The existing RQL we have so far will not change that much. However, a few new keywords will be added for additional functionality. GROUP BY will be evaluated after FROM and WHERE, but before ORDER BY and SELECT.

Here is the new full structure of an RQL query expression:
(SELECT <attrs> FROM <tables> WHERE <condition>
***GROUP BY <attrs> [<agg-f> <attr> AS <newattr>] …** ORDER BY <order-expr>)*

We use <attrs> after GROUP BY to specify the columns we want to group. There can be more than one attribute listed, in which case, the tuples that have the same values for every single one of those attributes will be grouped with each other, but not with the ones that are the same for only one of those attributes. For instance, if **GROUP BY '(“A” “B”) [SUM “C” AS “NEW-C”]** was used on the table **('(“A” “B” “C”) (1 1 2) (1 1 3) (1 2 5))** the result would be **('(“A” “B” “NEW-C”) (1 1 5) (1 2 5))**.

Since the rest of the attributes of the table that are not in <attrs> need to be aggregated for the table to remain valid, it is required to specify which aggregation function, <agg-f>, should be used for each of the remaining attributes along with their new names, <newattr>. The keyword AS must always appear before <newattr>. An aggregation function is a function that groups together the values of an attribute from multiple tuples to form a single value (More on this soon...). Note that all attributes of the table must be accounted for in either <attrs> or in the aggregation syntax after <attrs>, otherwise the query is invalid. Additionally, when one refers to an attribute in SELECT or ORDER BY after having included a GROUP BY, the new name must be used as opposed to the old one. WHERE, however, should still use the old attribute names in its condition so that tuples can be filtered out before grouping.

Say we have two tables:
PERSON: **'((“Name” “Age” “Height”) (“Jasmin” 50 1) (“David” 55 2) (“Haris” 60 3) (“Karen” 20 4) (“Angela” 35 8) (“Gary” 10 5) (“Tom” 35 9) (“Paul” 40 5) (“Jen” 30 6) (“Steve” 35 7)**
SILLY: **'((“A”) (1))**

Implementation wise, **(SELECT “NumOfPeople” “SumHeight” FROM [PERSON “P”][SILLY “S”] WHERE (> “Age” 30) ORDER BY “Age” GROUP BY “Age” [(COUNT “Name”) AS “NumOfPeople”] [(SUM “Height”) AS “SumHeight”] [“C” AS “Constant”])** would perform GROUP BY on the resulting table, **'((“Name” “Age” “Height” “A”) (“Jasmin” 50 1 1) (“David” 55 2 1) (“Haris” 60 3 1) (“Angela” 35 8 1) (“Tom” 35 9 1) (“Paul” 40 5 1) (“Steve” 35 7 1))** which was evaluated by FROM and WHERE.

It would group all the tuples by age and temporarily combine the other attributes in separate lists.
'((“Name” “Age” “Height” “A”) ('(“Jasmin”) 50 '(1) '(1)) ('(“David”) 55 '(2) '(1)) ('(“Haris”) 60 '(3) '(1)) ('(“Angela” “Tom” “Steve”) 35 '(8 9 7) '(1 1 1)) ('(“Paul”) 40 '(5) '(1)))

Then the aggregation functions would be applied to each list created according to what column they are in. Each aggregation function is really a fold that takes the lists of attributes created and merges them in some way into a single value. The user could either use built in keywords, such as SUM, which would be implemented beforehand, or they could define their own <aggr-f> for extra flexibility.
'((“Name” “Age” “Height” “A”) (1 50 1 “C”) (1 55 2 “C”) (1 60 3 “C”) (3 35 24 “C”) (1 40 5 “C”))

The attributes would then get renamed as specified.
'((“NumOfPeople” “Age” “SumHeight” “Constant”) (1 50 1 “C”) (1 55 2 “C”) (1 60 3 “C”) (3 35 24 “C”) (1 40 5 “C”))

Lastly, ORDER BY and SELECT would give the final result.

'((“NumOfPeople” “SumHeight”) (1 1) (1 2) (1 3) (3 24) (1 5))