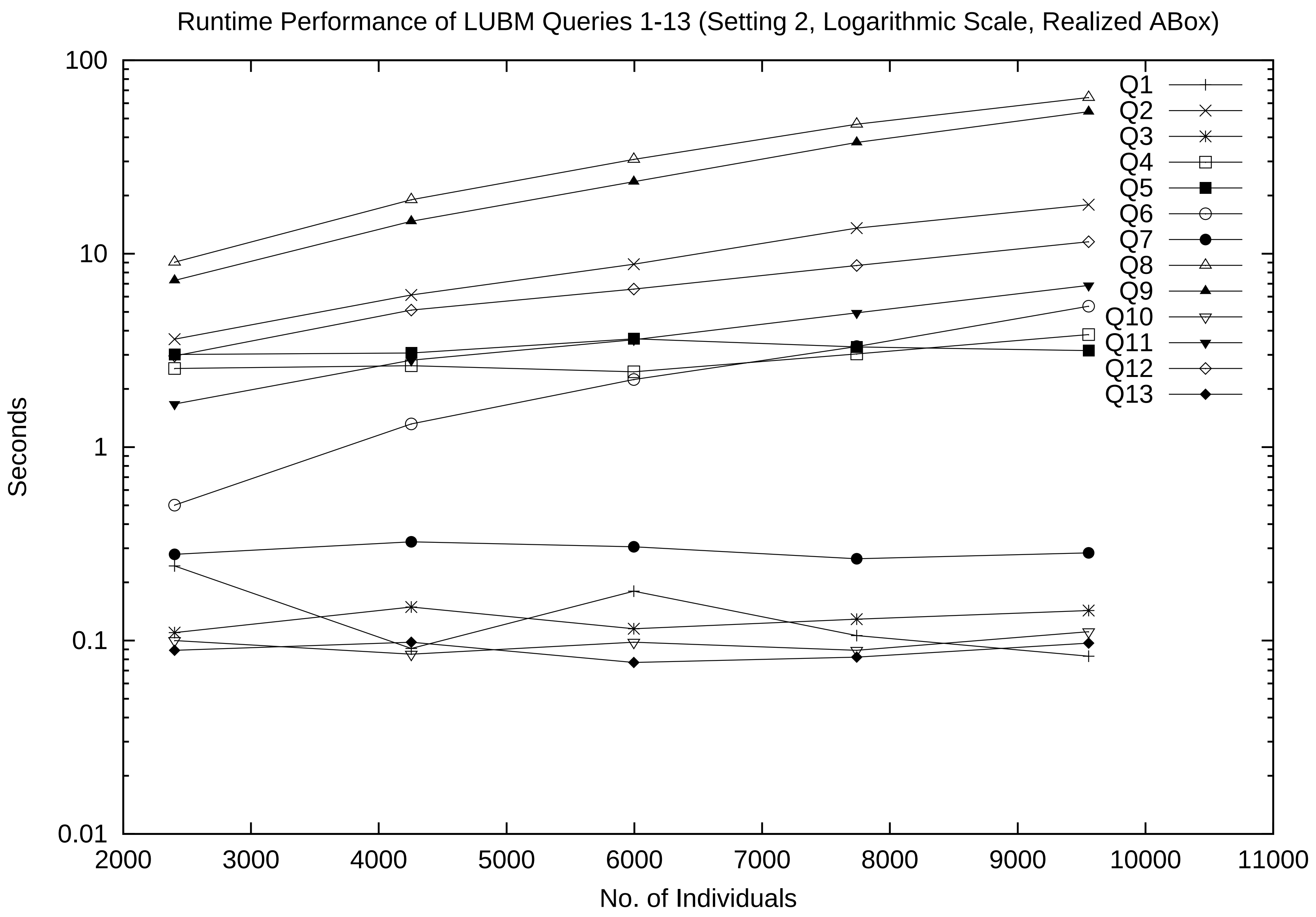
$\Rightarrow$ for each ABox axiom $C(i) \in \mathcal{A}$, for all $D \in \mathsf{concept\_ancestors}(C, TBox)$: put $D(i)$ into "ABox": $\mathcal{A} := \mathcal{A} \cup \{D(i)\}$

$\Rightarrow$ same for role relationships due to role hierarchies

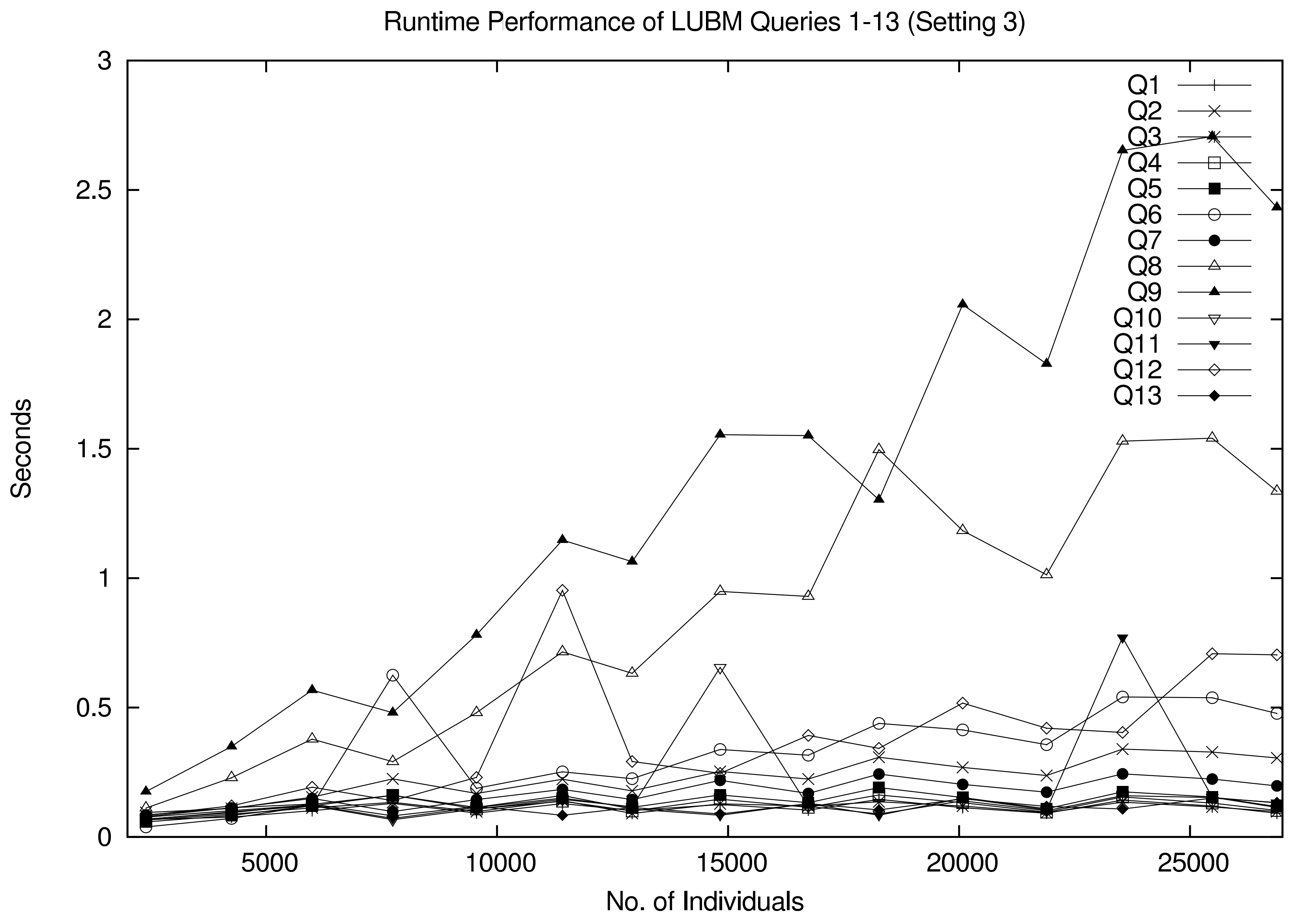
Results - Setting 1



Results - Setting 2



Results - Setting 3



Evaluation

- Using complete ABox querying we have to stop at approx. 10.000 LUBM individuals
- Initial ABox consistency test kills Racer
- If completeness is sacrificed, we can easily load and process more than 150.000 individuals
- All but Q8 and Q9 can be answered in fractions of a second
- Only one tuple is missed (for Q12)

$\Rightarrow$ not severely incomplete

$\Rightarrow$ in this scale, answering time is quite okay!

Thanks

for your

attention!