ECSQL – a text based command language for EC content

Contents

What is ECSQL? 2
ECSQL in detail 2
Basic data types in ECSQL 2
Predicates 4
ECSQL Parameters 4
Fully qualifying ECClasses in ECSQL 5
LIMIT and OFFSET 5
ECRelationships in ECSQL 5
Joins 6
Structs (Embedded types) 11
Arrays 11
Type operations and Polymorphism 11
Instance identification concepts 12
Functions 13
Extending ECSQL 13

What is ECSQL?

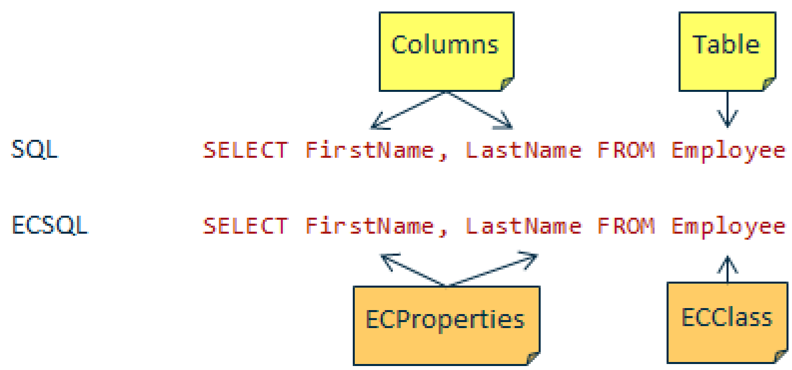
ECSQL is a **text-based command language** for CRUD (create, read, update, delete) operations **against ECInstances in an EC repository**.

ECSQL is an implementation of SQL—a proven, well-adopted text-based command language. It sticks to standard SQL (SQL-92 and SQL-99) wherever possible.

Especially the SQL-99 standard came with a lot of features ECObjects has too: boolean, date time, binary data types, structs, arrays, polymorphism. This allows ECSQL to deviate only in very few exceptions from standard SQL.

Anyone familiar with SQL should intuitively understand ECSQL.

The key difference between ECSQL and SQL is that ECSQL targets the logical schema, and not the underlying database's persistence schema.



In particular this means that ECSQL is repository agnostic (like SQL).

ECSQL in detail

This document will only list the exceptions to standard SQL and the cases where less known features of the standard are used. Standard SQL refers to SQL-92 (aka SQL 2), and to SQL-99 (aka SQL 3) whenever SQL-92 is not sufficient.

Basic data types in ECSQL

In addition to the common numeric and string data types ECSQL supports data types like date, date time, boolean and binary.

Boolean

For Boolean types ECSQL supports the literals `True` and `False`.

SQL compliance: SQL-99 with exception

Note: SQL-99 also supports the UNKNOWN literal for Booleans. This is not supported by ECSQL as there is no matching counterpart in ECObjects.

Examples

```
SELECT * FROM myschema.Foo WHERE HasWarranty = True
SELECT * FROM myschema.Foo WHERE HasWarranty <> False
```

DateTime

ECSQL supports `DATE` for dates without time and `TIMESTAMP` for dates with time.

SQL compliance: SQL-99 with exceptions

Note: SQL-99 also supports a TIME type (time in the day). ECSQL does not support that as ECObjects doesn't support that.
Likewise the SQL-99 features for local times and time zones are not supported because they imply implicit time zone conversions which are tricky to do right in general and portable way. Therefore time zone conversions be handled by the application.

Literals

```
DATE 'yyyy-mm-dd'
TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.n[nnnnn]]'
```

Basic functions

`CURRENT_DATE`	returns the current date
`CURRENT_TIMESTAMP`	returns the current timestamp **in UTC.** ***Note*** This is a deviation from SQL-99 which returns a timestamp in local time. As local times can imply implicit time zone conversions which are hard to implement genericly, ECSQL chose UTC.

Examples

```
SELECT * FROM myschema.Foo WHERE LastMaintenanceDate > DATE '2010-03-31'
SELECT * FROM myschema.Foo WHERE LastModDateTime > TIMESTAMP '2010-01-01 12:00:51.123456Z'

UPDATE myschema.Foo SET LastModDateTime = CURRENT_TIMESTAMP WHERE …
UPDATE myschema.Foo SET LastMaintenanceDate = CURRENT_DATE WHERE …
```

Points

Although there is no SQL-99 equivalent, points are a system primitive type in ECObjects and are therefore supported in ECSQL.

SQL compliance: none

Basic functions

`GetX`	returns the X coordinate of the Point2D or Point3D.
`GetY`	returns the Y coordinate of the Point2D or Point3D.
`GetZ`	returns the Z coordinate of the Point3D.

Examples

```
SELECT ECInstanceId FROM myschema.Foo WHERE GetX(SrsOrigin)>=3500000.0 AND
GetY(SrsOrigin)>=5700000.0
```

Blobs

ECSQL supports the data type `BLOB` (as in SQL-99) which maps to the ECObjects type `Binary`.

SQL compliance: SQL-99

Predicates

Predicates are functions or expressions that return true or false. When used in Boolean expressions (as in the where clause), it is not needed to compare them to true or false as they return a Boolean value already. (See SQL-99)

Examples

Suppose that the ECClass `Foo` has the ECProperties `Diameter` of type double and `HasWarranty` of type boolean:

```
SELECT * FROM myschema.Foo WHERE Diameter > 0 AND HasWarranty
```

Suppose that a custom function `Contains` was defined that returns true if the first string argument contains the second string (see also section CUSTOM FUNCTIONS):

```
SELECT * FROM myschema.Foo WHERE Diameter > 0 AND Contains(Name, ‘bla’)
```

Note: SQLite supports that, too.

The NOT operator

`NOT` is a standard SQL keyword that negates any Boolean expression. That said, `NOT` can be used with any predicate to negate its meaning.

Examples

```
SELECT * FROM myschema.Foo WHERE Diameter > 0 AND NOT HasWarranty
SELECT * FROM myschema.Foo WHERE Diameter > 0 AND NOT Contains(Name, ‘bla’)
```

ECSQL Parameters

To bind values to an ECSQL statement after preparation, the following parameter placeholders are supported.

?	Refers to a parameter whose index is one greater than the previous parameter in the ECSQL statement.
:aaa	Named parameter. This allows to bind the same value to more than one placeholder.

Example

```
SELECT Id FROM myschema.Pump WHERE State = ?
SELECT Id FROM myschema.Pump WHERE State = :state OR (Material = :mat AND State <> :state)
```

Fully qualifying ECClasses in ECSQL

If just the name of a class is ambiguous in the context of a current session, the class has to be fully qualified by the schema.

Syntax: `<Schema name or schema namespace prefix>.<Class Name>`

SQL compliance: SQL-92

Example

Suppose class Pump resides in the schema OpenPlant (namespace prefix *op*)

```
SELECT Pump.* FROM OpenPlant.Pump
```

or

```
SELECT Pump.* FROM op.Pump
```

LIMIT and OFFSET

One way for a client to implement paging is to use the LIMIT and OFFSET clauses with an ECSQL statement.

The LIMIT clause is used to limit the number of results returned from an ECSQL statement. Using LIMIT together with OFFSET allows specifying a range of rows to be returned. The OFFSET hereby specifies the first M instances to be omitted from the result set. The LIMIT specifies the next N instances to be returned.

If the ECSQL statement would return less than M+N instances, then the first M rows are skipped and the remaining rows (if any) are returned.

Examples:

```
SELECT p.* FROM myschema.Pump p WHERE p.State = 'IsOn' ORDER BY ECInstanceId LIMIT 50
```

returns the first 50 pumps of the result set.

```
SELECT p.* FROM myschema.Pump p WHERE p.State = 'IsOn' ORDER BY ECInstanceId LIMIT 50 OFFSET
100
```

returns the pumps 101 through 150 of the result set.

ECRelationships in ECSQL

Defining an ECRelationship in the ECSchema primarily means to specify the ECClasses that can be related through that relationship. The two ends of the ECRelationship are called **source** and **target** constraint. See *ECObjects* documentation for details on ECSchemas and ECRelationships.

Example

```
<ECRelationshipClass typeName="CompanysEmployees" isDomainClass="True" strength="embedding">
    <Source cardinality="(1,1)" roleLabel="Company">
        <Class class="Company" />
    </Source>
    <Target cardinality="(0,N)" roleLabel="Employee">
        <Class class="Employee" />
    </Target>
</ECRelationshipClass>
```

In the context of ECSQL you can therefore think of ECRelationships as virtual **link tables**. Just like link tables they are used to relate two classes to each other. They are virtual because they do not have to exist in the underlying DB.

In ECSQL an ECRelationship has the following system properties – in addition to any other properties defined by users on the ECRelationship:

ECRelationship structure in ECSQL

Property	Description
ECInstanceId	ECInstanceId of the relationship instance (inherited from ECClass)
SourceECInstanceId	ECInstanceId of the instance on the source end of the ECRelationship
SourceECClassId	ECClassId of the instance on the source end of the ECRelationship
TargetECInstanceId	ECInstanceId of the instance on the target end of the ECRelationship
TargetECClassId	ECClassId of the instance on the target end of the ECRelationship

See also section GETECCLASSID FUNCTION.

Examples

Returning the source and target constraints of all CompanysEmployees relationships translates to:

```
SELECT SourceECInstanceId, SourceECClassId, TargetECInstanceId, TargetECCClassId
FROM ONLY myschema.CompanysEmployees
```

Establishing a relationship between the company with ECInstanceId 12345 and the employee with ECInstanceId of 523323 translates to:

```
INSERT INTO myschema.CompanysEmployees (SourceECInstanceId, SourceECClassId,
TargetECInstanceId, TargetECClassId)
VALUES (12345, 1001, 523323, 1002)
```

Deleting the above relationship would be achieved by:

```
DELETE FROM ONLY myschema.CompanysEmployees
WHERE SourceECInstanceId = 12345 AND SourceECClassId = 1001 AND
      TargetECInstanceId = 23323 AND TargetECClassId = 1002
```

or if the ECInstanceId of the relationship was known:

```
DELETE FROM ONLY myschema.CompanysEmployees WHERE ECInstanceId = 31234
```

Joins

The ECSQL join support is a combination of leveraging the power of ECRelationships and the flexibility of regular SQL joins.

The key keywords are

- `JOIN`: specifies the other end point of the join, i.e. the related class ("*What to join*")
- `ON / USING`: specifies the join condition (*"How to join*")

JOIN … ON … / Theta style

The `JOIN ON` syntax and its equivalent, the *theta* style, are the standard and general form for joins in ECSQL. The syntax is standard SQL-92 and can therefore be used without limitation.

When used in conjunction with ECRelationships, the syntax builds upon the semantics of ECRelationships being virtual link table with well-defined system properties (see above).

SQL compliance: SQL-92

Example

Taking the example ECRelationship from above, querying for "all employees of company ACME which joined the company before January 1st, 2000" would translate to the following ECSQL statement:

JOIN ON syntax

```
SELECT e.* FROM ONLY myschema.Employee e
JOIN ONLY myschema.CompanysEmployees rel
ON e.ECInstanceId = rel.TargetECInstanceId AND
   e.GetECClassId() = rel.TargetECClassId
JOIN ONLY myschema.Company c
ON c.ECInstanceId = rel.SourceECInstanceId AND
   c.GetECClassId() = rel.SourceECClassId
WHERE c.Name = 'ACME' AND e.JoinDate < DATE '2000-01-01'
```

Theta style

```
SELECT e.* FROM ONLY myschema.Employee e,
              ONLY myschema.CompanysEmployees rel,
              ONLY myschema.Company c
WHERE e.ECInstanceId = rel.TargetECInstanceId AND
      e.GetECClassId() = rel.TargetECClassId AND
      c.ECInstanceId = rel.SourceECInstanceId AND
      c.GetECClassId() = rel.SourceECClassId AND
      c.Name = 'ACME' AND e.JoinDate < DATE '2000-01-01'
```

JOIN … USING …

The ECRelationship **implies** the actual SQL join condition. In many cases it therefore is sufficient to just specify the ECRelationship instead of the rather verbose ordinary join conditions via the JOIN ON or theta syntax.

Syntax: `JOIN <joined ECClass> USING <ECRelationshipClass>`

SQL compliance: deviation from SQL-92

Every JOIN … USING … clause can be translated into a set of JOIN … ON … clauses (see examples below).

Rules

- The class in the JOIN clause is either the source or target end of the relationship class in the USING clause.
- One of <u>all</u> other classes (except the class in JOIN) in the ECSQL is the other end of the relationship class

This implies that

- the USING clause is dependent on the JOIN clause
  Note: This is a difference to the JOIN ON / theta syntax where the ON clause is **in**dependent of JOIN clause,
- a JOIN USING clause is never dependent on any other JOIN clause in the ECSQL (same as for JOIN ON / theta),
- the order of multiple join clauses is irrelevant (same as for JOIN ON / theta).

For more complex joins like join types other than inner joins, or joins on nested select statements, the `JOIN USING` syntax is not unambiguous anymore. In those cases, the more general form of `JOIN ON` or the *theta* style has to be used (see section JOIN … ON … / THETA STYLE).

Note on terminology: In plain SQL the **`USING`** keyword indicates that the join condition can be implied from what comes after the `USING` – here the ECRelationship class. In ECSQL the semantic is the analog, but the syntax deviates from standard SQL.

Example

Transforming the ECSQL JOIN ON example from the previous section to using the JOIN USING syntax looks like this:

JOIN ON syntax

```
SELECT e.* FROM ONLY myschema.Employee e
JOIN ONLY myschema.CompanysEmployees rel
ON e.ECInstanceId = rel.TargetECInstanceId AND
   e.GetECClassId() = rel.TargetECClassId
JOIN ONLY myschema.Company c
ON c.ECInstanceId = rel.SourceECInstanceId AND
   c.GetECClassId() = rel.SourceECClassId
WHERE c.Name = ‘ACME’ AND e.JoinDate < DATE ‘2000-01-01’
```

JOIN USING syntax

```
SELECT e.* FROM ONLY myschema.Employee e
JOIN ONLY myschema.Company c USING myschema.CompanysEmployees
WHERE c.Name = ‘ACME’ AND e.JoinDate < DATE ‘2000-01-01’
```

Resolving ambiguity in a JOIN … USING … clause

The JOIN USING syntax is a convenient shortcut for the more verbose JOIN ON syntax. JOIN USING implies that the *source* and *target* of the relationship specified in the USING clause can unambiguously be determined from all classes used in the ECSQL.
However, in some special cases, this is not possible. In those cases the ECSQL needs to specify special keywords to resolve the ambiguity.

To resolve a JOIN USING clause, an ECSQL interpreter has to **find out which of the classes in the ECSQL**

- **is the source,**
- **is the target**

in the relationship class in the USING clause.

If source and target cannot be determined unambiguously, the keywords described in the following sections have to be used to resolve ambiguity.

Note: Alternatively, the JOIN ON syntax can be used. It does not need the special keywords, as it is always unambiguous in this respect.

FORWARD / BACKWARD

If a class in the ECSQL can either be the source or the target of the relationship in the USING clause, **`FORWARD`** or **`BACKWARD`** is used to resolve the ambiguity:

- `FORWARD`: indicates that the join goes from source to target class, i.e. the **class in the JOIN clause is the target constraint** of the relationship.
- `BACKWARD`: indicates that a join goes from target to source, i.e. the **class in the JOIN clause is the source constraint** of the relationship.

Example

Assume an ECClass `Folder` and an ECRelationship `FolderHasSubfolders` relating the `Folder` class to itself:

```
<ECRelationshipClass typeName="FolderHasSubfolders" isDomainClass="True" strength="embedding">
    <Source cardinality="(0,1)" roleLabel="Parent">
        <Class class="Folder" />
    </Source>
    <Target cardinality="(0,N)" roleLabel="Subfolder">
        <Class class="Folder" />
    </Target>
</ECRelationshipClass>
```

Querying the parent folder of a given folder would translate to

```
SELECT parent.Name FROM ONLY myschema.Folder parent
JOIN ONLY myschema.Folder subfolder USING myschema.FolderHasSubfolders FORWARD
WHERE subfolder.Name = ‘My Documents’
```

Querying all subfolders of a given folder would translate to

```
SELECT subfolder.Name FROM ONLY myschema.Folder subfolder
JOIN ONLY myschema.Folder parent USING FolderHasSubfolders BACKWARD
WHERE parent.Name = ‘My Pictures’
```

WITH <class or class alias>

If more than one class matches a constraint of the relationship in the USING clause, a **`WITH`** clause is used to resolve ambiguity.

Example

Assume ECClasses `A`, `B`, `C` and ECRelationships `AHasB` where `A` is source and `B` is target and `CHasAOrB` where `C` is source and `A` and `B` are possible target classes.

In the ECSQL

```
SELECT * FROM ONLY myschema.A
JOIN ONLY myschema.B USING myschema.AHasB
JOIN ONLY myschema.C USING myschema.CHasAOrB
```

the second JOIN USING clause is ambiguous (the first is not). Three classes are used in the ECSQL: `A`, `B`, `C`. `C` is unambiguously matching the source end of `CHasAOrB`. The target constraint however is matched by both `A` and `B`.

If **A** should be used as target constraint, the correct ECSQL is:

```
SELECT * FROM ONLY myschema.A
JOIN ONLY myschema.B USING myschema.AHasB
JOIN ONLY myschema.C USING myschema.CHasAOrB WITH A
```

If **B** should be used as target constraint, the correct ECSQL is:

```
SELECT * FROM ONLY myschema.A
JOIN ONLY myschema.B USING myschema.AHasB
JOIN ONLY myschema.C USING myschema.CHasAOrB WITH B
```

Summary: JOIN USING Resolution Logic

The resolution logic for a specific JOIN USING clause looks like this:
JOIN class: class in JOIN clause
Relationship: Relationship class in USING clause

1. Does *JOIN class* match source or target constraint of *relationship*?
   Yes: continue
   No: abort. ECSQL is invalid.
2. Does *JOIN class* match both source and target constraints?
   Yes: Is FORWARD or BACKWARD specified?
     Yes: continue
     No: abort. ECSQL is invalid
   No: continue

➔ One end of the *relationship* is resolved.

3. Do any of the other classes in the ECSQL match the other end of the *relationship*?
   Yes: continue
   No: abort. ECSQL is invalid
4. Does more than one class in the ECSQL match the other end of the *relationship*?
   Yes: Is WITH specified along with one of the matching classes?
     Yes: continue
     No: abort. ECSQL is invalid.
   No: continue

➔ Both ends of the *relationship* are resolved successfully.

Ad-hoc joins

Ad-hoc joins or on-the-fly joins are not defined by an ECRelationship but by a regular SQL join condition. They are called ad-hoc because they do not require to be predefined in the ECSchema, thus can be established at any time.

For ad-hoc joins, the regular SQL join syntax can be used, e.g. `JOIN ON` or the *theta* style.

Example

Suppose there is an ECClass `City` and an ECClass `State` for which no ECRelationship has been defined. We can still join the two classes via an ad-hoc join. Querying for all cities that are located in the state of Alaska translates to the following ECSQL:

JOIN ON Syntax

```
SELECT c.* FROM City c
JOIN State s ON c.StateId = s.ECInstanceId
WHERE s.Name = 'Alaska'
```

Theta style

```
SELECT c.* FROM City c, State s
WHERE c.StateId = s.ECInstanceId and
      s.Name = 'Alaska'
```

Notes

Further features which come from standard SQL, but may not be obvious:

- As many `JOIN … ON / USING …` clauses as needed can be specified in a statement, e.g. when more than one relationship is involved
- Other SQL join features, like inner joins are implicitly available in ECSQL, too..

Structs (Embedded types)

SQL compliance: SQL-99

The straightforward operator for referencing structs, i.e. members in the structs, is the '.'. This is a known token in standard SQL for referencing columns in tables.

Examples

Selecting all plants that are located in cities that start with 'San An' translates to:

```
SELECT * FROM myschema.Plant p
WHERE p.Address.City like 'San An%'
```

Updating the ZIP code of the plant's address would translate to:

```
UPDATE myschema.Plant p
SET p.Address.Zip = 13423
WHERE p.Id = 42
```

Any nesting depth can theoretically occur:

```
SELECT c.AStructProp.A.B.C FROM myschema.MyClass c
WHERE AnotherStructProp.M.N.O.Diameter > 50.0
```

Arrays

Like structs, arrays are covered by SQL-99, but not by SQL-92. ECSQL follows the SQL-99 standard for arrays.

Type operations and Polymorphism

ECObjects supports inheritance of ECClasses and polymorphic queries. SQL-99 does so, too (SQL-92 does not). Only the subset of the SQL-99 type functionality is included in ECSQL which is considered required in the EC world. If need be, more could be added later.

Polymorphic queries

As in SQL-99 the ECSQL syntax of polymorphic queries and non-polymorphic queries is as follows:

- `SELECT * FROM myschema.BaseClass` returns all instances from BaseClass and all its subclasses.
- `SELECT * FROM ONLY myschema.BaseClass` returns all instances from BaseClass but none from its subclasses.

GetECClassId Function

ECSQL provides the function **`GetECClassId()`** that returns the ECClassId of the class on which it is invoked (not part of SQL-99 standard)

Note Not all EC repositories support the ECClassId yet. The GetECClassId() function can therefore only be called on repositories that support the ECClassId.

Examples

```
SELECT GetECClassId() FROM myschema.Employee
SELECT e.GetECClassId(), c.GetECClassId() FROM myschema.Employee e
       JOIN myschema.Company c USING myschema.CompanyHasEmployees rel
```

Instance identification concepts

Like in ECObjects, ECSQL provides a generic abstract concept to refer to the primary key and the foreign keys of an instance. ECSQL users therefore don't need to know what columns a primary key or foreign key is made up of in the underlying database.

ECInstanceId

The ECInstanceId is the equivalent of ECObjects' InstanceId. It is the equivalent concept of a primary key in a database.

ECInstanceId Property

In the context of ECSQL each ECClass has a built-in system ECProperty called **`ECInstanceId`**. It can be referenced in ECSQL like any other ECProperty of the class, too. As it is a built-in property, the property is not needed to be defined in the ECSchema.

Examples

```
SELECT ECInstanceId FROM myschema.Foo WHERE

SELECT Name, ECInstanceId FROM myschema.Foo ORDER BY Owner ASC, ECInstanceId DESC
```

ECInstanceId Data Type

The ECInstanceId can either be an integral value or a string – and implicit conversions exist between the two.

The string type is mainly there for compatibility with ECObjects (as its InstanceId is always a string) or for future cases where the primary key is not a single numeric column.

So the following statements are both valid:

```
SELECT * FROM myschema.Foo WHERE ECInstanceId = 123424
SELECT * FROM myschema.Foo WHERE ECInstanceId = '123424'
```

Note: It is up to the implementation how or whether the implicit conversion between string and number is implemented.

Functions

ECSQL supports any function call. Of course, implementations of ECSQL must provide the implementation of the functions called.

The following functions are ECSQL core functions:

- GetECClassId (see section GETECCLASSID FUNCTION)
- CURRENT_DATE and CURRENT_TIMESTAMP (see BASIC FUNCTIONS in section DATETIME)
- GetX, GetY, GetZ (see BASIC FUNCTIONS in section POINTS)

It is up to the ECSQL implementations to provide built-in functions and/or APIs to register custom functions.
Likewise, any SQL function exposed by the underlying database can be used in ECSQL, too.

For example, all SQL functions in SQLite, whether built in or custom functions, can be used in ECSQL (see also http://www.sqlite.org/lang_corefunc.html).

Extending ECSQL

ECSQL provides a couple of extensibility points.

Custom ECSchemas

Anything that can be expressed through ECClasses or other entities of an ECSchema is implicitly available in ECSQL.

Custom Functions

Functions are a powerful mean to extend the functionality of ECSQL. As described in section FUNCTIONS ECSQL implementations can provide APIs to register custom ECSQL functions or custom functions for the underlying database. Both are immediately available in ECSQL.