Database Report - NELL

By Juncheng Liu, Lihang Liu, Zhaowei Tan

1. Database

1.1 Description

NELL (Never-Ending Language Learner) is a computer system that learns over time to read the web. So far it has accumulated over 50 million candidate beliefs by reading the web, and it is considering these at different levels of confidence. NELL has high confidence in 2,598,241 of these beliefs.

Nominally, each belief is an (Entity, Relation, Value) triple. From the definition of entity, relation and value and the data instances, we can further separate all the triples into three types: (Entity, Relation, Value), (Entity, Relation, Entity), (Entity, 'generalizations', Category). Notice that the Value of the first type is described as the value belonging to none concept or categories, different from the entity in the second type. We can separate them easily through the definition of the entity and the value.

Besides, the first two types are the relation instances while the third one is the category instances. It is easy to separate category instances from relation instances because the Relation field will always be "generalizations" for a category, and never for a relation. Eventually we should also notice that we should store the probability information in the (Entity, 'generalizations', Category) schema for the required query.

Concepts have names like "concept:coach:peyton_manning". To give a clear picture of three tuple types, I take

("concept:city:faaite", "concept:haswikipediaurl", "http://en.wikipedia.org/wiki/Faaite")
("concept:crustacean:bed", "concept:animaleatvegetable", concept:vegetable:leaves)
("concept:everypromotedthing:dallas_st", 'generalizations', concept:everypromotedthing)

as examples to describe the tuples respectively. It is very clear where the difference is.

As a learner system, the NELL builds a relationship between the concepts and the literals, the text referring to concepts. The same literal may have different concepts and the same concept, as a category, may have different literal strings belonging to it. We should notice that the data in the NELL may be misleading as there is no guarantee that the triples in the NELL show the real relationship. Therefore, it is essential to always look at the set of literal strings that refer to a concept, and to look at the set of categories to which a concept belongs in order to determine its true category membership. Simply stripping off the "concept:" prefix and category name will lead to incomplete and erroneous information.

1.2 Database Structure

As mentioned above, there are two parts play the key role in building the NELL database for the sake of query acceleration. Frist, we create the schema for all the (Entity, Relation, Value) relationships into three aforementioned types, with which we can clearly see the structure of different types. The acceleration is based on database separation for the query only needs to cover one part of the whole database. For instance, the required query "Given an entity, return all entities that are co-occurred with this entity in one triple" only needs to cover the (Entity, Relation, Entity) type and the "Given an entity, return all categories it belongs to" only needs to cover the (Entity, 'generalizations', Category). What's more, the (Entity, 'generalizations', Category) type will save the database space as there is no need to store the relationship column.

Except for these three schemas, we also need to create the schema to build the relationship in order to specific the value of each entity. For the same entity, the value of it may be different. Take "Apple" as an example, "Apple", "apple", and "apples" all can refer to the fruit. So we should use the same value to label the different types of the same entity. Thus we create the schema (Entity, Literal_String, BestLiteralString) to improve the query for the name.

So we finally create four mean schemas for the whole query, three for relations and one for literal strings.

1.3 Database Details

1. Table `LiteralString`
   a) Size: 312.8MB
   b) Lines: 2,303,750
2. Table `Relation_A`
   a) Size: 15.5MB
   b) Lines: 136,687
3. Table `Relation_B`
   a) Size: 42.6MB
   b) Lines: 310,827
4. Table `Relation_B`
   a) Size: 202.7MB
   b) Lines: 2,019,928
5. Table `Relation_A_2Step`
   a) Size: 202.7MB
   b) Lines: 2,019,928

2. Index Design & Optimization

In the next part, we will talk about how to accelerate the query by indexing the schemas. For the first four queries we use the index to accelerate. And we also execute another two complex queries by designing our ingenious methods.

2.1 Required Queries

2.1.1 Basic Description

There four required queries in the lab project and every query is related to only one schema.

1) The first query "Given a name, return all the entities that match the name" is related to the Literal string schema.

2) The second one "Given a category, return the top 50 typical entities belongs to this category" is related to (Entity, ('generalizations',) Category, Possibility).

3) The third one "Given an entity, return all categories it belongs to" is also related to (Entity, ('generalizations',) Category, Possibility) as it is a category instances query.

4) And the last query "Given an entity, return all entities that are co-occurred with this entity in one triple" is related to (Entity, Relation, Entity) triple.

2.1.2 Optimization

At first, we want to accelerate the query by putting them in specific order. And it is also time-saving to create the index among related entities. But after we inserted all the data into the database and built the index thorough the MySQL. By creating the B+ tree index, we find that the query speed is fast enough to return all the results. So we decided not to use more time to re-order the schemas.

2.2 Extended Queries

2.2.1 Basic Description

We design two extended queries to see the related entities. In this part, we consider every tuple as an edge of the graph and the direction of the edge is from entity to the value. We attempt to connect adjacent edges through the entities and thus dig out the deeper relationship.

1) In the 5-th query we tried to find the related entities given a specific name. It is a combination of query 1 and query 4. We firstly find the entity set for a specific name, and then find the co-occurred entities.

2) In the 6-th query, we take every tuple as an edge and we want to know which entities we can reach after we join all the adjacent edges. In this query, we will figure out all the entities that can

be reached after passing four edges. Both these two queries are composite query as they need to use multiple schemas.

2.2.2 Optimization

1) For the query 5, the improvement is also based on the index building, which is similar to the methods shown in required queries.
2) For the query 6, there are two solutions for it. We are aware that it is impossible for us to create a schema including all the information about the entities that can be reached after passing four edges from the initial entity. Because it will cost the fourth power of the initial space to store the whole information. So we tried to create the schema that hat can be reached after passing two edges from the initial entity and we can get the four times one by executing BFS for this schema. Besides, we can also use the BFS to search for all the possible entities that are related to the initial entity, and then repeat it three times to find all relevant entities. We compare these two methods and the result will be shown in the experiment result.

3. Experiment Result

In the experiments of the first four queries, we tried the query 20 times for each query and calculate the average execution time to represent the efficiency of the query process. In the experiments, we only record the time used to execute queries, no considering of the time used to generate SQL statement. We will choose the efficient query statement to execute the query.

3.1 Required Queries

The result from the Table 1 clearly show the improvement while applying the index. When applying the index, the average execution time is almost zero compared with the time before optimization.

	Before Optimization(s)	After Optimization(s)
Query 1	0.833991659	0.000794554
Query 2	0.741230786	0.030073881
Query 3	0.602750409	0.000775743
Query 4	0.063268626	0.000977874

Table 1: Average Execution Time for Different Queries

3.2 Extended Queries

For the query 5, the result is similar to the former queries.

	Before Optimization(s)	After Optimization(s)

Query 5	1.46959209442	0.075458049774

Table 2: Average Execution Time for Query 5

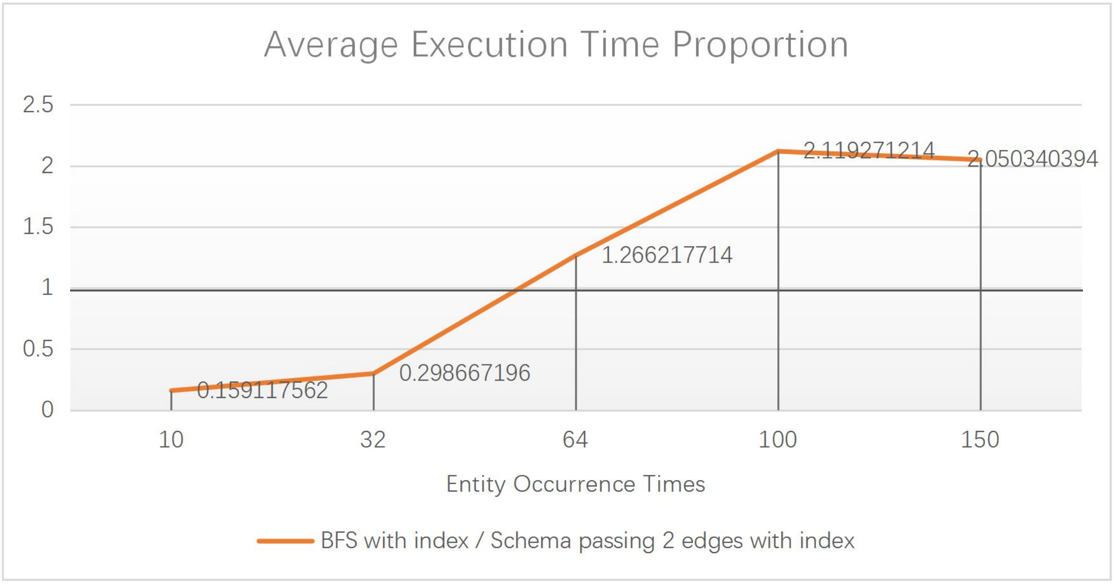
For the query 6, we have three methods to implement the query. We firstly tried the BFS without applying index. Compared with applying the index, it is very time consuming and cost more than one minute. For the BFS with index, it cost even much less time than the two times schema. The result can be clearly seen from the Table 3.

	Average Execution Time
BFS without index	>1 min
BFS with index	See Graph 1
Schema passing 2 edges with index	See Graph 1

Table 3: Average Execution Time of the sixth query for Different Methods

For this result, we postulate the reason that the two times schema occupies too much space so the query cost too much time for the query. From the database we find that the initial schema only covers 26.5 MB while the two times schema covers 6GB. So the query covers much more space and thus cost more time.

But the schema passing two edges seems to have a better performance when considering the number of the adjacent edges of the initial entity. When the initial entity has only a little adjacent edges, it will cause a long time to locate the required entity and thus the index plays a more important role in improving the efficiency. However, when the initial entity has more adjacent entities, the schema passing two edges will be more efficient as it will lower the MySQL query sessions. While the BFS will query four times for the initial schema, the new schema only needs to query twice. The result is clearly shown in the Graph 1. The new schema plays a more important role when the entity occurrence times increase.



Graph 1: Average Execution Time Proportion

4. User Interface

We choose to perform the human-computer interaction via a website, for a website is easy to construct, and we can put it in the server and let people browse it from anywhere. However, to build a good website and present the result intuitively is not an easy task. You can find our website at http://acemap.sjtu.edu.cn:9999/nell/index.php. In this section, we will describe the techniques we utilize in the website, as well as presenting some results.

4.1 Required Techniques and Libraries

Prompt to interact with MySQL, we choose PHP as the backend language, using the characteristic of its good compatibility with the database. In the frontend, with no question, we take the advantage of the languages including HTML, JavaScript and CSS.

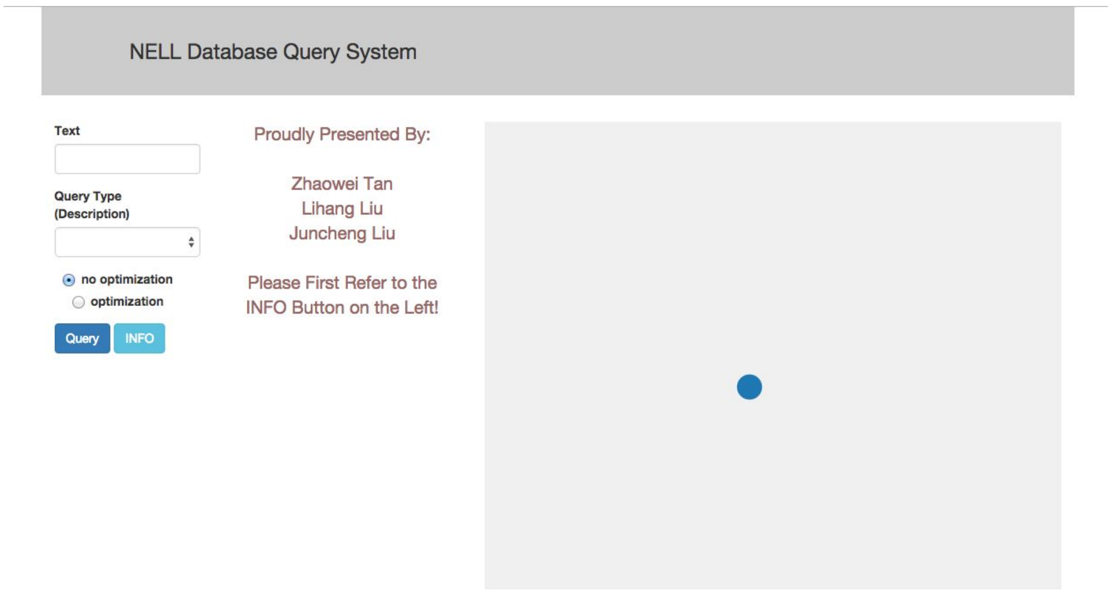
Here we should thank two famous JS and CSS libraries that contribute much to our interface.

The first is bootstrap project(http://www.bootcss.com/). We use the many components provided by bootstrap, for example, the result table css, the modal frame css & js, the button css, the "pin" effect js, etc.

The second package is d3js(http://d3js.org/). We add a fancy visualization of the results to our interface thanks to the amazing functions provided by d3js.

4.2 The Final Effects

a) Homepage
This is the homepage of our interface. The names of team member are listed in the middle of the page, as you can see. After clicking the "INFO" button, you can see the details of every query. Now you can type in the query keyword and choose the type and whether to use the optimization or not. Click the "Query" and then is the result.



NELL Database Query System

Text

Query Type (Description)

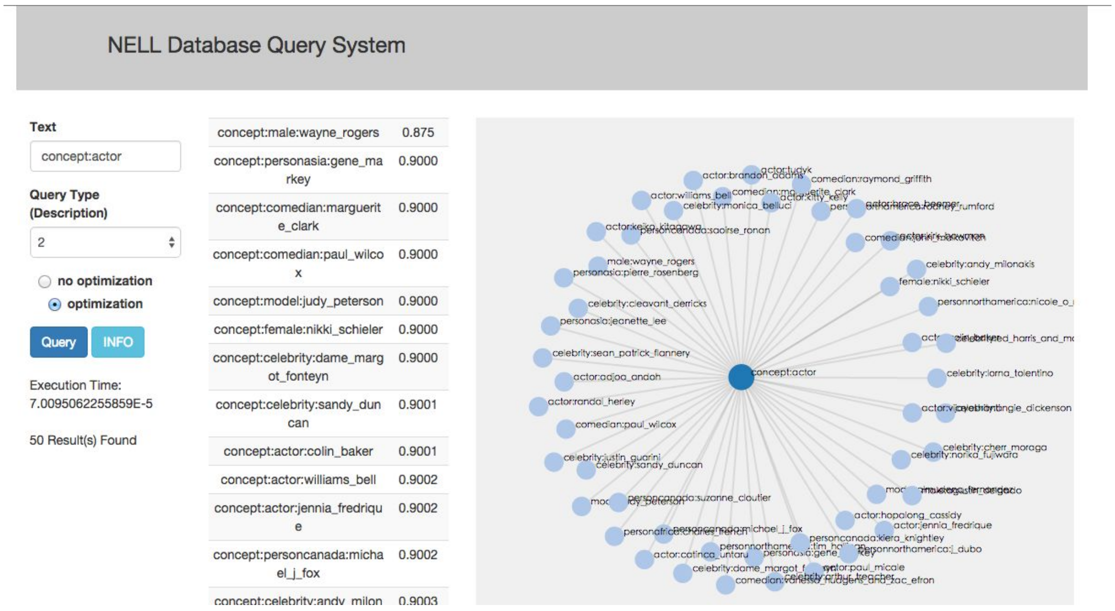
[✓ no optimization] optimization

Proudly Presented By:

Zhaowei Tan
Lihang Liu
Juncheng Liu

Please First Refer to the INFO Button on the Left!

b) This is the result page for basic queries (namely 1-4). The time and the number of the results are in the left part of the interface. While in the middle section, you can see the results presented in a table manner. On the right graph, this relationship is presented through a graph.



NELL Database Query System

Text: concept:actor

Query Type (Description): 2

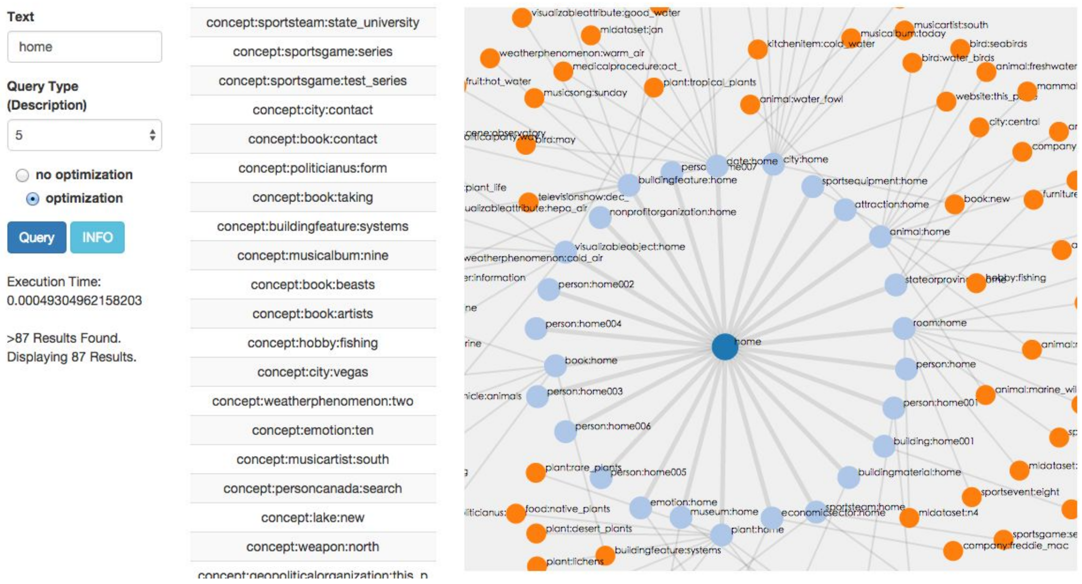
no optimization [✓ optimization]

Execution Time:
7.0095062255859E-5

50 Result(s) Found

concept:male:wayne_rogers	0.875
concept:personasia:gene_markey	0.9000
concept:comedian:marguerite_clark	0.9000
concept:comedian:paul_wilcox	0.9000
concept:model:judy_peterson	0.9000
concept:female:nikki_schieler	0.9000
concept:celebrity:dame_margot_fonteyn	0.9000
concept:celebrity:sandy_duncan	0.9001
concept:actor:colin_baker	0.9001
concept:actor:williams_bell	0.9002
concept:actor:jennia_fredrique	0.9002
concept:personcanada:michael_j_fox	0.9002
concept:celebrity:andy_milon	0.9003

c) For our query 5, the result is a bit different. Since the result is in a two-step manner, we use two different colors and different link strength to present this relationship between the keyword and the destinations found. Because there're too many reachable nodes from a single string, we only choose limited returned results.



Text

home

Query Type (Description)

5

no optimization [✓ optimization]

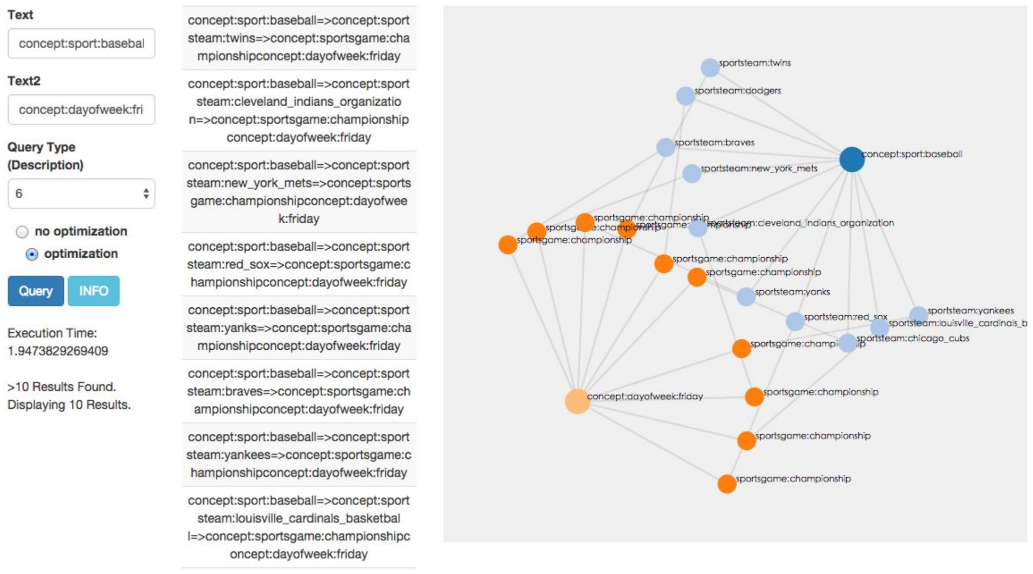
Query INFO

Execution Time:
0.00049304962158203

>87 Results Found.
Displaying 87 Results.

concept:sportsteam:state_university
concept:sportsgame:series
concept:sportsgame:test_series
concept:city:contact
concept:book:contact
concept:politicianus:form
concept:book:taking
concept:buildingfeature:systems
concept:musicalbum:nine
concept:book:beasts
concept:book:artists
concept:hobby:fishing
concept:city:vegas
concept:weatherphenomenon:two
concept:emotion:ten
concept:musicartist:south
concept:personcanada:search
concept:lake:new
concept:weapon:north
concept:geopoliticalorganization:this_n

d) For our query 6, the result is a bit different. If we choose the query type as 6, another text field will automatically appear. We can then query the path between the two entities. However, not able to afford so much computational resource and visualization cost in browser, we alter the query a bit different compared to the one in the experiment. We now aim to find the path between two entities via three steps. By this there're still a lot of results. We confine the number of the returned results and thus display the outcome.



Text

concept:sport:basebal

Text2

concept:dayofweek:fri

Query Type (Description)

6

no optimization [✓ optimization]

Query INFO

Execution Time:
1.9473829269409

>10 Results Found.
Displaying 10 Results.

concept:sport:baseball=>concept:sportsteam:twins=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:cleveland_indians_organization=>concept:sportsgame:championship concept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:new_york_mets=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:red_sox=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:yanks=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:braves=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:yankees=>concept:sportsgame:championshipconcept:dayofweek:friday
concept:sport:baseball=>concept:sportsteam:louisville_cardinals_basketball=>concept:sportsgame:championshipconcept:dayofweek:friday

Conclusion

In the experiment, we separate the initial data into three different parts and create four schemas to implement all the queries. We build the index for the schema and improve the query efficiency. We also design another two extra queries and try to combine the graphic theory with the database system. We complete the query based on BFS and the 2-step schema, with which we analyze the performance of the query and our assumption.