LiteDB

Indexes

LiteDB improves search performance by using indexes on document fields or expressions. Each index storess the value of a specific expression ordered by the value (and type). Without an index, LiteDB must execute a query using a full document scan. Full document scans are inefficient because LiteDB must deserialize every document in the collection.

Index Implementation

Indexes in LiteDB are implemented using **Skip lists**. Skip lists are double linked sorted list with up to 32 levels. Skip lists are very easy to implement (only 15 lines of code) and statistically balanced.

Insert and search operations have an average complexity of $O(\log n)$. This means that in a collection with 1 million documents, a search operation over an indexed expression will take about 13 steps to find the desired document. If you want to know more about skip lists, check this great video.

Given that collections are schema-less, it is possible for an expression to return different data types when applied over different documents. In such cases, the documents will be ordered by the type returned by the indexing expression, according to the table below:

BSON Type	Order
MinValue	1
Null	2
Int32, Int64, Double, Decimal	3
String	4
Document	5
Array	6
Binary	7
ObjectId	8
Guid	9
Boolean	10
DateTime	11
MaxValue	12

- When comparing documents, the values in the key-value pairs are compared in the order that they appear until a tie is broken.
- When comparing arrays, every position is compeared until a tie is broken.
- All numeric values are converted to `Decimal` before comparison.

Primary key (= auto id)

By default, an index over `_id` is created upon the first insertion.

EnsureIndex()

Indexes are created via `EnsureIndex`. This method creates the index if it does not exist and does nothing if already exists.

An index can be created over any valid `BsonExpression`.

```
{
    _id: 1,
    Address:
    {
        Street: "Av. Protasio Alves, 1331",
        City: "Porto Alegre",
        Country: "Brazil"
    }
}
```

- You can use `EnsureIndex("Address")` to create an index to all `Address` embedded document
- Or `EnsureIndex("Address.Street")` to create an index on `Street` using dotted notation
- Indexes are executed as `BsonDocument` fields. If you are using a custom `ResolvePropertyName` or `[BsonField]` attribute, you must use your document field name and not the property name. See Object Mapping.
- You can use a lambda expression to define an index field in a strongly typed collection: `EnsureIndex(x => x.Name)`
- Indexes are identified by an unique name

MultiKey Index

When you create an index in a array type field, all values are included on index keys and you can search for any value.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string[] Phones { get; set; }
}

var customers = db.GetCollection<Customer>("customers");

customers.Insert(new Customer { Name = "John", Phones = new string[] { "1", "2",
customers.Insert(new Customer { Name = "Doe", Phones = new string[] { "1", "8" }

customers.EnsureIndex(x => x.Phones);

var result = customers.Find(x => x.Phones.Contains("1")); // returns both docume
```

Expressions

It is now possible to create an index based on a expression execution with multikey values support. With this, you can index any kind of information that is not directly the value of a field.

- `collection.EnsureIndex("Name", "LOWER($.Name)")`
- `collection.EnsureIndex("Total", "SUM($.Items[*].Price)")`
- `collection.EnsureIndex("CheapBooks", "LOWER($.Books[@.Price < 20].Title)")`

See Expressions for more details about expressions.

Limitations

- Even if multiple indexed expressions are used on a query, only one of the indexes is used, with the remaining expressions being filtered using a full scan.
- Index keys must have at most 1023 bytes
- Up to 255 indexes per collections, including the `_id` primary key, but limited to 8096 bytes for index definition. Each index uses: `41` bytes + `LEN(name) + LEN(expression)`