Efficient and Convenient Search on **Very** Large Knowledge Bases

Lecture @ Reasoning Web Summer School
Luxembourg, September 22, 2018

Hannah Bast
Algorithms & Data Structures Group
Department of Computer Science
University of Freiburg, Germany

Introduction 1/4

- Knowledge Bases

The predicate is actually called
<Country of nationality>

  - A knowledge base can be represented as a collection of subject-predicate-object triples, for example:

<Neil Armstrong>	<Profession>	<Astronaut>
<Neil Armstrong>	<Nationality>	<USA>
<Liu Yang>	<Profession>	<Astronaut>
<Liu Yang>	<Nationality>	<China>
<Liu Yang>	<Gender>	<Female>

  - Identifiers refer to some entity or concept, for example:

    a particular person, a particular profession, a relation, …

  - Crucial: unique identifier for each entity or concept
  - Knowledge bases need not be "complete" (and typical are not)

Introduction 2/4

- The standard query language is SPARQL
  - A simple example query

```
SELECT ?person {
  ?person  <Profession>  <Astronaut> .
  ?person  <Gender>      <Female>
}
```

  - Returns a list of all **female astronauts** in the KB
  - Since no particular order is specified (see next slide), the results can be in any order
  - The order is then usually the order of the internal IDs of result entities … more about these IDs later

Introduction 3/4

- The standard query language is SPARQL
  - A slightly more complex example query

```
SELECT ?person ?nationality ?birthdate WHERE {
  ?person       <Profession>      <Astronaut> .
  ?person       <Gender>          <Female> .
  ?person       <Nationality>     ?nationality .
  ?person       <Date_of_birth>   ?date_of_birth .
  ?nationality  <Contained_by>    <Eurasia>
}
ORDER BY DESC(?date_of_birth)
```

  - Returns: all female astronauts from Eurasia, with their nationality and birth date, youngest first

    Note: the result is a **table**

Introduction 4/4

- Demos
  - The demos are from two of our research prototypes

    QLever a full-featured SPARQL engine + UI

    Broccoli a predecessor for a subclass of SPARQL
  - We will see more of them in the following
  - And also understand what is going on behind the scenes

    I will explain **fundamental** concepts and techniques of working with knowledge bases … as well as some of the specific developments behind QLever and Broccoli

Part 0: Introduction

Part 1: Knowledge Base Basics

Part 2: Indexing and Query Processing

Part 3: Combination with Text Search

Part 4: SPARQL User Interfaces

Knowledge Base Basics 1/11

a string object
is called a "literal"

- IDs and Literals
  - In the introduction, we saw these example triples:

    <Neil Armstrong> <Profession> <Astronaut>
    <Neil Armstrong> <Nationality> <USA>

  - This was a simplified example, actual knowledge bases use IDs and extra predicates for names and descriptions

wd:Q1615	wdt:P106	wd:Q11631
wd:Q1615	wdt:P27	wd:Q30
wd:Q1615	rdfs:label	"Neil Armstrong"@en
wd:Q11631	rdfs:label	"Astronaut"@en

  - The prefixes stand for URI prefixes, e.g. wd:Q1615 stands for <http://www.wikidata.org/entity/Q1615>

Knowledge Base Basics 2/11

- Languages
  - General-purpose knowledge bases are often multilingual, with names in many different languages

wd:Q90	rdfs:label	"Paris"@en	English
wd:Q90	rdfs:label	"Baariis"@so	Somali
wd:Q90	rdfs:label	"Bahliz"@za	Zhuang
wd:Q90	rdfs:label	"Bă-là̤"@cdo	Min Dong
wd:Q90	rdfs:label	"IParisi"@zu	Zulu
wd:Q90	rdfs:label	"Lutetia"@la	Latin
wd:Q90	rdfs:label	"Pa-lí"@nan	Min Nan
wd:Q90	rdfs:label	"Paarii"@kbp	Kabiye

… and so on (275 languages)

Knowledge Base Basics 3/11

- Reification of **n-ary** predicates as triples
  - Simple relations are easily cast in triple form

Douglas Adams	spouse	Jane Belson
wd:Q42	wdt:P26	wd:Q14623681

  - For complex "n-ary" information, we need intermediate entities

wd:Q42	p:P26		wds:Q42-b88670f8-456b-...
wds:Q42-b88...	pq:P580	start time	"1991-11-25"@xsd:dateTime
wds:Q42-b88...	pq:P582	end time	"2001-05-11"@xsd:dateTime
wds:Q42-b88...	pqv:P580		wdv:1c30ade7914d07287...
wds:Q42-b88...	pqv:P582		wdv:c8ae0d38443d4671d...
wdv:1c30ade...	wb:calendar		wd:Q1985727
wdv:1c30ade...	wb:precision		11 exact to the day
wdv:1c30ade...	wb:timeValue		"1991-11-25"@xsd:dateTime

https://en.wikibooks.org/wiki/SPARQL/WIKIDATA_Qualifiers,_References_and_Ranks

Knowledge Base Basics 4/11

- Reification in Wikidata
  - **Statement** entities are connected to "normal" entities

    They lead to ALL complex information about an entity
    (triples with the statement entity as subject)
  - **Value** and **qualifier** entities are connected to statement entities

    Multiple triples with value or qualifier entity as subject
  - **Provenance** entities are connected to statement entities

    Multiple triples with provenance entity as subject
  - **Rank** entity: exactly ONE per statement entity

    PreferredRank: if you want just one statement, take this
    DeprecatedRank: wrong or outdated statements
    NormalRank: all other statements

Knowledge Base Basics 5/11

- Wikidata's most important prefixes

wd:	http://www.wikidata.org/entity/
wdt:	http://www.wikidata.org/prop/direct/
wds:	http://www.wikidata.org/entity/statement/
wikibase:	http://wikiba.se/ontology-beta#
p:	http://www.wikidata.org/prop/
ps:	http://www.wikidata.org/prop/statement/
psv:	http://www.wikidata.org/prop/statement/value/
pq:	http://www.wikidata.org/prop/qualifier/
pqv:	http://www.wikidata.org/prop/qualifier/value/
rdfs:	http://www.w3.org/2000/01/rdf-schema#
schema:	http://schema.org/
prov:	http://www.w3.org/ns/prov#

Knowledge Base Basics 6/11

B = Billion

- Dimensions of the KBs used in this presentation
  - **Freebase**

    Started by Metaweb in 2007, acquired by Google in 2010

    1.9B triples, 125M entities, 345M literals

    Freebase Easy

    Easy-to-use curated version from Bast et al, WWW'14

    0.4B triples, 60M entities, 11M literals (English only)
  - **Wikidata**

    Latest data dump (3.5B triples without n-ary relations)

    7.1B triples, 862K entities, 418K literals

this is **huge**

Knowledge Base Basics 7/11

- Cross products in results
  - Since SPARQL results are always tables, there are often "cross-product" effect

```
SELECT ?person ?profession ?nationality WHERE {
  ?person  <Profession>   ?profession .
  ?person  <Nationality>  ?nationality
}
```

Arnold Schwarzenegger	Actor	Austria
Arnold Schwarzenegger	Bodybuilder	Austria
Arnold Schwarzenegger	Politician	Austria
Arnold Schwarzenegger	Actor	USA
Arnold Schwarzenegger	Bodybuilder	USA
Arnold Schwarzenegger	Actor	USA

excerpt of results

Knowledge Base Basics 8/11

- OPTIONAL

```
SELECT ?person ?profession ?nationality  WHERE {
  ?person <is-a> <Person> .
  OPTIONAL { ?person <Profession> ?profession } .
  OPTIONAL { ?person <Nationality> ?nationality }
}
```

  - Without the OPTIONAL, this would only display persons who have at least one profession and nationality in the KB

    This corresponds to an "**inner join**" ... see Part 2

  - With the OPTIONAL, get all persons and empty cells in the result table, if there is no match for the respective triple

    This corresponds to an "**outer join**" ... see Part 2

Knowledge Base Basics 9/11

- GROUP BY and ORDER BY
  - A simple example

```
SELECT ?profession (COUNT(?person) AS ?count) WHERE {
  ?person <is-a> <Person> .
  ?person <Profession> ?profession
}
GROUP BY ?profession
ORDER BY DESC(?count)
```

  - Groups all people in the knowledge base by profession and counts the number of people in each group
  - Outputs the profession and the count, largest count first

Knowledge Base Basics 10/11

- GROUP BY and ORDER BY
  - A more complex example

```
SELECT  ?profession
        (AVG(?height) AS ?average_height)
        (COUNT(?x) AS ?count) WHERE {
  ?x <is-a> <Person> .
  ?x <Height> ?height .
  FILTER (?height < 3) .
  ?x <Profession> ?profession
}
GROUP BY ?profession
ORDER BY DESC(?average_height)
HAVING (?count > 100)
```

  - Professions ordered by average height, only groups > 100

Knowledge Base Basics 11/11

Height	P2048
Metre	Q11573
Instance of	P31
Human	Q5
Occupation	P106
Gender	P21

- Queries quickly become huge in SPARQL
  - **Average height by profession and gender**

```
SELECT ?occupation ?gender (AVG(?height) AS ?average_height)
                           (COUNT(?height) AS ?count)
WHERE {
  ?x wdt:P31 wd:Q5 .
  ?x wdt:P21 ?gender_id . ?gender_id rdfs:label ?gender .
  FILTER langMatches(lang(?gender), "en") .
  ?x p:P2048 ?statement . ?statement psv:P2048 ?value .
  ?value wikibase:quantityNormalized ?quantity .
  ?quantity wikibase:quantityUnit wd:Q11573 .
  ?quantity wikibase:quantityAmount ?height . FILTER (?height < 3) .
  ?x wdt:P106 ?occupation_id . ?occupation_id rdfs:label ?occupation .
  FILTER langMatches(lang(?occupation), "en")
}
GROUP BY ?occupation ?gender
HAVING (?count > 100)
ORDER BY DESC(?average_height)
```

With QLever:
~ 1sec on
full Wikidata

Wikidata
Query Service:
Timeout
after 20s

Part 0: Introduction

Part 1: Knowledge Base Basics

Part 2: Indexing and Query Processing

Part 3: Combination with Text Search

Part 4: SPARQL User Interfaces

Indexing 1/12

these IDs have nothing to do with the string IDs used by some knowledge base

(e.g. Q30 for USA in Wikidata)

- IDs
  - Any sensible index data structure for a knowledge base will first transform all entities / predicates / literals to integer IDs

<Neil Armstrong>	<Profession>	<Astronaut>
<Neil Armstrong>	<Nationality>	<USA>
<Neil Armstrong>	<Date_of_birth>	"1930-08-05"
<Liu Yang>	<Profession>	<Astronaut>
<Liu Yang>	<Nationality>	<China>

  - If this were our complete KB, the IDs would be:

<Astronaut>	#1	<Neil Armstrong>	#6
<China>	#2	<Profession>	#7
<Date of birth>	#3	<USA>	#8
<Liu Yang>	#4	"1930-08-05"	#9
<Nationality>	#5		

We give IDs in lexicographic order of the words

That way, we can efficiently implement ORDER BY by sorting IDs, not strings

Indexing 2/12

- Values
  - We map values to an intermediate mantissa-exponent representation which we then map to IDs like on the previous slide

KB name	Intermediate repr.	ID
"3.1459"@xsd:double	"00E3.14590000"	#16
"42"@xsd:integer	"01E4.20000000"	#17
"1000000"@xsd:integer	"06E1.00000000"	#18

  - Then again, we can efficiently implement ORDER BY by simply sorting internal integer IDs

Indexing 3/12

<Astronaut>	#1
<China>	#2
<Date of birth>	#3
<Liu Yang>	#4
<Nationality>	#5
<Neil Armstrong>	#6
<Profession>	#7
<USA>	#8

- Naïve Index
  - Conceptually, our knowledge base is now an array of integer triples (usually huge, Wikidata = 7.1 **billion** triples)

    S=#6 P=#7 O=#1
    S=#6 P=#5 O=#8
    S=#6 P=#3 O=#9
    S=#4 P=#7 O=#1
    S=#4 P=#5 O=#2

    Here and on the following slides, we prefix IDs with a # and S, P, O depending on their position in the triple

    This is just for the sake of explanation, what is actually stored are just integer IDs

Indexing 4/12

- Permutations
  - We store all six permutations of our array of triples
  - We call these permutations **SPO**, **SOP**, **PSO**, **POS**, **OPS**, **POS**
  - E.g. SPO = sort by subject, then predicate, then object

SPO	PSO
S=#4 P=#5 O=#2	P=#3 S=#6 O=#9
S=#4 P=#7 O=#1	P=#5 S=#4 O=#2
S=#6 P=#3 O=#9	P=#5 S=#6 O=#8
S=#6 P=#5 O=#8	P=#7 S=#4 O=#1
S=#6 P=#7 O=#1	P=#7 S=#6 O=#1

  - Note: with 8 Bytes per ID and the 10B triples from our version of Wikidata, this requires 6 x 3 x 8 x 10B Bytes = **1.4 TB**

that's **Tera**bytes

Indexing 5/12

<China>	#2
<Liu Yang>	#4
<Nationality>	#5
<Neil Armstrong>	#6
<USA>	#8

- Why the permutations?
  - For SPARQL queries with a single triple, we now get the result by a simple scan of the right index

```
SELECT ?person ?nationality WHERE {
  ?person <Nationality> ?nationality
}
```

  - **P**redicate is fixed, **S**ubject and **O**bject are variables
    → take the PSO or POS permutation → let's take PSO

P=#3	S=#6	O=#9
P=#5	S=#4	O=#2
P=#5	S=#6	O=#8
P=#7	S=#4	O=#1
P=#7	S=#6	O=#1

Result table (IDs):

#4	#2
#6	#8

Result table (names):

Liu Yang	China
Neil Armstrong	USA

Efficient, because we can read it off from a **range**

Indexing 6/12

<Astronaut>	1
<China>	2
<Date of birth>	3
<Liu Yang>	4
<Nationality>	5
<Neil Armstrong>	6
<Profession>	7
<USA>	8

- Order is important
  - With more than one triple, ordering becomes more critical

```
SELECT ?person ?nationality ?profession WHERE {
  ?person <Nationality> ?nationality .
  ?person <Profession> ?profession
}
```

  - If we take the PSO index, we get two sets of triples already ordered by **S**ubject, so that we can easily **join** those

P=#3	S=#6	O=#9
P=#5	S=#4	O=#2
P=#5	S=#6	O=#8
P=#7	S=#4	O=#1
P=#7	S=#6	O=#1

For fixed P ordered by S

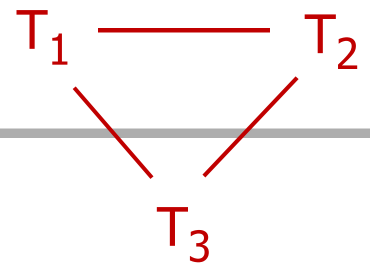
Result table (IDs):

#4	#2	#1
#6	#8	#1

Result table (names):

Liu Yang	China	Astronaut
Neil Armstrong	USA	Astronaut

Indexing 7/12



- Query planning
  - For complex queries, the order of the operations, in particular of the **joins**, is crucial for an efficient query processing

    SELECT ?person1 ?person2 ?movie WHERE {
    ?person1 <Film performance> ?film . T_1
    ?person2 <Film performance> ?film . T_2
    ?person1 <Spouse> ?person2 T_3
    }

  - General principle:

    joins are cheaper if the respective columns are sorted

    1. Build query graph: nodes = triples, edge if shared variable
    2. The nodes correspond to basic SCAN operations
    3. An edge corresponds to a JOIN operation
    4. Intermediate results are tables (of various widths)

Indexing 8/12

Is this a good query plan?

No, it's terrible, because the intermediate results from 2. will be **huge** (32M rows)

- Query planning

  SELECT ?person1 ?person2 ?movie WHERE {
  ?person1 <Film performance> ?film . T_1
  ?person2 <Film performance> ?film . T_2
  ?person1 <Spouse> ?person2 T_3
  }

  – One possible query plan:

  1. Process T_1 and T_2 (in any order) use POS scan (join on ?film)
  2. Join tables from T_1 and T_2 → table with three columns
  3. Sort result from 2. by ?person1 by ?person2 would also work
  4. Process T_3 use PSO scan (join on ?person1)
  5. Join results from 3. and 4. → table with three columns

Indexing 9/12

Is this a good query plan?

Yes, because only few spouses per person, hence result from 2. not much larger than result from T_1

- Query planning

  SELECT ?person1 ?person2 ?movie WHERE {
  ?person1 <Film performance> ?film . T_1
  ?person2 <Film performance> ?film . T_2
  ?person1 <Spouse> ?person2 T_3
  }

  – A better query plan

  1. Process T_1 and T_3 (in any order) — PSO (join on ?person1)
  2. Join tables from T_1 and T_3 — → table with three columns
  3. Sort by ?person2, then ?film — Or: by ?film, then ?person2
  4. Process T_2 — PSO (join on ?person2 **and** ?film)
  5. Join results from 3. and 4. — This is a **two-column** join

Indexing 10/12

- Cost estimation
  - How do we know which query plan is better, without executing them all and measuring the time?
  - Standard procedure: we estimate the cost of the various operations, for example:

    SCAN: we have to perform the scans anyway, so we might as well do all of them and determine the exact cost

    SORT: cost estimate $n \cdot \log n$, where n = #rows

    JOIN: we assume that the join columns have been sorted before → cost estimate #rows table 1 + #rows table 2
  - Bottom line: we need to estimate the size (#rows) of the intermediate results

Indexing 11/12

multiplicities play a major role in result size estimation for SPARQL queries

- Size estimation
  - Example 1: each table with two columns, join on first

Input Table 1

#14	#15
#38	#42
#57	#13

Input Table 2

#14	#97
#57	#55

Result Table

#14	#15	#97
#57	#13	#55

result has **two** rows

  - Example 2: same column dimensions, also join on first

Input Table 1

#57	#15
#57	#42
#57	#13

Input Table 2

#57	#97
#57	#55

Result Table

#57	#15	#97
#57	#15	#55
#57	#42	#97
#57	#42	#55
#57	#13	#97
#57	#13	#55

result has **six** rows

Indexing 12/12

- Simultaneous size and multiplicity estimation
  - Input table 1: $\mathbf{s_1}$ = #rows, $\mathbf{d_1}$ = #distinct values in join column
  - Input table 2: $\mathbf{s_2}$ = #rows, $\mathbf{d_2}$ = #distinct values in join column
  - Average multiplicity is related to #distinct values as follows:

    $m_1 = s_1 / d_1$ and $m_2 = s_2 / d_2$
  - Size estimate of the result table

    $s = \alpha \cdot m_1 \cdot m_2 \cdot \min(d_1, d_2)$

    $\alpha = 1$ would mean:

    all elements from smaller table occur in larger table
  - We also need to estimate the multiplicity of **each column** in the result table (not only the join column)

    $m_i = ???$

    this is tricky to even formulate
    see the QLever paper
    if you are interested

    (which, of course, you are)

Efficiency 1/3

- Details about the index lists
  - There are lot more implementation details which I did not mention, but which are critical for performance / usability:
  - Here are a few keywords:

    Store the SOP etc. indices such that for each kind of join, the SCAN ops have to scan only **exactly** what they need

    Keep as little as necessary in RAM, the rest on disk

    Use compression on disk (for faster reading)

    Resolve internal IDs to names as late as possible … next slide

Efficiency 2/3

- How / where to store the names
  - Recall: internally all operations work with integer IDs
  - In final result, internal IDs have to be replaced by names
  - Seems trivial: the IDs are consecutive, so just use an array
  - Problem: for **Wikidata**, the total size of all names is **80 GB**

    You don't want to require a machine with 80 GB of RAM

    Also, reading 80 GB into RAM at each startup takes long
  - Solution: identify names which are rarely needed in result sets, and store these on disk … here are two examples

    wds:q42-b88670f8-456b-3ecb-cf3d-2bca2cf7371e

    <long name of obscure movie in obscure language>

Efficiency 3/3

- Figures for the complete Wikidata
  - Input size: 9.8 billion triples
  - On disk index files: 1.4 Terabyte
  - Startup time: ~ 2 minutes
  - RAM usage: ~ 20 Gigabytes
  - Average query time: ~ 1 second
  - Average query time of Wikidata query service: frequent timeouts after 20 seconds

we added some triples

More about query times in the next part

Part 0: Introduction

Part 1: Knowledge Base Basics

Part 2: Indexing and Query Processing

Part 3: Combination with Text Search

Part 4: SPARQL User Interfaces

SPARQL+Text Search 1/12

- Motivation
  - A lot of information is naturally structured

    We have seen a lot of examples in the talk so far

  - But even more information is naturally unstructured, typically in text written in natural language

    Because it's the natural form of communications for humans

  - Also: certain information is hard or unnatural to cast in structured form, for example:

    "Neil Armstrong was the first person to walk on the moon"

    There is no meaningful predicate here, which we can reuse for other knowledge … for example, this would be **weird**

    <Neil Armstrong> <First to walk on planet> <Moon>

SPARQL+Text Search 2/12

- Linking a knowledge base with text
  - Identify mentions of an entity from the KB in the text … this is called **named entity recognition**
  - Annotate that mention with the correct ID from the KB … this is called **named entity disambiguation** or **linking**

Q1615 Q23548
In a 2010 interview, Armstrong explained that NASA limited
Q1615 his moon walk to two hours because …

Q1615 = the Wikidata-ID for Neil Armstrong
Q23548 = the Wikidata-ID for NASA

SPARQL+Text Search 3/12

- Dimensions of some text corpora linked to a KB
  - **Wikipedia+Freebase**

    A dump of all articles from the English Wikipedia with entity links to Freebase (provided by Bast et al, SIGIR'14)

    2.3B word occurrences, **0.5B** entity links
  - **Clueweb+Freebase**

    A web-scale corpus with entity links to Freebase (provided by Gabrilovich et al: http://lemurproject.org/clueweb12/FACC1)

    32.3B word occurrences, **3.3B** entity links

SPARQL+Text Search 4/12

- What SPARQL+Text is **not**
  - Consider the following example triples

```
<Neil Armstrong>  <Profession>      <Astronaut>
<Neil Armstrong>  <Nationality>     <USA>
<Neil Armstrong>  <Books written>   "First on the moon"
```

  - SPARQL engines like Virtuoso support **text search in literals**

```
SELECT ?x ?y WHERE {
  ?x  <Profession>  <Astronaut>  .
  ?x  <Books written>  ?y  .
  ?t  bif:contains  "walk AND moon"
}
```

Astronauts who have written a book with "walk" and "moon" in the title

no matches with the KB triples above

The namespace prefix "bif" stands for "built-in function"

SPARQL+Text Search 5/12

- What SPARQL+Text **is**
  - QLever supports two special predicates ql:contains-entity and ql:contains-word with the following semantics:

```
SELECT ?x  WHERE {
  ?x  <profession>  <Astronaut>  .
  ?t  ql:contains-entity  ?x  .
  ?t  ql:contains-word  "walk moon"
}
```

Wikipedia+FreebaseEasy:
33 hits for Neil Armstrong

Clueweb+Freebase:
2485 hits for Neil Armstrong

This finds all astronauts, which **anywhere in the whole text corpus** co-occur with the words "walk" and "moon"

For a large text corpus many such co-occurrences are a good signal that the respective astronaut indeed walked on the moon

SPARQL+Text Search 6/12

This is a hard **NLP** problem and would be a lecture on its own

NLP = natural language processing

- Sentence decomposition
  - Decompose each sentence into **segments** (= subsets of words) that semantically "belong together"

    The usable parts of rhubarb, a plant native to Eastern Asia, are the medicinally used roots and the edible stalks, however its leaves are toxic.

    "rhubarb", "edible", and "leaves" do **not** belong together
  - The correct contexts are (need not be grammatical):

    rhubarb a plant native to Eastern Asia
    The usable parts of rhubarb are the medicinally used root
    The usable parts of rhubarb are the edible stalks
    however rhubarb leaves are toxic

SPARQL+Text Search 7/12

- SPARQL+Text vs. SPARQL with text search in literals
  - Wouldn't the query from two slides ago also work with a text search in literals like Virtuoso's **bif:contains** ?

```
SELECT ?x WHERE {
  ?x <Profession> <Astronaut> .
  ?x <Description> ?d .
  ?d bif:contains "walk AND moon"
}
```

This only works for astronauts, who **in the KB** are explicitly connected to a description which contains these words

For most queries of this kind, this is very unlikely

The typical description associated with an entity in a KB is a very short summary, like the first sentence from Wikipedia

SPARQL+Text Searc

Again: understand that "doping" is unlikely to be mentioned in the knowledge base for every entity in question …

… but is likely to be well-covered via text-entity co-occurrence in a large text corpus

- A more complex example

```
PREFIX fb: http://rdf.freebase.com/ns/
SELECT ?person ?profession ?drug TEXT(?text) WHERE {
  ?text ql:contains-word "doping" .
  ?text ql:contains-entity ?person_id .
  ?text ql:contains-entity ?drug_id .
  ?drug_id fb:type.object.type ?drug_type .
  ?drug_type fb:type.object.name "Drug"@en .
  ?drug_id fb:type.object.name ?drug .
  ?person_id fb:type.object.name ?person .
  ?person_id fb:people.person.profession ?profession_id .
  ?profession_id fb:type.object.name ?profession
 } ORDER BY DESC(SCORE(?text))
```

"People involved in doping, with their profession and the drug"

SPARQL+Text Search 8/12

- Additional features

```
SELECT ?x TEXT(?t) WHERE {
  ?x  <profession>  <Astronaut>  .
  ?t  ql:contains-entity  ?x  .
  ?t  ql:contains-word  "walk moon"
}
ORDER BY DESC(SCORE(?t))
TEXTLIMIT 1
LIMIT 1000
```

These features are **very useful** when using SPARQL+Text queries in practice

Specify the text snippet as part of the result

Specify the number of text snippets to return per match

Rank by (some function of) the number of co-occurrences

SPARQL+Text Search 9/12

This is similar to how **keyword search** can be implemented with a **database** and **SQL**

- Simulation by standard SPARQL
  - SPARQL+Text can be "simulated" with standard SPARQL:

    Create a new entity for each word and explicitly create two new relations <contains-entity> and <contains-word>

    These are huge: for Clueweb+Freebase, <contains-entity> has **3.3B** triples, and <contains-word> has **32.3B** triples

  - Then we could simply write:

```
SELECT ?x ?t WHERE {
  ?x <profession> <Astronaut> .
  ?t <contains-entity> ?x .
  ?t <contains-word> <word:walk> .
  ?t <contains-word> <word:moon>
}
```

We will see: for state-of-the-art SPARQL engines, this is **very slow**

SPARQL+Text Search 10/12

- Simulation by SPARQL with shallow text search
  - If keyword search in literals (bif:contains) is available, we can do away with the explicit <contains-word> relation

```
SELECT ?x WHERE {
  ?x <profession> <Astronaut> .
  ?t <contains-entity> ?x .
  ?t bif:contains "walk AND moon"
}
```

We will see: this is better, but still **slow**

SPARQL+Text Search 11/12

- Other SPARQL engines

  The following two are well known / widely used:

  Virtuoso with keyword search in literals (bif:contains)

  Widely used in practice and often comes out on top
  in SPARQL performance evaluations

  RDF-3X using explicit <contains-word/entity> relations

  One of the best research prototypes: supports (almost)
  full SPARQL and can beat Virtuoso on medium-sized data

SPARQL+Text Search 12/1

Qlever uses a
special word-entity
co-occurrence index

See the QLever paper for details

Results on FreebaseEasy+Wikipedia (**0.5B** triples, **2.3B** words)

Query Type	RDF-3X	Virtuoso	QLever
SPARQL only, simple	98 ms	337 ms	**74 ms**
SPARQL only, complex	3,349 ms	14,237 ms	**262 ms**
SPARQL+Text, simple	1,776 ms	941 ms	**78 ms**
SPARQL+Text, complex	5,876 ms	13,612 ms	**208 ms**
SPARQL+Text, real	1,063 ms	766 ms	**74 ms**
SPARQL+Text, text only	10,749 ms	15,037 ms	**191 ms**
SPARQL+Text, huge result	aborted	3,673,492 ms	**605 ms**
Index Size	138 GB	124 GB	**73 GB**
Index without Text	17 GB	9 GB	**49 GB**
Memory used	30 GB	45 GB	**7 GB**

Part 0: Introduction

Part 1: Knowledge Base Basics

Part 2: Indexing and Query Processing

Part 3: Combination with Text Search

Part 4: SPARQL User Interfaces

SPARQL Autocompletion 1/4

- Example
  - Assume we have entered the following SPARQL triples:

    ?x <is-a> <Person> .
    ?x   we are about to type a predicate

  - Goal: ranked list of suggestions of predicates that actually lead to results … e.g. <Gender>, but not <Founded_by>
  - This can actually be expressed with a SPARQL query

```
SELECT ?predicate ((COUNT(?predicate) AS ?count)) {
  ?x <is-a> <Person> .
  ?x ?predicate ?object
}
GROUP BY ?predicate
ORDER BY DESC(?count)
```

SPARQL Autocompletion 2/4

- Challenge
  - Look at the two triples in the query from the previous slide

    ?x <is-a> <Person> .
    ?x ?predicate ?object

  - This will compute **all** triples with a person as subject

    For Freebase, these are **165,850,440** triples

  - All we need is how often each predicate occurs in these

<is-a>	3,970,856 times
<Gender>	2,276,150 times
<Date of birth>	1,915,174 times
<Profession>	1,237,192 times
…	

Can we compute this more efficiently?

SPARQL Autocompletion 3/4

For Freebase:
1,246,827,727 pairs

- Let's abstract the problem a bit
  - Input: **all** pairs of entity ID and predicate ID

1	**1**	2	**3**	**3**	**3**	4	5	5	5	6	6	**7**	**7**	8	8	8
A	**F**	C	**A**	**B**	**F**	A	A	B	F	A	D	**A**	**F**	A	B	F

  - Query: a list of entity ids

    1 3 7

For our example:
all **3,970,856** IDs of persons

  - Result: predicates from query, sorted by frequencies

    #A = 3, #B = 1, #F = 3

  - Naive solution: **intersect** (join) input list with query list

    This takes too long, even with the most efficient list intersection algorithms … because the input is so large

SPARQL Autocompletion 4/4

This yields interactive completion times on Freebase

- Our solution
  - Look at the entity-labels list again:

    1 **1** 2 **3** **3** **3** 4 **5** **5** **5** 6 6 **7** **7** **8** **8** **8**
    A **F** C **A** **B** **F** A **A** **B** **F** A D **A** **F** **A** **B** **F**

  - Observation: same **pattern** for many entities
  - Idea: store ids of frequent patterns in a simple array

    1 2 **3** 4 **5** **6** 7 8
    a x **b** x **b** **x** a b   a = AF, b = ABF, x = rare

  - Now, given **input list**, first collect patterns and aggregate:

    1 3 5 6 → #a = 1, #b = 2 → #A = 3, #B = 2, #F = 3

    then intersect with remaining rare-patterns entity-labels list

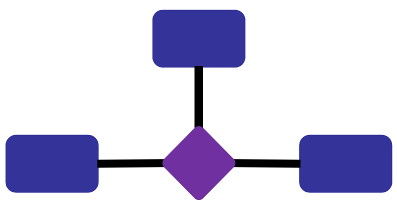
Natural-Language Queries 1/5

- The ultimate goal
  - Ask questions in **natural language**:

    Which character did Ellen DeGeneres play in Finding Nemo?
  - Or even more informally / telegraphically:

    who did ellen play in finding nemo?
  - **Goal:** automatically translate such a natural-language or keyword question into the corresponding SPARQL query

```
SELECT ?x WHERE {
  ?m  actor      Ellen DeGeneres .
  ?m  film       Finding Nemo .
  ?m  character  ?x
}
```



Natural-Language Queries 2/5

- Challenge 1: Linguistic variation
  - The same question can be asked in dozens of ways:

    which character did ellen degeneres play in finding nemo

    which character did ellen play in finding nemo

    who did ellen play in finding nemo

    ellen's role in finding nemo

    whose voice did ellen do in finding nemo

    role ellen nemo

    …

    This rules out simple pattern-based approaches

Natural-Language Queries 3/5

- Challenge 2: Ambiguous entity names
  - Ellen could mean

    Ellen DeGeneres

    Ellen Page

    Ellen Burstyn

    anyone called "Ellen"

    The Ellen Show

    The Ellen DeGeneres Show

    ...

    Over 100 different entities named "ellen" in Freebase

Natural-Language Queries 4/5

- Challenge 3: Ambiguous relation names
  - Like for entity names, but worse, because the relation can be implicit in the question, for example:

Question: who is the ceo of apple

Query:
```
SELECT ?x WHERE {
  ?m  job-title  "Managing Director" .
  ?m  company    "Apple Inc." .
  ?m  person     ?x .
}
```

None of the relation words "job title", "company", "person" appear in the question … nor synonyms of them

Natural-Language Queries 5/5

- Aqqu demo: http://aqqu.cs.uni-freiburg.de
  - Aqqu is a system that **learns** to translate natural language queries to SPARQL queries
  - The training data is only **question – answer pairs**

    The correct SPARQL query for a question is not needed, this makes it easy to generate lots of training data
  - Basic idea (very very very briefly):

    1. Generate a large number of candidate SPARQL queries (as possible interpretations of the question)
    2. Learn how to rank these candidates → pick the best one
  - Paper: More Accurate Question Answering on Freebase
    Hannah Bast and Elmar Haussmann, CIKM 2015

Literature

- Survey

Semantic Search on Text and Knowledge Bases

Hannah Bast, Björn Buchhold, Elmar Haußmann

Foundations and Trends in Information Retrieval 2016

  - It's about **everything** related to semantic search
  - It's big, but easy to read, and the various chapters can be read and understood stand-alone

    For example, there is a chapter about **NLP basics**

    Or one about **Search in Knowledge Bases**

  - If you interested in publications on QLever or Broccoli, you easily find them via Google ...

NLP = Natural Language Processing

Demos and Code

- Code
  - The demos I gave shown are freely available

    http://broccoli.informatik.uni-freiburg.de google: **broccoli search**

    http://qlever.informatik.uni-freiburg.de google: **qlever search**
  - The code is freely available on **GitHub**

    https://github.com/ad-freiburg/qlever
  - Easy to install + comes with several datasets to play around with: small, medium-sized, and large

    You can set up your own instance for one of these datasets in a few minutes … or build an instance from your own data

    Use it and let us know your **comments / suggestions**