QPModel Implementation Notes

QPModel Dev Team,  
10/2020

# Optimizer

## Requirements

## Pre-Optimization

This includes binding, normalization and simplify query.

* Binding is the process to resolve each name in the query to the database or query object references.
* Normalization is to rewrite the expression into normalized canonical form, so the later comparisons don’t have to deal with equivalents variants.
* Simplification stage is also called algebra optimization in textbook. The most important one is predicate push down, which push down the query predicate into the scan or closer to the scan to reduce rows flow in the plan. The former case is called SARG (search argument)-able scan. It can also include group-by columns reduction, generate implied predicates, remove redundant joins given PK/FK relationship etc.

## 1.2.1 Overview of binding in qpmodel.

Binding starts with top level statement calling Bind() method. Classes derived from SqlStatement implement Bind()method as required for that class. What follows is mostly from the point of view of SelectStmt class (The class implementing Select statement) with some overview of other classes. All of these classes are in the namespace of qpmodel.logic.

Select statement’s SelectStmt.Bind()is called with a parent context. Parent context will be null for the top level statement (Select statement and others). It will be set to the parent statement or context when there is an outer level statement.

Select statement’s Bind()creates a new BindContext to signify the creation of a new context or scope for all the object references in this context or scope. This new scope links to the parent context if it was not null. BindContext includes a global subquery counter a Dictionary of all the tables references in the FROM clause of the current SELECT statement (or other statements as required), the current SQLStatement and the parent’s BindContext. This context is passed down as required.

SelectStmt.Bind() is called from Index creation (CreateIndexStmt), Subquery binding, binding of DML statements and binding of UNION SELECTS. UNION in this section refers to UNION, EXCEPT, INTERSECT with and without ALL/DISTINCT qualifier for brevity, these are all managed or encapsulated by the class called SetOpTree.

SelectStmt.Bind() does one of two things depending the statement being a part of UNION SELECT or not. If it is part of UNION it calls Bind() defined in SetOpTree to handle the binding of select statements in the UNION followed by binding the ORDER BY defined after the last select in the UNION, if there is one. SetOpTree.ind()will eventually call BindWithContext() in SelectStmt.

If the SELECT is not part of a UNION, then SelectStmt.BindWithContext()is called. This is the main driver of the binding process of a select statement. First, the FROM clause is bound so that all other column references can be validated and bound correctly. During the process of binding of table references in the FROM clause, BindWithContext()may be entered recursively to bind the subqueries and QyeryRef, FromQueryRef (also known as derived tables, query expressions etc.,) found in the FROM clause.

After the tables references and derived tables are all bound, the select list elements are bound one after another. This involves finding the table reference to which a column reference belongs to and setting the type information of these columns and the expression they are part of. After the select list element is bound it is normalized.

Next, the WHERE clause is bound and normalized if there is WHERE clause. Next, GROUP BY, HAVING and ORDER BY clauses are bound and normalized if they are present.

## 1.2.2 Binding Table References.

A table reference in the FROM clause can be a simple base table reference (BaseTableRef), or an ExternalTableRef, or a table defined by one of QueryRef, FromQueryRef, JoinQueryRef (derived table). A base table is looked in the system catalog (Catalog class). If it is not found in the catalog, it is looked up the list of WITH clauses (CteExpr class) this FROM is part of.

If the named table is not found after all these searches then an error is raised which ends the binding. If the named table is found it is checked for duplicated use (no two table refences should refer to the same table by name or alias). If the table is found and it is valid use, then it is entered into the list of table references found in the current FROM clause.

**1.2.3 Binding Select list expressions.**

Although the section header mentions binding of select list expressions, the process of binding expressions in other contexts is almost the same. A select list element can be a \* (star), table.\*, simple column reference , a literal value, an expression or a scalar subquery returning single row and a single column. A \*, or table.\* is expanded to include all columns of all tables in the FROM clause and bound to the table to which each column belongs to.

All other elements in the select list are bound by the virtual method BoundAndNormalize in the Expr class. Some classes derived from Expr class override the Bind() method to customize the process of binding specific it. BindAndNormalize() calls Expr.Bind()followed by Expr.Normalize()to do the actual binding and normalization of the expression. Expr.Bind() calls Bind() method on each child first and replaces the original expression by the newly bound expression and resets aggregate table references, if any, in that expression to collect all tables referenced by the current expression, this list is maintained by tableRefs\_ in the Expr class.

Column references are bound by ColExpr.Bind() method. Column references bound by looking up the column reference by alias if one is provided or the name it self if there is no alias in the list of tables in the current context, if it is not found in the current context, it is looked up in the parent context recursively. If the named column is not found at the of this lookup, an error is raised. If the column reference is found in some parent context, then it is marked as a “Parameter” in this context and this context is now known to be correlated to the context in which the column reference was found and resolved.

A check is made to ensure no two column references are duplicates and also that the a given column reference is resolved by one and only one table in the current context.

The ColExpr class represents a column reference or a expression with one column. The column reference’s ordinal is set to the ordinal position of the column in the table definition and the type to that of the type of the column definition in the table definition.

If the column is resolved to a FromQuery (derived table) it is DeQueryRef’d. This involves identifying the column’s underlying base table. This is done so that if remove\_from optimization is enabled there is no clash between the name of the column as it is known before the optimization and the name it will be known outside (inlined version of the query) with the other names in the main statement.

Literals are represented by ConstExpr class, unary, binary and other function expressions are represented by UnaryExpr, BinExpr and FuncExpr classes respectively, most of them override Bind() and provide the specialization. For instance, BinExpr is specialization for all binary expressions. BinExpr.Bind() first lets the parent class do the binding of the two children. Next it does the semantic validation of the operation, types and other validation checks and sets up its own type based the type of the operation and the children.

When all children of an expression are bound the expression is marked as bounded.

**1.2.4 Binding WHERE and HAVING Clause**

The process of binding the WHERE clause is almost the same as binding any other expression with some extra processing. After binding the WHERE clause and normalizing it, it is possible that the entire WHERE clause may be replaced by a constant expression representing TRUE or FALSE indicating the fact that the WHERE clause evaluates unconditionally to TRUE or FALSE respectively. WHERE clause is validated so that there are no aggregate functions and that it’s type is Boolean. If remove\_from optimization is enabled, the WHERE and HAVING clause go through the process of DeQueryRef. The only difference in binding WHERE and HAVING is that HAVING can contain aggregates.

**1.2.5 Binding GROUP BY and ORDER BY Clause**

Before biding expressions in the GROUP BY and ORDER BY clause, the expressions are replaced by references to the respective expressions in the select list (order by 3, would make it order by the third select list element) if this is the case then the expression has already been bound. Other expressions in the GROUP BY and ORDER BY which are not position specifications are bound as any other expression. The presence of a GROUP BY expression sets a flag hasAgg\_ in the SelectStmt. Expressions in the GROUP BY are validated to contain no aggregates as arguments but if a FromQueryRef has been transformed (removed and merged or inlined with the main query) then nested aggregates do not raise error. At the same time, if a FromQuery has been removed, each expression in the GROUP BY and ORDER BY goes through DeQueryRef.

**1.2.6 Binding Query Expressions**

In Qpmodel code, Query Expressions (and joined table) have many different variations and they are called QueryRef, FromQueryRef, CTEQueryRef, JoinQueryRef and they are represented by class named so. FromQueryRef class deals with queries appearing the FROM clause. Each of these queries represents a virtual or derived table. They may require to have a name (derived table name), and also name the output expressions (derived column names), known as outside[x].outputName\_. FromQueryRef maintains a map of the names in the derived column names and (known outside the query definition) and the expression/names they represent inside. The process of DeQueryRef helps resolve outside names to inside names and to the base table to which a given column belongs to. These are all bound like any other SELECT with a few differences.

**1.2.7 Binding Sub Query Expressions**

All kinds of subqueries are derived from SubqueryExpr class, which is derived from Expr class. They are bound through process like SELECT with additional validations such as a scalar subquery must return only one column, one row etc. The entry point for binding subqueries is SubqueryExpr.BindQuery(). A SELECT statement (top level query) doesn’t have a DataType but other kinds of subqueries have a type. Binding sets up this type.

**1.2.8 Normalization of Expressions**

Expressions are bound and then they are converted to a normal or canonical form so that two semantically equivalent expressions written in different form/structure can be identified quickly. Another goal is to simplify the expressions as much as possible during the compile phase. Normlizer.cs implements Normalize(), a virtual method of Expr class.

Each derived class of Expr may override Expr.Normalize()specialize the needs of normalization of that class but only FuncExpr, CoalesceFunc, UnaryExpr, BinExpr, and CastExpr override to specialize normalization.

In order to avoid clutter in all the classes that have to specialize normalization, Expr and other classes which have to specialize Normalize(), they have been declared as partial classes and their Normalize() method is implemented in Normalize.cs.

Binding phase calls Normalize()on each expression after it has been bound. The base version simply calls Normalize()on all children of the current expression and returns possibly modified expression.

Logically, normalization does the following transformations to the expression. They are not

implemented in the exact manner they are described to avoid traversing the expression tree

several times and more than one transformation could happen at once. In the rules below 'op' and 'comp' denote a generic arithmetic and comparison operators when specificity is not required

* + - 1. Constant move. Bring all possible constants together so that later transformations can simplify or even remove some of the constants.

1) CONST + expr => expr + CONST

2) CONST \* expr => expr \* CONST

3) CONST comp expr => expr ~comp CONST

4) CONST < expr => expr > CONST

5) CONST > expr => expr < CONST

6) expr op CONST1 comp CONST2 => x comp CONST2 ~op CONST1

Ex. x + 1 comp 10 => x comp 10 – 1, and later it becomes 9.

* + - 1. Constant folding. Replace expressions involving constants with the value of that part of the expression.

1. CONST op NULL => NULL
2. CONST op CONST => EVAL
3. FUNC(CONST) => EVAL. FUNC is one of the aggregates MIN, MAX, AVG or other non-aggregate function.
4. FUNC(NULL) => NULL. FUNC is one of the aggregates MIN, MAX, AVG, SUM or other non-aggregate function.
   * + 1. Arithmetic Simplification. Eliminate unneeded computations
5. expr op NULL => NULL

Except when op is IS or IS NOT, in which case it values to TRUE or FALSE.

1. expr + 0 => expr
2. expr - 0 => expr
3. expr \* 1 => expr
4. expr / 1 => expr
5. Distribute multiplication over addition when there are constants. Ex. (x + 5) \* 10 => x \* 10 + 50
6. Distribute multiplication over multiplication where there are constants. Ex. (x \* 5 ) \* 10 => x \* 50.
   * + 1. Coalesce simplification.
7. COAL(x, a, b, .. const, …) => EVAL to const.
   * + 1. CAST simplification. Eliminate CAST of CAST of CAST and so on when the argument is constant.
       2. NOT simplification.
8. NOT NOT X => X
9. NOT (X AND Y) => NOT X OR NOT Y
10. NOT (X OR Y) => NOT X ND NOT Y
    * + 1. Comparison Simplification. Eliminate unneeded comparisons.
11. CONSTEXPR1 comp CONSTEXPR2 => EVAL
    * + 1. Relational operator simplification.
12. X + CONST1 = CONST2 => X = CONST2 – CONST1
13. X + CONST1 > CONST2 => X > CONST2 - CONST1
14. X + CONST1 >= CONST2 => X >= CONST2 - CONST1
15. X - CONST1 = CONST2 => X = CONST2 + CONST1
16. X - CONST1 >= CONST2 => X >= CONST2 + CONST1
    * + 1. Logical simplification.
17. CONST1 AND CONST2 => EVAL
18. CONST1 OR CONST2 => EVAL.

There are a few more simplification transformations possible but not implemented at this time.