LiteDB

Fork me on GitHub

Expressions

Expressions are path or formulas to access and modify the data inside a document. Based on the concept of JSON path (http://goessner.net/articles/JsonPath/), LiteDB supports a similar syntax to navigate inside a document.

`BsonExpression` is the class that parses a path expression and compiles it into a Linq Expression to be evaluated by LiteDB.

- Path starts with `$` : `$.Address.Street`, where `$` represents the root document. The `$` symbol are optional and default in document navigation (`Address.Street` works too)
- Int values are defined by `[0-9]*` : `123`
- Double values are defined by `[0-9].[0-9]` : `123.45`
- Strings are represented with a single/double quote: `'Hello World'`
- Null is represented by `null`
- Bool is represented using `true` or `false` keywords.
- Document starts with `{ key1: <value|expression>, key2: ... }`
- Arrays are represented with `[<value|expression>, <value|expression>, ...]`
- Functions are represented with `FUNCTION_NAME(par1, par2, ...)` : `LOWER($.Name)`

Examples:

- `$.Price`
- `$.Price + 100`
- `SUM($.Items[*].Price)`

```
var expr = new BsonExpression("SUM($.Items[*].Unity * $.Items[*].Price)");
var total = expr.Execute(doc, true).First().AsDecimal;
```

Expressions can be used in many ways:

- Creating an index based on an expression:
  - `collection.EnsureIndex("idx_name", "LOWER($.Name)", false)`
  - `collection.EnsureIndex(x => x.Name.ToLower())`
- Querying documents inside a collection based on expression (full scan search)
  - `collection.Find("SUBSTRING($.Name, 0, 1) = 'T'")`
- Update using SQL syntax
  - `UPDATE customers SET Name = LOWER($.Name) WHERE _id = 1`
- Creating new document result in SELECT shell command
  - `SELECT { upper_titles: ARRAY(UPPER($.Books[*].Title)) } WHERE $.Name LIKE "John%"`
- Querying documents using the SQL syntax
  - `SELECT $.Name, $.Phones[@.Type = "Mobile"] FROM customers`

Path

- `$` - Root
- `$.Name` - `Name` field
- `$.Name.First` - `First` field from `Name` subdocument
- `$.Books` - Returns an array of books
- `$.Books[0]` - Returns the first book inside Books array
- `$.Books[*]` - Returns every book inside Books
- `$.Books[*].Title` Returns the title from every book in Books
- `$.Books[-1]` - Returns the last book inside Books array

Path also supports expressions to filter child nodes

- `$.Books[@.Title = 'John Doe']` - Returns all books where `Title` is `'John Doe'`
- `$.Books[@.Price > 100].Title` - Returns all titles where `Price` is greater than `100`

Inside an array, `@` acts as a sub-iterator, pointing to the current sub-document. It's possible use functions inside expressions too:

- `$.Books[SUBSTRING(LOWER(@.Title), 0, 1) = 't']` - Returns all books whose `Title` starts with `'T'` or `'t'`.

Difference between `$` and `*`

In SQL query (at `SELECT` segment) is possible use both `$` and `*`. But they represent different things.

- `$` represent current document root. When use `$` (or ommited, because this symbol are optional and are default) you are referencing about root current document.
- `*` represent all grouped documents. It's not more about a single document but all documents in group (or in query result). Used when `GROUP BY` are present or when you want return a single value in query (like `SELECT COUNT(*) FROM customers`).

If you use `SELECT $ FROM customers` you will get `IEnumerable<BsonDocument>` result (`N` documents). If you use `SELECT * FROM customers` you will get a single value, a `BsonArray` with all documents result inside. Be carful because if your resultset are big you are creating a very large single array in memory.

Functions

Functions are used to manipulate data in expressions. A few examples will be provided for each category of functions. For a complete list of functions, check the API documentation.

Aggregate Functions

Aggregate functions take an array as input and return a single value.

- `COUNT(arr)` - Returns the number of elements in the array `arr`
- `AVG(arr)` - Returns the average value in the array `arr`
- `LAST(arr)` - Returns the last element in the array `arr`

DataType Functions

DataType functions provide explicit data type conversion.

- `STRING(expr)` - Returns the result of `expr` converted to string
- `INT32(expr)` - Tries to convert the result of `expr` to an `Int32`, returning `null` if not possible
- `DATETIME(expr)` - Tries to convert the result of `expr` to a `DateTime`, returning `null` if not possible

Date Functions

- `YEAR(date)` - Returns the year value from `date`
- `DATEADD('year', 3, date)` - Returns a new date with 3 years added to date
- `DATEDIFF('day', dateStart, dateEnd)` - Returns the difference in days between `dateEnd` and `dateStart`

Math Functions

- `ABS(num)` - Returns the absolute value of `num`
- `ROUND(num, digits)` - Returns `num` rounded to `digits` digits
- `POW(base, exp)` - Returns `base` to the power of `exp`

String Functions

- `UPPER(str)` - Returns `str` in uppercase
- `TRIM(str)` - Returns a new string without leading and trailing white spaces
- `REPLACE(str, old, new)` - Returns a new string with every ocurrence of `old` in `str` replaced by `new`

High-Order Functions

High-Order functions take an array and a lambda expression that is applied to every document in the array. Use `@` symbol to represent inner looped value.

- `MAP(arr => expr)` returns a new array with the map expression applied to each element
  - `MAP([1,2,3] => @*2)` returns `[2,4,6]`
  - `MAP([{a:1, b:2}, {a:3, b:4}] => @.a)` returns `[1,3]`
- `FILTER(arr => expr)` returns a new array containing only the elements for which the filter expression returns `true`
  - `FILTER([1,2,3,4,5] => @ > 3)` returns `[4,5]`

  - `FILTER([{a:1, b:2}, {a:2}] => @.b != null)` returns `[{a:1, b:2}]`

- `SORT(arr => expr)` returns a new array sorted by the result of `expr` in ascending order - `SORT([3,2,5,1,4] => @)` returns `[1,2,3,4,5]` - `SORT([{a:2}, {a:1, b:2}] => @.a)` returns `[{a:1, b:2}, {a:2}]`

- `SORT(arr => expr, order)` returns a new array sorted by the result of `expr` with the order defined by `order` (ascending if `order` is `1` or `'asc'`, descending if `order` is `-1` or `'desc'`) - `SORT([3,2,5,1,4] => @, 'desc')` returns `[5,4,3,2,1]` - `SORT([{a:1, b:2}, {a:2}] => @.a, -1)` returns `[{a:2}, {a:1, b:2}]`

Misc Functions

- `JSON(str)` - Takes a string representation of a JSON and returns a parsed `BsonValue` containing the document
- `CONCAT(arr1, arr2)` - Returns a new array containg the concatenation between arrays `arr1` and `arr2`
- `RANDOM(min, max)` - Returns a random `Int32` between `min` and `max`