Aquery to Q Compiler: Optimizations

Jose Cambronero

April 16, 2015

In the following document we describe the main optimizations in the Aquery to q compiler, along with any algorithms employed.

1 Important semantics

1.1 where-clause

1.1.1 Cross-row dependencies

It is important to note that where clauses in Aquery function differently than those in traditional SQL. In a traditional SQL query, selections are independent across rows, as operations are row-oriented. This means that the order of selections is irrelevant, and that they can be commuted without any issues. Indeed, a traditional optimization in a heuristic optimizer might be to take a conjunction of selections and make it a sequence of selections (reference xxx).

add reference

Meanwhile, Aquery operations are column-oriented, meaning that the results of a selection are not necessarily independent across rows. Namely, the use of an aggregate results in a selection that has cross-row dependencies. Consider:

```
select * from t assuming asc c1 where sums(10)>=10
```

1.1.2 Selections as sequences

In contrast to SQL, Aquery selections are treated as a sequence of selections to be applied in a given order. This results in richer expressiveness for selections. Consider that, if selections are performed as a sequence, then the following 2 queries have quite different meanings:

```
select * from portfolio assuming asc data where sums(
   volume)>=100 and ticker in ("ibm";"hp")

select * from portfolio assuming asc data where ticker in
    ("ibm";"hp") and sums(volume)>=100
```

The first returns transactions associated with ibm and hp stock after our portfolio had a cumulative size of 100 or greater. The second returns ibm and hp transactions after our purchases of ibm and hp stock cumulatively equaled 100 or more.

The only exception to treating selections as a sequence are any equality selections associated with an implicit join, which are extracted during the optimization process. We define these to be solely of the form

`Table1.column_nameX = Table2.column_nameY`

Given that it is the use of aggregates that results in cross-row dependencies, selections that take place between the use of aggregates can be safely commuted. Meaning, if we have $s_i ... s_n A_1 s_j ... s_m$, where A_1 is an aggregate selection, then we are free to rearrange s_i through s_n, apply those selections, then applying A_1, and then rearranging the remaining selections as we wish and then applying.

2 Main Optimizations

2.1 cross products to equijoins

If the query provided by the user has any cross-products, we scan the selections in the where clause for any selection that can be used as an implicit join clause. These are extracted and used to transform the cross-product into an equijoin on that selection.

2.2 Picking implicit joins

Joins explicitly provided by the user are executed as provided. Meaning, if the user provides `(A inner join B) inner join C` we perform the joins as instructed. However, for implicit joins, we apply a heuristic to attempt to find the most favorable equijoin. Given that this optimization is done at translation time, and we don't have access to table information (as the tables might exists outside of the Aquery environment), we create a heuristic that chooses crosses that result in the largest number of equality selections being applied to that join. The heuristic is based on the strong assumption that a larger number of equality selections is associated with a larger reduction in cardinality overall.

2.3 selection pushing

If the query's from clause features joins/cross-products, we automatically push down any equality selections as far as possible (i.e. the deepest node in the query plan that has all the tables necessary for the selection). Selections that are not equality-based, become part of new type of logical query plan node called **poss_push_filter**, short for possible push. These nodes are placed right above the deepest join involving the necessary tables. During code generation, these nodes are used to create 2 possible branches. If the tables join clause for the tables involves columns that have indices that can be taken advantage of, we join first and then apply the selections stored at the poss_push_filter node. If there are no indices involved, we push down those selections to the appropriate tables prior to joining.

2.4 sorting necessary columns

Rather than sort the entire table, we limit ourselves to sorting only those columns that require order, given the operations that are performed on them and any interactions with other columns.

3 Implementations

3.1 Sorting columns

Identifying columns that need sorting in an order-dependent Aquery expression requires a bit of work. The algorithm below addresses one possible way of doing this.

3.1.1 Example Expression

We take as an example the following expression, which we assume must be executed as one (i.e. we can not first execute order independent parts).

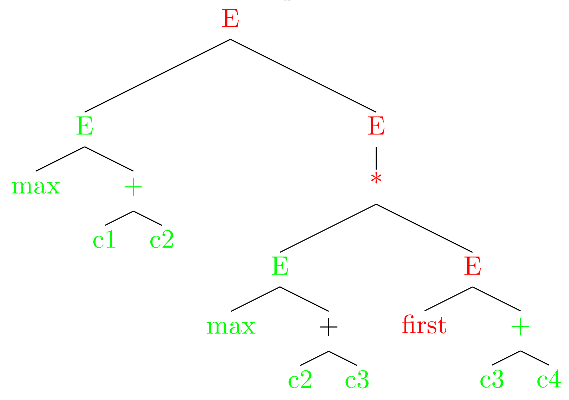
`max(c1,c2), max(c2 + c3) * first(c3)`

We assume `c1,c2,c3` are all columns. Notice how the only order dependent operation is `first(c3)`. However, given that we sort `c3` for this, we need to sort c2 due to `c2+c3` and for similar reasons we need to sort `c1`. The algorithm below finds all such columns.

3.1.2 AST Representation

Below an AST representation of this expression, which informs the algorithm below.

Bottom up order dependency information is shown in color, green is order-independent, red order-dependent. Note that this reflects information as we build the tree up, so e.g `first(c1)` would display `c1` as order independent, but the node calling `first` on `c1` as order dependent.



3.1.3 Identifying columns to sort

Below our approach to identifying these columns that need to be sorted.

```
1 - create an empty list S and an empty list of lists P
2 - During DFS traversal , for any subtree annotated as
    order-dependent , add any column names encountered
    until we encounter an order-independent operator (e.g.
     max/sum)
3 - Any columns encountered in an order-independent
    subtree are collected bottom-up as a list and added to
     P upon encountering an order-independent operator
```

```
4 - Once we are done with our DFS, we perform the
    following fixed-point algorithm:

added = 1
while(added) {
        for(p in P) {
        if(any column in p is in L)
        {
            add p to L;
            remove p from P;
            added = 1
        }
    }
}
```

3.2 Optimizing from clauses

We note that given the semantics of Aquery's where clauses, optimization of the from clause requires various careful steps.

```
Given a from clause, and a where clause:
1 - split into a list the from clause at every cross-
    product operator, calling this the split-from-list
2 - Extract any implicit join clauses from the where
    clause, call this join_clauses and remaining
    other_clauses
3 - Split other_clauses into 2 based on the index of the
    first order-dependent selection, call these oi_clauses
     and od_clauses
4 - Split oi_clauses into 2 based on the first use of an
    aggregate, call these no_agg_clauses and agg_clauses
5 - Group no_agg_clauses into 2: those involving columns
    with no clear table reference (i.e. they are not of
    the form A.x), call these known_clauses and
    unknown_clauses
6 - Split the known_clauses into equality selections and
    not, call these eq_filters and non_eq_filters
6 - Given the split-from-clause list, the join_filters,
    and the eq_filters, select the best joins given the
    heuristic, push eq_filters down as appropriate
7 - After all joining necessary has been arranged in the
    tree, push down non_eq_filters (which get placed above
     joins as possible pushes to perform based on index
    presence)
8 - Apply unknown_clauses
8 - Apply agg_clauses
9 - Perform sorting analysis and apply od_clauses
```

Below an outline of the join heuristic.

```
max_ct = 0
```

```
for on_clause in join_filters:
    for left in split_from_list:
       for right in split_from_list:
            if(left > right):
            if(join_possible(left, right,on_clause)):
                ct = ct_eq_filters(left,right, filters)
                if(ct > max_ct):
                    chosen_left = left
                    chosen_right = right
                    max_ct = ct
```

This returns the nodes that we should cross at each step. We remove both left and right from the list of nodes we have, and we create a new node that joins them based on the relevant condition and add this node to the list, so that it is available for the next iteration.