NOSQL Query Language

This query language is inspired by MongoDB.

A query consists of these parts:

1. fields to be extracted
2. table to extract the records from
3. expression for filtering the table rows
4. groupby - fields to group the data under
5. aggregate functions to be applied to columns in fields
6. orderby - fields to order the return data by
7. limit - an integer number of records to return.

Only the table and expression parameters are mandatory.

The NoSQL queries are then constructed into a SQL query of the following form:

```
SELECT fields with aggregation
FROM table
WHERE expression
GROUP BY groupby
ORDER BY orderby
LIMIT limit
```

NoSQL queries are constructed using JSON objects. Below is an example:

```
{
  object: String,
  q: Expression,
  fields: Array of String,
  groupBy: Array of String,
  aggregation: Object mapping fields to aggregate functions
}
```

For example, the shortest query you can write would be:

```
{ "object": "String", "q": "Expression" }
```

This NoSQL object is converted into:

```
SELECT *
FROM table
WHERE query
```

Examples

This simple query retrieves the name and salary of all employees in position of "Sales Manager":

```
{
    "object": "employees",
    "q": {
        "position" : "Sales Manager"
    },
    "fields": ["name", "salary"]
}
```

Queries can also be used to compare an object's fields to constant values using common comparison operators. For example, to retrieve all fields for all employees under the age of 25, you can use the following query:

```
{
    "object": "employees",
    "q": {
        "age": { "$lt" : 25 }
    }
}
```

Expressions

An expression can be either an AND expression, an OR expression, or a UNION query.

AND expressions

An AND expression is a conjunction of conditions on fields. An AND expression is a JSON of the form `{ A: condition, B: condition, ... }`

For example, to retrieve all employees that are 25-years-old, a Sales manager, AND live in Boston, you could use the following query:

```
{ "position": "Sales Manager", "age" : { "$lt" : 25 }, "city": "Boston" }
```

OR expressions

An OR expression is a disjunction of conditions, `{ $or: [ Expression1, Expression2, ...   ] }`

For example, use the following query to find all offices that are either larger than 30 employees, or located in Palo Alto:

```
{ "$or": [ { "num_employees": { "$gt": 30 } }, { "location": "Palo Alto" }  ]  }
```

UNION queries

A UNION query is a union of the results of queries: `{ $union: [ Query1, Query2, ... ] }`. For example:

```
{
    "$union":   [
        {
            "object" : "Employees",
            "q" : {
                "$or" : [
                    {
                        "Budget" : {
                            "$gt" : 20
                        }
                    },
                    {
                        "Location" : {
                            "$like" :  "Palo Alto"
                        }
                    }
                ]
            },
            "fields": ["Location", "country"]
        },
        {
            "object" : "Person",
            "q" : {
                "name": "john"
            },
            "fields": ["City", "country"],
            "limit": 11
        }
    ]
}
```

Conditions on Fields

A condition on a field is a predicate that can perform one of the following actions:

1. Test equality of field to a constant value, e.g. `{ A: 6 }` => `Is` A `equal to 6?`
2. Compare a field using a comparison operator, e.g. `{ A: { $gt: 8 }}` => `Is` A `greater than 8?`. The set of comparison operators is quite extensive and includes: `$lte, $lt, $gte, $gt, $eq, $neq, $not`
3. Test if the value of the field is IN or NOT IN the result of a sub-query.
4. Test for the negation of a comparison. For example, to test if the location field is not Boston, we can do:

```
{ "$not": { "location" : "Boston" }}
```

Sub Queries

The following sub-query retrieves the department ID of each department in New York:

```
{ "object": "department", "q": { "city" : "New York" }, "fields" : ["id"]}
```

Using this subquery, we can now test a new field - dept_id - with respect to the results of the subqeury. We simply use the `$in` operator, and the query, as follows:

```
{
    "dept_id": {
        "$in": {
            { "object": "department",
                "q": { "city" : "New York" },
                "fields" : ["id"]
            }
        }
    }
}
```

This technique relies upon retrieving a single field from the sub-query. Using more than one field would prove more complex.

We can now use this sub-query as a part of a larger query retrieving all employees employed in departments that are located in New York. In this example, the `deptId` field is a reference field referring the employees table to the department table:

```
{
    "object": "employees",
    "q" : {
        "deptId" : {
            "$in": {
                "object": "department",
                "q": {
                    "city" : "New York"
                },
                "fields" : ["id"]
            }
        }

    }
}
```

If we wanted to look at a more complex query, we could modify this a bit. Let's say we wanted to retrieve all employees whose department is located in New York, but the employee is located in Boston. To accomplish this, we use an AND expression to combine the two conditions:

```
{
    "object": "employees",
    "q" : {
        "deptId":
        {
            "$in": {
                "object": "department",
                "q": {
                    "city" : "New York"
                },
                "fields" : ["id"]
            }
        },
        "location": "Boston"
    }
}
```


Conditions on Fields

Formally, a condition on a field is a key-value expression of the form:

```
    Key : ValueExpression
```

Where the fields are defined as follows:

- Key - name of the field
- ValueExpression - An expression which has one of the following forms:
    1. Constant - is the field value equal to the constant
    2. Comparison with a comparison operator to a constant
    3. Inclusion or exclusion in result of a sub query
    4. Negation of another comparison

Negation may sometimes be swapped for comparison. For example, to test if the location field is not equal to Paris, we can use negation as follows:

```
{ $not: { location : "Paris" }}
```

Or we can also use a not-equal operator:

```
{ location: { $neq: "Paris" }}
```

Group By Queries

A group by query aggregates on fields, and then applies aggregation operators to the specified fields. For instance, to group by `Country`, and then concatenate the `Location` field, use the following example code:

```
{
    "object" : "Employees",
    "q" : {
        "$or" : [
            {
                "Budget" : {
                    "$gt" : 20
                }
            },
            {
                "Location" : {
                    "$like" :  "Palo Alto"
                }
            }
        ]
    },
    "fields": ["Location", "Country"],
    "order": [["Budget", "desc"]],
    "groupBy": ["Country"],
    "aggregate": {
        "Location": "$concat"
    }
}
```

Algorithm to Generate SQL from JSON Queries

The algorithm transforms from JSON to SQL using a top-down transformation.

Usage

```
transformJson(json, sqlSchema, isFilter, callback)
```

The parameters are:

1. `json` - JSON query or filter
2. `sqlSchema` - JSON schema of database
3. `isFilter` - boolean - true if `json` is a filter
4. `callback` - `function(err, result)`, called upon completion

The result is a structure with the following fields:

```
{
    str: <SQL statement for query>,
    select: <select clause>,
    from: <from clause>,
    where: <where clause>,
    group: <group by clause>,
    order: <order by clause>,
    limit: <limit clause>
}
```

Escaping

All constants appearing in the JSON query are escaped when transformed into SQL.

Filters

You also have the ability to mark a particular NoSQL query as a filter. This allows you to use variables in your query, which are populated on the server side from either parameters sent in with the filter, or from database data in your system. Variables take the form of:

```
{{<variable name>}}
```

Variables should be enclosed in quotes (e.g. `'{{variable_name}}'` instead of `{{variable_name}}`) so that the final objet sent to the server can be marked as valid JSON.

Variables are not escaped when used as part of a filter or query - only constants can be escaped by Backand. With this in mind, you want to make sure that variables tied directly to u properly sanitized before being sent to the back-end. The SQL statement generated for the filter

object will include the variables you provide verbatims. The variables will be substituted for the equivalent values prior to the execution of the query.