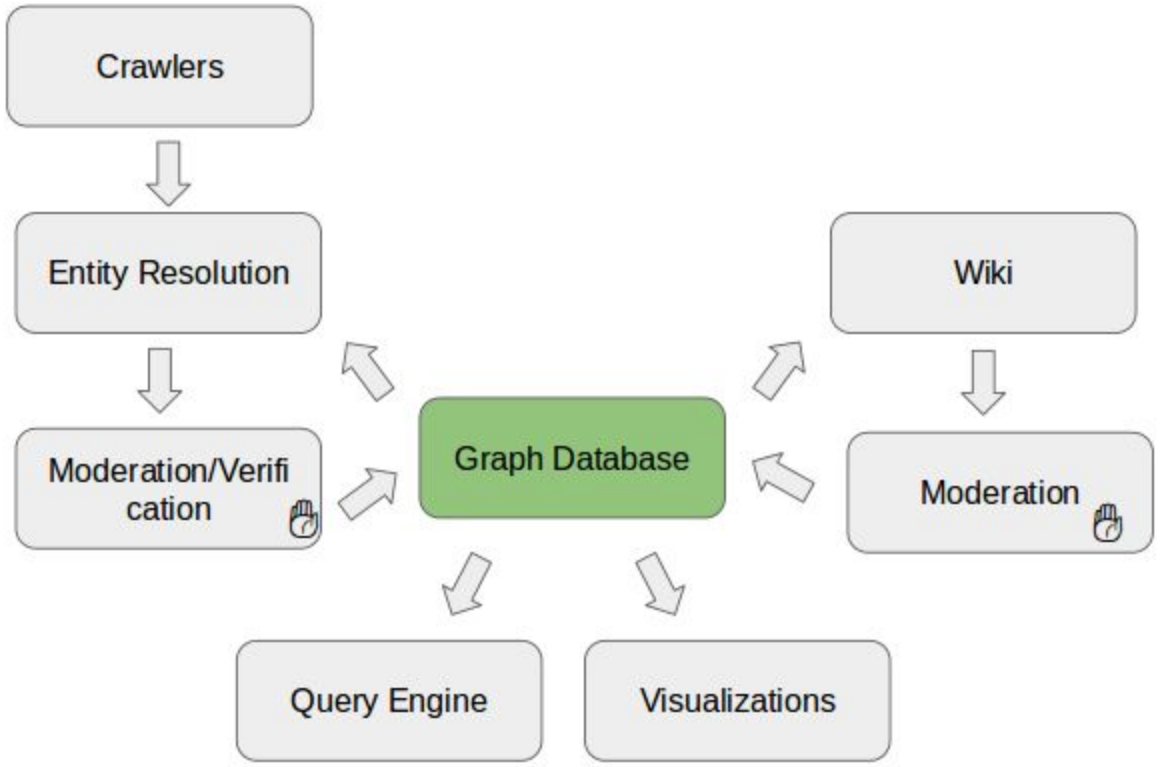
System Design Doc
[Project plan & Mostly how the crawler operates]

Goal: To construct a wiki-like system having political, corporate, bureaucratic data with overlapping networks which will help academicians, journalists, researchers, etc. to dig patterns, come up with potential red flags, interesting insights, etc.

Proposed system:



- Crawlers fetch data which is resolved against the graph database, and then verified by human moderator. Any new data is then fed to the graph database.
- Any end user can add more entities/relations to the graph database using the wiki-like interface. Any to-be-updated information is first human moderated.
- Query engine supports graph like queries over the graph database, and also lets the user see the query results in a graph form.

Class of human actors using the system: Moderator, Verifier, Admin, End-user.

- Moderator: Moderates all updates made to the graph database through the wiki interface.
- Verifier: Checks and verifies the entity resolution. Picks up the correct resolved entry.

Two Broad Views: The core data/graph-database is **either updated by an end-user or a bot.**

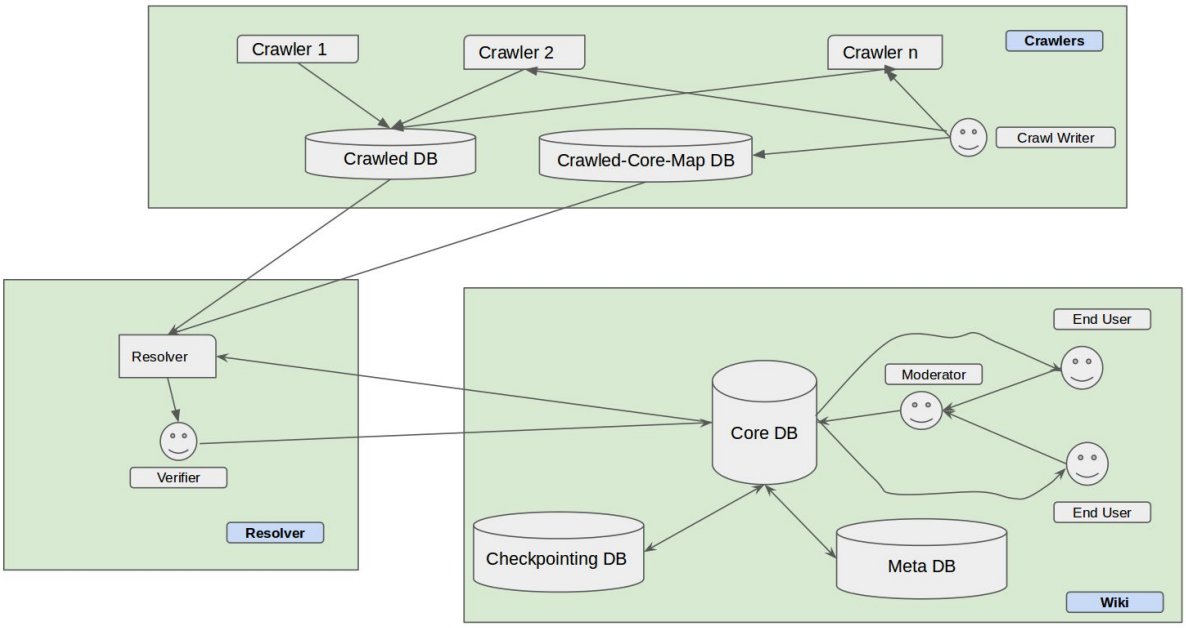
End user can interact the system in these ways:

1. Access an entity's related info: **[READ]**
    a. See an entity's page.
    b. Search for an entity.
    c. See relationship between two entities.
    d. Look at an entity's influence graph.
    e. See an entity's close connections - family/historical, political, educational, corporate, etc.

2. Update an entity's attributes: Update entity's properties/info, etc. **[UPDATE]**
3. Add an entity: If the entity is not already in the system, the user can create an entity, add attributes. **[CREATE]**
4. Merge request for two entities: The user can request merge of two entities, if he finds that both the entities are same. The merge request is approved by the moderator. **[MERGE]**
5. Merge request for two relations between same entities: The merge request is approved by the moderator. **[MERGE]**
6. Add a relation between two existing entities. **[CREATE]**
7. Update a relation's properties. **[UPDATE]**
8. Delete an entity and all its relations. Just mark it deleted. [The system would be able to revert back due to checkpointing/versioning] **[DELETE]**
9. Delete a relation. [The system would be able to revert back due to checkpointing/versioning] **[DELETE]**

The bot being a crucial part of the system is basically a combination of automated scripts[code] and verifiers[people]. Crawlers fetch data → store in some temporary layer of RDMS → the data is picked row by row/node by node and resolved with the core DB. The resolved results are shown to the verifier who verifies/picks a one-one matching. The data is correspondingly updated.

System Pipeline



Persistent Databases in the system

The system will have five logical databases.
(Physically it may be just one/two/three acting as five different ones)

1. One is the **core-DB** which will have the entire updated, current, resolved social network.
2. One is the **accompanying checkpointing-DB** which will have the entire historical information of the core DB to easily revert/switch to any checkpoint in the history of the core DB.
3. One is the **core-metadata-DB** which will map all the information (every update/insert/delete) to a source/site, etc.
4. One is the **crawled-DB** containing all the info crawled from various sources with the crawled metadata : source, date, etc.
5. One is the **crawled-core-map-DB** which will help in mapping the crawled info to the core DB data model.

The flow in the pipeline:

1. Information crawled is saved to the *crawled-DB*, with the crawled metadata. And is marked unresolved for now.

2. The developer who writes a crawler is also bound to provide a mapping info to resolve the crawled info with the info already in the *core-DB*. This map info is saved in *crawled-core-map-DB*.

3. Very crucial step : resolving + verifying.

    1. All the unresolved information is row by row presented to the verifier.
    2. First the entities are resolved for a row and then the relations. This way a row of the *crawled-DB* is like a subgraph which is to be mapped to the *core-DB*'s graph.
    3. If some entity/relation is resolved we update the new info, if it is not resolved we create a new entity/relation in the *core-DB*.
    4. We use the *crawled-core-map-DB* to handle the mapping.
    5. Any information that is resolved is marked resolved in the *crawled-DB*, so that next time the verifier is presented with new information.

4. The end users never see the *crawled-DB*, they only work on the *core-DB* directly. All updates/checkpoints are saved to the accompanying *checkpointing-DB*.

Logic behind Resolution+verification: Examples

Certain example cases are enlisted below. Sub-graphs below represent a row in the *crawled-DB*.

1. Sub-graph consists of just a node. If it is found in *core-DB*, the corresponding properties are updated. Else a new entity is created with corresponding label*s* and attributes.

2. Sub-graph consists of two nodes and a relation between them. Nodes are resolved first as described in point 1. After this step, we will have two node ids: the only thing left to check is if the said relationship exists between the two nodes. If the relationship is found in *core-DB*, the corresponding properties are updated. Else a new relation is created with corresponding label and attributes.

3. Sub-graph consists of three nodes and three relations between them which form a kind of circular graph. The key is that this can be broken down as: (A)-[R1]->(B). (B)-[R2]->(C). (C)-[R3]->(A). When we understand that any subgraph can be broken down into such components, all our problem is now broken down to case like in point 1 and 2.

Here, the nodes A,B,C are resolved as in point 1. After this step, we will have three node ids. And then we can go about resolving the relations R1, R2, and R3.

Detailed working of verifier+resolver

This can be best explained by taking examples.

Example 1:

Let us say that the crawler crawls data and saves it in a format like:

name	age	sex

Table 1.

Also the crawler has an accompanying map-data information which looks like this:

entityno	graph_label	graph_props	mysql_props	graph_reslove_props	mysql_resolve_props
1	person	name,age,sex	name,age,sex	name	name

Table 2.

graph_props have one to one order wise mapping with mysql_props.
graph_resolve_props have one to one order wise mapping with mysql_reslove_props.

So, basically in the crawled data, each row represents a node with label :person and attributes [name, age, sex] in the *core-DB.*

1. Now, when a verifier logs-in a row from the TABLE1 is fetched, the TABLE2 is used to frame a query to search a person with the name like in the row.
2. Some nodes are suggested after the query to the verifier.
3. If verifier selects one of the propsed nodes, the information if new is updated to the resolved node.
4. Else if the verifier doesn't find any matching node, a new node correspoding to the selected row is made and inserted in the *core-DB*.

Example 2:

Let us say that the crawler crawls data and saves it in a format like in the *crawled-DB*:

name1	name2	spouse_status
X	A	divorced
Y	B	current
Z	C	current

Table 3.

Also the crawler has an accompanying map-data information which looks like this in the **crawled-core-mapping-DB**

entity_no	graph_label	graph_props	mysql_props	graph_resolve_props	mysql_reslove_props
1	person	name	name1	name	name1
2	person	name	name2	name	name1

Table 4.

This example also has a relation the mapping info of which looks like:

rel_no	entity1	entity2	directionfrom	graph_label	graph_props	mysql_props	graph_resolve_props	mysql_reslove_props
1	1	2	any	spouse	status	spouse_status	status	spouse_status

Table 5.

So, basically in the crawled data, each row represents two nodes with label :person and *and* a spouse relation between them. Nodes are resolved first as described in example 1. After this step, we will have two node ids: the only thing left to check is if the said relationship exists netween the two nodes. If the relationship is found in *core-DB*, the corresposing properties are updated. Else a new relation is created with correspoding label and attributes.

Data sets-

Politician(id, Name,mynetaid,Location,Party, spouse_name,,spouse_profession,ls_year, (other_fields))
IAS(Name,DOJ,loc,year,DOB,IASID)
directors(Did,cid,name,cname,designation,state)
company_value(parag_id,income,expenditure,profit,value)
donation(id,company,party,amt_rs_lakh,year)

Nodes/Entities:

(:Person{uuid:"", name:"", address_location:"", DOB:""})
(:Politician {uuid:"", name:"", mynetaid_electionname_year:"", constituency:""})
(:Alias {uuid:"",name:"",context:""})
(:Party{uuid:"",name:"",type:"[national,state,regional]",start_year="",HQ=""})
(:Company{uuid:"",cin:"",name:"",location:"",income:"",expenditure:"",profit:"",value:""})
(:IAS {uuid:"",name:"",DOJ:"",year:"",IAS_ID:"",posting_location:""})
(:Employee{uuid:"",din:"",name:"",designation:""})
(:Govt_Body{uuid:"",name:"",location:""})

Relationships (Edges)

(:Politician{})--[:member_of{id:"",type:"",years:[]}]-->(:Party{})
(:Politician{})--[:member_of{id:"",type:"",years:[]}]-->(:Govt_Body{})
(:IAS{})--[:member_of{id:"",type:"",years:[]}]-->(:Govt_Body{})
(:Employee{})--[:employee_of{id:"",designation:"",years:[]}]-->(:Company{})
(:Company{})--[:donated{id:"",cin:"",party:"",amt:"",year:""}]-->(:Party{})
(:Person{})--[:family{id:"",is_biological:""}]-->(:Person{})
(:Person{})--[:profession{id:"",type:""}]-->(:Politician{})
(:Person{})--[:profession{id:"",type:""}]-->(:Corporate{})
(:Person{})--[:profession{id:"",type:""}]-->(:IAS{})
(:Person{})--[:aka{id:""}]-->(:Alias{})