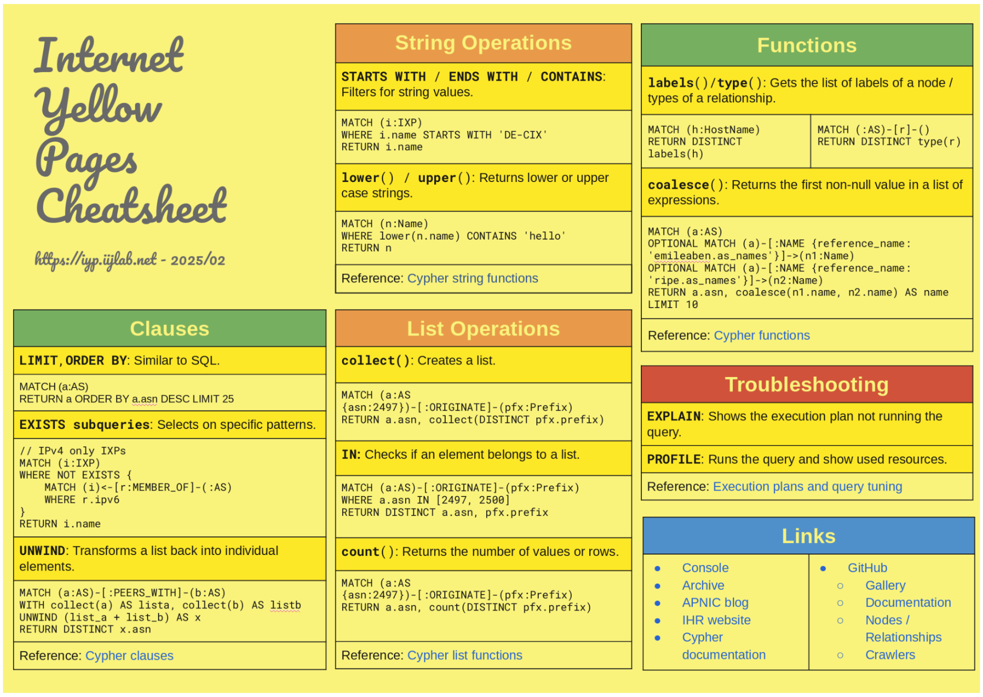
Internet Yellow Pages (IYP) Tutorial

Give your feedback! This tutorial is in a Google doc so that you can help us improve it. Don't hesitate to add comments to this document, later we'll integrate the changes and move the content to a public website.

This tutorial covers the following topics:

- **Get Started with IYP**: the basics of the Internet Yellow Pages and its querying language, Cypher.
- **Example IYP queries**: example queries to strengthen your understanding of IYP data and its querying language.
- **Accessing IYP from code**: interact with IYP from Python.
- **Hosting a local IYP instance**: run your own instance of IYP locally.
- **Adding data to IYP**: the steps to add your own data to IYP.

For writing queries see also the IYP cheatsheet:



Internet Yellow Pages Cheatsheet

https://iyp.iijlab.net - 2025/02

String Operations

`STARTS WITH / ENDS WITH / CONTAINS`: Filters for string values.

```
MATCH (i:IXP)
WHERE i.name STARTS WITH 'DE-CIX'
RETURN i.name
```

`lower()` / **`upper()`**: Returns lower or upper case strings.

```
MATCH (n:Name)
WHERE lower(n.name) CONTAINS 'hello'
RETURN n
```

Reference: Cypher string functions

Functions

`labels()`/`type()`: Gets the list of labels of a node / types of a relationship.

```
MATCH (h:HostName)
RETURN DISTINCT
labels(h)
```

```
MATCH (:AS)-[r]-()
RETURN DISTINCT type(r)
```

`coalesce()`: Returns the first non-null value in a list of expressions.

```
MATCH (a:AS)
OPTIONAL MATCH (a)-[:NAME {reference_name:
'emileaben.as_names'}]->(n1:Name)
OPTIONAL MATCH (a)-[:NAME {reference_name:
'ripe.as_names'}]->(n2:Name)
RETURN a.asn, coalesce(n1.name, n2.name) AS name
LIMIT 10
```

Reference: Cypher functions

Clauses

`LIMIT`, `ORDER BY`: Similar to SQL.

```
MATCH (a:AS)
RETURN a ORDER BY a.asn DESC LIMIT 25
```

`EXISTS subqueries`: Selects on specific patterns.

```
// IPv4 only IXPs
MATCH (i:IXP)
WHERE NOT EXISTS {
    MATCH (i)<-[r:MEMBER_OF]-(:AS)
    WHERE r.ipv6
}
RETURN i.name
```

`UNWIND`: Transforms a list back into individual elements.

```
MATCH (a:AS)-[:PEERS_WITH]-(b:AS)
WITH collect(a) AS lista, collect(b) AS listb
UNWIND (list_a + list_b) AS x
RETURN DISTINCT x.asn
```

Reference: Cypher clauses

List Operations

`collect()`: Creates a list.

```
MATCH (a:AS
{asn:2497})-[:ORIGINATE]-(pfx:Prefix)
RETURN a.asn, collect(DISTINCT pfx.prefix)
```

`IN`: Checks if an element belongs to a list.

```
MATCH (a:AS)-[:ORIGINATE]-(pfx:Prefix)
WHERE a.asn IN [2497, 2500]
RETURN DISTINCT a.asn, pfx.prefix
```

`count()`: Returns the number of values or rows.

```
MATCH (a:AS
{asn:2497})-[:ORIGINATE]-(pfx:Prefix)
RETURN a.asn, count(DISTINCT pfx.prefix)
```

Reference: Cypher list functions

Troubleshooting

`EXPLAIN`: Shows the execution plan not running the query.

`PROFILE`: Runs the query and show used resources.

Reference: Execution plans and query tuning

Links

- Console
- Archive
- APNIC blog
- IHR website
- Cypher documentation

- GitHub
  - Gallery
  - Documentation
  - Nodes / Relationships
  - Crawlers

Disclaimer

- Data quality: IYP makes no changes to imported datasets. **Users should be aware of the original datasets' limitations** to accurately interpret results and maximize the utility of IYP.
- Feedback: Report erroneous data found in IYP directly to the data providers, so that the original dataset can be fixed, the changes will be reflected in following IYP snapshots.
- Citation: If you publish work/tools based on IYP cite the original dataset used (and the IYP paper!).

- Heavy duty: For large scale analysis installing a local instance of IYP is recommended.
- Temporal analysis: IYP is not suitable for timely analysis (e.g. the infamous event-triggered-ark-measurements hackathon project) and longitudinal analysis.

Get Started with IYP

What is IYP?

The Internet Yellow Pages (IYP) is a graph database composed of over 50 Internet measurement datasets. IYP has been designed to facilitate the exploration and analysis of these datasets. An overview can be found in the paper "The Wisdom of the Measurement Crowd: Building the Internet Yellow Pages a Knowledge Graph for the Internet."
IYP is built on Neo4j, a graph database management system, which means that data is stored and queried through graph structures. Neo4j's query language is called Cypher and all programmatic interaction with IYP will require basic understanding of Cypher. In this tutorial we provide a Cypher overview and example queries for analyzing IYP data.

Overview of IYP data

Given the diversity of datasets integrated into IYP, one of the first difficulties when working with IYP is to get an overview of the data that is available. Then the main difficulty is to write Cypher queries. We briefly address these with the three following sections:

- Internet Health Report (IHR) provides a high level graphical interface to the type of queries and datasets that IYP supports.
- IYP data modeling provides an overview of the underlying data model.
- Cypher: querying IYP provides an overview of Cypher for IYP.

Internet Health Report

The easiest way to browse the IYP database is to visit the Internet Health Report website. There you can search for an Internet resource (e.g. AS, prefix, domain name) and get IYP data related to that resource.

1. Enter the resource into the **search field** in the top left corner.
2. Selecting the **Routing**, **DNS**, **Peering**, **Registration**, or **Rankings** tabs.
    a. The time series **Monitor** tab is external to the IYP.

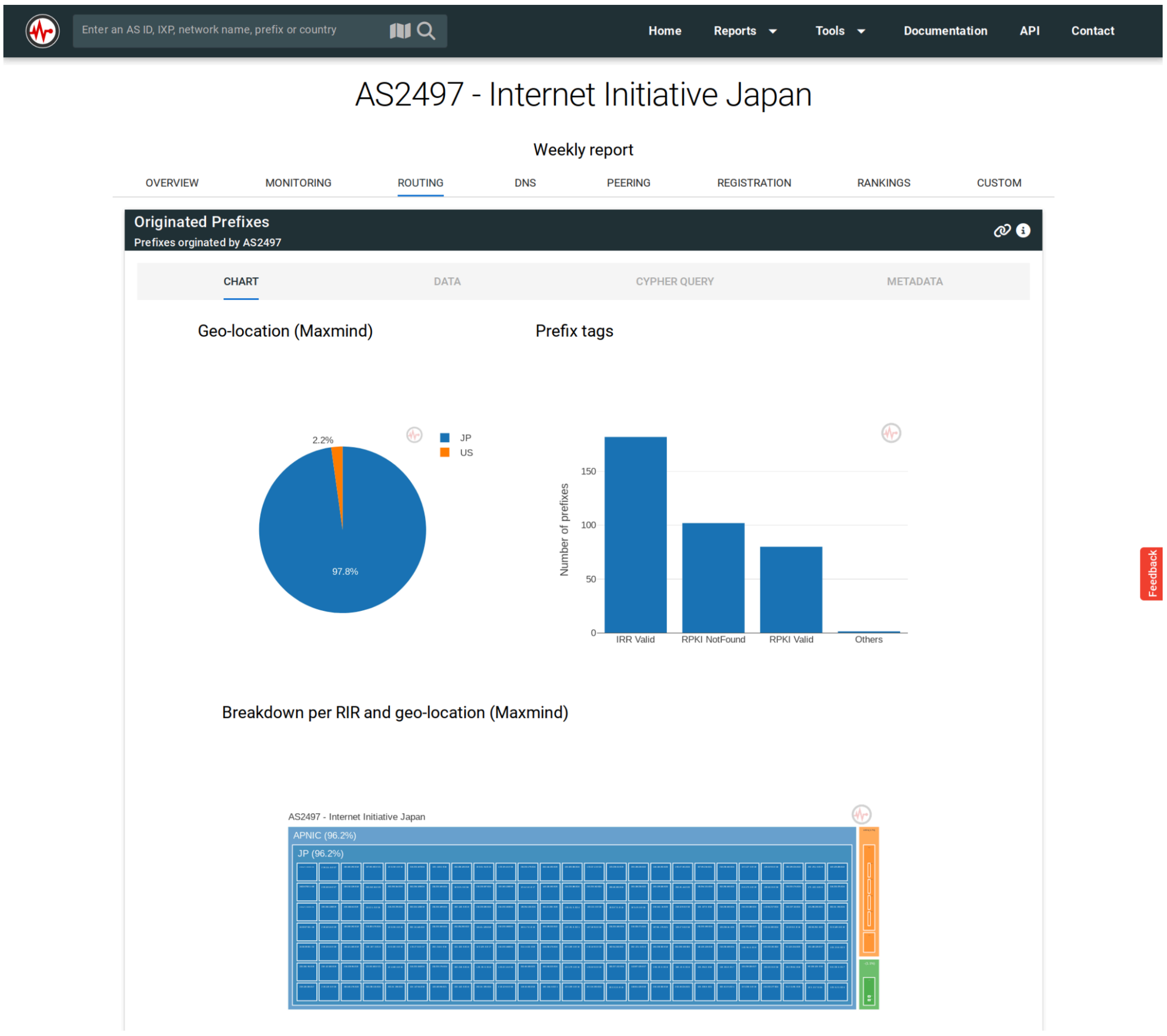


Figure: IYP routing data shown on IHR website (https://www.ihr.live/en/network/AS2497?active=routing).

The above figure illustrates the ‘routing’ tab for the Internet Initiative Japan network (AS2497). All other tabs (except the ‘monitoring’ one) are providing IYP data via different widgets.

For each widget, you’ll find a *chart, data, Cypher query*, and a *metadata* tabs.

- The Chart tab shows a visual representation of the data.
- The Data tab gives the raw data in a table format.
- **And the Cypher Query tab gives you the exact query we used to pull the data from IYP. You can reuse that to query directly the IYP database or craft your own queries.** More on how to write your own query below.
- The Metadata tab gives links to the original datasets and the freshness of the data.

IYP data modelling

Under the hood IYP is stored in a graph database where nodes represent mostly Internet resources and links give relationships between them. To do that we had to model all datasets integrated to IYP as graphs. For some datasets it makes a lot of sense, for some it may seem counter-intuitive. In any cases IYP's github repository (the iyp/crawler folder) contains a readme for each dataset where the modelling is documented.

Let's see a simple example. The below figure is an extract of BGP data showing the two ASes connected to the University of Tokyo (AS2501):

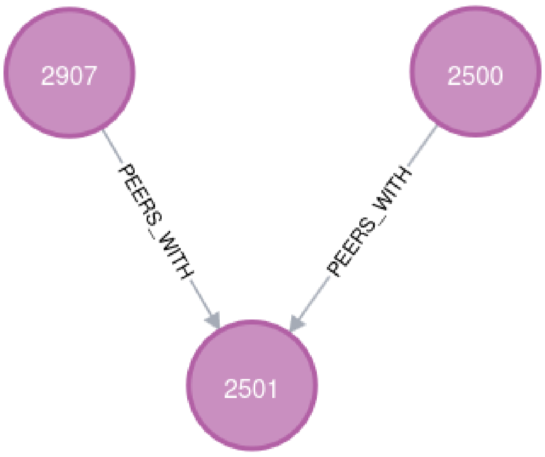


Figure: Networks peering with AS2501.

This is very similar to AS graphs we usually draw on a whiteboard. In addition the number displayed in each node shows the ASN which is actually a property set on the node.

For a counter-intuitive example see the graph below. These are the names associated with AS2501:

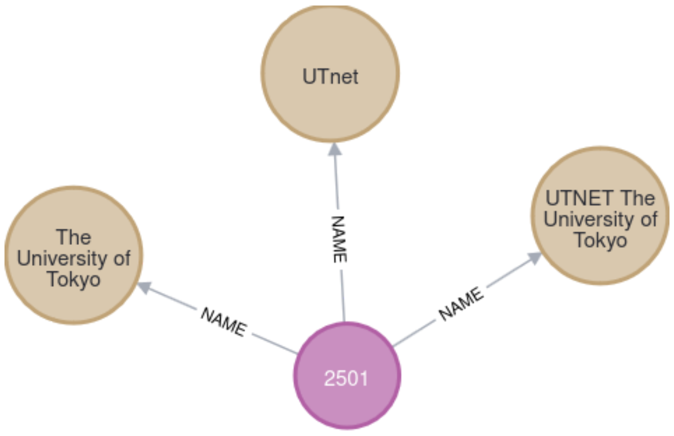


Figure: Names for AS2501.

Although we could have set a property 'name' to our AS nodes, we wouldn't know which value to use for this property. Different datasets give us different AS names. So instead of a property, a name is a unique entity that can be linked to other entities.

The previous example also illustrates two different types of nodes, one `AS` node and three `Name` nodes. In IYP every node and every relationship are typed. The list of node and relationship types is available on GitHub and grows as IYP integrates more datasets.

Nodes and links have properties. For example the `asn` property permits to uniquely identify the AS represented by an AS node. In the IYP console (introduced below) click on a link or node to see all its properties.
IYP uses properties for three different purposes:

- **Identification**: Some properties are immutable values that enable the user to select a specific thing in the database. For example, all AS nodes have the property `asn` that permits distinguishing the different ASes. Country nodes have the property `country_code`, HostName nodes have the property `name`, etc. These properties are documented in the list of node and relationship types.
- **Non-modelled data**: Some datasets provide a lot of detailed information, more than what IYP is modelling. In this case IYP keeps non-modelled data in the form of properties. For example, PeeringDB provides IXP member lists. This is modelled by connecting AS nodes to IXP nodes with the `MEMBER_OF` relationship. PeeringDB also provides the general peering policy of these ASes, which is not modelled in IYP but available via the property `policy_general` of the relationships. These properties are generally not documented in IYP and require a good understanding of the original dataset.
- **Reference**: Every relationship in IYP contains metadata referring to the origin of the data:
    - The `reference_org` property refers to the organization providing the original data.
    - `reference_time_fetch` is the date at which the data was imported.
    - `reference_time_modification` is the time at which the data was produced.
    - `reference_url_data` is the URL of the original dataset.
    - `reference_url_info` is the URL of the documentation of the original dataset.
    - `reference_name` is unique per dataset and is composed of the organization name and dataset name (e.g. `caida.as_relationships_v4`). This is particularly useful to filter per datasets.

Documentation

The IYP documentation contains complementary information that is essential for writing queries. There you will find:

- the different types of nodes and relationships available,
- the list of all integrated datasets,
- how each dataset is modelled (see README.md files in these folders)
- see also the IYP cheatsheet

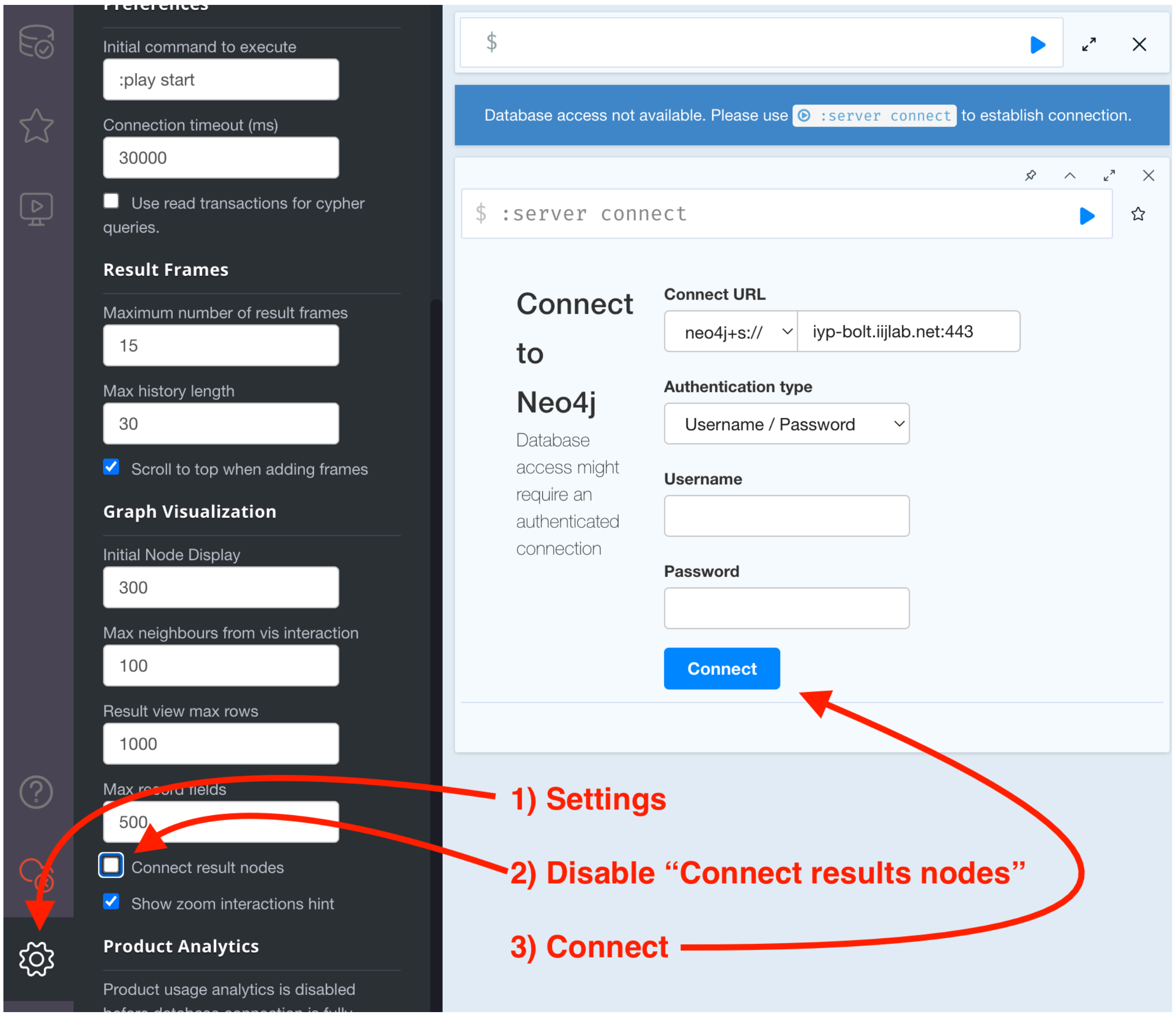
Cypher: Querying IYP

The main advantage of IYP is the possibility to query all available datasets at once. The learning curve is quite steep, though. It requires you to be familiar with how datasets are modelled in IYP and the Cypher querying language. We'll cover the basics here.

IYP console

The easiest way to execute IYP queries is to use IYP's public instance:

1. Go to the IYP console.
2. Click on the ‘Connect’ button (no username or password required).
3. And before you continue we recommend you to turn off the `Connect result nodes` option in the settings available from the cogwheel at bottom left.


Preferences

Initial command to execute: :play start

Connection timeout (ms): 30000

☐ Use read transactions for cypher queries.

Result Frames

Maximum number of result frames: 15

Max history length: 30

✓ Scroll to top when adding frames

Graph Visualization

Initial Node Display: 300

Max neighbours from vis interaction: 100

Result view max rows: 1000

Max record fields: 500

☐ Connect result nodes

✓ Show zoom interactions hint

Product Analytics

Product usage analytics is disabled before database connection is fully

Database access not available. Please use `:server connect` to establish connection.

```
$ :server connect
```

Connect to Neo4j

Database access might require an authenticated connection

Connect URL: neo4j+s:// iyp-bolt.iijlab.net:443

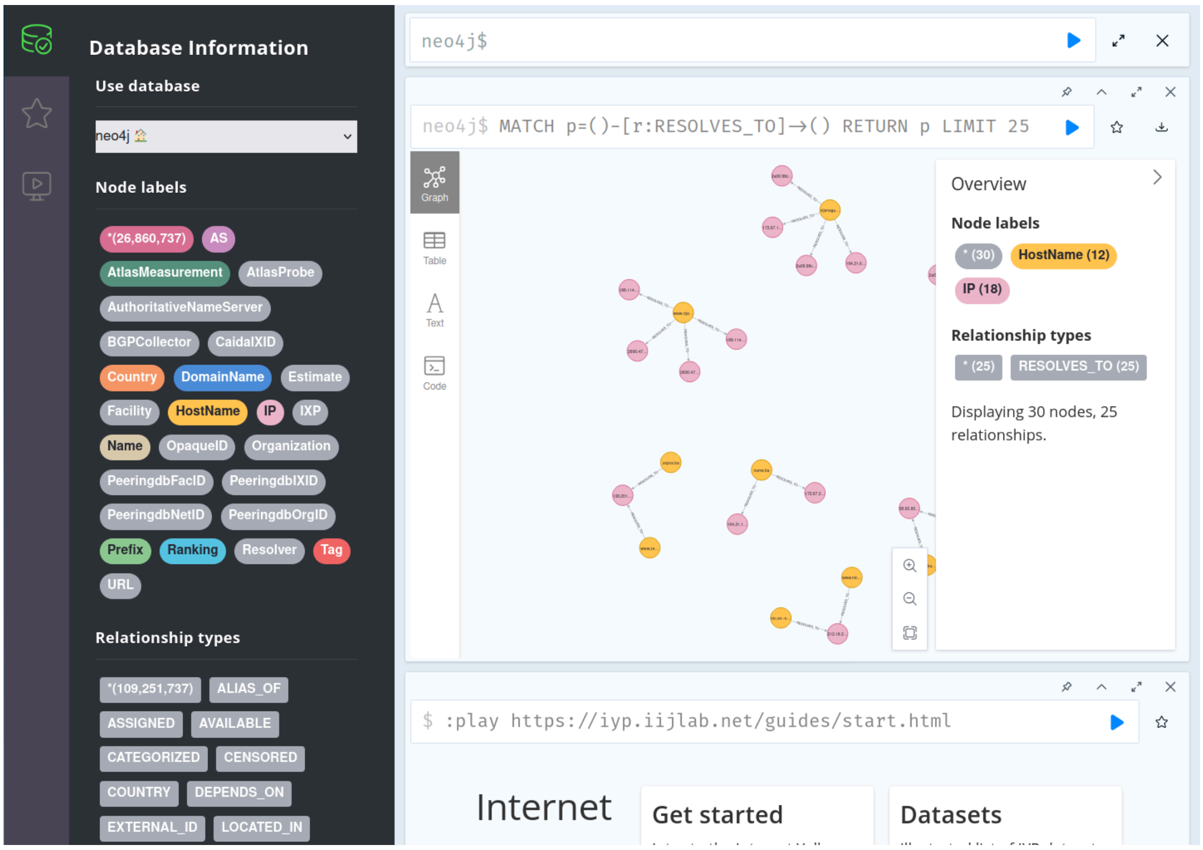
Authentication type: Username / Password

Username:

Password:

Figure: There is no username/password required to connect to the IYP console. Make sure you uncheck the `Connect result nodes` option in the settings.

The IYP console provides a summary of the database which is quite handy when writing queries. At the top left there is a database icon that shows all types of nodes and relationships available in the database. Clicking on a node or relationship type displays examples.



```
neo4j$ MATCH p=()-[r:RESOLVES_TO]→() RETURN p LIMIT 25
```

```
$ :play https://iyp.iijlab.net/guides/start.html
```

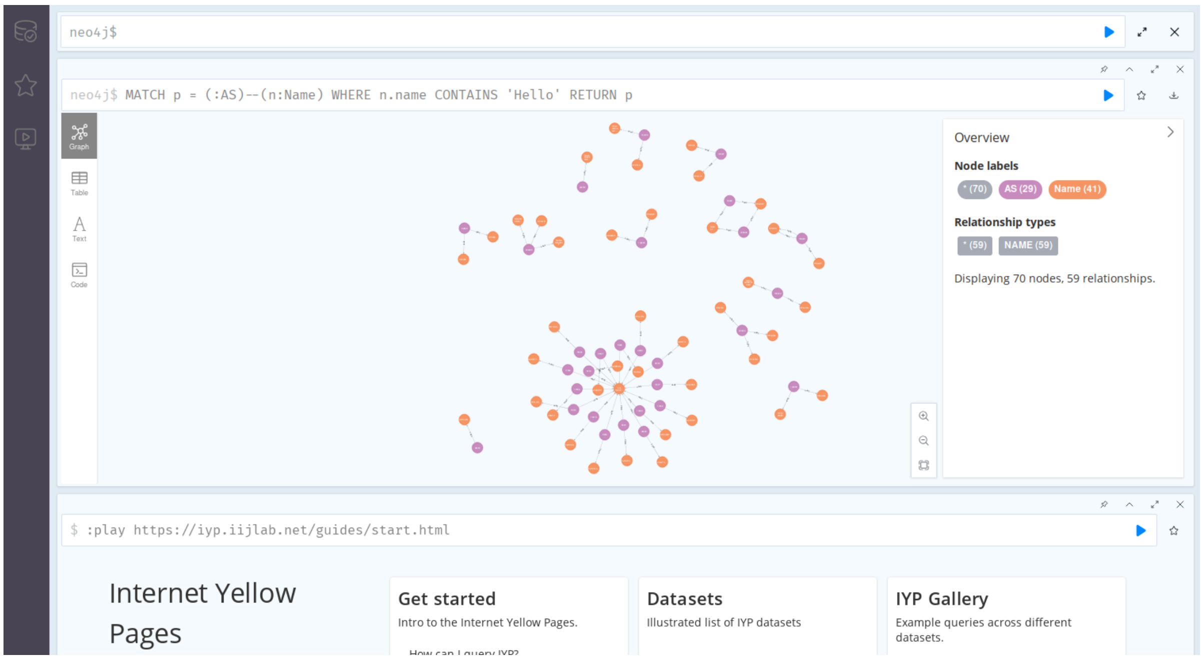
Figure: The top left tab in the IYP console displays all types of nodes and relationships available in the database. Click on one of those to see examples.

Hello world

You are ready for your first query. Copy/paste the below query in the top input box (next to `neo4j$`) and click the play button.

```cypher
MATCH p = (:AS)--(n:Name) WHERE n.name CONTAINS 'Hello' RETURN p
```

You should see something similar to this (click on the node labels at the top right corner to change the node's color and select the property which should be used as the caption of the node):



```
neo4j$
neo4j$ MATCH p = (:AS)--(n:Name) WHERE n.name CONTAINS 'Hello' RETURN p
```

Overview

Node labels

* (70) AS (29) Name (41)

Relationship types

* (59) NAME (59)

Displaying 70 nodes, 59 relationships.

```
$ :play https://iyp.iijlab.net/guides/start.html
```

Internet Yellow Pages

Get started
Intro to the Internet Yellow Pages.
How can I query IYP?

Datasets
Illustrated list of IYP datasets

IYP Gallery
Example queries across different datasets.

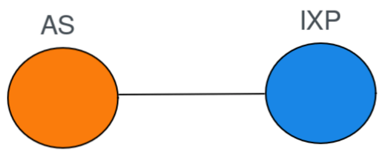
Figure: Results for the Hello World query.

This graph shows ASes that contain the word ‘Hello’ in their name. If you see a lot more links that means you haven’t un-checked the `Connect result nodes` option in the settings (please uncheck that option!).

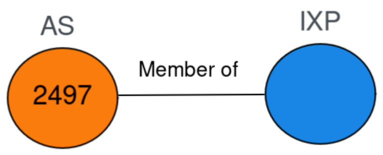
Cypher 101

Cypher is the main language used to query IYP. It has some similarities with SQL but it is designed to query graph databases. That means instead of looking for rows in tables, a Cypher query describes patterns to find in a graph.

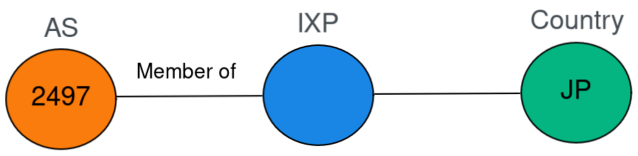
The pattern is given in an ASCII art representation where nodes are depicted by a pair of parentheses, `( )`, and relationships are depicted with two dashes `--` sometimes including more information in square brackets `-[]-`.



(:AS) -- (:IXP)



(:AS {asn: 2497}) -[:MEMBER_OF]- (:IXP)



(:AS {asn:2497})-[:MEMBER_OF]-(:IXP) -- (:Country {country_code: 'JP'})

Figure: Cypher's ASCII representation of graphs.

The simplest pattern we can look for is a node. The below query finds the AS node with ASN 2497 (try it in the IYP console!):

```cypher
MATCH (iij:AS)
WHERE iij.asn = 2497
RETURN iij
```

Now let's see how the query works:

- The `MATCH` clause defines the pattern to find in the graph.
- `(iij:AS)` is the pattern we are looking for. The parenthesis show that it is a node, `iij` is an arbitrary variable to refer to that node later in the query, and the type of node is given after the colon.
- The WHERE clause describes conditions for nodes or relationships that match the pattern. Here we specify that the node called iij should have a property `asn` that equals 2497.
- The `RETURN` clause describes the data we want to extract from the found patterns. Here we return `iij` the node that satisfies both the `MATCH` and `WHERE` clauses.

Another way to specify the condition for the node property is to set it within the search pattern. For example the following query returns exactly the same results as the above one:

```cypher
MATCH (iij:AS {asn: 2497})
RETURN iij
```
This is a more compact form, but really it doesn’t make a difference for the final result.

Slightly more complicated, the below query finds which IXPs AS2497 is a member of:

```cypher
MATCH p = (iij:AS)-[:MEMBER_OF]-(:IXP)
WHERE iij.asn = 2497
RETURN p
```

Thus (iij:AS)-[:MEMBER_OF]-(:IXP) describes a path that starts from a node we call iij that connects to another node typed `IXP`.
Similar to the node type, the type of a relationship is given after the colon, for example `-[:MEMBER_OF]-` is a relationship of type “member of”.

This query is also using a handy trick. Instead of assigning a variable for every node and relationship in the query, it uses one variable `p` that contains the whole pattern and specifies only `p` in the RETURN clause.

If needed we can assign variables to relationships and filter on their properties. For example, this query finds which IXPs AS2497 is a member of but not from PeeringDB data (the operator for inequality is `<>`):

```cypher
MATCH p = (iij:AS)-[mem:MEMBER_OF]-(:IXP)
WHERE iij.asn = 2497 AND mem.reference_org <> 'PeeringDB'
RETURN p
```

More Cypher

We don’t have the intention to cover the whole Cypher language in this tutorial. Cypher contains all operators you may expect from a modern querying language, including aggregating functions, OPTIONAL MATCH for patterns with optional parts, and many other clauses.

The Cypher tutorial and Cypher documentation are the more comprehensive places you should refer to when crafting your queries.

Also, in the IYP console the `:help` command provides documentation to any Cypher clause. Try `:help MATCH` or `:help cypher` in the IYP console.

More Cypher Hints

- Double click on a node in the UI to see its neighbors and links. The number of displayed nodes is limited, you can increase this limit in the settings (bottom left cog wheel).
- Add `LIMIT 10` at the end of your queries when experimenting.
- Add comments in your queries: Single line comments starting with `//` or multiple line comments using `/*` `*/`.

Example IYP queries

Writing IYP queries from scratch is daunting. But once you can read queries, you can then easily modify existing queries. In the following, we conduct small studies using IYP and provide all corresponding queries. This should give you enough material to start writing your own queries. Execute these examples in the IYP console, then try tuning or combining some of the queries. Also remember to click on the database icon (top left corner in the IYP console) to see all node and relationship types and get examples by clicking on any of those.

AS and IP Prefixes

First, we will learn about queries for finding specific prefixes and ASes. We also use these examples to explain how to use the IYP console interface and provide some tricks for making your own queries.

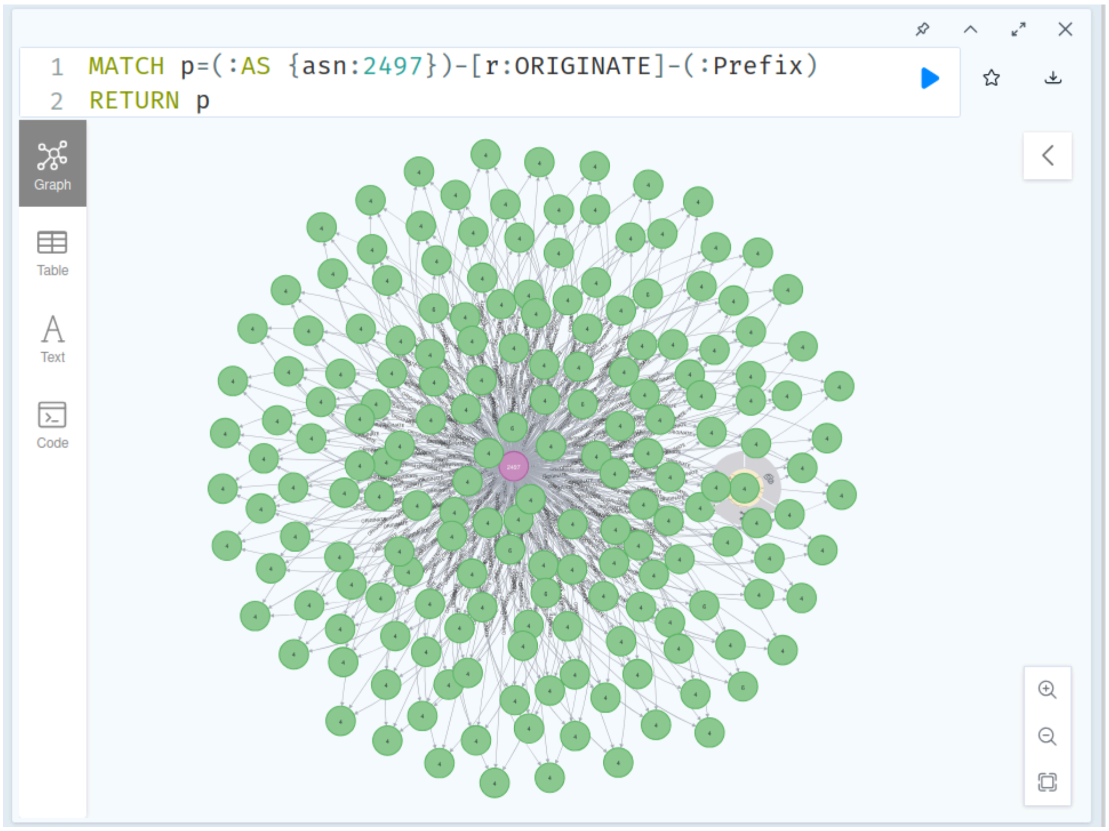
Find prefixes originated by an AS

We start by looking at the prefixes originated by a certain AS which is represented in IYP by the `ORIGINATE` relationship between `AS` and `Prefix` nodes.

Here is the query to find prefixes originated by AS2497:

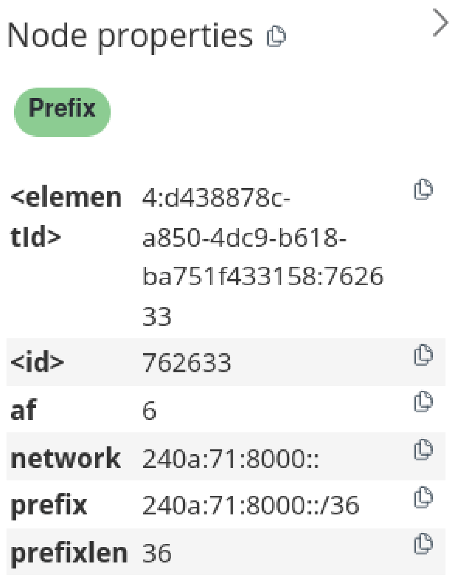
```cypher
MATCH p = (:AS {asn:2497})-[r:ORIGINATE]-(:Prefix)
RETURN p
```

Copy/paste this query into the IYP Console. You should obtain a busy graph like this:



```
MATCH p=(:AS {asn:2497})-[r:ORIGINATE]-(:Prefix)
RETURN p
```

Now let's try another query focusing only on the IPv6 prefixes. By clicking on any of the Prefix nodes you will see in the right side panel the properties of the node (if you don't see the panel then click the '<' icon), something like this:



Node properties

Prefix

<elementId>	4:d438878c-a850-4dc9-b618-ba751f433158:762633
<id>	762633
af	6
network	240a:71:8000::
prefix	240a:71:8000::/36
prefixlen	36

The `af` property (i.e. Address Family) tells us if the prefix is for IPv4 or IPv6. So to find all IPv6 prefixes originated by AS2497 we can filter prefixes using the `af` property:

```cypher
MATCH p = (:AS {asn:2497})-[r:ORIGINATE]-(:Prefix {af: 6})
RETURN p
```

`` ``` ``

Copy/paste this query into the IYP Console, you should obtain another cute graph. You may wonder why there are multiple links between the same pair of nodes. This is because multiple datasets provide us with this same information. Clicking on the links you can see that the `reference_org` property is different. Some are from BGPKIT, some are from Packet Clearing House and some are from IHR. IYP gives you the possibility to filter per dataset. If you want to query only data from BGPKIT, you can filter on this property (or even better on the `reference_name` property which is a unique name for the dataset):

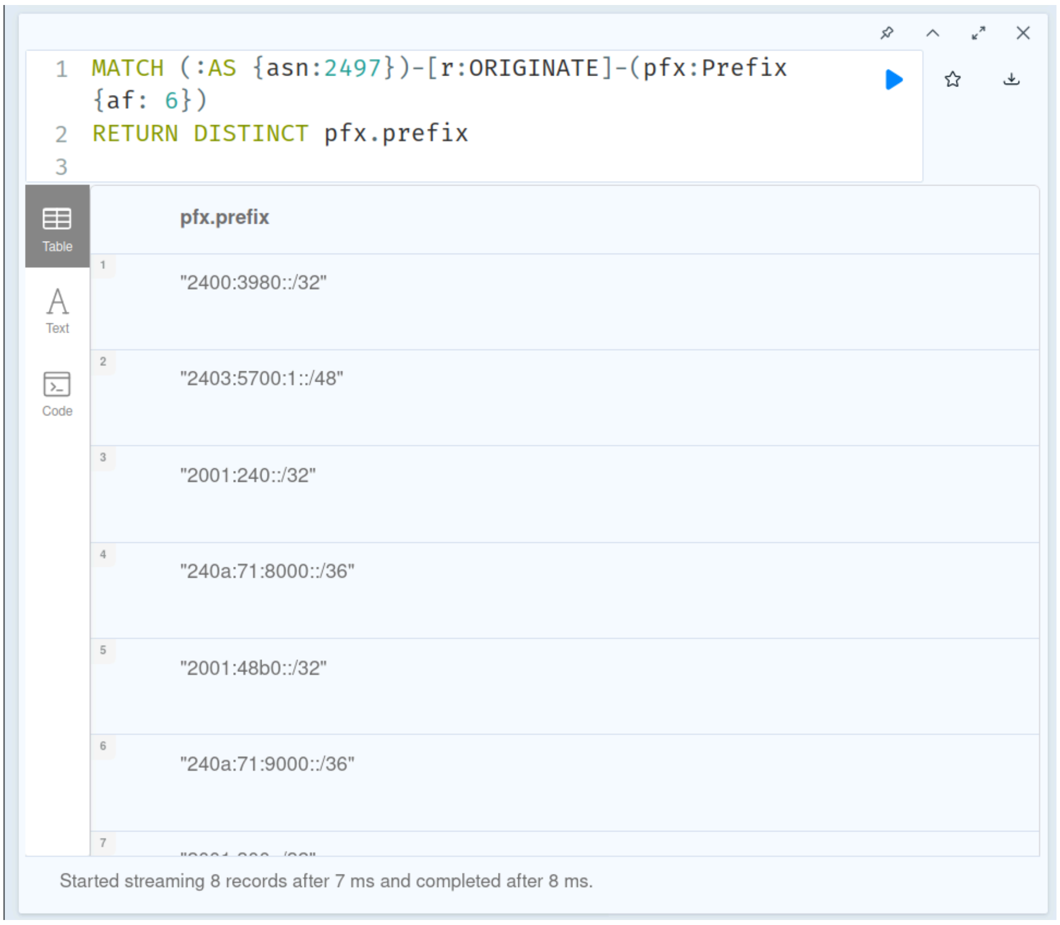
```cypher
MATCH p = (:AS {asn:2497})-[r:ORIGINATE]-(:Prefix {af: 6})
WHERE r.reference_name = 'bgpkit.pfx2asn'
RETURN p
```

For analysis we need the actual list of prefixes (not a cute graph). To do that we can ask Cypher to return property values instead of nodes and relationships. The following query returns a list of IPv6 prefixes originated by AS2497:

```cypher
MATCH (:AS {asn:2497})-[r:ORIGINATE]-(pfx:Prefix {af: 6})
RETURN DISTINCT pfx.prefix
```

Note the use of the keyword `DISTINCT` in the RETURN statement, this ensures that we retrieve only unique rows. Since we have multiple links that match this pattern the query would have returned multiple times the same prefix (try the query without `DISTINCT`).

Executing the above query you should see a table listing all prefixes:


```
MATCH (:AS {asn:2497})-[r:ORIGINATE]-(pfx:Prefix
{af: 6})
RETURN DISTINCT pfx.prefix
```

	pfx.prefix
1	"2400:3980::/32"
2	"2403:5700:1::/48"
3	"2001:240::/32"
4	"240a:71:8000::/36"
5	"2001:48b0::/32"
6	"240a:71:9000::/36"
7	

Started streaming 8 records after 7 ms and completed after 8 ms.

You can download the data in CSV or JSON format via the download icon at the top right corner.

Finally, we can also search for more complex patterns in the graph. The following query looks for prefixes that are originated by two different origin ASes. The return values are the prefix, the two origin ASes, and the `count` values provided by BGPKIT (the number of RIS and RouteViews peers that see the prefix/origin pair).

```cypher
MATCH (a:AS)-[ra:ORIGINATE {reference_org:
'BGPKIT'}]-(pfx:Prefix)-[rb:ORIGINATE {reference_org:
'BGPKIT'}]-(b:AS)
WHERE a <> b
RETURN DISTINCT pfx.prefix, a.asn, b.asn, ra.count, rb.count
LIMIT 100
```

As we write more complex Cypher queries the searched pattern may become very long and hard to read. In this case we can also use multiple `MATCH` clauses. The following query gives the exact same results as the previous one:

```cypher
MATCH (a:AS)-[ra:ORIGINATE {reference_org: 'BGPKIT'}]-(pfx:Prefix)
MATCH (pfx)-[rb:ORIGINATE {reference_org: 'BGPKIT'}]-(b:AS)
WHERE a<>b
RETURN DISTINCT pfx.prefix, a.asn, b.asn, ra.count, rb.count
LIMIT 100
```

Exercises

1. Write a query that fetches only IPv4 prefixes.
2. Write a query that fetches only /24 prefixes.
3. `ORIGINATE` is not the only type of relationship between ASes and Prefixes. For RPKI we have the ‘ROUTE_ORIGIN_AUTHORIZATION’ relationship between AS and Prefix nodes.
   Find prefixes that are announced by one AS and that have a ROA for another AS.

IP addresses and HostNames

Now we’ll see how to query more complex patterns and introduce other types of nodes and relationships: `IP`, `HostName`, `PART_OF`, `RESOLVES_TO`. We’ll also learn about the Cypher Aggregating function `collect()`

Finding popular IPs in a prefix

Some of the datasets integrated into IYP provide IP addresses and hostnames. A good example of that are the top popular websites and DNS nameservers provided by Tranco and OpenINTEL.

The query to fetch any hostnames (from any of the integrated dataset) hosted by AS2497 is:

```cypher
MATCH (:AS {asn:2497})-[:ORIGINATE]-(pfx:Prefix)-[:PART_OF]-(:IP)-[:RESOLVES_TO]-(h:HostName)
RETURN pfx.prefix, collect(DISTINCT h.name)
```

Note the usage of `collect` in the `RETURN` clause. This function is used to compile a list of all `HostName` names per prefix. If you use aggregation functions in the return clause, it implies (in SQL terms) a “GROUP BY” on all returned elements that are not aggregated (like pfx.prefix in this example).

However, the above query is returning only prefixes that are related to hostnames. It won’t return an empty hostname list. To list all prefixes and their corresponding hostnames (if they have any) we should break down the previous query into two parts and make one of the parts optional. Optional parts of a pattern are preceded by the keyword `OPTIONAL`, hence the previous query becomes:

```cypher
MATCH (:AS {asn:2497})-[:ORIGINATE]-(pfx:Prefix)
OPTIONAL MATCH (pfx)-[:PART_OF]-(:IP)-[:RESOLVES_TO]-(h:HostName)
RETURN pfx.prefix, collect(DISTINCT h.name)
```

Finding DNS authoritative nameservers and corresponding domains

Looking at the results of the above query you may see a lot of hostnames that start with 'ns'. Those are typically DNS nameservers. In IYP a node can have multiple types. The DNS nameservers are both `HostName` and `AuthoritativeNameServer`. Hence, the following query finds all authoritative nameservers hosted by AS2497 and the number of domains they manage.

```cypher
MATCH (:AS {asn:2497})-[:ORIGINATE]-(pfx:Prefix)
MATCH
(pfx)-[:PART_OF]-(:IP)-[:RESOLVES_TO]-(ns:AuthoritativeNameServer)
OPTIONAL MATCH (dn:DomainName)-[:MANAGED_BY]-(ns)
RETURN ns.name, count(DISTINCT dn.name) AS nb_domains,
collect(DISTINCT dn.name)
ORDER BY nb_domains DESC
```

Note the use of:

- the `count` function (similar to SQL) to count the number of domain names,
- the `AS` keyword to name a result column,
- the `ORDER BY` and `DESC` keywords to sort the results (similar to SQL).

More examples

IYP integrates a lot of different datasets, more that we can cover in this tutorial. To end this part we provide a list of small queries for diverse datasets available in IYP to help you start writing your own queries.

IXPs and their colocation facilities:
```cypher
MATCH p=(:IXP)-[r:LOCATED_IN]-(:Facility)-[:COUNTRY]-(:Country)
RETURN p LIMIT 25
```

Peering LANs of IXPs:
```cypher
MATCH p = (pfx:Prefix)-[:MANAGED_BY]-(:IXP)
```

```
RETURN p LIMIT 50
```

The **“best” name** for AS2497:
```cypher
MATCH (a:AS {asn:2497})
OPTIONAL MATCH (a)-[:NAME {reference_org: 'PeeringDB'}]->(n1:Name)
OPTIONAL MATCH (a)-[:NAME {reference_org: 'BGP.Tools'}]->(n2:Name)
OPTIONAL MATCH (a)-[:NAME {reference_org: 'RIPE NCC'}]->(n3:Name)
RETURN a.asn, coalesce(n1.name, n2.name, n3.name) AS name
```

RPKI ROAs for prefixes not seen in BGP:
```cypher
MATCH (roa_as:AS)-[:ROUTE_ORIGIN_AUTHORIZATION]-(pfx:Prefix)
WHERE NOT (pfx)-[:ORIGINATE]-(:AS)
RETURN pfx.prefix, roa_as.asn
```

RPKI invalid prefixes (all possible types: RPKI Valid / RPKI Invalid / RPKI NotFound):
```cypher
MATCH (pfx:Prefix)-[:CATEGORIZED]-(t:Tag)
WHERE t.label = "RPKI Invalid"
RETURN pfx.prefix
```

All the **parent domain names** of 'server.transfer.us-west-1.amazonaws.com':
```cypher
MATCH p=(:DomainName {name:
'server.transfer.us-west-1.amazonaws.com'})-[:PARENT*]->()
RETURN p
```

Top 1k domain names in **Tranco**:
```cypher
MATCH (dn:DomainName)-[r:RANK]-(:Ranking)
WHERE r.reference_name = 'tranco.top1m' AND r.rank < 1000
RETURN dn.name
```

Top 1k website from **CrUX** for France and the corresponding hosting ASes:
```cypher
MATCH (h:HostName)-[r:RANK]-(:Ranking)-[:COUNTRY]-(c:Country)
WHERE r.rank <= 1000
  AND r.reference_name = 'google.crux_top1m_country'
  AND c.country_code = 'FR'
```

```
MATCH (h)-[:RESOLVES_TO {reference_org:'OpenINTEL'}]-
  (:IP)-[:PART_OF]-(:Prefix)-[:ORIGINATE]-(net:AS)
RETURN h.name, COLLECT(DISTINCT net.asn)
```

Resources allocated to the same opaque ID (from **RIR's delegated stat files**) as AS15169 (Google):

```cypher
MATCH p = (:AS {asn:15169})-[:ASSIGNED]-(OpaqueID)-[:ASSIGNED]-()
RETURN p
```

All **RIS** peers providing more than 800k IPv4 prefixes (change to `route-views` to see **RouteViews'** peers):

```cypher
MATCH p=(rc:BGPCollector)-[peer:PEERS_WITH]-(:AS)
WHERE peer.num_v4_pfxs > 800000 and rc.project = 'riperis'
RETURN p
```

All **RIPE Atlas** measurements towards 'google.com' and participating probes:

```cypher
MATCH msm_target = (msm:AtlasMeasurement)-[r:TARGET]-(:HostName
{name:'google.com'})
OPTIONAL MATCH probes = (:AtlasProbe)-[:PART_OF]-(msm)
RETURN msm_target, probes
```

ASes classified as academic networks by BGP.Tools:

```cypher
MATCH p=(:AS)-[r:CATEGORIZED {reference_name:'bgptools.tags'}]-
  (:Tag {label:'Academic'})
RETURN p LIMIT 25
```

AS population in US aggregated per AS names:

```cypher
MATCH (eyeball:AS)-[pop:POPULATION]-(c:Country)
WHERE c.country_code = 'US'
// Find the name for each AS
OPTIONAL MATCH (eyeball)-[:NAME {reference_org:'BGP.Tools'}]-(n:Name)
// Group ASNs by name (first word of the name), list all ASNs, and the total population
RETURN head(split(n.name,' ')), collect(eyeball.asn), sum(pop.percent) as total_pop
ORDER BY total_pop DESC
```

Other examples online

More example queries are available at the following pages:

- Understanding the Japanese Internet with the Internet Yellow Pages, APNIC blog
- IYP Gallery
- RiPKI: The Tragic Story of RPKI Deployment in the Web Ecosystem, HotNets'15 (reproduction)
- Comments on DNS Robustness, IMC'18 (reproduction)

Exercise

- Find all hostnames in IYP that ends with '.gov'
- Find all hostnames in IYP that ends with '.gov' and resolves to IPs in RPKI NotFound prefixes
- Which ASes host the most popular content but are not tagged as 'Content' by BGP.Tools?
- Find popular domain names managed by authoritative nameservers hosted at UCSD (AS7377) and authoritative nameservers hosted at San Diego Supercomputer Center (AS195).

Accessing IYP from code

For a more systematic analysis you can access IYP via different programming languages. Neo4j offers a variety of drivers, but we will only discuss the Python driver here.

Setup

First, install the driver:

```bash
# To make the setup cleaner, you can also create a virtual environment and install neo4j in there
# For Linux:
# python3 -m venv .venv
# source .venv/bin/activate
# For Windows/Mac: ¯\_(ツ)_/¯

pip install neo4j
```

To verify that the setup worked, you can run this simple script:

```python
from neo4j import GraphDatabase

URI = 'neo4j://iyp-bolt.ihr.live:7687'
AUTH = None
db = GraphDatabase.driver(URI, auth=AUTH)

db.verify_connectivity()
db.close()
```

If you run a local instance of IYP (described below) you will need to specify a username and password like this:

```python
URI = 'neo4j://localhost:7687'
AUTH = ('neo4j', 'password')
```

Simple queries from Python

There are multiple ways to query the database, but for the purpose of this tutorial, we will stick to the simplest one: `execute_query()`. You can just pass the query to this function, (almost) like you did before in the browser interface.

```python
# Import of module and db setup excluded.

records, _, _ = db.execute_query(
    """
    MATCH (iij:AS)-[:MEMBER_OF]-(ix:IXP)
    WHERE iij.asn = 2497
    RETURN DISTINCT(ix.name) AS name
    """
)
for r in records:
    print(r['name'])
```

Note that there are some key differences in the query. Instead of specifying and returning a path `p`, like we did before, we only return a single property of the resulting nodes. Otherwise, we would retrieve the results in the form of a path, including all properties of all nodes and relationships, which is probably not what we are interested in. In addition, we assign a name to the result (using `AS`), which will be used as the key in the resulting dictionary.

Executing static queries is boring, so there is also the option to use placeholders (starting with $) in the query and pass their value via a function parameter. This example retrieves the number of IPv4 and IPv6 prefixes that IIJ originates:

```python
query = """
    MATCH (iij:AS)-[:ORIGINATE]-(pfx:Prefix)
    WHERE iij.asn = 2497
    AND pfx.af = $address_version
    RETURN COUNT(DISTINCT pfx) AS num_prefixes
"""
records, _, _ = db.execute_query(query, address_version=4)
# execute_query always returns a list, but we know it only has one entry.
num_v4_prefixes = records[0]['num_prefixes']
records, _, _ = db.execute_query(query, address_version=6)
num_v6_prefixes = records[0]['num_prefixes']
print(f'IIJ originates {num_v4_prefixes} IPv4 and {num_v6_prefixes} IPv6 prefixes.')
```

Using Pandas data frames

If your results are a bit more complex and you like working with Pandas data frames, you can use the keys returned by `execute_query()` to easily load the results into a data frame. For example, to get the top 1000 ASes from AS Rank with their name as known by RIPE and their registered country according to the NRO delegated stats:

```python
import pandas as pd

# [...] database init like above

query = """
    MATCH (a:AS)-[r:RANK {reference_name: 'caida.asrank'}]-(:Ranking)
    OPTIONAL MATCH (a)-[:NAME {reference_name: 'ripe.as_names'}]-(n:Name)
    OPTIONAL MATCH (a)-[:COUNTRY {reference_name: 'nro.delegated_stats'}]-(c:Country)
    RETURN a.asn AS asn, r.rank AS as_rank, n.name AS name, c.country_code as country
    ORDER BY as_rank
    LIMIT 1000
"""

records, _, keys = db.execute_query(query)
df = pd.DataFrame(records, columns=keys)

# Alternatively, Neo4j can directly transform the results into a
dataframe
# df = db.execute_query(query,
result_transformer_=neo4j.Result.to_df)
```

For querying IYP this is pretty much all you need to know. For more examples see the Jupyter notebooks we provide as part of our paper. However, you will notice that only the queries are more involved, the Python functions are the same.

References

- Neo4j Python driver manual
- Python driver API documentation

Hosting a local IYP instance

To perform extensive analysis or analysis including your own datasets, we recommend that you locally host your own instance of IYP. This is also useful if you are on-the-go without a stable Internet connectivity. Don’t worry, apart from disk space, IYP is pretty lightweight!

System requirements

- Docker + Docker Compose
- About 100GB of free disk space
- At least 2GB of RAM
- At least 1 CPU core :)

Clone the IYP repository

The first thing we have to do is to clone the IYP repository:

```bash
git clone
https://github.com/InternetHealthReport/internet-yellow-pages.git
```

Go to the directory `internet-yellow-pages`, the following steps (download database and setup IYP) should be done from this directory.

Download a database dump

Visit the database dump repository.

Dumps are organized by year, month, and day in this format:

```text
https://archive.ihr.live/ihr/iyp/YYYY/MM/DD/iyp-YYYY-MM-DD.dump
```

For the purpose of this tutorial, we recommend the latest dump (warning: link goes to a 8.6GB file):

```text
https://archive.ihr.live/ihr/iyp/2025/02/01/iyp-2025-02-01.dump
```

Also available on CAIDA's server:

```text
https://www.caida.org/~ark/iyp-2025-02-01.dump
```

This dump requires about 100GB of disk space once it is loaded. If this is too much for your machine, you can also use an older dump that is missing some datasets (CAIDA AS Relationship, Google CrUX, OONI censorship, some DNS CNAMEs), but only requires 40GB of disk space (file is 4.1GB).

```text
https://archive.ihr.live/ihr/iyp/2024/07/22/iyp-2024-07-22.dump
```

The dump file needs to be called `neo4j.dump` and needs to be put in a folder called `dumps` (`dumps/neo4j.dump`).

To create the folder and download a dump with `curl`:

```bash
mkdir dumps
curl https://archive.ihr.live/ihr/iyp/2025/02/01/iyp-2025-02-01.dump
-o dumps/neo4j.dump
```

Set up IYP

For Mac users with ARM-based machines: You might need to edit the Docker Compose file (docker-compose.yaml). Change the following line:
```
iyp_loader:
    image: neo4j/neo4j-admin:5.21.2
```
to
```
iyp_loader:
    image: neo4j/neo4j-admin:5.21.2-arm
```

To uncompress the dump and start the database run the following command:

```bash
mkdir -p data
uid="$(id -u)" gid="$(id -g)" docker compose --profile local up
```

This creates a `data` directory containing the database, loads the database dump, and starts the local IYP instance. This initial setup needs to be done only once but it takes some time to completely load the database and start IYP. **Please wait until IYP is fully loaded (indicated by the message `Started.`).**

This step won't work if the data directory already contains a database. To delete an existing database, simply delete the contents of the `data` directory.

This setup keeps the database instance running in the foreground. It can be stopped with `Ctrl+C`. Afterwards, you can simply start/stop IYP in the background
to use it.

Start/Stop IYP

To start the database, run the following command:

```bash
docker start iyp
```
To stop the database, run the following command:

``` bash
docker stop iyp
```

Adding data to IYP

Before you start hacking away, the first, and very important, step is to **model your data as a graph**. Take a look at the existing node and relationship types and see if and where your data could attach to the existing graph and if you can reuse existing relationship types. Also refer back to the IYP data modeling section. If you need help, feel free to start a discussion on GitHub.

Once you have modeled your data, you can start writing a *crawler*. The main tasks of a crawler are to fetch data, parse it, model it with IYP ontology, and push it to the IYP database. Most of these tasks are assisted by the IYP python library (described next).

IYP code structure

The repository and code is structured like this:

```
internet-yellow-pages/
├─ iyp/
│  ├─ __init__.py <- contains IYP module
│  ├─ crawlers/
│  │  ├─ org/ <- name of the organization
│  │  │  ├─ README.md <- README describing datasets and modelling
│  │  │  ├─ crawler1.py <- one crawler per dataset
│  │  │  ├─ crawler2.py
│  ├─ post/ <- for post-processing scripts
```

The canonical way to execute a crawler is:

```bash
python3 -m iyp.crawlers.org.crawler1
```

Writing a IYP crawler

A full explanation of how to write a crawler from scratch is outside the scope of this tutorial. To get you started, we point you to the existing documentation, the example crawler that you can use as a template, and the best practices for writing crawlers. You can also look at other existing crawlers and of course always contact us for help.

Making data publicly available

If you want to add private data to your own instance, feel free to do so. However, we welcome crawler contributions that add data to IYP!

The workflow for this is usually as follows:

1. Propose a new dataset by opening a discussion. The point of the discussion is to decide if a dataset should be included and how to model it. Please add a short description of why the dataset would be useful for you/others. This is just to prevent adding datasets for the sake of it (“because we can”) which inflates to database size. You also do not have to provide a perfect model at the start, we can figure this out together.
2. Once it is decided that we want to integrate the dataset and how to model it, the discussion will be converted into an issue. Then you (or someone else) can implement it.
3. Open a pull request with the crawler implementation.
4. We will merge it and the next dump will contain your dataset!