Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora

Soumen Chakrabarti
Kriti Puniyani
Sujatha Das

IIT Bombay

(In fewer words)

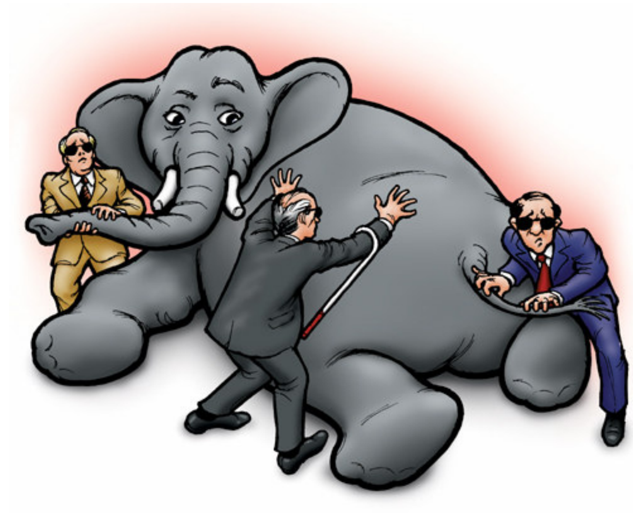
Ranking and Indexing for Semantic Search

Soumen Chakrabarti
Kriti Puniyani
Sujatha Das

IIT Bombay

Working notion of semantic search

- Exploiting in conjunction
  - “Strings with meaning” – entities and relations
  - “Uninterpreted strings” – as in IR
- This paper
  - Only “is-a” relation
  - Token match
  - Token proximity
- Can approximate many info needs

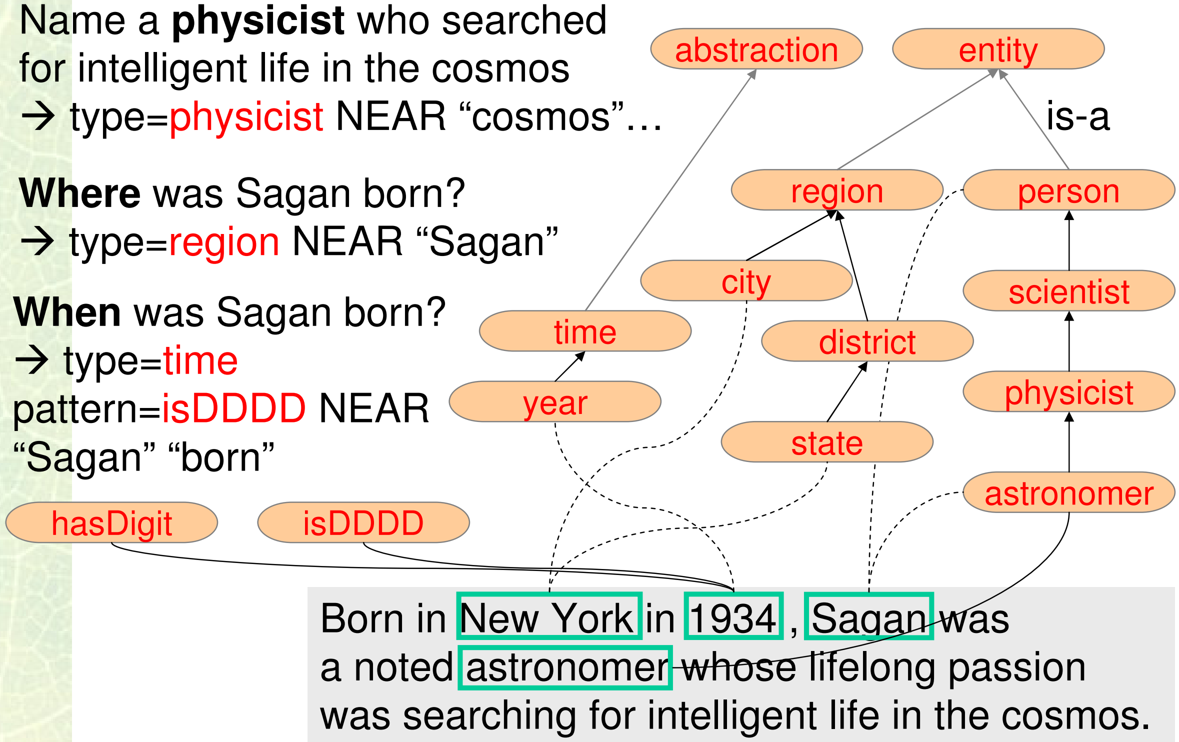


Type-annotated corpus and query e.g.

Name a **physicist** who searched for intelligent life in the cosmos
→ type=physicist NEAR “cosmos”…

Where was Sagan born?
→ type=region NEAR “Sagan”

When was Sagan born?
→ type=time pattern=isDDDD NEAR “Sagan” “born”



The query class we address

- Find a token span *w* (in context) such that
  - *w* is a mention of entity *e*
    - "Carl Sagan" or "Sagan" is a mention of the concept of that specific physicist
  - *e* is an instance of **atype** *a* given in the query
    - Which *a*=physicist …
  - *w* is "NEAR" a set of **selector** strings
    - "searched", "intelligent", "life", "cosmos"
- All uncertain/imprecise; we focus on #3
- Yet surprisingly powerful: correct answer within top 3—4 *w*'s for TREC QA benchmark

Contribution 1: What is "NEAR"?

- XQuery and XPath full text support
  - (distance at most|window) 10 words [ordered] – hard proximity clause, not learnt
  - ftcontains … with thesaurus at … relationship "narrower terms" at most ℓ levels
- No implementation combining "narrower terms" and "soft" proximity ranking
- Search engines favor proximity in proprietary ways

A learning framework for proximity

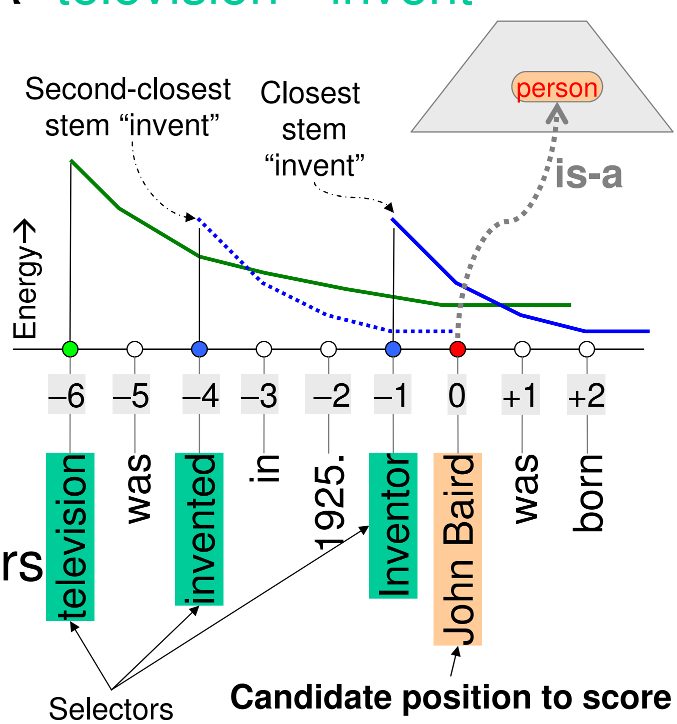
Contribution 2: Indexing annotations

- type=person NEAR theory relativity → type in {physicist, politician, cricketer,…} NEAR theory relativity
  - Large fanout at query time, impractical
- Complex annotation indexes tend to be large
  - Binding Engine (WWW 2005): 10x index size blowup with only a handful of entity types
  - Our target: 18000 atypes today, more later
- Workload-driven index and query optimization
  - Exploit skew in query atype workload

7

Part-1: Learning to score token spans

- type=person NEAR "television" "invent*"
- Rarity of selectors
- Distance from candidate position to selectors
- Many occurrences of one selector
  - Closest is good
- Combining scores from many selectors
  - Sum is good



8

Learning the shape of the decay function

- For simplicity assume left-right symmetry
- Parameters $(\beta_1,\ldots,\beta_W)$, W=max gap window
- Candidate position characterized by a feature vector $f = (f[1],\ldots,f[W])$
  - If there is a matched selector s at distance j and
  - This is the closest occurrence of s
  - Then set $f[j]$ to *energy*(s), … else 0
- Score of candidate position is $\beta \cdot f$
- If we like candidate u less than v ("$u \prec v$")
  - We want $\beta \cdot f_u \le \beta \cdot f_v$
  - Assess a penalty proportional to $\exp(\beta \cdot f_u - \beta \cdot f_v)$

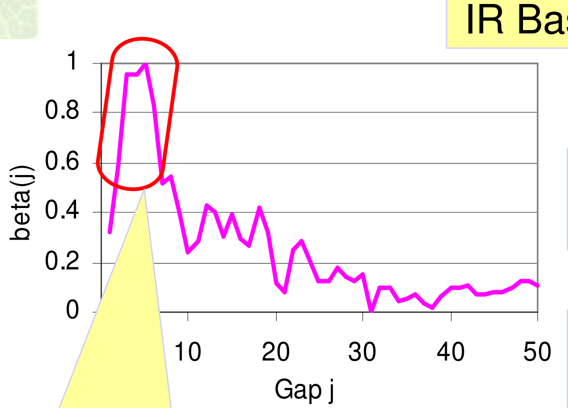
Learning decay function—results

$$\min_{\beta} \sum_{j=1}^{W} (\beta_j - \beta_{j+1})^2 + C \sum_{u \prec v} \exp(\beta \cdot f_u - \beta \cdot f_v)$$

Discourage adjacent βs from differing a lot

Penalize violations of preference order

IR Baseline



Train	Test	MRR
IR	2000	0.16
2001	2000	**0.29**

TREC year

Roughly unimodal around gap = 4 and 5

Mean reciprocal rank: Average over questions, reciprocal of the first rank where an answer token was found (large good)

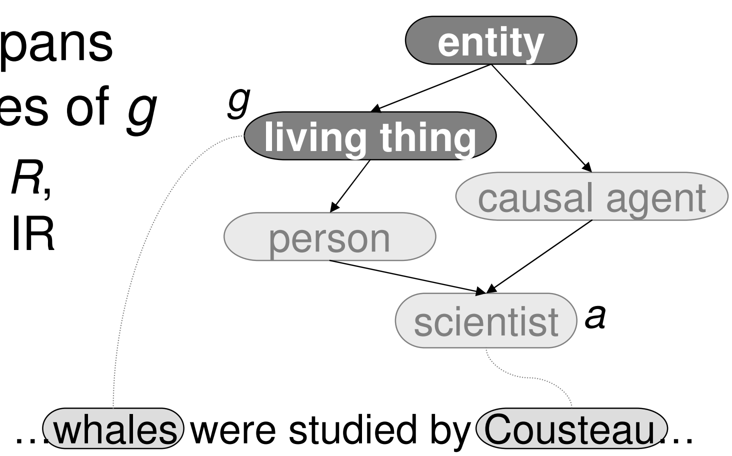
Part-2: Workload-driven indexing

- Type hierarchies are large and deep
  - 18000 internal and 80000 leaf types in WordNet
- Runtime atype expansion time-intensive
  - Even WordNet knows 650 scientists, 860 cities…
- Index each token as all generalizations
  - Sagan → physicist, scientist, person, living thing
  - Large index space bloat
- Index a subset of atypes

Corpus/Index	Gbytes
Original corpus	5.72
Gzipped corpus	1.33
Stem index	0.91
Full type index	4.30

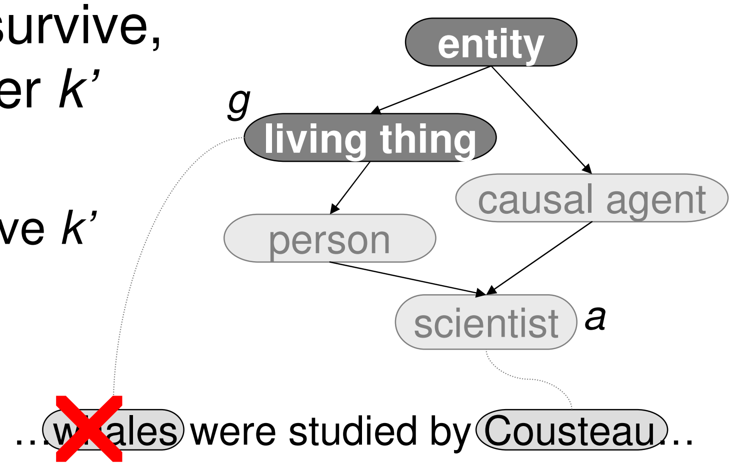
Pre-generalize (and post-filter)

- Full set of "atypes" (answer types) is A
- Index only a "registered" subset R of A
- Say query has atype a; want k answers
- Find a's "best" generalization $g \in R$
- Get best k' > k spans that are instances of g
  - Given index on R, this is standard IR (see paper)



(Pre-generalize and) post-filter

- Fetch each high-scoring span *w*
- Check if *w* is-a *a*
  - Fast compact "forward index" (doc,offset)→token
  - Fast small "reachability index", common in XML
- If fewer than *k* survive, restart with larger *k'*
  - Expensive
  - Pick conservative *k'*



Estimates needed by optimizer

- If we index token ancestors in *R* as against ancestors in all of *A*, how much index space will we save?
  - Cannot afford to try out and see for many *R*s
- If query atype *a* is not found in *R* and we must generalize to *g*, what will be the bloat factor in query processing time?
  - Need to average over a representative workload

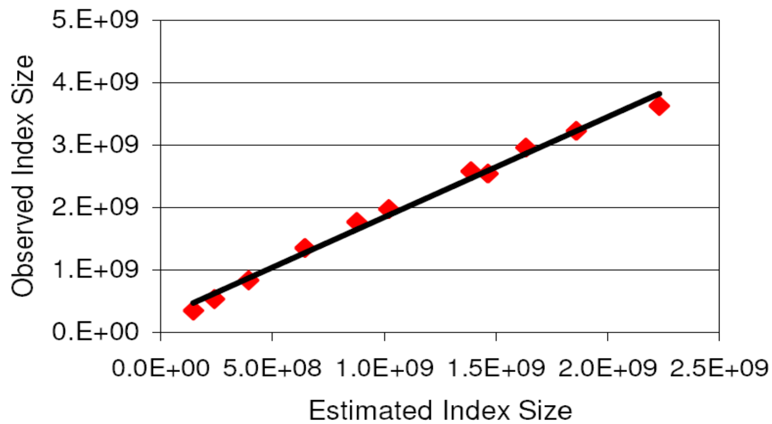
Index space estimate given R

- Each token occurrence leads to one posting entry
- Assume index compression is a constant factor
- Then total estimated index size is proportional to

$$\sum_{r \in R} corpusCount(r)$$

Number of tokens in corpus that connect up to *r*

- Surprisingly accurate!



Processing time bloat for one query

- If $R=A$, query takes time approximated by

$$t_{\text{scan}}\, corpusCount(a)$$

Time to score one candidate position while scanning postings

Number of occurrences of descendants of type *a*

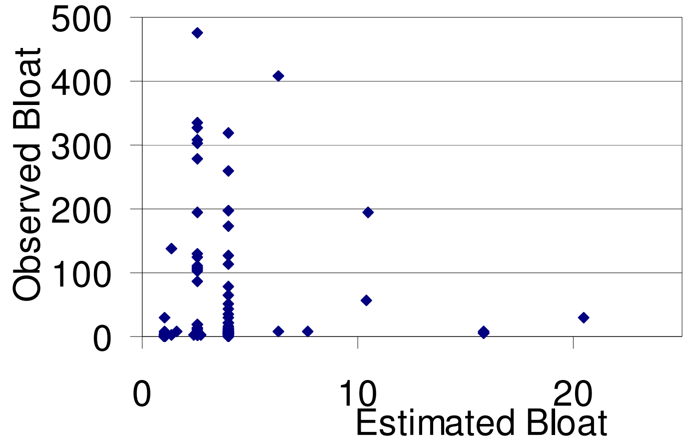
- If *a* cannot be found in *R*, the price paid for generalization to *g* consists of
  - Scanning more posting entries: $t_{\text{scan}}\, corpusCount(g)$
  - Post-filtering *k'* responses: $k'\, t_{\text{filter}}$
- Therefore, overall bloat factor is

$$queryBloat(a, R) = \frac{t_{\text{scan}} corpusCount(g) + k'\, t_{\text{filter}}}{t_{\text{scan}} corpusCount(a)}$$

Time to check if answer is instance of *a* as well

Query time bloat—results

- Observed bloat fit not as good as index space estimate



- While observed::estimated ratio for one query is noisy, average over many queries is much better

Expected bloat over many queries

Prob of new query having atype *a*

$$\sum_{a \in A} queryProb(a)\ queryBloat(a, R)$$

Already estimated

- Maximum likelihood estimate

$$queryProb_{\mathrm{Train}}(a) = \frac{queryCount_{\mathrm{Train}}(a)}{\sum_{a' \in A} queryCount_{\mathrm{Train}}(a')}$$

- Many *a*'s get zero training probability
→ Optimizer does not register *g* close to *a*
- Low-prob atypes appear in test → huge bloat
- Collectively matter a lot (heavy-tailed distrib)

Smoothing low-probability atypes

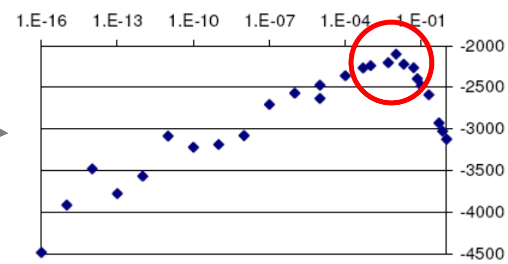
- Lidstone smoothing:

$$queryProb_{\text{Train}}(a) = \frac{queryCount_{\text{Train}}(a) + \ell}{\sum_{a' \in A}\left(queryCount_{\text{Train}}(a') + \ell\right)}$$

- Smoothing param ℓ fit by maximizing log-likelihood of held-out data:

$$\sum_{a \in \text{HeldOut}} queryCount_{\text{HeldOut}}(a) \log(queryProb_{\text{Train}}(a))$$

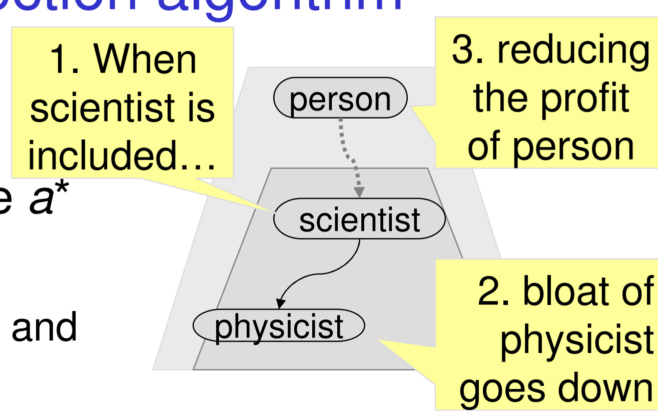
- Clear range of good fits for ℓ

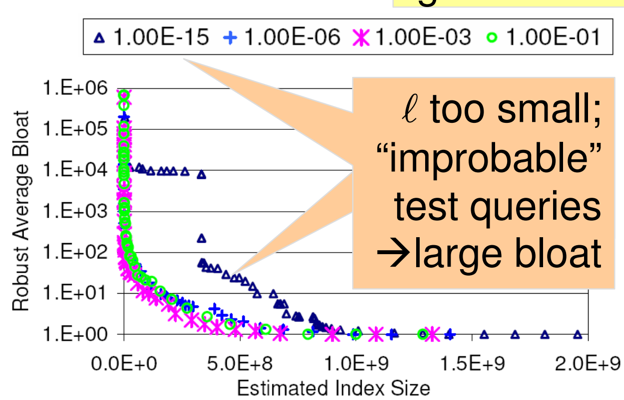


Smoothing param $\ell \rightarrow$

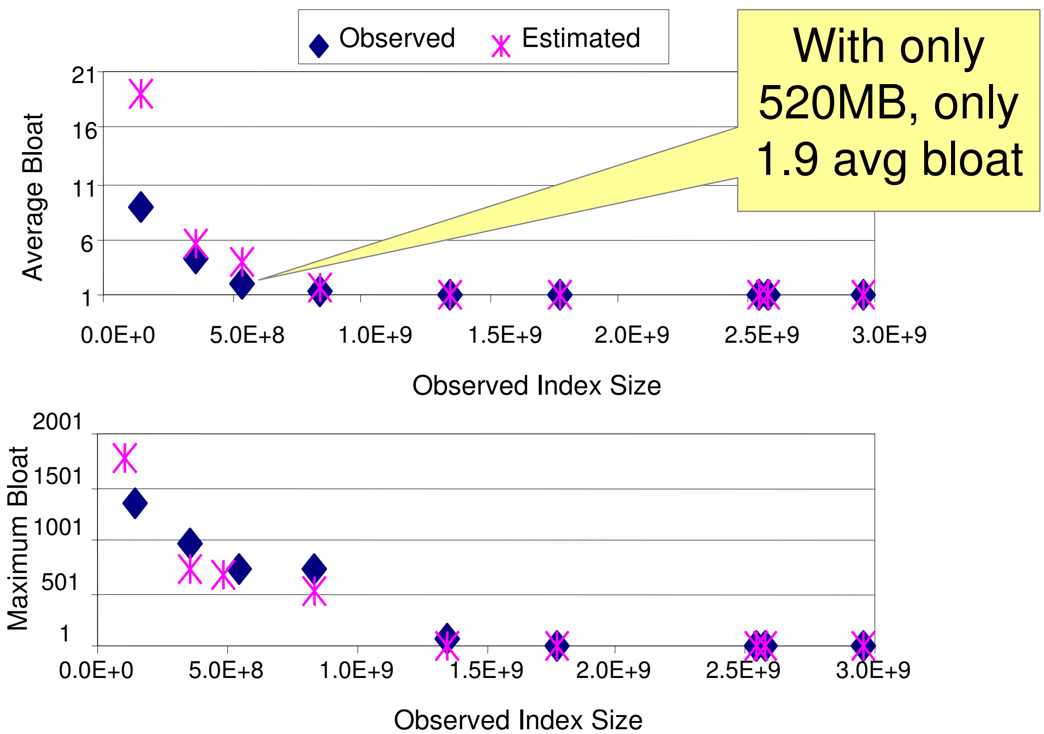
The R selection algorithm

- $R \leftarrow$ roots of A
- Greedily add the "most profitable" atype a^*
- Profit = ratio of
  - reduction in bloat of a^* and its descendants to
  - increase in index space
- Downward and upward traversals and updates
- Gives a tradeoff between index space and query bloat





Optimized space-time tradeoff



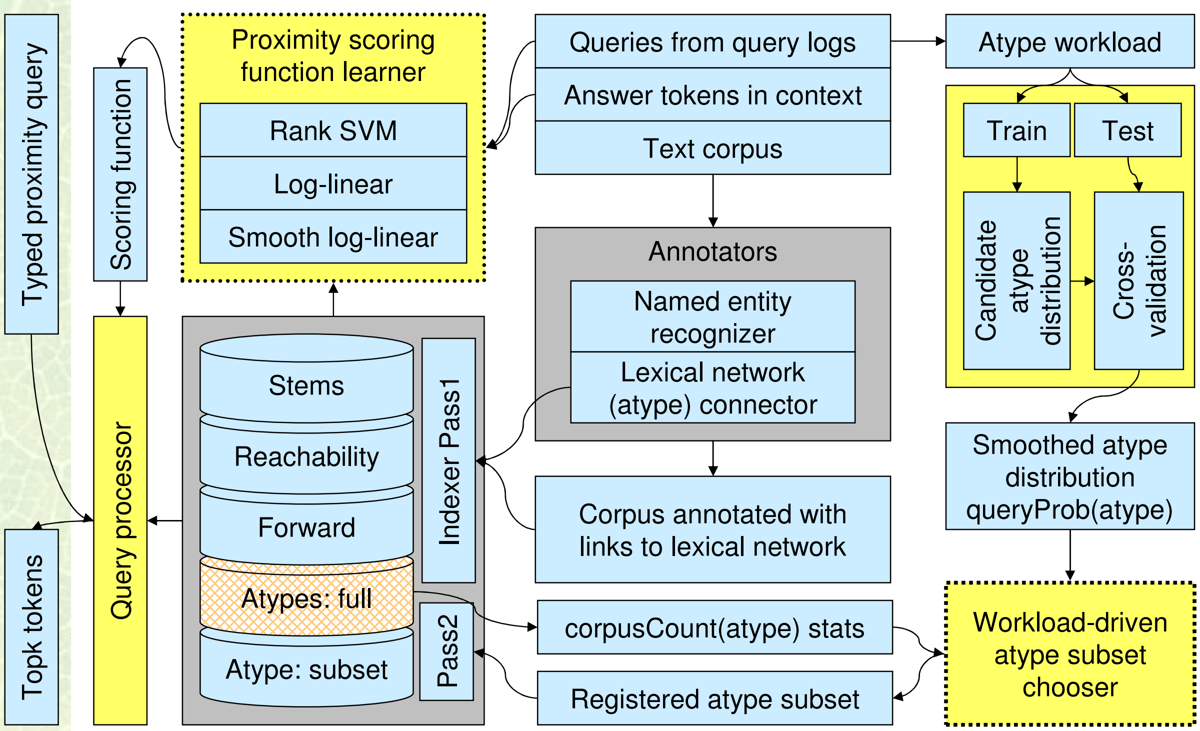
Optimized index sizes

Corpus/Index	Gbytes
Original corpus	5.72
Gzipped corpus	1.33
Stem index	0.91
Full type index	4.30
Reachability index	0.01
Forward index	1.16
Atype subset index	0.52

Summary

- Working prototype around Lucene and UIMA
  - Annotators attach tokens to type taxonomy
  - Query atype workload help compact index
  - Ranking function learnt from preference data
  - NL queries translated into atype+selectors
- Ongoing work
  - Indexing and searching relations other than is-a
  - More general notions of graph proximity
- Email soumen@cse.iitb.ac.in for code access

The big picture



Email soumen@cse.iitb.ac.in for code access