Aggregation in openCypher

- Partition tuples based on values for grouping key
- Return a single resulting tuple for each partition
- In openCypher: WITH/RETURN clauses
- E.g. count nodes in each class (.class property)

MATCH (n)
RETURN n.class, count(*)

József Marton - Aggregation semantics
Implicit grouping key

- Result definition of a query (step) defines aggregation
- **Neo4j 3.1 docs** tells

  ```sql
  RETURN n, count(*)
  
  We have two return expressions: `n`, and `count(*)`. The first, `n`, is not an aggregate function, and so it will be the grouping key. The latter, `count(*)` is an aggregate expression.
  ```

- What if mixing aggregate and non-aggregate expressions, e.g. the **weighted sum query**

  ```sql
  RETURN n.weight * sum(n.value)
  ```
Grouping key selection options

RETURN n.weight * sum(n.value)

1. **grouping key** is the tuple built from all variables (*) that appear outside of aggregate functions of a particular WITH/RETURN clause

   *: node, relationship, their properties or variables chained from previous query step

   Pros: clear in all situations, more flexible than option 2

   Cons: would change current Neo4j behavior

2. each item of the expression list in WITH/RETURN forced to contain either

   i  no aggregate function, or

   ii single aggregate function at the outermost level (this is the approach in #188, #218).

   Grouping key is the tuple built from items of type (i), i.e. those w/ no aggregates

   Pros: in line with current Neo4j behavior and the grouping operator in Ullman's Database systems -- The complete book, 2009

   Cons: poses restriction on WITH/RETURN clauses, can't handle the weighted sum query w/o rewriting as

   WITH n.weight as weight, sum(n.value) AS sum_val
   RETURN weight * sum_val
TODO: Choose

- Neither option restrict expressiveness might need some query rewrite
- Option 1 seems clear and flexible enough for practical queries
- Option 2 is in Neo4j, but complex aggregation and non-aggregation expressions might yield *counter-intuitive result*

Posing restrictions on creating complex expressions by mixing aggregations and non-aggregations is a safety net for beginners, but cumbersome for more complex queries.
Feel the difference

MATCH (n)
RETURN abs(n.weight) AS abs, count(*) AS cnt

Option2 gives:

<table>
<thead>
<tr>
<th>abs</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Option1 gives:

<table>
<thead>
<tr>
<th>abs</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Input graph:

ten nodes: two for each weight -2, -1, 0, 1, 2

Model Opt.2 in Opt1

MATCH (n)
WITH abs(n.weight) AS abs, n
RETURN abs, count(*) as cnt
Let’s get loud

1. **grouping key** is the tuple built from **all variables** (*) that appear **outside of aggregate** functions of a particular WITH/RETURN clause

   *: node, relationship, their properties or variables chained from previous query step

2. each item of the expression list in WITH/RETURN forced to contain either
   i. no aggregate function, or
   ii. single aggregate function at the outermost level (this is the approach in #188, #218).

   Grouping key is the tuple built from items of type (i), i.e. those w/ no aggregates
That’s all