Query Notation (CQN)

Table of Contents

Introduction

CQN is a canonical plain object representation of CDS queries. Such query objects can be obtained by parsing CQL, by using the query builder APIs, or by simply constructing respective objects directly in your code.

For example, the following three snippets all construct the same query object:

```js
// Parsing CQL tagged template strings
let query = cds.ql `SELECT from Foo`
```

```js
// Query building
let query = SELECT.from (ref`Foo`)
```

```js
// Constructing plain CQN objects
let query = {SELECT:{from:[{ref:['Foo']}]}}
```

Such queries can be executed with `cds.run` :

```js
let results = await cds.run (query)
```

Following is a detailed specification of the CQN as TypeScript declarations, including all query types and their properties, as well as the fundamental expression types. Find the full CQN type definitions in the appendix below.

SELECT

Following is the TypeScript declaration of `SELECT` query objects:

```tsx
class SELECT { SELECT: {
  distinct?    : true
  count?       : true
  one?         : true
  from         : source
  columns?     : column[]
  where?       : xo[]
  having?      : xo[]
  groupBy?     : expr[]
  orderBy?     : order[]
  limit?      : { rows: val, offset: val }
}}
```

> Using: *source* , *column* , *xo* , *expr* , *order* , *val*

CQL SELECT queries enhance SQL's SELECT statements with these noteworthy additions:

- The `from` clause supports `{ref}` paths with *infix filters*.
- The `columns` clause supports deeply *nested projections*.
- The `count` property requests the total count, similar to OData's `$count` .
- The `one` property causes a single row object to be read instead of an array.

Also `SELECT` statements with `from` as the only mandatory property are allowed, which is equivalent to SQL's `SELECT * from ...` .

`.from`

Property `from` specifies the source of the query, which can be a table, a view, or a subquery. It is specified with type `source` as follows:

```tsx
class SELECT { SELECT: { //...
  from : source
}}
```

```tsx
type source = ref &as | SELECT | {
  join : 'inner' | 'left' | 'right'
  args : [ source, source ]
  on?  : expr
}
```

> Using: *ref* , *as* , *expr*
> Used in: *SELECT*

`.columns`

Property `columns` specifies the columns to be selected, projected, or aggregated, and is specified as an array of `column` s:

```tsx
class SELECT { SELECT: { //...
  columns : column[]
}}
```

```tsx
type column = '*' | expr &as &cast | ref &as &(
  { expand?: column[] } |
  { inline?: column[] }
) &infix
```

```tsx
interface as { as?: name }
interface cast { cast?: {type:name} }
interface infix {
  orderBy?  : order[]
  where?    : expr
  limit?    : { rows: val, offset: val }
}
```

> Using: *expr* , *name* , *ref* ,
>
> Used in: *SELECT*

.where
.having
.search

Properties *where* , and *having* , specify the filter predicates to be applied to the rows selected, or grouped, respectively. Property *search* is of same kind and is used for full-text search.

```tsx
class SELECT { SELECT: {
  where  : xo[]
  having : xo[]
  search : xo[]
}}
```

.orderBy

tsx

```tsx
class SELECT { SELECT: { //...
  orderBy : order[]
}}
```

tsx

```tsx
type order = expr & {
  sort  : 'asc' | 'desc'
  nulls : 'first' | 'last'
}
```

> Using: *`expr`*
>
> Used in: *`SELECT`*

INSERT

UPSERT

CQN representations for *`INSERT`* and *`UPSERT`* are essentially identical:

tsx

```tsx
class INSERT { INSERT: UPSERT['UPSERT'] }
class UPSERT { UPSERT: {
  into        : ref
  entries?    : data[]
  columns?    : string[]
  values?     : scalar[]
  rows?       : scalar[][]
  from?       : SELECT
}}
```

tsx

```tsx
interface data  { [elm:string]: scalar | data | data[] }
```

> Using: *`ref`* , *`expr`* *`scalar`* , *`SELECT`*
>
> See also: *`UPDATE.data`* ,

Data to be inserted can be specified in one of the following ways:

- Using *`entries`* as an array of records with name-value pairs.

- Using `values` as in SQL's *values* clauses.
- Using `rows` as an array of one or more `values`.

The latter two options require a `columns` property to specify names of columns to be filled with the values in the same order.

.entries

Allows input data to be specified as records with name-value pairs, including *deep* inserts.

```js
let q = {INSERT:{ into: { ref: ['Books'] }, entries: [
  { ID:201, title:'Wuthering Heights' },
  { ID:271, title:'Catweazle' }
]}}
```

```js
let q = {INSERT:{ into: { ref: ['Authors'] }, entries: [
  { ID:150, name:'Edgar Allen Poe', books: [
    { ID:251, title:'The Raven' },
    { ID:252, title:'Eleonora' }
  ]}
]}}
```

↳ *See definition in `INSERT` summary*

.values

Allows input data to be specified as an single array of values, as in SQL.

```js
let q = {INSERT:{ into: { ref: ['Books'] },
  columns: [ 'ID', 'title', 'author_id', 'stock' ],
  values: [ 201, 'Wuthering Heights', 101, 12 ]
}}
```

↳ *See definition in `INSERT` summary*

.rows

Allows input data for multiple rows to be specified as arrays of values.

```js
let q = {INSERT:{ into: { ref: ['Books'] },
  columns: [
    'ID', 'title', 'author_id', 'stock'
  ],
  rows: [
    [ 201, 'Wuthering Heights', 101, 12 ],
    [ 252, 'Eleonora', 150, 234 ]
  ]
}}
```

↳ *See definition in `INSERT` summary*

UPDATE

```tsx
class UPDATE { UPDATE: {
  entity   : ref
  where?   : expr
  data     : data
  with     : changes
}}
```

> Using: *`ref`* , *`expr`* , *`data`* , *`changes`*

`.data`

Data to be updated can be specified in property `data` as records with name-value pairs, same as in *`INSERT.entries`* .

```tsx
interface data  { [element:name]: scalar | data | data[] }
```

> Using: *`name`* , *`scalar`*

`.with`

Property `with` specifies the changes to be applied to the data, very similar to property `data` with the difference to also allow expressions as values.

```tsx
interface changes { [element:name]: scalar | expr | changes | changes[] }
```

> Using: `name` , `expr` , `scalar`

DELETE

```js
class DELETE { DELETE: {
  from    : ref
  where?  : expr
}}
```

> Using: `ref` , `expr`

Expressions

Expressions can be entity or element references, query parameters, literal values, lists of all the former, function calls, sub selects, or compound expressions.

```tsx
type expr  = ref | val | xpr | list | func | param | SELECT
```

```tsx
type ref   = { ref: ( name | { id:name &infix })[] }
type val   = { val: scalar }
type xpr   = { xpr: xo[] }
type list  = { list: expr[] }
type func  = { func: string, args: expr[] }
type param = { ref: [ '?' | number | string ], param: true }
type xo       = expr | keyword | operator
type operator = '=' | '==' | '!=' | '<' | '<=' | '>' | '>='
type keyword  = 'in' | 'like' | 'and' | 'or' | 'not'
type scalar   = number | string | boolean | null
type name     = string
```

NOTE

CQN by intent does not *understand* expressions and therefore keywords and operators are just represented as plain strings in flat `xo` sequences. This allows us to translate to and from any other query languages, including support for native SQL features.

Full *cqn.d.ts* File

cqn.d.ts

```tsx
/**
 * `INSERT` and `UPSERT` queries are represented by the same internal
 * structures. The `UPSERT` keyword is used to indicate that the
 * statement should be updated if the targeted data exists.
 * The `into` property specifies the target entity.
 *
 * The data to be inserted or updated can be specified in different ways:
 *
 * - in the `entries` property as deeply nested records.
 * - in the `columns` and `values` properties as in SQL.
 * - in the `columns` and `rows` properties, with `rows` being array of `valu
 * - in the `from` property with a `SELECT` query to provide the data to be i
 *
 * The latter is the equivalent of SQL's `INSERT INTO ... SELECT ...` statemer
 */
export class INSERT { INSERT: UPSERT['UPSERT'] }
export class UPSERT { UPSERT: {
  into      : ref
  entries?  : data[]
  columns?  : string[]
  values?   : scalar[]
  rows?     : scalar[][]
  from?     : SELECT
}}


/**
 * `UPDATE` queries are used to capture modifications to existing data.
 * They support a `where` clause to specify the rows to be updated,
 * and a `with` clause to specify the new values. Alternatively, the
 * `data` property can be used to specify updates with plain data only.
 */
export class UPDATE { UPDATE: {
  entity  : ref
  where?  : expr
  data    : data
  with    : changes
}}


/**
 * `DELETE` queries are used to remove data from a target datasource.
 * They support a `where` clause to specify the rows to be deleted.
 */
export class DELETE { DELETE: {
  from    : ref
  where?  : expr
}}


/**
 * `SELECT` queries are used to retrieve data from a target datasource,
 * and very much resemble SQL's `SELECT` statements, with these noteworthy
 * additions:
 *
 * - The `from` clause supports `{ref}` paths with infix filters.
 * - The `columns` clause supports deeply nested projections.
 * - The `count` property requests the total count, similar to OData's `$coun
 * - The `one` property indicates that only a single record object shall be
 *   returned instead of an array.
 *
 * Also, CDS, and hence CQN, supports minimalistic `SELECT` statements with a
 * as the only mandatory property, which is equivalent to SQL's `SELECT * fro
 */
```

```
export class SELECT { SELECT: {
  distinct?   : true
  count?      : true
  one?        : true
  from        : source
  columns?    : column[]
  where?      : xo[]
  having?     : xo[]
  groupBy?    : expr[]
  orderBy?    : order[]
  limit?      : { rows: val, offset: val }
}}

type source = OneOf< ref &as | SELECT | {
  join : 'inner' | 'left' | 'right'
  args : [ source, source ]
  on?  : expr
}>

type column = OneOf< '*' | expr &as &cast | ref &as & OneOf<(
  { expand?: column[] } |
  { inline?: column[] }
)> &infix >

type order = expr & {
  sort  : 'asc' | 'desc'
  nulls : 'first' | 'last'
}


interface changes { [elm:string]: OneOf< scalar | expr | changes | changes[] >
interface data { [elm:string]: OneOf< scalar | data | data[] >}
interface as { as?: name }
interface cast { cast?: {type:name} }

interface infix {
  orderBy?  : order[]
  where?    : expr
  limit?    : { rows: val, offset: val }
}


/**
 * Expressions can be entity or element references, query parameters,
 * literal values, lists of all the former, function calls, sub selects,
 * or compound expressions.
 */
export type expr  = OneOf< ref | val | xpr | list | func | param | SELECT >
export type ref   = { ref: OneOf< name | { id:name &infix } >[] }
export type val   = { val: scalar }
export type xpr   = { xpr: xo[] }
export type list  = { list: expr[] }
export type func  = { func: string, args: expr[] }
export type param = { ref: [ '?' | number | string ], param: true }

/**
 * This is used in `{xpr}` objects as well as in `SELECT.where` clauses to
 * represent compound expressions as flat `xo` sequences.
 * Note that CQN by intent does not _understand_ expressions and therefore
 * keywords and operators are just represented as plain strings.
 * This allows us to translate to and from any other query languages,
 * including support for native SQL features.
 */
type xo       = OneOf< expr | keyword | operator >
type operator = '=' | '==' | '!=' | '<' | '<=' | '>' | '>='
type keyword  = 'in' | 'like' | 'and' | 'or' | 'not'
type scalar   = number | string | boolean | null
type name     = string



// ---------------------------------------------------------------------------
//  maybe coming later...

declare class CREATE { CREATE: {} }
declare class DROP { DROP: {} }



// ---------------------------------------------------------------------------
//  internal helpers...

type OneOf<U> = Partial<(U extends any ? (k:U) => void : never) extends (k: i
```

Edit this page

Last updated: 14/02/2025, 05:06

Previous page
Query Language (CQL)

Next page
Expressions (CXN)

Was this page helpful?

👍 👎