Cachalot DB - version 2



SQL and LINQ guide

# Introduction

Cachalot DB implements an advanced query system similar to the usual one used by relational databases. There are limitations due to its distributed nature, and there are exciting features not available to classical SQL databases.

## General considerations on the SQL syntax:

* Almost everything is **case insensitive**. The only exception is the literal string values used with "=" or "<>" operators
* A laxist syntax:
  + Quotes are optional on string and dates
  + "\*" is optional too ("select \* from table" is equivalent to "select from table"
  + Both "<>" and "!=" can be used for NOT EQUAL

## By-design limitations:

* The only place where the parenthesis can be used is after an "IN" (or "NOT IN") operator. "AND" takes precedence on "OR" if both are present. This allows for high-speed SQL parsing client-side (significantly faster than LINQ expression processing) and enables some neat optimizations server side. OR clauses are executed in parallel on each node, and the query optimizer can be much faster if it processes only AND clauses.
* Only one ORDER BY argument is processed server-side. This choice allows for a much simpler index selection in the query optimizer.

## Inherent limitations in distributed databases

We can not implement generic JOIN operators in a distributed system as the two sides on the join should be present on the **same node for server-side processing**. In theory, a workaround would be to use partition keys to collocate related items in different collections. But this would be significant responsibility on the client application and risk for the system's evolution.

## Expressing literal values

|  |  |
| --- | --- |
| Data Type | Syntax |
|  |  |
| Numbers | use "." as a decimal separator |
| Strings | simple or double quotes are optional |
| Dates | the **yyyy-mm-dd** format is preferred, and quotes are optional |
| Boolean | true / false |
| Enums | should be expressed as numeric values |

# EQUAL

var withLinq = products.Where(p=>p.Brand **==** "REVLON").ToList();

var withSql = products.SqlQuery("select from products where brand **=** REVLON").ToList();

With bool value

var withLinq = salesDetails.Where(s => s.IsDelivered).ToList();

var withSql = salesDetails.SqlQuery("select from sales\_detail where isdelivered = **true**").ToList();

With enum value

var withLinq = salesDetails

.Where(s => s.Channel == **Model.Channel.Facebook**).ToList();

var withSql = salesDetails

.SqlQuery("select from sales\_detail where channel = **1**").ToList();

# NOT EQUAL

In SQL, we can use both "<>" and "!= ".

var withLinq = products.Where(p=>p.Brand **!=** "REVLON").ToList();

var withSql1 = products.SqlQuery("select from products where brand **!=** REVLON").ToList();

var withSql2 = products.SqlQuery("select from products where brand **<>** REVLON").ToList();

# COMPARISON

var withLinq = salesDetails.Where(s =>

s.Date **>** new DateTime(2020, 1, 1) &&

s.Date **<=** new DateTime(2020, 1, 15))

.ToList();

var withSql = salesDetails.SqlQuery("select from sales\_detail where date > 2020-01-01 and date <= 2020-01-15").ToList();

When possible, the optimizer groups comparison operators as a "range operator," significantly improving index usage.

# IN

var brands = new[] {"REVLON", "Advanced Clinicals"};

var withLinq = products.Where(p=>brands.**Contains**(p.Brand)).ToList();

var withSql = products.SqlQuery("select from products where brand **in** (REVLON, Advanced Clinicals)").ToList();

# NOT IN

var brands = new[] {"REVLON", "DOVE"};

var withLinq = products.Where(p=>**!**brands.**Contains**(p.Brand)).ToList();

var withSql = products

.SqlQuery("select from products where brand **not** **in** (REVLON, DOVE)").ToList();

# CONTAINS

This operator is an extension of the usual SQL syntax. The left side of the operator refers to a collection property.

var withLinq = products

.Where(p=>p.Categories.**Contains**("lip stick")).ToList();

var withSql = products.SqlQuery("select from products where categories **contains** lip stick").ToList();

# NOT CONTAINS

var withLinq = products.Where(p=>**!**p.Categories.**Contains**("soap"))

.ToList();

var withSql = products.SqlQuery("select from products where categories **not contains** soap").ToList();

# STRING OPERATORS

var withLinq = products.Where(p=>p.Brand.**Contains**("clinical")).ToList();

var withSql = products.SqlQuery("select from products where brand **like** **%**clinical%").ToList();

var withLinq = products.Where(p=>p.Brand.**StartsWith**("advanced")).ToList();

var withSql = products.SqlQuery("select from products where brand **like** advanced**%**").ToList();

var withLinq = products.Where(p=>p.Brand.**EndsWith**("clinicals")).ToList();

# var withSql = products.SqlQuery("select from products where brand **like** **%**clinicals").ToList();

These string operators are **case insensitive**.

# PROJECTIONS

In LINQ, we have two different use-cases:

* Selecting a single scalar property returns a collection of this property type
* Selecting a collection property or multiple properties (all need to be server-side visible) returns a collection of objects containing this property (or properties). The type of object in the collection is an anonymous class containing only the selected properties.

When using SQL, a collection of the original type of the **DataSource** is returned, but **only the selected properties are filled**.

In both cases, the server sends through the network only the selected properties.

This example will return a collection of string:

var withLinq = products

.Where(p=>p.Brand == "REVLON").**Select**(p=>p.**Name**).ToList();

This one will return a collection of **Product** with only the Name property filled:

var withSql = products

.SqlQuery("select Name from products where brand = REVLON")

.ToList();

Example with only a collection property selected:

var withLinq = products.Where(p=>p.Brand == "REVLON").**Select**(p=>new { p.**Categories**}).ToList();

var withSql = products.SqlQuery("select **categories** from products where brand = REVLON").ToList();

Example with multiple properties selected

var withLinq = products.Where(p=>p.Brand == "REVLON").**Select**(p=>new {p.**Name**, p.**ScanCode**}).ToList();

var withSql = products.SqlQuery("select **name**, **scancode** from products where brand = REVLON").ToList();

# DISTINCT

With single property:

var withLinq = products.Select(p=>p.Brand).**Distinct**().ToList();

var withSql = products.SqlQuery("select **distinct** brand from products").ToList();

With multiple properties

var withLinq = products.Select(p=>new {p.Brand, p.Name}).**Distinct**().ToList();

var withSql = products.SqlQuery("select **distinct** brand, name from products").ToList();

# TAKE

var withLinq = salesDetails

.Where(s=>s.IsDelivered && s.Amount > 80).**Take**(10).ToList();

var withSql = salesDetails.SqlQuery("select from sales\_detail where isdelivered = true and amount > 80 take 10").ToList();

ORDER BY

Ascending

var withLinq = salesDetails.Where(s=>s.IsDelivered && s.Amount > 80).**OrderBy**(s=> s.**Amount**).ToList();

var withSql = salesDetails.SqlQuery("select from sales\_detail where isDelivered = true and amount > 80 order by **AMOUNT**").ToList();

Descending

var withLinq = salesDetails

.Where(s=>s.IsDelivered && s.Amount > 80)

.**OrderByDescending**(s=> s.**Amount**)

.ToList();

var withSql = salesDetails.SqlQuery("select from sales\_detail where isdelivered = true and amount > 80 **order by amount descending**")

.ToList();