Package 'splitGraph'

May 9, 2026

Title Dataset Dependency Graphs for Leakage-Aware Evaluation

Version 0.2.0

Description Represent biomedical dataset structure as typed dependency graphs so that sample provenance, repeated-measure structure, study design, batch effects, and temporal relationships are explicit and inspectable. Validates dataset structure, detects sample-level overlap, derives deterministic split constraints, and produces a tool-agnostic split specification for leakage-aware evaluation workflows.

License MIT + file LICENSE

URL https://github.com/selcukorkmaz/splitGraph

BugReports https://github.com/selcukorkmaz/splitGraph/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports graphics, igraph

Suggests jsonlite, knitr, pkgload, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

RoxygenNote 7.3.3

Author Selcuk Korkmaz [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4632-6850>)

Maintainer Selcuk Korkmaz <selcukorkmaz@gmail.com>

Repository CRAN

Date/Publication 2026-04-29 20:30:02 UTC

Contents

`as_split_spec` *Translate splitGraph Constraints into Stable Split Specifications*

Description

Translate graph-derived split constraints into a stable, inspectable structure for sample-level grouping, blocking, and ordering, perform preflight structural checks on that translation, and summarize structural leakage risks.

Usage

```
as_split_spec(constraint, graph = NULL)

validate_split_spec(x)

summarize_leakage_risks(
  graph,
  constraint = NULL,
  split_spec = NULL,
  validation = NULL
)
```

Arguments

`constraint`	A `split_constraint`.
`graph`	A `dependency_graph`.
`x`	A `split_spec`.
`split_spec`	An optional `split_spec`.
`validation`	An optional `depgraph_validation_report`.

Details

The translation layer always produces canonical sample-level columns including `sample_id`, `sample_node_id`, `group_id`, and `primary_group`. When available, it also carries `batch_group`, `study_group`, `timepoint_id`, `time_index`, and `order_rank`. Missing but relevant fields are retained as `NA` columns rather than omitted.

When only a subset of samples has ordering metadata, the translated split spec still exposes that partial ordering through `time_var`, but `ordering_required` remains `FALSE`. Ordering is only marked as required when the constraint implies complete ordering coverage.

The split-spec validator checks:

- missing required columns
- missing or duplicated sample identifiers
- missing grouping assignments
- singleton-only grouping structures
- missing ordering when ordering is required
- invalid or empty block variables

Repeated validation of the same split spec yields deterministic issue IDs and diagnostics, which makes the returned validation object stable across runs.

The produced `split_spec` is tool-agnostic. Downstream consumers are expected to provide their own adapters to convert a `split_spec` into their native split representation, so **splitGraph** has no runtime dependency on any of them.

`summarize_leakage_risks()` reuses `validate_graph()` and `split_constraint` metadata rather than duplicating downstream evaluation logic.

Value

`as_split_spec()` returns a `split_spec`. `validate_split_spec()` returns a `split_spec_validation`. `summarize_leakage_risks()` returns a `leakage_risk_summary`.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2")
)
g <- graph_from_metadata(meta)

constraint <- derive_split_constraints(g, mode = "subject")
spec <- as_split_spec(constraint, graph = g)
validate_split_spec(spec)
summarize_leakage_risks(g, constraint = constraint, split_spec = spec)
```

build_dependency_graph

Assemble and Validate Dependency Graphs

Description

Combine canonical node and edge tables into a typed dependency graph and perform structural, semantic, and graph-local leakage-aware validation.

Usage

```
build_dependency_graph(
  nodes,
  edges,
  graph_name = NULL,
  dataset_name = NULL,
  validate = TRUE,
  validation_overrides = list()
)

build_depgraph(
  nodes,
  edges,
  graph_name = NULL,
  dataset_name = NULL,
  validate = TRUE,
  validation_overrides = list()
)

as_igraph(x)

validate_graph(
  graph,
  checks = c("ids", "references", "cardinality", "schema", "time"),
  error_on_fail = FALSE,
  levels = NULL,
  severities = NULL,
  validation_overrides = NULL
)

validate_depgraph(
  graph,
  checks = c("ids", "references", "cardinality", "schema", "time"),
  error_on_fail = FALSE,
  levels = NULL,
  severities = NULL,
  validation_overrides = NULL
```

)

Arguments

nodes, edges	Lists of graph_node_set and graph_edge_set objects.
graph_name, dataset_name	Optional metadata labels.
validate	If TRUE, run validate_graph() before returning.
validation_overrides	Optional named list of explicit validation exceptions. Currently supported keys: allow_multi_subject_samples If TRUE, the semantic validator does not flag samples linked to multiple subjects, and derive_split_constraints(mode = "subject") silently keeps the first listed subject assignment (recording the ambiguity in metadata$warnings). Defaults to FALSE. When passed to validate_graph() or validate_depgraph(), the override is merged into the graph's existing validation_overrides for the duration of the call only.
x	A dependency_graph.
graph	A dependency_graph.
checks	**Deprecated.** Use levels and severities instead. Retained for backward compatibility with 0.1.0 callers.
error_on_fail	If TRUE, stop when validation errors are found across all detected issues from the selected validation levels, even if those errors are hidden from issues by severities.
levels	Optional validation layers to run.
severities	Optional severities to retain in the returned issues table. This filter does not change whether the graph is considered valid.

Value

For build_dependency_graph(), a dependency_graph. For validate_graph() and validate_depgraph(), a depgraph_validation_report. For as_igraph(), the underlying igraph object.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
subjects <- create_nodes(meta, type = "Subject", id_col = "subject_id")
edges <- create_edges(
  meta,
  "sample_id",
  "subject_id",
  "Sample",
```

```
  "Subject",
  "sample_belongs_to_subject"
)

g <- build_dependency_graph(list(samples, subjects), list(edges))
validate_graph(g)
```

`create_nodes` *Create Canonical Node and Edge Tables*

Description

Build canonical node and edge tables from ordinary metadata frames.

Usage

```
create_nodes(
  data,
  type,
  id_col,
  label_col = NULL,
  attr_cols = NULL,
  prefix = TRUE,
  dedupe = TRUE
)

create_edges(
  data,
  from_col,
  to_col,
  from_type,
  to_type,
  relation,
  attr_cols = NULL,
  allow_missing = FALSE,
  dedupe = TRUE,
  from_prefix = TRUE,
  to_prefix = TRUE
)
```

Arguments

`data`	A `data.frame` containing entity or relationship columns.
`type`, `from_type`, `to_type`	Supported node types such as `"Sample"` or `"Subject"`.
`id_col`	Column containing the source identifier for the node type.
`label_col`	Optional column used for node labels.

`attr_cols` Optional columns stored in the `attrs` list-column.

`prefix` If `TRUE`, prepend typed prefixes such as `sample:` to node identifiers.

`dedupe` If `TRUE`, collapse duplicate identifiers or duplicate edges only when the retained definition is identical.

`from_col`, `to_col` Source and target identifier columns for edge creation.

`relation` Canonical edge type.

`allow_missing` If `TRUE`, drop rows with missing edge endpoints instead of erroring.

`from_prefix`, `to_prefix` Whether to prepend typed prefixes when constructing the edge endpoint identifiers. Defaults preserve the canonical prefixed-ID format.

Details

The package uses typed node identifiers such as `sample:S1` as the canonical graph representation. If you create node sets with `prefix = FALSE`, the corresponding edge endpoints must use matching prefix settings via `from_prefix` and `to_prefix`.

When `dedupe = TRUE`, exact duplicate node or edge definitions are collapsed, but conflicting definitions for the same canonical node identifier or edge relation are rejected with an error.

Value

For `create_nodes()`, a `graph_node_set`. For `create_edges()`, a `graph_edge_set`.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
edges <- create_edges(
  meta,
  from_col = "sample_id",
  to_col = "subject_id",
  from_type = "Sample",
  to_type = "Subject",
  relation = "sample_belongs_to_subject"
)
```

depgraph_validation_report *Validation Report Object for splitGraph Graphs*

Description

depgraph_validation_report is the structured return type produced by validate_graph() and validate_depgraph().

Usage

```
depgraph_validation_report(
  graph_name = NULL,
  issues = NULL,
  metrics = list(),
  metadata = list(),
  valid = NULL,
  errors = NULL,
  warnings = NULL,
  advisories = NULL
)

split_spec(
  sample_data = NULL,
  group_var = "group_id",
  block_vars = character(),
  time_var = NULL,
  ordering_required = FALSE,
  constraint_mode = NULL,
  constraint_strategy = NULL,
  recommended_resampling = NULL,
  metadata = list()
)

split_spec_validation(issues = NULL, metadata = list())

leakage_risk_summary(
  overview = character(),
  diagnostics = NULL,
  validation_summary = list(),
  constraint_summary = list(),
  split_spec_summary = list(),
  metadata = list()
)
```

Arguments

graph_name	Graph label stored on the report.
issues	Canonical issue table. When NULL, an empty skeleton is constructed.
metrics	Named list of graph- and issue-level counts.
metadata	Named list of report metadata.
valid	Optional logical override for the overall validity flag.
errors, warnings, advisories	Optional character vectors of severity-specific messages.
sample_data	Sample-level mapping table carried by a split_spec.
group_var	Name of the grouping column.
block_vars	Optional blocking variable names.
time_var	Optional ordering column name.
ordering_required	Whether ordering is required for downstream evaluation.
constraint_mode, constraint_strategy	Constraint-derivation metadata.
recommended_resampling	Optional recommended resampling routine.
overview	Character vector of human-readable overview lines.
diagnostics	Diagnostics data frame for leakage risks.
validation_summary, constraint_summary, split_spec_summary	Named lists carrying pre-computed summaries.

Details

The report contains:

- graph_name: graph label when available
- valid: whether any error-severity issues were found
- issues: canonical issue table
- summary: counts by level, severity, and code
- metadata: report metadata
- errors, warnings, advisories: backward-compatible message vectors
- metrics: graph and issue counts

The canonical issue table includes the columns: issue_id, level, severity, code, message, node_ids, edge_ids, and details.

Value

An S3 object corresponding to the constructor that was called.

See Also

validate_graph

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2"),
  subject_id = c("P1", "P2")
)
g <- graph_from_metadata(meta)

report <- validate_graph(g)
report$valid
summary(report)
```

`derive_split_constraints`

Derive Split Constraints from Dependency Graphs

Description

Convert dataset dependency structure into deterministic sample-level grouping constraints suitable for leakage-aware evaluation design.

Usage

```
derive_split_constraints(
  graph,
  mode = c("subject", "batch", "study", "time", "composite"),
  samples = NULL,
  strategy = c("strict", "rule_based"),
  via = NULL,
  priority = NULL,
  include_warnings = TRUE
)

grouping_vector(x)
```

Arguments

graph	A dependency_graph.
mode	Constraint derivation mode.
samples	Optional sample identifiers or sample node IDs used to restrict the returned sample_map. All requested samples must resolve successfully.
strategy	Composite grouping strategy. Ignored for non-composite modes.
via	Optional dependency sources used for composite grouping. May be given as lower-case modes such as "subject" or node types such as "Subject".

priority Optional priority order used for strategy = "rule_based".

include_warnings Whether to retain human-readable warnings in the returned metadata.

x A split_constraint.

Details

Constraint derivation rules:

mode = "subject" Groups samples by the target of sample_belongs_to_subject. All samples linked to the same Subject receive the same group_id.

mode = "batch" Groups samples by the target of sample_processed_in_batch. Samples with no batch assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "study" Groups samples by the target of sample_from_study.

mode = "time" Groups samples by the target of sample_collected_at_timepoint. When Timepoint nodes have time_index metadata, that value is used to derive order_rank. If time_index is unavailable, the function attempts to derive ordering from timepoint_precedes edges over the timepoint subgraph.

mode = "composite", strategy = "strict" Projects the selected dependency relations onto a sample graph and assigns one group_id per connected component. This is the transitive-closure interpretation of composite dependency grouping.

mode = "composite", strategy = "rule_based" Evaluates dependency assignments in deterministic priority order and groups each sample by the highest-priority available dependency source. Lower-priority available dependencies are retained in the explanation field.

The returned split_constraint$sample_map always contains sample_id, sample_node_id, group_id, constraint_type, group_label, and explanation. Time-aware constraints also include time_index, timepoint_id, and order_rank when available.

Ambiguous direct assignments are rejected. A sample cannot be assigned to multiple batches, studies, or timepoints when deriving direct split constraints.

Value

derive_split_constraints() returns a split_constraint whose sample_map contains grouping assignments and, for time-aware constraints, ordering metadata. grouping_vector() returns a named character vector of group_id values keyed by sample_id.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2"),
  batch_id   = c("B1", "B2", "B1", "B2")
)
g <- graph_from_metadata(meta)

constraint <- derive_split_constraints(g, mode = "subject")
grouping_vector(constraint)
```

graph_from_metadata *Build a Dependency Graph Directly from a Metadata Table*

Description

One-shot convenience builder that auto-detects canonical columns in a metadata table, creates the corresponding node and edge sets, optionally derives timepoint ordering from time_index, and assembles a dependency_graph. Columns that are absent or entirely missing are silently skipped.

Usage

```
graph_from_metadata(
  meta,
  columns = NULL,
  dataset_name = NULL,
  graph_name = NULL,
  outcome_scope = c("sample", "subject"),
  time_precedence = TRUE,
  validate = TRUE,
  validation_overrides = list()
)
```

Arguments

meta	A data.frame containing one row per sample and optional canonical columns: sample_id (required), subject_id, batch_id, study_id, timepoint_id, time_index, assay_id, featureset_id, outcome_id, or outcome_value.
columns	Optional named character vector passed to ingest_metadata() to rename user columns to canonical names.
dataset_name, graph_name	Optional metadata labels.
outcome_scope	Either "sample" (default) or "subject". Controls whether outcome edges attach to samples or subjects.
time_precedence	If TRUE and time_index is present, derive timepoint_precedes edges from the ordering of time_index.
validate	Forwarded to build_dependency_graph().
validation_overrides	Forwarded to build_dependency_graph().

Value

A validated dependency_graph.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2"),
  batch_id   = c("B1", "B2", "B1", "B2"),
  timepoint_id = c("T1", "T2", "T1", "T2"),
  time_index = c(1, 2, 1, 2),
  outcome_id = c("ctrl", "case", "ctrl", "case")
)

g <- graph_from_metadata(meta, graph_name = "demo")
g
```

---

graph_node_set *Construct Core splitGraph S3 Objects*

Description

Low-level constructors for the core S3 classes used throughout **splitGraph**.

Usage

```
graph_node_set(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)

graph_edge_set(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)

dependency_graph(nodes, edges, graph, metadata = list(), caches = list())

new_depgraph_nodes(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)

new_depgraph_edges(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)
```

```
new_depgraph(nodes, edges, graph = NULL, metadata = list(), caches = list())

graph_query_result(
  query = "",
  params = list(),
  nodes = NULL,
  edges = NULL,
  table = NULL,
  metadata = list()
)

dependency_constraint(
  constraint_id,
  relation_types,
  sample_map,
  transitive = TRUE,
  metadata = list()
)

split_constraint(
  strategy,
  sample_map,
  recommended_downstream_args = list(),
  metadata = list()
)

leakage_constraint(
  issue_type,
  severity,
  affected_samples,
  evidence = NULL,
  recommendation = "",
  metadata = list()
)
```

Arguments

data	A data frame matching the canonical schema for nodes or edges.
schema_version	Schema version string stored on the object.
source	Optional source metadata.
nodes, edges	A graph_node_set and graph_edge_set.
graph	An internal igraph object.
metadata, caches, params, recommended_downstream_args	Named lists with auxiliary metadata.
query	Query label stored on a graph_query_result.
table	Tabular query result payload.

`constraint_id`, `relation_types`, `transitive`
: Fields describing a dependency constraint.

`sample_map`
: Sample-level mapping table for constraints.

`strategy`
: Split strategy identifier.

`issue_type`, `severity`, `affected_samples`, `evidence`, `recommendation`
: Fields describing a leakage warning.

Value

An S3 object corresponding to the constructor that was called.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
subjects <- create_nodes(meta, type = "Subject", id_col = "subject_id")
edges <- create_edges(
  meta,
  from_col = "sample_id",
  to_col = "subject_id",
  from_type = "Sample",
  to_type = "Subject",
  relation = "sample_belongs_to_subject"
)

nodes_set <- graph_node_set(rbind(samples$data, subjects$data))
edges_set <- graph_edge_set(edges$data)
nodes_set
edges_set
```

---

ingest_metadata *Standardize Sample Metadata*

Description

Normalize user-provided metadata into the canonical column contract used by **splitGraph**.

Usage

```
ingest_metadata(data, col_map = NULL, dataset_name = NULL, strict = TRUE)
```

Arguments

data	A sample-level data.frame.
col_map	Optional named character vector mapping canonical names to user-provided columns.
dataset_name	Optional dataset label stored as an attribute on the returned table.
strict	If TRUE, error when required columns are missing.

Value

A standardized data.frame with canonical identifier columns coerced to character.

Examples

```
meta <- ingest_metadata(
  data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
)
```

---

query_node_type *Query Dependency Graph Structure*

Description

Query graph neighborhoods, typed nodes and edges, path structure, projected sample dependency components, and direct shared dependencies within a dependency_graph.

Usage

```
query_node_type(graph, node_types, ids = NULL)

query_edge_type(graph, edge_types, node_ids = NULL)

query_neighbors(
  graph,
  node_ids,
  edge_types = NULL,
  node_types = NULL,
  direction = c("out", "in", "all")
)

query_paths(
  graph,
  from,
  to,
  edge_types = NULL,
  node_types = NULL,
  mode = c("out", "in", "all"),
```

```
  max_length = NULL
)

query_shortest_paths(
  graph,
  from,
  to,
  edge_types = NULL,
  node_types = NULL,
  mode = c("out", "in", "all")
)

detect_dependency_components(
  graph,
 via = c("Subject", "Batch", "Study", "Timepoint", "Assay", "FeatureSet", "Outcome"),
  edge_types = NULL,
  min_size = 1
)

detect_shared_dependencies(
  graph,
  via = c("Subject", "Batch", "Study", "Timepoint"),
  samples = NULL
)
```

Arguments

graph	A dependency_graph.
node_types	Optional node types used to filter node results or allowed path members.
ids	Optional node identifiers used to further restrict query_node_type().
edge_types	Optional edge types used to filter the traversal graph or edge table.
node_ids, from, to	Node identifiers to use as query seeds or endpoints.
direction, mode	Traversal direction.
max_length	Maximum path length (number of edges) for query_paths(). Defaults to a documented finite cap (8) so that igraph::all_simple_paths() cannot blow up on dense graphs. Pass Inf to opt out and search exhaustively; pass any non-negative integer for an explicit cap. Negative values and non-numeric inputs are rejected.
via	Dependency node types used for sample-level dependency detection.
min_size	Minimum component size retained by detect_dependency_components().
samples	Optional sample identifiers or sample node IDs used to restrict direct shared-dependency detection. All requested samples must resolve successfully.

Details

When a samples subset is supplied, partial matching is not allowed: unknown sample identifiers raise an error rather than being silently dropped.

Value

Each function returns a graph_query_result. Use as.data.frame() to obtain the tidy result table.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3"),
  subject_id = c("P1", "P1", "P2"),
  batch_id   = c("B1", "B2", "B1")
)
g <- graph_from_metadata(meta)

query_node_type(g, "Sample")
query_neighbors(g, "sample:S1", direction = "out")
detect_shared_dependencies(g, via = "Subject")
```

---

write_dependency_graph

Serialize a Dependency Graph to JSON

Description

Write a dependency_graph to a JSON file and read it back. The on-disk format is intentionally simple and stable: it captures the canonical node table, the canonical edge table (each with their list-column of attributes), the graph metadata (including validation_overrides), and the data-model schema_version. The internal igraph representation is not stored; it is rebuilt on read via dependency_graph().

Usage

```
write_dependency_graph(graph, path, pretty = TRUE)

read_dependency_graph(path)
```

Arguments

graph	A dependency_graph produced by build_dependency_graph() or graph_from_metadata().
path	Path to write to or read from.
pretty	If TRUE (default), the JSON is indented for human inspection. Set FALSE for a compact representation.

Details

This makes split_spec/dependency_graph objects portable across R sessions, and across language boundaries (any consumer that can read JSON can interpret the format).

Value

`write_dependency_graph()` invisibly returns `path`. `read_dependency_graph()` returns a validated `dependency_graph`.

JSON format

```
{
  "splitGraph_object": "dependency_graph",
  "schema_version": "0.1.0",
  "metadata": {
    "graph_name": "...",
    "dataset_name": "...",
    "created_at": "2026-04-29T10:11:12.000000+0000",
    "schema_version": "0.1.0",
    "validation_overrides": { ... }
  },
  "nodes": [
    { "node_id": "sample:S1", "node_type": "Sample",
      "node_key": "S1", "label": "S1", "attrs": { ... } },
    ...
  ],
  "edges": [
    { "edge_id": "sample_belongs_to_subject:1",
      "from": "sample:S1", "to": "subject:P1",
      "edge_type": "sample_belongs_to_subject", "attrs": { ... } },
    ...
  ]
}
```

Reading a file whose `schema_version` does not match the installed package emits a warning but still loads.

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(
    sample_id  = c("S1", "S2"),
    subject_id = c("P1", "P2")
  )
  g <- graph_from_metadata(meta, graph_name = "demo")

  tmp <- tempfile(fileext = ".json")
  write_dependency_graph(g, tmp)
  g2 <- read_dependency_graph(tmp)
  identical(g$nodes$data$node_id, g2$nodes$data$node_id)
  unlink(tmp)
}
```

`write_split_spec` *Serialize a Split Specification to JSON*

Description

Write a `split_spec` to a JSON file and read it back. The on-disk format captures the canonical sample-level table (`sample_data`) plus all spec-level fields needed by a downstream resampling adapter (`group_var`, `block_vars`, `time_var`, `ordering_required`, `constraint_mode`, `constraint_strategy`, `recommended_resampling`) and the spec metadata.

Usage

```
write_split_spec(spec, path, pretty = TRUE)

read_split_spec(path)
```

Arguments

`spec`	A `split_spec` produced by `as_split_spec()`.
`path`	Path to write to or read from.
`pretty`	If `TRUE` (default), the JSON is indented.

Details

`NA` values in `sample_data` are written as JSON `null` and read back as `NA`.

Value

`write_split_spec()` invisibly returns `path`. `read_split_spec()` returns a `split_spec`.

JSON format

```
{
  "splitGraph_object": "split_spec",
  "schema_version": "0.1.0",
  "group_var": "group_id",
  "block_vars": ["batch_group", "study_group"],
  "time_var": "order_rank",
  "ordering_required": false,
  "constraint_mode": "subject",
  "constraint_strategy": "subject",
  "recommended_resampling": "grouped_cv",
  "metadata": { ... },
  "sample_data": [
    { "sample_id": "S1", "group_id": "subject:P1", ... },
    ...
  ]
}
```

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(
    sample_id  = c("S1", "S2"),
    subject_id = c("P1", "P2")
  )
  g <- graph_from_metadata(meta)
  constraint <- derive_split_constraints(g, mode = "subject")
  spec <- as_split_spec(constraint, graph = g)

  tmp <- tempfile(fileext = ".json")
  write_split_spec(spec, tmp)
  spec2 <- read_split_spec(tmp)
  identical(spec$sample_data$group_id, spec2$sample_data$group_id)
  unlink(tmp)
}
```

Index