SpiderStore: A Native Main Memory Approach for Graph Storage

Robert Binna, Wolfgang Gassler, Eva Zangerle, Dominic Pacher, Günther Specht
Databases and Information Systems, Institute of Computer Science
University of Innsbruck, Austria
{firstname.lastname}@uibk.ac.at

ABSTRACT

The ever increasing amount of linked open data results in a demand for high performance graph databases. In this paper we therefore introduce a memory layout which is tailored to the storage of large RDF data sets in main memory. We present the memory layout *SpiderStore*. This layout features a node centric design which is in contrast to the prevailing systems using triple focused approaches. The benefit of this design is a native mapping between the nodes of a graph onto memory locations connected to each other. Based on this native mapping an addressing schema which facilitates relative addressing together with a snapshot mechanism is presented. Finally a performance evaluation, which demonstrates the capabilities, of the *SpiderStore* memory layout is performed using an RDF-data set consisting of about 190 mio triples.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Performance, Algorithms, Design, Experimentation

Keywords

RDF, Main Memory, Database, RDF Store, Triple Store, SPARQL, Addressing Scheme

1. INTRODUCTION

Due to the increasing significance of linked open data and publicly available SPARQL-endpoints, the need for high performance graph databases has increased. To meet those requirements several approaches for storing and retrieving large RDF (Resource Description Framework) graphs have been developed. Despite the highly connected nature of these graphs, the main approaches proposed in this context are facilitating technologies originating from relational databases. Even though these represent major and robust technologies, they were not tailored for the scenario of storing graph based structures. At the same time the ever increasing capacities of main memory and the increasing numbers of cores have lead to an architectural shift in the development of databases and information systems by moving from hard disk to main memory as the primary storage device. Whereas these architectural changes lead to enormous performance improvements, when implementing graph-operations like graph traversals they still have to be implemented through costly self join operations. Despite the possibility of supporting these operations with appropriate index structures they still take $\mathcal{O}(log(n))$ where n denotes the number of index entries. Therefore we present the SpiderStore storage concept as an in-memory storage approach, which allow to process edges in $\mathcal{O}(1)$. In contrast to previous work [3] the storage layout and space estimations are captured in more detail. In addition, a new relative addressing scheme is introduced. The successive sections are structured as follows. Chapter 2 deals with the memory layout and space estimations. Chapter 3 explains the relative addressing scheme used for faster restarts and for snapshot generation. In chapter 4 we present an evaluation of the presented technology using the YAGO2 [8] data set. Chapter 5 discusses the related work in the field of RDF-databases. Finally Chapter 6 draws a conclusion and makes forecasts for possible future work.

23rd GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 31.05.2011 - 03.06.2011, Obergurgl, Austria.
Copyright is held by the author/owner(s).

2. MEMORY LAYOUT

This section represents a detailed description over the *SpiderStore* memory layout. The aim of this memory layout is to provide an in-memory storage of graphs, where the basic operation of navigating between two vertices can be done in $\mathcal{O}(1)$. Therefore, the node is the core component of the layout. This is in contrast to the concept favored by most triple stores where the the triple represent the atomic building block. To realize this concept of a node centric layout, two factors have to be fulfilled. The first is that all edges belonging to a node need to be stored in a single place, which allows to navigate back and forth along all edges. The second factor is that there need to be a direct connection between those nodes that can be resolved within a single operation. These requirements can be fulfilled by an in-memory storage layout, which is designed as follows.
Each node has to be located at a unique location within

memory. At each of these locations the information about ingoing and outgoing edges is stored and grouped by the respective edge label. The information about edges itself is not stored directly but implicitly through pointers. Therefore, the traditional pointer structures, which themselves can be seen as graphs connecting different memory locations are used to represent arbitrary graphs.

Each node contains a set of ingoing and a set of outgoing edges (pointers). These pointers are grouped by the predicates labeling the corresponding edges. Grouping is done by the address of the predicates. This address is equal to the address of the property within the predicate index, which stores a list of all subjects occuring together with a specific property.
Beside the raw pointer data, implicit statistical information is stored. This is the case for the number of nodes, the number of predicates and the number of strings. Furthermore, for each subject the number of predicates and for each subject/predicate combination the number of objects is stored. The same information is stored the other way around (the number of predicates for an object and the number of predicate/subject combinations for an object).

To illustrate this layout, Figure 1 visualizes the memory layout of a short example. The main node emphasized in this example represents the category of a `wordnet_scientist`. It can be seen from this example that two different types of edges exist: ingoing and outgoing edges. Both types themselves group their destination pointers by the predicate node. In Figue 1, an outgoing edge with the property ⟨`hasLabel`⟩ and an incoming edge with the property ⟨`type`⟩ is featured. To simplify matters, URIs are abbreviated and marked with angle bracket while literals are put in quotes. As it can be seen in the example, all nodes independent of their type (URI, literals, ...) share a uniform node layout. For example the node `"scientist"` has the same structure as the node ⟨`wordner_scientist`⟩. The facts described in this example are that ⟨`Einstein`⟩ is of ⟨`type`⟩ ⟨`wordner_scientist`⟩ and that this category has a label with the name `"scientist"`. The triple notation of the example in Figure 1 is shown in the listing below:

```
...
⟨Einstein⟩ ⟨type⟩ ⟨wordner_scientist⟩
⟨wordner_scientist⟩ ⟨hasLabel⟩ "scientist"
...
```

2.1 Space Estimations

Given that SpiderStore is a main memory based graph store, space consumption becomes a crucial factor. Therefore we introduce and discuss a formula that can be used to calculate the expected amount of memory needed for a specific set of data. To describe a specific data set we use the following variables. The variable `#nodes` represents the total number of distinct nodes within a graph. A node can either be a URL or a character string and can be be used as subject, predicate or object. The second variable used for calculating the expected space consumption is the total number of triples or facts `#notriples`. The space consumption is then calculated as follows:

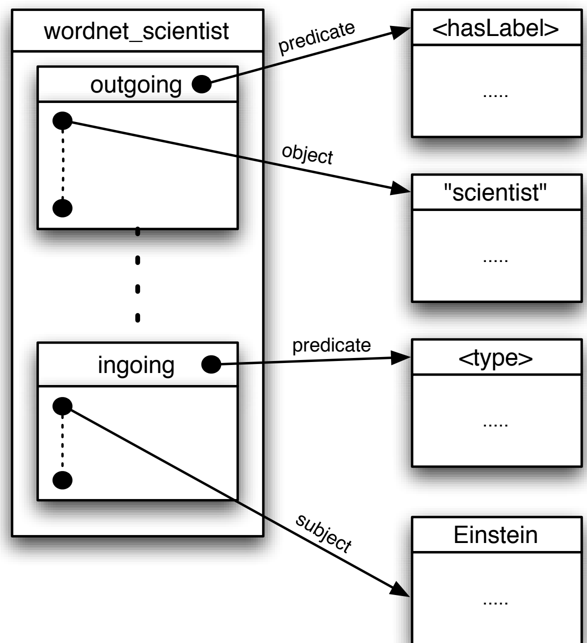

Figure 1: Memory Layout - Example

$$m = (\#nodes * (5 + degree * 2) + \#edges * 3) \\ * sizeof(pointer) + sizeof(dictionary)$$

This formula consists of three parts, i.e. the size of the data dictionary, the fraction influenced by the number of nodes and the fraction influenced by the number of edges. The part of this formula that depends on the number of nodes can be interpreted as follows. For each node a data structure is stored consisting of the number of incoming edges and a link to the table containing the incoming edges as well as the number of outgoing edges and a link to the table containing the outgoing edges. Furthermore a pointer to the corresponding entry within the dictionary table is stored within each node. The term $degree * 2$ can be explained by the pointers which group the edges by their corresponding predicates. For each predicate there exist a pointer to the edges itself and a counter, which is used for statistics. Because all edges are bidirectional, the estimation is quite accurate even though it does not take the number of distinct subjects or objects into account.
The part of the formula, which depends on the number of edges, can be derived of the following facts. For each edge the destination is stored on both ends. Furthermore, an additional entry is stored within the predicate index which allows to start the query execution not only at subject or object nodes but as well at predicate nodes.
As an example the YAGO2 [8] data set used throughout the evaluation consists of 194,35 million triples and of 28,74 million distinct nodes. The dictionary size in the case of the YAGO2 data set is roughly about 1.1 gigabytes. In this example 1.2 would be an appropriate value for the variable *degree*. This number can be derived by counting the number

of all distinct subject-predicate combinations and dividing it by the total number of nodes. By apply these values to the formula above a space consumption of about 7,4 gigabytes might be expected. In comparison to the real footprint, which is about 7.2 gigabytes this value is quite accurate. Hence knowing the basic facts about a data set allows to establish an adequate estimation about the expected memory footprint.

3. RELATIVE ADDRESSES

Using absolute pointer addresses for the location of nodes enables the navigation between two nodes in $\mathcal{O}(1)$ as it has been explained in the section before. The drawback of this concept is that storing a persistent state of a whole graph database can become quite complex. The reason for this is that the database can only be dumped either by serializing the whole graph and later deserializing the whole graph or by dumping the whole memory snapshot and swizzling the pointers to their correct representation at load time.
Another alternative to these approaches is to arrange the whole database content based on an offset address. Hence all memory locations have to be converted into relative addresses based on this offset. Resolving such relative pointers yields in an overhead, which is negligible compared to the total amount of time needed to fetch an arbitrary junk of memory. The time for fetching an arbitrary chunk of memory can vary between some cpu cycles, when accessing memory already located within L1-cache, up to some hundreds cpu cycles, when accessing a random chunk of memory which is not available in the cache hierarchy yet.
In the context of SpiderStore we decided to use relative addressing. This has two advantages. The first advantage is that the general architecture of SpiderStore is still applicable and the overhead introduced through relative addressing is insignificant in the SpiderStore concept as it has been explained before. The main advantage of this addressing scheme is that database restarts can be executed within a few milliseconds. This is possible by facilitating the Unix memory mapping techniques which does not dismiss the mapped pages unless another executable allocates large amounts of memory. Furthermore this concept allows to facilitate the copy on write approaches used by the HyPer project [10]. This approach benefits from the operating system's memory management, which would allow different processes to share large memory segments, while preserving an isolated view on the data.

Due to the lack of a customized memory allocator, the SpiderStore snapshot mechanism is currently implemented as a read only approach. Each snapshot is split up into five parts. One part is responsible for the storage of the node data structures, another for the indexes between node identifiers and node addresses. One more file stores the node identifiers. The other files are responsible for the storage and indexing of predicate nodes and for the storage of the edge information. The separation into several files prevents memory fragmentation and leads to shorter "addresses" for nodes, predicates and strings. For example all entries within the node, predicate or index files have uniform sizes and can therefore be seen as simple array structures.

4. EVALUATION

As a platform for the evaluation a server equipped with two Intel Xeon L5520 Quad Core CPUs, 2.27 GHz, Linux kernel 2.6.18, CentOS, 64-bit architecture and 96 GB main memory was used.

4.1 DataSet

For the evaluation we used the YAGO2 [8] data set. The YAGO2 data set is the successor of the YAGO [16] data set and represents a large semantic knowledge base which is generated on the basis of Wikipedia, WordNet and GeoNames. The data set consist of of 194,350,853 triples (98 predicates, 28,744,214 unique subjects and objects). The queries executed on this data set are derived from the queries on the YAGO data set used in the benchmark presenting the RDF-3X approach [12].

4.2 Evaluated Systems

For the evaluation of *SpiderStore* we used Virtuoso [6] in version 6.1.3, RDF-3X [12] in version 0.3.5 and Jena TDB [19] in version 0.8.9. The decision for these systems was taken to the best of our knowledge. Even though *SpiderStore* is the only main memory system tested, the decision for choosing the other systems is accurate. The reason for this is that those systems are assumed to be the currently fastest systems available and that the benchmark results are measured with warm caches on a system where the whole database would be able to fit into main memory. All systems were granted a maximum of 40 GB to ensure that sufficient space is available.

4.3 Evaluation Results

The test results are separated into two parts. The first part compares the bulk import times of the different systems. The bulk import time is specified as the time needed to load the data, to create the indexes and to ensure that a persistent state of the database is written to disk. A summary of the load times can be seen in Table 1. As can be seen, *SpiderStore* is significantly faster than any of the other systems. The reason for this is that *SpiderStore*, due to its implicit statistics, does not need to explicitly create statistics or indexes.

System	Load Time
SpiderStore	1:09:18
Jena	1:36:35
RDF-3X	1:21:12
Virtuoso	3:32:16

Table 1: Import Times (in hours, minutes and seconds)

The second part of the evaluation compares the query execution for each query on the YAGO2 data set. Queries with an execution time over 15 minutes without producing any output are marked with "DNF". For the calculation of the geometric mean, a query runtime of 15 minutes is assumed for each cell which is marked with "DNF". Due to large result sets for query C-1 this limit is set to 60 minutes. The results of this evaluation are shown in Table 2.

Query	A1	A2	B1	B2	B3	C1	C2	C3	geom. mean
SpiderStore	**0.0016**	0.2084	0.3688	**0.0119**	0.5143	DNF	16,5464	319.0703	0.4521
Jena	55.203	142.155	DNF	DNF	DNF	DNF	DNF	DNF	578.3126
RDF-3X	0.0496	**0.0524**	**0.0471**	0.0936	**0.0482**	**657.2100**	**0.2056**	**2.4741**	0.3414
Virtuoso	0.766	0.127	0.71	0.46	3.223	2197,672	2.401	36.474	3,4420
#results	3	5	274	128	166	6,876,673	5313	35811	

Table 2: Query Runtimes on the YAGO2 data set (in seconds).

Considering the average execution time, RDF-3X performs better than all other stores tested. While *SpiderStore* in the average case is the second fastest system, two queries exist where *SpiderStore* outperforms the other stores. In the case of those queries, the coarse heuristics were able to generate a highly efficient execution plan. On the other side considering the queries C1-C3 *SpiderStore* has significant problems to arrange and optimize the execution order to obtain better results. Regarding query C1 even though intermediate results where present the total execution time did exceed the time limit of one hour. The reason for the performance shortcomings in the case of those queries is that the selectivity estimations based on the triple selectivity used for the generation of the execution plan can in some cases produce misleading execution orders. This is due to the fact that the cardinality of properties is not considered in the current algorithm. Regarding the other evaluated systems Jena TDB seem to have severe problems when executing queries on such large knowledge bases, because only two queries were able to determine the results within the given time frame. Whereas Virtuoso can be considered as the third fastest system, which has a stable performance without any negative outliers.

5. RELATED WORK

Several approaches exist for storing graph based data in databases. In the particular case of this paper we focus on RDF-stores because *SpiderStore*, the developed system, can be considered as part of this category. Hence we give a short overview about the different approaches available for storing RDF data.

For storing RDF-data, two approaches are prevailing. On the one hand the approach of mapping the RDF-data onto relational schema exists while on the other hand the approach of native RDF-stores exist.

The mapping of RDF-data onto relational databases is done either by facilitating a large triple table, where the columns correspond to the RDF atoms subject, predicate and object or by clustering the triples according to their predicate into several tables. The latter approach is called property tables. Both approaches are less than perfect because both suffer from severe performance drawbacks imposed by the architecture of relational databases. For example in the property tables approach the number of tables is equal to the number of properties in the worst case. Several approaches which extend these two main approaches when mapping RDF-data onto relational database have been developed and are benchmarked in [17]. Beside mappings to traditional relational database systems, mappings which make use of column oriented databases exist [1, 15]. In the case of native stores two major categories exist: (i) systems which have a main memory architecture and (ii) systems which use secondary memory storage as their primary storage layer. Systems falling into the first category are for example Brahms [9], Grin [18], Swift-OWLIM [11] or BitMat [2] as well as our system *SpiderStore* [3]. While Brahms is highly optimized for association finding and Grin for answering long path queries the goal of *SpiderStore* is to provide efficient query processing for arbitrary SPARQL queries. Swift-OWLIM is also a general purpose RDF-store, which has a strong emphasis on OWL-reasoning. Bitmat on the other hand represents a lightweight index structure which uses bitmap indexes to store space efficient projections of the three dimensional triple space. In contrast to the main memory based systems YARS2 [7], RDF-3X [12] can be considered as native systems of type (ii). Both systems make heavy use of index structures. While YARS facilitates six index structures for subject, predicate, object and the context, RDF-3X [12] generates index structures for all possible combinations and orderings of subject, predicate and object. Beside these huge number of index structures, RDF-3x makes heavy use of statistical information and has a highly sophisticated query execution engine, which is described in [13]. While such an enormous effort results in a good query performance, the management of these specific data structures can become quite complex. Neumann *et al.* therefore describe in [14] how a query centric RDF-engine can be extended to provide full-fledged support for updates, versioning and transactions. Beside these systems which can be clearly dedicated to either the category of memory-native, secondary memory-native or relational based systems several semantic web frameworks exist, which provide storage engines fitting in several or all of these categories. Examples of such frameworks are Sesame [5], Jena [19] and Virtuoso [6]. For all the systems described in this section several benchmarks exist [4, 12], which extensively compare those systems.

6. CONCLUSION AND FUTURE WORK

In this paper we presented the *SpiderStore* memory layout, which is able to store arbitrary graphs. The node centric layout has been discussed in detail and a formula for the estimation of the space consumption was described. An enhancement to the basic layout introducing a relative addressing schema was presented. Finally our experiments showed that a node centric layout is able to perform arbitrary SPARQL-queries on knowledge bases of up to 190 mio nodes with a performance comparable to highly sophisticated RDF-stores. This promises further performance improvements because the query optimisation approach, which has been developed in [3] is rather simple. Therefore future

work on *SpiderStore* will emphasize on the query execution engine to achieve excellent performance results on a scale of up to a billion triple.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 41–50, New York, NY, USA, 2010. ACM.

[3] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. Spiderstore: Exploiting main memory for efficient rdf graph representation and fast querying. *In Proceedings of the 1st International Workshop on Semantic Data Management (Sem-Data) at the 36th International Conference on Very Large Databases, Singapore*, Jan 2010.

[4] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 5(1), 2009.

[5] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. *The Semantic WebâĂŤISWC 2002*, pages 54–68, 2002.

[6] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24.

[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. *The Semantic Web*, pages 211–224.

[8] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: a spatially and temporally enhanced knowledge base from wikipedia. Research Report MPI-I-2010-5-007, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, November 2010.

[9] M. Janik and K. Kochut. Brahms: A workbench RDF store and high performance memory system for semantic association discovery. *The Semantic Web–ISWC 2005*, pages 431–445.

[10] A. Kemper and T. Neumann. Hyper: Hybrid OLTP & OLAP high performance database system. Technical Report TU-I1010, TU Munich, Institute of Computer Science, Germany, May 2010.

[11] A. Kiryakov, D. Ognyanov, and D. Manov. Owlim–a pragmatic semantic repository for owl. *Web Information Systems Engineering–WISE 2005 Workshops*, pages 182–192, Jan 2005.

[12] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[13] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 627–640, New York, NY, USA, 2009. ACM.

[14] T. Neumann and G. Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *Proceedings of the VLDB Endowment*, 3(1-2), Jan 2010.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.

[16] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.

[17] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. *The Semantic Web–ISWC 2005*, pages 685–701, 2005.

[18] O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A graph based RDF index. In *Proceedings of the National Conference on Articial Intelligence*, volume 22, page 1465. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[19] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, et al. Efficient RDF storage and retrieval in Jena2. In *Proceedings of SWDB*, volume 3, pages 7–8. Citeseer, 2003.

APPENDIX

A. QUERIES

A.1 YAGO data set

prefix rdfs:⟨http://www.w3.org/2000/01/rdf-schema#⟩
prefix xsd:⟨http://www.w3.org/2001/XMLSchema#⟩
prefix owl:⟨http://www.w3.org/2002/07/owl#⟩
prefix rdf:⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩
prefix yago:⟨http://www.mpii.de/yago/resource/⟩

A1: SELECT ?GivenName ?FamilyName WHERE { ?p yago:hasGivenName ?GivenName. ?p yago:hasFamilyName ?FamilyName. ?p rdf:type ?scientist. ?scientist rdfs:label "scientist". ?p yago:wasBornIn ?city. ?city yago:isLocatedIn ?switzerland. ?switzerland yago:hasPreferredName "Switzerland". ?p yago:hasAcademicAdvisor ?a. ?a yago:wasBornIn ?city2. ?city2 yago:isLocatedIn ?germany. ?germany yago:hasPreferredName "Germany". }

A2: SELECT ?name WHERE { ?a yago:hasPreferredName ?name. ?a rdf:type ?actor. 1 ?actor rdfs:label "actor". ?a yago:actedIn ?m1. ?m1 rdf:type ?movie. ?movie rdfs:label "movie". ?m1 yago:hasWikipediaCategory "German films". ?a yago:directed ?m2. ?m2 rdf:type ?movie2. ?movie2 rdfs:label "movie". ?m2 yago:hasWikipediaCategory "Canadian films". }

B1: SELECT ?name1 ?name2 WHERE { ?a1 yago:hasPreferredName ?name1. ?a2 yago:hasPreferredName ?name2. ?a1 rdf:type yago:wikicategory_English_actors. ?a2 rdf:type yago:wikicategory_English_actors. ?a1 yago:actedIn ?movie. ?a2 yago:actedIn ?movie. FILTER (?a1 != ?a2) }

B2: SELECT ?name1 ?name2 WHERE { ?p1 yago:hasPreferredName ?name1. ?p2 yago:hasPreferredName ?name2. ?p1 yago:isMarriedTo ?p2. ?p1 yago:wasBornIn ?city. ?p2 yago:wasBornIn ?city. }

B3: SELECT distinct ?name1 ?name2 WHERE { ?p1 yago:hasFamilyName ?name1. ?p2 yago:hasFamilyName ?name2. ?p1 rdf:type ?scientist1. ?p2 rdf:type ?scientist2. ?scientist1 rdfs:label "scientist". ?scientist2 rdfs:label "scientist". ?p1 yago:hasWonPrize ?award. ?p2 yago:hasWonPrize ?award. ?p1 yago:wasBornIn ?city. ?p2 yago:wasBornIn ?city. FILTER (?p1 != ?p2) }

C1: SELECT DISTINCT ?name1 ?name2 WHERE { ?p1 yago:hasFamilyName ?name1. ?p2 yago:hasFamilyName ?name2. ?p1 rdf:type ?scientist. ?p2 rdf:type ?scientist. ?scientist rdfs:label "scientist". ?p1 ?c ?city. ?p2 ?c2 ?city. ?city rdf:type ?cityType. ?cityType rdfs:label "city". }

C2: SELECT DISTINCT ?name WHERE { ?p yago:hasPreferredName ?name. ?p ?any1 ?c1. ?p ?any2 ?c2. ?c1 rdf:type ?city . ?c2 rdf:type ?city2. ?city2 rdfs:label "city". ?city rdfs:label "city". ?c1 yago:isCalled "London". ?c2 yago:isCalled "Paris". }

C3: SELECT ?p1 ?predicate ?p2 WHERE { ?p1 ?anypredicate1 ?city1. ?city1 yago:isCalled "Paris". ?p1 ?predicate ?p2. ?p2 ?anypredicate2 ?city2. ?city2 yago:isCalled "Hong Kong". }