Query Language (CQL)

CDS Query Language (CQL) is based on standard SQL, which it enhances by...

Table of Contents

Postfix Projections

CQL allows to put projections, that means, the *SELECT* clause, behind the *FROM* clause enclosed in curly braces. For example, the following are equivalent:

```sql
SELECT name, address.street from Authors
```

```sql
SELECT from Authors { name, address.street }
```

Nested Expands beta

Postfix projections can be appended to any column referring to a struct element or an association and hence be nested. This allows **expand** results along associations and hence read deeply structured documents:

```sql
SELECT from Authors {
   name, address { street, town { name, country }}
};
```

This actually executes three correlated queries to authors, addresses, and towns and returns a structured result set like that:

```js
results = [
  {
    name: 'Victor Hugo',
    address: {
      street: '6 Place des Vosges', town: {
        name: 'Paris',
        country: 'France'
      }
    }
  }, {
    name: 'Emily Brontë', …
  }, …
]
```

> This is rather a feature tailored to NoSQL databases and has no equivalent in standard SQL as it requires structured result sets. Some SQL vendors allow things like that with

> non-scalar subqueries in SELECT clauses.

> **WARNING**
>
> Nested Expands following *to-many* associations are not supported.

Alias

As the name of the struct element or association preceding the postfix projection appears in the result set, an alias can be provided for it:

```sql
SELECT from Authors {
   name, address as residence { street, town as city { name, country }}
};
```

The result set now is:

```js
results = [
  {
    name: 'Victor Hugo',
    residence: {
      street: '6 Place des Vosges', city: {
        name: 'Paris',
        country: 'France'
      }
    }
  }, …
]
```

Expressions

Nested Expands can contain expressions. In addition, it's possible to define new structures that aren't present in the data source. In this case an alias is mandatory and is placed *behind* the `{…}`:

```sql
SELECT from Books {
   title,
   author { name, dateOfDeath - dateOfBirth as age },
   { stock as number, stock * price as value } as stock
};
```

The result set contains two structured elements:

```js
results = [
  {
    title: 'Wuthering Heights',
    author: {
      name: 'Emily Brontë',
      age: 30
    },
    stock: {
      number: 12,
      value: 133.32
    }
  }, …
]
```

Nested Inlines beta

Put a `"."` before the opening brace to **inline** the target elements and avoid writing lengthy lists of paths to read several elements from the same target. For example:

```sql
SELECT from Authors {
   name, address.{ street, town.{ name, country }}
};
```

... is equivalent to:

```sql
SELECT from Authors {
  name,
  address.street,
  address.town.name,
  address.town.country
};
```

Nested Inlines can contain expressions:

```sql
SELECT from Books {
   title,
   author.{
     name, dateOfDeath - dateOfBirth as author_age,
     address.town.{ concat(name, '/', country) as author_town }
   }
};
```

The previous example is equivalent to the following:

```sql
  SELECT from Books {
    title,
    author.name,
    author.dateOfDeath - author.dateOfBirth as author_age,
    concat(author.address.town.name, '/', author.address.town.country) as auth
  };
```

Smart `*` Selector

Within postfix projections, the `*` operator queries are handled slightly different than in plain SQL select clauses.

Example:

```sql
  SELECT from Books { *, author.name as author }
```

Queries like in our example, would result in duplicate element effects for `author` in SQL. In CQL, explicitly defined columns following an `*` replace equally named columns that have been inferred before.

Excluding Clause

Use the `excluding` clause in combination with `SELECT *` to select all elements except for the ones listed in the exclude list.

```sql
  SELECT from Books { * } excluding { author }
```

The effect is about **late materialization** of signatures and staying open to late extensions. For example, assume the following definitions:

```cds
entity Foo { foo : String; bar : String; car : String; }
entity Bar as select from Foo excluding { bar };
entity Boo as select from Foo { foo, car };
```

A `SELECT * from Bar` would result into the same as a query of `Boo` :

```sql
SELECT * from Bar --> { foo, car }
SELECT * from Boo --> { foo, car }
```

Now, assume a consumer of that package extends the definitions as follows:

```cds
extend Foo with { boo : String; }
```

With that, queries on `Bar` and `Boo` would return different results:

```sql
SELECT * from Bar --> { foo, car, boo }
SELECT * from Boo --> { foo, car }
```

In Nested Expands beta

If the `*` selector is used following an association, it selects all elements of the association target. For example, the following queries are equivalent:

```sql
SELECT from Books { title, author { * } }
```

```sql
SELECT from Books { title, author { ID, name, dateOfBirth, … } }
```

A `*` selector following a struct element selects all elements of the structure and thus is equivalent to selecting the struct element itself. The following queries are all equivalent:

```sql
SELECT from Authors { name, struc { * } }
SELECT from Authors { name, struc { elem1, elem2 } }
SELECT from Authors { name, struc }
```

The `excluding` clause can also be used for Nested Expands:

```sql
SELECT from Books { title, author { * } excluding { dateOfDeath, placeOfDeath
```

In Nested Inlines beta

The expansion of `*` in Nested Inlines is analogous. The following queries are equivalent:

```sql
SELECT from Books { title, author.{ * } }
SELECT from Books { title, author.{ ID, name, dateOfBirth, … } }
```

The `excluding` clause can also be used for Nested Inlines:

```sql
SELECT from Books { title, author.{ * } excluding { dateOfDeath, placeOfDeath
```

Path Expressions

Use path expressions to navigate along associations and/or struct elements in any of the SQL clauses as follows:

In `from` clauses:

```sql
SELECT from Authors[name='Emily Brontë'].books;
SELECT from Books:authors.towns;
```

In `select` clauses:

```sql
SELECT title, author.name from Books;
SELECT *, author.address.town.name from Books;
```

In `where` clauses:

```sql
SELECT from Books where author.name='Emily Brontë'
```

The same is valid for `group by`, `having`, and `order by`.

Path Expressions in `from` Clauses

Path expressions in from clauses allow to fetch only those entries from a target entity, which are associated to a parent entity. They unfold to *SEMI JOINS* in plain SQL queries. For example, the previous mentioned queries would unfold to the following plain SQL counterparts:

```sql
SELECT * from Books WHERE EXISTS (
  SELECT 1 from Authors WHERE Authors.ID = Books.author_ID
    AND Authors.name='Emily Brontë'
);
```

```sql
SELECT * from Towns WHERE EXISTS (
  SELECT 1 from Authors WHERE Authors.town_ID = Towns.ID AND EXISTS (
    SELECT 1 from Books WHERE Books.author_ID = Authors.ID
  )
);
```

Path Expressions in All Other Clauses

Path expressions in all other clauses are very much like standard SQL's column expressions with table aliases as single prefixes. CQL essentially extends the standard behavior to paths with multiple prefixes, each resolving to a table alias from a corresponding `LEFT OUTER JOIN`. For example, the path expressions in the previous mentioned queries would unfold to the following plain SQL queries:

```sql
-- plain SQL
SELECT Books.title, author.name from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID;
```

```sql
-- plain SQL
SELECT Books.*, author_address_town.name from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID
LEFT JOIN Addresses author_address ON author_address.ID = author.address_ID
LEFT JOIN Towns author_address_town ON author_address_town.ID = author_addres
```

```sql
-- plain SQL
SELECT Books.* from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID
WHERE author.name='Emily Brontë'
```

> **TIP**
>
> All column references get qualified → in contrast to plain SQL joins there's no risk of ambiguous or conflicting column names.

With Infix Filters

Append infix filters to associations in path expressions to narrow the resulting joins. For example:

```sql
SELECT books[genre='Mystery'].title from Authors
 WHERE name='Agatha Christie'
```

... unfolds to:

```sql
SELECT books.title from Authors
LEFT JOIN Books books ON ( books.author_ID = Authors.ID )
  AND ( books.genre = 'Mystery' )  //--> from Infix Filter
WHERE Authors.name='Agatha Christie';
```

If an infix filter effectively reduces the cardinality of a *to-many* association to *one*, make this explicit with:

```sql
SELECT name, books[1: favorite=true].title from Authors
```

Exists Predicate

Use a filtered path expression to test if any element of the associated collection matches the given filter:

```sql
SELECT FROM Authors {name} WHERE EXISTS books[year = 2000]
```

...unfolds to:

```sql
SELECT name FROM Authors
WHERE EXISTS (
        SELECT 1 FROM Books
        WHERE Books.author_id = Authors.id
            AND Books.year = 2000
    )
```

Exists predicates can be nested:

```sql
SELECT FROM Authors { name }
    WHERE EXISTS books[year = 2000 and EXISTS pages[wordcount > 1000]]
```

A path with several associations is rewritten as nested exists predicates. The previous query is equivalent to the following query.

```sql
SELECT FROM Authors { name }
    WHERE EXISTS books[year = 2000].pages[wordcount > 1000]
```

> **WARNING**
>
> Paths *inside* the filter are not yet supported.

Casts in CDL

There are two different constructs commonly called casts. SQL casts and CDL casts. The former produces SQL casts when rendered into SQL, whereas the latter does not:

```sql
SELECT cast (foo+1 as Decimal) as bar from Foo;  -- standard SQL
SELECT from Foo { foo+1 as bar : Decimal };      -- CDL-style
```

↳ *Learn more about CDL type definitions*

Use SQL casts when you actually want a cast in SQL. CDL casts are useful for expressions such as `foo+1` as the compiler does not deduce types. For the OData backend, by specifying a type, the compiler will also assign the correct EDM type in the generated EDM(X) files.

> **TIP**
>
> You don't need a CDL cast if you already use a SQL cast. The compiler will extract the type from the SQL cast.

Use enums

In queries, you can use enum symbols instead of the respective literals in places where the corresponding type can be deduced:

```cds
type Status : String enum { open; closed; in_progress; };

entity OpenOrder as projection on Order {

  case status when #open        then 0
              when #in_progress then 1 end
    as status_int : Integer,

  (status = #in_progress ? 'is in progress' : 'is open')
    as status_txt : String,

} where status = #open or status = #in_progress;
```

Association Definitions

Query-Local Mixins

Use the `mixin...into` clause to logically add unmanaged associations to the source of the query, which you can use and propagate in the query's projection. This is only supported in postfix notation.

```sql
SELECT from Books mixin {
  localized : Association to LocalizedBooks on localized.ID = ID;
} into {
  ID, localized.title
};
```

In the select list

Define an unmanaged association directly in the select list of the query to add the association to the view's signature. This association cannot be used in the query itself. In contrast to mixins, these association definitions are also possible in projections.

```cds
entity BookReviews as select from Reviews {
  ...,
  subject as bookID,
  book : Association to Books on book.ID = bookID
};
```

In the ON condition you can, besides target elements, only reference elements of the select list. Elements of the query's data sources are not accessible.

This syntax can also be used to add new unmanaged associations to a projection or view via `extend`:

```cds
extend BookReviews with columns {
  subject as bookID,
  book : Association to Books on book.ID = bookID
};
```

👍 👎