Package: OdysseusCharacterizationModule (via r-universe)

May 23, 2026

Title Handy and Minimalistic Common Data Model Characterization

Version 0.0.1

Description Extracts covariates from Observational Medical Outcomes Partnership (OMOP) Common Data Model (CDM) domains using an R-only pipeline. Supports configurable temporal windows, domain-specific covariates for drug exposure, drug era (including Anatomical Therapeutic Chemical (ATC) groupings), condition occurrence, condition era, concept sets and cohorts. Methods are based on the Observational Health Data Sciences and Informatics (OHDSI) framework described in Hripcsak et al. (2015) <doi:10.1038/sdata.2015.35> and ``The Book of OHDSI'' OHDSI (2019, ISBN:978-1-7923-0589-8).

License Apache License (>= 2)

Depends R (>= 4.1.0)

Imports checkmate, DatabaseConnector, jsonlite, SqlRender, stats

Suggests Andromeda, Eunomia, FeatureExtraction, knitr, rmarkdown, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Author Alexander Alexeyuk [aut, cre]

Maintainer Alexander Alexeyuk <alexanderAlexeyuk@gmail.com>

Config/pak/sysreqs make default-jdk libicu-dev libx11-dev

Repository https://a1exanderalexeyuk.r-universe.dev

Date/Publication 2026-04-22 14:13:08 UTC

RemoteUrl https://github.com/cran/OdysseusCharacterizationModule

RemoteRef HEAD

RemoteSha 75e39ce275e2b5c4e6e51aa09e951e7f390aea12

Contents

`buildConceptSetQueries`

Build SQL for Multiple Concept-Set Expressions

Description

A convenience wrapper around `buildConceptSetQuery` that resolves every element of a named list of concept-set expressions to SQL in a single call.

Usage

```
buildConceptSetQueries(
  conceptSetList,
  conceptSetNames = names(conceptSetList),
  vocabularyDatabaseSchema = "@vocabulary_database_schema"
)
```

Arguments

`conceptSetList`	A named list of concept-set expressions. Each element must conform to the format accepted by `buildConceptSetQuery` (an R list with an `items` element, or a JSON character string). The names of the list are used as the `conceptSetName` for each query.
`conceptSetNames`	Character vector of concept-set labels, one per element of `conceptSetList`. Defaults to `names(conceptSetList)`. These values are embedded in the output SQL as the `cs_name` column.

vocabularyDatabaseSchema
: Character string. The fully qualified schema containing the OMOP vocabulary tables. Passed through to buildConceptSetQuery. Defaults to "@vocabulary_database_schema" for deferred rendering with render.

Details

The function iterates over conceptSetList and conceptSetNames in parallel using map2, calling buildConceptSetQuery for each pair.

Value

A named list of character strings, each containing the SQL query for the corresponding concept set. Names match those of conceptSetList. Elements whose concept sets resolve to no included concepts are returned as "" (empty strings).

See Also

buildConceptSetQuery for the single-expression resolver and full documentation of the expression format.

Examples

```
csList <- list(
  diabetes = list(items = list(
    list(concept = list(CONCEPT_ID = 201820),
         includeDescendants = TRUE)
  )),
  hypertension = list(items = list(
    list(concept = list(CONCEPT_ID = 316866),
         includeDescendants = TRUE)
  ))
)

queries <- buildConceptSetQueries(
  csList,
  vocabularyDatabaseSchema = "cdm_v5"
)

# Each element is a SQL string
cat(queries$diabetes)
cat(queries$hypertension)
```

`buildConceptSetQuery` *Build SQL to Resolve a CIRCE Concept-Set Expression*

Description

Translates a single CIRCE-format concept-set expression into a stand-alone SQL query that resolves to the set of `concept_id` values implied by the expression. The function handles the four combination of per-item flags (`includeDescendants`, `includeMapped`) as well as concept exclusion (`isExcluded`), and does *not* require Java or the **CirceR** package.

Usage

```
buildConceptSetQuery(
  conceptSetExpression,
  conceptSetName = "plug",
  vocabularyDatabaseSchema = "@vocabulary_database_schema"
)
```

Arguments

`conceptSetExpression`
: A concept-set expression in one of two forms:

  List An R list with an `items` element. Each item is itself a list containing at minimum a `concept` list with a numeric `CONCEPT_ID`. Optional per-item logical flags: `includeDescendants`, `includeMapped`, `isExcluded`.

  JSON string A single character string of valid JSON that parses to the list structure described above (via `fromJSON`).

  See **Details** for the full item specification.

`conceptSetName`
: Character string. A label embedded in the output SQL as the `cs_name` column value, useful for distinguishing results when multiple concept sets are unioned together. Defaults to `"plug"`.

`vocabularyDatabaseSchema`
: Character string. The fully qualified schema containing the OMOP vocabulary tables (`CONCEPT`, `CONCEPT_ANCESTOR`, `CONCEPT_RELATIONSHIP`). Defaults to `"@vocabulary_database_schema"` so that the returned SQL can be further parameterised with `render`.

Details

Item specification: Each element of `conceptSetExpression$items` must be a list with the following structure:

`concept` Required list. Must contain `CONCEPT_ID` — a single, non-`NA`, whole-number numeric value.

`includeDescendants` Optional logical scalar. When `TRUE`, all descendant concepts (via `CONCEPT_ANCESTOR`) are included. Defaults to `FALSE` when absent.

includeMapped Optional logical scalar. When TRUE, concepts reached through "Maps to" relationships (via CONCEPT_RELATIONSHIP) are included. Defaults to FALSE when absent.

isExcluded Optional logical scalar. When TRUE, the resolved concepts for this item are *removed* from the final set via a LEFT JOIN . . . WHERE . . . IS NULL anti-join pattern. Defaults to FALSE when absent.

Resolution logic: Items are first partitioned into *included* and *excluded* groups based on isExcluded. Within each group, items are further classified into four categories by their flag combinations:

1. **Plain** — neither descendants nor mapped.
2. **Descendants only** — includeDescendants = TRUE.
3. **Mapped only** — includeMapped = TRUE.
4. **Descendants and mapped** — both flags TRUE.

A UNION of the appropriate SQL blocks is built for each group. If excluded items exist, the excluded set is anti-joined against the included set.

SQL rendering: The conceptSetName value is injected into the SQL via render using the @cs_name token. The vocabularyDatabaseSchema is interpolated directly via sprintf.

Value

A single character string containing a SQL SELECT statement that produces two columns: cs_name (the concept-set label) and concept_id. Returns "" (an empty string) when items is an empty list or when all items are excluded and no included items remain.

Input validation

The function performs extensive validation of all inputs and stops with an informative error message if:

- vocabularyDatabaseSchema is not a single non-empty character string.
- conceptSetExpression is neither a list nor a valid JSON string.
- The items element is missing or is not a list.
- Any item lacks a concept element or a valid CONCEPT_ID.
- Any optional logical flag is present but not a single logical value.

A warning is issued when a logical flag is NA (treated as FALSE).

See Also

buildConceptSetQueries for batch resolution of multiple concept sets, render for SQL parameterisation, fromJSON for JSON parsing.

Examples

```
# A concept set with descendants
expr <- list(items = list(
  list(concept = list(CONCEPT_ID = 201820),
       includeDescendants = TRUE),
```

```
  list(concept = list(CONCEPT_ID = 433962))
))

sql <- buildConceptSetQuery(
  conceptSetExpression    = expr,
  conceptSetName          = "diabetes",
  vocabularyDatabaseSchema = "cdm_v5"
)
cat(sql)

# From a JSON string
json <- '{"items":[{"concept":{"CONCEPT_ID":316866},"includeDescendants":true}]}'
sql2 <- buildConceptSetQuery(json, conceptSetName = "hypertension")
cat(sql2)

# Exclusion example
expr_excl <- list(items = list(
  list(concept = list(CONCEPT_ID = 201820),
       includeDescendants = TRUE),
  list(concept = list(CONCEPT_ID = 201254),
       isExcluded = TRUE)
))
sql3 <- buildConceptSetQuery(expr_excl, conceptSetName = "diabetes_refined")
cat(sql3)
```

`createConceptSetTempTable` *Materialise Concept-Set Queries into a Temporary Table*

Description

Takes the output of `buildConceptSetQueries` — a named list of SQL `SELECT` statements — unions them together, and writes the result into a single temporary database table with columns `cs_name` and `concept_id`.

Usage

```
createConceptSetTempTable(
  connection,
  csQueries,
  vocabularyDatabaseSchema,
  tempTableName = "#concept_sets_c",
  tempEmulationSchema = NULL
)
```

Arguments

connection	A connect database connection object.
csQueries	A named list of SQL query strings as returned by buildConceptSetQueries. Each element must be a single-string SELECT that produces columns cs_name (character) and concept_id (integer). Empty strings ("") are silently dropped.
vocabularyDatabaseSchema	Character string. The fully qualified schema containing the OMOP vocabulary tables. This value is substituted for the @vocabulary_database_schema token present in the queries via render.
tempTableName	Character string. Name of the temporary table to create. Defaults to "#concept_sets". Must start with "#" on SQL Server; on other platforms the leading "#" is handled by renderTranslateExecuteSql.
tempEmulationSchema	Character string or NULL (default). Schema used for temporary-table emulation on platforms that do not natively support temp tables (e.g., Oracle, BigQuery). Passed through to renderTranslateExecuteSql.

Details

The function performs the following steps:

1. Filters out any empty query strings from csQueries.
2. Unions all remaining queries into a single SELECT statement.
3. Wraps the union in a SELECT * INTO <tempTableName> statement.
4. Renders the @vocabulary_database_schema parameter and translates the SQL for the target DBMS.
5. Executes the SQL via renderTranslateExecuteSql.
6. Creates a non-unique index on (cs_name, concept_id) for efficient downstream joins.

If all queries are empty after filtering, the function stops with an informative error.

Value

Invisibly returns the name of the created temporary table (tempTableName). Called primarily for its side effect of creating the table in the database.

Table schema

The created temporary table contains two columns:

cs_name Character. The concept-set label (derived from the names of csQueries).

concept_id Integer. A resolved OMOP standard concept identifier.

See Also

buildConceptSetQueries for generating the input, buildConceptSetQuery for the single-expression builder, renderTranslateExecuteSql.

Examples

```
## Not run:
library(DatabaseConnector)

conn <- connect(dbms = "postgresql",
                server = "localhost/ohdsi",
                user = "user",
                password = "pass")

csList <- list(
  diabetes = list(items = list(
    list(concept = list(CONCEPT_ID = 201820),
         includeDescendants = TRUE),
    list(concept = list(CONCEPT_ID = 433962))
  )),
  hypertension = list(items = list(
    list(concept = list(CONCEPT_ID = 316866),
         includeDescendants = TRUE)
  ))
)

csQueries <- buildConceptSetQueries(csList,
                                    vocabularyDatabaseSchema = "cdm_v5")

createConceptSetTempTable(
  connection               = conn,
  csQueries                = csQueries,
  vocabularyDatabaseSchema = "cdm_v5",
  tempTableName            = "#concept_sets"
)

# Query the resulting table
result <- querySql(conn, "SELECT * FROM #concept_sets;")
head(result)

disconnect(conn)

## End(Not run)
```

`createOcmCovariateSettings` *Create Custom Covariate Settings for FeatureExtraction*

Description

Creates a `covariateSettings` object that can be passed directly to `FeatureExtraction::getDbCovariateData()` as a custom covariate builder. The settings specify which OdysseusCharacterizationModule analyses to run, including time windows, base features, cohort features, and concept-set features.

Usage

```
createOcmCovariateSettings(
 analysisWindows = defineAnalysisWindows(startDays = c(-365, -180, -30, 0, 1, 30, 180,
    365), endDays = c(-1, -1, -1, 0, 30, 180, 365, 700)),
 useBaseFeatures = list(drug_exposure = list(include = FALSE, atc = FALSE, atcLevels =
  c(1L, 2L, 3L, 4L, 5L)), condition_occurrence = list(include = FALSE, type = "start"),
  condition_era = list(include = FALSE, type = "start"), drug_era = list(include =
   FALSE, type = "start", atc = FALSE, atcLevels = c(5L)), procedure_occurrence =
   list(include = FALSE), observation = list(include = FALSE), device_exposure =
  list(include = FALSE), visit_occurrence = list(include = TRUE, type = "start"),
    measurement = list(include = FALSE)),
 useCohortFeatures = list(include = FALSE, type = "start", cohortIds = NULL, cohortNames
    = NULL, cohortTable = NULL, covariateSchema = NULL),
 useConceptSetFeatures = list(conceptSets = NULL, include = FALSE, type = "binary")
)
```

Arguments

analysisWindows	An analysisWindows object as returned by defineAnalysisWindows.
useBaseFeatures	Named list of domain configurations, same structure as in planAnalysis.
useCohortFeatures	List specifying cohort-based feature extraction, same structure as in planAnalysis.
useConceptSetFeatures	List specifying concept-set-based feature extraction, same structure as in planAnalysis.

Value

An S3 object of class covariateSettings with an attribute fun set to "getDbOcmCovariateData".
This object can be passed as the covariateSettings argument to FeatureExtraction::getDbCovariateData(),
or used standalone by calling getDbOcmCovariateData directly.

See Also

getDbOcmCovariateData for the corresponding builder function, planAnalysis for parameter documentation.

Examples

```
settings <- createOcmCovariateSettings(
  analysisWindows = defineAnalysisWindows(
    startDays = c(-365, -30),
    endDays   = c(-1, -1)
  ),
  useBaseFeatures = list(
    drug_exposure = list(include = TRUE, atc = FALSE),
    condition_occurrence = list(include = TRUE, type = "start"),
    condition_era = list(include = FALSE),
```

```
    drug_era = list(include = FALSE),
    procedure_occurrence = list(include = FALSE),
    observation = list(include = FALSE),
    device_exposure = list(include = FALSE),
    visit_occurrence = list(include = FALSE),
    measurement = list(include = FALSE)
  )
)
```

`defineAnalysisWindows` *Define analysis windows*

Description

Define analysis windows

Usage

```
defineAnalysisWindows(
  startDays = c(-365, -180, -30, 0, 1, 30, 180, 365),
  endDays = c(-1, -1, -1, 0, 30, 180, 365, 700),
  windowNames = NULL
)
```

Arguments

`startDays`	Integer vector of start days relative to cohort start date.
`endDays`	Integer vector of end days relative to cohort start date.
`windowNames`	Optional character vector of names for each window.

Value

A list of `analysisWindow` objects.

`executeSpec` *Execute a Single Analysis Specification*

Description

Renders the SQL for a `singleNodeSpec` object, translates it to the target DBMS dialect, executes it against an open `connect` connection, and returns the result set as a `data.frame`.

Usage

```
executeSpec(connection, spec, targetDialect = NULL, tempEmulationSchema = NULL)
```

Arguments

connection	A connect database connection object.
spec	A singleNodeSpec object as returned by singleNodeSetting.
targetDialect	Character string. The target DBMS dialect for SQL translation (e.g., "postgresql", "redshift", "sql server", "oracle", "spark"). Defaults to the dialect stored on the connection via dbms.
tempEmulationSchema	Character string or NULL. Schema used for temporary-table emulation on platforms that do not natively support temp tables (e.g., Oracle, BigQuery). Passed through to translate and executeSql / querySql.

Details

The function performs the following steps:

1. Renders all @placeholder tokens via renderSpecSql.
2. Splits the rendered SQL into individual statements.
3. For aggregated specs, executes the INSERT statement(s) via executeSql, then runs the final SELECT (aggregation) via querySql.
4. For non-aggregated specs, executes all statements via executeSql, then queries the temp table for rows matching the current analysis_id.

Value

For aggregated specs: a data.frame with columns cohort_definition_id, covariate_id, covariate_name, concept_id, analysis_id, sum_value, mean_value. For non-aggregated specs: a data.frame with the raw patient-level rows inserted into #domain_raw_results.

See Also

executeSpecs for batch execution of multiple specs, renderSpecSql for SQL rendering, singleNodeSetting for creating specs.

Examples

```
## Not run:
conn <- DatabaseConnector::connect(dbms = "postgresql",
                                   server = "localhost/ohdsi",
                                   user = "user", password = "pass")

result <- executeSpec(conn, specs[[1]])
head(result)

DatabaseConnector::disconnect(conn)

## End(Not run)
```

executeSpecs *Execute Multiple Analysis Specifications*

Description

Iterates over a `singleNodeSettingList` and executes each spec sequentially against the database. All specs share the same `#domain_raw_results` temp table across executions.

Usage

```
executeSpecs(
  connection,
  specs,
  targetDialect = NULL,
  tempEmulationSchema = NULL,
  cleanTempTables = FALSE,
  stopOnError = TRUE
)
```

Arguments

`connection`	A `connect` database connection object.
`specs`	A `singleNodeSettingList` object as returned by `singleNodeSetting`.
`targetDialect`	Character string or `NULL`. Target DBMS dialect. Defaults to the dialect on the connection.
`tempEmulationSchema`	Character string or `NULL`. Schema for temp-table emulation.
`cleanTempTables`	Logical. If `TRUE`, attempts to drop the
`stopOnError`	Logical. If `TRUE` (default), execution stops on the first error. If `FALSE`, errors are captured and reported in the output, and execution continues with the next spec.

Details

The function:

1. Logs a summary header.
2. Calls `executeSpec` for each spec in order.
3. Drops the shared `#domain_raw_results` temp table after all specs have been executed (cleanup).
4. Returns all results as a named list.

Value

A named list of `data.frame` objects, one per spec. Names are the analysis IDs (as character strings). When `stopOnError = FALSE`, failed specs produce a `data.frame` with zero rows and an `"error"` attribute containing the error message.

See Also

executeSpec for single-spec execution, singleNodeSetting for creating specs.

Examples

```
## Not run:
conn <- DatabaseConnector::connect(dbms = "postgresql",
                                   server = "localhost/ohdsi",
                                   user = "user", password = "pass")

results <- executeSpecs(conn, specs)
lapply(results, head)

DatabaseConnector::disconnect(conn)

## End(Not run)
```

---

getDbOcmCovariateData *Get Custom Covariate Data from the Database*

Description

Builder function that implements the FeatureExtraction custom covariate builder interface. It executes the OdysseusCharacterizationModule pipeline and returns a `CovariateData` object (an Andromeda object with `covariates`, `covariateRef`, and `analysisRef` tables).

Usage

```
getDbOcmCovariateData(
  connection,
  tempEmulationSchema = NULL,
  cdmDatabaseSchema,
  cdmVersion = "5",
  cohortTable = "#cohort_person",
  cohortIds = c(-1),
  rowIdField = "subject_id",
  covariateSettings,
  aggregated = FALSE,
  minCharacterizationMean = 0,
  ...
)
```

Arguments

`connection`	A connect database connection object.

`tempEmulationSchema`
: Character or `NULL`. Schema for temp-table emulation on platforms like Oracle or BigQuery.

`cdmDatabaseSchema`
: Character. Schema containing the OMOP CDM tables.

`cdmVersion`
: Character. OMOP CDM version (`"5"`).

`cohortTable`
: Character. Fully qualified name of the cohort table (e.g., `"results.cohort"` or `"#cohort_person"`).

`cohortIds`
: Integer vector. Cohort definition IDs to extract covariates for. Use `c(-1)` for all cohorts.

`rowIdField`
: Character. Column name in the cohort table used as the row identifier. Typically `"subject_id"` or a custom row key.

`covariateSettings`
: An object created by `createOcmCovariateSettings`.

`aggregated`
: Logical. Currently only `FALSE` (person-level) is supported. If `TRUE` an error is raised.

`minCharacterizationMean`
: Numeric. Minimum mean value for filtering (currently unused; present for interface compatibility).

`...`
: Additional arguments passed by `FeatureExtraction::getDbCovariateData()` (e.g., `targetCovariateTable`, `targetCovariateRefTable`). Silently ignored.

Details

This function is normally not called directly. Instead, create a settings object with `createOcmCovariateSettings` and pass it to `FeatureExtraction::getDbCovariateData()`.

Value

A `CovariateData` object (Andromeda) with:

`covariates` Sparse table: `rowId`, `covariateId`, `covariateValue`.

`covariateRef` Reference: `covariateId`, `covariateName`, `analysisId`, `conceptId`.

`analysisRef` Reference: `analysisId`, `analysisName`, `domainId`, `startDay`, `endDay`, `isBinary`, `missingMeansZero`.

See Also

`createOcmCovariateSettings` for creating the settings object.

planAnalysis *Plan Characterization Analyses*

Description

Creates a comprehensive characterization analysis plan for patient-level feature extraction from an OMOP Common Data Model (CDM) database. The plan defines time windows, base clinical features, cohort-based features, and concept-set-based features to be extracted relative to a target cohort's index date.

Creates a comprehensive characterization analysis plan for patient-level feature extraction from an OMOP Common Data Model (CDM) database. The plan defines time windows, base clinical features, cohort-based features, and concept-set-based features to be extracted relative to a target cohort's index date.

Usage

```
planAnalysis(
 analysisWindows = defineAnalysisWindows(startDays = c(-365, -180, -30, 0, 1, 30, 180,
    365), endDays = c(-1, -1, -1, 0, 30, 180, 365, 700)),
 useBaseFeatures = list(drug_exposure = list(include = FALSE, atc = FALSE, atcLevels =
  c(1L, 2L, 3L, 4L, 5L)), condition_occurrence = list(include = FALSE, type = "start"),
  condition_era = list(include = FALSE, type = "start"), drug_era = list(include =
   FALSE, type = "start", atc = FALSE, atcLevels = c(5L)), procedure_occurrence =
   list(include = FALSE), observation = list(include = FALSE), device_exposure =
  list(include = FALSE), visit_occurrence = list(include = FALSE, type = "start"),
    measurement = list(include = FALSE)),
 useCohortFeatures = list(include = FALSE, type = "start", cohortIds = NULL, cohortNames
    = NULL, cohortTable = NULL, covariateSchema = NULL),
 useConceptSetFeatures = list(conceptSets = NULL, include = FALSE, type = "binary")
)

planAnalysis(
 analysisWindows = defineAnalysisWindows(startDays = c(-365, -180, -30, 0, 1, 30, 180,
    365), endDays = c(-1, -1, -1, 0, 30, 180, 365, 700)),
 useBaseFeatures = list(drug_exposure = list(include = FALSE, atc = FALSE, atcLevels =
  c(1L, 2L, 3L, 4L, 5L)), condition_occurrence = list(include = FALSE, type = "start"),
  condition_era = list(include = FALSE, type = "start"), drug_era = list(include =
   FALSE, type = "start", atc = FALSE, atcLevels = c(5L)), procedure_occurrence =
   list(include = FALSE), observation = list(include = FALSE), device_exposure =
  list(include = FALSE), visit_occurrence = list(include = FALSE, type = "start"),
    measurement = list(include = FALSE)),
 useCohortFeatures = list(include = FALSE, type = "start", cohortIds = NULL, cohortNames
    = NULL, cohortTable = NULL, covariateSchema = NULL),
 useConceptSetFeatures = list(conceptSets = NULL, include = FALSE, type = "binary")
)
```

Arguments

`analysisWindows`
: An object of class `analysisWindows` as returned by `defineAnalysisWindows`, defining one or more time windows relative to the cohort index date. Each window is defined by a `startDay` and `endDay` (inclusive). Negative values indicate days before the index date; positive values indicate days after.

`useBaseFeatures`
: A named list of domain configurations. Each element name must correspond to a supported OMOP CDM table (e.g., `"drug_exposure"`, `"condition_occurrence"`). Each element is a list with the following components:

  `include` Logical scalar. If `TRUE`, features are extracted from this domain. Default varies by domain (see Usage).

  `type` Character scalar (`"start"` or `"overlap"`). Determines how records are matched to time windows. `"start"` uses the record start date; `"overlap"` checks whether the record's date range overlaps the window. Only applicable to era tables and `visit_occurrence`. Default: `"start"`.

  `atc` Logical scalar. If `TRUE`, drug concepts are rolled up to ATC classification levels. Only applicable to `drug_exposure` and `drug_era`. Default: `TRUE`.

  `atcLevels` Integer vector. ATC levels to include, from `1L` (anatomical main group) to `5L` (chemical substance). Only applicable when `atc = TRUE`. Default: `c(1L, 2L, 3L, 4L, 5L)`.

`useCohortFeatures`
: A list specifying cohort-based feature extraction with the following components:

  `include` Logical scalar. If `TRUE`, cohort-based features are included. Default: `TRUE`.

  `type` Character scalar (`"start"` or `"overlap"`). Determines how cohort membership is matched to time windows. Default: `"start"`.

  `cohortIds` Integer vector or `NULL`. Cohort definition IDs to use as features. Required when `include = TRUE`.

  `cohortNames` Character vector or `NULL`. Human-readable names corresponding to `cohortIds`. Must be the same length as `cohortIds`.

  `cohortTable` Character scalar or `NULL`. Name of the cohort table containing the feature cohorts. Required when `include = TRUE`.

  `covariateSchema` Character scalar or `NULL`. Database schema where `cohortTable` resides. Required when `include = TRUE`.

`useConceptSetFeatures`
: A list specifying concept-set-based feature extraction with the following components:

  `conceptSets` A named list of concept set definitions. Each element is a list with:

  `items` A list of concept items. Each item is a list with:

  `concept` A list containing at minimum `CONCEPT_ID` (integer).

  `includeDescendants` Logical scalar. If `TRUE`, all descendant concepts are included via the `concept_ancestor` table. Default: `FALSE`.

`tables` Character vector. OMOP CDM table names to search for matching concepts (e.g., `"condition_occurrence"`, `"drug_exposure"`).

`include` Logical scalar. If `TRUE`, concept-set-based features are included. Default: `TRUE`.

`type` Character scalar (`"binary"` or `"counts"`). `"binary"` produces 0/1 indicators; `"counts"` produces record counts within each time window. Default: `"binary"`.

Details

This function assembles a `characterizationSettings` object that serves as a blueprint for downstream feature extraction. It supports three complementary feature extraction strategies:

Base Features:

Standard OMOP CDM domain tables are used to construct binary or count-based covariates. Supported domains include:

- **drug_exposure / drug_era**: Drug concepts, optionally rolled up to ATC hierarchy levels 1–5.
- **condition_occurrence / condition_era**: Condition concepts with configurable temporal logic.
- **procedure_occurrence**: Procedure concepts.
- **observation**: Observation concepts.
- **device_exposure**: Device concepts.
- **visit_occurrence**: Visit concepts.
- **measurement**: Measurement concepts.

Each domain accepts:

`include` Logical. Whether to extract features from this domain.

`type` Character. Temporal logic: `"start"` uses the record start date; `"overlap"` uses era-style overlap with the time window. Applicable to era tables and visit_occurrence.

`atc` Logical. Whether to roll up drug concepts to ATC hierarchy levels. Applicable to drug_exposure and drug_era only.

`atcLevels` Integer vector. ATC hierarchy levels to include (1–5). Applicable when `atc = TRUE`.

Cohort Features:

Pre-defined cohorts (stored in a cohort table) are used as binary covariates, indicating whether a patient belongs to each specified cohort within each time window.

Concept Set Features:

User-defined concept sets (analogous to ATLAS concept sets) are used to create targeted covariates. Each concept set specifies one or more concepts (optionally including descendants) and the CDM tables to search. Output type can be `"binary"` (presence/absence) or `"counts"` (frequency).

This function assembles a `characterizationSettings` object that serves as a blueprint for downstream feature extraction. It supports three complementary feature extraction strategies:

Base Features:

Standard OMOP CDM domain tables are used to construct binary or count-based covariates. Supported domains include:

- **drug_exposure / drug_era**: Drug concepts, optionally rolled up to ATC hierarchy levels 1–5.
- **condition_occurrence / condition_era**: Condition concepts with configurable temporal logic.
- **procedure_occurrence**: Procedure concepts.
- **observation**: Observation concepts.
- **device_exposure**: Device concepts.
- **visit_occurrence**: Visit concepts.
- **measurement**: Measurement concepts.

Each domain accepts:

`include` Logical. Whether to extract features from this domain.

`type` Character. Temporal logic: `"start"` uses the record start date; `"overlap"` uses era-style overlap with the time window. Applicable to era tables and visit_occurrence.

`atc` Logical. Whether to roll up drug concepts to ATC hierarchy levels. Applicable to drug_exposure and drug_era only.

`atcLevels` Integer vector. ATC hierarchy levels to include (1–5). Applicable when `atc = TRUE`.

Cohort Features:

Pre-defined cohorts (stored in a cohort table) are used as binary covariates, indicating whether a patient belongs to each specified cohort within each time window.

Concept Set Features:

User-defined concept sets (analogous to ATLAS concept sets) are used to create targeted covariates. Each concept set specifies one or more concepts (optionally including descendants) and the CDM tables to search. Output type can be `"binary"` (presence/absence) or `"counts"` (frequency).

Value

An S3 object of class `characterizationSettings` containing:

`analysisWindows` The validated analysis windows.

`useBaseFeatures` The validated base feature configuration.

`useCohortFeatures` The validated cohort feature configuration.

`useConceptSetFeatures` The validated concept set feature configuration.

An S3 object of class `characterizationSettings` containing:

`analysisWindows` The validated analysis windows.

`useBaseFeatures` The validated base feature configuration.

`useCohortFeatures` The validated cohort feature configuration.

`useConceptSetFeatures` The validated concept set feature configuration.

See Also

`defineAnalysisWindows` for creating time window definitions.

`defineAnalysisWindows` for creating time window definitions.

Examples

```
# Minimal plan with default settings
plan <- planAnalysis()

# Custom plan: conditions and drugs only, two time windows
plan <- planAnalysis(
  analysisWindows = defineAnalysisWindows(
    startDays = c(-365, 1),
    endDays   = c(-1, 365)
  ),
  useBaseFeatures = list(
    drug_exposure = list(
      include = TRUE,
      atc = TRUE,
      atcLevels = c(3L, 5L)
    ),
    condition_occurrence = list(
      include = TRUE,
      type = "start"
    ),
    condition_era       = list(include = FALSE),
    drug_era            = list(include = FALSE),
    procedure_occurrence = list(include = FALSE),
    observation         = list(include = FALSE),
    device_exposure     = list(include = FALSE),
    visit_occurrence    = list(include = FALSE),
    measurement         = list(include = FALSE)
  ),
  useCohortFeatures = list(include = FALSE),
  useConceptSetFeatures = list(include = FALSE)
)

# Plan with cohort features
plan <- planAnalysis(
  useCohortFeatures = list(
    include = TRUE,
    type = "start",
    cohortIds = c(101L, 102L, 103L),
    cohortNames = c("T2DM", "Hypertension", "CKD"),
    cohortTable = "my_cohort_table",
    covariateSchema = "results_schema"
  )
)

# Plan with custom concept sets
plan <- planAnalysis(
  useConceptSetFeatures = list(
    conceptSets = list(
      diabetes = list(
        items = list(
          list(concept = list(CONCEPT_ID = 201820L),
               includeDescendants = TRUE)
```

```
        ),
        tables = c("condition_occurrence")
      )
    ),
    include = TRUE,
    type = "counts"
  )
)

# Minimal plan with default settings
plan <- planAnalysis()

# Custom plan: conditions and drugs only, two time windows
plan <- planAnalysis(
  analysisWindows = defineAnalysisWindows(
    startDays = c(-365, 1),
    endDays   = c(-1, 365)
  ),
  useBaseFeatures = list(
    drug_exposure = list(
      include = TRUE,
      atc = TRUE,
      atcLevels = c(3L, 5L)
    ),
    condition_occurrence = list(
      include = TRUE,
      type = "start"
    ),
    condition_era       = list(include = FALSE),
    drug_era            = list(include = FALSE),
    procedure_occurrence = list(include = FALSE),
    observation         = list(include = FALSE),
    device_exposure     = list(include = FALSE),
    visit_occurrence    = list(include = FALSE),
    measurement         = list(include = FALSE)
  ),
  useCohortFeatures = list(include = FALSE),
  useConceptSetFeatures = list(include = FALSE)
)

# Plan with cohort features
plan <- planAnalysis(
  useCohortFeatures = list(
    include = TRUE,
    type = "start",
    cohortIds = c(101L, 102L, 103L),
    cohortNames = c("T2DM", "Hypertension", "CKD"),
    cohortTable = "my_cohort_table",
    covariateSchema = "results_schema"
  )
)

# Plan with custom concept sets
```

```
plan <- planAnalysis(
  useConceptSetFeatures = list(
    conceptSets = list(
      diabetes = list(
        items = list(
          list(concept = list(CONCEPT_ID = 201820L),
               includeDescendants = TRUE)
        ),
        tables = c("condition_occurrence")
      )
    ),
    include = TRUE,
    type = "counts"
  )
)
```

`print.characterizationSettings` *Print Characterization Settings*

Description

Prints a human-readable summary of a `characterizationSettings` object.

Prints a human-readable summary of a `characterizationSettings` object.

Usage

```
## S3 method for class 'characterizationSettings'
print(x, ...)

## S3 method for class 'characterizationSettings'
print(x, ...)
```

Arguments

`x`	A `characterizationSettings` object.
`...`	Additional arguments (ignored).

Value

Invisibly returns `x`.

Invisibly returns `x`.

print.singleNodeSettingList *Print Single Node Setting List*

Description

Print Single Node Setting List

Usage

```
## S3 method for class 'singleNodeSettingList'
print(x, ...)
```

Arguments

x	A singleNodeSettingList object.
...	Additional arguments (ignored).

Value

Invisibly returns x.

print.singleNodeSpec *Print Single Node Spec*

Description

Print Single Node Spec

Usage

```
## S3 method for class 'singleNodeSpec'
print(x, ...)
```

Arguments

x	A singleNodeSpec object.
...	Additional arguments (ignored).

Value

Invisibly returns x.

renderAllSpecSql *Render SQL for All Specs in a singleNodeSettingList*

Description

Convenience wrapper that calls `renderSpecSql` on every element of a `singleNodeSettingList`.

Usage

```
renderAllSpecSql(specs, targetDialect = NULL, tempEmulationSchema = NULL)
```

Arguments

`specs`	A `singleNodeSettingList` object.
`targetDialect`	Character or `NULL`. Passed to `renderSpecSql`.
`tempEmulationSchema`	Character or `NULL`. Passed to `renderSpecSql`.

Value

A named character vector of rendered SQL statements, one per spec. Names are the analysis IDs (as character).

renderSpecSql *Render a Single-Node SQL Specification*

Description

Takes a `singleNodeSpec` object whose `sql` field contains a parameterised SQL template and resolves every `@placeholder` using the spec's own fields.

Usage

```
renderSpecSql(spec, targetDialect = NULL, tempEmulationSchema = NULL)
```

Arguments

`spec`	A `singleNodeSpec` object as produced by `singleNodeSetting`.
`targetDialect`	Character string (optional). When supplied, the rendered SQL is additionally translated to the target DBMS dialect via `translate`. Common values: `"postgresql"`, `"redshift"`, `"oracle"`, `"bigquery"`, `"spark"`. Default `NULL` returns SQL Server syntax.
`tempEmulationSchema`	Character string or `NULL`. Passed to `translate` for platforms that emulate temp tables (Oracle, BigQuery).

Value

A single character string of executable SQL.

`singleNodeSetting` *Create a Single Node Analysis Specification*

Description

Translates a `characterizationSettings` object (as returned by `planAnalysis`) into a list of executable SQL-based analysis specifications. Each specification ("node") pairs a feature domain, a time window, and the appropriate SQL template with all placeholders resolved.

Usage

```
singleNodeSetting(
  plan,
  cohortId,
  cohortDatabaseSchema,
  cohortTable,
  cdmDatabaseSchema,
  vocabularyDatabaseSchema = cdmDatabaseSchema,
  aggregated = TRUE,
  rowIdField = "subject_id"
)
```

Arguments

`plan`	A `characterizationSettings` object as returned by `planAnalysis`.
`cohortId`	Integer scalar. The target cohort definition ID.
`cohortDatabaseSchema`	Character scalar. Schema containing the target cohort table.
`cohortTable`	Character scalar. Name of the target cohort table.
`cdmDatabaseSchema`	Character scalar. Schema containing the OMOP CDM tables.
`vocabularyDatabaseSchema`	Character scalar. Schema containing the OMOP vocabulary tables. Used for concept name lookups in aggregated output. Defaults to `cdmDatabaseSchema`.
`aggregated`	Logical scalar. If `TRUE`, features are aggregated across patients (counts and proportions). If `FALSE`, patient-level rows are returned. Default: `TRUE`.
`rowIdField`	Character scalar. Name of the column in the cohort table to use as the row identifier in the output. Defaults to `"subject_id"`. Pass a custom field (e.g., a unique row key) when multiple cohort entries exist per person, such as when used inside `FeatureExtraction::getDbCovariateData()`.

Details

This function iterates over every enabled domain in `useBaseFeatures`, `useCohortFeatures`, and `useConceptSetFeatures`, crosses each with every analysis window, and produces a fully parameterised run specification.

The returned list can be passed directly to an execution engine that renders and translates the SQL via `SqlRender`.

Analysis ID Assignment:

Each specification receives a unique `analysisId` constructed as: `domainIndex * 1000 + windowIndex`, ensuring stable, reproducible identifiers across runs.

Value

A list of S3 objects of class `singleNodeSpec`. Each element contains:

`analysisId` Integer. Unique analysis identifier.

`analysisName` Character. Human-readable analysis label.

`domainTable` Character. CDM table name.

`conceptIdCol` Character. Concept ID column in the domain table.

`dateCol` Character. Start date column.

`dateColEnd` Character or `NULL`. End date column (for overlap logic).

`startDay` Integer. Window start day relative to index.

`endDay` Integer. Window end day relative to index.

`type` Character. Temporal logic (`"start"` or `"overlap"`).

`overlap` Logical. Whether overlap logic is used.

`atc` Logical. Whether ATC roll-up is applied.

`atcLevels` Integer vector or `NULL`. ATC levels.

`conceptSet` Logical. Whether a concept set filter is applied.

`conceptSetItems` List or `NULL`. Concept set items.

`aggregated` Logical. Aggregation flag.

`cohortId` Integer. Target cohort ID.

`cohortDatabaseSchema` Character. Cohort schema.

`cohortTable` Character. Cohort table name.

`cdmDatabaseSchema` Character. CDM schema.

`sql` Character. Parameterised SQL template.

`source` Character. Origin: `"base"`, `"cohort"`, or `"conceptSet"`.

See Also

`planAnalysis` for creating the analysis plan.

Examples

```
plan <- planAnalysis(
  useBaseFeatures = list(
    condition_occurrence = list(include = TRUE, type = "start"),
    drug_exposure = list(include = FALSE),
    condition_era = list(include = FALSE),
    drug_era = list(include = FALSE),
    procedure_occurrence = list(include = FALSE),
    observation = list(include = FALSE),
    device_exposure = list(include = FALSE),
    visit_occurrence = list(include = FALSE),
    measurement = list(include = FALSE)
  ),
  useCohortFeatures = list(include = FALSE),
  useConceptSetFeatures = list(include = FALSE)
)

specs <- singleNodeSetting(
  plan = plan,
  cohortId = 1L,
  cohortDatabaseSchema = "results",
  cohortTable = "cohort",
  cdmDatabaseSchema = "cdm"
)
```

Index