# Package 'rsolr'

May 9, 2026

**Type** Package

**Title** R to Solr Interface

**Version** 0.0.13

**Author** Michael Lawrence, Gabe Becker, Jan Vogel

**Maintainer** Michael Lawrence <michafla@gene.com>

**Description** A comprehensive R API for querying Apache Solr databases.
A Solr core is represented as a data frame or list that
supports Solr-side filtering, sorting,
transformation and aggregation, all through the familiar
base R API. Queries are processed
lazily, i.e., a query is only sent to the database when
the data are required.

**License** Apache License (== 2.0)

**VignetteBuilder** knitr

**Imports** restfulr (>= 0.0.2), graph, S4Vectors (>= 0.14.3), rjson, XML,
RCurl

**Depends** R (>= 3.4.0), BiocGenerics (>= 0.15.1), methods

**Suggests** nycflights13, RUnit, MASS, knitr

**Collate** utils.R pminmax.R Context-class.R DocCollection-class.R
Expression-class.R Facets-class.R FieldInfo-class.R
FieldType-class.R Promise-class.R SolrExpression-class.R
SolrQuery-class.R SolrSchema-class.R SolrCore-class.R
SolrResult-class.R SolrSummary-class.R Solr-class.R
SolrList-class.R SolrFrame-class.R SolrPromise-class.R
GroupedSolrFrame-class.R test.R zzz.R

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2022-05-18 07:10:02 UTC

# Contents



---

`Context-class` *Evaluation Contexts*

---

## Description

The `Context` class is for representing contexts in which expressions are evaluated. This might be an R environment, a database, or some other external system.

## Translation

Contexts play an important role in translation. When extracting an object by name, the context can delegate to a `SymbolFactory` to create a `Symbol` object that is a lazy reference to the object. The reference is expressed in the target language. If there is no `SymbolFactory`, i.e., it has been set to `NULL`, then evaluation is eager.

The intent is to decouple the type of the context from a particular language, since a context could support the evaluation of multiple languages. The accessors below effectively allow one to specify the desired target language.

- `symbolFactory(x)`, `symbolFactory(x) <- value`: Get or set the current `SymbolFactory` (may be NULL).

## Author(s)

Michael Lawrence

## DocCollection-class *DocCollection*

### Description

DocCollection is a virtual class for all representations of document collections. It is made concrete by DocList and DocDataFrame. This is mostly to achieve an abstraction around tabular and list representations of documents.

### Accessors

These are the accessors that should apply equivalently to any derivative of DocCollection, which provides reasonable default implementations for most of them.

- ndoc(x): Gets the number of documents
- nfield(x): Gets the number of fields
- ids(x), ids(x) <- value: Gets or sets the document unique identifiers (may be NULL)
- fieldNames(x, includeStatic=TRUE, ...): Gets the field names
- docs(x): Just returns x, as x already represents a set of documents
- meta(x): Gets an auxillary collection of "meta" fields that hold fields that describe, rather than compose, the documents. This feature should be considered unstable. Stay away for now.
- unmeta(x): Clears the metadata.

### Author(s)

Michael Lawrence

### See Also

DocList and DocDataFrame for concrete implementations

## DocDataFrame-class *DocDataFrame*

### Description

The DocDataFrame object wraps a data.frame in a document-oriented interface that is shared with DocList. This is mostly to achieve an abstraction around tabular and list representations of documents. DocDataFrame should behave just like a data.frame, except it adds the accessors described below.

### Accessors

These are some accessors that DocDataFrame adds on top of the basic data frame accessors. Using these accessors allows code to be agnostic to whether the data are stored as a list or data.frame.

- ndoc(x): Gets the number of documents (rows)
- nfield(x): Gets the number of fields (columns)
- ids(x), ids(x) <- value: Gets or sets the document unique identifiers (may be NULL, treated as rownames)
- fieldNames(x, includeStatic=TRUE, ...): Gets the field (column) names
- docs(x): Just returns x, as x already represents a set of documents
- meta(x): Gets an auxillary data.frame of "meta" columns that hold fields that describe, rather than compose, the documents. This feature should be considered unstable. Stay away for now.
- unmeta(x): Clears the metadata.

### Author(s)

Michael Lawrence

### See Also

DocList for representing a document collection as a list instead of a table

---

## DocList-class *DocList*

---

### Description

The DocList object wraps a list in a document-oriented interface that is shared with DocDataFrame. This is mostly to achieve an abstraction around tabular and list representations of documents. DocList should behave just like a list, except it adds the accessors described below.

### Accessors

These are some accessors that DocList adds on top of the basic list accessors. Using these accessors allows code to be agnostic to whether the data are stored as a list or data.frame.

- ndoc(x): Gets the number of documents (elements)
- nfield(x): Gets the number of unique field names over all of the documents
- ids(x), ids(x) <- value: Gets or sets the document unique identifiers (may be NULL, treated as names)
- fieldNames(x, includeStatic=TRUE, ...): Gets the set of unique field names
- meta(x): Gets an auxillary list of "meta" documents (lists) that hold fields that describe, rather than compose, the actual documents. This feature should be considered unstable. Stay away for now.
- unmeta(x): Clears the metadata.

### Author(s)

Michael Lawrence

### See Also

DocDataFrame for representing a document collection as a table instead of a list

---

## Expression-class *Expressions and Translation*

---

### Description

Underlying rsolr is a simple, general framework for representing, manipulating and translating between expressions in arbitrary languages. The two foundational classes are `Expression` and `Symbol`, which are partially implemented by `SimpleExpression` and `SimpleSymbol`, respectively.

### Translation

The `Expression` framework defines a translation strategy based on evaluating source language expressions, using promises to represent the objects, such that the result is a promise with its deferred computation expressed in the target language.

The primary entry point is the `translate` generic, which has a default method that abstractly implements this strategy. The first step is to obtain a `SymbolFactory` instance for the target expression type via a method on the `SymbolFactory` generic. The `SymbolFactory` (a simple R function) is set on the `Context`, which should define (perhaps through inheritance) all symbols referenced in the source expression. The translation happens when the source expression is `eval`uated in the context. The context calls the factory to construct `Symbol` objects which are passed, along with the context, to the `Promise` generic, which wraps them in the appropriate type of promise. Typically, R is the source language, and the `eval` method evaluates the R expression on the promises. Each method for the specific type of promise will construct a new promise with an expression that encodes the computation, building on the existing expression. When evaluation is finished, we simply extract the expression from the returned promise.

- `translate(x, target, context, ...)`: Translates the source expression `x` to the `target Expression`, where the symbols in the source expression are resolved in `context`, which is usually an R environment or some sort of database. The `...` are passed to `symbolFactory`.
- `symbolFactory(x)`: Gets the `SymbolFactory` object that will construct the appropriate type of symbol for the target expression `x`.

### Note on Laziness

In general, translation requires access to the referenced data. There may be certain operations that cannot be deferred, so evaluation is allowed to be eager, in the hope that the result can be embedded directly into the larger expression. Or, at the very least, the translation machinery needs to know whether the data actually exist, and whether the data are typed or have other constraints. Since the data and schema are not always available when translation is requested, such as when building a

database query that will be sent to by another module to an as-yet-unspecified endpoint, translation itself must be deferred. The `TranslationRequest` class provides a foundation for capturing translations and evaluating them later.

### Author(s)

Michael Lawrence

---

`Facets-class` *Facets*

---

### Description

The `Facets` object represents the result of a Solr facet operation and is typically obtained by calling `facets` on a `SolrCore`. Most users should just call `aggregate` or `xtabs` instead of directly manipulating `Facets` objects.

### Details

`Facets` extends `list` and each node adds a grouping factor to the set defined by its ancestors. In other words, parent-child relationships represent interactions between factors. For example, `x$a$b` gets the node corresponding to the interaction of `a` and `b`.

In a single request to Solr, statistics may be calculated for multiple interactions, and they are stored as a `data.frame` at the corresponding node in the tree. To retrieve them, call the `stats` accessor, e.g., `stats(x$a$b)`, or `as.table` for getting the counts as a table (Solr always computes the counts).

### Accessors

- `x$name`, `x[[i]]`: Get the node that further groups by the named factor. The `i` argument can be a formula, where `[[` will recursively extract the corresponding element.
- `x[i]`: Extract a new `Facets` object, restricted to the named groupings.
- `stats(x)`: Gets the statistics at the current facet level.

### Coercion

- `as.table(x)`: Converts the current node to a table of conditional counts.

### Author(s)

Michael Lawrence

### See Also

`aggregate` for a simpler interface that computes statistics for only a single interaction

FieldInfo-class *FieldInfo*

## Description

The FieldInfo object is a vector of field entries from the Solr schema. Typically, one retrieves an instance with fields and shows it on the console to get an overview of the schema. The vector-like nature means that functions like [ and length behave as expected.

## Accessors

These functions get the "columns" from the field information "table":

- name(x): Gets the name of the field.
- typeName(x): Gets the name of the field type, see fieldTypes.
- dynamic(x): Gets whether the field is dynamic, i.e., whether its name is treated as a wildcard glob. If a document field does not match a static field name, it takes its properties from the first dynamic field (in schema order) that it matches.
- multiValued(x): Gets whether the field accepts multiple values. A multi-valued field is manifested in R as a list.
- required(x): Gets whether the field must have a value in every document. A non-required field will sometimes have NAs. This is useful for both ensuring data integrity and optimizations.
- indexed(x): Gets whether the field has been indexed. A field must be indexed for us to filter by it. Faceting requires a field to be indexed or have doc values.
- stored(x): Gets whether the data for a field have been stored in the database. We can search on any (indexed) field, but we can only retrieve data from stored fields.
- docValues(x): Gets whether the data have been additionally stored in a columnar format that accelerates Solr function calls (transform) and faceting (aggregate).

## Utilities

- x %in% table: Returns whether each field name in x matches a field defined in table, a FieldInfo object. This convenience is particularly needed when the schema contains dynamic fields.

## Author(s)

Michael Lawrence

## See Also

SolrSchema that holds an instance of this object

## FieldType-class *FieldType*

### Description

The FieldType object represents the type of a document field. A list of these objects is formally represented as FieldTypeList object, an instance of which is provided by SolrSchema. Internally, FieldType objects are central to the conversion between R and Solr types. At the user level, they are mostly useful for displaying the schema.

### Author(s)

Michael Lawrence

### See Also

SolrSchema, which communicates information on field types using these classes

## GroupedSolrFrame-class *GroupedSolrFrame*

### Description

The GroupedSolrFrame is a *highly experimental* extension of SolrFrame that models each column as a list, formed by splitting the original vector by a common set of grouping factors.

### Details

A GroupedSolrFrame should more or less behave analogously to a data frame where every column is split by a common grouping. Unlike SolrFrame, columns are *always* extracted lazily. Typical usage is to construct a GroupedSolrFrame by calling group on a SolrFrame, and then to extract columns (as promises) and aggregate them (by e.g. calling mean).

Functions that group the data, such as group and aggregate, simply add to the existing grouping. To clear the grouping, call ungroup or just coerce to a SolrFrame or SolrList.

### Accessors

As GroupedSolrFrame inherits much of its functionality from SolrFrame; here we only outline concerns specific to grouped data.

- ndoc(x): Gets the number of documents per group
- rownames(x): Forms unique group identifiers by concatenating the grouping factor values.

- `x[i, j] <- value`: Inserts `value` into the Solr core, where `value` is a data.frame of lists, or just a list (representing a single column). Preferably, `i` is a promise, because we need to the IDs of the selected documents in order to perform the atomic update, and the promise lets us avoid downloading *all* of the IDs. But otherwise, if `i` is atomic, then it indexes into the groups. If `i` is a list, then its names are matched to the group names, and its elements index into the matching group. The list does not need to be named if the elements are character vectors (and thus represent document IDs).
- `x[i, j, drop=FALSE]`: Extracts data from `x`, as usual, but see the entry immediate above this one for the expectations of `i`. Try to make it a promise, so that we do not need to download IDs and then try to serialize them into a query, which has length limitations.

## Extended API

Most of the typical data frame accessors and data manipulation functions will work analogously on `GroupedSolrFrame` (see Details). Below, we list some of the non-standard methods that might be seen as an extension of the data frame API.

- `heads(x, n)`, `tails(x, n)`, `windows(x, start, end)`: Perform `head`, `tail` or `window` on each group separately, returning a data.frame with grouped (list) columns.
- `ngroup(x)`: The number of groups, i.e., the number of rows.

## Author(s)

Michael Lawrence

---

`Grouping-class` *Grouping*

---

## Description

The `Grouping` object represents a collection of documents split by some interaction of factors. It is extremely low-level, and its only use is to be coerced to something else, either a `list` or `data.frame`, via `as`.

## Author(s)

Michael Lawrence

## See Also

`ListSolrResult`, which provides this object via its `groupings` method.

## ListSolrResult-class *ListSolrResult*

### Description

The SolrResult object represents the result of a Solr query and usually contains a collection of documents and/or facets. The default implementation, ListSolrResult, directly stores the canonical JSON response from Solr. It is usually obtained by evaluating a SolrQuery on a SolrCore, which most users will never do.

### Accessors

Since ListSolrResult inherits from list, one can access the raw JSON fields directly through the ordinary list accessors. One should only directly manipulate the Solr response when extending rsolr/Solr at a deep level. Higher-level accessors are described below.

- docs(x): Returns the found documents as a DocList
- ndoc(x): Returns the number of documents found
- facets(x): Returns any computed Facets
- groupings(x): If Solr was asked to group the documents in the response, this returns each Grouping (there can be more than one) in a list
- ngroup(x): Returns the number of groups in each grouping

### Author(s)

Michael Lawrence

### See Also

docs and facets on SolrCore are more convenient and usually sufficient

## Promise-class *Promises*

### Description

The Promise class formally and abstractly represents the potential result of a deferred computation.

## Details

Lazy programming is useful in a number of contexts, including interaction with external/remote systems like databases, where we want the computation to occur within the external system, despite appearances to the contrary. Typically, the user constructs one or more promises referring to pre-existing objects. Operations on those objects produce new promises that encode the additional computations. Eventually, usually after some sort of restriction and/or aggregation, the promise is "fulfilled" to yield a materialized, eager object, such as an R vector.

`Promise` and its partial implementation `SimplePromise` provide a foundation for implementations that mostly helps with creating and fulfilling promises, while the implementation is responsible for deferring particular computations, which is language-dependent.

## Construction

- `Promise(expr, context, ...)`: A generic constructor that dispatches on `expr` to construct a `Promise` object, the specific type of which corresponds to the language of `expr`. The `context` argument should be a `Context` object, in which `expr` will be evaluated when the promise is fulfilled. The `...` are passed to methods.

## Fulfillment

- `fulfill(x)`: Fulfills the promise by evaluating the deferred computation and returning a materialized object.

The basic coercion functions in R, like `as.vector` and `as.data.frame`, have methods for `Promise` that simply call `fulfill` on the promise, and then perform the coercion. Coercion is preferred to calling `fulfill` directly.

## Author(s)

Michael Lawrence

---

`SolrCore-class` *SolrCore*

---

## Description

The `SolrCore` object represents a core hosted by a Solr instance. A core is essentially a queryable collection of documents that share the same schema. It is usually not necessary to interact with a `SolrCore` directly.

## Details

The typical usage (by advanced users) would be to construct a custom `SolrQuery` and execute it via the `docs`, `facets` or (the very low-level) `eval` methods.

### Accessor methods

In the code snippets below, x is a SolrCore object.

- name(x): Gets the name of the core (specified by the schema).
- ndoc(x, query = SolrQuery()): Gets the number of documents in the core, given the query restriction.
- schema(x): Gets the SolrSchema satisfied by all documents in the core.
- fieldNames(x, query = NULL, onlyStored = FALSE, onlyIndexed = FALSE, includeStatic = FALSE): Gets the field names, given any restriction and/or transformation in query, which is a SolrQuery or a character vector of field patterns. The onlyIndexed and onlyStored arguments restrict the fields to those indexed and stored, respectively (see FieldInfo for more details). Setting includeStatic to TRUE ensures that all of the static fields in the schema are returned.
- version(x): Gets the version of the Solr instance hosting the core.

### Constructor

- SolrCore(uri, ...): Constructs a new SolrCore instance, representing a Solr core located at uri, which should be a string or a RestUri object. If a string, then the ... are passed to the RestUri constructor.

### Reading

- docs(x, query = SolrQuery(), as=c("list", "data.frame")): Get the documents selected by query, in the form indicated by as, i.e., either a list or a data frame.
- read(x, ...): Just an alias for docs.

### Summarizing

- facets(x, by, ...): Gets the Facets results as requested by by, a SolrQuery. The ... are passed down to facets on ListSolrResult.
- groupings(x, by, ...): Gets the list of Grouping objects as requested by the grouped query by. The ... are passed down to groupings on ListSolrResult.
- ngroup(x): Gets the number of groupings that would be returned by groupings.

### Updating

- update(object, value, commit = TRUE, atomic = FALSE, ...): Load the documents in value (typically a list or data frame) into the SolrCore given by object. If commit is TRUE, we request that Solr commit the changes to its index on disk, with arguments in ... fine-tuning the commit (see commit). If atomic is TRUE, then the existing documents are modified, rather than replaced, by the documents in value.
- delete(x, which = SolrQuery(), ...): Deletes the documents specified by which (all by default), where the ... are passed down to update.
- commit(x, waitSearcher=TRUE, softCommit=FALSE, expungeDeletes=FALSE, optimize=TRUE, maxSegments=if (optimize) 1L): Commits the changes to the Solr index; see the Solr documentation for the meaning of the parameters.

- `purgeCache(x)`: Purges the client-side HTTP cache, which is useful if the Solr instance is using expiration-based HTTP caching and one needs to see the result of an update immediately.

### Evaluation

- `eval(expr, envir, enclos)`: Evaluates the query `expr` in the core `envir`, ignoring `enclos`. Unless otherwise requested by the query response type, the result should be returned as a `ListSolrResult`.

### Coercion

- `as.data.frame(x, row.names=NULL, optional=FALSE, ...)`:

### Author(s)

Michael Lawrence

### See Also

`SolrFrame`, the typical way to interact with a Solr core.

### Examples

```
solr <- TestSolr()
sc <- SolrCore(solr$uri)
name(sc)
ndoc(sc)

delete(sc)

docs <- list(
   list(id="2", inStock=TRUE, price=2, timestamp_dt=Sys.time()),
   list(id="3", inStock=FALSE, price=3, timestamp_dt=Sys.time()),
   list(id="4", price=4, timestamp_dt=Sys.time()),
   list(id="5", inStock=FALSE, price=5, timestamp_dt=Sys.time())
)
update(sc, docs)

q <- SolrQuery(id %in% as.character(2:4))
read(sc, q)

solr$kill()
```

## SolrExpression-class *SolrExpression*

### Description

There is a formal framework for constructing and manipulating the Solr languages that is not yet exposed. Please inform the authors if exposing the framework would be helpful. Perhaps it would be helpful in support of implementing new functionality on top of `SolrPromise`.

### Author(s)

Michael Lawrence

## SolrFrame-class *SolrFrame*

### Description

The `SolrFrame` object makes Solr data accessible through a data.frame-like interface. This is the typical way an R user accesses data from a Solr core. Much of its methods are shared with `SolrList`, which has very similar behavior.

### Details

A `SolrFrame` should more or less behave analogously to a data frame. It provides the same basic accessors (`nrow`, `ncol`, `length`, `rownames`, `colnames`, `[`, `[<-`, `[[`, `[[<-`, `$`, `$<-`, `head`, `tail`, etc) and can be coerced to an actual data frame via `as.data.frame`. Supported types of data manipulations include `subset`, `transform`, `sort`, `xtabs`, `aggregate`, `unique`, `summary`, etc.

Mapping a collection of documents to a tablular data structure is not quite natural, as the document collection is ragged: a given document can have any arbitrary set of fields, out of a set that is essentially infinite. Unlike some other document stores, however, Solr constrains the type of every field through a schema. The schema achieves flexibility through "dynamic" fields. The name of a dynamic field is a wildcard pattern, and any document field that matches the pattern is expected to obey the declared type and other constraints.

When determining its set of columns, `SolrFrame` takes every actual field present in the collection, and (by default) adds all non-dynamic (static) fields, in the order specified by the schema. Note that is very likely that many columns will consist entirely or almost entirely of NAs.

If a collection is extremly ragged, where few fields are shared between documents, it may make more sense to treat the data as a list, through `SolrList`, which shares almost all of the functionality of `SolrFrame` but in a different shape.

The rownames are taken from the field declared in the schema to represent the unique document key. Schemas are not strictly required to declare such a field, so if there is no unique key, the rownames are `NULL`.

Field restrictions passed to e.g. [ or subset(fields=) may be specified by name, or wildcard pattern (glob). Similarly, a row index passed to [ must be either a character vector of identifiers (of length <= 1024, NAs are not supported, and this requires a unique key in the schema) or a SolrPromise/SolrExpression, but note that if it evaluates to NAs, the corresponding rows are excluded from the result, as with subset. Using a SolrPromise or SolrExpression is recommended, as filtering happens at the database.

A special feature of SolrFrame, vs. an ordinary data frame, is that it can be grouped into a GroupedSolrFrame, where every column is modeled as a list, split by some combination of grouping factors. This is useful for aggregation and supports the implementation of the aggregate method, which is the recommended high-level interface.

Another interesting feature is laziness. One can defer a SolrFrame, so that all column retrieval, e.g., via $ or eval, returns a SolrPromise object. Many operations on promises are deferred, until they are finally fulfilled by being shown or through explicit coercion to an R vector.

A note for developers: SolrList and SolrFrame share common functionality through the base Solr class. Much of the functionality mentioned here is actually implemented as methods on the Solr class.

## Accessors

These are some accessors that SolrFrame adds on top of the basic data frame accessors. Most of these are for advanced use only.

- ndoc(x): Gets the number of documents (rows); serves as an abstraction over SolrFrame and SolrList
- nfield(x): Gets the number of fields (columns); serves as an abstraction over SolrFrame and SolrList
- ids(x): Gets the document unique identifiers (may be NULL, treated as rownames); serves as an abstraction over SolrFrame and SolrList
- fieldNames(x, includeStatic=TRUE, ...): Gets the name of each field represented by any document in the Solr core, with ... being passed down to fieldNames on SolrCore. Fields must be indexed to be reported, with the exception that when includeStatic is TRUE, we ensure all static (non-dynamic) fields are present in the return value. Names are returned in an order consistent with the order in the schema. Note that two different "instances" of the same dynamic field do not have a specified order in the schema, so we use the index order (lexicographical) for those cases.
- core(x): Gets the SolrCore wrapped by x
- query(x): Gets the query that is being constructed by x

## Extended API

Most of the typical data frame accessors and data manipulation functions will work analogously on SolrFrame (see Details). Below, we list some of the non-standard methods that might be seen as an extension of the data frame API.

- aggregate(x, data, FUN, ..., subset, na.action, simplify = TRUE, count = FALSE): If x is a formula, aggregates data, grouping by x, by either applying FUN, or evaluating an aggregating expression in ..., on each group. If count is TRUE, a "count" column is added with

the number of elements in each group. The rest of the arguments behave like those for the base aggregate.

There are two main modes: aggregating with FUN, or, as an extension to the base aggregate, aggregating with expressions in ..., similar to the interface for transform. If FUN is specified, then behavior is much like the original, except one can omit the LHS on the formula, in which case the entire frame is passed to FUN. In the second mode, there is a column in the result for each argument in ..., and there must not be an LHS on the formula.

See the documentation for the underlying facet function for details on what is supported on the formula RHS.

For global aggregation, simply pass the SolrFrame as x, in which case the data argument does not exist.

Note that the function or expressions are only *conceptually* evaluated on each group. In reality, the computations occur on grouped columns/promises, which are modeled as lists. Thus, there is potential for conflict, in particular with length, which return the number of groups, instead of operating group-wise. One should use the abstraction ndoc instead of length, since ndoc always returns document counts, and thus will return the size of each group.

- rename(x, ...): Renames the columns of x, where the names and character values of ... indicates the mapping (newname = oldname).
- group(x, by): Returns a GroupedSolrFrame that is grouped by the factors in by, typically a formula. To get back to x, call ungroup(x).
- grouping(x): Just returns NULL, since a SolrFrame is not grouped (unless extended to be groupable).
- defer(x): Returns a SolrFrame that yields SolrPromise objects instead of vectors whenever a field is retrieved
- searchDocs(x, q): Performs a conventional document search using the query string q. The main difference to filtering is that (by default) Solr will order the result by score, i.e., how well each document matches the query.

## Constructor

- SolrFrame(uri): Constructs a new SolrFrame instance, representing a Solr core located at uri, which should be a string or a RestUri object. The ... are passed to the SolrQuery constructor.

## Evaluation

- eval(expr, envir, enclos): Evaluates expr in the SolrFrame envir, using enclos as the enclosing environment. The expr can be an R language object or a SolrExpression, either of which are lazily evaluated if defer has been called on envir.

## Coercion

- as.data.frame(x, row.names=NULL, optional=FALSE, fill=TRUE): Downloads the data into an actual data.frame, specifically an instance of DocDataFrame. If fill is FALSE, only the fields represented in at least one document are added as columns.
- as.list(x): Essentially as.list(as.data.frame(x)), except returns a list of promises if x is deferred.

## Author(s)

Michael Lawrence

## See Also

SolrList for representing a Solr collection as a list instead of a table

## Examples

```
schema <- deriveSolrSchema(mtcars)
solr <- TestSolr(schema)
sr <- SolrFrame(solr$uri)
sr[] <- mtcars
dim(sr)
head(sr)
subset(sr, mpg > 20 & cyl == 4)
solr$kill()
## see the vignette for more
```

---

# SolrList-class *SolrList*

---

## Description

The SolrList object makes Solr data accessible through a list-like interface. This interface is appropriate when the data are highly ragged.

## Details

A SolrList should more or less behave analogously to a list. It provides the same basic accessors (length, names, [, [<-, [[, [[<-, $, $<-, head, tail, etc) and can be coerced to a list via as.list. Supported types of data manipulations include subset, transform, sort, xtabs, aggregate, unique, summary, etc.

An obvious difference between a SolrList and an ordinary list is that we know the SolrList contains only documents, which are themselves represented as named lists of fields, usually vectors of length one. This constraint enables us to provide the convenience of accessing fields by slicing across every document. We can pass a field selection to the second argument of [. Like data frame, selecting a single column with e.g. x[,"foo"] will return the field as a vector, filling NAs whereever a document lacks a value for the field.

The names are taken from the field declared in the schema to represent the unique document key. Schemas are not strictly required to declare such a field, so if there is no unique key, the names are NULL.

Field restrictions passed to e.g. [ or subset(fields=) may be specified by name, or wildcard pattern (glob). Similarly, a row index passed to [ must be either a character vector of identifiers (of length <= 1024, NAs are not supported, and this requires a unique key in the schema) or a SolrPromise/SolrExpression, but note that if it evaluates to NAs, the corresponding rows are

excluded from the result, as with subset. Using a SolrPromise or SolrExpression is recommended, as filtering happens at the database.

A SolrList can be made lazy by calling defer on a SolrList, so that all column retrieval, e.g., via [, returns a SolrPromise object. Many operations on promises are deferred, until they are finally fulfilled by being shown or through explicit coercion to an R vector.

A note for developers: SolrFrame and SolrList share common functionality through the base Solr class. Much of the functionality mentioned here is actually implemented as methods on the Solr class.

## Accessors

These are some accessors that SolrList adds on top of the basic data frame accessors. Most of these are for advanced use only.

- ndoc(x): Gets the number of documents (rows); serves as an abstraction over SolrFrame and SolrList
- nfield(x): Gets the number of fields (columns); serves as an abstraction over SolrFrame and SolrList
- ids(x): Gets the document unique identifiers (may be NULL, treated as rownames); serves as an abstraction over SolrFrame and SolrList
- fieldNames(x, ...): Gets the name of each field represented by any document in the Solr core, with ... being passed down to fieldNames on SolrCore.
- core(x): Gets the SolrCore wrapped by x
- query(x): Gets the query that is being constructed by x

## Extended API

Most of the typical data frame accessors and data manipulation functions will work analogously on SolrList (see Details). Below, we list some of the non-standard methods that might be seen as an extension of the data frame API.

- rename(x, ...): Renames the columns of x, where the names and character values of ... indicates the mapping (newname = oldname).
- defer(x): Returns a SolrList that yields SolrPromise objects instead of vectors whenever a field is retrieved
- searchDocs(x, q): Performs a conventional document search using the query string q. The main difference to filtering is that (by default) Solr will order the result by score, i.e., how well each document matches the query.

## Constructor

- SolrList(uri, ...): Constructs a new SolrList instance, representing a Solr core located at uri, which should be a string or a RestUri object. The ... are passed to the SolrQuery constructor.

### Evaluation

- `eval(expr, envir, enclos)`: Evaluates R language `expr` in the `SolrList` `envir`, using `enclos` as the enclosing environment.

### Coercion

- `as.data.frame(x, row.names=NULL, optional=FALSE, fill=FALSE)`: Downloads the data into an actual data.frame, specifically an instance of `DocDataFrame`. If `fill` is FALSE, only the fields represented in at least one document are added as columns.
- `as.list(x)`, `as(x, "DocCollection")`: Coerces `x` into the corresponding list, specifically an instance of `DocList`.

### Author(s)

Michael Lawrence

### See Also

`SolrFrame` for representing a Solr collection as a table instead of a list

### Examples

```
solr <- TestSolr()
sr <- SolrList(solr$uri)
length(sr)
head(sr)
sr[["GB18030TEST"]]
# Solr tends to crash for some reason running this inside R CMD check
## Not run:
as.list(subset(sr, price > 100))[,"price"]

## End(Not run)
solr$kill()
```

---

## SolrPromise-class *SolrPromise*

---

### Description

`SolrPromise` is a vector-like representation of a deferred computation within Solr. It may promise to simply return a field, to perform arithmetic on a combination of fields, to aggregate a field, etc. Methods on `SolrPromise` allow the R user to manipulate Solr data with the ordinary R API. The typical way to fulfill a promise is to explicitly coerce the promise to a materialized data type, such as an R vector.

## Details

In general, SolrPromise acts just like an R vector. It supports all of the basic vector manipulations, including the Logic, Compare, Arith, Math, and Summary group generics, as well as length, lengths, %in%, complete.cases, is.na, [, grepl, grep, round, signif, ifelse, pmax, pmin, cut, mean, quantile, median, weighted.mean, IQR, mad, anyNA. All of these functions are lazy, in that they return another promise.

The promise is really only known to rsolr, as all actual Solr queries are eager. SolrPromise does its best to defer computations, but the computations will be forced if one performs an operation that is not supported by Solr.

These functions are also supported, but they are eager: cbind, rbind, summary, window, head, tail, unique, intersect, setdiff, union, table and ftable. These functions from the Math group generic are eager: cummax, cummin, cumprod, cumsum, log2, and *gamma.

The [<- function will be lazy as long as both x and i are promises. i is assumed to represent a logical subscript. Otherwise, [<- is eager.

SolrPromise also extends the R API with some new operations: nunique (number of unique elements), rescale (rescale to within a min/max), ndoc, windows, heads, tails.

## Limitations

This section outlines some limitations of SolrPromise methods, compared to the base vector implementation. The primary limitation is that binary operations generally only work between two promises that derive from the same data source, including all pending manipulations (filters, ordering, etc). Operations between a promise and an ordinary vector usually only work if the vector is of length one (a scalar).

Some specific notes:

- x[i]: The index i is ideally a promise. The return value will be restricted such that it will only combine with promises with the same restriction.
- x %in% table: The x argument must always refer to a simple field, and the table argument should be either a field, potentially predicated via table[i] (where the index i is a promise), or a "short" vector.
- grepl(pattern, x, fixed = FALSE): Applies when x is a promise. Besides pattern, only the fixed argument is supported from the base function.
- grep(pattern, x, value = FALSE, fixed = FALSE, invert = FALSE): One must always set value=TRUE. Beyond that, only fixed and invert are supported from the base function.
- cut(x, breaks, include.lowest = FALSE, right = TRUE): Only supports uniform (constant separation) breaks.
- mad(x, center = median(x, na.rm=na.rm), constant = 1.4826, na.rm = FALSE, low = FALSE, high = FALSE): The low and high parameters must be FALSE. If there any NAs, then na.rm must be TRUE. Does not work when the context is grouped.

## Author(s)

Michael Lawrence

## See Also

SolrFrame, which yields promises when it is deferred.

---

SolrQuery-class *SolrQuery*

---

## Description

The SolrQuery object represents a query to be sent to a SolrCore. This is a low-level interface to query construction but will not be useful to most users. The typical reason to directly manipulate a query would be to batch more operations than is possible with the high-level SolrFrame, e.g., combining multiple aggregations.

## Details

A SolrQuery API borrows many of the same verbs from the base R API, including subset, transform, sort, xtabs, head, tail, rev, etc.

The typical workflow is to construct a query, perform various manipulations, and finally retrieve a result by passing the query to a SolrCore, typically via the docs or facets functions.

## Accessors

- params(x), params(x) <- value: Gets/sets the parameters of the query, which roughly correspond to the parameters of a Solr "select" request. The only reason to manipulate the underlying query parameters is to either initiate a headache or to do something really tricky with Solr, which implies the former.

## Querying

- subset(x, subset, select, fields, select.from = character()): Behaves like the base subset, with some extensions. The fields argument is exclusive with select, and should be a character vector of field names, potentially with wildcards. The select.from argument gives the names that are filtered by select, since SolrQuery is not associated with any SolrCore, and thus does not know the field set (in the future, we might use laziness to avoid this problem).
- searchDocs(x, q): Performs a conventional document search using the query string q. The main difference to filtering (subset) is that (by default) Solr will order the result by score, i.e., how well each document matches the query.

## Constructor

- SolrQuery(expr): Constructs a new SolrQuery instance. If expr is non-missing, it is passed to subset and thus serves as an initial restriction.

### Faceting

The Solr facet component counts documents and calculates statistics on a group-wise basis.

- `facet(x, by, ..., useNA=FALSE, sort=NULL, decreasing=FALSE, limit=NA_integer_)`: Returns a query that will compute the number of documents in each group, where the grouping is given as `by`, typically a formula, or `NULL` for global aggregation. Arguments in `...` are quoted and should be expressions that summarize fields, or mathematical combinations of fields. The names of the statistics are taken from the argument names; if a name is omitted, a best guess is made from the expression. If `useNA` is `TRUE`, statistics and counts are computed for the bin where documents have a missing value for one the grouping variables. If `sort` is non-NULL, it should name a statistic by which the results should be sorted. This is mostly useful in conjunction if a `limit` is specified, so that only the top-N statistics are returned.

  The formula should consist of Solr field names, or calls that evaluate to logical and refer to one or more Solr fields. If the latter, the results are grouped by `TRUE`, `FALSE` and (optionally) `NA` for that term. As a special case, a term can be a call to `cut` on any numeric or date field, which will group by bin.

### Grouping

The Solr grouping component causes results to be returned nested into groups. The main use case would be to restrict to the first or last N documents in each group. This functionality is *not* related to aggregation; see `facet`.

- `group(x, by, limit = .Machine$integer.max, offset = 0L, env = emptyenv())`: Returns the grouping of `x` according to `by`, which might be a formula, or an expression that evaluates (within `env`) to a factor. The current sort specification applies within the groups, and any subsequent sorting applies to the groups themselves, by using the maximum value within the each group. Only the top `limit` documents, starting after the first `offset`, are returned from each group. Restricting that limit is probably the main reason to use this functionality.

### Coercion

These two functions are very low-level; users should almost never need to call these.

- `translate(x, target, core)`: Translates the query `x` into the language of Solr, where `core` specifies the destination `SolrCore`. The `target` argument should be missing.
- `as.character(x)`: Converts the query into a string to be sent to Solr. Remember to translate first, if necessary.

### Author(s)

Michael Lawrence

### See Also

`SolrFrame`, the recommended high-level interface for interacting with Solr

`SolrCore`, which gives an example of constructing and evaluating a query

SolrSchema-class *SolrSchema*

## Description

The SolrSchema object represents the schema of a Solr core. Not all of the information in the schema is represented; only the relevant elements are included. The user should not need to interact with this class very often.

One can infer a SolrSchema from a data.frame with deriveSolrSchema and then write it out to a file for use with Solr.

## Accessors

- name(x): Gets the name of the schema/dataset.
- uniqueKey(x): Gets the field that serves as the unique key, i.e., the document identifier.
- fields(x, which): Gets a FieldInfo object, restricted to the fields indicated by which.
- fieldTypes(x, fields): Gets a FieldTypeList object, containing the type definition for each field named in fields.
- copyFields(x): Gets the copy field relationships as a graph.

## Generation and Export

It may be convenient for R users to autogenerate a Solr schema from a prototypical data frame. Note that to harness the full power of Solr, it pays to get familiar with the details. After deriving a schema with deriveSolrSchema, save it to the standard XML format with saveXML. See the vignette for an example.

- deriveSolrSchema(x, name, version="1.5", uniqueKey=NULL, required=colnames(Filter(Negate(anyEmpty x)), indexed=colnames(x), stored=colnames(x), includeVersionField=TRUE): Derives a SolrSchema from a data.frame (or data.frame-coercible) x. The name is taken by quoting x, by default. Specify a unique key via uniqueKey. The required fields are those that are not allowed to contain missing/empty values. By default, we guess that a field is required if it does not contain any NAs or empty strings (both are the same as far as Solr is concerned). The indexed and stored arguments name the fields that should be indexed and stored, respectively (see Solr docs for details). If includeVersionField is TRUE, the magic _version_ field is added to the schema, and Solr will use it to track document versions, which is needed for certain advanced features and generally recommended.
- saveXML(doc, file = NULL, compression = 0, indent = TRUE, prefix = "<?xml version=\"1.0\"?>\n", doctype = NULL, encoding = getEncoding(doc), ...): Writes the schema to XML. See saveXML for more details.

## Author(s)

Michael Lawrence

TestSolr *Testing Solr*

## Description

Launches an instance of the embedded Solr and creates a core for testing and demonstration purposes.

## Usage

```
TestSolr(schema = NULL, start = TRUE, restart = FALSE)
```

## Arguments

| | |
|---|---|
| schema | The SolrSchema object describing the schema for the new Solr core |
| start | Whether to actually start the server (it can be started later by interacting with the returned object). If there is already a server running, the return value points to that instance. |
| restart | Force the Solr server to restart. |

## Value

An instance of ExampleSolr, a reference class. Typically, one just accesses the uri field, and passes it to a constructor of SolrFrame or SolrCore.

## Author(s)

Michael Lawrence

# Index