Arbori Starter Manual

Eugene Perkov

What is Arbori?

Arbori is a *query language* that takes a *parse tree* as an input and builds a *result set*[1] per specifications defined in a query.

What is Parse Tree?

A *parse tree* is a data structure produced by (obviously) *a parser*. A parser is a program that takes a text written in some programming language and outputs a parse tree according to *a grammar* of that programming language.

Arbori works with SQL and PL/SQL languages. It also understands SQL*Plus scripting language. Therefore in this document we'll be talking about SQL and PL/SQL parsers and parse trees that they produce.

Let's have a quick look at how SQL and PL/SQL parse trees might look like.

SQL Query

```
SELECT
      EMP_NAME,
      DEPT_NAME
FROM
      EMPLOYEES,
      DEPARTMENTS
WHERE
      EMPLOYEES.DEPT_ID = DEPARTMENTS.DEPT_ID;
```

1 More correct term (from the Relational Theory standpoint) would be *a relation*.

Parse Tree

```
[0,17)   library_unit  sql_statement  sql_statements  unlabeled_nonblock_stmt
  [0,16)   query_block  select  simple_set_expr  sql_query_or_dml_stmt  sql_stmt  subquery
    [0,4)   select_clause
      [0,1)   'SELECT'
      [1,4)   select_list
        [1,2)   column  expr  expr#  identifier  select_list  select_term  simple_expression
        [2,3)   ','
        [3,4)   column  expr  expr#  identifier  select_term  simple_expression
    [4,8)   from_clause
      [4,5)   'FROM'
      [5,8)   cartesian_product
        [5,6)   cartesian_product  identifier  query_table_expression  table_reference  table_reference_or_join_clause
        [6,7)   ','
        [7,8)   identifier  query_table_expression  table_reference  table_reference_or_join_clause
    [8,16)   where_clause
      [8,9)   'WHERE'
      [9,16)   comparison_condition  condition  simple_comparison_condition
        [9,12)   column  expr  expr#  simple_expression
          [9,11)
            [9,10)   identifier
            [10,11)   '.'
          [11,12)   identifier
        [12,13)   '='  cmp_op
        [13,16)   column  expr  expr#  simple_expression
          [13,15)
            [13,14)   identifier
            [14,15)   '.'
          [15,16)   identifier
  [16,17)   ';'
```

PL/SQL Code

```
BEGIN
      i:=1+2;
      x:='abc';
END;
```

Parse Tree

```
[0,15)   block_stmt  labeled_block_stmt  library_unit  sql_statement  sql_statements
  [0,1)   'BEGIN'
  [1,13)   seq_of_stmts
    [1,8)   assignment_stmt  labeled_nonblock_stmt  sim_stmt  stmt  unlabeled_nonblock_stmt
      [1,2)   identifier  name  name_wo_function_call
      [2,3)   ':'
      [3,4)   '='
      [4,7)   and_expr  arith_expr  boolean_primary  pls_expr  rel  sim_expr
        [4,5)   arith_expr  digits  factor  numeric_literal  pri  term
        [5,6)   '+'  binary_add_op
        [6,7)   digits  factor  numeric_literal  pri  term
      [7,8)   ';'
    [8,13)   assignment_stmt  labeled_nonblock_stmt  sim_stmt  stmt  stmt_list_opt  unlabeled_nonblock_stmt
      [8,9)   identifier  name  name_wo_function_call
      [9,10)   ':'
      [10,11)   '='
      [11,12)   and_expr  arith_expr  boolean_primary  factor  pls_expr  pri  rel  sim_expr  string_literal  term
      [12,13)   ';'
  [13,14)   'END'
  [14,15)   ';'
```

Parse Tree Structure

It is important to understand how parse tree is structured and formatted. This knowledge is crucial for writing correct and efficient Arbori programs.

The parser sees SQL or PL/SQL program as a stream of *tokens*. A *token* can be thought of as *a word*. For example the latter PL/SQL example can be represented as follows:

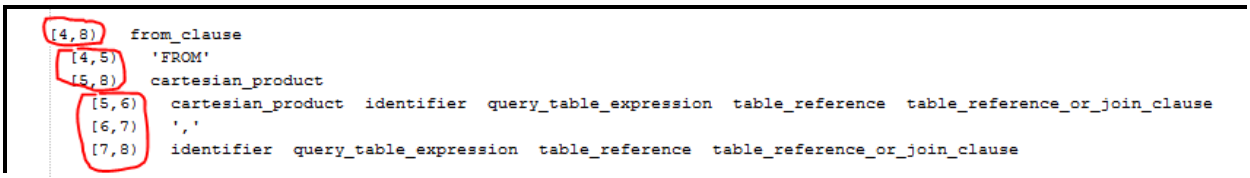
Token	Token Position
BEGIN	0
i	1
:	2
=	3
1	4
+	5
2	6
;	7
x	8
:	9
=	10
'abc'	11
;	12
END	13
;	14

The parser operates in terms of *semi-open intervals*. Using intervals makes it easy to operate in terms of *parent-child*, *sibling*, *ancestor-descendants*, etc. relationships, in other words they naturally represent *hierarchies* which parse trees essentially are.

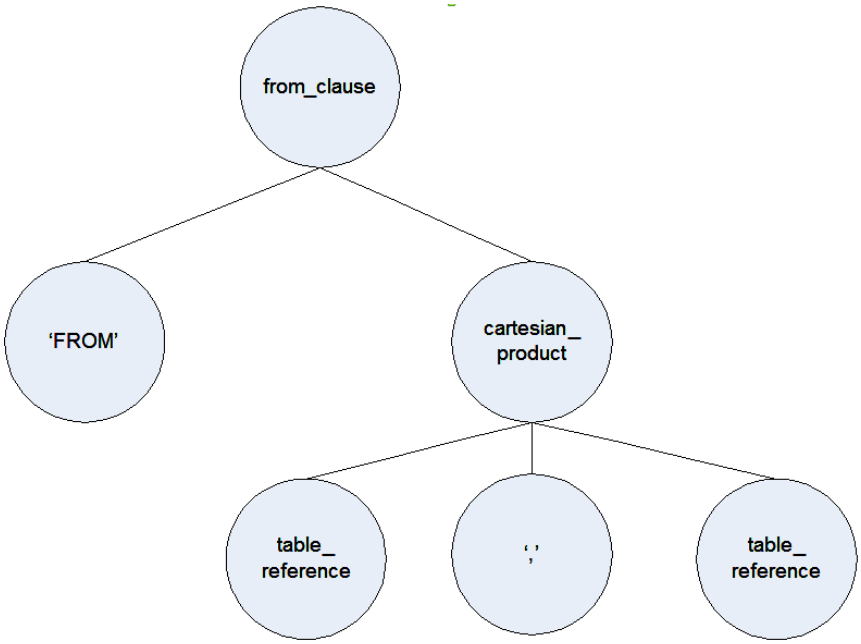
A *parse tree* is a tree (in terms of *graph theory*) which *internal nodes* are *grammar categories* and *leaf nodes* are *tokens*. Examples of grammar categories for SQL language are:

- select_clause
- where_clause
- simple_comparison_condition, etc.

Let's take a look at a fragment of parse tree for our SQL query.



Here from_clause category is a parent of all nodes from 4 up to but not including 8. Leaf node 'FROM' and internal node cartesian_product are siblings. Node 'FROM' is an *older sibling* (think of it in terms of *earlier date of birth* – 'FROM'-nodes interval start – 4, is less than cartesian_product's which is 5). Further, cartesian_product node has three children. Below is a more conventional view of this *parse sub-tree*.



Arbori Program

As mentioned before Arbori is a *query language* therefore Arbori program is essentially *a query* or *a statement*[2]. Actually Arbori program can be a *list of statements* but it appears that Arbori Editor of SQL Developer only supports programs made of a single statement. Since this is a *starter* manual focused on SQL Developer we shall limit ourselves to single-statement programs here.

A statement has the following structure:

```
identifier: predicate ;
```

It helps to think of a statement as a *named query*. Predicate is a program (a query) and identifier is its name[3].

---

2 Terms *query* and *statement* are considered equal and used interchangeably in this text.

3 As a matter of fact identifier is allowed to be wrapped inside double quotes. For identifiers made up from several words double quotes are mandatory. All identifiers below are valid ones:

- exists_predicate
- "exists_predicate"
- "exists predicate"

However identifier

- exists predicate

Now let's take a look at *predicate*. A simplest form of predicate is *atomic predicate*. And the simplest form of atomic predicate has the following structure:

```
[node) payload
```

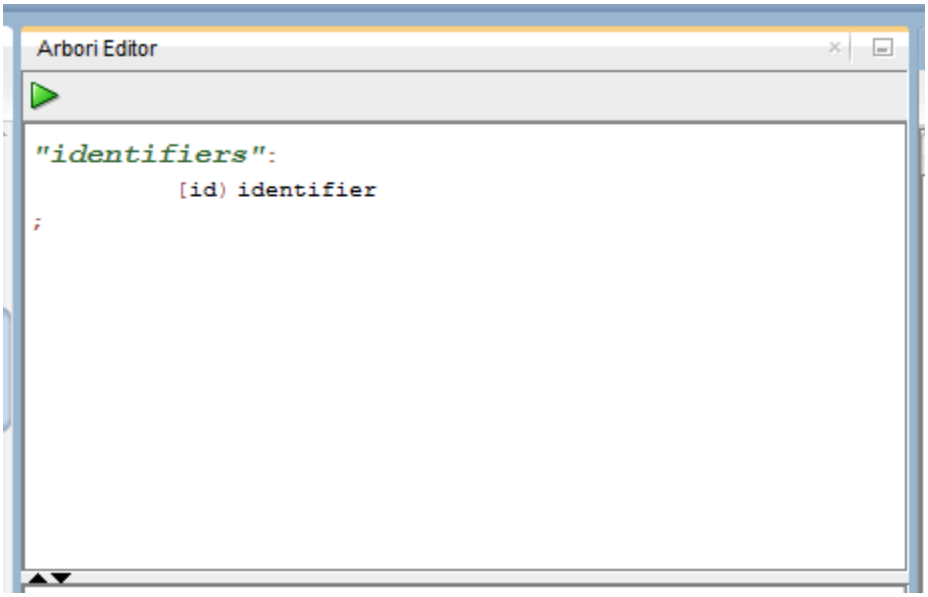
Node part itself can represent some node of parse tree or it can be a reference to other nodes. From my own experience I know that it can be quite confusing at first so let's take it slowly and play with the simplest case.

For the simplest case you can think of predicate above as variable definition. Here *node* is variable and *payload* is its value. Let's build a few simple examples.

```
"identifiers": [id) identifier;
```

This piece of code defines a named query called "identifiers". In a predicate of this query we define variable *id* which accepts any node of the parse tree whose grammar category is *identifier*.

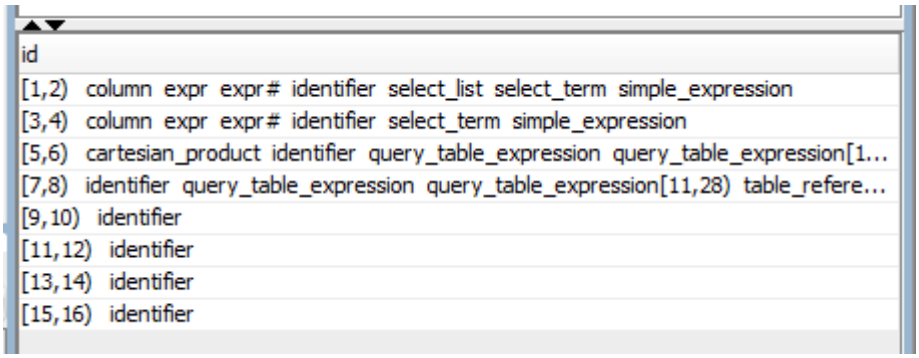
In SQL Developer:



Executing this query will produce a *result set* (a *relation*) with only one column (or *attribute* in terma of Relational Theory).

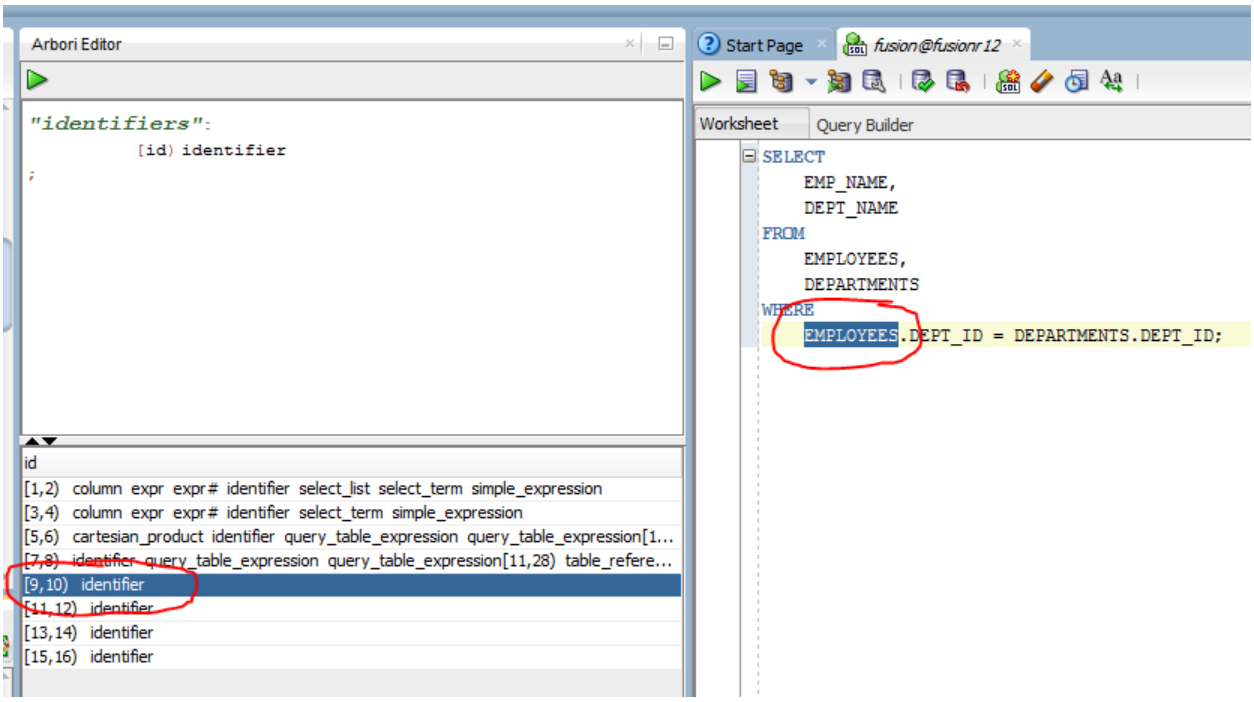
---

is invalid. The naming convention for identifiers is therefore pretty much the same as in SQL.

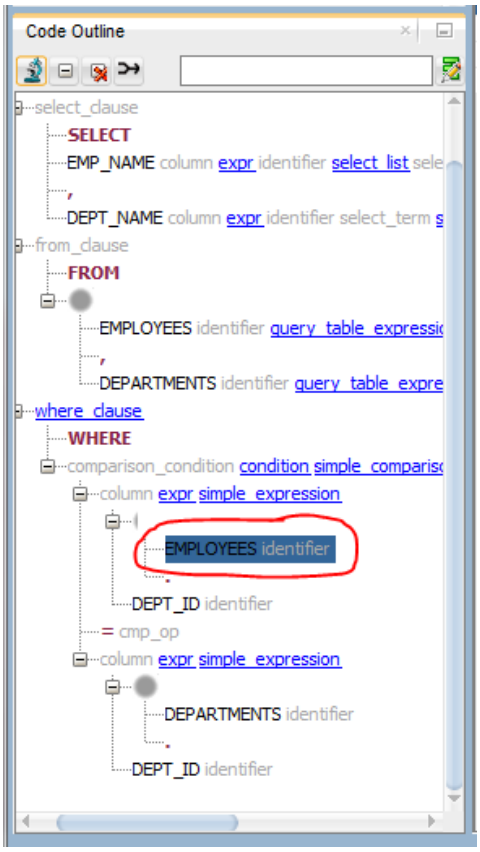


This result set has 8 rows and one column. Number of rows is determined by how many *identifier* nodes are found in our query. Number of columns depends on how many *variables* we have defined in Arbori query. In our simplest case we have defined only one variable, *id*, hence one column in the result set.

Now we can highlight any row in the result set and corresponding identifier will be highlighted in SQL Worksheet area:



Code Outline view will also highlight corresponding node in the parse tree:



Let's try to add another variable definition into our named query. For that we would need to extend definition of Arbori *statement* a little bit. Let's recall how our initial definition looked like:

```
identifier: predicate ;
```

Term *predicate* here is inherited from mathematical logic where it basically means a *Boolean-valued function* (i.e. a statement that can be evaluated to *true* or *false*). A logical *expression* can be built from predicates using *Boolean operators* such as '&' (AND), '|' (OR), etc. But since we are dealing with programs that return *result sets* there is another (set-theoretic) meaning to these operators. For example operator & also means *UNION of attributes of* two result sets or, in other words *JOIN* of two relations[4]. Usual binding rules apply i.e. & binds stronger than | so in expression A & B | C, A & B is evaluated first and then the result is evaluated with C over |. Parentheses can be used to change evaluation order: A & (B | C).

The next iteration of Arbori statement definition is as follows:

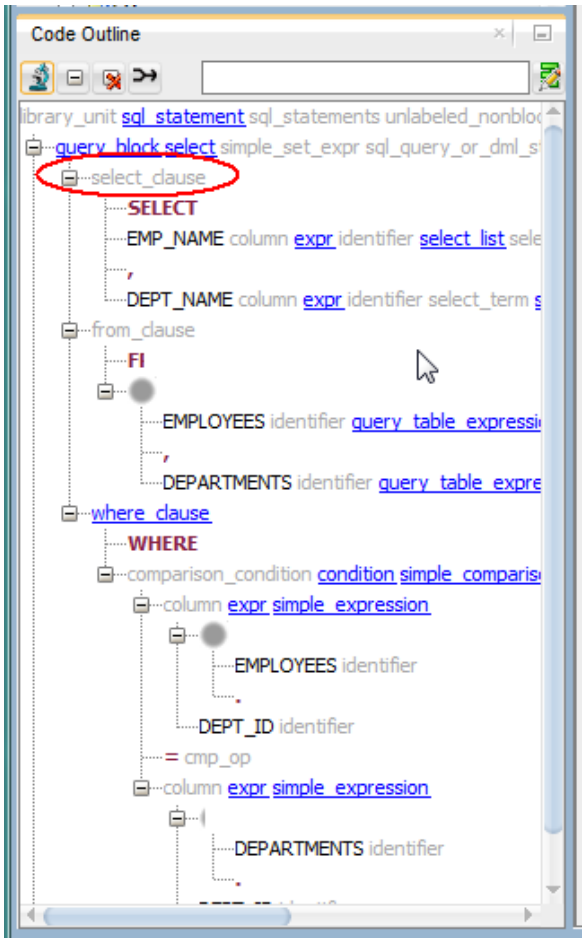
```
identifier: predicate1 OP predicate2 OP … OP predicateN ;
```

4 Just to remind everyone, Relational Theory defines [Cartesian] join of two relations $R_1(A_1, \ldots, A_n)$ *and* $R_2(C_1, \ldots, C_m)$ as relation $R_3(A_1, \ldots, A_n, C_1, \ldots, C_m)$ where each tuple of R_1 is concatenated with each tuple of R_2. But since Relational Theory does not care about order of attributes in a tuple, operation of *concatenation* is actually *UNION* of two sets of attributes.

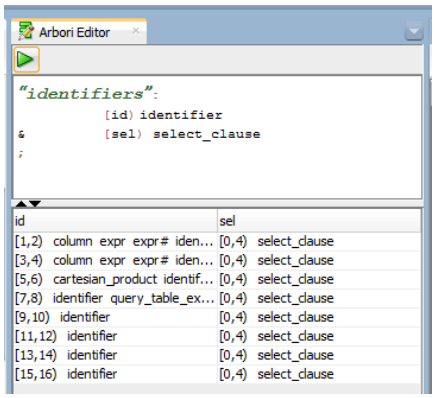
Where OP belongs to {&, |} and predicate is variable definition. Let's update our simple Arbori program:

```
"identifiers": [id) identifier & [sel) select_clause;
```

Here we used SQL Developer Code Outline window to pick another parse tree node for our example.

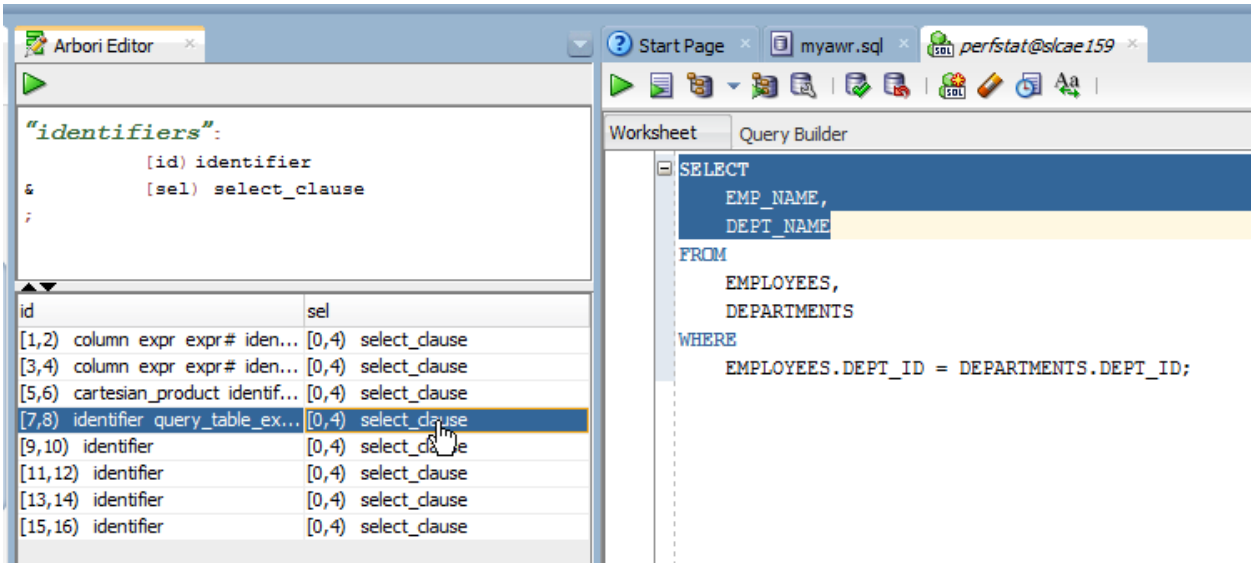


We now expect to see a result set with *two attributes* (identifier and select_clause) and Cartesian product of two result sets, one for identifier (8 rows) and one for select_clause (1 row since we only have one select clause in our simple query).



id	sel
[1,2) column expr expr# iden...	[0,4) select_clause
[3,4) column expr expr# iden...	[0,4) select_clause
[5,6) cartesian_product identif...	[0,4) select_clause
[7,8) identifier query_table_ex...	[0,4) select_clause
[9,10) identifier	[0,4) select_clause
[11,12) identifier	[0,4) select_clause
[13,14) identifier	[0,4) select_clause
[15,16) identifier	[0,4) select_clause

Apparently we've obtained a Cartesian product of two result sets with one attribute in each! We can pick select_clause in any row; the result will always be the same.



As in database applications Cartesian joins are not very useful unless we are building mostly meaningless but huge result set for testing purposes. The same is almost always true for Arbori queries. There are two primary ways of bringing Cartesian product back to senses: filters and join predicates. We'll start backwards from the latter one.

Once we defined our variables we may have to establish some meaningful relationships between them according to our intentions. Since Arbori works on trees these relationships are come naturally: parent – child, ancestor – descendants, and siblings – we've mentioned them before already[5].

Let's say that we want to find all identifiers contained inside SELECT list of the query. More formally, we want to locate all *identifier* nodes whose ancestor is *select_clause* node. Can we do better?

Arbori offers special syntax for 'ancestor – descendants' relationship. To express 'node A is an ancestor of node B' in Arbori we say

```
A < B
```

Here A and B are *variables* defined elsewhere in the same Arbori program. So our improved query "identifiers" will now look like[6]

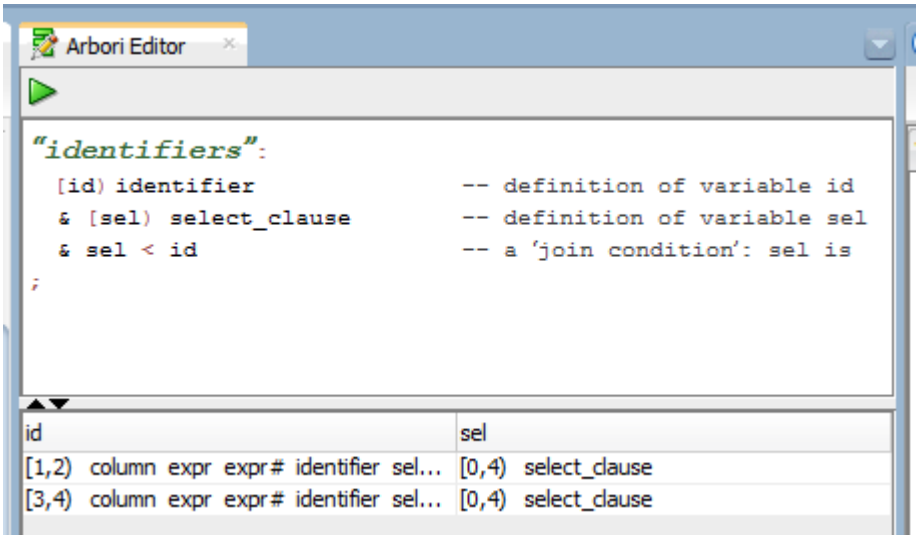
```
"identifiers":
    [id) identifier          -- definition of variable id
```

5 Arbori also offers less trivial types of relationships: node containing other nodes, node contained within other node, set operations, etc. We'll keep them outside of the scope of this starter manual.

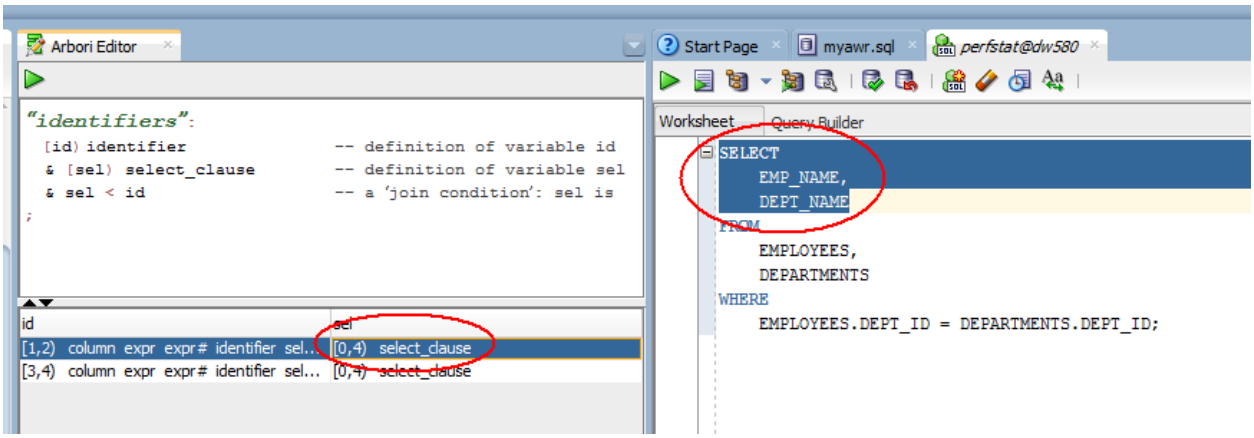
6 We have also changed formatting of our query to avoid too long line. We also used comments; comments in Arbori are marked same way as in SQL*Plus (two dashes).

```
    & [sel) select_clause     -- definition of variable sel
    & sel < id                -- a 'join condition': sel is
;                             -- a parent of id
```

Please note that when defining of node variables we must use *semi-open* interval symbols: [id), [sel), etc. However these brackets must be dropped when we refer to already defined variables like in a 'join condition' above.



Now we only have two identifiers in the result set properly joined with select_clause they belong to.



Relationship *parent – children* is stricter than just *ancestor - descendants*[7]. A parent is an *immediate* ancestor. If node A is a parent of node B we express it in Arbori as

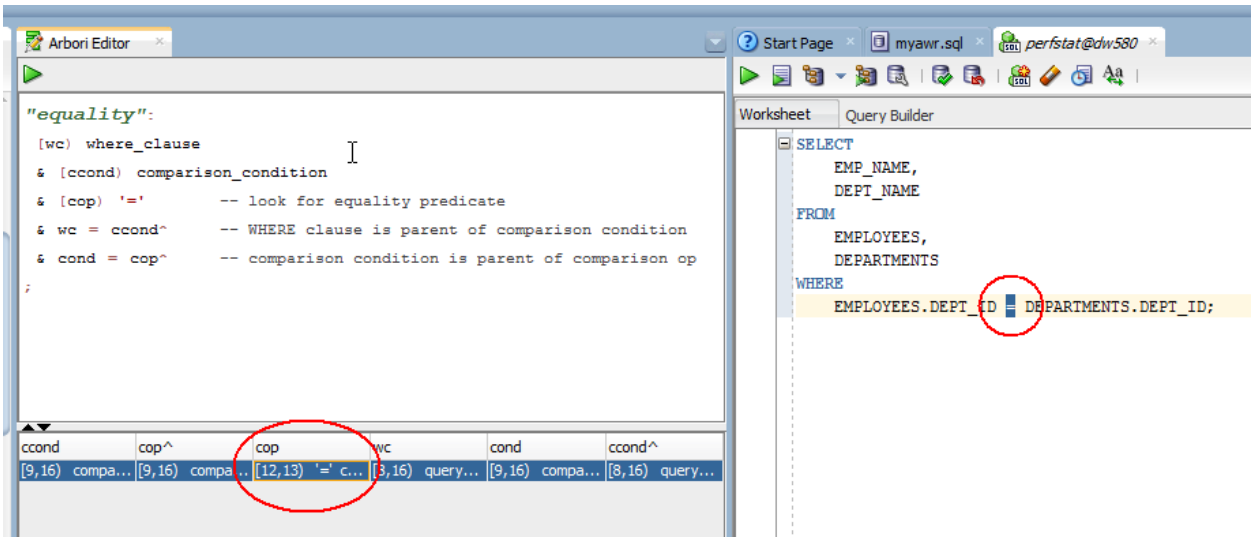
```
A = B^
```

For example we want to find all equality predicates in WHERE clause.

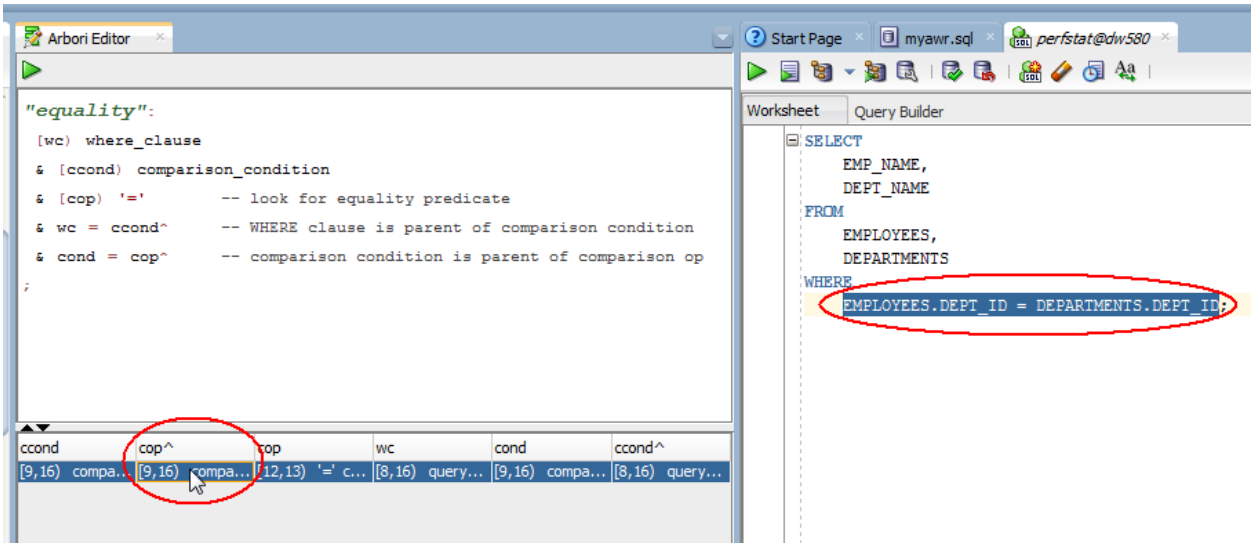
```
"equality":
```

7 Every *parent* is also an *ancestor* but not every *ancestor* is a *parent*

```
 [wc) where_clause
 & [ccond) comparison_condition
 & [cop) '='       -- look for equality predicate
 & wc = ccond^     -- WHERE clause is parent of comparison condition
 & cond = cop^     -- comparison condition is parent of comparison op
;
```



Here the result set has 5 attributes; three for declared variables and two for parent references. Arbori adds referenced nodes as attributes (even if they are duplicates of already existing attributes[8]) which can be helpful for less than trivial queries. Say if we wanted to check the parent of node cop we would click on attribute cop^:



It may sound trivial for our rather simple example but in big queries with complex predicates it can be really handy.

8 In this example node cond^ is by definition wc and node cop^ is cond