Package 'solrium'

November 2, 2017

Title General Purpose R Interface to 'Solr'

Description Provides a set of functions for querying and parsing data from 'Solr' (<http://lucene.apache.org/solr>) 'endpoints' (local and remote), including search, 'faceting', 'highlighting', 'stats', and 'more like this'. In addition, some functionality is included for creating, deleting, and updating documents in a 'Solr' 'database'.

Version 1.0.0

License MIT + file LICENSE

URL https://github.com/ropensci/solrium

BugReports https://github.com/ropensci/solrium/issues

VignetteBuilder knitr

Imports utils, dplyr (>= 0.5.0), plyr (>= 1.8.4), crul (>= 0.4.0), xml2 (>= 1.0.0), jsonlite (>= 1.0), tibble (>= 1.2), R6

Suggests roxygen2 (>= 6.0.1), testthat, knitr

RoxygenNote 6.0.1

NeedsCompilation no

Author Scott Chamberlain [aut, cre]

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2017-11-02 09:42:40 UTC

R topics documented:

`solrium-package` *General purpose R interface to Solr.*

Description

This package has support for all the search endpoints, as well as a suite of functions for managing a Solr database, including adding and deleting documents.

Important search functions

- `solr_search` - General search, only returns documents
- `solr_all` - General search, including all non-documents in addition to documents: facets, highlights, groups, mlt, stats.
- `solr_facet` - Faceting only (w/o general search)
- `solr_highlight` - Highlighting only (w/o general search)
- `solr_mlt` - More like this (w/o general search)
- `solr_group` - Group search (w/o general search)
- `solr_stats` - Stats search (w/o general search)

Important Solr management functions

- `update_json` - Add or delete documents using json in a file
- `add` - Add documents via an R list or data.frame
- `delete_by_id` - Delete documents by ID
- `delete_by_query` - Delete documents by query

Vignettes

See the vignettes for help `browseVignettes(package = "solrium")`

Performance

v0.2 and above of this package will have wt=csv as the default. This should give significant performance improvement over the previous default of wt=json, which pulled down json, parsed to an R list, then to a data.frame. With wt=csv, we pull down csv, and read that in directly to a data.frame.

The http library we use, **crul**, sets gzip compression header by default. As long as compression is used server side, you're good to go on compression, which should be a good peformance boost. See https://wiki.apache.org/solr/SolrPerformanceFactors#Query_Response_Compression for notes on how to enable compression.

There are other notes about Solr performance at https://wiki.apache.org/solr/SolrPerformanceFactors that can be used server side/in your Solr config, but aren't things to tune here in this R client.

Let us know if there's any further performance improvements we can make.

Author(s)

Scott Chamberlain <myrmecocystus@gmail.com>

---

add *Add documents from R objects*

Description

Add documents from R objects

Usage

```
add(x, conn, name, commit = TRUE, commit_within = NULL, overwrite = TRUE,
  boost = NULL, wt = "json", raw = FALSE, ...)
```

Arguments

x	Documents, either as rows in a data.frame, or a list.
conn	A solrium connection object, see SolrClient
name	(character) A collection or core name. Required.
commit	(logical) If TRUE, documents immediately searchable. Default: TRUE
commit_within	(numeric) Milliseconds to commit the change, the document will be added within that time. Default: NULL
overwrite	(logical) Overwrite documents with matching keys. Default: TRUE
boost	(numeric) Boost factor. Default: NULL
wt	(character) One of json (default) or xml. If json, uses fromJSON to parse. If xml, uses read_xml to parse
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to crul::HttpClient

Details

Works for Collections as well as Cores (in SolrCloud and Standalone modes, respectively)

See Also

update_json, update_xml, update_csv for adding documents from files

Examples

```
## Not run:
(cli <- SolrClient$new())

# create the boooks collection
if (!collection_exists(cli, "books")) {
  collection_create(cli, name = "books", numShards = 1)
}

# Documents in a list
ss <- list(list(id = 1, price = 100), list(id = 2, price = 500))
add(ss, cli, name = "books")
cli$get(c(1, 2), "books")

# Documents in a data.frame
## Simple example
df <- data.frame(id = c(67, 68), price = c(1000, 500000000))
add(df, cli, "books")
df <- data.frame(id = c(77, 78), price = c(1, 2.40))
add(df, "books")

## More complex example, get file from package examples
# start Solr in Schemaless mode first: bin/solr start -e schemaless
file <- system.file("examples", "books.csv", package = "solrium")
x <- read.csv(file, stringsAsFactors = FALSE)
class(x)
head(x)
if (!collection_exists("mybooks")) {
  collection_create(name = "mybooks", numShards = 2)
}
add(x, "mybooks")

# Use modifiers
add(x, "mybooks", commit_within = 5000)

# Get back XML instead of a list
ss <- list(list(id = 1, price = 100), list(id = 2, price = 500))
# parsed XML
add(ss, name = "books", wt = "xml")
# raw XML
add(ss, name = "books", wt = "xml", raw = TRUE)

## End(Not run)
```

`collections` *List collections or cores*

Description

List collections or cores

Usage

```
collections(conn, ...)

cores(conn, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`...`	curl options passed on to crul::HttpClient

Details

Calls `collection_list()` or `core_status()` internally, and parses out names for you.

Value

character vector

Examples

```
## Not run:
# connect
(conn <- SolrClient$new())

# list collections
conn$collection_list()
collections(conn)

# list cores
conn$core_status()
cores(conn)

## End(Not run)
```

`collection_addreplica` *Add a replica*

Description

Add a replica to a shard in a collection. The node name can be specified if the replica is to be created in a specific node

Usage

```
collection_addreplica(conn, name, shard = NULL, route = NULL, node = NULL,
  instanceDir = NULL, dataDir = NULL, async = NULL, raw = FALSE,
  callopts = list(), ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`shard`	(character) The name of the shard to which replica is to be added. If `shard` is not given, then `route` must be.
`route`	(character) If the exact shard name is not known, users may pass the `route` value and the system would identify the name of the shard. Ignored if the `shard` param is also given
`node`	(character) The name of the node where the replica should be created
`instanceDir`	(character) The instanceDir for the core that will be created
`dataDir`	(character) The directory in which the core should be created
`async`	(character) Request ID to track this action which will be processed asynchronously
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://cwiki.apache.org/confluence/display/solr/Defining+core.properties)

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("foobar")) {
  conn$collection_create(name = "foobar", numShards = 2)
  # OR bin/solr create -c foobar
}

# status
```

```
conn$collection_clusterstatus()$cluster$collections$foobar

# add replica
if (!conn$collection_exists("foobar")) {
  conn$collection_addreplica(name = "foobar", shard = "shard1")
}

# status again
conn$collection_clusterstatus()$cluster$collections$foobar
conn$collection_clusterstatus()$cluster$collections$foobar$shards
conn$collection_clusterstatus()$cluster$collections$foobar$shards$shard1

## End(Not run)
```

`collection_addreplicaprop` *Add a replica property*

Description

Assign an arbitrary property to a particular replica and give it the value specified. If the property already exists, it will be overwritten with the new value.

Usage

```
collection_addreplicaprop(conn, name, shard, replica, property, property.value,
  shardUnique = FALSE, raw = FALSE, callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`shard`	(character) Required. The name of the shard the replica belongs to
`replica`	(character) Required. The replica, e.g. core_node1.
`property`	(character) Required. The property to add. Note: this will have the literal 'property.' prepended to distinguish it from system-maintained properties. So these two forms are equivalent: `property=special` and `property=property.special`
`property.value`	(character) Required. The value to assign to the property
`shardUnique`	(logical) If `TRUE`, then setting this property in one replica will (1) remove the property from all other replicas in that shard Default: `FALSE`
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("addrep")) {
  conn$collection_create(name = "addrep", numShards = 1)
  # OR bin/solr create -c addrep
}

# status
conn$collection_clusterstatus()$cluster$collections$addrep$shards

# add the value world to the property hello
conn$collection_addreplicaprop(name = "addrep", shard = "shard1",
  replica = "core_node1", property = "hello", property.value = "world")

# check status
conn$collection_clusterstatus()$cluster$collections$addrep$shards
conn$collection_clusterstatus()$cluster$collections$addrep$shards$shard1$replicas$core_node1

## End(Not run)
```

---

collection_addrole *Add a role to a node*

Description

Assign a role to a given node in the cluster. The only supported role as of 4.7 is 'overseer' . Use this API to dedicate a particular node as Overseer. Invoke it multiple times to add more nodes. This is useful in large clusters where an Overseer is likely to get overloaded . If available, one among the list of nodes which are assigned the 'overseer' role would become the overseer. The system would assign the role to any other node if none of the designated nodes are up and running

Usage

```
collection_addrole(conn, role = "overseer", node, raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
role	(character) Required. The name of the role. The only supported role as of now is overseer (set as default).
node	(character) Required. The name of the node. It is possible to assign a role even before that node is started.
raw	(logical) If TRUE, returns raw data
...	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# get list of nodes
nodes <- conn$collection_clusterstatus()$cluster$live_nodes
collection_addrole(conn, node = nodes[1])

## End(Not run)
```

`collection_balancesharduniqueue` *Balance a property*

Description

Insures that a particular property is distributed evenly amongst the physical nodes that make up a collection. If the property already exists on a replica, every effort is made to leave it there. If the property is not on any replica on a shard one is chosen and the property is added.

Usage

```
collection_balancesharduniqueue(conn, name, property, onlyactivenodes = TRUE,
  shardUnique = NULL, raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`property`	(character) Required. The property to balance. The literal "property." is prepended to this property if not specified explicitly.
`onlyactivenodes`	(logical) Normally, the property is instantiated on active nodes only. If `FALSE`, then inactive nodes are also included for distribution. Default: `TRUE`
`shardUnique`	(logical) Something of a safety valve. There is one pre-defined property (preferredLeader) that defaults this value to `TRUE`. For all other properties that are balanced, this must be set to `TRUE` or an error message is returned
`raw`	(logical) If `TRUE`, returns raw data
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("addrep")) {
  conn$collection_create(name = "mycollection")
  # OR: bin/solr create -c mycollection
}

# balance preferredLeader property
conn$collection_balanceshardunique("mycollection", property = "preferredLeader")

# examine cluster status
conn$collection_clusterstatus()$cluster$collections$mycollection

## End(Not run)
```

`collection_clusterprop` *Add, edit, delete a cluster-wide property*

Description

Important: whether add, edit, or delete is used is determined by the value passed to the `val` parameter. If the property name is new, it will be added. If the property name exists, and the value is different, it will be edited. If the property name exists, and the value is `NULL` or empty the property is deleted (unset).

Usage

```
collection_clusterprop(conn, name, val, raw = FALSE, callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) Name of the core or collection
`val`	(character) Required. The value of the property. If the value is empty or null, the property is unset.
`raw`	(logical) If `TRUE`, returns raw data in format specified by `wt` param
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# add the value https to the property urlScheme
collection_clusterprop(conn, name = "urlScheme", val = "https")

# status again
collection_clusterstatus(conn)$cluster$properties

# delete the property urlScheme by setting val to NULL or a 0 length string
collection_clusterprop(conn, name = "urlScheme", val = "")

## End(Not run)
```

`collection_clusterstatus` *Get cluster status*

Description

Fetch the cluster status including collections, shards, replicas, configuration name as well as collection aliases and cluster properties.

Usage

```
collection_clusterstatus(conn, name = NULL, shard = NULL, raw = FALSE,
  ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`shard`	(character) The shard(s) for which information is requested. Multiple shard names can be specified as a character vector.
`raw`	(logical) If `TRUE`, returns raw data
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())
conn$collection_clusterstatus()
res <- conn$collection_clusterstatus()
```

```
res$responseHeader
res$cluster
res$cluster$collections
res$cluster$collections$gettingstarted
res$cluster$live_nodes

## End(Not run)
```

`collection_create` *Add a collection*

Description

Add a collection

Usage

```
collection_create(conn, name, numShards = 1, maxShardsPerNode = 1,
  createNodeSet = NULL, collection.configName = NULL,
  replicationFactor = 1, router.name = NULL, shards = NULL,
  createNodeSet.shuffle = TRUE, router.field = NULL,
  autoAddReplicas = FALSE, async = NULL, raw = FALSE, callopts = list(),
  ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`numShards`	(integer) The number of shards to be created as part of the collection. This is a required parameter when using the 'compositeId' router.
`maxShardsPerNode`	(integer) When creating collections, the shards and/or replicas are spread across all available (i.e., live) nodes, and two replicas of the same shard will never be on the same node. If a node is not live when the CREATE operation is called, it will not get any parts of the new collection, which could lead to too many replicas being created on a single live node. Defining maxShardsPerNode sets a limit on the number of replicas CREATE will spread to each node. If the entire collection can not be fit into the live nodes, no collection will be created at all. Default: 1
`createNodeSet`	(logical) Allows defining the nodes to spread the new collection across. If not provided, the CREATE operation will create shard-replica spread across all live Solr nodes. The format is a comma-separated list of node_names, such as localhost:8983_solr, localhost:8984_solr, localhost:8985_solr. Default: `NULL`
`collection.configName`	(character) Defines the name of the configurations (which must already be stored in ZooKeeper) to use for this collection. If not provided, Solr will default to the collection name as the configuration name. Default: `compositeId`

`replicationFactor`
: (integer) The number of replicas to be created for each shard. Default: 1

`router.name`
: (character) The router name that will be used. The router defines how documents will be distributed among the shards. The value can be either `implicit`, which uses an internal default hash, or `compositeId`, which allows defining the specific shard to assign documents to. When using the 'implicit' router, the shards parameter is required. When using the 'compositeId' router, the numShards parameter is required. For more information, see also the section Document Routing. Default: `compositeId`

`shards`
: (character) A comma separated list of shard names, e.g., shard-x,shard-y,shard-z . This is a required parameter when using the 'implicit' router.

`createNodeSet.shuffle`
: (logical) Controls wether or not the shard-replicas created for this collection will be assigned to the nodes specified by the createNodeSet in a sequential manner, or if the list of nodes should be shuffled prior to creating individual replicas. A 'false' value makes the results of a collection creation predictible and gives more exact control over the location of the individual shard-replicas, but 'true' can be a better choice for ensuring replicas are distributed evenly across nodes. Ignored if createNodeSet is not also specified. Default: `TRUE`

`router.field`
: (character) If this field is specified, the router will look at the value of the field in an input document to compute the hash and identify a shard instead of looking at the uniqueKey field. If the field specified is null in the document, the document will be rejected. Please note that RealTime Get or retrieval by id would also require the parameter *route* (or shard.keys) to avoid a distributed search.

`autoAddReplicas`
: (logical) When set to true, enables auto addition of replicas on shared file systems. See the section autoAddReplicas Settings for more details on settings and overrides. Default: `FALSE`

`async`
: (character) Request ID to track this action which will be processed asynchronously

`raw`
: (logical) If `TRUE`, returns raw data

`callopts`
: curl options passed on to crul::HttpClient

`...`
: You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
# connect
(cli <- SolrClient$new())

if (!cli$collection_exists("helloWorld")) {
  cli$collection_create(name = "helloWorld")
}
if (!cli$collection_exists("tablesChairs")) {
  cli$collection_create(name = "tablesChairs")
```

```
}

## End(Not run)
```

`collection_createalias` *Create an alias for a collection*

Description

Create a new alias pointing to one or more collections. If an alias by the same name already exists, this action will replace the existing alias, effectively acting like an atomic "MOVE" command.

Usage

```
collection_createalias(conn, alias, collections, raw = FALSE,
  callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`alias`	(character) Required. The alias name to be created
`collections`	(character) Required. A character vector of collections to be aliased
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to `HttpClient`

Examples

```
## Not run:
(conn <- SolrClient$new())

if (!conn$collection_exists("thingsstuff")) {
  conn$collection_create(name = "thingsstuff")
}

conn$collection_createalias("tstuff", "thingsstuff")
conn$collection_clusterstatus()$cluster$collections$thingsstuff$aliases

## End(Not run)
```

`collection_createshard` *Create a shard*

Description

Create a shard

Usage

```
collection_createshard(conn, name, shard, createNodeSet = NULL, raw = FALSE,
  ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`shard`	(character) Required. The name of the shard to be created.
`createNodeSet`	(character) Allows defining the nodes to spread the new collection across. If not provided, the CREATE operation will create shard-replica spread across all live Solr nodes. The format is a comma-separated list of node_names, such as localhost:8983_solr, localhost:8984_s olr, localhost:8985_solr.
`raw`	(logical) If `TRUE`, returns raw data
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())
## FIXME - doesn't work right now
# conn$collection_create(name = "trees")
# conn$collection_createshard(name = "trees", shard = "newshard")

## End(Not run)
```

collection_delete *Add a collection*

Description

Add a collection

Usage

```
collection_delete(conn, name, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

if (!conn$collection_exists("helloWorld")) {
  conn$collection_create(name = "helloWorld")
}

collection_delete(conn, name = "helloWorld")

## End(Not run)
```

collection_deletealias *Delete a collection alias*

Description

Delete a collection alias

Usage

```
collection_deletealias(conn, alias, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
alias	(character) Required. The alias name to be created
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

if (!conn$collection_exists("thingsstuff")) {
  conn$collection_create(name = "thingsstuff")
}

conn$collection_createalias("tstuff", "thingsstuff")
conn$collection_clusterstatus()$cluster$collections$thingsstuff$aliases # new alias
conn$collection_deletealias("tstuff")
conn$collection_clusterstatus()$cluster$collections$thingsstuff$aliases # gone

## End(Not run)
```

---

collection_deletereplica

Delete a replica

Description

Delete a replica from a given collection and shard. If the corresponding core is up and running the core is unloaded and the entry is removed from the clusterstate. If the node/core is down , the entry is taken off the clusterstate and if the core comes up later it is automatically unregistered.

Usage

```
collection_deletereplica(conn, name, shard = NULL, replica = NULL,
  onlyIfDown = FALSE, raw = FALSE, callopts = list(), ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) Required. The name of the collection.
shard	(character) Required. The name of the shard that includes the replica to be removed.
replica	(character) Required. The name of the replica to remove.
onlyIfDown	(logical) When TRUE will not take any action if the replica is active. Default: FALSE

raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient
...	You can pass in parameters like property.name=value to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://cwiki.apache.org/confluence/display/solr/Defining+core.properties)

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("foobar2")) {
  conn$collection_create(name = "foobar2", maxShardsPerNode = 2)
}

# status
conn$collection_clusterstatus()$cluster$collections$foobar2$shards$shard1

# add replica
conn$collection_addreplica(name = "foobar2", shard = "shard1")

# delete replica
## get replica name
nms <- names(conn$collection_clusterstatus()$cluster$collections$foobar2$shards$shard1$replicas)
conn$collection_deletereplica(name = "foobar2", shard = "shard1", replica = nms[1])

# status again
conn$collection_clusterstatus()$cluster$collections$foobar2$shards$shard1

## End(Not run)
```

---

collection_deletereplicaprop

Delete a replica property

Description

Deletes an arbitrary property from a particular replica.

Usage

```
collection_deletereplicaprop(conn, name, shard, replica, property,
  raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
shard	(character) Required. The name of the shard the replica belongs to.
replica	(character) Required. The replica, e.g. core_node1.
property	(character) Required. The property to delete. Note: this will have the literal 'property.' prepended to distinguish it from system-maintained properties. So these two forms are equivalent: `property=special` and `property=property.special`
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("deleterep")) {
  conn$collection_create(name = "deleterep")
  # OR bin/solr create -c deleterep
}

# status
conn$collection_clusterstatus()$cluster$collections$deleterep$shards

# add the value bar to the property foo
conn$collection_addreplicaprop(name = "deleterep", shard = "shard1",
  replica = "core_node1", property = "foo", property.value = "bar")

# check status
conn$collection_clusterstatus()$cluster$collections$deleterep$shards
conn$collection_clusterstatus()$cluster$collections$deleterep$shards$shard1$replicas$core_node1

# delete replica property
conn$collection_deletereplicaprop(name = "deleterep", shard = "shard1",
   replica = "core_node1", property = "foo")

# check status - foo should be gone
conn$collection_clusterstatus()$cluster$collections$deleterep$shards$shard1$replicas$core_node1

## End(Not run)
```

`collection_deleteshard`

Delete a shard

Description

Deleting a shard will unload all replicas of the shard and remove them from clusterstate.json. It will only remove shards that are inactive, or which have no range given for custom sharding.

Usage

```
collection_deleteshard(conn, name, shard, raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) Required. The name of the collection that includes the shard to be deleted
shard	(character) Required. The name of the shard to be deleted
raw	(logical) If TRUE, returns raw data
...	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("buffalo")) {
  conn$collection_create(name = "buffalo")
  # OR: bin/solr create -c buffalo
}

# find shard names
names(conn$collection_clusterstatus()$cluster$collections$buffalo$shards)

# split a shard by name
collection_splitshard(conn, name = "buffalo", shard = "shard1")

# now we have three shards
names(conn$collection_clusterstatus()$cluster$collections$buffalo$shards)

# delete shard
conn$collection_deleteshard(name = "buffalo", shard = "shard1_1")

## End(Not run)
```

`collection_exists` *Check if a collection exists*

Description

Check if a collection exists

Usage

```
collection_exists(conn, name, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core. If not given, all cores.
`...`	curl options passed on to crul::HttpClient

Details

Simply calls `collection_list()` internally

Value

A single boolean, `TRUE` or `FALSE`

Examples

```
## Not run:
# start Solr with Cloud mode via the schemaless eg: bin/solr -e cloud
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below
(conn <- SolrClient$new())

# exists
conn$collection_exists("gettingstarted")

# doesn't exist
conn$collection_exists("hhhhhh")

## End(Not run)
```

collection_list *List collections*

Description

List collections

Usage

```
collection_list(conn, raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

conn$collection_list()
conn$collection_list()$collections
collection_list(conn)

## End(Not run)
```

collection_migrate *Migrate documents to another collection*

Description

Migrate documents to another collection

Usage

```
collection_migrate(conn, name, target.collection, split.key,
  forward.timeout = NULL, async = NULL, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
target.collection	(character) Required. The name of the target collection to which documents will be migrated
split.key	(character) Required. The routing key prefix. For example, if uniqueKey is a!123, then you would use split.key=a!
forward.timeout	(integer) The timeout (seconds), until which write requests made to the source collection for the given `split.key` will be forwarded to the target shard. Default: 60
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If `TRUE`, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("migrate_from")) {
  conn$collection_create(name = "migrate_from")
  # OR: bin/solr create -c migrate_from
}

# create another collection
if (!conn$collection_exists("migrate_to")) {
  conn$collection_create(name = "migrate_to")
  # OR bin/solr create -c migrate_to
}

# add some documents
file <- system.file("examples", "books.csv", package = "solrium")
x <- read.csv(file, stringsAsFactors = FALSE)
conn$add(x, "migrate_from")

# migrate some documents from one collection to the other
## FIXME - not sure if this is actually working....
# conn$collection_migrate("migrate_from", "migrate_to", split.key = "05535")

## End(Not run)
```

`collection_overseerstatus` *Get overseer status*

Description

Returns the current status of the overseer, performance statistics of various overseer APIs as well as last 10 failures per operation type.

Usage

```
collection_overseerstatus(conn, raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`raw`	(logical) If `TRUE`, returns raw data
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())
conn$collection_overseerstatus()
res <- conn$collection_overseerstatus()
res$responseHeader
res$leader
res$overseer_queue_size
res$overseer_work_queue_size
res$overseer_operations
res$collection_operations
res$overseer_queue
res$overseer_internal_queue
res$collection_queue

## End(Not run)
```

`collection_rebalanceleaders` *Rebalance leaders*

Description

Reassign leaders in a collection according to the preferredLeader property across active nodes

Usage

```
collection_rebalanceleaders(conn, name, maxAtOnce = NULL,
  maxWaitSeconds = NULL, raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`maxAtOnce`	(integer) The maximum number of reassignments to have queue up at once. Values <=0 are use the default value Integer.MAX_VALUE. When this number is reached, the process waits for one or more leaders to be successfully assigned before adding more to the queue.
`maxWaitSeconds`	(integer) Timeout value when waiting for leaders to be reassigned. NOTE: if maxAtOnce is less than the number of reassignments that will take place, this is the maximum interval that any single wait for at least one reassignment. For example, if 10 reassignments are to take place and maxAtOnce is 1 and maxWaitSeconds is 60, the upper bound on the time that the command may wait is 10 minutes. Default: 60
`raw`	(logical) If `TRUE`, returns raw data
`...`	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("mycollection2")) {
  conn$collection_create(name = "mycollection2")
  # OR: bin/solr create -c mycollection2
}

# balance preferredLeader property
conn$collection_balancesharduniique("mycollection2", property = "preferredLeader")
```

```
# balance preferredLeader property
conn$collection_rebalanceleaders("mycollection2")

# examine cluster status
conn$collection_clusterstatus()$cluster$collections$mycollection2

## End(Not run)
```

`collection_reload` *Reload a collection*

Description

Reload a collection

Usage

```
collection_reload(conn, name, raw = FALSE, callopts)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

if (!conn$collection_exists("helloWorld")) {
  conn$collection_create(name = "helloWorld")
}

conn$collection_reload(name = "helloWorld")

## End(Not run)
```

collection_removerole *Remove a role from a node*

Description

Remove an assigned role. This API is used to undo the roles assigned using `collection_addrole`

Usage

```
collection_removerole(conn, role = "overseer", node, raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
role	(character) Required. The name of the role. The only supported role as of now is overseer (set as default).
node	(character) Required. The name of the node.
raw	(logical) If TRUE, returns raw data
...	You can pass in parameters like `property.name=value` to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

Examples

```
## Not run:
(conn <- SolrClient$new())

# get list of nodes
nodes <- conn$collection_clusterstatus()$cluster$live_nodes
conn$collection_addrole(node = nodes[1])
conn$collection_removerole(node = nodes[1])

## End(Not run)
```

collection_requeststatus *Get request status*

Description

Request the status of an already submitted Asynchronous Collection API call. This call is also used to clear up the stored statuses.

Usage

```
collection_requeststatus(conn, requestid, raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
requestid	(character) Required. The user defined request-id for the request. This can be used to track the status of the submitted asynchronous task. -1 is a special request id which is used to cleanup the stored states for all of the already completed/failed tasks.
raw	(logical) If TRUE, returns raw data
...	You can pass in parameters like property.name=value to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/7_0/defining-core-properties.html)

---

collection_splitshard *Create a shard*

Description

Create a shard

Usage

```
collection_splitshard(conn, name, shard, ranges = NULL, split.key = NULL,
  async = NULL, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
shard	(character) Required. The name of the shard to be split
ranges	(character) A comma-separated list of hash ranges in hexadecimal e.g. ranges=0-1f4,1f5-3e8,3e9-5dc
split.key	(character) The key to use for splitting the index
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

# create collection
if (!conn$collection_exists("trees")) {
  conn$collection_create("trees")
}

# find shard names
names(conn$collection_clusterstatus()$cluster$collections$trees$shards)

# split a shard by name
conn$collection_splitshard(name = "trees", shard = "shard1")

# now we have three shards
names(conn$collection_clusterstatus()$cluster$collections$trees$shards)

## End(Not run)
```

---

commit *Commit*

Description

Commit

Usage

```
commit(conn, name, expunge_deletes = FALSE, wait_searcher = TRUE,
  soft_commit = FALSE, wt = "json", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) A collection or core name. Required.
expunge_deletes	merge segments with deletes away. Default: FALSE
wait_searcher	block until a new searcher is opened and registered as the main query searcher, making the changes visible. Default: TRUE
soft_commit	perform a soft commit - this will refresh the 'view' of the index in a more performant manner, but without "on-disk" guarantees. Default: FALSE
wt	(character) One of json (default) or xml. If json, uses jsonlite::fromJSON() to parse. If xml, uses xml2::read_xml() to parse
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to crul::HttpClient

References

<>

Examples

```
## Not run:
(conn <- SolrClient$new())

conn$commit("gettingstarted")
conn$commit("gettingstarted", wait_searcher = FALSE)

# get xml back
conn$commit("gettingstarted", wt = "xml")
## raw xml
conn$commit("gettingstarted", wt = "xml", raw = TRUE)

## End(Not run)
```

`config_get` *Get Solr configuration details*

Description

Get Solr configuration details

Usage

```
config_get(conn, name, what = NULL, wt = "json", raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core. If not given, all cores.
`what`	(character) What you want to look at. One of solrconfig or schema. Default: solrconfig
`wt`	(character) One of json (default) or xml. Data type returned. If json, uses `fromJSON` to parse. If xml, uses `read_xml` to parse.
`raw`	(logical) If `TRUE`, returns raw data in format specified by `wt`
`...`	curl options passed on to crul::HttpClient

Details

Note that if `raw=TRUE`, `what` is ignored. That is, you get all the data when `raw=TRUE`.

Value

A list, `xml_document`, or character

Examples

```
## Not run:
# start Solr with Cloud mode via the schemaless eg: bin/solr -e cloud
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# all config settings
conn$config_get("gettingstarted")

# just znodeVersion
conn$config_get("gettingstarted", "znodeVersion")

# just znodeVersion
conn$config_get("gettingstarted", "luceneMatchVersion")

# just updateHandler
conn$config_get("gettingstarted", "updateHandler")

# just updateHandler
conn$config_get("gettingstarted", "requestHandler")

## Get XML
conn$config_get("gettingstarted", wt = "xml")
conn$config_get("gettingstarted", "updateHandler", wt = "xml")
conn$config_get("gettingstarted", "requestHandler", wt = "xml")

## Raw data - what param ignored when raw=TRUE
conn$config_get("gettingstarted", raw = TRUE)

## End(Not run)
```

`config_overlay` *Get Solr configuration overlay*

Description

Get Solr configuration overlay

Usage

```
config_overlay(conn, name, omitHeader = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core. If not given, all cores.

omitHeader	(logical) If TRUE, omit header. Default: FALSE
...	curl options passed on to crul::HttpClient

Value

A list with response from server

Examples

```
## Not run:
# start Solr with Cloud mode via the schemaless eg: bin/solr -e cloud
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# get config overlay
conn$config_overlay("gettingstarted")

# without header
conn$config_overlay("gettingstarted", omitHeader = TRUE)

## End(Not run)
```

`config_params` *Set Solr configuration params*

Description

Set Solr configuration params

Usage

```
config_params(conn, name, param = NULL, set = NULL, unset = NULL,
  update = NULL, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core. If not given, all cores.
param	(character) Name of a parameter
set	(list) List of key:value pairs of what to set. Create or overwrite a parameter set map. Default: NULL (nothing passed)
unset	(list) One or more character strings of keys to unset. Default: NULL (nothing passed)

update	(list) List of key:value pairs of what to update. Updates a parameter set map. This essentially overwrites the old parameter set, so all parameters must be sent in each update request.
...	curl options passed on to crul::HttpClient

Details

The Request Parameters API allows creating parameter sets that can override or take the place of parameters defined in solrconfig.xml. It is really another endpoint of the Config API instead of a separate API, and has distinct commands. It does not replace or modify any sections of solrconfig.xml, but instead provides another approach to handling parameters used in requests. It behaves in the same way as the Config API, by storing parameters in another file that will be used at runtime. In this case, the parameters are stored in a file named params.json. This file is kept in ZooKeeper or in the conf directory of a standalone Solr instance.

Value

A list with response from server

Examples

```
## Not run:
# start Solr in standard or Cloud mode
# connect
(conn <- SolrClient$new())

# set a parameter set
myFacets <- list(myFacets = list(facet = TRUE, facet.limit = 5))
config_params(conn, "gettingstarted", set = myFacets)

# check a parameter
config_params(conn, "gettingstarted", param = "myFacets")

## End(Not run)
```

---

config_set *Set Solr configuration details*

Description

Set Solr configuration details

Usage

```
config_set(conn, name, set = NULL, unset = NULL, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core. If not given, all cores.
set	(list) List of key:value pairs of what to set. Default: NULL (nothing passed)
unset	(list) One or more character strings of keys to unset. Default: NULL (nothing passed)
...	curl options passed on to crul::HttpClient

Value

A list with response from server

Examples

```
## Not run:
# start Solr with Cloud mode via the schemaless eg: bin/solr -e cloud
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# set a property
conn$config_set("gettingstarted",
  set = list(query.filterCache.autowarmCount = 1000))

# unset a property
conn$config_set("gettingstarted", unset = "query.filterCache.size",
  verbose = TRUE)

# both set a property and unset a property
conn$config_set("gettingstarted", unset = "enableLazyFieldLoading")

# many properties
conn$config_set("gettingstarted", set = list(
   query.filterCache.autowarmCount = 1000,
   query.commitWithin.softCommit = 'false'
 )
)

## End(Not run)
```

---

core_create *Create a core*

Description

Create a core

Usage

```
core_create(conn, name, instanceDir = NULL, config = NULL, schema = NULL,
  dataDir = NULL, configSet = NULL, collection = NULL, shard = NULL,
  async = NULL, raw = FALSE, callopts = list(), ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
instanceDir	(character) Path to instance directory
config	(character) Path to config file
schema	(character) Path to schema file
dataDir	(character) Name of the data directory relative to instanceDir.
configSet	(character) Name of the configset to use for this core. For more information, see https://lucene.apache.org/solr/guide/6_6/config-sets.html
collection	(character) The name of the collection to which this core belongs. The default is the name of the core. collection.<param>=<val ue> causes a property of <param>=<value> to be set if a new collection is being created. Use collection.configNa me=<configname> to point to the configuration for a new collection.
shard	(character) The shard id this core represents. Normally you want to be auto-assigned a shard id.
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient
...	You can pass in parameters like property.name=value to set core property name to value. See the section Defining core.properties for details on supported properties and values. (https://lucene.apache.org/solr/guide/6_6/defining-core-properties.html)

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or create as below

# connect
(conn <- SolrClient$new())

# Create a core
path <- "~/solr-7.0.0/server/solr/newcore/conf"
dir.create(path, recursive = TRUE)
files <- list.files("~/solr-7.0.0/server/solr/configsets/sample_techproducts_configs/conf/",
full.names = TRUE)
```

```
invisible(file.copy(files, path, recursive = TRUE))
conn$core_create(name = "newcore", instanceDir = "newcore",
  configSet = "sample_techproducts_configs")

## End(Not run)
```

`core_exists` *Check if a core exists*

Description

Check if a core exists

Usage

```
core_exists(conn, name, callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`callopts`	curl options passed on to crul::HttpClient

Details

Simply calls `core_status()` internally

Value

A single boolean, `TRUE` or `FALSE`

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or create as below

# connect
(conn <- SolrClient$new())

# exists
conn$core_exists("gettingstarted")

# doesn't exist
conn$core_exists("hhhhhh")

## End(Not run)
```

`core_mergeindexes` *Merge indexes (cores)*

Description

Merges one or more indexes to another index. The indexes must have completed commits, and should be locked against writes until the merge is complete or the resulting merged index may become corrupted. The target core index must already exist and have a compatible schema with the one or more indexes that will be merged to it.

Usage

```
core_mergeindexes(conn, name, indexDir = NULL, srcCore = NULL,
  async = NULL, raw = FALSE, callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`indexDir`	(character) Multi-valued, directories that would be merged.
`srcCore`	(character) Multi-valued, source cores that would be merged.
`async`	(character) Request ID to track this action which will be processed asynchronously
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#  bin/solr start -e schemaless

# connect
(conn <- SolrClient$new())

## FIXME: not tested yet

# use indexDir parameter
conn$core_mergeindexes(core="new_core_name",
   indexDir = c("/solr_home/core1/data/index",
   "/solr_home/core2/data/index"))

# use srcCore parameter
conn$core_mergeindexes(name = "new_core_name", srcCore = c('core1', 'core2'))

## End(Not run)
```

`core_reload` *Reload a core*

Description

Reload a core

Usage

```
core_reload(conn, name, raw = FALSE, callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#  bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# Status of particular cores
conn$core_reload("gettingstarted")
conn$core_status("gettingstarted")

## End(Not run)
```

`core_rename` *Rename a core*

Description

Rename a core

Usage

```
core_rename(conn, name, other, async = NULL, raw = FALSE,
  callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
other	(character) The new name of the core. Required.
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# Status of particular cores
path <- "~/solr-7.0.0/server/solr/testcore/conf"
dir.create(path, recursive = TRUE)
files <- list.files(
"~/solr-7.0.0/server/solr/configsets/sample_techproducts_configs/conf/",
full.names = TRUE)
invisible(file.copy(files, path, recursive = TRUE))
conn$core_create("testcore") # or create in CLI: bin/solr create -c testcore

# rename
conn$core_rename("testcore", "newtestcore")
## status
conn$core_status("testcore") # core missing
conn$core_status("newtestcore", FALSE) # not missing

# cleanup
conn$core_unload("newtestcore")

## End(Not run)
```

---

core_requeststatus *Request status of asynchronous CoreAdmin API call*

Description

Request status of asynchronous CoreAdmin API call

Usage

```
core_requeststatus(conn, requestid, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
requestid	The name of one of the cores to be removed. Required
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless

# FIXME: not tested yet...
# (conn <- SolrClient$new())
# conn$core_requeststatus(requestid = 1)

## End(Not run)
```

---

core_split *Split a core*

Description

SPLIT splits an index into two or more indexes. The index being split can continue to handle requests. The split pieces can be placed into a specified directory on the server's filesystem or it can be merged into running Solr cores.

Usage

```
core_split(conn, name, path = NULL, targetCore = NULL, ranges = NULL,
  split.key = NULL, async = NULL, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
path	(character) Two or more target directory paths in which a piece of the index will be written
targetCore	(character) Two or more target Solr cores to which a piece of the index will be merged

ranges	(character) A list of number ranges, or hash ranges in hexadecimal format. If numbers, they get converted to hexidecimal format before being passed to your Solr server.
split.key	(character) The key to be used for splitting the index
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Details

The core index will be split into as many pieces as the number of path or targetCore parameters.

Either path or targetCore parameter must be specified but not both. The ranges and split.key parameters are optional and only one of the two should be specified, if at all required.

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg: bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# Swap a core
## First, create two cores
# conn$core_split("splitcoretest0") # or create in the CLI: bin/solr create -c splitcoretest0
# conn$core_split("splitcoretest1") # or create in the CLI: bin/solr create -c splitcoretest1
# conn$core_split("splitcoretest2") # or create in the CLI: bin/solr create -c splitcoretest2

## check status
conn$core_status("splitcoretest0", FALSE)
conn$core_status("splitcoretest1", FALSE)
conn$core_status("splitcoretest2", FALSE)

## split core using targetCore parameter
conn$core_split("splitcoretest0", targetCore = c("splitcoretest1", "splitcoretest2"))

## split core using split.key parameter
### Here all documents having the same route key as the split.key i.e. 'A!'
### will be split from the core index and written to the targetCore
conn$core_split("splitcoretest0", targetCore = "splitcoretest1", split.key = "A!")

## split core using ranges parameter
### Solr expects hash ranges in hexidecimal, but since we're in R,
### let's not make our lives any harder, so you can pass in numbers
### but you can still pass in hexidecimal if you want.
rgs <- c('0-1f4', '1f5-3e8')
conn$core_split("splitcoretest0", targetCore = c("splitcoretest1", "splitcoretest2"), ranges = rgs)
rgs <- list(c(0, 500), c(501, 1000))
```

```
conn$core_split("splitcoretest0", targetCore = c("splitcoretest1", "splitcoretest2"), ranges = rgs)

## End(Not run)
```

`core_status` *Get core status*

Description

Get core status

Usage

```
core_status(conn, name = NULL, indexInfo = TRUE, raw = FALSE,
  callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`indexInfo`	(logical)
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# Status of all cores
conn$core_status()

# Status of particular cores
conn$core_status("gettingstarted")

# Get index info or not
## Default: TRUE
conn$core_status("gettingstarted", indexInfo = TRUE)
conn$core_status("gettingstarted", indexInfo = FALSE)

## End(Not run)
```

core_swap *Swap a core*

Description

SWAP atomically swaps the names used to access two existing Solr cores. This can be used to swap new content into production. The prior core remains available and can be swapped back, if necessary. Each core will be known by the name of the other, after the swap

Usage

```
core_swap(conn, name, other, async = NULL, raw = FALSE, callopts = list())
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) The name of the core to be created. Required
other	(character) The name of one of the cores to be swapped. Required.
async	(character) Request ID to track this action which will be processed asynchronously
raw	(logical) If TRUE, returns raw data
callopts	curl options passed on to crul::HttpClient

Details

Do not use core_swap with a SolrCloud node. It is not supported and can result in the core being unusable. We'll try to stop you if you try.

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless
# you can create a new core like: bin/solr create -c corename
# where <corename> is the name for your core - or creaate as below

# connect
(conn <- SolrClient$new())

# Swap a core
## First, create two cores
conn$core_create("swapcoretest1")
# - or create on CLI: bin/solr create -c swapcoretest1
conn$core_create("swapcoretest2")
# - or create on CLI: bin/solr create -c swapcoretest2

## check status
conn$core_status("swapcoretest1", FALSE)
```

```
conn$core_status("swapcoretest2", FALSE)

## swap core
conn$core_swap("swapcoretest1", "swapcoretest2")

## check status again
conn$core_status("swapcoretest1", FALSE)
conn$core_status("swapcoretest2", FALSE)

## End(Not run)
```

`core_unload` *Unload (delete) a core*

Description

Unload (delete) a core

Usage

```
core_unload(conn, name, deleteIndex = FALSE, deleteDataDir = FALSE,
  deleteInstanceDir = FALSE, async = NULL, raw = FALSE,
  callopts = list())
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) The name of the core to be created. Required
`deleteIndex`	(logical) If `TRUE`, will remove the index when unloading the core. Default: `FALSE`
`deleteDataDir`	(logical) If `TRUE`, removes the data directory and all sub-directories. Default: `FALSE`
`deleteInstanceDir`	(logical) If `TRUE`, removes everything related to the core, including the index directory, configuration files and other related files. Default: `FALSE`
`async`	(character) Request ID to track this action which will be processed asynchronously
`raw`	(logical) If `TRUE`, returns raw data
`callopts`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr with Schemaless mode via the schemaless eg:
#   bin/solr start -e schemaless

# connect
(conn <- SolrClient$new())
```

```
# Create a core
conn$core_create(name = "books")

# Unload a core
conn$core_unload(name = "books")
## not found
# conn$core_unload(name = "books")
# > Error: 400 - Cannot unload non-existent core [books]

## End(Not run)
```

`delete` *Delete documents by ID or query*

Description

Delete documents by ID or query

Usage

```
delete_by_id(conn, ids, name, commit = TRUE, commit_within = NULL,
  overwrite = TRUE, boost = NULL, wt = "json", raw = FALSE, ...)

delete_by_query(conn, query, name, commit = TRUE, commit_within = NULL,
  overwrite = TRUE, boost = NULL, wt = "json", raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`ids`	Document IDs, one or more in a vector or list
`name`	(character) A collection or core name. Required.
`commit`	(logical) If `TRUE`, documents immediately searchable. Deafult: `TRUE`
`commit_within`	(numeric) Milliseconds to commit the change, the document will be added within that time. Default: `NULL`
`overwrite`	(logical) Overwrite documents with matching keys. Default: `TRUE`
`boost`	(numeric) Boost factor. Default: `NULL`
`wt`	(character) One of json (default) or xml. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses `xml2::read_xml()` to parse
`raw`	(logical) If `TRUE`, returns raw data in format specified by `wt` param
`...`	curl options passed on to crul::HttpClient
`query`	Query to use to delete documents

Details

We use json internally as data interchange format for this function.

Examples

```
## Not run:
(cli <- SolrClient$new())

# add some documents first
ss <- list(list(id = 1, price = 100), list(id = 2, price = 500))
cli$add(ss, name = "gettingstarted")

# Now, delete them
# Delete by ID
cli$delete_by_id(ids = 1, "gettingstarted")
## Many IDs
cli$delete_by_id(ids = c(3, 4), "gettingstarted")

# Delete by query
cli$delete_by_query(query = "manu:bank", "gettingstarted")

## End(Not run)
```

---

is.sr_facet *Test for sr_facet class*

Description

Test for sr_facet class

Test for sr_high class

Test for sr_search class

Usage

```
is.sr_facet(x)

is.sr_high(x)

is.sr_search(x)
```

Arguments

x	Input

makemultiargs — *Function to make make multiple args of the same name from a single input with length > 1*

Description

Function to make make multiple args of the same name from a single input with length > 1

Usage

```
makemultiargs(x)
```

Arguments

x	Value

ping — *Ping a Solr instance*

Description

Ping a Solr instance

Usage

```
ping(conn, name, wt = "json", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) Name of a collection or core. Required.
wt	(character) One of json (default) or xml. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses [xml2::read_xml)] to parse [xml2::read_xml)]: R:xml2::read_xml)
raw	(logical) If `TRUE`, returns raw data in format specified by `wt` param
...	curl options passed on to crul::HttpClient

Details

You likely may not be able to run this function against many public Solr services as they hopefully don't expose their admin interface to the public, but works locally.

Value

if `wt="xml"` an object of class `xml_document`, if `wt="json"` an object of class `list`

Examples

```
## Not run:
# start Solr, in your CLI, run: `bin/solr start -e cloud -noprompt`
# after that, if you haven't run `bin/post -c gettingstarted docs/` yet,
# do so

# connect: by default we connect to localhost, port 8983
(cli <- SolrClient$new())

# ping the gettingstarted index
cli$ping("gettingstarted")
ping(cli, "gettingstarted")
ping(cli, "gettingstarted", wt = "xml")
ping(cli, "gettingstarted", verbose = FALSE)
ping(cli, "gettingstarted", raw = TRUE)

ping(cli, "gettingstarted", wt="xml", verbose = TRUE)

## End(Not run)
```

---

schema *Get the schema for a collection or core*

Description

Get the schema for a collection or core

Usage

```
schema(conn, name, what = "", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	(character) Name of a collection or core. Required.
what	(character) What to retrieve. By default, we retrieve the entire schema. Options include: fields, dynamicfields, fieldtypes, copyfields, name, version, uniquekey, similarity, "solrqueryparser/defaultoperator"
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to crul::HttpClient

Examples

```
## Not run:
# start Solr, in your CLI, run: `bin/solr start -e cloud -noprompt`
# after that, if you haven't run `bin/post -c gettingstarted docs/` yet, do so
```

```
# connect: by default we connect to localhost, port 8983
(cli <- SolrClient$new())

# get the schema for the gettingstarted index
schema(cli, name = "gettingstarted")

# Get parts of the schema
schema(cli, name = "gettingstarted", "fields")
schema(cli, name = "gettingstarted", "dynamicfields")
schema(cli, name = "gettingstarted", "fieldtypes")
schema(cli, name = "gettingstarted", "copyfields")
schema(cli, name = "gettingstarted", "name")
schema(cli, name = "gettingstarted", "version")
schema(cli, name = "gettingstarted", "uniquekey")
schema(cli, name = "gettingstarted", "similarity")
schema(cli, name = "gettingstarted", "solrqueryparser/defaultoperator")

# get raw data
schema(cli, name = "gettingstarted", "similarity", raw = TRUE)
schema(cli, name = "gettingstarted", "uniquekey", raw = TRUE)

# start Solr in Schemaless mode: bin/solr start -e schemaless
# schema(cli, "gettingstarted")

# start Solr in Standalone mode: bin/solr start
# then add a core: bin/solr create -c helloWorld
# schema(cli, "helloWorld")

## End(Not run)
```

`SolrClient` *Solr connection client*

Description

Solr connection client

Arguments

`host`	(character) Host url. Deafault: 127.0.0.1
`path`	(character) url path.
`port`	(character/numeric) Port. Default: 8389
`scheme`	(character) http scheme, one of http or https. Default: http
`proxy`	List of arguments for a proxy connection, including one or more of: url, port, username, password, and auth. See crul::proxy for help, which is used to construct the proxy connection.
`errors`	(character) One of `"simple"` or `"complete"`. Simple gives http code and error message on an error, while complete gives both http code and error message, and stack trace, if available.

Details

`SolrClient` creates a R6 class object. The object is not cloneable and is portable, so it can be inherited across packages without complication.

`SolrClient` is used to initialize a client that knows about your Solr instance, with options for setting host, port, http scheme, and simple vs. complete error reporting

Value

Various output, see help files for each grouping of methods.

SolrClient methods

Each of these methods also has a matching standalone exported function that you can use by passing in the connection object made by calling `SolrClient$new()`. Also, see the docs for each method for parameter definitions and their default values.

- `ping(name, wt = 'json', raw = FALSE, ...)`
- `schema(name, what = '', raw = FALSE, ...)`
- `commit(name, expunge_deletes = FALSE, wait_searcher = TRUE, soft_commit = FALSE, wt = 'json', raw`
- `optimize(name, max_segments = 1, wait_searcher = TRUE, soft_commit = FALSE, wt = 'json', raw = FAL`
- `config_get(name, what = NULL, wt = "json", raw = FALSE, ...)`
- `config_params(name, param = NULL, set = NULL, unset = NULL, update = NULL, ...)`
- `config_overlay(name, omitHeader = FALSE, ...)`
- `config_set(name, set = NULL, unset = NULL, ...)`
- `collection_exists(name, ...)`
- `collection_list(raw = FALSE, ...)`
- `collection_create(name, numShards = 1, maxShardsPerNode = 1, createNodeSet = NULL, collection.con`
- `collection_addreplica(name, shard = NULL, route = NULL, node = NULL, instanceDir = NULL, dataDir =`
- `collection_addreplicaprop(name, shard, replica, property, property.value, shardUnique = FALSE, ra`
- `collection_addrole(role = "overseer", node, raw = FALSE, ...)`
- `collection_balanceshardunique(name, property, onlyactivenodes = TRUE, shardUnique = NULL, raw = F`
- `collection_clusterprop(name, val, raw = FALSE, callopts=list())`
- `collection_clusterstatus(name = NULL, shard = NULL, raw = FALSE, ...)`
- `collection_createalias(alias, collections, raw = FALSE, ...)`
- `collection_createshard(name, shard, createNodeSet = NULL, raw = FALSE, ...)`
- `collection_delete(name, raw = FALSE, ...)`
- `collection_deletealias(alias, raw = FALSE, ...)`
- `collection_deletereplica(name, shard = NULL, replica = NULL, onlyIfDown = FALSE, raw = FALSE, cal`
- `collection_deletereplicaprop(name, shard, replica, property, raw = FALSE, callopts=list())`
- `collection_deleteshard(name, shard, raw = FALSE, ...)`
- `collection_migrate(name, target.collection, split.key, forward.timeout = NULL, async = NULL, raw`

- collection_overseerstatus(raw = FALSE, ...)
- collection_rebalanceleaders(name, maxAtOnce = NULL, maxWaitSeconds = NULL, raw = FALSE, ...)
- collection_reload(name, raw = FALSE, ...)
- collection_removerole(role = "overseer", node, raw = FALSE, ...)
- collection_requeststatus(requestid, raw = FALSE, ...)
- collection_splitshard(name, shard, ranges = NULL, split.key = NULL, async = NULL, raw = FALSE, ...
- core_status(name = NULL, indexInfo = TRUE, raw = FALSE, callopts=list())
- core_exists(name, callopts = list())
- core_create(name, instanceDir = NULL, config = NULL, schema = NULL, dataDir = NULL, configSet = NU
- core_unload(name, deleteIndex = FALSE, deleteDataDir = FALSE, deleteInstanceDir = FALSE, async =
- core_rename(name, other, async = NULL, raw = FALSE, callopts=list())
- core_reload(name, raw = FALSE, callopts=list())
- core_swap(name, other, async = NULL, raw = FALSE, callopts=list())
- core_mergeindexes(name, indexDir = NULL, srcCore = NULL, async = NULL, raw = FALSE, callopts = lis
- core_requeststatus(requestid, raw = FALSE, callopts = list())
- core_split(name, path = NULL, targetCore = NULL, ranges = NULL, split.key = NULL, async = NULL, ra
- search(name = NULL, params = NULL, body = NULL, callopts = list(), raw = FALSE, parsetype = 'df',
- facet(name = NULL, params = NULL, body = NULL, callopts = list(), raw = FALSE, parsetype = 'df', co
- stats(name = NULL, params = list(q = '*:*', stats.field = NULL, stats.facet = NULL), body = NULL,
- highlight(name = NULL, params = NULL, body = NULL, callopts=list(), raw = FALSE, parsetype = 'df',
- group(name = NULL, params = NULL, body = NULL, callopts=list(), raw=FALSE, parsetype='df', concat
- mlt(name = NULL, params = NULL, body = NULL, callopts=list(), raw=FALSE, parsetype='df', concat='
- all(name = NULL, params = NULL, body = NULL, callopts=list(), raw=FALSE, parsetype='df', concat='
- get(ids, name, fl = NULL, wt = 'json', raw = FALSE, ...)
- add(x, name, commit = TRUE, commit_within = NULL, overwrite = TRUE, boost = NULL, wt = 'json', raw
- delete_by_id(ids, name, commit = TRUE, commit_within = NULL, overwrite = TRUE, boost = NULL, wt =
- delete_by_query(query, name, commit = TRUE, commit_within = NULL, overwrite = TRUE, boost = NULL,
- update_json(files, name, commit = TRUE, optimize = FALSE, max_segments = 1, expunge_deletes = FALS
- update_xml(files, name, commit = TRUE, optimize = FALSE, max_segments = 1, expunge_deletes = FALSE
- update_csv(files, name, separator = ',', header = TRUE, fieldnames = NULL, skip = NULL, skipLines
- update_atomic_json(body, name, wt = 'json', raw = FALSE, ...)
- update_atomic_xml(body, name, wt = 'json', raw = FALSE, ...)

Examples

```
## Not run:
# make a client
(cli <- SolrClient$new())

# variables
cli$host
cli$port
cli$path
cli$scheme

# ping
## ping to make sure it's up
cli$ping("gettingstarted")

# version
## get Solr version information
cli$schema("gettingstarted")
cli$schema("gettingstarted", "fields")
cli$schema("gettingstarted", "name")
cli$schema("gettingstarted", "version")$version

# Search
cli$search("gettingstarted", params = list(q = "*:*"))
cli$search("gettingstarted", body = list(query = "*:*"))

# set a different host
SolrClient$new(host = 'stuff.com')

# set a different port
SolrClient$new(host = 3456)

# set a different http scheme
SolrClient$new(scheme = 'https')

# set a proxy
SolrClient$new(proxy = list(url = "187.62.207.130:3128"))

prox <- list(url = "187.62.207.130:3128", user = "foo", pwd = "bar")
cli <- SolrClient$new(proxy = prox)
cli$proxy

# A remote Solr instance to which you don't have admin access
(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))
cli$search(params = list(q = "memory"))

## End(Not run)
```

`solr_all` *All purpose search*

Description

Includes documents, facets, groups, mlt, stats, and highlights

Usage

```
solr_all(conn, name = NULL, params = NULL, body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", concat = ",",
  optimizeMaxRows = TRUE, minOptimizedRows = 50000L, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	Name of a collection or core. Or leave as NULL if not needed.
params	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
body	(list) a named list of parameters, if given a POST request will be performed
callopts	Call options passed on to [crul::HttpClient]
raw	(logical) If TRUE, returns raw data in format specified by wt param
parsetype	(character) One of 'list' or 'df'
concat	(character) Character to concatenate elements of longer than length 1. Note that this only works reliably when data format is json (wt='json'). The parsing is more complicated in XML format, but you can do that on your own.
optimizeMaxRows	(logical) If TRUE, then rows parameter will be adjusted to the number of returned results by the same constraints. It will only be applied if rows parameter is higher than minOptimizedRows. Default: TRUE
minOptimizedRows	(numeric) used by optimizedMaxRows parameter, the minimum optimized rows. Default: 50000
...	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

Parameters

- q Query terms, defaults to '*:*', or everything.
- sort Field to sort on. You can specify ascending (e.g., score desc) or descending (e.g., score asc), sort by two fields (e.g., score desc, price asc), or sort by a function (e.g., sum(x_f, y_f) desc, which sorts by the sum of x_f and y_f in a descending order).
- start Record to start at, default to beginning.
- rows Number of records to return. Default: 10.

- pageDoc If you expect to be paging deeply into the results (say beyond page 10, assuming rows=10) and you are sorting by score, you may wish to add the pageDoc and pageScore parameters to your request. These two parameters tell Solr (and Lucene) what the last result (Lucene internal docid and score) of the previous page was, so that when scoring the query for the next set of pages, it can ignore any results that occur higher than that item. To get the Lucene internal doc id, you will need to add [docid] to the &fl list. e.g., q=*:*&start=10&pageDoc=5&pageScore=1.345
- pageScore See pageDoc notes.
- fq Filter query, this does not affect the search, only what gets returned. This parameter can accept multiple items in a lis or vector. You can't pass more than one parameter of the same name, so we get around it by passing multiple queries and we parse internally
- fl Fields to return, can be a character vector like `c('id', 'title')`, or a single character vector with one or more comma separated names, like `'id,title'`
- defType Specify the query parser to use with this request.
- timeAllowed The time allowed for a search to finish. This value only applies to the search and not to requests in general. Time is in milliseconds. Values <= 0 mean no time restriction. Partial results may be returned (if there are any).
- qt Which query handler used. Options: dismax, others?
- NOW Set a fixed time for evaluating Date based expresions
- TZ Time zone, you can override the default.
- echoHandler If `TRUE`, Solr places the name of the handle used in the response to the client for debugging purposes. Default:
- echoParams The echoParams parameter tells Solr what kinds of Request parameters should be included in the response for debugging purposes, legal values include:
  - none - don't include any request parameters for debugging
  - explicit - include the parameters explicitly specified by the client in the request
  - all - include all parameters involved in this request, either specified explicitly by the client, or implicit because of the request handler configuration.
- wt (character) One of json, xml, or csv. Data type returned, defaults to 'csv'. If json, uses [jsonlite::fromJSON()] to parse. If xml, uses [xml2::read_xml()] to parse. If csv, uses [read.table()] to parse. 'wt=csv' gives the fastest performance at least in all the cases we have tested in, thus it's the default value for 'wt'

References

See http://wiki.apache.org/solr/#Search_and_Indexing for more information.

See Also

solr_highlight(), solr_facet()

Examples

```
## Not run:
# connect
(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))
```

```
solr_all(cli, params = list(q='*:*', rows=2, fl='id'))

# facets
solr_all(cli, params = list(q='*:*', rows=2, fl='id', facet="true",
  facet.field="journal"))

# mlt
solr_all(cli, params = list(q='ecology', rows=2, fl='id', mlt='true',
  mlt.count=2, mlt.fl='abstract'))

# facets and mlt
solr_all(cli, params = list(q='ecology', rows=2, fl='id', facet="true",
  facet.field="journal", mlt='true', mlt.count=2, mlt.fl='abstract'))

# stats
solr_all(cli, params = list(q='ecology', rows=2, fl='id', stats='true',
  stats.field='counter_total_all'))

# facets, mlt, and stats
solr_all(cli, params = list(q='ecology', rows=2, fl='id', facet="true",
  facet.field="journal", mlt='true', mlt.count=2, mlt.fl='abstract',
  stats='true', stats.field='counter_total_all'))

# group
solr_all(cli, params = list(q='ecology', rows=2, fl='id', group='true',
 group.field='journal', group.limit=3))

# facets, mlt, stats, and groups
solr_all(cli, params = list(q='ecology', rows=2, fl='id', facet="true",
 facet.field="journal", mlt='true', mlt.count=2, mlt.fl='abstract',
 stats='true', stats.field='counter_total_all', group='true',
 group.field='journal', group.limit=3))

# using wt = xml
solr_all(cli, params = list(q='*:*', rows=50, fl=c('id','score'),
  fq='doc_type:full', wt="xml"), raw=TRUE)

## End(Not run)
```

`solr_facet` *Faceted search*

Description

Returns only facet items

Usage

```
solr_facet(conn, name = NULL, params = list(q = "*:*"), body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", concat = ",", ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	Name of a collection or core. Or leave as NULL if not needed.
params	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
body	(list) a named list of parameters, if given a POST request will be performed
callopts	Call options passed on to [crul::HttpClient]
raw	(logical) If TRUE (default) raw json or xml returned. If FALSE, parsed data returned.
parsetype	(character) One of 'list' or 'df'
concat	(character) Character to concatenate elements of longer than length 1. Note that this only works reliably when data format is json (wt='json'). The parsing is more complicated in XML format, but you can do that on your own.
...	Further args, usually per field arguments for faceting.

Details

A number of fields can be specified multiple times, in which case you can separate them by commas, like `facet.field='journal,subject'`. Those fields are:

- facet.field
- facet.query
- facet.date
- facet.date.other
- facet.date.include
- facet.range
- facet.range.other
- facet.range.include
- facet.pivot

Options for some parameters:

facet.sort: This param determines the ordering of the facet field constraints.

- count sort the constraints by count (highest count first)
- index to return the constraints sorted in their index order (lexicographic by indexed term). For terms in the ascii range, this will be alphabetically sorted.

The default is count if facet.limit is greater than 0, index otherwise. This parameter can be specified on a per field basis.

facet.method: This parameter indicates what type of algorithm/method to use when faceting a field.

- enum Enumerates all terms in a field, calculating the set intersection of documents that match the term with documents that match the query. This was the default (and only) method for faceting multi-valued fields prior to Solr 1.4.

- fc (Field Cache) The facet counts are calculated by iterating over documents that match the query and summing the terms that appear in each document. This was the default method for single valued fields prior to Solr 1.4.
- fcs (Field Cache per Segment) works the same as fc except the underlying cache data structure is built for each segment of the index individually

The default value is fc (except for BoolField which uses enum) since it tends to use less memory and is faster then the enumeration method when a field has many unique terms in the index. For indexes that are changing rapidly in NRT situations, fcs may be a better choice because it reduces the overhead of building the cache structures on the first request and/or warming queries when opening a new searcher – but tends to be somewhat slower then fc for subsequent requests against the same searcher. This parameter can be specified on a per field basis.

facet.date.other: This param indicates that in addition to the counts for each date range constraint between facet.date.start and facet.date.end, counts should also be computed for...

- before All records with field values lower then lower bound of the first range
- after All records with field values greater then the upper bound of the last range
- between All records with field values between the start and end bounds of all ranges
- none Compute none of this information
- all Shortcut for before, between, and after

This parameter can be specified on a per field basis. In addition to the all option, this parameter can be specified multiple times to indicate multiple choices – but none will override all other options.

facet.date.include: By default, the ranges used to compute date faceting between facet.date.start and facet.date.end are all inclusive of both endpoints, while the "before" and "after" ranges are not inclusive. This behavior can be modified by the facet.date.include param, which can be any combination of the following options...

- lower All gap based ranges include their lower bound
- upper All gap based ranges include their upper bound
- edge The first and last gap ranges include their edge bounds (ie: lower for the first one, upper for the last one) even if the corresponding upper/lower option is not specified
- outer The "before" and "after" ranges will be inclusive of their bounds, even if the first or last ranges already include those boundaries.
- all Shorthand for lower, upper, edge, outer

This parameter can be specified on a per field basis. This parameter can be specified multiple times to indicate multiple choices.

facet.date.include: This param indicates that in addition to the counts for each range constraint between facet.range.start and facet.range.end, counts should also be computed for...

- before All records with field values lower then lower bound of the first range
- after All records with field values greater then the upper bound of the last range
- between All records with field values between the start and end bounds of all ranges
- none Compute none of this information
- all Shortcut for before, between, and after

This parameter can be specified on a per field basis. In addition to the all option, this parameter can be specified multiple times to indicate multiple choices – but none will override all other options.

facet.range.include: By default, the ranges used to compute range faceting between facet.range.start and facet.range.end are inclusive of their lower bounds and exclusive of the upper bounds. The "before" range is exclusive and the "after" range is inclusive. This default, equivalent to lower below, will not result in double counting at the boundaries. This behavior can be modified by the facet.range.include param, which can be any combination of the following options...

- lower All gap based ranges include their lower bound
- upper All gap based ranges include their upper bound
- edge The first and last gap ranges include their edge bounds (ie: lower for the first one, upper for the last one) even if the corresponding upper/lower option is not specified
- outer The "before" and "after" ranges will be inclusive of their bounds, even if the first or last ranges already include those boundaries.
- all Shorthand for lower, upper, edge, outer

Can be specified on a per field basis. Can be specified multiple times to indicate multiple choices. If you want to ensure you don't double-count, don't choose both lower & upper, don't choose outer, and don't choose all.

Value

Raw json or xml, or a list of length 4 parsed elements (usually data.frame's).

Facet parameters

- name Name of a collection or core. Or leave as `NULL` if not needed.
- q Query terms. See examples.
- facet.query This param allows you to specify an arbitrary query in the Lucene default syntax to generate a facet count. By default, faceting returns a count of the unique terms for a "field", while facet.query allows you to determine counts for arbitrary terms or expressions. This parameter can be specified multiple times to indicate that multiple queries should be used as separate facet constraints. It can be particularly useful for numeric range based facets, or prefix based facets – see example below (i.e. price:[* TO 500] and price:[501 TO *]).
- facet.field This param allows you to specify a field which should be treated as a facet. It will iterate over each Term in the field and generate a facet count using that Term as the constraint. This parameter can be specified multiple times to indicate multiple facet fields. None of the other params in this section will have any effect without specifying at least one field name using this param.
- facet.prefix Limits the terms on which to facet to those starting with the given string prefix. Note that unlike fq, this does not change the search results – it merely reduces the facet values returned to those beginning with the specified prefix. This parameter can be specified on a per field basis.
- facet.sort See Details.
- facet.limit This param indicates the maximum number of constraint counts that should be returned for the facet fields. A negative value means unlimited. Default: 100. Can be specified on a per field basis.

- facet.offset This param indicates an offset into the list of constraints to allow paging. Default: 0. This parameter can be specified on a per field basis.
- facet.mincount This param indicates the minimum counts for facet fields should be included in the response. Default: 0. This parameter can be specified on a per field basis.
- facet.missing Set to "true" this param indicates that in addition to the Term based constraints of a facet field, a count of all matching results which have no value for the field should be computed. Default: FALSE. This parameter can be specified on a per field basis.
- facet.method See Details.
- facet.enum.cache.minDf This param indicates the minimum document frequency (number of documents matching a term) for which the filterCache should be used when determining the constraint count for that term. This is only used when facet.method=enum method of faceting. A value greater than zero will decrease memory usage of the filterCache, but increase the query time. When faceting on a field with a very large number of terms, and you wish to decrease memory usage, try a low value of 25 to 50 first. Default: 0, causing the filterCache to be used for all terms in the field. This parameter can be specified on a per field basis.
- facet.threads This param will cause loading the underlying fields used in faceting to be executed in parallel with the number of threads specified. Specify as facet.threads=# where # is the maximum number of threads used. Omitting this parameter or specifying the thread count as 0 will not spawn any threads just as before. Specifying a negative number of threads will spin up to Integer.MAX_VALUE threads. Currently this is limited to the fields, range and query facets are not yet supported. In at least one case this has reduced warmup times from 20 seconds to under 5 seconds.
- facet.date Specify names of fields (of type DateField) which should be treated as date facets. Can be specified multiple times to indicate multiple date facet fields.
- facet.date.start The lower bound for the first date range for all Date Faceting on this field. This should be a single date expression which may use the DateMathParser syntax. Can be specified on a per field basis.
- facet.date.end The minimum upper bound for the last date range for all Date Faceting on this field (see facet.date.hardend for an explanation of what the actual end value may be greater). This should be a single date expression which may use the DateMathParser syntax. Can be specified on a per field basis.
- facet.date.gap The size of each date range expressed as an interval to be added to the lower bound using the DateMathParser syntax. Eg: facet.date.gap=+1DAY. Can be specified on a per field basis.
- facet.date.hardend A Boolean parameter instructing Solr what to do in the event that facet.date.gap does not divide evenly between facet.date.start and facet.date.end. If this is true, the last date range constraint will have an upper bound of facet.date.end; if false, the last date range will have the smallest possible upper bound greater then facet.date.end such that the range is exactly facet.date.gap wide. Default: FALSE. This parameter can be specified on a per field basis.
- facet.date.other See Details.
- facet.date.include See Details.
- facet.range Indicates what field to create range facets for. Example: facet.range=price&facet.range=age
- facet.range.start The lower bound of the ranges. Can be specified on a per field basis. Example: f.price.facet.range.start=0.0&f.age.facet.range.start=10

- facet.range.end The upper bound of the ranges. Can be specified on a per field basis. Example: f.price.facet.range.end=1000.0&f.age.facet.range.start=99
- facet.range.gap The size of each range expressed as a value to be added to the lower bound. For date fields, this should be expressed using the DateMathParser syntax. (ie: facet.range.gap=+1DAY). Can be specified on a per field basis. Example: f.price.facet.range.gap=100&f.age.facet.range.gap=10
- facet.range.hardend A Boolean parameter instructing Solr what to do in the event that facet.range.gap does not divide evenly between facet.range.start and facet.range.end. If this is true, the last range constraint will have an upper bound of facet.range.end; if false, the last range will have the smallest possible upper bound greater then facet.range.end such that the range is exactly facet.range.gap wide. Default: FALSE. This parameter can be specified on a per field basis.
- facet.range.other See Details.
- facet.range.include See Details.
- facet.pivot This param allows you to specify a single comma-separated string of fields to allow you to facet within the results of the parent facet to return counts in the format of SQL group by operation
- facet.pivot.mincount This param indicates the minimum counts for facet fields to be included in the response. Default: 0. This parameter should only be specified once.
- start Record to start at, default to beginning.
- rows Number of records to return.
- key API key, if needed.
- wt (character) Data type returned, defaults to 'json'. One of json or xml. If json, uses `fromJSON` to parse. If xml, uses `xmlParse` to parse. csv is only supported in `solr_search` and `solr_all`.

References

See http://wiki.apache.org/solr/SimpleFacetParameters for more information on faceting.

See Also

`solr_search()`, `solr_highlight()`, `solr_parse()`

Examples

```
## Not run:
# connect - local Solr instance
(cli <- SolrClient$new())
cli$facet("gettingstarted", params = list(q="*:*", facet.field='name'))
cli$facet("gettingstarted", params = list(q="*:*", facet.field='name'),
  callopts = list(verbose = TRUE))
cli$facet("gettingstarted", body = list(q="*:*", facet.field='name'),
  callopts = list(verbose = TRUE))

# Facet on a single field
solr_facet(cli, "gettingstarted", params = list(q='*:*', facet.field='name'))

# Remote instance
```

```
(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))

# Facet on multiple fields
solr_facet(cli, params = list(q='alcohol',
  facet.field = c('journal','subject')))

# Using mincount
solr_facet(cli, params = list(q='alcohol', facet.field='journal',
  facet.mincount='500'))

# Using facet.query to get counts
solr_facet(cli, params = list(q='*:*', facet.field='journal',
  facet.query=c('cell','bird')))

# Using facet.pivot to simulate SQL group by counts
solr_facet(cli, params = list(q='alcohol', facet.pivot='journal,subject',
             facet.pivot.mincount=10))
## two or more fields are required - you can pass in as a single
## character string
solr_facet(cli, params = list(q='*:*', facet.pivot = "journal,subject",
  facet.limit =  3))
## Or, pass in as a vector of length 2 or greater
solr_facet(cli, params = list(q='*:*', facet.pivot = c("journal", "subject"),
  facet.limit =  3))

# Date faceting
solr_facet(cli, params = list(q='*:*', facet.date='publication_date',
  facet.date.start='NOW/DAY-5DAYS', facet.date.end='NOW',
  facet.date.gap='+1DAY'))
## two variables
solr_facet(cli, params = list(q='*:*',
  facet.date=c('publication_date', 'timestamp'),
  facet.date.start='NOW/DAY-5DAYS', facet.date.end='NOW',
  facet.date.gap='+1DAY'))

# Range faceting
solr_facet(cli, params = list(q='*:*', facet.range='counter_total_all',
  facet.range.start=5, facet.range.end=1000, facet.range.gap=10))

# Range faceting with > 1 field, same settings
solr_facet(cli, params = list(q='*:*',
  facet.range=c('counter_total_all','alm_twitterCount'),
  facet.range.start=5, facet.range.end=1000, facet.range.gap=10))

# Range faceting with > 1 field, different settings
solr_facet(cli, params = list(q='*:*',
  facet.range=c('counter_total_all','alm_twitterCount'),
  f.counter_total_all.facet.range.start=5,
  f.counter_total_all.facet.range.end=1000,
  f.counter_total_all.facet.range.gap=10,
  f.alm_twitterCount.facet.range.start=5,
  f.alm_twitterCount.facet.range.end=1000,
  f.alm_twitterCount.facet.range.gap=10))
```

```
# Get raw json or xml
## json
solr_facet(cli, params = list(q='*:*', facet.field='journal'), raw=TRUE)
## xml
solr_facet(cli, params = list(q='*:*', facet.field='journal', wt='xml'),
  raw=TRUE)

# Get raw data back, and parse later, same as what goes on internally if
# raw=FALSE (Default)
out <- solr_facet(cli, params = list(q='*:*', facet.field='journal'),
  raw=TRUE)
solr_parse(out)
out <- solr_facet(cli, params = list(q='*:*', facet.field='journal',
  wt = 'xml'), raw=TRUE)
solr_parse(out)

# Using the USGS BISON API (https://bison.usgs.gov/#solr)
## The occurrence endpoint
(cli <- SolrClient$new(host = "bison.usgs.gov", scheme = "https",
  path = "solr/occurrences/select", port = NULL))
solr_facet(cli, params = list(q='*:*', facet.field='year'))
solr_facet(cli, params = list(q='*:*', facet.field='computedStateFips'))

# using a proxy
# cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL,
#   proxy = list(url = "http://54.195.48.153:8888"))
# solr_facet(cli, params = list(facet.field='journal'),
#   callopts=list(verbose=TRUE))

## End(Not run)
```

`solr_get` *Real time get*

Description

Get documents by id

Usage

```
solr_get(conn, ids, name, fl = NULL, wt = "json", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
ids	Document IDs, one or more in a vector or list
name	(character) A collection or core name. Required.

fl	Fields to return, can be a character vector like `c('id', 'title')`, or a single character vector with one or more comma separated names, like `'id,title'`
wt	(character) One of json (default) or xml. Data type returned. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses `xml2::read_xml()` to parse.
raw	(logical) If `TRUE`, returns raw data in format specified by `wt` param
...	curl options passed on to crul::HttpClient

Details

We use json internally as data interchange format for this function.

Examples

```
## Not run:
(cli <- SolrClient$new())

# add some documents first
ss <- list(list(id = 1, price = 100), list(id = 2, price = 500))
add(cli, ss, name = "gettingstarted")

# Now, get documents by id
solr_get(cli, ids = 1, "gettingstarted")
solr_get(cli, ids = 2, "gettingstarted")
solr_get(cli, ids = c(1, 2), "gettingstarted")
solr_get(cli, ids = "1,2", "gettingstarted")

# Get raw JSON
solr_get(cli, ids = 1, "gettingstarted", raw = TRUE, wt = "json")
solr_get(cli, ids = 1, "gettingstarted", raw = TRUE, wt = "xml")

## End(Not run)
```

---

solr_group *Grouped search*

Description

Returns only group items

Usage

```
solr_group(conn, name = NULL, params = NULL, body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", concat = ",", ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	Name of a collection or core. Or leave as NULL if not needed.
params	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
body	(list) a named list of parameters, if given a POST request will be performed
callopts	Call options passed on to [crul::HttpClient]
raw	(logical) If TRUE, returns raw data in format specified by wt param
parsetype	(character) One of 'list' or 'df'
concat	(character) Character to concatenate elements of longer than length 1. Note that this only works reliably when data format is json (wt='json'). The parsing is more complicated in XML format, but you can do that on your own.
...	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

Group parameters

- q Query terms, defaults to '*:*', or everything.
- fq Filter query, this does not affect the search, only what gets returned
- fl Fields to return
- wt (character) Data type returned, defaults to 'json'. One of json or xml. If json, uses `fromJSON` to parse. If xml, uses `xmlParse` to parse. csv is only supported in `solr_search` and `solr_all`.
- key API key, if needed.
- group.field [fieldname] Group based on the unique values of a field. The field must currently be single-valued and must be either indexed, or be another field type that has a value source and works in a function query - such as ExternalFileField. Note: for Solr 3.x versions the field must by a string like field such as StrField or TextField, otherwise a http status 400 is returned.
- group.func [function query] Group based on the unique values of a function query. <!> Solr4.0 This parameter only is supported on 4.0
- group.query [query] Return a single group of documents that also match the given query.
- rows [number] The number of groups to return. Defaults to 10.
- start [number] The offset into the list of groups.
- group.limit [number] The number of results (documents) to return for each group. Defaults to 1.
- group.offset [number] The offset into the document list of each group.
- sort How to sort the groups relative to each other. For example, sort=popularity desc will cause the groups to be sorted according to the highest popularity doc in each group. Defaults to "score desc".

- group.sort How to sort documents within a single group. Defaults to the same value as the sort parameter.
- group.format One of grouped or simple. If simple, the grouped documents are presented in a single flat list. The start and rows parameters refer to numbers of documents instead of numbers of groups.
- group.main (logical) If true, the result of the last field grouping command is used as the main result list in the response, using group.format=simple
- group.ngroups (logical) If true, includes the number of groups that have matched the query. Default is false. <!> Solr4.1 WARNING: If this parameter is set to true on a sharded environment, all the documents that belong to the same group have to be located in the same shard, otherwise the count will be incorrect. If you are using SolrCloud, consider using "custom hashing"
- group.cache.percent [0-100] If > 0 enables grouping cache. Grouping is executed actual two searches. This option caches the second search. A value of 0 disables grouping caching. Default is 0. Tests have shown that this cache only improves search time with boolean queries, wildcard queries and fuzzy queries. For simple queries like a term query or a match all query this cache has a negative impact on performance

References

See http://wiki.apache.org/solr/FieldCollapsing for more information.

See Also

solr_highlight(), solr_facet()

Examples

```
## Not run:
# connect
(cli <- SolrClient$new())

# by default we do a GET request
cli$group("gettingstarted",
  params = list(q='*:*', group.field='compName_s'))
# OR
solr_group(cli, "gettingstarted",
  params = list(q='*:*', group.field='compName_s'))

# connect
(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))

# Basic group query
solr_group(cli, params = list(q='ecology', group.field='journal',
  group.limit=3, fl=c('id','score')))
solr_group(cli, params = list(q='ecology', group.field='journal',
  group.limit=3, fl='article_type'))

# Different ways to sort (notice diff btw sort of group.sort)
# note that you can only sort on a field if you return that field
```

```
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
   fl=c('id','score')))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
   fl=c('id','score','alm_twitterCount'), group.sort='alm_twitterCount desc'))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
   fl=c('id','score','alm_twitterCount'), sort='score asc',
   group.sort='alm_twitterCount desc'))

# Two group.field values
out <- solr_group(cli, params = list(q='ecology', group.field=c('journal','article_type'),
  group.limit=3, fl='id'), raw=TRUE)
solr_parse(out)
solr_parse(out, 'df')

# Get two groups, one with alm_twitterCount of 0-10, and another group
# with 10 to infinity
solr_group(cli, params = list(q='ecology', group.limit=3, fl=c('id','alm_twitterCount'),
 group.query=c('alm_twitterCount:[0 TO 10]','alm_twitterCount:[10 TO *]')))

# Use of group.format and group.simple.
## The raw data structure of these two calls are slightly different, but
## the parsing inside the function outputs the same results. You can
## of course set raw=TRUE to get back what the data actually look like
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), group.format='simple'))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), group.format='grouped'))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), group.format='grouped', group.main='true'))

# xml back
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), wt = "xml"))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), wt = "xml"), parsetype = "list")
res <- solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl=c('id','score'), wt = "xml"), raw = TRUE)
library("xml2")
xml2::read_xml(unclass(res))

solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl='article_type', wt = "xml"))
solr_group(cli, params = list(q='ecology', group.field='journal', group.limit=3,
  fl='article_type', wt = "xml"), parsetype = "list")

## End(Not run)
```

---

solr_highlight *Highlighting search*

Description

Returns only highlight items

Usage

```
solr_highlight(conn, name = NULL, params = NULL, body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	Name of a collection or core. Or leave as NULL if not needed.
params	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
body	(list) a named list of parameters, if given a POST request will be performed
callopts	Call options passed on to [crul::HttpClient]
raw	(logical) If TRUE (default) raw json or xml returned. If FALSE, parsed data returned.
parsetype	One of list of df (data.frame)
...	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

Facet parameters

- q Query terms. See examples.
- hl.fl A comma-separated list of fields for which to generate highlighted snippets. If left blank, the fields highlighted for the LuceneQParser are the defaultSearchField (or the df param if used) and for the DisMax parser the qf fields are used. A '*' can be used to match field globs, e.g. 'text_*' or even '*' to highlight on all fields where highlighting is possible. When using '*', consider adding hl.requireFieldMatch=TRUE.
- hl.snippets Max no. of highlighted snippets to generate per field. Note: it is possible for any number of snippets from zero to this value to be generated. This parameter accepts per-field overrides. Default: 1.
- hl.fragsize The size, in characters, of the snippets (aka fragments) created by the highlighter. In the original Highlighter, "0" indicates that the whole field value should be used with no fragmenting. See http://wiki.apache.org/solr/HighlightingParameters for more info.
- hl.q Set a query request to be highlighted. It overrides q parameter for highlighting. Solr query syntax is acceptable for this parameter.
- hl.mergeContiguous Collapse contiguous fragments into a single fragment. "true" indicates contiguous fragments will be collapsed into single fragment. This parameter accepts per-field overrides. This parameter makes sense for the original Highlighter only. Default: FALSE.

- hl.requireFieldMatch If TRUE, then a field will only be highlighted if the query matched in this particular field (normally, terms are highlighted in all requested fields regardless of which field matched the query). This only takes effect if "hl.usePhraseHighlighter" is TRUE. Default: FALSE.
- hl.maxAnalyzedChars How many characters into a document to look for suitable snippets. This parameter makes sense for the original Highlighter only. Default: 51200. You can assign a large value to this parameter and use hl.fragsize=0 to return highlighting in large fields that have size greater than 51200 characters.
- hl.alternateField If a snippet cannot be generated (due to no terms matching), you can specify a field to use as the fallback. This parameter accepts per-field overrides.
- hl.maxAlternateFieldLength If hl.alternateField is specified, this parameter specifies the maximum number of characters of the field to return. Any value less than or equal to 0 means unlimited. Default: unlimited.
- hl.preserveMulti Preserve order of values in a multiValued list. Default: FALSE.
- hl.maxMultiValuedToExamine When highlighting a multiValued field, stop examining the individual entries after looking at this many of them. Will potentially return 0 snippets if this limit is reached before any snippets are found. If maxMultiValuedToMatch is also specified, whichever limit is hit first will terminate looking for more. Default: Integer.MAX_VALUE
- hl.maxMultiValuedToMatch When highlighting a multiValued field, stop examining the individual entries after looking at this many matches are found. If maxMultiValuedToExamine is also specified, whichever limit is hit first will terminate looking for more. Default: Integer.MAX_VALUE
- hl.formatter Specify a formatter for the highlight output. Currently the only legal value is "simple", which surrounds a highlighted term with a customizable pre- and post text snippet. This parameter accepts per-field overrides. This parameter makes sense for the original Highlighter only.
- hl.simple.pre The text which appears before and after a highlighted term when using the simple formatter. This parameter accepts per-field overrides. The default values are "" and "" This parameter makes sense for the original Highlighter only. Use hl.tag.pre and hl.tag.post for FastVectorHighlighter (see example under hl.fragmentsBuilder)
- hl.simple.post The text which appears before and after a highlighted term when using the simple formatter. This parameter accepts per-field overrides. The default values are "" and "" This parameter makes sense for the original Highlighter only. Use hl.tag.pre and hl.tag.post for FastVectorHighlighter (see example under hl.fragmentsBuilder)
- hl.fragmenter Specify a text snippet generator for highlighted text. The standard fragmenter is gap (which is so called because it creates fixed-sized fragments with gaps for multi-valued fields). Another option is regex, which tries to create fragments that "look like" a certain regular expression. This parameter accepts per-field overrides. Default: "gap"
- hl.fragListBuilder Specify the name of SolrFragListBuilder. This parameter makes sense for FastVectorHighlighter only. To create a fragSize=0 with the FastVectorHighlighter, use the SingleFragListBuilder. This field supports per-field overrides.
- hl.fragmentsBuilder Specify the name of SolrFragmentsBuilder. This parameter makes sense for FastVectorHighlighter only.
- hl.boundaryScanner Configures how the boundaries of fragments are determined. By default, boundaries will split at the character level, creating a fragment such as "uick brown fox jumps

over the la". Valid entries are breakIterator or simple, with breakIterator being the most commonly used. This parameter makes sense for FastVectorHighlighter only.

- hl.bs.maxScan Specify the length of characters to be scanned by SimpleBoundaryScanner. Default: 10. This parameter makes sense for FastVectorHighlighter only.
- hl.bs.chars Specify the boundary characters, used by SimpleBoundaryScanner. This parameter makes sense for FastVectorHighlighter only.
- hl.bs.type Specify one of CHARACTER, WORD, SENTENCE and LINE, used by BreakIteratorBoundaryScanner. Default: WORD. This parameter makes sense for FastVectorHighlighter only.
- hl.bs.language Specify the language for Locale that is used by BreakIteratorBoundaryScanner. This parameter makes sense for FastVectorHighlighter only. Valid entries take the form of ISO 639-1 strings.
- hl.bs.country Specify the country for Locale that is used by BreakIteratorBoundaryScanner. This parameter makes sense for FastVectorHighlighter only. Valid entries take the form of ISO 3166-1 alpha-2 strings.
- hl.useFastVectorHighlighter Use FastVectorHighlighter. FastVectorHighlighter requires the field is termVectors=on, termPositions=on and termOffsets=on. This parameter accepts per-field overrides. Default: FALSE
- hl.usePhraseHighlighter Use SpanScorer to highlight phrase terms only when they appear within the query phrase in the document. Default: TRUE.
- hl.highlightMultiTerm If the SpanScorer is also being used, enables highlighting for range/wildcard/fuzzy/prefix queries. Default: FALSE. This parameter makes sense for the original Highlighter only.
- hl.regex.slop Factor by which the regex fragmenter can stray from the ideal fragment size (given by hl.fragsize) to accomodate the regular expression. For instance, a slop of 0.2 with fragsize of 100 should yield fragments between 80 and 120 characters in length. It is usually good to provide a slightly smaller fragsize when using the regex fragmenter. Default: .6. This parameter makes sense for the original Highlighter only.
- hl.regex.pattern The regular expression for fragmenting. This could be used to extract sentences (see example solrconfig.xml) This parameter makes sense for the original Highlighter only.
- hl.regex.maxAnalyzedChars Only analyze this many characters from a field when using the regex fragmenter (after which, the fragmenter produces fixed-sized fragments). Applying a complicated regex to a huge field is expensive. Default: 10000. This parameter makes sense for the original Highlighter only.
- start Record to start at, default to beginning.
- rows Number of records to return.
- wt (character) Data type returned, defaults to 'json'. One of json or xml. If json, uses `fromJSON` to parse. If xml, uses `xmlParse` to parse. csv is only supported in `solr_search` and `solr_all`.
- fl Fields to return
- fq Filter query, this does not affect the search, only what gets returned

References

See http://wiki.apache.org/solr/HighlightingParameters for more information on highlighting.

See Also

solr_search(), solr_facet()

Examples

```
## Not run:
# connect
(conn <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))

# highlight search
solr_highlight(conn, params = list(q='alcohol', hl.fl = 'abstract', rows=10),
  parsetype = "list")
solr_highlight(conn, params = list(q='alcohol', hl.fl = c('abstract','title'),
  rows=3), parsetype = "list")

# Raw data back
## json
solr_highlight(conn, params = list(q='alcohol', hl.fl = 'abstract', rows=10),
   raw=TRUE)
## xml
solr_highlight(conn, params = list(q='alcohol', hl.fl = 'abstract', rows=10,
   wt='xml'), raw=TRUE)
## parse after getting data back
out <- solr_highlight(conn, params = list(q='theoretical math',
   hl.fl = c('abstract','title'), hl.fragsize=30, rows=10, wt='xml'),
   raw=TRUE)
solr_parse(out, parsetype='list')

## End(Not run)
```

---

solr_mlt *"more like this" search*

Description

Returns only more like this items

Usage

```
solr_mlt(conn, name = NULL, params = NULL, body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", concat = ",",
  optimizeMaxRows = TRUE, minOptimizedRows = 50000L, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
name	Name of a collection or core. Or leave as NULL if not needed.
params	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
body	(list) a named list of parameters, if given a POST request will be performed
callopts	Call options passed on to [crul::HttpClient]
raw	(logical) If TRUE, returns raw data in format specified by wt param
parsetype	(character) One of 'list' or 'df'
concat	(character) Character to concatenate elements of longer than length 1. Note that this only works reliably when data format is json (wt='json'). The parsing is more complicated in XML format, but you can do that on your own.
optimizeMaxRows	(logical) If TRUE, then rows parameter will be adjusted to the number of returned results by the same constraints. It will only be applied if rows parameter is higher than minOptimizedRows. Default: TRUE
minOptimizedRows	(numeric) used by optimizedMaxRows parameter, the minimum optimized rows. Default: 50000
...	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

More like this parameters

- q Query terms, defaults to '*:*', or everything.
- fq Filter query, this does not affect the search, only what gets returned
- mlt.count The number of similar documents to return for each result. Default is 5.
- mlt.fl The fields to use for similarity. NOTE: if possible these should have a stored TermVector DEFAULT_FIELD_NAMES = new String[] "contents"
- mlt.mintf Minimum Term Frequency - the frequency below which terms will be ignored in the source doc. DEFAULT_MIN_TERM_FREQ = 2
- mlt.mindf Minimum Document Frequency - the frequency at which words will be ignored which do not occur in at least this many docs. DEFAULT_MIN_DOC_FREQ = 5
- mlt.minwl minimum word length below which words will be ignored. DEFAULT_MIN_WORD_LENGTH = 0
- mlt.maxwl maximum word length above which words will be ignored. DEFAULT_MAX_WORD_LENGTH = 0
- mlt.maxqt maximum number of query terms that will be included in any generated query. DEFAULT_MAX_QUERY_TERMS = 25

- mlt.maxntp maximum number of tokens to parse in each example doc field that is not stored with TermVector support. DEFAULT_MAX_NUM_TOKENS_PARSED = 5000
- mlt.boost [true/false] set if the query will be boosted by the interesting term relevance. DEFAULT_BOOST = false
- mlt.qf Query fields and their boosts using the same format as that used in DisMaxQParserPlugin. These fields must also be specified in mlt.fl.
- fl Fields to return. We force 'id' to be returned so that there is a unique identifier with each record.
- wt (character) Data type returned, defaults to 'json'. One of json or xml. If json, uses `fromJSON` to parse. If xml, uses `xmlParse` to parse. csv is only supported in `solr_search` and `solr_all`.
- start Record to start at, default to beginning.
- rows Number of records to return. Defaults to 10.
- key API key, if needed.

References

See http://wiki.apache.org/solr/MoreLikeThis for more information.

Examples

```
## Not run:
# connect
(conn <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))

# more like this search
conn$mlt(params = list(q='*:*', mlt.count=2, mlt.fl='abstract', fl='score',
  fq="doc_type:full"))
conn$mlt(params = list(q='*:*', rows=2, mlt.fl='title', mlt.mindf=1,
  mlt.mintf=1, fl='alm_twitterCount'))
conn$mlt(params = list(q='title:"ecology" AND body:"cell"', mlt.fl='title',
  mlt.mindf=1, mlt.mintf=1, fl='counter_total_all', rows=5))
conn$mlt(params = list(q='ecology', mlt.fl='abstract', fl='title', rows=5))
solr_mlt(conn, params = list(q='ecology', mlt.fl='abstract',
  fl=c('score','eissn'), rows=5))
solr_mlt(conn, params = list(q='ecology', mlt.fl='abstract',
  fl=c('score','eissn'), rows=5, wt = "xml"))

# get raw data, and parse later if needed
out <- solr_mlt(conn, params=list(q='ecology', mlt.fl='abstract', fl='title',
 rows=2), raw=TRUE)
solr_parse(out, "df")

## End(Not run)
```

`solr_optimize` *Optimize*

Description

Optimize

Usage

```
solr_optimize(conn, name, max_segments = 1, wait_searcher = TRUE,
  soft_commit = FALSE, wt = "json", raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	(character) A collection or core name. Required.
`max_segments`	optimizes down to at most this number of segments. Default: 1
`wait_searcher`	block until a new searcher is opened and registered as the main query searcher, making the changes visible. Default: `TRUE`
`soft_commit`	perform a soft commit - this will refresh the 'view' of the index in a more performant manner, but without "on-disk" guarantees. Default: `FALSE`
`wt`	(character) One of json (default) or xml. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses `xml2::read_xml()` to parse
`raw`	(logical) If `TRUE`, returns raw data in format specified by `wt` param
`...`	curl options passed on to crul::HttpClient

Examples

```
## Not run:
(conn <- SolrClient$new())

solr_optimize(conn, "gettingstarted")
solr_optimize(conn, "gettingstarted", max_segments = 2)
solr_optimize(conn, "gettingstarted", wait_searcher = FALSE)

# get xml back
solr_optimize(conn, "gettingstarted", wt = "xml")
## raw xml
solr_optimize(conn, "gettingstarted", wt = "xml", raw = TRUE)

## End(Not run)
```

solr_parse *Parse raw data from solr_search, solr_facet, or solr_highlight.*

Description

Parse raw data from solr_search, solr_facet, or solr_highlight.

Usage

```
solr_parse(input, parsetype = NULL, concat)

## S3 method for class 'sr_high'
solr_parse(input, parsetype = "list", concat = ",")

## S3 method for class 'sr_search'
solr_parse(input, parsetype = "list", concat = ",")

## S3 method for class 'sr_all'
solr_parse(input, parsetype = "list", concat = ",")

## S3 method for class 'sr_mlt'
solr_parse(input, parsetype = "list", concat = ",")

## S3 method for class 'sr_stats'
solr_parse(input, parsetype = "list", concat = ",")

## S3 method for class 'sr_group'
solr_parse(input, parsetype = "list", concat = ",")
```

Arguments

input	Output from solr_facet
parsetype	One of 'list' or 'df' (data.frame)
concat	Character to conactenate strings by, e.g,. ',' (character). Used in solr_parse.sr_search only.

Details

This is the parser used internally in solr_facet, but if you output raw data from solr_facet using raw=TRUE, then you can use this function to parse that data (a sr_facet S3 object) after the fact to a list of data.frame's for easier consumption. The data format type is detected from the attribute "wt" on the sr_facet object.

`solr_search` *Solr search*

Description

Returns only matched documents, and doesn't return other items, including facets, groups, mlt, stats, and highlights.

Usage

```
solr_search(conn, name = NULL, params = list(q = "*:*"), body = NULL,
  callopts = list(), raw = FALSE, parsetype = "df", concat = ",",
  optimizeMaxRows = TRUE, minOptimizedRows = 50000L, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	Name of a collection or core. Or leave as `NULL` if not needed.
`params`	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
`body`	(list) a named list of parameters, if given a POST request will be performed
`callopts`	Call options passed on to [crul::HttpClient]
`raw`	(logical) If TRUE, returns raw data in format specified by wt param
`parsetype`	(character) One of 'list' or 'df'
`concat`	(character) Character to concatenate elements of longer than length 1. Note that this only works reliably when data format is json (wt='json'). The parsing is more complicated in XML format, but you can do that on your own.
`optimizeMaxRows`	(logical) If `TRUE`, then rows parameter will be adjusted to the number of returned results by the same constraints. It will only be applied if rows parameter is higher than `minOptimizedRows`. Default: `TRUE`
`minOptimizedRows`	(numeric) used by `optimizedMaxRows` parameter, the minimum optimized rows. Default: 50000
`...`	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

Parameters

- q Query terms, defaults to '*:*', or everything.
- sort Field to sort on. You can specify ascending (e.g., score desc) or descending (e.g., score asc), sort by two fields (e.g., score desc, price asc), or sort by a function (e.g., sum(x_f, y_f) desc, which sorts by the sum of x_f and y_f in a descending order).
- start Record to start at, default to beginning.
- rows Number of records to return. Default: 10.
- pageDoc If you expect to be paging deeply into the results (say beyond page 10, assuming rows=10) and you are sorting by score, you may wish to add the pageDoc and pageScore parameters to your request. These two parameters tell Solr (and Lucene) what the last result (Lucene internal docid and score) of the previous page was, so that when scoring the query for the next set of pages, it can ignore any results that occur higher than that item. To get the Lucene internal doc id, you will need to add [docid] to the &fl list. e.g., q=*:*&start=10&pageDoc=5&pageScore=1.345
- pageScore See pageDoc notes.
- fq Filter query, this does not affect the search, only what gets returned. This parameter can accept multiple items in a lis or vector. You can't pass more than one parameter of the same name, so we get around it by passing multiple queries and we parse internally
- fl Fields to return, can be a character vector like `c('id', 'title')`, or a single character vector with one or more comma separated names, like `'id,title'`
- defType Specify the query parser to use with this request.
- timeAllowed The time allowed for a search to finish. This value only applies to the search and not to requests in general. Time is in milliseconds. Values <= 0 mean no time restriction. Partial results may be returned (if there are any).
- qt Which query handler used. Options: dismax, others?
- NOW Set a fixed time for evaluating Date based expresions
- TZ Time zone, you can override the default.
- echoHandler If `TRUE`, Solr places the name of the handle used in the response to the client for debugging purposes. Default:
- echoParams The echoParams parameter tells Solr what kinds of Request parameters should be included in the response for debugging purposes, legal values include:
  - none - don't include any request parameters for debugging
  - explicit - include the parameters explicitly specified by the client in the request
  - all - include all parameters involved in this request, either specified explicitly by the client, or implicit because of the request handler configuration.
- wt (character) One of json, xml, or csv. Data type returned, defaults to 'csv'. If json, uses [jsonlite::fromJSON()] to parse. If xml, uses [xml2::read_xml()] to parse. If csv, uses [read.table()] to parse. 'wt=csv' gives the fastest performance at least in all the cases we have tested in, thus it's the default value for 'wt'

Note

SOLR v1.2 was first version to support csv. See https://issues.apache.org/jira/browse/SOLR-66

References

See http://wiki.apache.org/solr/#Search_and_Indexing for more information.

See Also

solr_highlight(), solr_facet()

Examples

```
## Not run:
# Connect to a local Solr instance
(cli <- SolrClient$new())
cli$search("gettingstarted", params = list(q = "features:notes"))

solr_search(cli, "gettingstarted")
solr_search(cli, "gettingstarted", params = list(q = "features:notes"))
solr_search(cli, "gettingstarted", body = list(query = "features:notes"))

(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))
cli$search(params = list(q = "*:*"))
cli$search(params = list(q = "title:golgi", fl = c('id', 'title')))

cli$search(params = list(q = "*:*", facet = "true"))


# search
solr_search(cli, params = list(q='*:*', rows=2, fl='id'))

# search and return all rows
solr_search(cli, params = list(q='*:*', rows=-1, fl='id'))

# Search for word ecology in title and cell in the body
solr_search(cli, params = list(q='title:"ecology" AND body:"cell"',
  fl='title', rows=5))

# Search for word "cell" and not "body" in the title field
solr_search(cli, params = list(q='title:"cell" -title:"lines"', fl='title',
  rows=5))

# Wildcards
## Search for word that starts with "cell" in the title field
solr_search(cli, params = list(q='title:"cell*"', fl='title', rows=5))

# Proximity searching
## Search for words "sports" and "alcohol" within four words of each other
solr_search(cli, params = list(q='everything:"sports alcohol"~7',
  fl='abstract', rows=3))

# Range searches
## Search for articles with Twitter count between 5 and 10
solr_search(cli, params = list(q='*:*', fl=c('alm_twitterCount','id'),
  fq='alm_twitterCount:[5 TO 50]', rows=10))
```

```
# Boosts
## Assign higher boost to title matches than to body matches
## (compare the two calls)
solr_search(cli, params = list(q='title:"cell" abstract:"science"',
  fl='title', rows=3))
solr_search(cli, params = list(q='title:"cell"^1.5 AND abstract:"science"',
  fl='title', rows=3))

# FunctionQuery queries
## This kind of query allows you to use the actual values of fields to
## calculate relevancy scores for returned documents

## Here, we search on the product of counter_total_all and alm_twitterCount
## metrics for articles in PLOS Journals
solr_search(cli, params = list(q="{!func}product($v1,$v2)",
  v1 = 'sqrt(counter_total_all)',
  v2 = 'log(alm_twitterCount)', rows=5, fl=c('id','title'),
  fq='doc_type:full'))

## here, search on the product of counter_total_all and alm_twitterCount,
## using a new temporary field "_val_"
solr_search(cli,
  params = list(q='_val_:"product(counter_total_all,alm_twitterCount)"',
  rows=5, fl=c('id','title'), fq='doc_type:full'))

## papers with most citations
solr_search(cli, params = list(q='_val_:"max(counter_total_all)"',
   rows=5, fl=c('id','counter_total_all'), fq='doc_type:full'))

## papers with most tweets
solr_search(cli, params = list(q='_val_:"max(alm_twitterCount)"',
   rows=5, fl=c('id','alm_twitterCount'), fq='doc_type:full'))

## many fq values
solr_search(cli, params = list(q="*:*", fl=c('id','alm_twitterCount'),
   fq=list('doc_type:full','subject:"Social networks"',
           'alm_twitterCount:[100 TO 10000]'),
   sort='counter_total_month desc'))

## using wt = csv
solr_search(cli, params = list(q='*:*', rows=50, fl=c('id','score'),
  fq='doc_type:full', wt="csv"))
solr_search(cli, params = list(q='*:*', rows=50, fl=c('id','score'),
  fq='doc_type:full'))

# using a proxy
# cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL,
#   proxy = list(url = "http://186.249.1.146:80"))
# solr_search(cli, q='*:*', rows=2, fl='id', callopts=list(verbose=TRUE))

# Pass on curl options to modify request
## verbose
```

```
solr_search(cli, params = list(q='*:*', rows=2, fl='id'),
  callopts = list(verbose=TRUE))

## End(Not run)
```

`solr_stats` *Solr stats*

Description

Returns only stat items

Usage

```
solr_stats(conn, name = NULL, params = list(q = "*:*", stats.field = NULL,
  stats.facet = NULL), body = NULL, callopts = list(), raw = FALSE,
  parsetype = "df", ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`name`	Name of a collection or core. Or leave as `NULL` if not needed.
`params`	(list) a named list of parameters, results in a GET reqeust as long as no body parameters given
`body`	(list) a named list of parameters, if given a POST request will be performed
`callopts`	Call options passed on to [crul::HttpClient]
`raw`	(logical) If TRUE, returns raw data in format specified by wt param
`parsetype`	(character) One of 'list' or 'df'
`...`	Further args to be combined into query

Value

XML, JSON, a list, or data.frame

Stats parameters

- q Query terms, defaults to '*:*', or everything.
- stats.field The number of similar documents to return for each result.
- stats.facet You can not facet on multi-valued fields.
- wt (character) Data type returned, defaults to 'json'. One of json or xml. If json, uses `fromJSON` to parse. If xml, uses `xmlParse` to parse. csv is only supported in `solr_search` and `solr_all`.
- start Record to start at, default to beginning.
- rows Number of records to return. Defaults to 10.
- key API key, if needed.

References

See http://wiki.apache.org/solr/StatsComponent for more information on Solr stats.

See Also

solr_highlight(), solr_facet(), solr_search(), solr_mlt()

Examples

```
## Not run:
# connect
(cli <- SolrClient$new(host = "api.plos.org", path = "search", port = NULL))

# get stats
solr_stats(cli, params = list(q='science', stats.field='counter_total_all'),
  raw=TRUE)
solr_stats(cli, params = list(q='title:"ecology" AND body:"cell"',
   stats.field=c('counter_total_all','alm_twitterCount')))
solr_stats(cli, params = list(q='ecology',
  stats.field=c('counter_total_all','alm_twitterCount'),
  stats.facet='journal'))
solr_stats(cli, params = list(q='ecology',
  stats.field=c('counter_total_all','alm_twitterCount'),
  stats.facet=c('journal','volume')))

# Get raw data, then parse later if you feel like it
## json
out <- solr_stats(cli, params = list(q='ecology',
  stats.field=c('counter_total_all','alm_twitterCount'),
  stats.facet=c('journal','volume')), raw=TRUE)
library("jsonlite")
jsonlite::fromJSON(out)
solr_parse(out) # list
solr_parse(out, 'df') # data.frame

## xml
out <- solr_stats(cli, params = list(q='ecology',
  stats.field=c('counter_total_all','alm_twitterCount'),
  stats.facet=c('journal','volume'), wt="xml"), raw=TRUE)
library("xml2")
xml2::read_xml(unclass(out))
solr_parse(out) # list
solr_parse(out, 'df') # data.frame

# Get verbose http call information
solr_stats(cli, params = list(q='ecology', stats.field='alm_twitterCount'),
   callopts=list(verbose=TRUE))

## End(Not run)
```

update_atomic_json *Atomic updates with JSON data*

Description

Atomic updates to parts of Solr documents

Usage

```
update_atomic_json(conn, body, name, wt = "json", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
body	(character) JSON as a character string
name	(character) Name of the core or collection
wt	(character) One of json (default) or xml. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses `xml2::read_xml()` to parse
raw	(logical) If `TRUE`, returns raw data in format specified by `wt` param
...	curl options passed on to crul::HttpClient

References

https://lucene.apache.org/solr/guide/7_0/updating-parts-of-documents.html

Examples

```
## Not run:
# start Solr in Cloud mode: bin/solr start -e cloud -noprompt

# connect
(conn <- SolrClient$new())

# create a collection
if (!conn$collection_exists("books")) {
  conn$collection_delete("books")
  conn$collection_create("books")
}

# Add documents
file <- system.file("examples", "books2.json", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_json(file, "books")

# get a document
conn$get(ids = 343334534545, "books")
```

```
# atomic update
body <- '[{
 "id": "343334534545",
 "genre_s": {"set": "mystery" },
 "pages_i": {"inc": 1 }
}]'
conn$update_atomic_json(body, "books")

# get the document again
conn$get(ids = 343334534545, "books")

# another atomic update
body <- '[{
 "id": "343334534545",
 "price": {"remove": "12.5" }
}]'
conn$update_atomic_json(body, "books")

# get the document again
conn$get(ids = 343334534545, "books")

## End(Not run)
```

`update_atomic_xml` *Atomic updates with XML data*

Description

Atomic updates to parts of Solr documents

Usage

```
update_atomic_xml(conn, body, name, wt = "json", raw = FALSE, ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`body`	(character) XML as a character string
`name`	(character) Name of the core or collection
`wt`	(character) One of json (default) or xml. If json, uses `jsonlite::fromJSON()` to parse. If xml, uses `xml2::read_xml()` to parse
`raw`	(logical) If `TRUE`, returns raw data in format specified by `wt` param
`...`	curl options passed on to crul::HttpClient

References

https://lucene.apache.org/solr/guide/7_0/updating-parts-of-documents.html

Examples

```
## Not run:
# start Solr in Cloud mode: bin/solr start -e cloud -noprompt

# connect
(conn <- SolrClient$new())

# create a collection
if (!conn$collection_exists("books")) {
  conn$collection_delete("books")
  conn$collection_create("books")
}

# Add documents
file <- system.file("examples", "books.xml", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_xml(file, "books")

# get a document
conn$get(ids = '978-0641723445', "books", wt = "xml")

# atomic update
body <- '
<add>
 <doc>
   <field name="id">978-0641723445</field>
   <field name="genre_s" update="set">mystery</field>
   <field name="pages_i" update="inc">1</field>
 </doc>
</add>'
conn$update_atomic_xml(body, name="books")

# get the document again
conn$get(ids = '978-0641723445', "books", wt = "xml")

# another atomic update
body <- '
<add>
 <doc>
   <field name="id">978-0641723445</field>
   <field name="price" update="remove">12.5</field>
 </doc>
</add>'
conn$update_atomic_xml(body, "books")

# get the document again
conn$get(ids = '978-0641723445', "books", wt = "xml")

## End(Not run)
```

update_csv *Update documents with CSV data*

Description

Update documents with CSV data

Usage

```
update_csv(conn, files, name, separator = ",", header = TRUE,
  fieldnames = NULL, skip = NULL, skipLines = 0, trim = FALSE,
  encapsulator = NULL, escape = NULL, keepEmpty = FALSE, literal = NULL,
  map = NULL, split = NULL, rowid = NULL, rowidOffset = NULL,
  overwrite = NULL, commit = NULL, wt = "json", raw = FALSE, ...)
```

Arguments

conn	A solrium connection object, see SolrClient
files	Path to a single file to load into Solr
name	(character) Name of the core or collection
separator	Specifies the character to act as the field separator. Default: ','
header	TRUE if the first line of the CSV input contains field or column names. Default: TRUE. If the fieldnames parameter is absent, these field names will be used when adding documents to the index.
fieldnames	Specifies a comma separated list of field names to use when adding documents to the Solr index. If the CSV input already has a header, the names specified by this parameter will override them. Example: fieldnames=id,name,category
skip	A comma separated list of field names to skip in the input. An alternate way to skip a field is to specify it's name as a zero length string in fieldnames. For example, fieldnames=id,name,category&skip=name skips the name field, and is equivalent to fieldnames=id,,category
skipLines	Specifies the number of lines in the input stream to discard before the CSV data starts (including the header, if present). Default: 0
trim	If true remove leading and trailing whitespace from values. CSV parsing already ignores leading whitespace by default, but there may be trailing whitespace, or there may be leading whitespace that is encapsulated by quotes and is thus not removed. This may be specified globally, or on a per-field basis. Default: FALSE
encapsulator	The character optionally used to surround values to preserve characters such as the CSV separator or whitespace. This standard CSV format handles the encapsulator itself appearing in an encapsulated value by doubling the encapsulator.
escape	The character used for escaping CSV separators or other reserved characters. If an escape is specified, the encapsulator is not used unless also explicitly specified since most formats use either encapsulation or escaping, not both.

keepEmpty	Keep and index empty (zero length) field values. This may be specified globally, or on a per-field basis. Default: FALSE
literal	Adds fixed field name/value to all documents. Example: Adds a "datasource" field with value equal to "products" for every document indexed from the CSV literal.datasource=products
map	Specifies a mapping between one value and another. The string on the LHS of the colon will be replaced with the string on the RHS. This parameter can be specified globally or on a per-field basis. Example: replaces "Absolutely" with "true" in every field map=Absolutely:true. Example: removes any values of "RemoveMe" in the field "foo" f.foo.map=RemoveMe:&f.foo.keepEmpty=false
split	If TRUE, the field value is split into multiple values by another CSV parser. The CSV parsing rules such as separator and encapsulator may be specified as field parameters. See https://wiki.apache.org/solr/UpdateCSV#split for examples.
rowid	If not null, add a new field to the document where the passed in parameter name is the field name to be added and the current line/rowid is the value. This is useful if your CSV doesn't have a unique id already in it and you want to use the line number as one. Also useful if you simply want to index where exactly in the original CSV file the row came from
rowidOffset	In conjunction with the rowid parameter, this integer value will be added to the rowid before adding it the field.
overwrite	If true (the default), check for and overwrite duplicate documents, based on the uniqueKey field declared in the solr schema. If you know the documents you are indexing do not contain any duplicates then you may see a considerable speed up with &overwrite=false.
commit	Commit changes after all records in this request have been indexed. The default is commit=false to avoid the potential performance impact of frequent commits.
wt	(character) One of json (default) or xml. If json, uses fromJSON to parse. If xml, uses read_xml to parse
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to crul::HttpClient

Note

SOLR v1.2 was first version to support csv. See https://issues.apache.org/jira/browse/SOLR-66

See Also

Other update: update_json, update_xml

Examples

```
## Not run:
# start Solr: bin/solr start -f -c -p 8983
```

```
# connect
(cli <- SolrClient$new())

if (!cli$collection_exists("helloWorld")) {
  cli$collection_create(name = "helloWorld", numShards = 2)
}

df <- data.frame(id=1:3, name=c('red', 'blue', 'green'))
write.csv(df, file="df.csv", row.names=FALSE, quote = FALSE)
conn$update_csv("df.csv", "helloWorld", verbose = TRUE)

# give back raw xml
conn$update_csv("df.csv", "helloWorld", wt = "xml")
## raw json
conn$update_csv("df.csv", "helloWorld", wt = "json", raw = TRUE)

## End(Not run)
```

`update_json` *Update documents with JSON data*

Description

Update documents with JSON data

Usage

```
update_json(conn, files, name, commit = TRUE, optimize = FALSE,
  max_segments = 1, expunge_deletes = FALSE, wait_searcher = TRUE,
  soft_commit = FALSE, prepare_commit = NULL, wt = "json", raw = FALSE,
  ...)
```

Arguments

`conn`	A solrium connection object, see SolrClient
`files`	Path to a single file to load into Solr
`name`	(character) Name of the core or collection
`commit`	(logical) If `TRUE`, documents immediately searchable. Deafult: `TRUE`
`optimize`	Should index optimization be performed before the method returns. Default: `FALSE`
`max_segments`	optimizes down to at most this number of segments. Default: 1
`expunge_deletes`	merge segments with deletes away. Default: `FALSE`
`wait_searcher`	block until a new searcher is opened and registered as the main query searcher, making the changes visible. Default: `TRUE`
`soft_commit`	perform a soft commit - this will refresh the 'view' of the index in a more performant manner, but without "on-disk" guarantees. Default: `FALSE`

prepare_commit	The prepareCommit command is an expert-level API that calls Lucene's IndexWriter.prepareCommit(). Not passed by default
wt	(character) One of json (default) or xml. If json, uses fromJSON to parse. If xml, uses read_xml to parse
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to HttpClient

Details

You likely may not be able to run this function against many public Solr services, but should work locally.

See Also

Other update: update_csv, update_xml

Examples

```
## Not run:
# start Solr: bin/solr start -f -c -p 8983

# connect
(conn <- SolrClient$new())

# Add documents
file <- system.file("examples", "books2.json", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_json(files = file, name = "books")
update_json(conn, files = file, name = "books")

# Update commands - can include many varying commands
## Add file
file <- system.file("examples", "updatecommands_add.json",
  package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_json(file, "books")

## Delete file
file <- system.file("examples", "updatecommands_delete.json",
  package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_json(file, "books")

# Add and delete in the same document
## Add a document first, that we can later delete
ss <- list(list(id = 456, name = "cat"))
conn$add(ss, "books")

## End(Not run)
```

update_xml *Update documents with XML data*

Description

Update documents with XML data

Usage

```
update_xml(conn, files, name, commit = TRUE, optimize = FALSE,
  max_segments = 1, expunge_deletes = FALSE, wait_searcher = TRUE,
  soft_commit = FALSE, prepare_commit = NULL, wt = "json", raw = FALSE,
  ...)
```

Arguments

conn	A solrium connection object, see SolrClient
files	Path to a single file to load into Solr
name	(character) Name of the core or collection
commit	(logical) If TRUE, documents immediately searchable. Deafult: TRUE
optimize	Should index optimization be performed before the method returns. Default: FALSE
max_segments	optimizes down to at most this number of segments. Default: 1
expunge_deletes	merge segments with deletes away. Default: FALSE
wait_searcher	block until a new searcher is opened and registered as the main query searcher, making the changes visible. Default: TRUE
soft_commit	perform a soft commit - this will refresh the 'view' of the index in a more performant manner, but without "on-disk" guarantees. Default: FALSE
prepare_commit	The prepareCommit command is an expert-level API that calls Lucene's IndexWriter.prepareCommit(). Not passed by default
wt	(character) One of json (default) or xml. If json, uses fromJSON to parse. If xml, uses read_xml to parse
raw	(logical) If TRUE, returns raw data in format specified by wt param
...	curl options passed on to HttpClient

Details

You likely may not be able to run this function against many public Solr services, but should work locally.

See Also

Other update: update_csv, update_json

Examples

```
## Not run:
# start Solr: bin/solr start -f -c -p 8983

# connect
(conn <- SolrClient$new())

# create a collection
if (!conn$collection_exists("books")) {
  conn$collection_create(name = "books", numShards = 2)
}

# Add documents
file <- system.file("examples", "books.xml", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_xml(file, "books")

# Update commands - can include many varying commands
## Add files
file <- system.file("examples", "books2_delete.xml", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_xml(file, "books")

## Delete files
file <- system.file("examples", "updatecommands_delete.xml",
package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_xml(file, "books")

## Add and delete in the same document
## Add a document first, that we can later delete
ss <- list(list(id = 456, name = "cat"))
conn$add(ss, "books")
## Now add a new document, and delete the one we just made
file <- system.file("examples", "add_delete.xml", package = "solrium")
cat(readLines(file), sep = "\n")
conn$update_xml(file, "books")

## End(Not run)
```

Index