Querying collections in a SQL-like style

Groovy's `groovy-ginq` module provides a higher-level abstraction over collections. It could perform queries against in-memory collections of objects in a SQL-like style. Also, querying XML, JSON, YAML, etc. could also be supported because they can be parsed into collections. As GORM and jOOQ are powerful enough to support querying DB, we will cover collections first.

GINQ a.k.a. Groovy-Integrated Query

GINQ is a DSL for querying with SQL-like syntax, which consists of the following structure:

```
GQ, i.e. abbreviation for GINQ
|__ from
|   |__ <data_source_alias> in <data_source>
|__ [innerjoin/leftjoin/rightjoin/fulljoin/crossjoin]*
|   |__ <data_source_alias> in <data_source>
|   |__ on <condition> ((&& | ||) <condition>)* (NOTE: `crossjoin` does not need `on`
clause)
|__ [where]
|   |__ <condition> ((&& | ||) <condition>)*
|__ [groupby]
|   |__ <expression> [as <alias>] (, <expression> [as <alias>])*
|   |__ [having]
|       |__ <condition> ((&& | ||) <condition>)*
|__ [orderby]
|   |__ <expression> [in (asc|desc)] (, <expression> [in (asc|desc)])*
|__ [limit]
|   |__ [<offset>,] <size>
|__ select
    |__ <expression> [as <alias>] (, <expression> [as <alias>])*
```

NOTE	`[]` means the related clause is optional, `*` means zero or more times, and `+` means one or more times. Also, the clauses of GINQ are order sensitive, so the order of clauses should be kept as the above structure

As we could see, the simplest GINQ consists of a `from` clause and a `select` clause, which looks like:

```
from n in [0, 1, 2]
select n
```

NOTE	*ONLY ONE* `from` clause is required in GINQ. Also, GINQ supports multiple data sources through `from` and the related joins.

As a DSL, GINQ should be wrapped with the following block to be executed:

```
GQ { /* GINQ CODE */ }
```

For example,

```
def numbers = [0, 1, 2]
assert [0, 1, 2] == GQ {
    from n in numbers
    select n
}.toList()
```

```
import java.util.stream.Collectors

def numbers = [0, 1, 2]
assert '0#1#2' == GQ {
    from n in numbers
    select n
}.stream()
    .map(e -> String.valueOf(e))
    .collect(Collectors.joining('#'))
```

And it is strongly recommended to use `def` to define the variable for the result of GINQ execution, which is a `Queryable` instance that is lazy.

```
def result = GQ {
    /* GINQ CODE */
}
def stream = result.stream() // get the stream from GINQ result
def list = result.toList() // get the list from GINQ result
```

WARNING	Currently GINQ can not work well when STC is enabled.

GINQ Syntax

Data Source

The data source for GINQ could be specified by `from` clause. Currently GINQ supports `Iterable`, `Stream`, array and GINQ result set as its data source:

`Iterable` Data Source

```
from n in [1, 2, 3] select n
```

`Stream` Data Source

```
from n in [1, 2, 3].stream() select n
```

Array Data Source

```
from n in new int[] {1, 2, 3} select n
```

GINQ Result Set Data Source

```
def vt = GQ {from m in [1, 2, 3] select m}
assert [1, 2, 3] == GQ {
    from n in vt select n
}.toList()
```

Projection

The column names could be renamed with `as` clause:

```
def result = GQ {
    from n in [1, 2, 3]
    select Math.pow(n, 2) as powerOfN
}
assert [[1, 1], [4, 4], [9, 9]] == result.stream().map(r -> [r[0], r.powerOfN]).
toList()
```

NOTE	The renamed column could be referenced by its new name, e.g. `r.powerOfN`. Also, it could be referenced by its index, e.g. `r[0]`

```
assert [[1, 1], [2, 4], [3, 9]] == GQ {
    from v in (
        from n in [1, 2, 3]
        select n, Math.pow(n, 2) as powerOfN
    )
    select v.n, v.powerOfN
}.toList()
```

NOTE	`select P1, P2, ⋯, Pn` is a simplifed syntax of `select new NamedRecord(P1, P2, ⋯, Pn)` when and only when `n` >= 2. Also, `NamedRecord` instance will be created if `as` clause is used. The values stored in the `NamedRecord` could be referenced by their names.

Construct new objects as column values:

```
@groovy.transform.EqualsAndHashCode
class Person {
    String name
    Person(String name) {
        this.name = name
    }
}
def persons = [new Person('Daniel'), new Person('Paul'), new Person('Eric')]
assert persons == GQ {
    from n in ['Daniel', 'Paul', 'Eric']
    select new Person(n)
}.toList()
```

Filtering

```
from n in [0, 1, 2, 3, 4, 5]
where n > 0 && n <= 3
select n * 2
```

In

```
from n in [0, 1, 2]
where n in [1, 2]
select n
```

```
from n in [0, 1, 2]
where n in (
    from m in [1, 2]
    select m
)
select n
```

```
import static groovy.lang.Tuple.tuple
assert [0, 1] == GQ {
    from n in [0, 1, 2]
    where tuple(n, n + 1) in (
        from m in [1, 2]
        select m - 1, m
    )
    select n
}.toList()
```

Not In

```
from n in [0, 1, 2]
where n !in [1, 2]
select n
```

```
from n in [0, 1, 2]
where n !in (
    from m in [1, 2]
    select m
)
select n
```

```
import static groovy.lang.Tuple.tuple
assert [2] == GQ {
    from n in [0, 1, 2]
    where tuple(n, n + 1) !in (
        from m in [1, 2]
        select m - 1, m
    )
    select n
}.toList()
```

Exists

```
from n in [1, 2, 3]
where (
    from m in [2, 3]
    where m == n
    select m
).exists()
select n
```

Not Exists

```
from n in [1, 2, 3]
where !(
    from m in [2, 3]
    where m == n
    select m
).exists()
select n
```

Joining

More data sources for GINQ could be specified by join clauses.

```
from n1 in [1, 2, 3]
join n2 in [1, 3] on n1 == n2
select n1, n2
```

NOTE	`join` is preferred over `innerjoin` and `innerhashjoin` as it has better readability, and it is smart enough to choose the correct concrete join(i.e. `innerjoin` or `innerhashjoin`) by its `on` clause.

```
from n1 in [1, 2, 3]
innerjoin n2 in [1, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
leftjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [2, 3, 4]
rightjoin n2 in [1, 2, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
fulljoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
crossjoin n2 in [3, 4, 5]
select n1, n2
```

hash join is especially efficient when data sources contain lots of objects

```
from n1 in [1, 2, 3]
innerhashjoin n2 in [1, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
lefthashjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [2, 3, 4]
righthashjoin n2 in [1, 2, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
fullhashjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

NOTE Only binary expressions(==, &&) are allowed in the on clause of hash join

Grouping

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, count(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
having n >= 3
select n, count(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
having count() < 3
select n, count()
```

The group columns could be renamed with as clause:

```
from s in ['ab', 'ac', 'bd', 'acd', 'bcd', 'bef']
groupby s.size() as length, s[0] as firstChar
select length, firstChar, max(s)
```

```
from s in ['ab', 'ac', 'bd', 'acd', 'bcd', 'bef']
groupby s.size() as length, s[0] as firstChar
having length == 3 && firstChar == 'b'
select length, firstChar, max(s)
```

Aggregate Functions

GINQ provides some built-in aggregate functions, e.g. `count`, `min`, `max`, `sum`, `avg`, `median` and the most powerful function `agg`.

> **NOTE**
>
> `count(…)`, `min(…)`, `max(…)`, `avg(…)` and `median(…)` just operate on non-`null` values, and `count()` is similar to `count(*)` in SQL.

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, count()
```

```
from s in ['a', 'b', 'cd', 'ef']
groupby s.size() as length
select length, min(s)
```

```
from s in ['a', 'b', 'cd', 'ef']
groupby s.size() as length
select length, max(s)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, sum(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, avg(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, median(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, agg(_g.stream().map(r -> r.n).reduce(BigDecimal.ZERO, BigDecimal::add))
```

NOTE	`_g` is an implicit variable for `agg` aggregate function, it represents the grouped `Queryable` object and its record(e.g. `r`) could reference the data source by alias(e.g. `n`)

Also, we could apply the aggregate functions for the whole GINQ result, i.e. no `groupby` clause is needed:

```
assert [3] == GQ {
    from n in [1, 2, 3]
    select max(n)
}.toList()
```

```
assert [[1, 3, 2, 2, 6, 3, 3, 6]] == GQ {
    from n in [1, 2, 3]
    select min(n), max(n), avg(n), median(n), sum(n), count(n), count(),
            agg(_g.stream().map(r -> r.n).reduce(BigDecimal.ZERO, BigDecimal::add))
}.toList()
```

Sorting

```
from n in [1, 5, 2, 6]
orderby n
select n
```

NOTE	`in asc` is optional when sorting `in` ascending order

```
from n in [1, 5, 2, 6]
orderby n in asc
select n
```

```
from n in [1, 5, 2, 6]
orderby n in desc
select n
```

```
from s in ['a', 'b', 'ef', 'cd']
orderby s.length() in desc, s in asc
select s
```

```
from s in ['a', 'b', 'ef', 'cd']
orderby s.length() in desc, s
select s
```

Pagination

`limit` is similar to the `limit` clause of MySQL, which could specify the `offset`(first argument) and `size`(second argument) for paginating, or just specify the only one argument as `size`

```
from n in [1, 2, 3, 4, 5]
limit 3
select n
```

```
from n in [1, 2, 3, 4, 5]
limit 1, 3
select n
```

Nested GINQ

Nested GINQ in `from` clause

```
from v in (
    from n in [1, 2, 3]
    select n
)
select v
```

Nested GINQ in `where` clause

```
from n in [0, 1, 2]
where n in (
    from m in [1, 2]
    select m
)
select n
```

```
from n in [0, 1, 2]
where (
    from m in [1, 2]
    where m == n
    select m
).exists()
select n
```

Nested GINQ in `select` clause

```
assert [null, 2, 3] == GQ {
    from n in [1, 2, 3]
    select (
        from m in [2, 3, 4]
        where m == n
        limit 1
        select m
    )
}.toList()
```

NOTE	It's recommended to use `limit 1` to restrict the count of sub-query result because `TooManyValuesException` will be thrown if more than one values returned

We could use `as` clause to name the sub-query result

```
assert [[1, null], [2, 2], [3, 3]] == GQ {
    from n in [1, 2, 3]
    select n, (
        from m in [2, 3, 4]
        where m == n
        select m
    ) as sqr
}.toList()
```

Window Functions

Window can be defined by `partitionby`, `orderby` and `rows`:

```
over(
    [partitionby <expression> (, <expression>)*]
    [orderby <expression> (, <expression>)*
       [rows <expression>, <expression>]]
)
```

Also, GINQ provides some built-in window functions, e.g. `rowNumber`, `rank`, `denseRank`, `lead`, `lag`,

`firstValue`, `lastValue`, `min`, `max`, `count`, `sum`, `avg`, `median`, etc.

`rowNumber`

```
assert [[2, 1], [1, 0], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (rowNumber() over(orderby n))
}.toList()
```

NOTE The parentheses around the window function is required.

`rank` **and** `denseRank`

```
assert [['a', 1, 1], ['b', 2, 2], ['b', 2, 2],
        ['c', 4, 3], ['c', 4, 3], ['d', 6, 4],
        ['e', 7, 5]] == GQ {
    from s in ['a', 'b', 'b', 'c', 'c', 'd', 'e']
    select s,
        (rank() over(orderby s)),
        (denseRank() over(orderby s))
}.toList()
```

`lead` **and** `lag`

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n in asc))
}.toList()
```

```
assert [['a', 'bc'], ['ab', null], ['b', 'a'], ['bc', 'ab']] == GQ {
    from s in ['a', 'ab', 'b', 'bc']
    select s, (lead(s) over(orderby s.length(), s in desc))
}.toList()
```

```
assert [['a', null], ['ab', null], ['b', 'a'], ['bc', 'ab']] == GQ {
    from s in ['a', 'ab', 'b', 'bc']
    select s, (lead(s) over(partitionby s.length() orderby s.length(), s in desc))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lag(n) over(orderby n))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lag(n) over(orderby n in desc))
}.toList()
```

```
assert [['a', null], ['b', 'a'], ['aa', null], ['bb', 'aa']] == GQ {
    from s in ['a', 'b', 'aa', 'bb']
    select s, (lag(s) over(partitionby s.length() orderby s))
}.toList()
```

```
assert [[2, 3, 1], [1, 2, null], [3, null, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n)), (lag(n) over(orderby n))
}.toList()
```

The offset can be specified other than the default offset 1:

```
assert [[2, null, null], [1, 3, null], [3, null, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n, 2) over(orderby n)), (lag(n, 2) over(orderby n))
}.toList()
```

The default value can be returned when the index specified by offset is out of window, e.g. 'NONE':

```
assert [[2, 'NONE', 'NONE'], [1, 3, 'NONE'], [3, 'NONE', 1]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n, 2, 'NONE') over(orderby n)), (lag(n, 2, 'NONE') over(orderby n
))
}.toList()
```

`firstValue` and `lastValue`

```
assert [[2, 1], [1, 1], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows -1, 1))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, 1))
}.toList()
```

```
assert [[2, 2], [1, 1], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows 0, 1))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows -2, -1))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -2, -1))
}.toList()
```

```
assert [[2, 3], [1, 3], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows 1, 2))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows 1, 2))
}.toList()
```

NOTE `0` in the `rows` clause is equivalent to SQL's `CURRENT ROW`, negative means `PRECEDING`, positive means `FOLLOWING`

```
assert [[2, 2], [1, 1], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, 0))
}.toList()
```

```
assert [[2, 1], [1, 1], [3, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows null, 1))
}.toList()
```

NOTE `null` used as the lower bound of `rows` clause is equivalent to SQL's `UNBOUNDED PRECEDING`

```
assert [[2, 3], [1, 3], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, null))
}.toList()
```

NOTE `null` used as the upper bound of `rows` clause is equivalent to SQL's `UNBOUNDED FOLLOWING`

```
assert [['a', 'a', 'b'], ['aa', 'aa', 'bb'], ['b', 'a', 'b'], ['bb', 'aa', 'bb']] ==
GQ {
    from s in ['a', 'aa', 'b', 'bb']
    select s, (firstValue(s) over(partitionby s.length() orderby s)),
            (lastValue(s) over(partitionby s.length() orderby s))
}.toList()
```

`min`, `max`, `count`, `sum`, `avg` **and** `median`

```
assert [['a', 'a', 'b'], ['b', 'a', 'b'], ['aa', 'aa', 'bb'], ['bb', 'aa', 'bb']] ==
GQ {
    from s in ['a', 'b', 'aa', 'bb']
    select s, (min(s) over(partitionby s.length())), (max(s) over(partitionby s.
length()))
}.toList()
```

```
assert [[1, 2, 2, 2, 1, 1], [1, 2, 2, 2, 1, 1],
        [2, 2, 2, 4, 2, 2], [2, 2, 2, 4, 2, 2],
        [3, 2, 2, 6, 3, 3], [3, 2, 2, 6, 3, 3]] == GQ {
    from n in [1, 1, 2, 2, 3, 3]
    select n, (count() over(partitionby n)),
              (count(n) over(partitionby n)),
              (sum(n) over(partitionby n)),
              (avg(n) over(partitionby n)),
              (median(n) over(partitionby n))
}.toList()
```

```
assert [[2, 6, 3, 1, 3, 4], [1, 6, 3, 1, 3, 4],
        [3, 6, 3, 1, 3, 4], [null, 6, 3, 1, 3, 4]] == GQ {
    from n in [2, 1, 3, null]
    select n, (sum(n) over()),
              (max(n) over()),
              (min(n) over()),
              (count(n) over()),
              (count() over())
}.toList()
```

GINQ Tips

Row Number

`_rn` is the implicit variable representing row number for each record in the result set. It starts with `0`

```
from n in [1, 2, 3]
select _rn, n
```

List Comprehension

List comprehension is an elegant way to define and create lists based on existing lists:

```
assert [4, 16, 36, 64, 100] == GQ {from n in 1..<11 where n % 2 == 0 select n ** 2}
.toList()
```

```
assert [4, 16, 36, 64, 100] == GQ {from n in 1..<11 where n % 2 == 0 select n ** 2} as
List
```

```
assert [4, 16, 36, 64, 100] == GQL {from n in 1..<11 where n % 2 == 0 select n ** 2}
```

NOTE `GQL {···}` is the abbreviation of `GQ {···} as List`

GINQ could be used as list comprehension in the loops directly:

```
def result = []
for (def x : GQ {from n in 1..<11 where n % 2 == 0 select n ** 2}) {
    result << x
}
assert [4, 16, 36, 64, 100] == result
```

Query JSON

```
        import groovy.json.JsonSlurper
        def json = new JsonSlurper().parseText('''
            {
                "fruits": [
                    {"name": "Orange", "price": 11},
                    {"name": "Apple", "price": 6},
                    {"name": "Banana", "price": 4},
                    {"name": "Mongo", "price": 29},
                    {"name": "Durian", "price": 32}
                ]
            }
        ''')

        def expected = [['Mongo', 29], ['Orange', 11], ['Apple', 6], ['Banana', 4]]
        assert expected == GQ {
            from f in json.fruits
            where f.price < 32
            orderby f.price in desc
            select f.name, f.price
        }.toList()
```

Query & Update

This is like `update` in SQL

```
import groovy.transform.*
@TupleConstructor
@EqualsAndHashCode
@ToString
class Person {
    String name
    String nickname
}

def linda = new Person('Linda', null)
def david = new Person('David', null)
def persons = [new Person('Daniel', 'ShanFengXiaoZi'), linda, david]
def result = GQ {
    from p in persons
    where p.nickname == null
    select p
}.stream()
    .peek(p -> { p.nickname = 'Unknown' }) // update `nickname`
    .toList()

def expected = [new Person('Linda', 'Unknown'), new Person('David', 'Unknown')]
assert expected == result
assert ['Unknown', 'Unknown'] == [linda, david]*.nickname // ensure the original
objects are updated
```

Parallel Querying

Parallel querying is especially efficient when querying big data sources. It is disabled by default, but we could enable it by hand:

```
assert [[1, 1], [2, 2], [3, 3]] == GQ(parallel: true) {
    from n1 in 1..1000
    innerhashjoin n2 in 1..10000 on n2 == n1
    where n1 <= 3 && n2 <= 5
    select n1, n2
}.toList()
```

Customize GINQ

For advanced users, you could customize GINQ behaviour by specifying your own target code generator. For example, we could specify the qualified class name `org.apache.groovy.ginq.provider.collection.GinqAstWalker` as the target code generator to generate GINQ method calls for querying collections, which is the default behaviour of GINQ:

```
assert [0, 1, 2] == GQ(astWalker:
'org.apache.groovy.ginq.provider.collection.GinqAstWalker') {
    from n in [0, 1, 2]
    select n
}.toList()
```

Optimize GINQ

GINQ optimizer is enabled by default for better performance. It will transform the GINQ AST to achieve better execution plan. We could disable it by hand:

```
assert [[2, 2]] == GQ(optimize: false) {
    from n1 in [1, 2, 3]
    innerjoin n2 in [1, 2, 3] on n1 == n2
    where n1 > 1 &&  n2 < 3
    select n1, n2
}.toList()
```

GINQ Examples

Generate Multiplication Table

```
from v in (
    from a in 1..9
    innerjoin b in 1..9 on a <= b
    select a as f, b as s, "$a * $b = ${a * b}".toString() as r
)
groupby v.s
select max(v.f == 1 ? v.r : '') as v1,
       max(v.f == 2 ? v.r : '') as v2,
       max(v.f == 3 ? v.r : '') as v3,
       max(v.f == 4 ? v.r : '') as v4,
       max(v.f == 5 ? v.r : '') as v5,
       max(v.f == 6 ? v.r : '') as v6,
       max(v.f == 7 ? v.r : '') as v7,
       max(v.f == 8 ? v.r : '') as v8,
       max(v.f == 9 ? v.r : '') as v9
```

More examples

link: the latest GINQ examples

NOTE	Some examples in the above link require the latest SNAPSHOT version of Groovy to run.