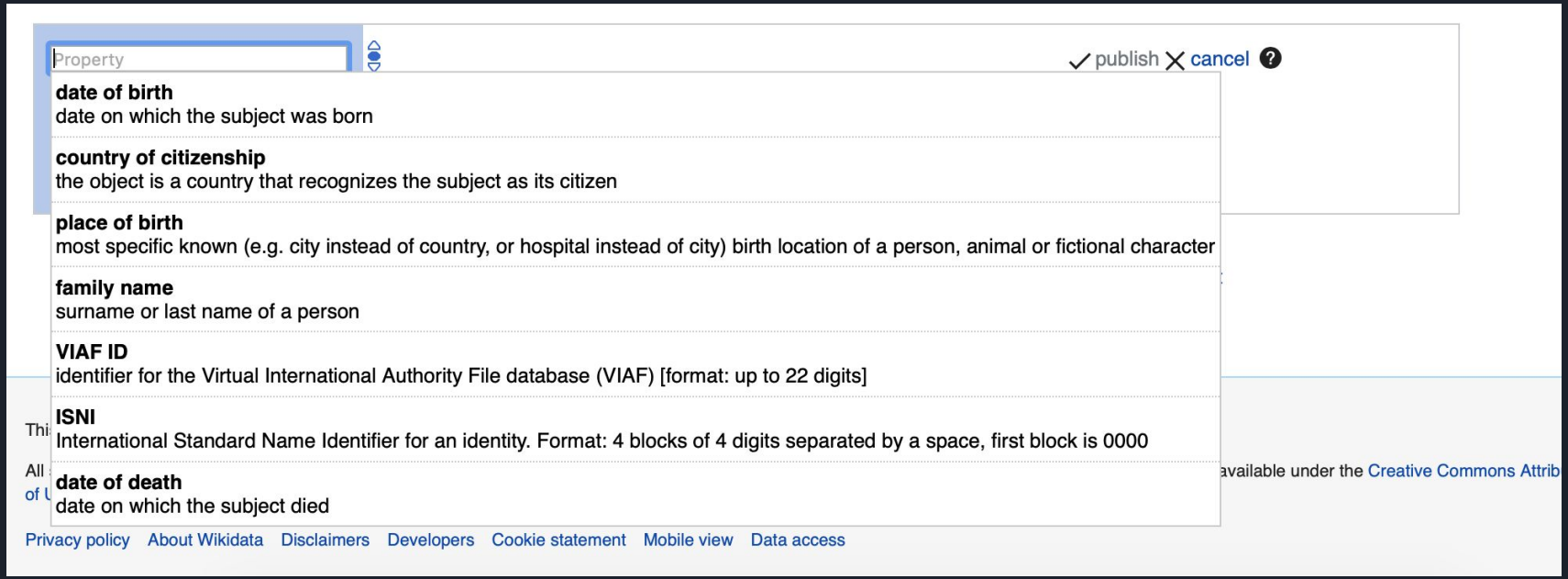
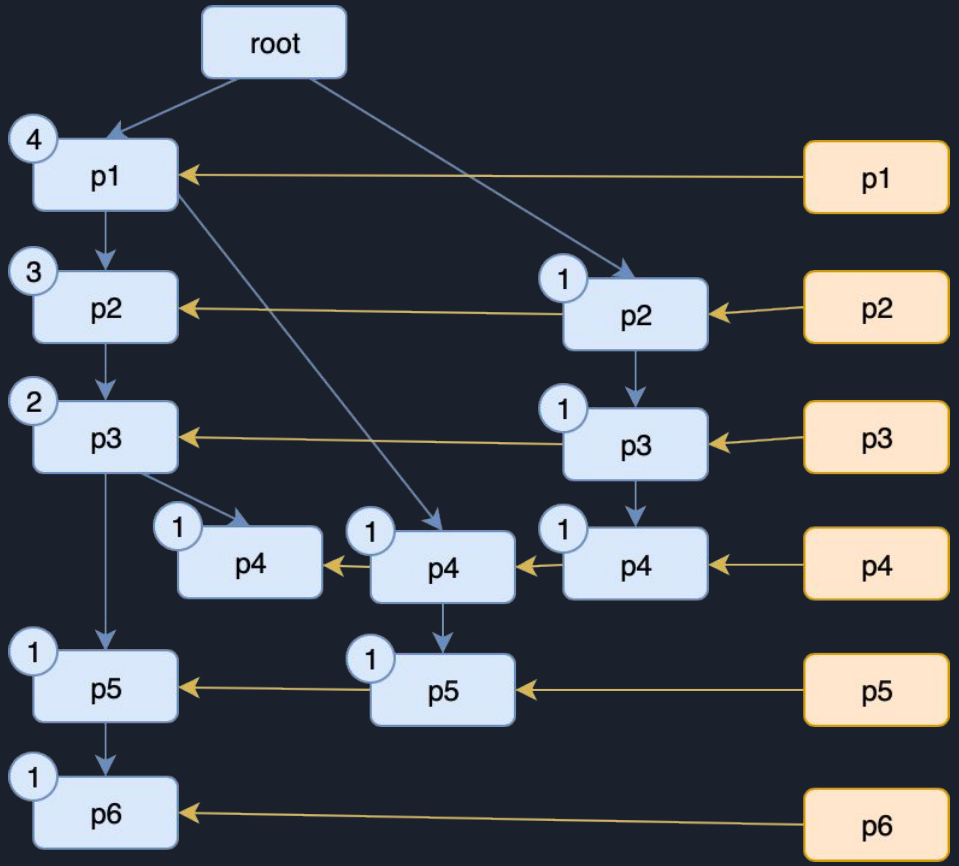
Recommender for Wikidata

Final Presentation - KGLab 2019

The Wikidata Recommender



Recommendation Example

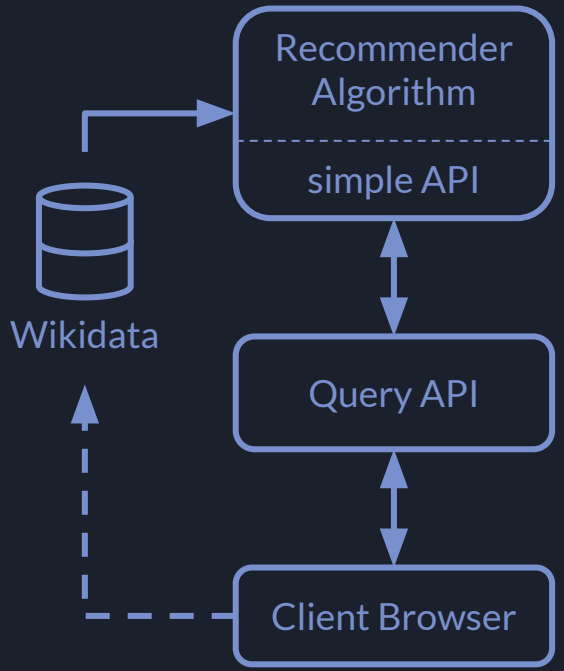


- [ ] structure
- [ ] prepare
- [ ] recommend
- [ ] output
- [ ] evaluate

Roadmap

1. Planning the Structure
    a. System planned vs. System developed
    b. Usability of the Recommender
2. Our Implementation
    a. Data Preparation Pipeline
    b. Integration of Types
    c. Integration of Backoff Strategies
    d. Glossary
    e. HTTP Input and Output
3. Evaluation

Initial System Diagram



↗ Improve

- ↗ add type information
- ↗ low info: minimum attributes
- ↗ maintainability: auto testing
- ↗ hierarchical types
- ↗ too much info: splitting
- ↗ optimization: less resources

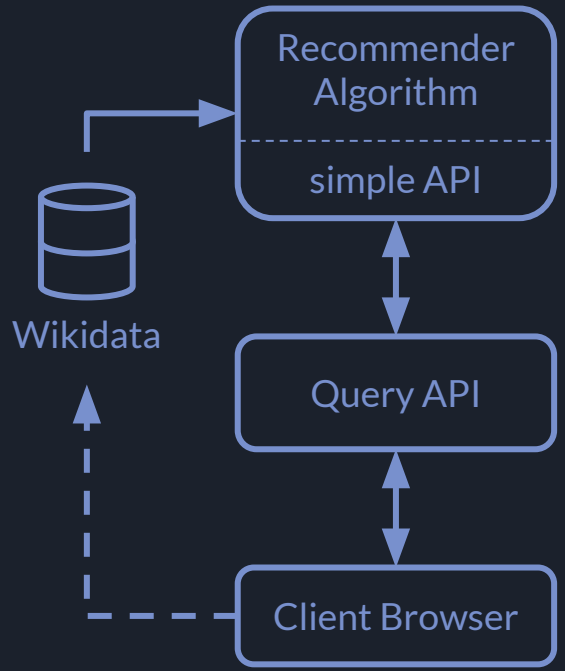
⇄ Integrate

- ⇄ add API: single responsibility
- ⇄ include user input
- ⇄ autocomplete: robust querying
- ⇄ feedback loop

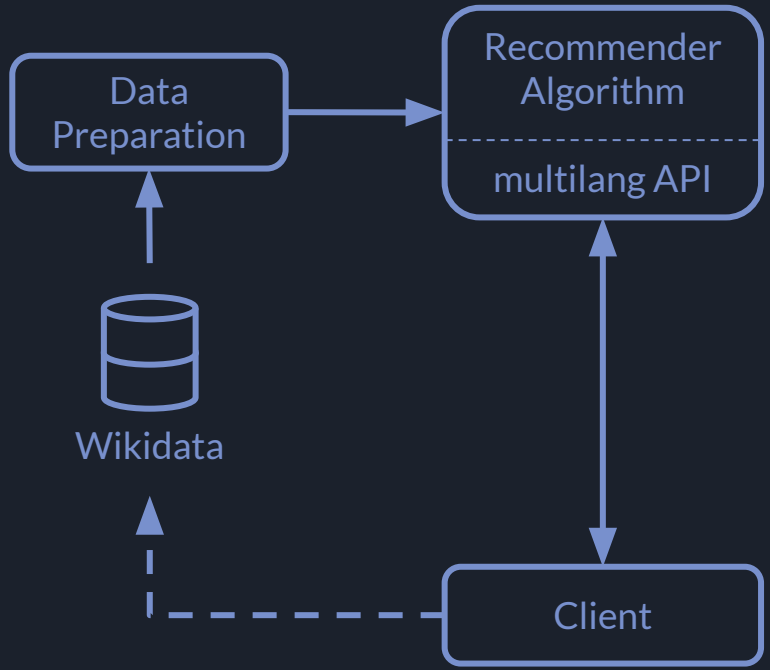
Evolution of the System Diagram

structure
prepare
recommend
output
evaluate

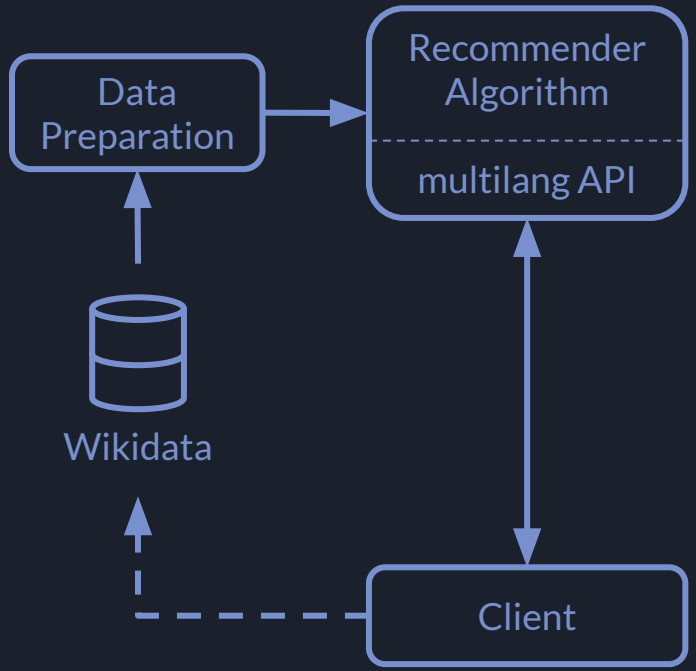
INITIAL IDEAS



ACTUAL DEVELOPMENT



Actual System Diagram



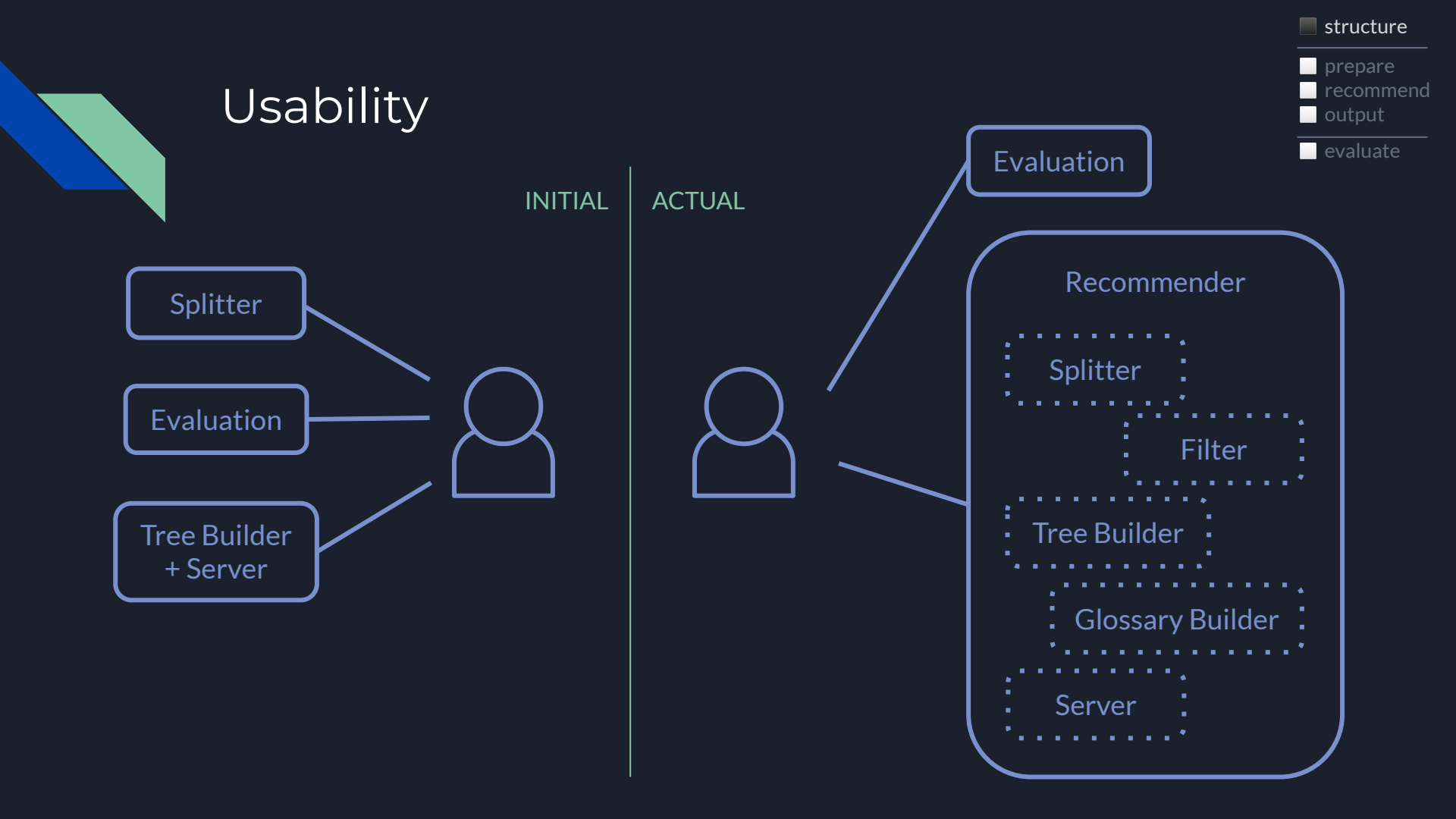
↗ Improve

- ↗ type information
- ↗ improved evaluation
- ↗ data preparation pipeline
- ↗ automatic testing
- ↗ backoff strategies
- ↗ strategy tuning
- ↗ multilang labelling
- ↗ code architecture

⇄ Integrate

- ⇄ extend current API
- ⇄ application usability

Usability

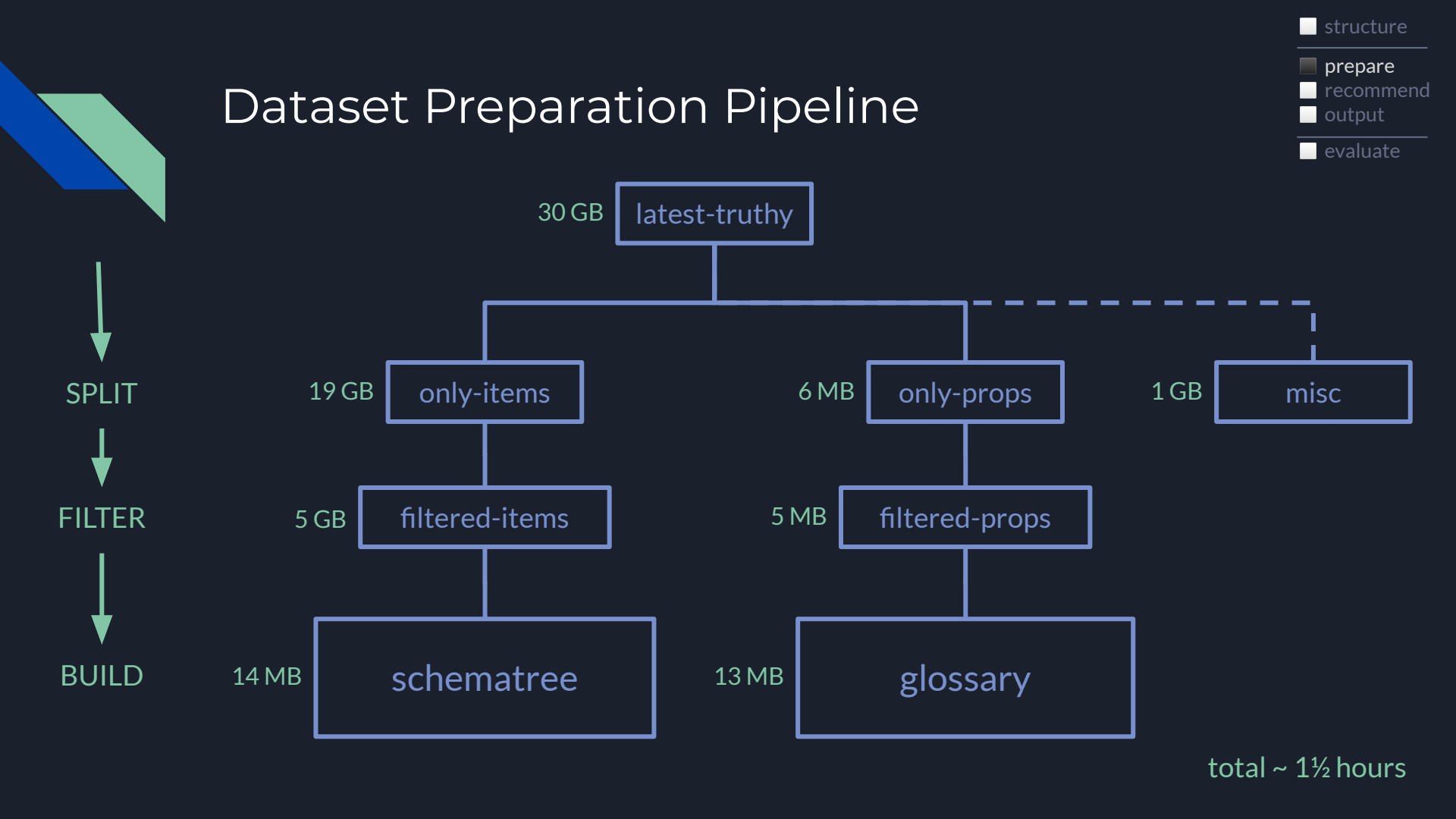

structure
prepare
recommend
output
evaluate
INITIAL
ACTUAL
Splitter
Evaluation
Tree Builder
+ Server
Evaluation
Recommender
Splitter
Filter
Tree Builder
Glossary Builder
Server

Usability

```
Usage:
  recommender [command]

Available Commands:
  build-dot        Build a DOT file from a schematree binary
  build-glossary   Build the Glossary that maps properties to multi-lingual descriptions
  build-tree       Build the SchemaTree model
  build-tree-typed Build the SchemaTree model with types
  filter-dataset   Filter a dataset using various methods
  help             Help about any command
  serve            Serve a SchemaTree model via an HTTP Server
  split-dataset    Split a dataset using various methods

Flags:
      --cpuprofile file   write cpu profile to file
  -h, --help              help for recommender
      --memprofile file   write memory profile to file
  -t, --time              measure time of command execution
      --trace file        write execution trace to file

Use "recommender [command] --help" for more information about a command.
```

```
$ recommender split-dataset by-prefix <dataset> -t
```

Dataset Preparation Pipeline



total ~ 1½ hours

Integrating Type Information

RDF data:

Entity	Property	Value
E1	P1	v1
E1	P2	v2
E1	P2	v3
E1	Ptype	T2
E1	Ptype	T3

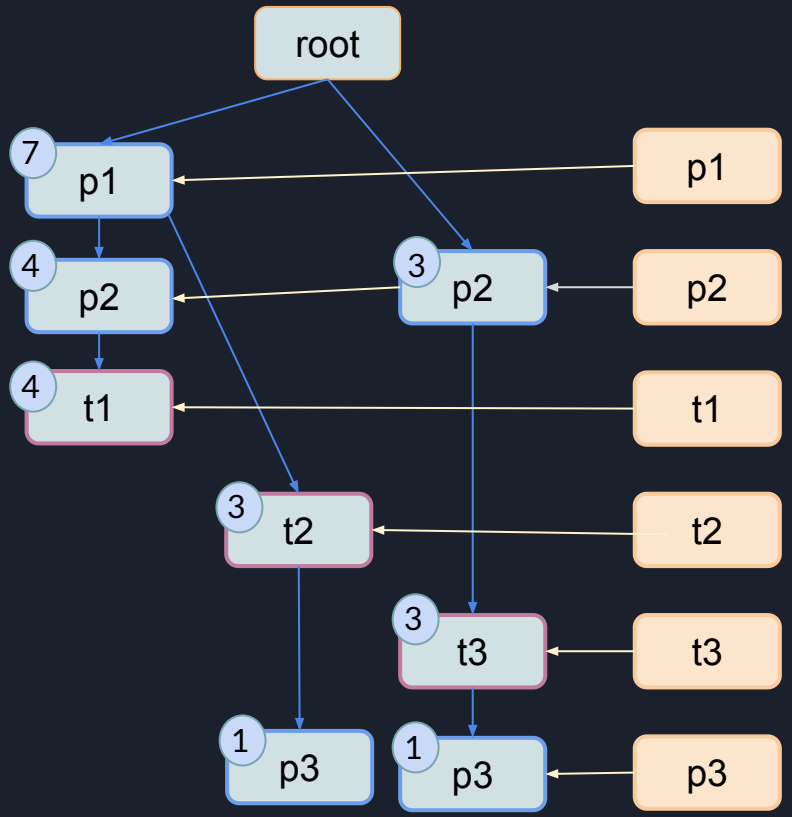
Parsed data:

{ (P1, 1), (P2, 2), (Ptype, 2) }

Parsed data with types:

{ (P1, 1), (P2, 2), (Ptype, 2), (T2, 1), (T3, 1) }

Integrating Type Information



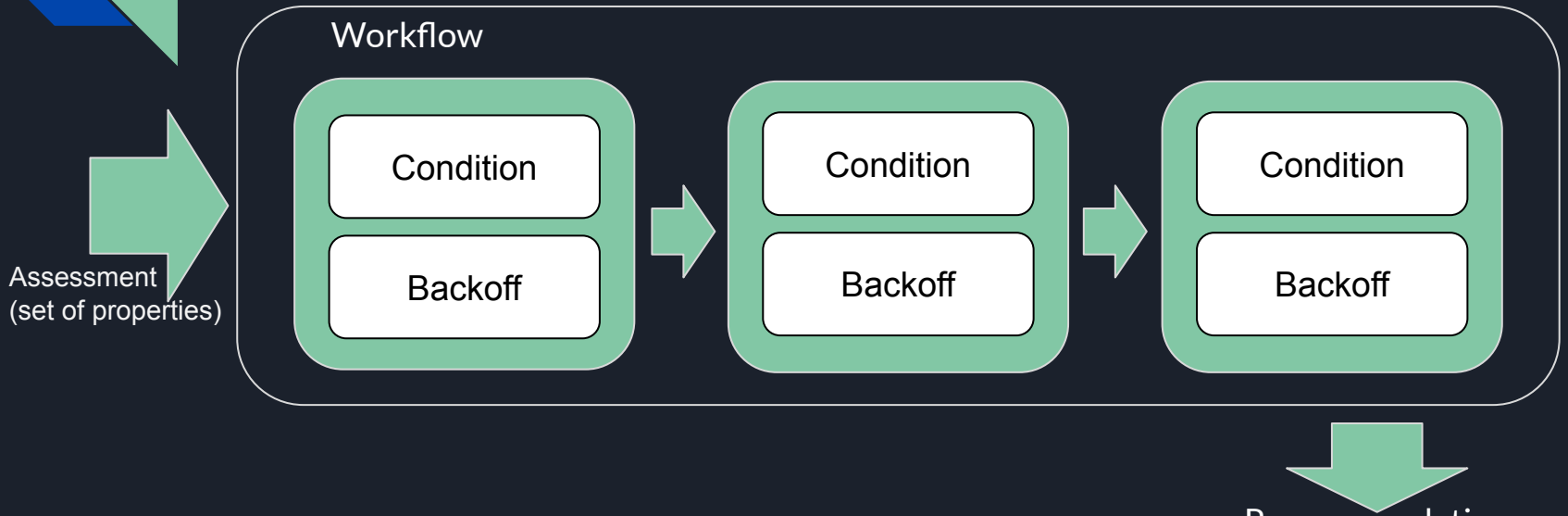
Query: {p3, t3}

Candidates: {p2}

Query: {p1, t2}

Candidates: {p3}

Integrating Backoff Strategies - Workflow

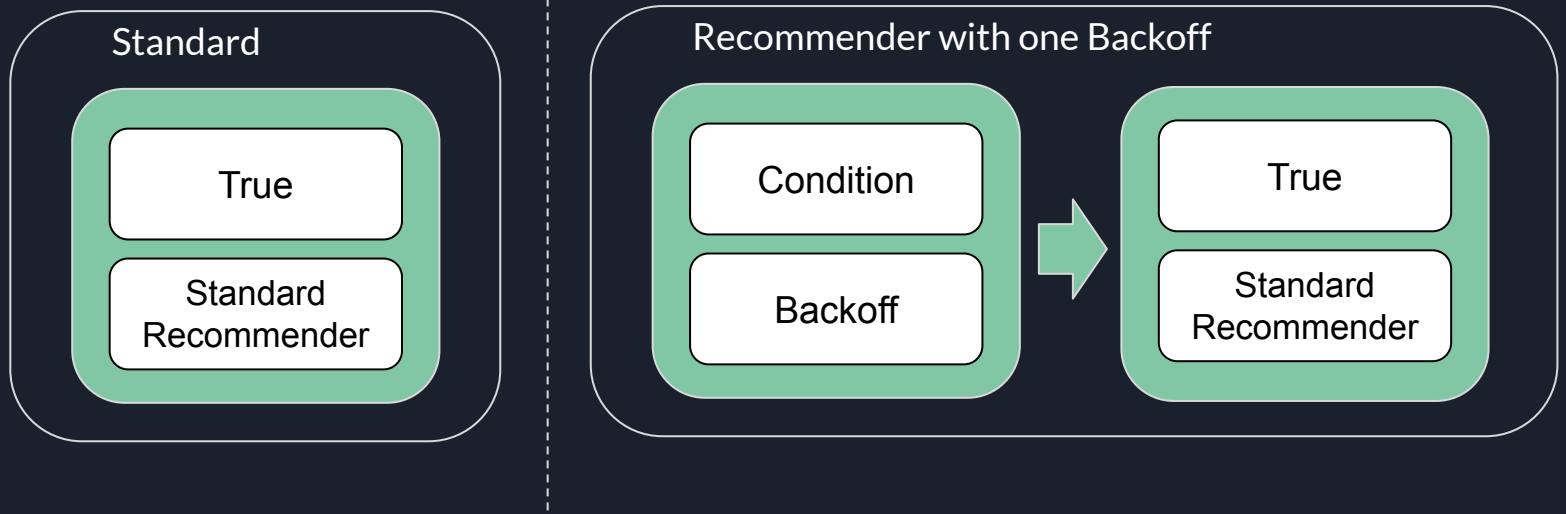


The workflow is highly adaptable

- realized as a stack like structure
- read from a config file when starting the recommender

property	probability
P201	0.83
...	...

Integrating Backoff Strategies - Workflow



Integrating Backoff Strategies - Conditions

1. Run the standard recommender
2. Based on the standards recommender result the condition enables the backoff:

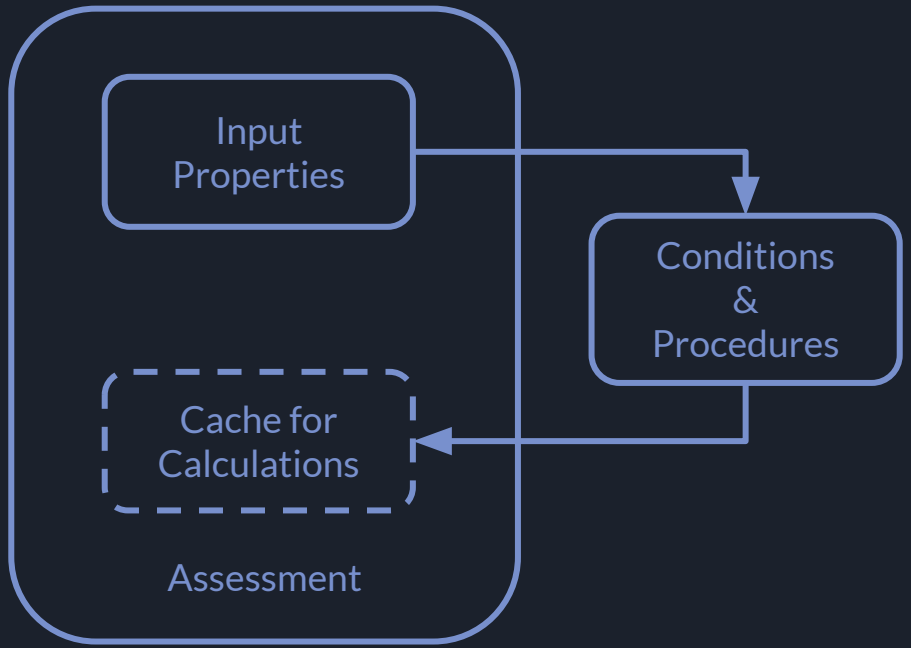
TooFewRecommendations(*n*):

➡ Enables if the standard recommendation got less than *n* properties

TooUnlikelyRecommendation(*f*):

➡ **Enables if the top 10 recommendations got less than *f* average probability**

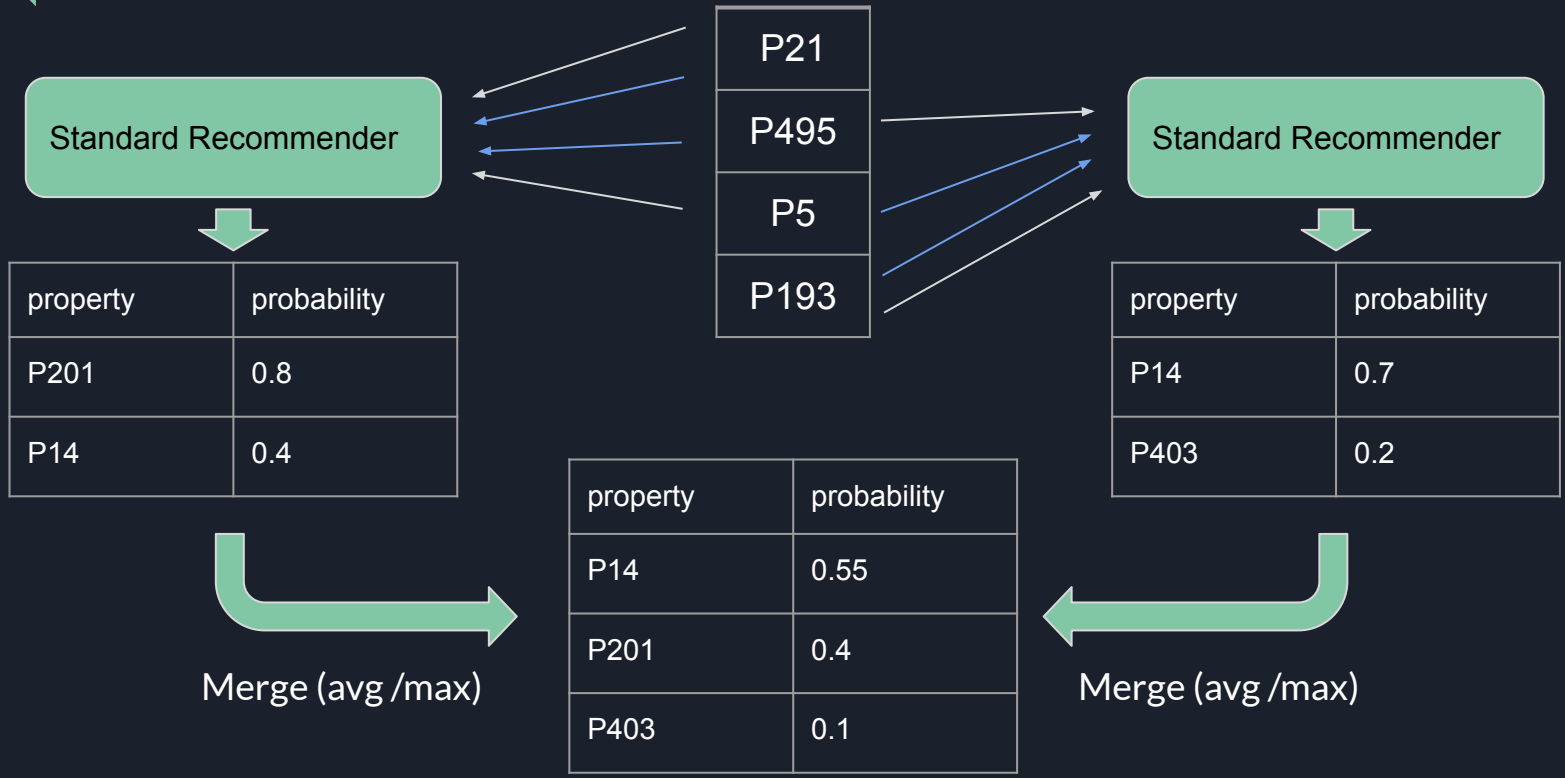
Integrating Backoff Strategies - Assessments



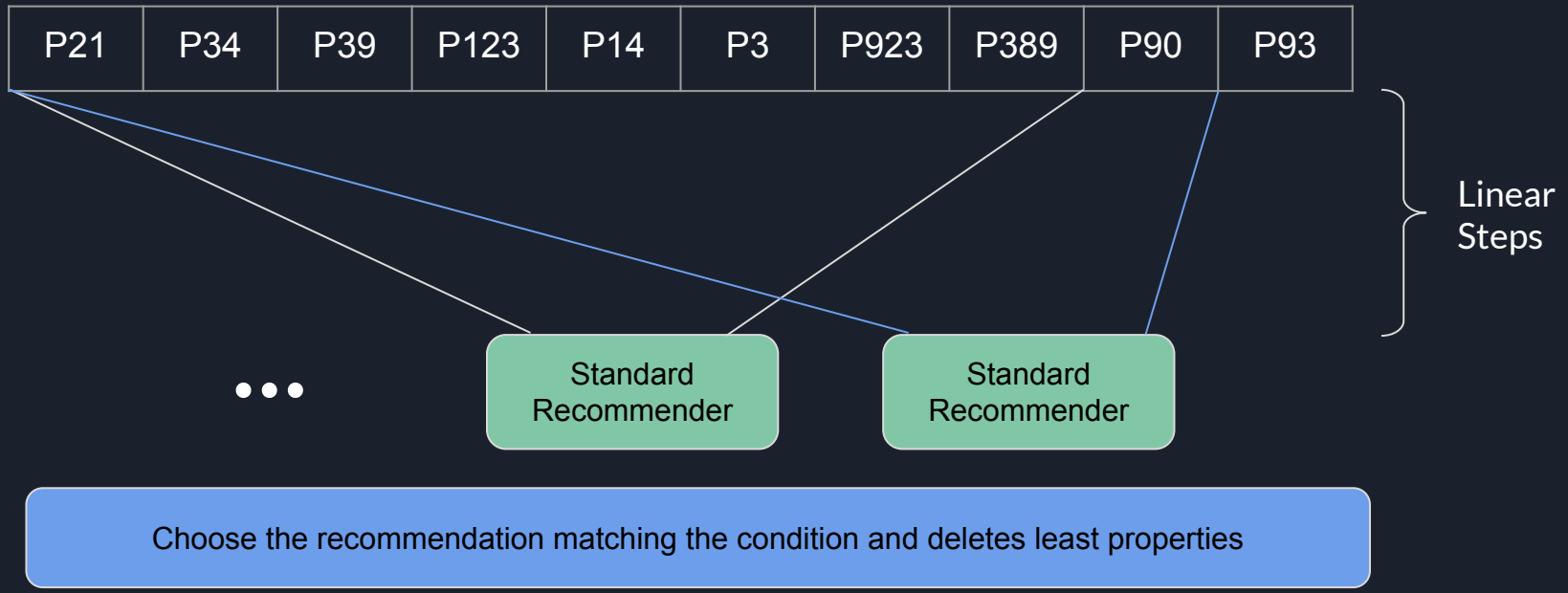
Assessments store data throughout the workflow

Re-use previous calculations

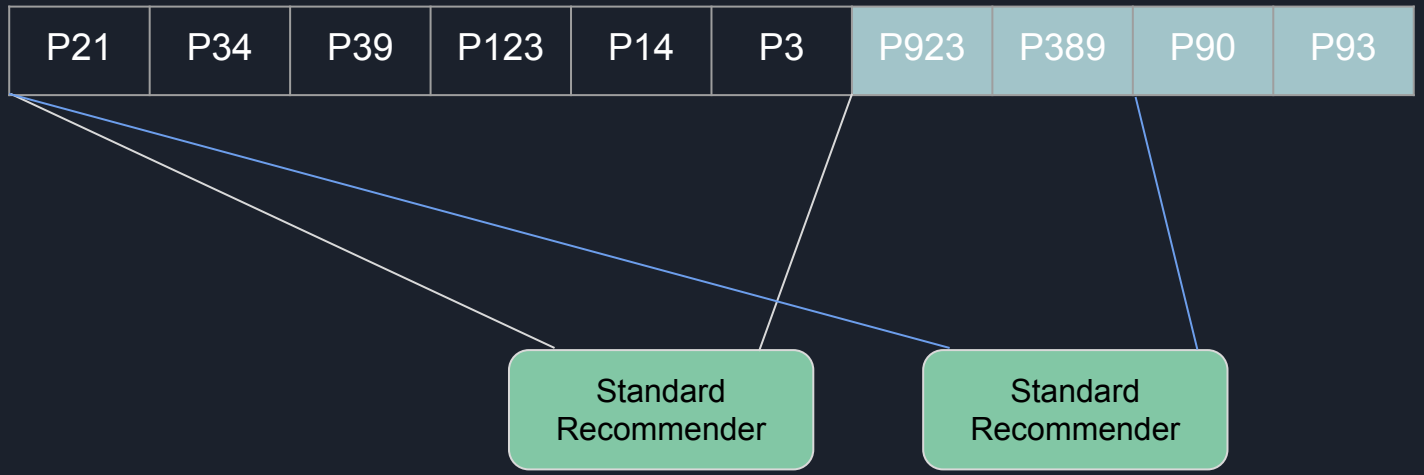
Integrating Backoff Strategies - Split Property Backoff



Integrating Backoff Strategies - Delete Low Frequency Items



Integrating Backoff Strategies - Delete Low Frequency Items



Glossary

- the recommender internally works with unique property identifiers, but the recommendations should be presented in an readable way..

=> the glossary maps property identifier to their human understandable representations

- each property has
  - a description (language dependent)
  - a label (language dependent)
  - an identifier

<local://prop/Has> <http://schema.org/name> "has"@en

<local://prop/Has> <http://schema.org/description> "contains in it"@en

---

glossary[“local://prop/Has”, “en”] = {Label = “has”, Description = “contains in it”}

HTTP Request: Input and Output Example

input

```
{
  "lang": "en",
  "properties": [
    "http://www.wikidata.org/prop/direct/P1476",
    "http://www.wikidata.org/prop/direct/P433"
  ],
  "types": [
    "http://www.wikidata.org/entity/Q13442814"
  ]
}
```

output

```
{
  "recommendations": [
    {
      "property": "http://www.wikidata.org/prop/direct/P828",
      "probability": 0.964267264,
      "label": "has cause",
      "description": "thing that resulted in this effect"
    },
    ...
  ]
}
```

Evaluation - Procedure

A = {p1, p2, p3}

- input: A' = {p1, p2} -> expecting p3 to be recommended
- recommender returns sorted list of recommendations (highest probability first)
- determine position (rank) of p3 in this list
- if a left out property is not recommended, then we assign rank 500 (recommender returns at most 500 recommendations) and marks it as “miss”

- analogue for all other possible subsets and entities in the dataset
- present results grouped by input set size
- calculate statistical classification criterias

Evaluation - Criterias

- Size of input set (number of properties per entity)
- Median
- Mean
- Variance
- **Percentage of hits in top 1**
- **Percentage of hits in top 5**
- **Percentage of hits in top 10**
- Average over worst 5 percent
- **Hit rate**
- Recommendation time
- Memory allocation
- Subject count (evaluated entities)
- Sample size (evaluated properties)

Evaluation - Evaluation Framework

- structure
- prepare
- recommend
- output
- evaluate

Generate Config Files:

1. Create a Creater Config File (see ./configs/generate_all.json and ./configs/README.md)
2. Build go build .
3. Run `./evaluation -createConfigs -creater currentCreater``

Run Batch Test:

Runs all the config files ./configs/config_i.json in in 1...n

1. go build .
2. Run `./evaluation -batchTest -numberConfigs n -testSet ../testdata/10M.nt_1in2_test.gz -model ../testdata/10M.nt_1in2_train.gz.schemaTree.bin`

Run Single Test:

Runs the standard recommender

1. go build .
2. Run `./evaluation -testSet ../testdata/10M.nt_1in2_test.gz -model ../testdata/10M.nt_1in2_train.gz.schemaTree.bin`

Run the recommender with types (note that the schematree needs to support type info then)

1. go build .
2. Run `./evaluation -model ../testdata/10M.nt_1in2_train.gz.schemaTree.typed.bin -testSet ../testdata/10M.nt_1in2_test.gz -typed`

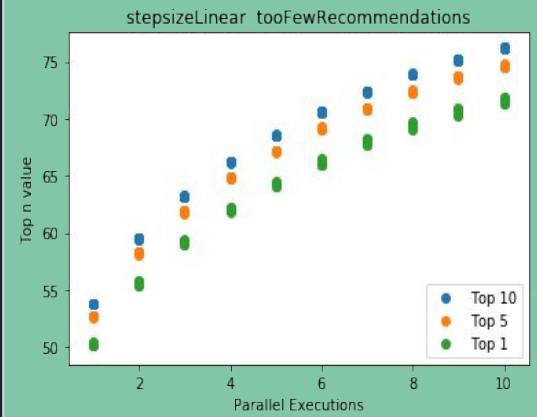
Run the recommender with types and workflow config (note that the schematree needs to support type info then)

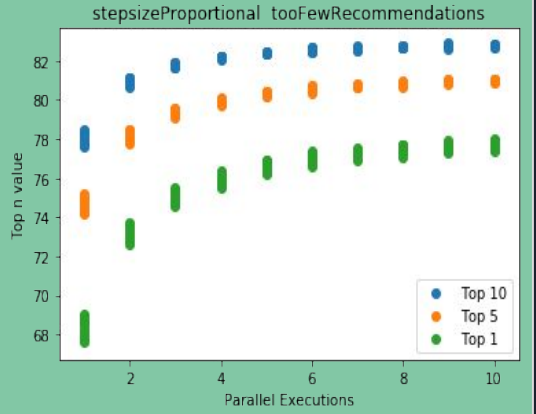
1. go build .
2. Run `./evaluation -model ../testdata/10M.nt_1in2_train.gz.schemaTree.typed.bin -testSet ../testdata/10M.nt_1in2_test.gz -typed -workflow ../testdata/workflow.json`

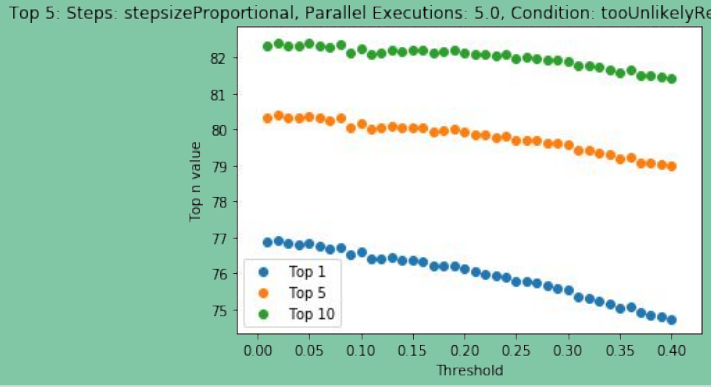
Task it performs

1. Run a Single test with workflow from config file
2. Generate a bunch of config files
3. Batch test and log the aggregated results in a csv file
4. Visualize via python jupyter notebook (not golang)

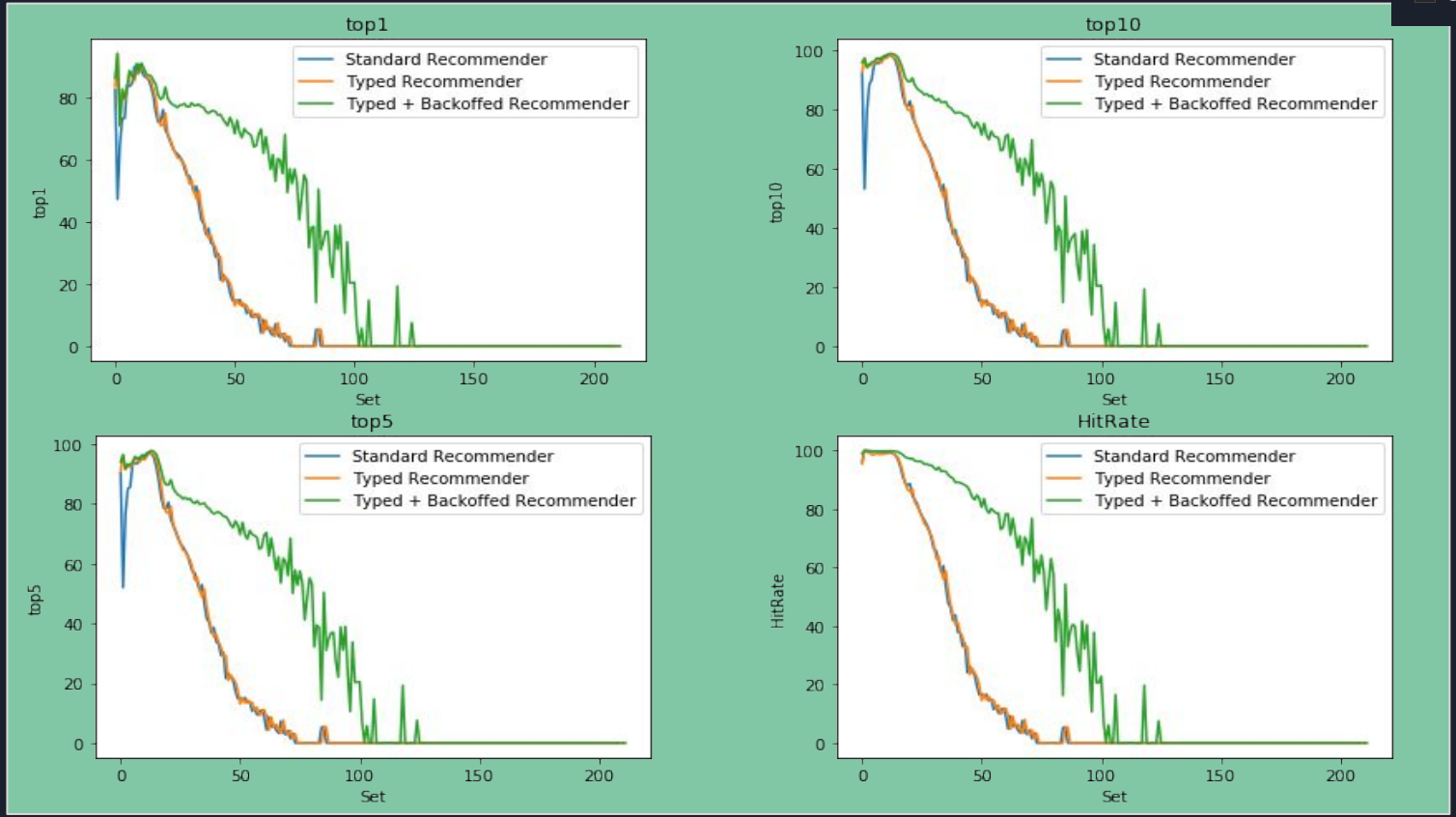
Evaluation - Evaluation of the Strategies



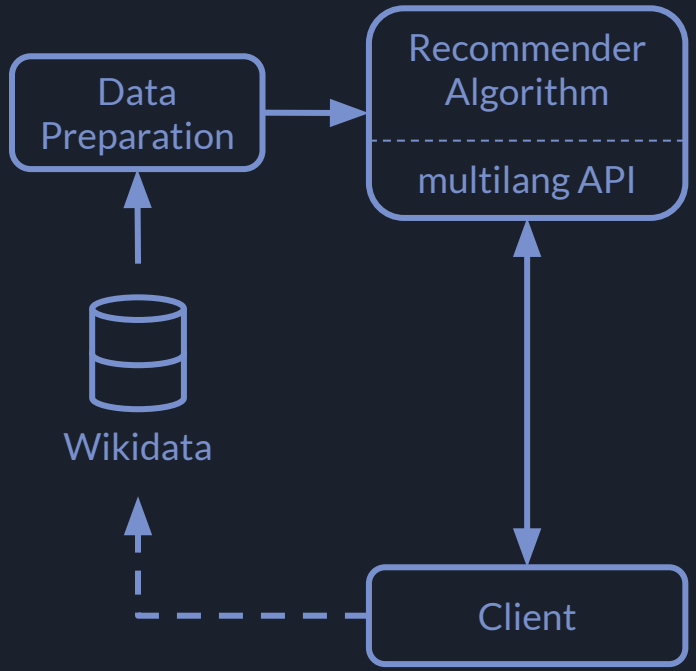




Evaluation - Benchmarking (Whole Set)



Actual System Diagram



↗ Improve

- ↗ type information
- ↗ improved evaluation
- ↗ data preparation pipeline
- ↗ automatic testing
- ↗ backoff strategies
- ↗ strategy tuning
- ↗ multilang labelling
- ↗ code architecture

⇄ Integrate

- ⇄ extend current API
- ⇄ application usability