

# Contents

- [Transfer Query Language](#)
  - [The Transfer Query Object](#)
    - [Creating a Query Object](#)
    - [TQL Query Methods](#)
      - [Query.setAliasColumns\(aliasColumns\)](#)
      - [Query.setDistinctMode\(distinctMode\)](#)
      - [Query.setParam\(name, \[value\], \[type\], \[list\], \[isNull\]\)](#)
      - [Query.setCacheEvaluation\(cacheEvaluation\)](#)
  - [TQL Custom Tags](#)
  - [Transfer Query Language Syntax](#)
    - [Identifiers](#)
    - [Basic List Queries](#)
    - [Selecting specific columns](#)
    - [Aliasing column names](#)
    - [Aliasing columns names, while also selecting all](#)
    - [Aliasing Classes](#)
    - [Selecting properties from aliased classes](#)
    - [Where statements with operators](#)
    - [Where statement with an IN statement](#)
    - [Where statement with a NULL statement](#)
    - [Where statement with combinations of operators, NULL and IN statements](#)
    - [Where statements with aliased classes](#)
    - [Auto Joins](#)
    - [Specific Join](#)
    - [Manual Joins](#)
    - [Multiple Joins](#)
    - [Left and Right Outer Joins](#)
    - [Order By](#)
    - [Not supported in TQL](#)

## Transfer Query Language

Transfer Query Language is another abstraction layer between the database and the Object model that the developer has created, that allows you to perform SQL like queries against the database.

This has been developed so that you can leverage the knowledge you already have about your Object Oriented model to do your gateway queries, as well as being able to do queries with less code, as Transfer already knows what columns are required per table, and the relationships between objects.

Transfer then takes the responsibility to translate the TQL script, combined with your Object configuration file into SQL that is then run against the database.

## The Transfer Query Object

The Transfer Query object is a CFC of type [transfer.com.tql.Query](#) , and is used as the main holder for information about the TQL Script.

To do a [Transfer.listByQuery\(\)](#) or a [Transfer.readByQuery\(\)](#) you must first create a new TQL Query object. For more information on these methods, check the [Database Management Methods](#) section.

### Creating a Query Object

To create a query object, call the [createQuery\(tql\)](#) method on the [transfer.com.Transfer](#) CFC with the required TQL Script as an argument. This will return the TQL Query object that is based on the entered TQL script.

For example:

```
query = getTransfer().createQuery( "from post.Post" );
```

Where 'query' is the resultant [transfer.com.tql.Query](#) object.

### TQL Query Methods

The TQL Query Object has a series of methods that can effect how the TQL script is resolved by Transfer:

#### Query.setAliasColumns(aliasColumns)

##### (API)

By default, database columns in the resultant query are aliased to the name of the *Property* that they are assigned in the Object configuration XML file. To turn this off, so that the query uses the original column names:

```
query.setAliasColumns( false );
```

By default, aliasing columns is set to 'true'.

#### Query.setDistinctMode(distinctMode)

##### (API)

By default, the resultant SQL query from the TQL script is not a SELECT DISTINCT... query. Since there is no DISTINCT keyword in TQL, the query must be set to 'DistinctMode' to produce the same effect. e.g.

```
query.setDistinctMode( true );
```

**Query.setParam(name, [value], [type], [list], [isNull])**

[\(API\)](#)

TQL has the ability to specify place holders for mapped paramters within the TQL by using a ':' in front of the mapped parameter's name. For example, the TQL:

```
from post.Post as Post where Post.name = :postName
```

Has the mapped parameter 'postName' that has to be set before the query can be run.

To set the mapped parameter a value, the method `Query.setParam` is used. For example to set the 'postName' mapped param to 'foo':

```
query.setParam("postName", "foo", "string");
```

The type argument can be set to the same 'type' attributes that can be found on *Property* elements in the object configuration xml file, i.e. string, numeric, boolean, date, binary, GUID, and UUID. By default, it is set to 'string'.

The 'list' argument tells the Query object that the value being set to the mapped parameter is actually a list of values, exactly like the 'list' attribute on `<cfqueryparam>`. By default this set to false.

Again, exactly like `<cfqueryparam>`, the 'null' argument takes a boolean that sets the value of the mapped parameter to NULL. By default, this is set to false.

**Query.setCacheEvaluation(cacheEvaluation)**

[\(API\)](#)

Transfer caches the SQL which is generated from your TQL query when executing queries to increase performance. This cached evaluation is not specific to the mapped parameter values that are used in the query. This cache is enabled by default.

When dynamically generating TQL queries that will *never* be executed again, or executing a large number of vastly different TQL queries you may disable the evaluation cache to save memory.

In general you should not modify this setting.

To disable the cache call:

```
query.setCacheEvaluation( false );
```

By default evaluation caching is set to true.

Prior to version 1.1 of Transfer, the default setting for cache evaluation is 'false',

not 'true'

## TQL Custom Tags

If you don't wish to use the TQL Query Object, you can also use the [TQL Custom Tags](#) that are also available, and provides the same functionality with a different style of syntax.

## Transfer Query Language Syntax

The Transfer Query Language is its own scripting language that was written in Java and ColdFusion. The following is the syntax that is allowed when using TQL.

Legend [ x ] - 'x' can appear 0 or 1 times  
 [ x ]\* - 'x' can appear 0 to infinity times.  
 ( x ) - 'x' must appear  
 x | y - 'x' or 'y'  
 "(" - a real (  
 ... - and so on...

### Identifiers

- *class* - A class name as defined in the Transfer Object Configuration file, i.e. 'post.Post' or 'system.Category'
- *property* - The name of a property on the class that it is associated with. i.e. 'post.Post.Name' 'Name' would be the property 'Name' on 'post.Post'.
- *columnAlias* - The name you wish the column, i.e. the *property* to be in the resultant query. Can be of the pattern [ a-z | A-Z ][ a-z | A-Z | 0-9 | \_ ]\* for Example: 'post.Post.name as postName', where 'postName' is the columnAlias.
- *classAlias* - The shorthand version of the *class* you wish this to be in the query. Can be of the pattern [ a-z | A-Z ][ a-z | A-Z | 0-9 | \_ ]\*. For example: 'post.Post as Post', where 'Post' is the classAlias.
- *mappedParameter* - An identifier of pattern :[ a-z | A-Z | 0-9 | \_ ]\*, that is set in the TQL Query object before being evaluated. For example: 'from post.Post where post.Post.name = :name' where ':name' is a mappedParameter.
- *subSelectStatement* - A sub-select TQL statement. Must start with a 'select...'. For example: 'from post.Post as Post where post.Name IN (select User.name from user.User as User)' where 'select User.name from user.User as User' is a subSelectStatement.

### Basic List Queries

```
from class
```

This produces a query that selects all the columns that defined in the *classes* object definitions that are present in the 'from' statement.

For Example: `from post.Post` will return all the columns as defined from the 'post.Post' object, from the table that 'post.Post' is configured to.

```
select * from class
```

This code, produces exactly the same result as above. The '\*' operator tells Transfer to resolve all properties of the given *classes* in the 'from' statement.

For example: `from post.Post` is exactly the same as `select * from post.Post`

### Selecting specific columns

```
select class.property [, class.property]* from class
```

To select specific properties from a query, you can specify them much like you would columns in a regular SQL expression.

For example: `select post.Post.postName, post.Post.postDate from post.Post` will select the 'postName' and the 'postDate' properties from the table that represents 'post.Post'.

### Aliasing column names

```
select class.property [as columnAlias] from class
```

It is also possible to manually alias columns to specific names.

For example: `select post.Post.name as title from post.Post where 'title'` is the alias, will alias the property 'postName' to 'title' in the query resultset.

### Aliasing columns names, while also selecting all

```
select class.property [as columnAlias] [, class.property [as columnAlias] | * ]* from class
```

Often you will only want to alias some columns in a TQL query, but still automatically select the rest. By including the '\*' operator into the 'select' part of the TQL script, this tells Transfer to include all properties into the query result.

For example, to select the 'name' property of 'post.Post' as 'title', but also to select all properties on 'post.Post', you would use: `select post.Post.name as title, * from post.Post`

### Aliasing Classes

```
from class as classAlias
```

It is also possible to alias *class* declarations in your From statements. This is useful for writing complicated Where statements, and also for ease of development.

For example, to give the class 'post.Post' the alias of 'Post' for the query, simply use: `from post.Post as Post`

### Selecting properties from aliased classes

```
select classAlias.property from class as classAlias
```

When aliasing *classes*, you **must** refer to them via their *classAlias*, and not via their

full *class*.

Thus, when selecting a specific property from an aliased *class*, it is imperative that you preface the statement with the *classAlias*

For example, when selecting the 'name' property on a 'post.Post' object in the TQL script which has been aliased to 'Post', the TQL must look like: `select Post.name from post.Post as Post`

### Where statements with operators

```
from class where class.property ( = | > | < | != | <> | <= | >= |
like ) ( mappedParameter | ( class | classAlias ).property )
```

TQL also has Where statements, in which you can place conditional statements to filter out data from your TQL queries.

For example, to get a list of Posts, with a filter on a 'like' condition: `from post.Post where post.Post like :name`. From there you are able to set the *mappedParameter* to a value using the TQL Query object, as described above.

### Where statement with an IN statement

```
from class where class.property [NOT] IN "(" ( mappedParameter | sub:
subSelectStatement ) ")"
```

Where statements can utilise IN statements, in the same way that SQL can. IN statements can either take a *mappedParameter*, or a TQL subselect statement.

For example, if your using a *mappedParameter* statement within your TQL script, it could look something like: `select from post.Post where post.Post.IDPost IN ( :idpostList )`. From there you would be setting the *mappedParameter* via the query object, most likely with the 'list' argument set to true.

You can also use a sub select within an IN statement. A sub select **must** have only a single property selected on it. For example: `from post.Post where post.Post.name IN ( select user.User.name from user.User )` will select all Posts with the same name as a User.

### Where statement with a NULL statement

```
from class where class.property IS [NOT] NULL
```

Where Statements can also contains conditions against database NULL values. For example, to get all Posts where the Name of the post is NULL: `from post.Post where post.Post.name IS NULL`

### Where statement with combinations of operators, NULL and IN statements

A combination of operator, NULL, and IN statements can be made, seperated by 'and' or 'or' boolean operators, as well as grouped with parenthesis.

For example:

```

from
  post.Post
where
  post.Post.createdDate >= :createDate
  AND
  (
    post.Post.body like :search
    OR
    post.Post.name like :search
  )

```

will select all Posts that were created past a certain date, and whose Name or Body is LIKE a certain search term.

### Where statements with aliased classes

```
from class as classAlias where classAlias.property ...
```

In the same way that selecting *properties* from an aliased *class*, in Where statement, if a *class* has been aliased, it **must** be referred to by its *classAlias*, rather than by its full name.

For example: `from post.Post as Post where Post.name != :name` we have used the *classAlias* of 'Post' as appropriate in the Where statement.

### Auto Joins

```
from class [as classAlias] [ join class [as classAlias] ]*
```

Since Transfer knows about the relationships between classes and therefore their corresponding tables, TQL can be used to automatically build SQL join statement for you.

For example, if we assume that 'post.Post' has a *onetomany* relationship to 'post.Comments', we could join all posts to their comments simply by using the script: `from post.Post join post.Comments`

To continue that, if we wanted the Post joined with its Comments and the User that created the Post, we could also have: `from post.Post join post.Comment join user.User`

### Specific Join

```
from class [as classAlias] join class [as classAlias] ON ( class | classAlias ).composite [ ( and | or ) ( class | classAlias ).composite ]*
```

Sometimes when there are multiple relationships between *classes*, you will want to only specify a particular relationship, or set of relationships that you wish to join on. This is possible by specifying the name of the composite element in a ON statement.

For example, if a 'post.Post' had relationships to two 'user.User's, one a the user that created it, and a user that the Post is about, we could specify that we only want the Author to be joined by saying:

```
from post.Post as Post
  join user.User
    ON Post.Author
```

where 'Author' is the name of the *manytoone* composite element on the 'post.Post' object.

## Manual Joins

```
from class [as classAlias] join class [as classAlias] ON ( class | c:
classAlias ).property ( ( = | < | ... )... | IN... | IS [NOT]
NULL... ) [ (AND|OR)...]
```

Depending on your object configuration file, there may be no relationship for the 'join' to follow, in that case it is possible to specify joins in almost exactly the same way you do in SQL, with the properties of the configured *classes*.

These conditions follow the same patterns as conditions in Where statements.

For example, to join all 'post.Post's with 'user.User' on their name property:

```
from
  post.Post as Post
  join
    user.User as Author
    ON Author.name = Post.name
```

## Multiple Joins

Multiple joins on a single query may be executed, including Joins of different types, for example:

```
from
  post.Post as Post
  join
  post.Comments
  join
  user.User as User
    on Post.Author
  join
  User.UserSetting as Setting
    on User.settingKey = Setting.key
```

## Left and Right Outer Joins

```
from class [[ left | right ] outer] join class
```

It is also possible to do outer joins with TQL. By default, TQL performs INNER JOIN statements on a 'join', but you can specify an outer join as well. By default, TQL performs a LEFT OUTER JOIN, but you can specify a RIGHT OUTER JOIN as required.

```
For example, to do an outer join on Users with their Posts: from user.User  
outer join post.Post
```

## Order By


```
from class [as classAlias] order by ( class | classAlias  
) .property [ asc | desc ] [ , ( class | classAlias ) .property [ asc | desc ] ]*
```

In the same way you can do Order By statements in SQL to order your rows in your query, you can also do so in TQL.

```
For example, to order a list of Users by their name: from user.User order by  
user.User.name
```

## Not supported in TQL

- Aggregate functions
- SQL Functions
- Subselect in FROM statements.
- Subselect in SELECT Column statements.

 Categories:

- [TQL](#)