INDRA Database Documentation

Release 1.0.0

P. A. Greene, J. A. Bachman, B. M. Gyori

Feb 01, 2022

CONTENTS

The INDRA (Integrated Network and Dynamical Reasoning Assembler) Database is a framework for creating, maintaining, and accessing a database of content, readings, and statements. This implementation is currently designed to work primarily with Amazon Web Services RDS running Postrgres 9+. Used as a backend to INDRA, the INDRA Database provides a systematic way of scaling the knowledge acquired from other databases, reading, and manual input, and puts that knowledge at your fingertips through a direct Python client and a REST api.

CHAPTER

ONE

KNOWLEDGE SOURCES

The INDRA Database currently integrates and distills knowledge from several different sources, both biology-focused natural language processing systems and other pre-existing databases

1.1 Daily Readers

We have read all available content, and every day we run the following readers:

- REACH
- Sparser

we read all new content with the following readers:

- Eidos
- ISI
- MTI - used specifically to tag content with topic terms.

we read a limited subset of new content with the following readers:

- TRIPS

on the latest content drawn from:

- PubMed - ~19 million abstracts and ~29 million titles
- PubMed Central - ~2.7 million fulltext
- Elsevier - ~0.7 million fulltext (requires special access)

1.2 Other Readers

We also include more or less static content extracted from the following readers:

- RLIMS-P

1.3 Other Databases

We include the information from these pre-existing databases:

- Pathway Commons database
- BEL Large Corpus
- SIGNOR
- BioGRID
- TAS
- TRRUST
- PhosphoSitePlus
- Causal Biological Networks Database
- VirHostNet
- CTD
- Phospho.ELM
- DrugBank
- CONIB
- CRoG
- DGI

These databases are retrieved primarily using the tools in `indra.sources`. The statements extracted from all of these sources are stored and updated in the database.

CHAPTER

TWO

KNOWLEDGE ASSEMBLY

The INDRA Database uses the powerful internal assembly tools available in INDRA but implemented for large-scale incremental assembly. The resulting corpus of cleaned and de-duplicated statements, each with fully maintained provenance, is the primary product of the database.

For more details on the internal assembly process of INDRA, see the INDRA documentation.

CHAPTER

THREE

ACCESS

The content in the database can be accessed by those that created it using the `indra_db.client` submodule. This repo also implements a REST API which can be used by those without direct acccess to the database. For access to our REST API, please contact the authors.

The INDRA database only works for Python 3.6+, though some parts are still compatible with 3.5.

First, install INDRA, then simply clone this repo, and make sure that it is visible in your `PYTHONPATH`.

The development of INDRA DB is funded under the DARPA Communicating with Computers program (ARO grant W911NF-15-1-0544).

CHAPTER

FOUR

FURTHER INDRA DATABASE DOCUMENTATION

4.1 License and funding

The INDRA was developed with funding from ARO grant W911NF-14-1-0397, “Programmatic modelling for reasoning across complex mechanisms” under the DARPA Big Mechanism program, and the INDRA database was developed as an extention of that core project. Work has continued under W911NF-14-1-0391, “Active context” under the DARPA Communicating with Computers program, and the DARPA Automated Scientific Discovery Framework project.

4.2 INDRA Database modules

4.2.1 The Client

The purpose of the client is to be the gateway for external access to the content of the databases. Here we define high level access functions for getting data out of the database in a natural way. This is where the queries used by the REST API are defined, and most users looking to access knowledge on the database should use the client if they can, as it is heavily optimized.

Our system utilizes 2 databases, one which represents the “ground truth”, as we know it, and is structured naturally for performing updates on our knowledge; it will always be the most up to date. We also have a “readonly” database that we used for our outward facing services. This database is optimized for fast queries and the content in it is updated weekly. Each database has its own set of access tools.

The Principal Database Client

This is the set of client tools to access the most-nearly ground truth knowledge stored on the principal database.

Access Readings and Text Content (`indra_db.client.principal.content`)

This defines a simple API to access the content that we store on the database for external purposes.

`indra_db.client.principal.content.`**`get_content_by_refs`**(*db*, *pmid_list=None*, *trid_list=None*, *sources=None*, *formats=None*, *content_type='abstract'*, *unzip=True*)

Return content from the database given a list of PMIDs or text ref ids.

Note that either pmid_list OR trid_list must be set, and only one can be set at a time.

Parameters

- **db** (`DatabaseManager`) – Reference to the DB to query
- **pmid_list** (*list[str] or None*) – A list of pmids. Default is None, in which case trid_list must be given.
- **trid_list** (*list[int] or None*) – A list of text ref ids. Default is None, in which case pmid list must be given.
- **sources** (*list[str] or None*) – A list of sources to include (e.g. 'pmc_oa', or 'pubmed'). Default is None, indicating that all sources will be included.
- **formats** (*list[str]*) – A list of the formats to be included ('xml', 'text'). Default is None, indicating that all formats will be included.
- **content_type** (*str*) – Select the type of content to load ('abstract' or 'fulltext'). Note that not all refs will have any, or both, types of content.
- **unzip** (*Optional[bool]*) – If True, the compressed output is decompressed into clear text. Default: True

Returns **content_dict** – A dictionary whose keys are text ref ids, with each value being the the corresponding content.

Return type dict

`indra_db.client.principal.content.`**`get_reader_output`**(*db*, *ref_id*, *ref_type='tcid'*, *reader=None*, *reader_version=None*)

Return reader output for a given text content.

Parameters

- **db** (`DatabaseManager`) – Reference to the DB to query
- **ref_id** (*int or str*) – The text reference ID whose reader output should be returned
- **ref_type** (*Optional[str]*) – The type of ID to look for, options include 'tcid' for the database's internal unique text content ID, or 'pmid', 'pmcid', 'doi, 'pii', 'manuscript_id' Default: 'tcid'
- **reader** (*Optional[str]*) – The name of the reader whose output is of interest
- **reader_version** (*Optional[str]*) – The specific version of the reader

Returns **reading_results** – A dict of reader outputs that match the query criteria, indexed first by text content id, then by reader.

Return type dict{dict{list[str]}}

Submit and Retrieve Curations (indra_db.client.principal.curation)

On our services, users have the ability to curate the results we present, indicating whether they are correct or not, and how they may be incorrect. The API for adding and retrieving that input is defined here.

indra_db.client.principal.curation.**get_curations**(*db=None*, ***params*)

Get all curations for a certain level given certain criteria.

indra_db.client.principal.curation.**get_grounding_curations**(*db=None*)

Return a dict of curated groundings from a given database.

Parameters **db** (*Optional[DatabaseManager]*) – A database manager object used to access the database. If not given, the database configured as primary is used.

Returns A dict whose keys are raw text strings and whose values are dicts of DB name space to DB ID mappings corresponding to the curated grounding.

Return type dict

indra_db.client.principal.curation.**submit_curation**(*hash_val*, *tag*, *curator*, *ip*, *text=None*, *ev_hash=None*, *source='direct_client'*, *pa_json=None*, *ev_json=None*, *db=None*)

Submit a curation for a given preassembled or raw extraction.

Parameters

- **hash_val** (*int*) – The hash corresponding to the statement.
- **tag** (*str*) – A very short phrase categorizing the error or type of curation.
- **curator** (*str*) – The name or identifier for the curator.
- **ip** (*str*) – The ip address of user's computer.
- **text** (*str*) – A brief description of the problem.
- **ev_hash** (*int*) – A hash of the sentence and other evidence information. Elsewhere referred to as *source_hash*.
- **source** (*str*) – The name of the access point through which the curation was performed. The default is 'direct_client', meaning this function was used directly. Any higher-level application should identify itself here.
- **pa_json** (*Optional[dict]*) – The JSON of a preassembled or raw statement that was curated. If None, we will try to get the pa_json from the database.
- **ev_json** (*Optional[dict]*) – The JSON of the evidence that was curated. This cannot be retrieved from the database if not given.
- **db** (DatabaseManager) – A database manager object used to access the database.

Get Raw Statements (`indra_db.client.principal.raw_statements`)

Get the raw, uncleaned and un-merged Statements based on agent and type or by paper(s) of origin.

`indra_db.client.principal.raw_statements.`**`get_raw_stmt_jsons`**(*clauses=None*, *db=None*, *max_stmts=None*, *offset=None*)

Get Raw Statements from the principle database, given arbitrary clauses.

`indra_db.client.principal.raw_statements.`**`get_raw_stmt_jsons_from_agents`**(*agents=None*, *stmt_type=None*, *db=None*, *max_stmts=None*, *offset=None*)

Get Raw statement jsons from a list of agent refs and Statement type.

`indra_db.client.principal.raw_statements.`**`get_raw_stmt_jsons_from_papers`**(*id_list*, *id_type='pmid'*, *db=None*, *max_stmts=None*, *offset=None*)

Get raw statement jsons for a given list of papers.

Parameters

- **`id_list`** (*`list`*) – A list of ints or strs that are ids of papers of type *id_type*.
- **`id_type`** (*`str`*) – Default is 'pmid'. The type of ids given in id_list, e.g. 'pmid', 'pmcid', 'trid'.
- **`db`** (`DatabaseManager`) – Optionally specify a database manager that attaches to something besides the primary database, for example a local database instance.

Returns **result_dict** – A dictionary keyed by id (of *id_type*) with a list of raw statement json objects as each value. Ids for which no statements are found will not be included in the dict.

Return type dict

The Readonly Client

Here are our primary tools intended for retrieving Statements, in particular Pre-Assembled (PA) Statements, from the readonly database. This is some of the most heavily optimized access code in the repo, and is the backbone of most external or outward facing applications.

The readonly database, as the name suggests, is designed to take only read requests, and is updated via dump only once a week. This allows users of our database to access it even as we perform daily updates on the principal database, without worrying about queries interfering.

Construct composable queries (`indra_db.client.readonly.query`)

This is a sophisticated system of classes that can be used to form queires for preassembled statements from the readonly database.

`class` `indra_db.client.readonly.query.`**`Query`**(*empty=False*, *full=False*)

The core class for all queries; not functional on its own.

`copy`()

Get a _copy of this query.

invert()

A useful way to get the inversion of a query in order of operations.

When chain operations, *~q* is evaluated after all . terms. This allows you to cleanly bypass that issue, having:

> HasReadings().invert().get_statements(ro)

rather than

> (~HasReadings()).get_statements()

which is harder to read.

set_print_only(*print_only*)

Choose to only print the SQL and not execute it.

This is very useful for debugging the SQL queries that are generated.

get_statements(*ro=None*, *limit=None*, *offset=None*, *sort_by='ev_count'*, *ev_limit=None*, *evidence_filter=None*) → Optional[indra.sources.indra_db_rest.query_results.StatementQueryResult]

Get the statements that satisfy this query.

Parameters

- **ro** (DatabaseManager) – A database manager handle that has valid Readonly tables built.
- **limit** (*int*) – Control the maximum number of results returned. As a rule, unless you are quite sure the query will result in a small number of matches, you should limit the query.
- **offset** (*int*) – Get results starting from the value of offset. This along with limit allows you to page through results.
- **sort_by** (*str*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter.
- **ev_limit** (*int*) – Limit the number of evidence returned for each statement.
- **evidence_filter** (*None or* EvidenceFilter) – If None, no filtering will be applied. Otherwise, an EvidenceFilter class must be provided.

Returns **result** – An object holding the JSON result from the database, as well as the metadata for the query.

Return type StatementQueryResult

get_hashes(*ro=None*, *limit=None*, *offset=None*, *sort_by='ev_count'*, *with_src_counts=True*) → Optional[indra.sources.indra_db_rest.query_results.QueryResult]

Get the hashes of statements that satisfy this query.

Parameters

- **ro** (DatabaseManager) – A database manager handle that has valid Readonly tables built.
- **limit** (*int*) – Control the maximum number of results returned. As a rule, unless you are quite sure the query will result in a small number of matches, you should limit the query.
- **offset** (*int*) – Get results starting from the value of offset. This along with limit allows you to page through results.
- **sort_by** (*str*) – 'ev_count' or 'belief': select the parameter by which results are sorted.
- **with_src_counts** (*bool*) – Choose whether source counts are included with the result or not. The default is True (included), but the query may be marginally faster with source counts excluded (False).

Returns **result** – An object holding the results of the query, as well as the metadata for the query definition.

Return type QueryResult

`get_interactions`(*ro=None*, *limit=None*, *offset=None*, *sort_by='ev_count'*) → Optional[indra.sources.indra_db_rest.query_results.QueryResult]

Get the simple interaction information from the Statements metadata.

Each entry in the result corresponds to a single preassembled Statement, distinguished by its hash.

Parameters

- **ro** (`DatabaseManager`) – A database manager handle that has valid Readonly tables built.
- **limit** (*int*) – Control the maximum number of results returned. As a rule, unless you are quite sure the query will result in a small number of matches, you should limit the query.
- **offset** (*int*) – Get results starting from the value of offset. This along with limit allows you to page through results.
- **sort_by** (*str*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter.

`get_relations`(*ro=None*, *limit=None*, *offset=None*, *sort_by='ev_count'*, *with_hashes=False*) → Optional[indra.sources.indra_db_rest.query_results.QueryResult]

Get the agent and type information from the Statements metadata.

Each entry in the result corresponds to a relation, meaning an interaction type, and the names of the agents involved.

Parameters

- **ro** (`DatabaseManager`) – A database manager handle that has valid Readonly tables built.
- **limit** (*int*) – Control the maximum number of results returned. As a rule, unless you are quite sure the query will result in a small number of matches, you should limit the query.
- **offset** (*int*) – Get results starting from the value of offset. This along with limit allows you to page through results.
- **sort_by** (*str*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter.
- **with_hashes** (*bool*) – Default is False. If True, retrieve all the hashes that fit within each relational grouping.

`get_agents`(*ro=None*, *limit=None*, *offset=None*, *sort_by='ev_count'*, *with_hashes=False*, *complexes_covered=None*) → Optional[indra.sources.indra_db_rest.query_results.QueryResult]

Get the agent pairs from the Statements metadata.

Each entry is simply a pair (or more) of Agents involved in an interaction.

Parameters

- **ro** (*Optional[*`DatabaseManager`*]*) – A database manager handle that has valid Readonly tables built.
- **limit** (*Optional[int]*) – Control the maximum number of results returned. As a rule, unless you are quite sure the query will result in a small number of matches, you should limit the query.

- **offset** (`Optional[int]`) – Get results starting from the value of offset. This along with limit allows you to page through results.
- **sort_by** (`str`) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter.
- **with_hashes** (`bool`) – Default is False. If True, retrieve all the hashes that fit within each agent pair grouping.
- **complexes_covered** (`Optional[set]`) – The set of hashes for complexes that you have already seen and would like skipped.

to_json() → dict

Get the JSON representation of this query.

classmethod **from_simple_json**(*json_dict*)

Generate a proper query from a simplified JSON.

list_component_queries() → list

Get a list of the query elements included, in no particular order.

build_hash_query(*ro*, *type_queries=None*)

[Internal] Build the query for hashes.

is_inverse_of(*other*)

Check if a query is the exact opposite of another.

class indra_db.client.readonly.query.**Intersection**(*query_list*)

The Intersection of multiple queries.

Baring special handling, this is what results from q1 & q2.

NOTE: the inverse of an Intersection is a Union (De Morgans's Law)

ev_filter()

Get an evidence filter composed of the "and" of sub-query filters.

is_inverse_of(*other*)

Check if this query is the inverse of another.

class indra_db.client.readonly.query.**Union**(*query_list*)

The union of multiple queries.

Baring special handling, this is generally the result of q1 | q2.

NOTE: the inverse of a Union is an Intersection (De Morgans's Law)

ev_filter()

Get an evidence filter composed of the "or" of sub-query filters.

is_inverse_of(*other*)

Check if this query is the inverse of another.

class indra_db.client.readonly.query.**MergeQuery**(*query_list*, **args*, ***kwargs*)

This is the parent of the two merge classes: Intersection and Union.

This class of queries is extremely special, in that the "table" is actually constructed on the fly. This presents various subtle challenges. Moreover an intersection/union is an expensive process, so I go to great lengths to minimize its use, making the __init__ methods quite hefty. It is also in Intersections and Unions that *full* and *empty* states are most likely to occur, and in some wonderfully subtle and hard to find ways.

class indra_db.client.readonly.query.**HasAgent**(*agent_id=None*, *namespace='NAME'*, *role=None*, *agent_num=None*)

Get Statements that have a particular agent in a particular role.

NOTE: At this time 2 agent queries do NOT necessarily imply that the 2 agents are different. E.g. `` `HasAgent("MEK") & HasAgent("MEK")` `` will get any Statements that have agent with name MEK, not Statements with two agents called MEK. This may change in the future, however in the meantime you can get around this fairly well by specifying the roles:

```
>>> HasAgent("MEK", role="SUBJECT") & HasAgent("MEK", role="OBJECT")
```

Or for a more complicated case, consider a query for Statements where one agent is MEK and the other has namespace FPLX. Naturally any agent labeled as MEK will also have a namespace FPLX (MEK is a famplex identifier), and in general you will not want to constrain which role is MEK and which is the "other" agent. To accomplish this you need to use `` `|` ``:

```
>>> (
>>>   HasAgent("MEK", role="SUBJECT")
>>>   & HasAgent(namespace="FPLX", role="OBJECT")
>>> ) | (
>>>   HasAgent("MEK", role="OBJECT")
>>>   & HasAgent(namespace="FPLX", role="SUBJECT")
>>> )
```

Parameters

- **agent_id** (*Optional[str]*) – The ID string naming the agent, for example 'ERK' (FPLX or NAME) or 'plx' (TEXT), and so on. If None, the query must then be constrained by the namespace. (Default is None)
- **namespace** (*Optional[str]*) – By default, this is NAME, indicating the canonical name of the agent. Other options for namespace include FPLX (FamPlex), CHEBI, CHEMBL, HGNC, UP (UniProt), TEXT (for raw text mentions), and many more. If you use the namespace AUTO, GILDA will be used to try and guess the proper namespace and agent ID. If *agent_id* is None, namespace must be specified and must not be NAME, TEXT, or AUTO.
- **role** (*Optional[str]*) – Options are "SUBJECT", "OBJECT", or "OTHER". (Default is None)
- **agent_num** (*Optional[int]*) – The regularized position of the agent in the Statement's list of agents. (Default is None)

class indra_db.client.readonly.query.**FromMeshIds**(*mesh_ids: list*)

Find Statements whose text sources were given one of a list of MeSH IDs.

This object can be constructed from a list of mixed "D" and "C" type mesh IDs, but for reasons of querying, those IDs will be separated into two separate classes and a *Union* of the two classes returned.

Parameters **mesh_ids** (*list*) – A canonical MeSH ID, of the "C" or "D" variety, e.g. "D000135".

mesh_ids

The immutable tuple of mesh IDs, on their original string form.

Type tuple

_mesh_type

"C" or "D" indicating which types of IDs are held in this object.

Type str

_mesh_nums

The mesh IDs converted to integers, stripped of their prefix.

Type list[int]

`ev_filter()`

Get an evidence filter to enforce mesh constraints at ev level.

`class indra_db.client.readonly.query.HasHash(`*stmt_hashes*`)`

Find Statements from a list of hashes.

Parameters `stmt_hashes` (*list or set or tuple*) – A collection of integers, where each integer is a shallow matches key hash of a Statement (frequently simply called "mk_hash" or "hash")

`class indra_db.client.readonly.query.HasSources(`*sources*`)`

Find Statements that include a set of sources.

For example, find Statements that have support from both medscan and reach.

Parameters `sources` (*list or set or tuple*) – A collection of strings, each string the canonical name for a source. The result will include statements that have evidence from ALL sources that you include.

`class indra_db.client.readonly.query.HasOnlySource(`*only_source*`)`

Find Statements that come exclusively from a particular source.

For example, find statements that come only from sparser.

Parameters `only_source` (*str*) – The only source that spawned the statement, e.g. signor, or reach.

`class indra_db.client.readonly.query.HasReadings`

Find Statements that have readings.

`class indra_db.client.readonly.query.HasDatabases`

Find Statements that have databases.

`class indra_db.client.readonly.query.SourceQuery(`*empty=False*, *full=False*`)`

The core of all queries that use SourceMeta.

`class indra_db.client.readonly.query.SourceIntersection(`*source_queries*`)`

A special type of intersection between children of SourceQuery.

All SourceQuery queries use the same table, so when doing an intersection it doesn't make sense to do an actual intersection operation, and instead simply apply all the filters of each query to build a normal multi- conditioned query.

`is_inverse_of(`*other*`)`

Check if this query is the inverse of another.

`class indra_db.client.readonly.query.HasType(`*stmt_types*, *include_subclasses=False*`)`

Find Statements that are one of a collection of types.

For example, you can find Statements that are Phosphorylations or Activations, or you could find all subclasses of RegulateActivity.

NOTE: when used in an Intersection with other queries, type is handled specially, with each sub query having a type constraint added to it.

Parameters

- `stmt_types` (*set or list or tuple*) – A collection of Strings, where each string is a class name for a type of Statement. Spelling and capitalization are necessary.
- `include_subclasses` (*bool*) – (optional) default is False. If True, each Statement type given in the list will be expanded to include all of its sub classes.

`item_type`

alias of `str`

class `indra_db.client.readonly.query.`**`IntrusiveQuery`**(*value_list*)

This is the parent of all queries that draw on info in all meta tables.

Thus, when using these queries in an Intersection, they are applied to each sub query separately.

class `indra_db.client.readonly.query.`**`HasNumAgents`**(*agent_nums*)

Find Statements with any one of a listed number of agents.

> For example, *HasNumAgents([1,3,4])* will return agents with either 2, 3, or 4 agents (the latter two mostly being complexes).

NOTE: when used in an Interaction with other queries, the agent numbers are handled specially, with each sub-query having an agent_count constraint applied to it.

Parameters **`agent_nums`** (*`tuple`*) – A list of integers, each indicating a number of agents.

`item_type`

alias of `int`

class `indra_db.client.readonly.query.`**`HasNumEvidence`**(*evidence_nums*)

Find Statements with one of a given number of evidence.

For example, HasNumEvidence([2,3,4]) will return Statements that have either 2, 3, or 4 evidence.

NOTE: when used in an Interaction with other queries, the evidence count is handled specially, with each sub-query having an ev_count constraint added to it.

Parameters **`evidence_nums`** (*`tuple`*) – A list of numbers greater than 0, each indicating a number of evidence.

`item_type`

alias of `int`

class `indra_db.client.readonly.query.`**`FromPapers`**(*paper_list*)

Find Statements that have evidence from particular papers.

Parameters **`paper_list`** (*`list[(<id_type>, <paper_id>)]`*) – A list of tuples, where each tuple indicates and id-type (e.g. 'pmid') and an id value for a particular paper.

class `indra_db.client.readonly.query.`**`EvidenceFilter`**(*filters=None*, *joiner='and'*)

Object for handling filtering of evidence.

We need to be able to perform logical operations between evidence to handle important cases:

- `HasSource(['reach']) & FromMeshIds(['D0001'])`: we might reasonably want to filter evidence for the second subquery but not the first.
- `HasOnlySource(['reach']) & FromMeshIds(['D00001'])`: Here we would likely want to filter the evidence for both sub queries.
- `HasOnlySource(['reach']) | FromMeshIds(['D000001'])`: It is not clear what this even means (its purpose) or what we'd do for evidence filtering when the original statements are or'ed
- `HasDatabases() & FromMeshIds(['D000001'])`: Here you COULDN'T perform an & on the evidence, because the two sources are mutually exclusive (only readings connect to mesh annotations). However it could make sense you would want to do an "or" between the evidence, so the evidence is either from a database or from a mesh annotated document.

Both "filter all the evidence" and "filter none of the evidence" should definitely be options. Although "Filter for all" might run into uses with the "HasDatabase and FromMeshIds" scenario. I think no evidence filter should be the default, and if you attempt a bogus "filter all evidence" (as with that scenario) you get an error.

class `indra_db.client.readonly.query.`**`FromAgentJson`**(*agent_json*, *stmt_type=None*, *hashes=None*)

A Very special type of query that is used for digging into results.

class `indra_db.client.readonly.query.`**`HasEvidenceBound`**(*evidence_bounds: Iterable[Union[str, indra_db.client.readonly.query.Bound]]*)

Find Statements that fit given evidence bounds.

A list of bounds will be combined using the logic of "or", so ["<1", ">3"] will return Statements that are _either_ less than 1 OR greater than 3.

Parameters **`evidence_bounds`** – An iterable containing bounds for the evidence support of Statements to be returned, such as *Bound("< 10")* or simply "< 10" (the string will be parsed into a Bound object, if possible).

Miscellaneous Client APIs (Mostly Deprecated)

There are some, generally archaic, client functions which use both readonly and principal resources. I make no guarantee that these will work.

Get Datasets (`indra_db.client.datasets`)

An early attempt at something very like the `indra_db.client.readonly.interactions` approach to getting superficial data out of the database.

`indra_db.client.datasets.`**`export_relation_dict_to_tsv`**(*relation_dict*, *out_base*, *out_types=None*)

Export a relation dict (from get_relation_dict) to a tsv.

Available output types are:

- "full_tsv" : get a tsv with directed pairs of entities (e.g. HGNC symbols), the type of relation (e.g. Phosphorylation) and the hash of the preassembled statement. Columns are agent_1, agent_2 (where agent_1 affects agent_2), type, hash.
- "short_tsv" : like the above, but without the hashes, so only one instance of each pair and type trio occurs. However, the information cannot be traced. Columns are agent_1, agent_2, type, where agent_1 affects agent_2.
- "pairs_tsv" : like the above, but without the relation type. Similarly, each row is unique. In addition, the agents are undirected. Thus this is purely a list of pairs of related entities. The columns are just agent_1 and agent_2, where nothing is implied by the ordering.

Parameters

- **`relation_dict`** (*`dict`*) – This should be the output from *get_relation_dict*, or something equivalently constructed.
- **`out_base`** (*`str`*) – The base-name for the output files.
- **`out_types`** (*`list[str]`*) – A list of the types of tsv to output. See above for details.

`indra_db.client.datasets.`**`get_relation_dict`**(*db*, *groundings=None*, *with_evidence_count=False*, *with_support_count=False*)

Get a dictionary of entity interactions from the database.

Use only metadata from the database to rapidly get simple interaction data. This is much faster than handling the full Statement jsons, while providing some basic valuable functionality.

Parameters

- **`db`** (*`DatabaseManager instance`*) – An instance of a database manager.

- **groundings** (*list[str] or None*) – Select which types of grounding namespaces to include, e.g. HGNC, or FPLX, or both. Only agent refs with these groundings will be selected. If None, only HGNC is used.
- **with_evidence_count** (*bool*) – Default is False. If True, an additional query will be made for each statement to get the count of supporting evidence, which is a useful proxy for belief.
- **with_support_count** (*bool*) – Default is False. Like *with_evidence_count*, except the number of supporting statements is counted.

indra_db.client.datasets.**get_statement_essentials**(*clauses*, *count=1000*, *db=None*, *preassembled=True*)

Get the type, agents, and id data for the specified statements.

This function is useful for light-weight searches of basic mechanistic information, without the need to follow as many links in the database to populate the Statement objects.

To get full statements, use *get_statements*.

Parameters

- **clauses** (*list*) – list of sqlalchemy WHERE clauses to pass to the filter query.
- **count** (*int*) – Number of statements to retrieve and process in each batch.
- **db** (DatabaseManager) – Optionally specify a database manager that attaches to something besides the primary database, for example a local database instance.
- **preassembled** (*bool*) – If true, statements will be selected from the table of pre-assembled statements. Otherwise, they will be selected from the raw statements. Default is True.

Returns *(uuid, sid, hash, type, (agent_1, agent_2, . . .)).*

Return type A list of tuples containing

Get Statements (indra_db.client.statements)

The first round of tools written to get Statements out of the database, utilizing far too many queries and taking absurdly long to complete. Most of their functions have been outmoded, with the exception of getting PA Statements from the principal database, which (as of this writing) has yet to be implemented.

indra_db.client.statements.**get_evidence**(*pa_stmt_list*, *db=None*, *fix_refs=True*, *use_views=True*)

Fill in the evidence for a list of pre-assembled statements.

Parameters

- **pa_stmt_list** (*list[Statement]*) – A list of unique statements, generally drawn from the database pa_statement table (via *get_statemetns*).
- **db** (*DatabaseManager instance or None*) – An instance of a database manager. If None, defaults to the “primary” database, as defined in the db_config.ini file in .config/indra.
- **fix_refs** (*bool*) – The paper refs within the evidence objects are not populated in the database, and thus must be filled using the relations in the database. If True (default), the *pmid* field of each Statement Evidence object is set to the correct PMIDs, or None if no PMID is available. If False, the *pmid* field defaults to the value populated by the reading system.

Return type None - modifications are made to the Statements “in-place”.

`indra_db.client.statements.get_statements`(*clauses*, *count=1000*, *do_stmt_count=False*, *db=None*, *preassembled=True*, *with_support=False*, *fix_refs=True*, *with_evidence=True*)

Select statements according to a given set of clauses.

Parameters

- **clauses** (*list*) – list of sqlalchemy WHERE clauses to pass to the filter query.
- **count** (*int*) – Number of statements to retrieve and process in each batch.
- **do_stmt_count** (*bool*) – Whether or not to perform an initial statement counting step to give more meaningful progress messages.
- **db** (DatabaseManager) – Optionally specify a database manager that attaches to something besides the primary database, for example a local database instance.
- **preassembled** (*bool*) – If true, statements will be selected from the table of pre-assembled statements. Otherwise, they will be selected from the raw statements. Default is True.
- **with_support** (*bool*) – Choose whether to populate the supports and supported_by list attributes of the Statement objects. General results in slower queries.
- **with_evidence** (*bool*) – Choose whether or not to populate the evidence list attribute of the Statements. As with *with_support*, setting this to True will take longer.
- **fix_refs** (*bool*) – The paper refs within the evidence objects are not populated in the database, and thus must be filled using the relations in the database. If True (default), the *pmid* field of each Statement Evidence object is set to the correct PMIDs, or None if no PMID is available. If False, the *pmid* field defaults to the value populated by the reading system.

Return type list of Statements from the database corresponding to the query.

`indra_db.client.statements.get_statements_by_gene_role_type`(*agent_id=None*, *agent_ns='HGNC-SYMBOL'*, *role=None*, *stmt_type=None*, *count=1000*, *db=None*, *do_stmt_count=False*, *preassembled=True*, *fix_refs=True*, *with_evidence=True*, *with_support=False*, *essentials_only=False*)

Get statements from the DB by stmt type, agent, and/or agent role.

WARNING: This function will be removed in the future. Please look to indra_db.client.readonly.query and indra_db.client.principal.raw_statements for alternatives.

Parameters

- **agent_id** (*str*) – String representing the identifier of the agent from the given namespace. Note: if the agent namespace argument, *agent_ns*, is set to 'HGNC-SYMBOL', this function will treat *agent_id* as an HGNC gene symbol and perform an internal lookup of the corresponding HGNC ID. Default is 'HGNC-SYMBOL'.
- **agent_ns** (*str*) – Namespace for the identifier given in *agent_id*.
- **role** (*str*) – String corresponding to the role of the agent in the statement. Options are 'SUBJECT', 'OBJECT', or 'OTHER' (in the case of *Complex*, *SelfModification*, and *ActiveForm* Statements).
- **stmt_type** (*str*) – Name of the Statement class.

- **count** (*int (DEPRECATED)*) – Number of statements to retrieve in each batch (passed to `get_statements()`).
- **db** (`DatabaseManager`) – Optionally specify a database manager that attaches to something besides the primary database, for example a local databse instance.
- **do_stmt_count** (*bool (DEPRECATED)*) – Whether or not to perform an initial statement counting step to give more meaningful progress messages.
- **preassembled** (*bool (DEPRECATED)*) – If true, statements will be selected from the table of pre-assembled statements. Otherwise, they will be selected from the raw statements. Default is True.
- **with_support** (*bool (DEPRECATED)*) – Choose whether to populate the supports and supported_by list attributes of the Statement objects. Generally results in slower queries. DEFAULT IS CURRENTLY False.
- **with_evidence** (*bool*) – Choose whether or not to populate the evidence list attribute of the Statements. As with *with_support*, setting this to True will take longer.
- **fix_refs** (*bool (DEPRECATED)*) – The paper refs within the evidence objects are not populated in the database, and thus must be filled using the relations in the database. If True (default), the *pmid* field of each Statement Evidence object is set to the correct PMIDs, or None if no PMID is available. If False, the *pmid* field defaults to the value populated by the reading system.
- **essentials_only** (*bool (DEPRECATED)*) – Default is False. If True, retrieve only some metadata regarding the statements. Implicitly *with_support*, *with_evidence*, *fix_refs*, and *do_stmt_count* are all False, as none of the relevant features apply.

Returns

- *if essentials_only is False* – list of Statements from the database corresponding to the query.
- *else* – list of tuples containing basic data from the statements.

`indra_db.client.statements.`**`get_statements_by_paper`**(*id_list*, *id_type='pmid'*, *db=None*, *preassembled=True*)

Get the statements from a list of paper ids.

WARNING: This function will be removed in the future. Please look to indra_db.client.readonly.query and indra_db.client.principal.raw_statements for alternatives.

Parameters

- **id_list** (*list or set*) – A list of ints or strs that are ids of papers of type *id_type*.
- **id_type** (*str*) – The type of id used (default is pmid). Options include pmid, pmcid, doi, pii, url, or manuscript_id. Note that pmid is generally the best means of getting a paper.
- **db** (`DatabaseManager`) – Optionally specify a database manager that attaches to something besides the primary database, for example a local databse instance.
- **preassembled** (*bool*) – If True, statements will be selected from the table of pre-assembled statements. Otherwise, they will be selected from the raw statements. Default is True.

Returns **stmt_dict** – A dict of Statements from the database keyed the paper id given. Papers that yielded no statements are not included. If *preassembled* is True, there may be ids which were not present in the original dataset, and there may be a key None for statements that has evidence from refs that did not have that id_type of reference.

Return type dict

`indra_db.client.statements.`**`get_statements_from_hashes`**`(`*statement_hashes*, *preassembled=True*, *db=None*, ***kwargs*`)`

Retrieve statement objects given only statement hashes.

WARNING: This function will be removed in the future. Please look to indra_db.client.readonly.query and indra_db.client.principal.raw_statements for alternatives.

`indra_db.client.statements.`**`get_support`**`(`*statements*, *db=None*, *recursive=False*`)`

Populate the supports and supported_by lists of the given statements.

4.2.2 Pipeline Management CLI

This module creates a CLI for managing the pipelines used to update content and knowledge in the database, and move or transform that knowledge on a regular basis.

indra-db

INDRA Database Infrastructure CLI

The INDRA Database is both a physical database and an infrastructure for managing and updating the content of that physical database. This CLI is used for executing these management commands.

```
indra-db [OPTIONS] COMMAND [ARGS]...
```

content

Manage the text refs and content on the database.

```
indra-db content [OPTIONS] COMMAND [ARGS]...
```

list

List the current knowledge sources and their status.

```
indra-db content list [OPTIONS]
```

Options

`-l, --long`

Include a list of the most recently added content for all source types.

run

Upload/update text refs and content on the database.

Usage tasks are:

- upload: use if the knowledge bases have not yet been added.
- update: if they have been added, but need to be updated.

The currently available sources are “pubmed”, “pmc_oa”, and “manuscripts”.

```
indra-db content run [OPTIONS] {upload|update}
                     [[pubmed|pmc_oa|manuscripts]]...
```

Options

-c, --continuing
Continue uploading or updating, picking up where you left off.

-d, --debug
Run with debugging level output.

Arguments

TASK
Required argument

SOURCES
Optional argument(s)

dump

Manage the data dumps from Principal to files and Readonly.

```
indra-db dump [OPTIONS] COMMAND [ARGS]...
```

hierarchy

Dump hierarchy of Dumper classes to S3.

```
indra-db dump hierarchy [OPTIONS]
```

list

List existing dumps and their s3 paths.

State options:

- "started": get all dumps that have started (have "start.json" in them).
- "done": get all dumps that have finished (have "end.json" in them).
- "unfinished": get all dumps that have started but not finished.

If no option is given, all dumps will be listed.

```
indra-db dump list [OPTIONS] [[started|done|unfinished]]
```

Arguments

STATE
Optional argument

load-readonly

Load the readonly database with readonly schema dump.

```
indra-db dump load-readonly [OPTIONS]
```

Options

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

print-database-stats

Print the summary counts for the content on the database.

```
indra-db dump print-database-stats [OPTIONS]
```

run

Run dumps.

```
indra-db dump run [OPTIONS] COMMAND [ARGS]...
```

all

Generate new dumps and list existing dumps.

```
indra-db dump run all [OPTIONS]
```

Options

-c, --continuing
Indicate whether you want the job to continue building an existing dump corpus, or if you want to start a new one.

-d, --dump-only
Only generate the dumps on s3.

-l, --load-only
Only load a readonly dump from s3 into the given readonly database.

--delete-existing
Delete and restart an existing readonly schema in principal.

belief

Dump a dict of belief scores keyed by hash

```
indra-db dump run belief [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

end

Mark the dump as complete.

```
indra-db dump run end [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

full-pa-json

Dumps all statements found in FastRawPaLink as jsonl

```
indra-db dump run full-pa-json [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

full-pa-stmts

Dumps all statements found in FastRawPaLink as a pickle

```
indra-db dump run full-pa-stmts [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

mti-mesh-ids

Dump a mapping from Statement hashes to MeSH terms.

```
indra-db dump run mti-mesh-ids [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

principal-statistics

Dump a CSV of extensive counts of content in the principal database.

```
indra-db dump run principal-statistics [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

readonly

Generate the readonly schema, and dump it using pgdump.

```
indra-db dump run readonly [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

res-pos

Dumps a dict of dicts with residue/position data from Modifications

```
indra-db dump run res-pos [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

sif

Dumps a pandas dataframe of preassembled statements

```
indra-db dump run sif [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

source-count

Dumps a dict of dicts with source counts per source api per statement

```
indra-db dump run source-count [OPTIONS]
```

Options

-c, --continuing
Continue a partial dump, if applicable.

-d, --date-stamp <date_stamp>
Provide a datestamp with which to mark this dump. The default is same as the start dump from which this is built.

-f, --force
Run the build even if the dump file has already been produced.

--from-dump <from_dump>
Indicate a specific start dump from which to build. The default is the most recent.

start

Initialize the dump on s3, marking the start datetime of the dump.

```
indra-db dump run start [OPTIONS]
```

Options

-c, --continuing
Add this flag to only create a new start if an unfinished start does not already exist.

kb

Manage the Knowledge Bases used by the database.

```
indra-db kb [OPTIONS] COMMAND [ARGS]...
```

list

List the knowledge sources and their status.

```
indra-db kb list [OPTIONS]
```

run

Upload/update the knowledge bases used by the database.

Usage tasks are:

- upload: use if the knowledge bases have not yet been added.
- update: if they have been added, but need to be updated.

Specify which knowledge base sources to update by their name, e.g. "Pathway Commons" or "pc". If not specified, all sources will be updated.

```
indra-db kb run [OPTIONS] {upload|update} [SOURCES]...
```

Arguments

`TASK`
Required argument

`SOURCES`
Optional argument(s)

pa

Manage the preassembly pipeline.

```
indra-db pa [OPTIONS] COMMAND [ARGS]...
```

list

List the latest updates for each type of Statement.

```
indra-db pa list [OPTIONS]
```

Options

-r, --with-raw
Include the latest datetimes for raw statements of each type. This will take much longer.

run

Manage the indra_db preassembly.

Tasks:
- "create": populate the pa_statements table for the first time (this
  requires that the table be empty).
- "update": update the existing content in pa_statements with the latest
  from raw statements.

A project name is required to tag the AWS instances with a "project" tag.

```
indra-db pa run [OPTIONS] {create|update} [PROJECT_NAME]
```

Arguments

TASK
Required argument

PROJECT_NAME
Optional argument

pipeline-stats

Manage the pipeline stats gathered on s3.

All major upload and update pipelines have basic timeing and success-failure stats gather on them using the `DataGatherer` class wrapper.

These stats are displayed on the `/monitor` endpoint of the database service.

Tasks are:
- gather: gather the individual job JSONs into an aggregated file.

```
indra-db pipeline-stats [OPTIONS] {gather}
```

Arguments

TASK
Required argument

reading

Manage the reading jobs.

```
indra-db reading [OPTIONS] COMMAND [ARGS]...
```

list

List the readers and their most recent runs.

```
indra-db reading list [OPTIONS]
```

run

Manage the the reading of text content on AWS.

Tasks:
- "all": Read all the content available.
- "new": Read only the new content that has not been read.

```
indra-db reading run [OPTIONS] {all|new}
```

Options

-b, --buffer <buffer>
Set the number of buffer days to read prior to the most recent update. The default is 1 day.

--project-name <project_name>
Set the project name to be different from the config default.

Arguments

TASK
Required argument

run-local

Run reading locally, save the results on the database.

Tasks:
- "all": Read all the content available.
- "new": Read only the new content that has not been read.

```
indra-db reading run-local [OPTIONS] {all|new}
```

Options

-b, --buffer <buffer>
Set the number of buffer days to read prior to the most recent update. The default is 1 day.

-n, --num-procs <num_procs>
Select the number of processors to use.

Arguments

TASK
Required argument

xdd

Manage xDD runs.

```
indra-db xdd [OPTIONS] COMMAND [ARGS]...
```

run

Process the latest outputs from xDD.

```
indra-db xdd run [OPTIONS]
```

4.2.3 Pipeline CLI Implementations

Content (`indra_db.cli.content`)

The Content CLI manages the text content that is stored in the database. A parent class is defined, and managers for different sources (e.g. PubMed) can be defined by inheriting from this parent. This file is also used as the shell command to run updates of the content.

exception `indra_db.cli.content.`**`UploadError`**

class `indra_db.cli.content.`**`ContentManager`**

Abstract class for all upload/update managers.

This abstract class provides the api required for any object that is used to manage content between the database and the content.

`upload_text_content`(*db*, *data*)

Insert text content into the database using COPY.

`make_text_ref_str`(*tr*)

Make a string from a text ref using tr_cols.

`add_to_review`(*desc*, *msg*)

Add an entry to the review document.

`filter_text_refs`(*db*, *tr_data_set*, *primary_id_types=None*)

Try to reconcile the data we have with what's already on the db.

Note that this method is VERY slow in general, and therefore should be avoided whenever possible.

The process can be sped up considerably by multiple orders of magnitude if you specify a limited set of id types to query to get text refs. This does leave some possibility of missing relevant refs.

classmethod **`get_latest_update`**(*db*)

Get the date of the latest update.

`populate`(*db*)

A stub for the method used to initially populate the database.

`update`(*db*)

A stub for the method used to update the content on the database.

class `indra_db.cli.content.`**`Pubmed`**(**args*, *categories=None*, *tables=None*, *max_annotations=500000*, ***kwargs*)

Manager for the pubmed/medline content.

For relevant updates from NCBI on the managemetn and upkeep of the PubMed Abstract FTP server, see here:

https://www.nlm.nih.gov/databases/download/pubmed_medline.html

static **`fix_doi`**(*doi*)

Sometimes the doi is doubled (no idea why). Fix it.

`load_annotations`(*db*, *tr_data*)

Load annotations into the database.

`load_text_refs`(*db*, *tr_data*, *update_existing=False*)

Sanitize, update old, and upload new text refs.

`iter_contents`(*archives=None*)

Iterate over the files in the archive, yielding ref and content data.

Parameters **`archives`** (`Optional[Iterable[str]]`) – The names of the archive files from the FTP server to processes. If None, all available archives will be iterated over.

Yields

- **label** (*tuple*) – A key representing the particular XML: (XML File Name, Entry Number, Total Entries)
- **text_ref_dict** (*dict*) – A dictionary containing the text ref information.
- **text_content_dict** (*dict*) – A dictionary containing the text content information.

`load_files`(*db*, *files*, *continuing=False*, *carefully=False*, *log_update=True*)
Load the files in subdirectory indicated by `dirname`.

`dump_annotations`(*db*)
Dump all the annotations that have been saved so far.

`populate`(*db*, *continuing=False*)
Perform the initial input of the pubmed content into the database.

Parameters

- `db` (*`indra.db.DatabaseManager instance`*) – The database to which the data will be uploaded.
- `continuing` (*`bool`*) – If true, assume that we are picking up after an error, or otherwise continuing from an earlier process. This means we will skip over source files contained in the database. If false, all files will be read and parsed.

`update`(*db*)
Update the contents of the database with the latest articles.

`class indra_db.cli.content.PmcManager`(**args*, ***kwargs*)
Abstract class for uploaders of PMC content: PmcOA and Manuscripts.

`update`(*db*)
A stub for the method used to update the content on the database.

`static get_missing_pmids`(*db*, *tr_data*)
Try to get missing pmids using the pmc client.

`filter_text_content`(*db*, *tc_data*)
Filter the text content to identify pre-existing records.

`upload_batch`(*db*, *tr_data*, *tc_data*)
Add a batch of text refs and text content to the database.

`get_data_from_xml_str`(*xml_str*, *filename*)
Get the data out of the xml string.

`get_license`(*pmcid*)
Get the license for this pmcid.

`download_archive`(*archive*, *continuing=False*)
Download the archive.

`iter_xmls`(*archives=None*, *continuing=False*, *pmcid_set=None*)
Iterate over the xmls in the given archives.

Parameters

- `archives` (*`Optional[Iterable[str]]`*) – The names of the archive files from the FTP server to processes. If None, all available archives will be iterated over.
- `continuing` (*`Optional[Bool]`*) – If True, look for locally saved archives to parse, saving the time of downloading.

- **pmcid_set** (*Optional[set[str]]*) – A set of PMCIDs whose content you want returned from each archive. Many archives are massive repositories with 10s of thousands of papers in each, and only a fraction may need to be returned. Extracting and processing XMLs can be time consuming, so skipping those you don't need can really pay off!

Yields

- **label** (*Tuple*) – A key representing the particular XML: (Archive Name, Entry Number, Total Entries)
- **xml_name** (*str*) – The name of the XML file.
- **xml_str** (*str*) – The extracted XML string.

iter_contents(*archives=None*, *continuing=False*, *pmcid_set=None*)

Iterate over the files in the archive, yielding ref and content data.

Parameters

- **archives** (*Optional[Iterable[str]]*) – The names of the archive files from the FTP server to processes. If None, all available archives will be iterated over.
- **continuing** (*Optional[Bool]*) – If True, look for locally saved archives to parse, saving the time of downloading.
- **pmcid_set** (*Optional[set[str]]*) – A set of PMCIDs whose content you want returned from each archive. Many archives are massive repositories with 10s of thousands of papers in each, and only a fraction may need to be returned. Extracting and processing XMLs can be time consuming, so skipping those you don't need can really pay off!

Yields

- **label** (*tuple*) – A key representing the particular XML: (Archive Name, Entry Number, Total Entries)
- **text_ref_dict** (*dict*) – A dictionary containing the text ref information.
- **text_content_dict** (*dict*) – A dictionary containing the text content information.

upload_archives(*db*, *archives=None*, *continuing=False*, *pmcid_set=None*, *batch_size=10000*)

Do the grunt work of downloading and processing a list of archives.

Parameters

- **db** (*PrincipalDatabaseManager*) – A handle to the principal database.
- **archives** (*Optional[Iterable[str]]*) – An iterable of archive names from the FTP server.
- **continuing** (*bool*) – If True, best effort will be made to avoid repeating work already done using some cached files and downloaded archives. If False, it is assumed the caches are empty.
- **pmcid_set** (*set[str]*) – A set of PMC Ids to include from this list of archives.
- **batch_size** (*Optional[int]*) – Default is 10,000. The number of pieces of content to submit to the database at a time.

populate(*db*, *continuing=False*)

Perform the initial population of the pmc content into the database.

Parameters

- **db** (*indra.db.DatabaseManager instance*) – The database to which the data will be uploaded.

- **continuing** (*bool*) – If true, assume that we are picking up after an error, or otherwise continuing from an earlier process. This means we will skip over source files contained in the database. If false, all files will be read and parsed.

Returns **completed** – If True, an update was completed. Othewise, the updload was aborted for some reason, often because the upload was already completed at some earlier time.

Return type bool

`get_pmcid_file_dict()`

Get a dict keyed by PMCID mapping them to file names.

`get_csv_files(path)`

Get a list of CSV files from the FTP server.

`class indra_db.cli.content.PmcOA(*args, **kwargs)`

ContentManager for the pmc open access content.

For further details on the API, see the parent class: PmcManager.

`get_license(pmcid)`

Get the license for this pmcid.

`get_file_data()`

Retrieve the metadata provided by the FTP server for files.

`get_archives_after_date(min_date)`

Get the names of all single-article archives after the given date.

`update(db)`

A stub for the method used to update the content on the database.

`class indra_db.cli.content.Manuscripts(*args, **kwargs)`

ContentManager for the pmc manuscripts.

For further details on the API, see the parent class: PmcManager.

`get_license(pmcid)`

Get the license for this pmcid.

`get_file_data()`

Retrieve the metadata provided by the FTP server for files.

`get_tarname_from_filename(fname)`

Get the name of the tar file based on the file name (or a pmcid).

`enrich_textrefs(db)`

Method to add manuscript_ids to the text refs.

`update(db)`

Add any new content found in the archives.

Note that this is very much the same as populating for manuscripts, as there are no finer grained means of getting manuscripts than just looking through the massive archive files. We do check to see if there are any new listings in each files, minimizing the amount of time downloading and searching, however this will in general be the slowest of the update methods.

The continuing feature isn't implemented yet.

`class indra_db.cli.content.Elsevier(*args, **kwargs)`

Content manager for maintaining content from Elsevier.

`populate(db, n_procs=1, continuing=False)`

Load all available elsevier content for refs with no pmc content.

`update`(*db*, *n_procs=1*, *buffer_days=15*)
Load all available new elsevier content from new pmids.

Reading (`indra_db.cli.reading`)

The Reading CLI handles the reading of the text contend and the processing of those readings into statements. As with Content CLI, different reading pipelines can be handled by defining children of a parent class.

`exception indra_db.cli.reading.`**`ReadingUpdateError`**

`class indra_db.cli.reading.`**`ReadingManager`**(*reader_names*, *buffer_days=1*, *only_unread=False*)
Abstract class for managing the readings of the database.

Parameters

- **`reader_names`** (*`lsit [str]`*) – A list of the names of the readers to be used in a given run of reading.
- **`buffer_days`** (*`int`*) – The number of days before the previous update/initial upload to look for "new" content to be read. This prevents any issues with overlaps between the content upload pipeline and the reading pipeline.
- **`only_unread`** (*`bool`*) – Only read papers that have not been read (making the determination can be expensive).

`static get_latest_updates`(*db*)
Get the date of the latest update.

`read_all`(*db*, *reader_name*)
Perform an initial reading all content in the database (populate).

This must be defined in a child class.

`read_new`(*db*, *reader_name*)
Read only new content (update).

This must be defined in a child class.

`class indra_db.cli.reading.`**`BulkReadingManager`**(*reader_names*, *buffer_days=1*, *only_unread=False*)
An abstract class which defines methods required for reading in bulk.

This takes exactly the parameters used by *`ReadingManager`*.

`read_all`(*db*, *reader_name*)
Read everything available on the database.

`read_new`(*db*, *reader_name*)
Update the readings and raw statements in the database.

`class indra_db.cli.reading.`**`BulkAwsReadingManager`**(**args*, ***kwargs*)
This is the reading manager when updating using AWS Batch.

This takes all the parameters used by *`BulkReadingManager`*, and in addition:

Parameters **`project_name`** (*`str`*) – You can select a name for the project for which this reading is being run. This name has a default value set in your config file. The batch jobs used in reading will be tagged with this project name, for accounting purposes.

`class indra_db.cli.reading.`**`BulkLocalReadingManager`**(**args*, ***kwargs*)
This is the reading manager to be used when running reading locally.

This takes all the parameters used by *`BulkReadingManager`*, and in addition:

Parameters

- **n_proc** (*int*) – The number of processed to dedicate to reading. Note the some of the readers (e.g. REACH) do not always obey these restrictions.
- **verbose** (*bool*) – If True, more detailed logs will be printed. Default is False.

PreAssembly (`indra_db.cli.preassembly`)

The Preassembly CLI manages the preassembly pipeline, running deploying preassembly jobs to Batch.

`indra_db.cli.preassembly.`**`list_last_updates`**`(`*db*`)`

Return a dict of the most recent updates for each statement type.

`indra_db.cli.preassembly.`**`list_latest_raw_stmts`**`(`*db*`)`

Return a dict of the most recent new raw statement for each type.

`indra_db.cli.preassembly.`**`run_preassembly`**`(`*mode*, *project_name*`)`

Construct a submitter and begin submitting jobs to Batch for preassembly.

This function will determine which statement types need to be updated and how far back they go, and will create the appropriate `PreassemblySubmitter` instance, and run the jobs with pre-set parameters on statement types that need updating.

Parameters **project_name** (*str*) – This name is used to gag the various AWS resources used for accounting purposes.

Knowledge Bases (`indra_db.cli.knowledgebase`)

The INDRA Databases also derives much of its knowledge from external databases and other resources not extracted from plain text, referred to in this repo as “knowledge bases”, so as to avoid the ambiguity of “database”. This CLI handles the updates of those knowledge bases, each of which requires different handling.

`class` `indra_db.cli.knowledgebase.`**`TasManager`**

This manager handles retrieval and processing of the TAS dataset.

`class` `indra_db.cli.knowledgebase.`**`CBNManager`**`(`*archive_url=None*`)`

This manager handles retrieval and processing of CBN network files

`class` `indra_db.cli.knowledgebase.`**`HPRDManager`**

`class` `indra_db.cli.knowledgebase.`**`SignorManager`**

`class` `indra_db.cli.knowledgebase.`**`BiogridManager`**

`class` `indra_db.cli.knowledgebase.`**`BelLcManager`**

`class` `indra_db.cli.knowledgebase.`**`PathwayCommonsManager`**`(`**args*, ***kwargs*`)`

`class` `indra_db.cli.knowledgebase.`**`RlimspManager`**

`class` `indra_db.cli.knowledgebase.`**`TrrustManager`**

`class` `indra_db.cli.knowledgebase.`**`PhosphositeManager`**

`class` `indra_db.cli.knowledgebase.`**`CTDManager`**

`class` `indra_db.cli.knowledgebase.`**`VirHostNetManager`**

`class` `indra_db.cli.knowledgebase.`**`PhosphoElmManager`**

`class` `indra_db.cli.knowledgebase.`**`DrugBankManager`**

Static Dumps (indra_db.cli.dump)

This handles the generation of static dumps, including the readonly database from the principal database.

indra_db.cli.dump.**list_dumps**(*started=None*, *ended=None*)

List all dumps, optionally filtered by their status.

Parameters

- **started** (*Optional[bool]*) – If True, find dumps that have started. If False, find dumps that have NOT been started. If None, do not filter by start status.
- **ended** (*Optional[bool]*) – The same as *started*, but checking whether the dump is ended or not.

Returns

Each S3Path object contains the bucket and key prefix information for a set of dump files, e.g.

[S3Path(bigmech, indra-db/dumps/2020-07-16/), S3Path(bigmech, indra-db/dumps/2020-08-28/), S3Path(bigmech, indra-db/dumps/2020-09-18/), S3Path(bigmech, indra-db/dumps/2020-11-12/), S3Path(bigmech, indra-db/dumps/2020-11-13/)]

Return type list of S3Path objects

indra_db.cli.dump.**get_latest_dump_s3_path**(*dumper_name*)

Get the latest version of a dump file by the given name.

Searches dumps that have already been *started* and gets the full S3 file path for the latest version of the dump of that type (e.g. “sif”, “belief”, “source_count”, etc.)

Parameters **dumper_name** (*str*) – The standardized name for the dumper classes defined in this module, defined in the *name* class attribute of the dumper object. E.g., the standard dumper name “sif” can be obtained from `Sif.name`.

Return type *Union*[S3Path, None]

exception indra_db.cli.dump.**DumpOrderError**

class indra_db.cli.dump.**Start**(**args*, ***kwargs*)

Initialize the dump on s3, marking the start datetime of the dump.

load(*dump_path*)

Load manifest from the Start of the given dump path.

classmethod **from_date**(*dump_date: datetime.datetime*)

Select a dump based on the given datetime.

class indra_db.cli.dump.**PrincipalStats**(*start=None*, *date_stamp=None*, ***kwargs*)

Dump a CSV of extensive counts of content in the principal database.

class indra_db.cli.dump.**Belief**(*start=None*, *date_stamp=None*, ***kwargs*)

Dump a dict of belief scores keyed by hash

class indra_db.cli.dump.**Readonly**(*start=None*, *date_stamp=None*, ***kwargs*)

Generate the readonly schema, and dump it using pgdump.

class indra_db.cli.dump.**SourceCount**(*start*, *use_principal=True*, ***kwargs*)

Dumps a dict of dicts with source counts per source api per statement

class indra_db.cli.dump.**ResiduePosition**(*start*, *use_principal=True*, ***kwargs*)

Dumps a dict of dicts with residue/position data from Modifications

class indra_db.cli.dump.**FullPaStmts**(*start*, *use_principal=False*, ***kwargs*)

Dumps all statements found in FastRawPaLink as a pickle

class indra_db.cli.dump.**FullPaJson**(*start*, *use_principal=False*, ***kwargs*)

Dumps all statements found in FastRawPaLink as jsonl

class indra_db.cli.dump.**Sif**(*start*, *use_principal=False*, ***kwargs*)

Dumps a pandas dataframe of preassembled statements

class indra_db.cli.dump.**StatementHashMeshId**(*start*, *use_principal=False*, ***kwargs*)

Dump a mapping from Statement hashes to MeSH terms.

class indra_db.cli.dump.**End**(*start=None*, *date_stamp=None*, ***kwargs*)

Mark the dump as complete.

indra_db.cli.dump.**dump**(*principal_db*, *readonly_db=None*, *delete_existing=False*, *allow_continue=True*, *load_only=False*, *dump_only=False*)

Run the suite of dumps in the specified order.

Parameters

- **principal_db** (*indra_db.databases.PrincipalDatabaseManager*) – A handle to the principal database.
- **readonly_db** (*indra_db.databases.ReadonlyDatabaseManager*) – A handle to the readonly database. Optional when running dump only.
- **delete_existing** (*bool*) – If True, clear out the existing readonly build from the principal database. Otherwise it will be continued. (Default is False)
- **allow_continue** (*bool*) – If True, each step will assume that it may already have been done, and where possible the work will be picked up where it was left off. (Default is True)
- **load_only** (*bool*) – No new dumps will be created, but an existing dump will be used to populate the given readonly database. (Default is False)
- **dump_only** (*bool*) – Do not load a new readonly database, only produce the dump files on s3. (Default is False)

indra_db.cli.dump.**DumperChild**

alias of *indra_db.cli.dump.End*

4.2.4 Database Integrated Reading Tools

Here are defined the procedures for reading content on the database, stashing the reading outputs, and producing statements from the readings, and inserting those raw statements into the database.

The Database Readers (indra_db.reading.read_db)

A reader is defined as a python class which implements the machinery needed to process the text content we store, read it, and extract Statements from the reading results, storing the readings along the way. The reader must conform to a standard interface, which then allows readers to be run in a plug-and-play manner.

This module provides essential tools to run reading using indra's own database. This may also be run as a script; for details run: `python read_pmids_db --help`

exception indra_db.reading.read_db.**ReadDBError**

`indra_db.reading.read_db.generate_reading_id`(*tcid*, *reader_name*, *reader_version*)

Generate the unique reading ID hash from content ID, reader, and version.

The format of the hash is `AABBCCCCCCCCCC`, where `A` is the placeholder for the reader ID, `B` is the placeholder for the reader version integer, and `C` is reserved for the text content ID (it is loosely assumed we will not exceed 10^11 pieces of text content).

Parameters

- **tcid** (*str*) – The string-ified text content ID.
- **reader_name** (*str*) – The name of the reader. It must be one of the readers in *readers*.
- **reader_version** (*str*) – The version of the reader, which must be in the list of versions for the given `reader_name` in *reader_versions*.

class `indra_db.reading.read_db.DatabaseResultData`(*result*, *reading_id=None*, *db_info_id=None*, *indra_version=None*)

Contains metadata for statements, as well as the statement itself.

This, like ReadingData, is primarily designed for use with the database, carrying valuable information and methods for such.

Parameters

- **result** (*an indra Result instance*) – The result whose extra meta data this object encapsulates.
- **reading_id** (*int or None*) – The id number of the entry in the *readings* table of the database. None if no such id is available.
- **indra_version** (*str or None*) – Override the default indra version, which is the version of indra currently installed.

class `indra_db.reading.read_db.DatabaseStatementData`(**args*, ***kwargs*)

static `get_cols`()

Get the columns for the tuple returned by *make_tuple*.

`make_tuple`(*batch_id*)

Make a tuple for copying into the database.

class `indra_db.reading.read_db.DatabaseMeshRefData`(*result*, *reading_id=None*, *db_info_id=None*, *indra_version=None*)

static `get_cols`()

Get the columns for the tuple returned by *make_tuple*.

`make_tuple`(*batch_id*)

Make a tuple for copying into the database.

class `indra_db.reading.read_db.DatabaseReader`(*tcids*, *reader*, *verbose=True*, *reading_mode='unread'*, *rslt_mode='all'*, *batch_size=1000*, *db=None*, *n_proc=1*)

An class to run readings utilizing the database.

Parameters

- **tcids** (*iterable of ints*) – An iterable (set, list, tuple, generator, etc) of integers referring to the primary keys of text content in the database.
- **reader** (*Reader*) – An INDRA Reader object.
- **verbose** (*bool*) – Optional, default False - If True, log and print the output of the commandline reader utilities, if False, don't.

- **reading_mode** (*str : 'all', 'unread', or 'none'*) – Optional, default 'undread' - If 'all', read everything (generally slow); if 'unread', only read things that were unread, (the cache of old readings may still be used if *rslt_mode='all'* to get everything); if 'none', don't read, and only retrieve existing readings.
- **rslt_mode** (*str : 'all', 'unread', or 'none'*) – Optional, default 'all' - If 'all', produce results for all content for all readers. If the readings were already produced, they will be retrieved from the database if *read_mode* is 'none' or 'unread'. If this option is 'unread', only the newly produced readings will be processed. If 'none', no rs will be produced.
- **batch_size** (*int*) – Optional, default 1000 - The number of text content entries to be yielded by the database at a given time.
- **db** (*indra_db.DatabaseManager instance*) – Optional, default is None, in which case the primary database provided by *get_db('primary')* function is used. Used to interface with a different database.

dump_readings_to_db()
Put the reading output on the database.

dump_readings_to_pickle(*pickle_file*)
Dump the reading results into a pickle file.

get_readings()
Get the reading output for the given ids.

dump_results_to_db()
Upload the results to the database.

dump_results_to_pickle(*pickle_file*)
Dump the results into a pickle file.

get_results()
Convert the reader output into a list of ResultData instances.

make_results(*reading_data_list*, *num_proc=1*)
Convert a list of ReadingData instances into ResultData instances.

indra_db.reading.read_db.**process_content**(*text_content*)
Get the appropriate content object from the text content.

indra_db.reading.read_db.**construct_readers**(*reader_names*, ***kwargs*)
Construct the Reader objects from the names of the readers.

indra_db.reading.read_db.**read**(*db_reader*, *rslt_mode*, *reading_pickle*, *rslts_pickle*, *upload_readings*, *upload_rslts*)
Read for a single reader

indra_db.reading.read_db.**run_reading**(*readers*, *tcids*, *verbose=True*, *reading_mode='unread'*, *rslt_mode='all'*, *batch_size=1000*, *reading_pickle=None*, *stmts_pickle=None*, *upload_readings=True*, *upload_stmts=True*, *db=None*)
Run the reading with the given readers on the given text content ids.

The Database Script for Running on AWS (`indra_db.reading.read_db_aws`)

This is the script used to run reading on AWS Batch, generally run from an AWS Lambda function.

This script is intended to be run on an Amazon ECS container, so information for the job either needs to be provided in environment variables (e.g., the REACH version and path) or loaded from S3 (e.g., the list of PMIDs).

`indra_db.reading.read_db_aws.`**`is_trips_datestring`**(*s*)

Indicate whether a string has the form of a TRIPS log dir.

A Class to Manage and Monitor AWS Batch Jobs (`indra_db.reading.submitter`)

Allow a manager to monitor the Batch jobs to prevent runaway jobs, and smooth out job runs and submissions.

“This file acts as a script to run large batch jobs on AWS.

The key components are the DbReadingSubmitter class, and the submit_db_reading function. The function is provided as a shallow wrapper for backwards compatibility, and may eventually be removed. The preferred method for running large batches via the ipython, or from a python environment, is the following:

>> sub = DbReadingSubmitter(‘name_for_run’, [‘reach’, ‘sparser’]) >> sub.set_options(prioritize=True) >> sub.submit_reading(‘file/location/of/ids_to_read.txt’, 0, None, ids_per_job=1000) >> sub.watch_and_wait(idle_log_timeout=100, kill_on_timeout=True)

Additionally, this file may be run as a script. For details, run

bash$ python submit_reading_pipeline.py –help

In your favorite command line.

4.2.5 Database Integrated Preassembly Tools

The database runs incremental preassembly on the raw statements to generate the preassembled (PA) Statements. The code to accomplish this task is defined here, principally in *`DbPreassembler`*. This module also defines proceedures for running these jobs on AWS.

Database Preassembly (`indra_db.preassembly.preassemble_db`)

This module defines a class that manages preassembly for a given list of statement types on the local machine.

`exception` `indra_db.preassembly.preassemble_db.`**`IndraDBPreassemblyError`**

`exception` `indra_db.preassembly.preassemble_db.`**`UserQuit`**

`class` `indra_db.preassembly.preassemble_db.`**`DbPreassembler`**(*batch_size=10000*, *s3_cache=None*, *print_logs=False*, *stmt_type=None*, *yes_all=False*, *ontology=None*)

Class used to manage the preassembly pipeline

Parameters **`batch_size`** (*`int`*) – Select the maximum number of statements you wish to be handled at a time. In general, a larger batch size will somewhat be faster, but require much more memory.

`create_corpus`(*db*, *continuing=False*)

Initialize the table of preassembled statements.

This method will find the set of unique knowledge represented in the table of raw statements, and it will populate the table of preassembled statements (PAStatements/pa_statements), while maintaining links between the raw statements and their unique (pa) counterparts. Furthermore, the refinement/support relationships between unique statements will be found and recorded in the PASupportLinks/pa_support_links table.

For more detail on preassembly, see indra/preassembler/__init__.py

`supplement_corpus`(*db*, *continuing=False*)

Update the table of preassembled statements.

This method will take any new raw statements that have not yet been incorporated into the preassembled table, and use them to augment the preassembled table.

The resulting updated table is indistinguishable from the result you would achieve if you had simply re-run preassembly on _all_ the raw statements.

`indra_db.preassembly.preassemble_db.shash`(*s*)

Get the shallow hash of a statement.

`indra_db.preassembly.preassemble_db.make_graph`(*unique_stmts*, *match_key_maps*)

Create a networkx graph of the statement and their links.

A Class to Manage and Monitor AWS Batch Jobs (`indra_db.preassembly.submitter`)

Allow a manager to monitor the Batch jobs to prevent runaway jobs, and smooth out job runs and submissions.

4.2.6 Database Schemas

Here are defined the schemas for the principal and readonly databases, as well as some useful mixin classes.

Principal Database Schema (`indra_db.schemas.principal_schema`)

The Principal Schema

The Principal database is the core representation of our data, the ultimate authority on what we know. It is heavily optimized for the *input* and maintenance of our data.

class `indra_db.schemas.principal_schema.PrincipalSchema`(*Base*)

The Principal schema class organizes the table constructors.

The tables can be divided into various groups, with a clear order of creation for many of them.

Core Tables

First are the core tables representing our knowledge:

1. *text_ref*
2. *text_content*
3. *reading*
4. *db_info*
5. *raw_statements*
6. *raw_unique_links*
7. *pa_statements*
8. *pa_support_links*

Statement Attribute Tables

Then there are the tables that represent attributes of statements. The set of tables is identical for the raw statements:

- *raw_activity*
- *raw_agents*
- *raw_muts*
- *raw_mods*

and the preassembled statements:

- *pa_activity*
- *pa_agents*
- *pa_muts*
- *pa_mods*

Curation Table

This table is where we record the curations submitted by ourselves and our users, which we use to improve our results.

- *curations*

Ancillary Tables

We also have several tables that we use to keep track of processing metadata, and some artifacts useful in that processing.

- *updates*
- *source_file*
- *reading_updates*
- *preassembly_updates*
- *xdd_updates*
- *rejected_statements*
- *discarded_statements*

text_ref()

Represent a piece of text, as per its identifiers.

Each piece of text will be made available in different forms through different services, most commonly abstracts through pubmed and full text through pubmed central. However they are from the same *paper*, which has various different identifiers, such as pmids, pmcids, and dois.

We do our best to merge the different identifiers and for the most part each paper has exactly one text ref. Where that is not the case it is mostly impossible to automatically reconcile the different identifiers (this often has to do with inconsistent versioning of a paper and mixups over what is IDed).

Size: medium

Basic Columns

These are the core columns representing the different IDs we use to represent a paper.

- **id** `integer PRIMARY KEY`: The primary key of the TextRef entry. Elsewhere this is often referred to as a "text ref ID" or "trid" for short.
- **pmid** `varchar(20)`: The identifier from pubmed.
- **pmcid** `varchar(20)`: The identifier from PubMed Central (e.g. "PMC12345")
- **doi** `varchar(100)`: The ideally universal identifier.

- **pii** `varchar(250)`: The identifier used by Springer.
- **url** `varchar UNIQUE`: For sources found exclusively online (e.g. wikipedia) use their URL.
- **manuscript_id** `varchar(100) UNIQUE`: The ID assigned documents given to PMC author manuscripts.

Metadata Columns

In addition we also track some basic metadata about the entry and updates to the data in the table.

- **create_date** `timestamp without time zone`: The date the record was added.
- **last_updated** `timestamp without time zone`: The most recent time the record was edited.
- **pub_year** `integer`: The year the article was published, based on the first report we find (in order of PubMed, PMC, then PMC Manuscripts).

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **pmid-doi**: `UNIQUE(pmid, doi)`
- **pmid-pmcid**: `UNIQUE(pmid, pmcid)`
- **pmcid-doi**: `UNIQUE(pmcid, doi)`

Lookup Columns

Some columns are hard to look up when they are in their native string format, so they are processed and broken down into integer parts, as far as possible.

- **pmid_num** `integer`: the int-ified pmid, faster for lookup.
- **pmcid_num** `integer`: the int-portion of the PMCID, so "PMC12345" would here be 12345.
- **pmcid_version** `integer`: although rarely used, occasionally a PMC ID will have a version, indicated by a dot, e.g. PMC12345.3, in which case the "3" would be stored in this column.
- **doi_ns** `integer`: The DOI system works by assigning organizations (such as a journal) namespace IDs, and that organization is then responsible for maintaining a unique ID system internally. These namespaces are always numbers, and are stored here as such.
- **doi_id** `varchar`: The custom ID given by the publishing organization.

`mesh_ref_annotations()`

Represent the MeSH annotations of papers provided by PubMed.

Each abstract/entry in PubMed is accompanied by human-curated MeSH IDs indicating the topics of the paper. Each paper will have many IDs in general, so a separate table is used, liked to the *`text_ref`* table by an un-constrained PMID. This make insertion of the data easier because the custom TRIDs need not be retrieved to dump the mesh refs.

Size: large

Columns

- **id** `integer PRIMARY KEY`: The primary database-assigned ID of the row.
- **pmid_num** `integer NOT NULL`: The int-ified pmid that is used to link entries in this table with those in the *`text_ref`* table.
- **mesh_num** `` `integer NOT NULL ``: The intified MeSH ID (with the prefix removed). The **is_concept** column indicates whether the prefix was D (False) or C (True).

- **qual_num** `integer`: The qualifier number that is sometimes included with the annotation (Prefix Q).
- **major_topic** `boolean DEFAULT false`: The major topic flag indicates whether the ID describes a primary purpose of the paper.
- **is_concept** `boolean DEFAUL false`: Indicate whether the prefix was C (true) or D (false).

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **mesh-uniqueness**: `UNIQUE(pmid_num, mesh_num, qual_num, is_concept)`

`mti_ref_annotaions_test()`

Represent the MeSH annotations of abstracts as inferred by MTI.

MTI is a machine learned model that attempts to predict MeSH annotations on new un-annotated abstracts after training on the existing annotations.

Size: medium

Columns

- **id** `integer PRIMARY KEY`: The primary database-assigned ID of the row.
- **pmid_num** `integer NOT NULL`: The int-ified pmid that is used to link entries in this table with those in the *text_ref* table.
- **mesh_num** `` `integer NOT NULL ``: The intified MeSH ID (with the prefix removed). The **is_concept** column indicates whether the prefix was D (False) or C (True).
- **qual_num** `integer`: The qualifier number that is sometimes included with the annotation (Prefix Q).
- **major_topic** `boolean DEFAULT false`: The major topic flag indicates whether the ID describes a primary purpose of the paper.
- **is_concept** `boolean DEFAUL false`: Indicate whether the prefix was C (true) or D (false).

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **mesh-uniqueness**: `UNIQUE(pmid_num, mesh_num, qual_num, is_concept)`

`text_content()`

Represent the content of a text retrieved from a particular source.

For each paper as a logical entity, there are many places where you can acquire the actual article or parts of it. For example you can get an abstract from PubMed for most content, and for a minority subset you can get full text from PubMed Central, either their Open-Access corpus or their author's Manuscripts.

Both the text itself and the metadata for the source of the text are represented in this table.

Size: large

Basic Columns

- **id** `integer PRIMARY KEY`: The auto-generated primary key of the table. These are elsewhere called Text Content IDs, or TCIDs.
- **text_ref_id** `integer NOT NULL`: A foreign-key constrained reference to the appropriate entry in the *text_ref* table.
- **source** `varchar(250) NOT NULL`: The name of the source, e.g. "pubmed" or "pmc_oa". The list of content names can be found in the class attributes in content managers.

- **format** `varchar(250) NOT NULL`: The file format of the content, e.g. "XML" or "TEXT".
- **text_type** `varchar(250) NOT NULL`: The type of the text, e.g. "abstract" of "fulltext".
- **preprint** `boolean`: Indicate whether the content is from a preprint.
- **license** [`varchar`]: Record the license that applies to the content.
- **content** `bytea`: The raw compressed bytes of the content.

Metadata Columns

- **insert_data** `timestamp without time zone`: The date the record was added.
- **last_updated** `timestamp without time zone`: The most recent time the record was edited.

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **content-uniqueness**: `UNIQUE(text_ref_id, source, format, text_type)`

reading()

Represent a reading of a piece of text.

We have multiple readers and of course many thousands of pieces of text content. Each entry in this table applies to a given reader applied to a given pieces of content.

As such, the primary ID is a hash constructed from the text content ID prepended with integers that are assigned to each reader-reader version pair. The function *generate_reading_id* implements the particular process used. The reader numbers are assigned in the *readers* global, and the reader version number is the index of the version listed for the given reader in the *reader_versions* dictionary in the same module.

Size: very large

Basic Columns

- **id** `bigint PRIMARY KEY`: A hash ID constructed from a reader number, reader version number, and the text content ID of the content that was read.
- **text_content_id** `integer NOT NULL`: A foreign-key constrained reference to the appropriate entry in the *text_content* table.
- **batch_id** `integer NOT NULL`: A simple random integer (not unique) that is assigned each batch of inserted readings. It is used in the moments after the insert to easily retrieve the content that was just added, potentially plus some extra.
- **reader** `varchar(20) NOT NULL`: The name of the reader, e.g. "REACH" or "SPARSER".
- **reader_version** `varchar(20) NOT NULL`: The version of the reader, which may be any arbitrary string in principle. This allows each reader to define its own versioning scheme.
- **format** `varchar(20) NOT NULL`: The file format of the reading result, e.g. "XML" or "JSON".
- **bytes** `bytea`: The raw compressed bytes of the reading result.

Metadata Columns

- **create_date** `timestamp without time zone`: The date the record was added.
- **last_updated** `timestamp without time zone`: The most recent time the record was edited.

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **reading-uniqeness**: `UNIQUE(text_content_id, reader, reader_version)`

`db_info()`

Represent the provenance and metadata for an external knowledge base.

INDRA DB takes content not just from our own readings but also merges that with many pre-existing knowledge bases, many of them human curated. These knowledge bases are defined and managed by classes contained in `knowledgebase_manager`.

No real data is contained in this column, simply records of which knowledge bases have been added, updated, and when.

Size: very small

Basic Columns

- **id** `integer PRIMARY KEY`: A database-assigned integer unique ID for each database entry. These are elsewhere referred to as *db_info_ids* or *dbids*.
- **db_name** `varchar NOT NULL`: A short lowercase string that is used internally to identify the knowledge base, e.g. "pc" for Pathway Commons.
- **db_full_name** `varchar NOT NULL`: The full name of the knowledge base, neatly formatted, e.g. "Pathway Commons".
- **source_api** `varchar NOT NULL`: The indra source API that was used to extract Statements from the knowledge base, e.g. "biopax".

Metadata Columns

- **create_date** `timestamp without time zone`: The date the record was added.
- **last_updated** `timestamp without time zone`: The most recent time the record was edited.

`raw_statements()`

Represent Statements exactly as extracted by their source apis.

INDRA Defines several source APIs for different file types from which we can extract INDRA Statements. The goal of these APIs is primarily to accurately convey the contents of the files, and minimal fixes are made at this stage (e.g. grounding is saved for preassembly).

Thus this table contains statements that are considered "messy" in two key ways:

- they have a lot of repetition of information, and
- they have whatever grounding the original source gave them.

However these Statements also have the Evidence object JSON contained in their **json** column, and this Evidence information is **NOT** copied into the *`pa_statements`* table, which allows for a flexible incremental updates. A "lateral join" on this table can be used to get the first N evidence associated with each PA Statement.

Size: very large

Basic Columns

- **id** `integer PRIMARY KEY`: A database-assigned integer unique ID for each database entry. These are elsewhere referred to as "Statement ID"s, or "sid"s.
- **uuid** `varchar UNIQUE NOT NULL`: A UUID generated when a Statement object is first created. This can be used for tracking particular objects through the code.
- **batch_id** `integer NOT NULL`: A simple random integer (not unique) that is assigned each batch of inserted Statements. It is used in the moments after the insert to easily retrieve the content that was just added, potentially plus some extra.

- **mk_hash** `bigint NOT NULL`: A hash of the `matches_key` of a Statement. This should be unique for any statement containing the same information.
- **text_hash** `bigint`: A hash of a the evidence text, used to detect exact duplicate Statements (same information from the same exact source, right down to the text) that sometimes occur due to bugs
- **source_hash** `bigint NOT NULL`: A hash of the source information.
- **db_info_id** `integer`: A foreign key into the *db_info* table, for those statements that come from knowledge bases.
- **reading_id** `bigint`: A foreign key into the *reading* table, for those statements that come from a reading.
- **type** `varchar(100) NOT NULL`: The type of the Statement, e.g. "Phosphorylation".
- **indra_version** `varchar(100) NOT NULL`: The version of INDRA that was used to generate this Statement, specifically as returned by `indra.util.get_version.get_version()`.
- **json** `bytea NOT NULL`: The bytes of the Statement JSON (including exactly **one** Evidence JSON)

Metadata Columns

- **create_date** `timestamp without time zone`: The date the Statement was added.

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **reading_raw_statement_uniqueness**: `UNIQUE(mk_hash, text_hash, reading_id)`
- **db_info_raw_statement_uniqueness**: `UNIQUE(mk_hash, source_hash, db_info_id)`

`raw_activity()`

Represent the activity of a raw statement (an ActiveForm).

`raw_agents()`

Represent an identifier for an agent of a raw statement.

`raw_mods()`

Represent a modification of an agent of a raw statement.

`raw_muts()`

Represent a mutation of an agent of a raw statement.

`raw_unique_links()`

Represent links between raw statements and preassembled statements.

Each preassembled statement is constructed from multiple raw statements, in general. This maps each *pa_statement* to the *raw statements* that were merged to form it. It is through this table that evidence can be gathered for pa_statements.

The astute reader may note that the *raw_statements*-to -*pa_statement* relationship is many-to-one, which can be represented simply using a foreign-key in the "many" table, in this case *raw_statements*. This is not done because the *pa_statement* does not, in general, exist when the *raw_statement* is added to the database.

Constructed as it is, these links can be copied in bulk during preassembly, as opposed to having to modify as many as a million entries with a newly created foreign-key map.

Size: large

Basic Columns

- **id** `integer PRIMARY KEY`: A database-assigned integer unique ID for each database entry.

- **raw_stmt_id** `integer NOT NULL REFERENCES raw_statements(id)`: The Raw Statement ID foreign key to the *raw_statements* table.
- **pa_stmt_mk_hash** `bigint NOT NULL REFERENCES pa_statements(mk_hash)`: The PA Statement matches-key hash foreign key to the *pa_statements* table.

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **stmt-link-uniqueness**: `UNIQUE(raw_stmt_id, pa_stmt_mk_hash)`

`pa_statements()`

Represent preassembled statements.

> Preassmebled Statements are generated from Raw Statements using INDRA's preassembly tools. Specifically:
>
> - agents are grounded,
> - agent groundings are disambiguated (using adeft),
> - sites are fixed (using protmapper),
> - and finally, repeated information is consolidated, for example `Phosphorylation(MEK(), ERK())` is represented only once in this corpus, with links to the many instances that information was extracted, which are stored in the *raw_statements* table.

Each entry is linked back to the (in general multiple) raw statements it was derived from in the *raw_unique_links* table.

Size: medium large

Basic Columns

- **mk_hash** `bigint PRIMARY KEY`: a hash of the statement matches key, which is unique for the _knowledge_ of the Statement.
- **matches_key** `varchar NOT NULL`: The matches-key that was hashed.
- **uuid** `varchar UNIQUE NOT NULL`: A UUID generated when a Statement object is first created. This can be used for tracking particular objects through the code. The UUID is distinct from any of the raw statement UUIDs that compose this Statement.
- **type** `varchar(100) NOT NULL`: The type of the Statement, e.g. "Phosphorylation".
- **indra_version** `varchar(100) NOT NULL`: The version of INDRA that was used to generate this Statement, specifically as returned by `indra.util.get_version.get_version()`.
- **json** `bytea NOT NULL`: The bytes of the Statement JSON (including exactly **one** Evidence JSON)

Metadata Columns

- **create_date** `timestamp without time zone`: The date the Statement was added.

`pa_support_links()`

Represent the links of support calculated during preassembly.

In INDRA, we look for cases where more specific Statements may lend support to more general Statements, and potentially vice versa, to better gauge whether an extraction is reliable.

Size: large

Basic Columns

- **id** `integer PRIMARY KEY`: A database-assigned integer unique ID for each database entry.

- **supporting_mk_hash** `bigint NOT NULL REFERENCES pa_statements(mk_hash)`: A foreign key to the PA Statement that is giving the support (that is, the more specific Statement).
- **supported_mk_hash** `bigint NOT NULL REFERENCES pa_statements(mk_hash)`: A foreign key to the PA Statement that is given the support (that is, the more generic Statement).

Constraints

Postgres is extremely efficient at detecting conflicts, and we use this to help ensure our entries do not have any duplicates.

- **pa_support_links_link_uniqueness**: `UNIQUE(supporting_mk_hash, supported_mk_hash)`

`pa_activity()`

Represent the activity of a preassembled Statement.

`pa_agents()`

Represent an identifier for an agent of a preassembled statement.

`pa_mods()`

Represent a modification of an agent of a preassembled statement.

`pa_muts()`

Represent a mutation of an agent of a preassembled statement.

`curations()`

Represent the curations of our content.

At various points in our APIs and UIs it is possible to curate the content we have extracted, recording whether it is an accurate extraction from the source text, and if not the reason why.

Size: small

Basic Columns

- **id** `integer PRIMARY KEY`: A database-assigned integer unique ID for each database entry.
- **pa_hash** `bigint REFERENCES pa_statements(mk_hash)`: A reference into the *`pa_statements`* table to the the pa statement whose evidence was curated.
- **source_hash** `bigint`: A hash that represents the source of this Statement (e.g. reader and piece of content).
- **tag** `varchar`: A text code indicating the type of error curated. The domain of these strings is regulated in code elsewhere.
- **text** `varchar`: A free-form text description by the curator of what they think went wrong (or right).
- **curator** `varchar NOT NULL`: The identity of the curator. This has elsewhere been standardized to be their email.
- **auth_id** `varchar`: [deprecated]
- **source** `varchar`: A string indicating where this curation originated, e.g. "DB REST API" for the INDRA Database REST service.
- **ip** `inet`: The IP address from which the curation was submitted.
- **date** `timestamp without time zone`: The date the curation was added.
- **pa_json** `jsonb`: the preassembled Statement JSON that was curated.
- **ev_json** `jsonb`: the Evidence JSON that was curated (including the text).

`source_file()`

Record the pubmed source file that was processed.

updates()
Record when text ref and content updates were performed.

reading_updates()
Record runs of the readers on the content we have found.

xdd_updates()
Record the times we process dumps from xDD.

rejected_statements()
Represent raw statements that were rejected.

discarded_statements()
Record the reasons for which some statements were discarded.

preassembly_updates()
Record updates of the preassembled corpus.

Readonly Database Schema (indra_db.schemas.readonly_schema)

Defines the *get_schema* function for the readonly database, which is used by external services to access the Statement knowledge we acquire.

class indra_db.schemas.readonly_schema.**ReadonlySchema**(*Base*)
Schema for the Readonly database.

We use a readonly database to allow fast and efficient load of data, and to add a layer of separation between the processes of updating the content of the database and accessing the content of the database. However, it is not practical to have the views created through sqlalchemy: instead they are generated and updated manually (or by other non-sqlalchemy scripts).

Before building these tables, the `belief` table must already have been loaded into the readonly database.

The following views must be built in this specific order (temp):

1. *raw_stmt_src*
2. *fast_raw_pa_link*
3. *pa_agent_counts*
4. (*pa_stmt_src*)
5. *evidence_counts*
6. *reading_ref_link*
7. (*pa_ref_link*)
8. (*mesh_terms*)
9. (*mesh_concepts*)
10. (*hash_pmid_counts*)
11. *mesh_term_ref_counts*
12. *mesh_concept_ref_counts*
13. *raw_stmt_mesh_terms*
14. *raw_stmt_mesh_concepts*
15. (*pa_meta*)
16. *source_meta*

17. *text_meta*
18. *name_meta*
19. *other_meta*
20. *mesh_term_meta*
21. *mesh_concept_meta*
22. agent_interaction

Note that the order of views below is determined not by the above order but by constraints imposed by use-case.

Meta Tables

Any table that has "meta" in the name is intended as a primary lookup table. This means it will have both the data indicated in the name of the table, such at (agent) "text", (agent) "name", or "source", but also a collection of columns with metadata essential for sorting and grouping of hashes:

- Sorting:
  - **belief**
  - **ev_count**
  - **agent_count**
- Grouping:
  - **type_num**
  - **activity**
  - **is_active**

Temporary Tables

There are some intermediate results that it is worthwhile to calculate and store for future table construction. Sometimes these were once permanent tables but are no longer used for their own sake, and it was simply simpler to delete them after their derivatives were completed. In other cases the temporary tables are more principled: created because many future tables draw on them and using a "with" clause for each one would be impractical.

Whatever the reason, deleting the temporary tables greatly reduces the size of the readonly database. Such tables are marked in with "(temp)" at the beginning of their doc string.

belief()

The belief of preassembled statements, keyed by hash.

Columns

- **mk_hash** `bigint`
- **belief** `real`

Indices

- **mk_hash**

evidence_counts()

The evidence counts of pa statements, keyed by hash.

Columns

- **mk_hash** `bigint`
- **ev_count** `integer`

Indices

- **mk_hash**

`reading_ref_link()`

The source metadata for readings, keyed by reading ID.

Columns

- **trid** `integer`
- **pmid** `varchar(20)`
- **pmid_num** `integer`
- **pmcid** `varchar(20)`
- **pmcid_num** `integer`
- **pmcid_version** `integer`
- **doi** `varchar(100)`
- **doi_ns** `integer`
- **doi_id** `varchar`
- **pii** `varchar(250)`
- **url** `varchar(250)`
- **manuscript_id** `varchar(100)`
- **tcid** `integer`
- **source** `varchar(250)`
- **rid** `integer`
- **reader** `varchar(20)`

Indices

- **rid**
- **pmid**
- **pmid_num**
- **pmcid**
- **pmcid_num**
- **doi**
- **doi_ns**
- **doi_id**
- **manuscript_id**
- **tcid**
- **trid**

`fast_raw_pa_link()`

Join of PA JSONs and Raw JSONs for faster lookup.

Columns

- **id** `integer`
- **raw_json** `bytea`

- **reading_id** `bigint`
- **db_info_id** `integer`
- **mk_hash** `bigint`
- **pa_json** `bytea`
- **type_num** `smallint`
- **src** `varchar`

Indices

- **mk_hash**
- **reading_id**
- **db_info_id**
- **src**

`pa_agent_counts()`

The number of agents for each Statement, keyed by hash.

Columns

- **mk_hash** `bigint`
- **agent_count** `integer`

Indices

- **mk_hash**

`raw_stmt_src()`

The source (e.g. reach, pc) of each raw statement, keyed by SID.

Columns

- **sid** `integer`
- **src** `varchar`

Indices

- **sid**
- **src**

`pa_stmt_src()`

(temp) The number of evidence from each source for a PA Statement.

This table is constructed by forming a column for every source short name present in the *`raw_stmt_src`*.

Columns

- **mk_hash** `bigint`
- …one column for each source… `integer`

Indices

- **mk_hash**

`pa_ref_link()`

(temp) A quick-lookup from mk_hash to basic text ref data.

Columns

- **mk_hash** `bigint`
- **trid** `integer`
- **pmid_num** `varchar`
- **pmcid_num** `varchar`
- **source** `varchar`
- **reader** `varchar`

Indices

- **mk_hash**
- **trid**
- **pmid_num**

`mesh_terms()`

(temp) All mesh annotations with D prefix, keyed by PMID int.

Columns

- **mesh_num** `integer`
- **pmid_num** `integer`

Indices

- **pmid_num**

`mesh_concepts()`

(temp) All mesh annotations with C prefix, keyed by PMID int.

Columns

- **mesh_num** `integer`
- **pmid_num** `integer`

Indices

- **pmid_num**

`hash_pmid_counts()`

(temp) The number of pmids for each PA Statement, keyed by hash.

Columns

- **mk_hash** `bigint`
- **pmid_count** `integer`

Indices

- **mk_hash**

`mesh_term_ref_counts()`

The D-type mesh IDs with pmid and ref counts, keyed by hash and mesh.

Columns

- **mk_hash** `bigint`
- **mesh_num** `integer`
- **ref_count** `integer`

- **pmid_count** `integer`

Indices

- **mesh_num**
- **mk_hash**

`mesh_concept_ref_counts()`

The C-type mesh IDs with pmid and ref counts, keyed by hash and mesh.

Columns

- **mk_hash** `bigint`
- **mesh_num** `integer`
- **ref_count** `integer`
- **pmid_count** `integer`

Indices

- **mesh_num**
- **mk_hash**

`raw_stmt_mesh_terms()`

The D-type mesh number raw statement ID mapping.

Columns

- **sid** `integer`
- **mesh_num** `integer`

Indices

- **sid**
- **mesh_num**

`raw_stmt_mesh_concepts()`

The C-type mesh number raw statement ID mapping.

Columns

- **sid** `integer`
- **mesh_num** `integer`

Indices

- **sid**
- **mesh_num**

`pa_meta()`

(temp) The metadata most valuable for querying PA Statements.

This table is used to generate the more scope-limited *`name_meta`*, *`text_meta`*, and *`other_meta`*. The reason is that NAME and TEXT (in particular) agent groundings are vastly overrepresented.

Columns

- **ag_id** `integer`
- **ag_num** `integer`
- **db_name** `varchar`

- **db_id** `varchar`
- **role_num** `smallint`
- **type_num** `smallint`
- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`
- **is_complex_dup** `boolean`

Indices

- **db_name**
- **mk_hash**

`source_meta()`

All the source-related metadata condensed using JSONB, keyed by hash.

Columns

- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **num_srcs** `integer`
- **src_json** `json`
- **only_src** `varchar`
- **has_rd** `boolean`
- **has_db** `boolean`
- **type_num** `smallint`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`

Indices

- **mk_hash**
- **only_src**
- **activity**
- **type_num**
- **num_srcs**

`text_meta()`

The metadata most valuable for querying PA Statements by agent TEXT.

This table is generated from *pa_meta*, because TEXT is extremely overrepresented among agent groundings. Removing these and NAMEs from the "OTHER" efficiently narrows the search very rapidly, and for the larger sets of NAME and TEXT removes an index-search.

Columns

- **ag_id** `integer`
- **ag_num** `integer`
- **db_id** `varchar`
- **role_num** `smallint`
- **type_num** `smallint`
- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`
- **is_complex_dup** `boolean`

Indices

- **mk_hash**
- **db_id**
- **type_num**
- **activity**

`name_meta()`

The metadata most valuable for querying PA Statements by agent NAME.

This table is generated from *pa_meta*, because NAME is overrepresented among agent groundings. Removing these and NAMEs from the "OTHER" efficiently narrows the search very rapidly, and for the larger sets of NAME and TEXT removes an index-search.

Columns

- **ag_id** `integer`
- **ag_num** `integer`
- **db_id** `varchar`
- **role_num** `smallint`
- **type_num** `smallint`
- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **activity** `varchar`

- **is_active** `boolean`
- **agent_count** `integer`
- **is_complex_dup** `boolean`

Indices

- **mk_hash**
- **db_id**
- **type_num**
- **activity**

`other_meta()`

The metadata most valuable for querying PA Statements.

This table is a copy of *pa_meta* with rows with agent groundings besides NAME and TEXT removed.

Columns

- **ag_id** `integer`
- **ag_num** `integer`
- **db_name** `varchar`
- **db_id** `varchar`
- **role_num** `smallint`
- **type_num** `smallint`
- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`
- **is_complex_dup** `boolean`

Indices

- **mk_hash**
- **db_name**
- **db_id**
- **type_num**
- **activity**

`mesh_term_meta()`

A lookup for hashes by D-type mesh IDs.

Columns

- **mk_hash** `bigint`
- **mesh_num** `integer`
- **tr_count** `integer`

- **ev_count** `integer`
- **belief** `real`
- **type_num** `smallint`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`

Indices

- **mk_hash**
- **type_num**
- **activity**

`mesh_concept_meta()`

A lookup for hashes by C-type mesh IDs.

Columns

- **mk_hash** `bigint`
- **mesh_num** `integer`
- **tr_count** `integer`
- **ev_count** `integer`
- **belief** `real`
- **type_num** `smallint`
- **activity** `varchar`
- **is_active** `boolean`
- **agent_count** `integer`

Indices

- **mk_hash**
- **type_num**
- **activity**

`agent_interactions()`

Agent and type data in simple JSONs for rapid lookup, keyed by hash.

This table is used for retrieving interactions, agent pairs, and relations (any kind of return that is more generic than full Statements).

Columns

- **mk_hash** `bigint`
- **ev_count** `integer`
- **belief** `real`
- **type_num** `smallint`
- **activity** `varchar`
- **is_active** `boolean`

- **agent_count** `integer`
- **agent_json** `jsonb`
- **src_json** `jsonb`
- **is_complex_dup** `boolean`

Indices

- **mk_hash**
- **agent_json**
- **type_num**

Class Mix-ins (`indra_db.schemas.mixins`)

This defines class mixins that are used to add general features to SQLAlchemy table objects via multiple inheritance.

`exception indra_db.schemas.mixins.`**`DbIndexError`**

`class indra_db.schemas.mixins.`**`IndraDBTableMetaClass`**`(`**args*, ***kwargs*`)`

This serves as a meta class for all tables, allowing *str* to be useful.

In particular, this makes it so that the string gives a representation of the SQL table, including columns.

`class indra_db.schemas.mixins.`**`IndraDBRefTable`**

Define an API and methods for a table of text references.

`classmethod` **`pmid_in`**`(`*pmid_list*, *filter_ids=False*`)`

Get sqlalchemy clauses for entries IN a list of pmids.

`classmethod` **`pmid_notin`**`(`*pmid_list*, *filter_ids=False*`)`

Get sqlalchemy clauses for entries NOT IN a list of pmids.

`classmethod` **`pmcid_in`**`(`*pmcid_list*, *filter_ids=False*`)`

Get the sqlalchemy clauses for entries IN a list of pmcids.

`classmethod` **`pmcid_notin`**`(`*pmcid_list*, *filter_ids=False*`)`

Get the sqlalchemy clause for entries NOT IN a list of pmcids.

`classmethod` **`doi_in`**`(`*doi_list*, *filter_ids=False*`)`

Get clause for looking up entities IN a list of dois.

`classmethod` **`doi_notin`**`(`*doi_list*, *filter_ids=False*`)`

Get clause for looking up entities NOT IN a list of dois.

`classmethod` **`has_ref`**`(`*id_type*, *id_list*, *filter_ids=False*`)`

Get clause for entries IN the given ID list.

`classmethod` **`not_has_ref`**`(`*id_type*, *id_list*, *filter_ids=False*`)`

Get clause for entries NOT IN the given ID list

`get_ref_dict``()`

Return the refs as a dictionary keyed by type.

`class indra_db.schemas.mixins.`**`Schema`**`(`*Base*`)`

General class for schemas

Indexes (`indra_db.schemas.indexes`)

This defines the classes needed to create and maintain indices in the database, the other part of the infrastructure of which is included in the *IndraDBTable* class mixin definition.

4.2.7 Utilities

Here live the more mundane and backend utilities used throughout other modules of the codebase, and potentially elsewhere, although they are not intended for external use in general. Several more-or-less bespoke scripts are also stored here.

Database Session Constructors (`indra_db.util.constructors`)

Constructors to get interfaces to the different databases, selecting among the various physical instances defined in the config file.

`indra_db.util.constructors.`**`get_db`**`(`*db_label*, *protected=False*`)`

Get a db instance base on it's name in the config or env.

If the label does not exist or the database labeled can't be reached, None is returned.

`indra_db.util.constructors.`**`get_primary_db`**`(`*force_new=False*`)`

Get a DatabaseManager instance for the primary database host.

The primary database host is defined in the defaults.txt file, or in a file given by the environment variable DEFAULTS_FILE. Alternatively, it may be defined by the INDRADBPRIMARY environment variable. If none of the above are specified, this function will raise an exception.

Note: by default, calling this function twice will return the same *DatabaseManager* instance. In other words:

```
db1 = get_primary_db()
db2 = get_primary_db()
db1 is db2
```

This means also that, for example *db1.select_one(db2.TextRef)* will work, in the above context.

It is still recommended that when creating a script or function, or other general application, you should not rely on this feature to get your access to the database, as it can make substituting a different database host both complicated and messy. Rather, a database instance should be explicitly passed between different users as is done in *get_statements_by_gene_role_type* function's call to *get_statements* in *indra.db.query_db_stmts*.

Parameters **`force_new`** (*`bool`*) – If true, a new instance will be created and returned, regardless of whether there is an existing instance or not. Default is False, so that if this function has been called before within the global scope, a the instance that was first created will be returned.

Returns **primary_db** – An instance of the database manager that is attached to the primary database.

Return type `DatabaseManager`

`indra_db.util.constructors.`**`get_ro`**`(`*ro_label*, *protected=True*`)`

Get a readonly database instance, based on its name.

If the label does not exist or the database labeled can't be reached, None is returned.

`indra_db.util.constructors.`**`get_ro_host`**`(`*ro_label*`)`

Get the host of the current readonly database.

Scripts to Get Content (`indra_db.util.content_scripts`)

General scripts for getting content by various IDs.

`indra_db.util.content_scripts.`**`get_stmts_with_agent_text_like`**(*pattern*, *filter_genes=False*, *db=None*)

Get statement ids with agent with rawtext matching pattern

Parameters

- **pattern** (*str*) – a pattern understood by sqlalchemy's like operator. For example '__' for two letter agents
- **filter_genes** (*Optional[bool]*) – if True, only returns map for agent texts for which there is at least one HGNC grounding in the database. Default: False
- **db** (Optional[`DatabaseManager`]) – User has the option to pass in a database manager. If None the primary database is used. Default: None

Returns dict mapping agent texts to statement ids. agent text are those matching the input pattern. Each agent text maps to the list of statement ids for statements containing an agent with that TEXT in its db_refs

Return type dict

`indra_db.util.content_scripts.`**`get_text_content_from_stmt_ids`**(*stmt_ids*, *db=None*)

Get text content for statements from a list of ids

Gets the fulltext if it is available, even if the statement came from an abstract.

Parameters

- **stmt_ids** (*list of str*) –
- **db** (Optional[`DatabaseManager`]) – User has the option to pass in a database manager. If None the primary database is used. Default: None

Returns **ref_dict** – dict mapping statement ids to identifiers for pieces of content. These identifiers take the form `<text_ref_id>/<source>/<text_type>'. No entries exist for statements with no associated text content (these typically come from databases)

Return type dict

text_dict: dict dict mapping content identifiers used as values in the ref_dict to best available text content. The order of preference is fulltext xml > plaintext abstract > title

Distilling Raw Statements (`indra_db.util.distill_statements`)

Do some pre-pre-assembly cleansing of the raw Statements to account for various kinds of duplicity that are artifacts of our content collection and reading pipelines rather than representing actually duplicated knowledge in the literature.

`indra_db.util.distill_statements.`**`delete_raw_statements_by_id`**(*db*, *raw_sids*, *sync_session=False*, *remove='all'*)

Delete raw statements, their agents, and their raw-unique links.

It is best to batch over this function with sets of 1000 or so ids. Setting sync_session to False will result in a much faster resolution, but you may find some ORM objects have not been updated.

`indra_db.util.distill_statements.`**`distill_stmts`**(*db*, *get_full_stmts=False*, *clauses=None*, *handle_duplicates='error'*)

Get a corpus of statements from clauses and filters duplicate evidence.

Parameters

- **db** (DatabaseManager) – A database manager instance to access the database.
- **get_full_stmts** (*bool*) – By default (False), only Statement ids (the primary index of Statements on the database) are returned. However, if set to True, serialized INDRA Statements will be returned. Note that this will in general be VERY large in memory, and therefore should be used with caution.
- **clauses** (*None or list of sqlalchemy clauses*) – By default None. Specify sqlalchemy clauses to reduce the scope of statements, e.g. *clauses=[db.Statements.type == 'Phosphorylation']* or *clauses=[db.Statements.uuid.in_([<uuids>])]*.
- **handle_duplicates** (*'error', 'delete', or a string file path*) – Choose whether you want to delete the statements that are found to be duplicates ('delete'), or write a pickle file with their ids (at the string file path) for later handling, or raise an exception ('error'). The default behavior is 'error'.

Returns stmt_ret – A set of either statement ids or serialized statements, depending on *get_full_stmts*.

Return type set

`indra_db.util.distill_statements.`**`get_filtered_db_stmts`**(*db*, *get_full_stmts=False*, *clauses=None*)

Get the set of statements/ids from databases minus exact duplicates.

`indra_db.util.distill_statements.`**`get_filtered_rdg_stmts`**(*stmt_nd*, *get_full_stmts*, *linked_sids=None*)

Get the set of statements/ids from readings minus exact duplicates.

`indra_db.util.distill_statements.`**`get_reading_stmt_dict`**(*db*, *clauses=None*, *get_full_stmts=True*)

Get a nested dict of statements, keyed by ref, content, and reading.

Script to Create a SIF Dump (`indra_db.util.dump_sif`)

Create an interactome from metadata in the database and dump the results as a sif file.

`indra_db.util.dump_sif.`**`dump_sif`**(*src_count_file*, *res_pos_file*, *belief_file*, *df_file=None*, *db_res_file=None*, *csv_file=None*, *reload=True*, *reconvert=True*, *ro=None*, *normalize_names: bool = True*)

Build and dump a sif dataframe of PA statements with grounded agents

Parameters

- **src_count_file** (Union*[str, S3Path]*) – A location to load the source count dict from. Can be local file path, an s3 url string or an S3Path instance.
- **res_pos_file** (Union*[str, S3Path]*) – A location to load the residue-postion dict from. Can be local file path, an s3 url string or an S3Path instance.
- **belief_file** (Union*[str, S3Path]*) – A location to load the belief dict from. Can be local file path, an s3 url string or an S3Path instance.
- **df_file** (*Optional[*Union*[str, S3Path]]*) – If provided, dump the sif to this location. Can be local file path, an s3 url string or an S3Path instance.
- **db_res_file** (*Optional[*Union*[str, S3Path]]*) – If provided, save the db content to this location. Can be local file path, an s3 url string or an S3Path instance.
- **csv_file** (*Optional[str, S3Path]*) – If provided, calculate dataframe statistics and save to local file or s3. Can be local file path, an s3 url string or an S3Path instance.

- **reconvert** (*bool*) – Whether to generate a new DataFrame from the database content or to load and return a DataFrame from *df_file*. If False, *df_file* must be given. Default: True.
- **reload** (*bool*) – If True, load new content from the database and make a new dataframe. If False, content can be loaded from provided files. Default: True.
- **ro** (*Optional[PrincipalDatabaseManager]*) – Provide a DatabaseManager to load database content from. If not provided, *get_ro('primary')* will be used.
- **normalize_names** – If True, detect and try to merge name duplicates (same entity with different names, e.g. Loratadin vs loratadin). Default: False

`indra_db.util.dump_sif.`**`get_source_counts`**`(`*pkl_filename=None*, *ro=None*`)`

Returns a dict of dicts with evidence count per source, per statement

The dictionary is at the top level keyed by statement hash and each entry contains a dictionary keyed by the source that support the statement where the entries are the evidence count for that source.

`indra_db.util.dump_sif.`**`load_db_content`**`(`*ns_list*, *pkl_filename=None*, *ro=None*, *reload=False*`)`

Get preassembled stmt metadata from the DB for export.

Queries the NameMeta, TextMeta, and OtherMeta tables as needed to get agent/stmt metadata for agents from the given namespaces.

Parameters

- **ns_list** (*list of str*) – List of agent namespaces to include in the metadata query.
- **pkl_filename** (*str*) – Name of pickle file to save to (if reloading) or load from (if not reloading). If an S3 path is given (i.e., pkl_filename starts with *s3:*), the file is loaded to/saved from S3. If not given, automatically reloads the content (overriding reload).
- **ro** (*ReadonlyDatabaseManager*) – Readonly database to load the content from. If not given, calls *get_ro('primary')* to get the primary readonly DB.
- **reload** (*bool*) – Whether to re-query the database for content or to load the content from from *pkl_filename*. Note that even if *reload* is False, if no *pkl_filename* is given, data will be reloaded anyway.

Returns Set of tuples containing statement information organized by agent. Tuples contain (stmt_hash, agent_ns, agent_id, agent_num, evidence_count, stmt_type).

Return type set of tuples

`indra_db.util.dump_sif.`**`load_res_pos`**`(`*ro=None*`)`

Return residue/position data keyed by hash

`indra_db.util.dump_sif.`**`make_dataframe`**`(`*reconvert*, *db_content*, *res_pos_dict*, *src_count_dict*, *belief_dict*, *pkl_filename=None*, *normalize_names: bool = False*`)`

Make a pickled DataFrame of the db content, one row per stmt.

Parameters

- **reconvert** (*bool*) – Whether to generate a new DataFrame from the database content or to load and return a DataFrame from the given pickle file. If False, *pkl_filename* must be given.
- **db_content** (*set of tuples*) – Set of tuples of agent/stmt data as returned by *load_db_content*.
- **res_pos_dict** (*Dict[str, Dict[str, str]]*) – Dict containing residue and position keyed by hash.
- **src_count_dict** (*Dict[str, Dict[str, int]]*) – Dict of dicts containing source counts per source api keyed by hash.

- **belief_dict** (*Dict[str, float]*) – Dict of belief scores keyed by hash.
- **pkl_filename** (*str*) – Name of pickle file to save to (if reconverting) or load from (if not reconverting). If an S3 path is given (i.e., pkl_filename starts with *s3:*), the file is loaded to/saved from S3. If not given, reloads the content (overriding reload).
- **normalize_names** – If True, detect and try to merge name duplicates (same entity with different names, e.g. Loratadin vs loratadin). Default: False

Returns DataFrame containing the content, with columns: 'agA_ns', 'agA_id', 'agA_name', 'agB_ns', 'agB_id', 'agB_name', 'stmt_type', 'evidence_count', 'stmt_hash'.

Return type pandas.DataFrame

General Helper Functions (`indra_db.util.helpers`)

Functions with broad utility throughout the repository, but otherwise miscellaneous.

`indra_db.util.helpers.`**`get_raw_stmts_frm_db_list`**(*db*, *db_stmt_objs*, *fix_refs=True*, *with_sids=True*)

Convert table objects of raw statements into INDRA Statement objects.

`indra_db.util.helpers.`**`get_statement_object`**(*db_stmt*)

Get an INDRA Statement object from a db_stmt.

Routines for Inserting Statements and Content (`indra_db.util.insert`)

Inserting content into the database can be a rather involved process, but here are defined high-level utilities to uniformly accomplish the task.

`indra_db.util.insert.`**`extract_agent_data`**(*stmt*, *stmt_id*)

Create the tuples for copying agents into the database.

`indra_db.util.insert.`**`insert_db_stmts`**(*db*, *stmts*, *db_ref_id*, *verbose=False*, *batch_id=None*)

Insert statement, their database, and any affiliated agents.

Note that this method is for uploading statements that came from a database to our database, not for inserting any statements to the database.

Parameters

- **db** (`DatabaseManager`) – The manager for the database into which you are loading statements.
- **stmts** (list [`indra.statements.Statement`]) – (Cannot be a generator) A list of unassembled indra statements, each with EXACTLY one evidence and no exact duplicates, to be uploaded to the database.
- **db_ref_id** (*int*) – The id to the db_ref entry corresponding to these statements.
- **verbose** (*bool*) – If True, print extra information and a status bar while compiling statements for insert. Default False.
- **batch_id** (*int or None*) – Select a batch id to use for this upload. It can be used to trace what content has been added.

`indra_db.util.insert.`**`insert_pa_stmts`**(*db*, *stmts*, *verbose=False*, *do_copy=True*, *ignore_agents=False*, *commit=True*)

Insert pre-assembled statements, and any affiliated agents.

Parameters

- **db** (`DatabaseManager`) – The manager for the database into which you are loading pre-assembled statements.
- **stmts** (iterable [`indra.statements.Statement`]) – A list of pre-assembled indra statements to be uploaded to the datbase.
- **verbose** (*bool*) – If True, print extra information and a status bar while compiling statements for insert. Default False.
- **do_copy** (*bool*) – If True (default), use pgcopy to quickly insert the agents.
- **ignore_agents** (*bool*) – If False (default), add agents to the database. If True, then agent insertion is skipped.
- **commit** (*bool*) – If True (default), commit the result immediately. Otherwise the results are not committed (thus allowing multiple related insertions to be neatly rolled back upon failure.)

`indra_db.util.insert.`**`insert_raw_agents`**(*db*, *batch_id*, *stmts=None*, *verbose=False*, *num_per_yield=100*, *commit=True*)

Insert agents for statements that don't have any agents.

Parameters

- **db** (`DatabaseManager`) – The manager for the database into which you are adding agents.
- **batch_id** (*int*) – Every set of new raw statements must be given an id unique to that copy That id is used to get the set of statements that need agents added.
- **stmts** (*list[indra.statements.Statement]*) – The list of statements that include those whose agents are being uploaded.
- **verbose** (*bool*) – If True, print extra information and a status bar while compiling agents for insert from statements. Default False.
- **num_per_yield** (*int*) – To conserve memory, statements are loaded in batches of *num_per_yeild* using the *yeild_per* feature of sqlalchemy queries.
- **commit** (*bool*) – Optionally do not commit at the end. Default is True, meaning a commit will be executed.

`indra_db.util.insert.`**`regularize_agent_id`**(*id_val*, *id_ns*)

Change agent ids for better search-ability and index-ability.

4.2.8 Some Miscellaneous Modules

Here are some modules and files that live on their own, and don't fit neatly into other categories.

Low Level Database Interface (`indra_db.databases`)

The Database Manager classes are the lowest level interface with the database, implemented with SQLAlchemy, providing useful short-cuts but also allowing full access to SQLAlchemy's API.

class `indra_db.databases.`**`DatabaseManager`**(*url*, *label=None*, *protected=False*)

An object used to access INDRA's database.

This object can be used to access and manage indra's database. It includes both basic methods and some useful, more high-level methods. It is designed to be used with postgresql, or sqlite.

This object is primarily built around sqlalchemy, which is a required package for its use. It also optionally makes use of the pgcopy package for large data transfers.

If you wish to access the primary database, you can simply use the *get_db* function to get an instance of this object using the default settings.

Parameters

- **url** (*str*) – The database to which you want to interface.
- **label** (*OPTIONAL[str]*) – A short string to indicate the purpose of the db instance. Set as db_label when initialized with get_db(db_label).

Example

If you wish to access the primary database and find the the metadata for a particular pmid, 1234567:

```
from indra.db import get_db
db = get_db('primary')
res = db.select_all(db.TextRef, db.TextRef.pmid == '1234567')
```

You will get a list of objects whose attributes give the metadata contained in the columns of the table.

For more sophisticated examples, several use cases can be found in indra.tests.test_db.

classmethod create_instance(*instance_name*, *size*, *tag_dict=None*)
Allocate the resources on RDS for a database, and return handle.

get_config_string()
Print a config entry for this handle.

This is useful after using *create_instance*.

get_env_string()
Generate the string for an environment variable.

This is useful after using *create_instance*.

grab_session()
Get an active session with the database.

get_tables()
Get a list of available tables.

show_tables(*active_only=False*, *schema=None*)
Print a list of all the available tables.

get_active_tables(*schema=None*)
Get the tables currently active in the database.

Parameters **schema** (*None or st*) – The name of the schema whose tables you wish to see. The default is public.

get_schemas()
Return the list of schema names currently in the database.

create_schema(*schema_name*)
Create a schema with the given name.

drop_schema(*schema_name*, *cascade=True*)
Drop a schema (rather forcefully by default)

get_column_names(*table*)
"Get a list of the column labels for a table.

Note that if the table involves a schema, the schema name must be prepended to the table name.

`get_column_objects`(*table*)
Get a list of the column object for the given table.

Note that if the table involves a schema, the schema name must be prepended to the table name.

`commit`(*err_msg*)
Commit, and give useful info if there is an exception.

`link`(*table_1*, *table_2*)
Get the joining clause between two tables, if one exists.

If no link exists, an exception will be raised. Note that this only works for directly links.

`get_values`(*entry_list*, *col_names=None*, *keyed=False*)
Get the column values from the entries in entry_list

`insert`(*table*, *ret_info=None*, ***input_dict*)
Insert a an entry into specified table, and return id.

`insert_many`(*table*, *input_data_list*, *ret_info=None*, *cols=None*)
Insert many records into the table given by table_name.

`delete_all`(*entry_list*)
Remove the given records from the given table.

`get_copy_cursor`()
Execute SQL queries in the context of a copy operation.

`make_copy_batch_id`()
Generate a random batch id for copying into the database.

This allows for easy retrieval of the assigned ids immediately after copying in. At this time, only Reading and RawStatements use the feature.

`copy_report_lazy`(*tbl_name*, *data*, *cols=None*, *commit=True*, *constraint=None*, *return_cols=None*, *order_by=None*)
Copy lazily, and report what rows were skipped.

`copy_detailed_report_lazy`(*tbl_name*, *data*, *inp_cols=None*, *ret_cols=None*, *commit=True*, *constraint=None*, *skipped_cols=None*, *order_by=None*)
Copy lazily, returning data from some of the columns such as IDs.

`copy_lazy`(*tbl_name*, *data*, *cols=None*, *commit=True*, *constraint=None*)
Copy lazily, skip any rows that violate constraints.

`copy_push`(*tbl_name*, *data*, *cols=None*, *commit=True*, *constraint=None*)
Copy, pushing any changes to constraint violating rows.

`copy_report_push`(*tbl_name*, *data*, *cols=None*, *commit=True*, *constraint=None*, *return_cols=None*, *order_by=None*)
Report on the rows skipped when pushing and copying.

`copy`(*tbl_name*, *data*, *cols=None*, *commit=True*)
Use pg_copy to copy over a large amount of data.

`filter_query`(*tbls*, **args*)
Query a table and filter results.

`count`(*tbl*, **args*)
Get a count of the results to a query.

`get_primary_key`(*tbl*)
Get an instance for the primary key column of a given table.

`select_one`(tbls, *args)

Select the first value that matches requirements.

Requirements are given in kwargs from table indicated by tbl_name. See *select_all*.

Note that if your specification yields multiple results, this method will just return the first result without exception.

`select_all`(tbls, *args, **kwargs)

Select any and all entries from table given by tbl_name.

The results will be filtered by your keyword arguments. For example if you want to get a text ref with pmid '10532205', you would call:

```
db.select_all('text_ref', db.TextRef.pmid == '10532205')
```

Note that double equals are required, not a single equal. Equivalently you could call:

```
db.select_all(db.TextRef, db.TextRef.pmid == '10532205')
```

For a more complicated example, suppose you want to get all text refs that have full text from pmc oa, you could select:

```
db.select_all(
    [db.TextRef, db.TextContent],
    db.TextContent.text_ref_id == db.TextRef.id,
    db.TextContent.source == 'pmc_oa',
    db.TextContent.text_type == 'fulltext'
    )
```

Parameters

- **tbls** – See above for usage.
- ***args** – See above for usage.
- ****kwargs** – yield_per: int or None If the result to your query is expected to be large, you can choose to only load *yield_per* items at a time, using the eponymous feature of sqlalchemy queries. Default is None, meaning all results will be loaded simultaneously.

`select_all_batched`(batch_size, tbls, *args, skip_idx=None, order_by=None)

Load the results of a query in batches of size batch_size.

Note that this differs from using yeild_per in that the results are not returned as a single iterable, but as an iterator of iterables.

Note also that the order of results, and thus the contents of offsets, may vary for large queries unless an explicit order_by clause is added to the query.

`select_sample_from_table`(number, table, *args, **kwargs)

Select a number of random samples from the given table.

Parameters

- **number** (`int`) – The number of samples to return
- **table** (`str, table class, or column attribute of table class`) – The table or table column to be sampled.
- ***args** – All other arguments are passed to *select_all*, including any and all filtering clauses.

- **kwargs – All other arguments are passed to *select_all*, including any and all filtering clauses.

Return type A list of sqlalchemy orm objects

`has_entry(tbls, *args)`
Check whether an entry/entries matching given specs live in the db.

`pg_dump(dump_file, **options)`
Use the pg_dump command to dump part of the database onto s3.

The *pg_dump* tool must be installed, and must be a compatible version with the database(s) being used.

All keyword arguments are converted into flags/arguments of pg_dump. For documentation run *pg_dump –help*. This will also confirm you have *pg_dump* installed.

By default, the "General" and "Connection" options are already set. The most likely specification you will want to use is *–table* or *–schema*, specifying either a particular table or schema to dump.

Parameters dump_file (*S3Path or str*) – The location on s3 where the content should be dumped.

`pg_restore(dump_file, **options)`
Load content into the database from a dump file on s3.

`exception indra_db.databases.IndraDbException`

`indra_db.databases.readers = {'EIDOS': 5, 'ISI': 4, 'MTI': 6, 'REACH': 1, 'SPARSER': 2, 'TRIPS': 3}`
A dict mapping each reader a unique integer ID.

These ID's are used in creating the reading primary ID hashes. Thus, for a new reader to be fully integrated, it must be added to the above dictionary.

`indra_db.databases.reader_versions = {'eidos': ['0.2.3-SNAPSHOT'], 'isi': ['20180503'], 'mti': ['1.0'], 'reach': ['61059a-biores-e9ee36', '1.3.3-61059a-biores-', '1.6.1', '1.6.3-e48717'], 'sparser': ['sept14-linux\n', 'sept14-linux', 'June2018-linux', 'October2018-linux', 'February2020-linux', 'April2020-linux'], 'trips': ['STATIC', '2019Nov14', '2021Jan26']}`
A dict of list values keyed by reader name, tracking reader versions.

The oldest versions are to the left, and the newest to the right. We keep track of all past versions as it is often not practical nor necessary to re-run a reading on all content. Even in cases where it is, it is often useful to be able to compare results.

As with the `readers` variable above, this is used in the creation of the unique hash for a reading entry. For a new reader version to work, it must be added to the appropriate list.

`class indra_db.databases.PrincipalDatabaseManager(host, label=None, protected=False)`
This class represents the methods special to the principal database.

`generate_readonly(belief_dict, allow_continue=True)`
Manage the materialized views.

Parameters

- belief_dict (*dict*) – The dictionary, keyed by hash, of belief calculated for Statements.
- allow_continue (*bool*) – If True (default), continue to build the schema if it already exists. If False, give up if the schema already exists.

`dump_readonly(dump_file=None)`
Dump the readonly schema to s3.

create_tables(*tbl_list=None*)
Create the public tables for INDRA database.

drop_tables(*tbl_list=None*, *force=False*)
Drop the tables for INDRA database given in tbl_list.

If tbl_list is None, all tables will be dropped. Note that if *force* is False, a warning prompt will be raised to asking for confirmation, as this action will remove all data from that table.

class indra_db.databases.**ReadonlyDatabaseManager**(*host*, *label=None*, *protected=True*)
This class represents the readonly database.

get_config_string()
Print a config entry for this handle.

This is useful after using *create_instance*.

get_source_names() → set
Get a list of the source names as they appear in SourceMeta cols.

get_active_tables(*schema='readonly'*)
Get the tables currently active in the database.

Parameters **schema** (*None or st*) – The name of the schema whose tables you wish to see. The default is readonly.

load_dump(*dump_file*, *force_clear=True*)
Load from a dump of the readonly schema on s3.

Belief Calculator (indra_db.belief)

The belief in the knowledge of a Statement is a measure of our confidence that the Statement is an accurate representation of the text, _NOT_ our confidence in the validity of what was in that text. Given the size of the content in the database, some special care is needed when calculating this value, which depends heavily on the support relations between pre-assembled Statements.

This file contains tools to calculate belief scores for the database.

Scores are calculated using INDRA's belief engine, with MockStatements and MockEvidence derived from shallow metadata on the database, allowing the entire corpus to be processed locally in RAM, in very little time.

exception indra_db.belief.**LoadError**

class indra_db.belief.**MockEvidence**(*source_api*, ***annotations*)
A class to imitate real INDRA Evidence for calculating belief.

class indra_db.belief.**MockStatement**(*mk_hash*, *evidence=None*, *supports=None*, *supported_by=None*)
A class to imitate real INDRA Statements for calculating belief.

indra_db.belief.**load_mock_statements**(*db*, *hashes=None*, *sup_links=None*)
Generate a list of mock statements from the pa statement table.

indra_db.belief.**populate_support**(*stmts*, *links*)
Populate the supports supported_by lists of statements given links.

Parameters

- **stmts** (*list[MockStatement/Statement]*) – A list of objects with supports and supported_by attributes which are lists or equivalent.
- **links** (*list[tuple]*) – A list of pairs of hashes or matches_keys, where the first supports the second.

CHAPTER

FIVE

INDRA DATABASE REST SERVICE

5.1 INDRA Database REST API

The INDRA Database software has been developed to create and maintain a database of text references, content, reading results, and ultimately INDRA Statements extracted from those reading results. The software also manages the generation and update process of cleaning, deduplicating, and finding relations between the raw Statement extractions, into what are called pre-assembled Statements. All INDRA Statements can be represented as JSON, which is the format returned by the API.

This web API provides the code necessary to support a REST service which allows access to the pre-assembled Statements in a database. The system is still under heavy development so capabilities are always expanding, but as of this writing, the API supports:

- ``*statements/from_agents*` <#from-agents>`_, getting Statements by agents, using various ids or names, by statement type (e.g. Phosphorylation), or
- ``*statements/from_hash*` <#from-hash>`_ and ``*statements/from_hashes*` <#from-hashes>`_, getting Statements by Statement hash, either singly or in batches, and
- ``*statements/from_papers*` <#from-papers>`_, getting Statements using the paper ids from which they were extracted, and
- ``*curation/submit/<hash>*` <#curation>`_ you can also curate Statements, helping us improve the quality and accuracy of our content.

As mentioned, the service is changing rapidly, and this documentation may at times be out of date. For the latest, check github or contact us.

You need the following information to access a running web service:

- The address of the web service (below shown with the placeholder `api.host`)
- An API key which needs to be sent in the header of each request to the service, or any other credentials that are implemented.

If you want to use our implementation of the web API, you can contact us for the path and an API key.

The code to support the REST service can be found in `api.py`, implemented using the Flask Python package. The means of hosting this api are left to the user. We have had success using Zappa and AWS Lambda, and recommend it for a quick and efficient way to get the API up and running.

5.1.1 The Statement Endpoints

For all queries, an API key is required, which is passed as a parameter `api_key` to any/all queries. Below is detailed documentation for the different endpoints of the API that return statements (i.e. those with the root `/statements`). All endpoints that return statements have the following options to control the size and order of the response:

- **`format`**: The endpoint is capable of returning both HTML and JSON content by setting the format parameter to "html" or "json", respectively. See the *section on output formats* below.
- **`max_stmts`**: Set the maximum number of statements you wish to receive. The REST API maximum is 1000, which cannot be overridden by this argument (to prevent request timeouts).
- **`ev_limit`**: The default varies, but in general the amount of Evidence returned for each statement is limited. A single statement can have upwards of 10,000 pieces of evidence, so this allows queries to be run reliably. There is no limitation on this value, so use with caution. Setting too high a value may cause a request to time out or be too large to return.
- **`best_first`**: This is set to "true" by default, so statements with the most evidence are returned first. These are generally the most reliable, however they are also generally the most canonical. Set this parameter to "false" to get statements in an arbitrary order. This can also speed up a query. You may however find you get a lot of low-quality content.

The output formats

The output format is controlled by the **`format`** option described above, with options to return JSON or HTML.

JSON: The default value, intended for programmatic use, is "json". The JSON that is returned is of the following form (with many made-up but reasonable numbers filled in):

```
{
  "statements": {  # Dict of statement JSONs keyed by hash
    "12345234234": {...},  # Statement JSON 1
    "-246809323482": {...},  # Statement JSON 2
    ...},
  "offset": 2000,  # offset of SQL query
  "evidence_limit": 10,  # evidence limit used
  "statement_limit": 1000,  # REST API Limit
  "evidence_totals": {  # dict of available evidence for each statement keyed by hash
    "12345234234": 7657,
    "-246809323482": 870,
    ...},
  "total_evidence": 163708, # The total amount of evidence available
  "evidence_returned": 10000  # The total amount of evidence returned
}
```

where the `"statements"` element contains a dictionary of INDRA Statement JSONs keyed by a shallow statement hash (see *here* for more details on these hashes). You can look at the JSON schema on github for details on the Statement JSON. To learn more about INDRA Statements, you can read the documentation.

HTML: The other `format` parameter option, designed for easier manual usage, is "html". The service will then return an HTML document that, when opened in a web browser and if logged in, provides a graphical user interface for viewing and curating statements at the evidence level. The web page also allows you to easily query for more evidence for a given statement. Documentation for the html output (produced by INDRA's HTML assembler) can be found here.

Get Statements by agents (and type): `GET api.host/statements/from_agents`

This endpoint allows you to get statements filtering by their agents and the type of Statement. The query parameters are as follows:

- **subject**, **object**: The HGNC gene symbol of the subject or object of the Statement. **Note**: only one of each of `subject` and `object` will be accepted per query.
  - Example 1: if looking for Statements where MAP2K1 is a subject (e.g. "What does MAP2K1 phosphorylate?"), specify `subject=MAP2K1` as a query parameter
  - Example 2: if looking for Statements where MAP2K1 is the subject and MAPK1 is the object, add both `subject=MAP2K1` and `object=MAPK1` as query parameters.
  - Example 3: you can specify the agent id namespace by appending `@<namespace>` to the agent id in the parameter, e.g. `subject=6871@HGNC`.
- **agent***: This parameter is used if the specific role of the agent (subject or object) is irrelevant, or the distinction doesn't apply to the type of Statement of interest (e.g. Complex, Translocation, ActiveForm). **Note**: You can include as many `agent*` queries as you like, however you will only get Statements that include all agents you query, in addition to those queried for `subject` and `object`. Furthermore, to include multiple agents on our particular implementation, which uses the AWS API Gateway, you must include a suffix to each agent key, such as `agent0` and `agent1`, or else all but one agent will be stripped out. Note that you need not use integers, you can add any suffix you like, e.g. `agentOfDestruction=TP53` would be entirely valid.
  - Example 1: To obtain Statements that involve SMAD2 in any role, add `agent=SMAD2` to the query.
  - Example 2: As with `subject` and `object`, you can specify the namespace for an agent by appending `@<namespace>` to the agent's id, e.g. `agent=ERK@TEXT`.
  - Example 3: If you wanted to query multiple statements, you could include `agent0=MEK@FPLX` and `agent1=ERK@FPLX`. Note that the value of the integers has no real bearing on the ordering, and only serves to make the agents uniquely keyed. Thus `agent1=MEK@FPLX` and `agent0=ERK@FPLX` will give exactly the same result.
- **type**: This parameter can be used to specify what type of Statement of interest (e.g. Phosphorylation, Activation, Complex).
  - Example: To answer the question "Does MAP2K1 phosphorylate MAPK1?" the parameter `type=Phosphorylation` can be included in your query. Note that this field is not case sensitive, so `type=phosphorylation` would give the same result.

Get a Statement by hash: `GET api.host/statements/from_hash/<hash>`

INDRA Statement objects have a method, `get_hash`, which produces hash from the content of the Statement. A shallow hash only considers the meaning of the statement (agents, type, modifications, etc.), whereas a deeper hash also considers the list of evidence available for that Statement. The shallow hash is what is used in this application, as it has the same uniqueness properties used in deduplication. As mentioned above, the Statements are returned keyed by their hash. In addition, if you construct a Statement in python, you may get its hash and quickly find any evidence for that Statement in the database.

This endpoint has no extra parameters, but rather takes an extension to the path. So, to look up the hash 123456789, you would use `statements/from_hash/123456789`.

Because this only returns one statement, the default evidence limit is extremely generous, set to 10,000. Thus you are most likely to get all the evidence for a given statement this way. As described above, the evidence limit can also be raised, at the risk of a timed out request.

Get Statements from many hashes: `POST api.host/statements/from_hashes`

Like the previous endpoint, this endpoint uses hashes to retrieve Statements, however instead of only being allowed one at a time, a bach of hashes may be sent as json data. Because data is sent, this is a POST request, even though you are in practice "getting" information. There are no special parameters for this endpoint. The json data should be formatted as:

```
{"hashes": [12345, 246810]}
```

with up to 1,000 hashes given in the list.

Get Statements from paper ids: `POST api.host/statements/from_papers`

Using this endpoint, you can pretend you have a fleet of text extraction tools that run in seconds! Specifically, you can get the INDRA Statements with evidence from a given list of papers by passing one of the ids of those papers. As with the above method, the fact that data (paper ids) is send requires this to be a POST request. The papers ids should be formatted as:

```
{"ids": [{"id": "12345", "type": "pmid"},
         {"id": "234525", "type": "tcid"},
         {"id": "PMC23423", "type": "pmcid"}]}
```

a list of dicts, each containing id type and and id value.

5.1.2 Curation

Because the mechanisms represented by our Statements come in large part from automatic extractions, there can often be errors. For this reason, we always provide the sentences from which a Statement was extracted (if we extracted it, some of our content comes from other curated databases), as well as provenance to lead back to the content (abstract, full text, etc.) that was read, which allows you to use your own judgement regarding the validity of a Statement.

If you find something wrong with a Statement, you can use this curation endpoint to record your observation. This will not necessarily have any immediate effect on the output, however, over time it will help us improve the readers we use, our methods for extracting Statements from those reader outputs, could help us filter erroneous content, and will help us improve our pre-assembly algorithms.

Further instruction on curation best practices can be found here.

Curate statements: `POST api.host/curation/submit/<hash>`

If you wish to curate a Statement, you must first decide whether you are curating the Statement as generally incorrect, or whether a particular sentence supports a given Statement. This is the “level” of your curation:

- **pa**: At this level, you are curating the knowledge in a **p**re-**a**ssembled Statement. For example, if a Statement indicates that “differentiation binds apoptosis”, regardless of whether the reader(s) made a valid extraction, it is clearly wrong.
- **raw**: At this level, you are curating a particular raw extraction, in other words stating that an automatic reader made an error. Even more explicitly, you can judge whether the sentence supports the extracted Statement. For example the (hypothetical) sentence “KRAS was found to actively inhibit BRAF” does not support the Statement “KRAS activates BRAF”. As another example (here a grounding error), would be that the sentence “IR causes cell death”, where IR is Ionizing Radiation does not support the extraction “‘Insulin Receptor’ causes cell death”.

The two different levels also have different hashes. At the *pa* level, the hashes discussed *above* are used, as they are calculated from the knowledge contained in the statement, independent of the evidence. At the *raw* level, a different hash must be included: the `source_hash`, which identifies a specific piece of evidence, without considering the Statement extracted. Within a Statement JSON, there is a key “evidence”, with a list of Evidence JSON, which includes an entry for “source_hash”:

```
{"evidence": [{"source_hash": 98687578576598, ...}, ...], ...}
```

Once you know the level, and you have the correct hash(es) (the shallow pre-assembly hash and/or the source hash), you can curate a statement by POSTing a request with JSON data to the endpoint, as shown in the heading. The JSON data should contain the following fields:

- **tag**: A very short word or phrase categorizing the error, for example “grounding” for a grounding error.
- **text**: A brief description of what you think is most wrong.
- **curator**: Your name, initials, email, or other way to identify yourself. Whichever you choose, please be consistent.

Note that you can also indicate that a Statement is *correct*. In particular, if you find that a Statement has some evidence that supports the Statement and some that does not, curating examples of both is valuable. In general, flagging correct Statements can be just as valuable as flagging incorrect Statements.

5.1.3 Usage examples

The web service accepts standard HTTP requests, and any client that can send such requests can be used to interact with the service. Here we provide usage examples with the `curl` command line tool and `python` of some of the endpoints. This is by no means a comprehensive list, but rather demonstrates some of the crucial features discussed above.

In the examples, we assume the path to the web API is `https://api.host/`, and that the API key is `12345`.

`curl` is a command line tool on Linux and Mac, making it a convenient tool for making calls to this web API.

Using `curl` to query Statements about “MAP2K1 phosphorylates MAPK1”:

```
curl -X GET "http://api.host/statements/from_agents?subject=MAP2K1&object=MAPK1&type=phosphorylation&api_key=12345"
```

```
{
  "statements": {
    "-1072112758478440": {
```

(continues on next page)

(continued from previous page)

```
      "id": "5c3dff5f-6660-4494-96d2-0142076e9b2f",
      "enz": {
        "name": "MAP2K1",
        "db_refs": {
          "UP": "Q02750",
          "HGNC": "6840"
        },
        "sbo": "http://identifiers.org/sbo/SBO:0000460"
      },
      "sbo": "http://identifiers.org/sbo/SBO:0000216",
      "evidence": [
        {
          "source_api": "reach",
          "epistemics": {
            "section_type": null,
            "direct": true
          },
          "text": "Thus, free non visual arrestins moderately facilitate the␣
↪phosphorylation of ERK2 by MEK1.",
          "pmid": "22174878",
          "annotations": {
            "agents": {
              "raw_text": [
                "MEK1",
                "ERK2"
              ]
            },
            "content_source": "pmc_oa",
            "prior_uuids": [
              "55afb6fc-5649-4315-94bc-3ce0651fc1d3"
            ],
            "found_by": "Phosphorylation_syntax_1a_noun"
          }
        }
      ],
      "type": "Phosphorylation",
      "sub": {
        "name": "MAPK1",
        "db_refs": {
          "UP": "P28482",
          "HGNC": "6871"
        },
        "sbo": "http://identifiers.org/sbo/SBO:0000015"
      }
    }
  },
  "offset": null,
  "total_evidence": 106,
  "evidence_totals": {
    "-1072112758478440": 106
  },
  "evidence_returned": 1,
```

(continues on next page)

(continued from previous page)

```
  "evidence_limit": "1",
  "statement_limit": 1000
}
```

</p> </details>

Python is another convenient way to use this web API, and has the important advantage that Statements returned from the service can directly be used directly with INDRA tools.

You can use python to get JSON Statements for the same query:

```
import requests
resp = requests.get('http://api.host/statements/from_agents',
                    params={'subject': 'MAP2K1',
                            'object': 'MAPK1',
                            'type': 'phosphorylation',
                            'api_key': 12345})
resp_json = resp.json()
```

which can now be turned into INDRA Statement objects using `stmts_from_json`:

```
from indra.statements import stmts_from_json
stmts = stmts_from_json(resp_json['statements'].values())
```

For those familiar with pre-assembled INDRA Statements, note that the `supports` and `supported_by` lists of the python Statement objects are not populated.

INDRA also supports a client to this API, which is documented in detail elsewhere, however using that client, the above query is simply:

```
from indra.sources import indra_db_rest as idbr
processor = idbr.get_statements(subject='MAP2K1', object='MAPK1', stmt_type=
↪'phosphorylation')
stmts = processor.statements
```

Where the URL and API key are located n a config file. A key advantage of this client is that queries that return more than 1000 statement are paged behind the scenes, so that all the statements which match the query are retrieved with a single command.

By setting the `format` option to `html` in the web API address, an HTML document that presents a graphical user interface when displayed in a web browser will be returned. The example below queries for statements where BRCA1 is subject and BRCA2 is object:

```
http://api.host/statements/from_agents?subject=BRCA1&object=BRCA2&api_key=12345&
↪format=html
```

The interface is restricted to users with login access. If you are not logged in, you will be prompted to so before you can view the loaded statements. Once logged in, the queried statements will be loaded and you will be able to curate statements on the level of individual evidences. Links to various source databases (depending on availability) are available for each piece of evidence to facilitate accurate curation. Find out more about the HTML interface in the HTML assembler documentation. For instructions on how to use it and more about the login restriction, see the manual.

Use curl to query for any kind of interaction between SMURF2 and SMAD2, returning at most 10 statements with 3 evidence each:

```
curl -X GET "http://api.host/statements/from_agents?agent0=SMURF2&agent1=SMAD2&api_
↪key=12345&limit=10&ev_limit=3"
```

As above, in python this could be handled using the `requests` module, or with the client:

```
import requests
from indra.statements import stmts_from_json
from indra.sources import indra_db_rest as idbr

# With requests
resp = requests.get('http://api.host/statements/from_agents',
                    params={'agent0': 'SMURF2', 'agent1': 'SMAD',
                            'api_key': 12345, 'limit': 10,
                            'ev_limit': 3})
resp_json = resp.json()
stmts = stmts_from_json(resp_json['statements'].values())

# With the client
stmts = idbr.get_statements(agents=['SMURF2', 'SMAD'], max_stmts=10,
                            ev_limit=3, simple_response=True)
```

Note the use of the `@FPLX` suffix to denote the namespace used in identifying the agent to query for things that inhibit MEK, using curl:

```
curl -X GET "http://api.host/statements/from_agents?object=MEK@FPLX&type=inhibition&api_
↪key=12345"
```

Python requests:

```
resp = requests.get('http://api.host/statements/from_agents',
                    params={'agent': 'MEK@FPLX', 'type': 'inhibition',
                            'api_key': 12345})
```

and INDRA's client:

```
stmts = idbr.get_statements(agents=['MEK@FPLX'], stmt_type='inhibition')
```

Query for a statement with the hash -1072112758478440, retrieving at most 1000 evidence, using curl:

```
curl -X GET "http://api.host/statements/from_hash/-1072112758478440?api_key=12345&ev_
↪limit=1000"
```

or INDRA's client:

```
stmts = idbr.get_statements_by_hash([-1072112758478440], ev_limit=1000)
```

Note that client does not actually use the same endpoint here, but rather uses the `/from_hashes` endpoint.

Get the statements from a paper with the pmid 22174878, and another paper with the doi 10.1259/0007-1285-34-407-693, first create the json file, call it `papers.json` with the following:

```
{
  "ids": [
    {"id": "22174878", "type": "pmid"},
```

(continues on next page)

(continued from previous page)

```
    {"id": "10.1259/0007-1285-34-407-693", "type": "doi"}
  ]
}
```

and post it to the REST API with curl:

```
curl -X POST "http://api.host/statements/from_papers?api_key=12345" -d @papers.json -H
↪"Content-Type: application/json"
```

or just use INDRA's client:

```
stmts = idbr.get_statments_for_paper([('pmid', '22174878'),
                                      ('doi', '10.1259/0007-1285-34-407-693')])
```

Curate a Statement at the pre-assembled (pa) level for a Statement with hash -1072112758478440, using curl:

```
curl -X POST "http://api.host/curation/submit/-1072112758478440?api_key=12345" -d '{"tag
↪": "correct", "text": "This Statement is OK.", "curator": "Alice"}' -H "Content-Type:␣
↪application/json"
```

or INDRA's client:

```
idbr.submit_curation(-1072112758478440, 'correct', 'This Statement is OK.', 'Alice')
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX