The aggregate and buffer effect in the Robust Nearest Neighbor Join operator

Facharbeit im Fach Informatik,
vorgelegt von
Urs Vögeli, Rümlang, Zürich, Schweiz
Matrikelnummer 09-704-404

Angefertigt am
Institut für Informatik
der Universitt Zürich
Prof. Dr. M. Böhlen

Betreuer: Francesco Cafagna

Abgabe der Arbeit: 03.04.2014

Contents

1. Introduction

In this report we will have a look at the robust nearest neighbor join. Given an outer table $\mathbf{r}[G, T, E]$ and an inner table $\mathbf{s}[G, T, E, M]$, the rnn-join operator computes an equijoin on E and a nearest neighbor join on T for the tuples of the outer table without an equijoin match. In the case of a nearest neighbor join the aggregation function Φ aggregates multiple nearest neighbors. The goal of this report is to build an extension to the RNNJ operator that does not aggregate the attribute M for the nearest neighbor join but returns tuples with the M value of every nearest neighbor of the outer tuple. This extension was implemented in the kernel of PostgreSQL and we are now able to query the server as follows:

```
SELECT *
FROM R RNNJ S EQUAL ON E NN BY G USING T [AGG M]
```

Where the omission of AGG M results in the output not being aggregated. We will introduce two ways of implementing the rnn-join without aggregation, one is an implementation with the use of buffers and nearest neighbor sets and the other is an implementation with the use of backtracking and without the use of any buffers and nearest neighbor sets. In the introduction of the first implementation we will give an overview of the algorithm with the help of pseudocode and will show how it was implemented in the kernel of PostgreSQL in the C programming language. In the description of the implementation with backtracking we will focus on the algorithm that was used to realize it and explain it with a running example. We then give an overview of how to compile, install and debug PostgreSQL on the Ubuntu operating system and rum some experiments.

2. Problem Definition And Preliminaries

Definition 1. *Given an outer relation $\boldsymbol{r}$ of schema $R = [E, G, T]$ and an inner relation $\boldsymbol{s}$ of schema $S = [E, G, T, M]$ the rnn-join operator is defined as follows:*

$$\boldsymbol{r} \underset{EQ(E)}{\overset{NN(T),\Phi}{\bowtie}} \boldsymbol{s} = \begin{cases} \boldsymbol{r} \bowtie_{E,G} \boldsymbol{s} \cup ((\boldsymbol{r} \rhd_{E,G} \boldsymbol{s}) \overset{NN(T)}{\bowtie_G} \boldsymbol{s}), & \text{if } \Phi = \emptyset. \\ {}_{s.*}\vartheta_{\Phi(s.M)}(\boldsymbol{r} \bowtie_{E,G} \boldsymbol{s} \cup ((\boldsymbol{r} \rhd_{E,G} \boldsymbol{s}) \overset{NN(T)}{\bowtie_G} \boldsymbol{s})), & \text{otherwise.} \end{cases}$$

where $\bowtie_{E,G}$ is th equijoin operator on E and G, $\rhd_{E,G}$ the antijoin operator on E and G, ϑ the aggregation operator and $\overset{NN(T)}{\bowtie_G}$ is the nearest neighbor operator on T with group G.

The following definitions will be used in the pseudocode of the next chapters:

Definition 2. *Given a tuple of the outer relation r and a tuple of the inner relation s we define the temporal distance $d(r, s)$ as follows:*

$$d(r, s) = |r.T - s.T|$$

Definition 3. *Given a tuple of the outer relation r and a tuple of the inner relation s we define the combination $r \circ s$ as a copy of s with the values of r in E, G and T for the RNNJ with $\Phi = avg$ or as a tuple with the E,G and T values of r and the E,G,T and M values of s for the RNNJ with $\Phi = \emptyset$.*

2.1. **Preliminaries.** In order for the algorithm to work, we have to set some conditions on the two relations. They have to be sorted by (G, T, E) where G and T are the group and the distance attribute in the nearest neighbor join and E is used for the equijoin. Given $r_i \in \mathbf{r}$, sorting the two input relations by (G, T, E) ensures that:

- all tuples $s \in \mathbf{s}$ with $s.G = r_i.G$ will be adjacent elements in $\mathbf{s}$
- all nearest neighbors on T of r_i will be adjacent elements in $\mathbf{s}$
- if an equijoin match for r_i on E exists, then the tuple is also a nearest neighbor on T for r_i
- the nearest neighbor on T of r_{i+1} will always be at the same or at a position following those of r_i

3. Algorithm

The algorithm has been implemented inside the kernel of PostgreSQL. The algorithm is split in four main states which we will describe in the following sections.

3.1. **Initialization.** We start the scan of the two relations by initializing r with the first $\mathbf{r}$ tuple and s_p, s_c, s_n with the first $\mathbf{s}$ tuple. The algorithm will set, during its iterations, s_c to the current $\mathbf{s}$ tuple, s_p to the last fetched tuple with previous T value, and s_n to the next $\mathbf{s}$ tuple. Nearest neighbor sets N_L and N store respectively all outer tuples with $T = s_c.T$ and $G = s_c.G$ and all tuples with $T \in \{s_p.T, s_c.T\}$ and $G = s_c.G$.

State 1: Initialize

```
begin                                   1
    $r \longleftarrow$ fetchRow($\mathbf{r}$)              2
    $s_c \longleftarrow$ fetchRow($\mathbf{s}$)            3
    $s_n \longleftarrow s_c$                          4
    $s_p \longleftarrow s_c$                          5
    $N \longleftarrow \emptyset$                           6
    $N_L \longleftarrow \emptyset$                         7
    Go to NextOuter                     8
end                                     9
```

3.2. **Next Outer.** In this state we scan the outer relation $\mathbf{r}$ until an equijoin match on E is found for s_c: since the tuples are sorted, we can predict that the outer tuple (if any) holding the equality on E with s_c can only be r or a tuple following r. There are three possible cases where the next outer tuple must be fetched:

(1) An equijoin match for s_c exists, therefore we scan $\mathbf{r}$ until the equijoin match is found.
(2) An equijoin match does not exist and all tuples for which s_c is a nearest neighbor must be fetched.
(3) If r has a smaller G than s_c, then no equijoin or nearest neighbor join matches exist for r (since the join matches must belong to the same group G), therefore we keep on scanning $\mathbf{r}$.

If there is equality we produce an output tuple (line 3-5), otherwise we add r to the buffer B_r, which is a small sized buffer storing all and only all outer tuples for which s_c is a nearest neighbor (line 7).

State 2: NextOuter

```
begin                                                                1
    while                                                            2
    $(r.G = s_c.G \wedge (\overbrace{(r.T \leq s_c.T \wedge r.E \leq s_c.E)}^{1} \vee \overbrace{d(r, s_c) < d(r, s_n)}^{2})) \vee \overbrace{r.G < s_c.G}^{3}$
    do
        if $r.G = sc.G$ then                                         3
            if $r.E = s_c.E$ then                                    4
                $z \longleftarrow z \cup (r \circ s_c.M)$            5
            else                                                     6
                append($B_r, r$)                                     7
        $r \longleftarrow$ fetchRow($\mathbf{r}$)                     8
    Go to JoinTuples                                                 9
```

3.3. **Next Inner.** In this state we initially set/reset the nearest neighbor sets (lines 3-9), then we fetch a new inner tuple (lines 10-11). The local nearest neighbor set N_L stores the local nearest neighbors, i.e. the set of all inner tuples with $T = s_c.T$, therefore it is reset if and only if the next inner tuple s_n has a different T value than s_c (line 3). The nearest neighbor set N stores the global nearest neighbors, i.e. the set of all inner tuples in $\mathbf{s}$ with $T \in \{s_c.T, s_n.T\}$, therefore if s_n is closer to r than s_c it is reset otherwise it is set to the old local nearest neighbor set in order to contain all the nearest neighbors of r with $T = r.T - \Delta T$ and $T = r.T + \Delta T$ (line 5). The sets are then updated with the new inner tuple (lines 12-13).

State 3: NextInner

```
begin
    if !$Null(s_n)$ then
        if $s_c.T \neq s_n.T \vee s_c.G < s_n.G$ then
            if $d(r, s_c) = d(r, s_n) \wedge s_n.G = s_c.G$ then
                $N \longleftarrow N_L$
            else
                $N \longleftarrow \emptyset$
            $s_p \longleftarrow s_c$
            $N_L \longleftarrow \emptyset$
        $s_c \longleftarrow s_n$
        $s_n \longleftarrow$ fetchRow($\mathbf{s}$)
        append($N, s_c$)
        append($N_L, s_c$)
    if !$Null(r)$ then
        Go to NextOuter
    else
        Go to JoinTuples
```

3.4. **Join Tuples.** In this state we join each tuple b_r in the buffer B_r with the set of all its nearest neighbors. The buffer is scanned and, for each element b_r, if its distance to s_n is greater than the one to s_c then we are sure that no nearest neighbour for b_r exists any more: output tuples are produced and b_r is removed from the buffer (lines 5-13). The set N_L is used when the nearest neighbors of r have all the same timestamp (lines 10-12) and N is used when r has two nearest neighbours: one for $r.T - \Delta T$ and one for $r.T + \Delta T$ (lines 6-8).
The algorithm ends in the state Join Tuples when no outer tuple to process are left.

State 4: JoinTuples

```
begin                                                                          1
    $b_r \longleftarrow$ first from $B_r$                                      2
    while $!Null(b_r)$ do                                                      3
        if $Null(s_n) \lor d(b_r, s_n) < d(b_r, s_c) \lor (b_r.G = s_c.G \land b_r.G \neq s_n.G)$ then   4
            if $d(b_r, s_p) = d(b_r, s_c) \land s_p.G = s_c.G$ then            5
                tem $\longleftarrow$ first from $N$                            6
                while $!Null(tem)$ do                                          7
                    $z \longleftarrow z \cup b_r \circ$ tem                    8
            else                                                               9
                tem $\longleftarrow$ first from $N_L$                          10
                while $!Null(tem)$ do                                          11
                    $z \longleftarrow z \cup b_r \circ$ tem                    12
            remove $b_r$ from $B_r$                                            13
        $b_r \longleftarrow$ next from $B_r$                                   14
    if $!Null(r) \lor len(B_r) > 0$ then                                       15
        Go to NextInner                                                        16
```

4. Implementation

In this chapter we will show the most important steps taken to implement the algorithm presented in the last chapter.

4.1. *gram.y*.

The first file to alter was */src/backend/parser/gram.y*, which is where one defines grammar rules for queries on a PostgreSQL server. We had to define the syntax for a query using the RNNJ operator so we could query it on a PostgreSQL server as:

```
SELECT *
FROM R RNNJ S EQUAL ON E NN BY G USING T [AGG M]
```

We therefore set the boolean `aggOff` to `FALSE` if we want to aggregate the result and to `TRUE` if we do not. The variable `aggOff` was defined in */src/include/nodes/primnodes.h*, where definitions for primitive node types are made. The first 19 lines of the snippet code from *gram.y* show the syntax for the RNNJ with $\Phi = avg$ with `aggOff` being set to false on line 16. The second part of the snippet shows the syntax for the RNNJ with $\Phi = \emptyset$ with `aggOff` being set to true on line 331.

```
1 ...
table_ref RNNJ table_ref EQUAL ON name_list NN BY  name_list USING
    name AGG name //query pattern
3       {
          JoinExpr *n = makeNode(JoinExpr);
5         n->jointype = JOIN_LEFT;
          n->larg = $1;
7         n->rarg = $3;
          n->equalOnClause = $6;
9         n->nnByClause = $9;
          n->usingClause = list_make1(makeString($11));
          n->aggClause = list_make1(makeString($13));
          n->alias = NULL;
          n->quals = NULL;
          ...
          n->pos_M=-1;
          n->aggOff=FALSE;
          $$ = (Node *) n;
        }
        |  table_ref RNNJ table_ref EQUAL ON name_list NN BY
    name_list USING name
        {
          JoinExpr *n = makeNode(JoinExpr);
          n->jointype = JOIN_LEFT;
          n->larg = $1;
          n->rarg = $3;
          n->equalOnClause = $6;
          n->nnByClause = $9;
          n->usingClause = list_make1(makeString($11));
          n->aggClause = NIL;
            ...
          n->aggOff=TRUE;
          $$ = (Node *) n;
        }
        ...
```

Listing 1. The altered part of gram.y

4.2. ***execProcnode.c.*** The next file to discuss is */src/backend/executor/execProcnode.c*, which contains dispatch functions which call the appropriate ”initialize” (`ExecInitNode`), ”get a tuple” (`ExecProcNode`), and ”cleanup” (`ExecEndNode`) routines for a given node type (e.g. `ExecInitIndexScan` for an index scan or `ExecEndSeqScan` for a sequential scan). In the case of the RNNJ operator we tell the program to execute the function `ExecInitRNNJoinWoAgg` (lines 2-22), `ExecRNNJoinWoAgg` (lines 23-38) and `ExecEndRNNJoinWoAgg` (lines 39-48)from the file */src/backend/executor/nodeRNNJwoAGG.c* if we do not want to aggregate. The value of `aggOff` is transferred here from the *primnodes.h* file via the following files:

- */src/include/nodes/relation.h* (contains definitions for planner’s internal data structures) through */src/backend/optimizer/plan/initsplan.c* (contains target lists, qualifications and joininfo initialization routines)
- */src/include/nodes/plannodes.h* (contains definitions for query plan nodes) through */src/backend/optimizer/plan/createplan.c* (contains routines to create the desired plan for processing a query.)

```
 1 ...
   PlanState *
 3 ExecInitNode(Plan *node, EState *estate, int eflags)
   {
 5 ...
   switch (nodeTag(node))
 7   {
       ...
 9       // join nodes
       ...
11     case T_MergeJoin:
      if ((((MergeJoin *) node)->sample_pos_outer) && (((MergeJoin *)
    node)->aggOff))
        result = (PlanState *) ExecInitRNNJoinWoAgg((MergeJoin *)node,
                                   estate, eflags);
      else if (((MergeJoin *) node)->sample_pos_outer)
        result = (PlanState *) ExecInitRNNJoin((MergeJoin *) node,
                                           estate, eflags);
      else
        result = (PlanState *) ExecInitMergeJoin((MergeJoin *) node,
                              estate, eflags);
      break;
...
TupleTableSlot *
ExecProcNode(PlanState *node)
{
...
switch (nodeTag(node))
  {
    ...
    case T_RNNJoinState:
      if (((RNNJoinState  *) node)->aggOff){
        result = ExecRNNJoinWoAgg((RNNJoinState *) node);
      }
      else{
        result = ExecRNNJoin((RNNJoinState *) node);
      }
      break;
  ...
void
ExecEndNode(PlanState *node)
{
...
case T_RNNJoinState:
      if (((MergeJoin *) node)->aggOff)
        ExecEndRNNJoinWoAgg((RNNJoinState *) node);
      else
        ExecEndRNNJoin((RNJoinState *) node);
      break;
...
```

Listing 2. The altered part of execProcnode.c

4.3. ***parse_clause.c.*** In */src/backend/parser/parse_clause.c*, which handles SQL clauses in the parser, we find the positions of G (`j->usingClause`), T (`j->nnByClause`) and E (`j->equalOnClause`) in the schemas of the outer and the inner tables.

```
...
 /* get position of T in inner and outer input*/
  if(j->nnByClause)
    {
      List  *ucols = j->nnByClause;
      ListCell   *ucol;
      foreach(ucol, ucols)
      {
        char     *u_colname = strVal(lfirst(ucol));
        ListCell   *col;
        int      ndx;
        int      l_index = -1;
        int      r_index = -1;
        Var       *l_colvar ,
        *r_colvar;
        foreach(col , res_colnames)
        {
          char      *res_colname = strVal(lfirst(col));
          ...
        /* Find it in left input */
        ndx = 0;
        foreach(col , l_colnames)
        {
          char      *l_colname = strVal(lfirst(col));
          if (strcmp(l_colname , u_colname) == 0)
          {
            ...
            l_index = ndx;
            j->group_pos_outer=lappend_int(j->group_pos_outer ,l_index
    +1);
          }
          ndx++;
        }
        ...
        /* Find it in right input same as lines s-d with r_colnames
    and r_index */
        ...
        l_colvar = list_nth(l_colvars , l_index);
        l_usingvars = lappend(l_usingvars , l_colvar);
        r_colvar = list_nth(r_colvars , r_index);
        r_usingvars = lappend(r_usingvars , r_colvar);

        res_colnames = lappend(res_colnames , lfirst(ucol));
        res_colvars = lappend(res_colvars ,
            buildMergedJoinVar(pstate ,
                j->jointype ,
                l_colvar ,
                r_colvar));
      }
      /* repeat for G and E */
    ...
```

Listing 3. The altered part of parse_clause.c (part 1)

Then we decide how we want to print the output tuples: if `aggOff` is true (i.e. RNNJ with $\Phi = \emptyset$) we print the G, T and E values of the inner and the outer tuples (lines 3-10), else we only print G, T and E of the outer tuple because we aggregated the M value and do not have a unique value for the T and E entries (line 11-21) .

```
1
    ...
3 if (j->aggOff){  //display G,T and E off both relations
        if (j->alias)
5           {
              *namespace = list_make1(makeDefaultNSItem(rte));
7             list_free(my_relnamespace);
            }
9           else
              *namespace = my_relnamespace;
    }else {       //display G,T and E off outer relations
        if (j->alias != NULL)
          my_relnamespace = NIL;
        else
          setNamespaceColumnVisibility(my_relnamespace, false);
        *namespace = lappend(my_relnamespace,
                    makeNamespaceItem(rte,
                               (j->alias != NULL),
                               true,
                               false,
                               true));
    }
```

Listing 4. The altered part of parse_clause.c (part 2)

4.4. ***nodeRNNJwoAGG.c.*** This is the file where we implement the algorithm. We describe the code by looking at the different states and how we implemented them.

4.4.1. *Initialize.* The initialization is split in two states: one is initialize outer und one is initialize inner. In the initialize outer state we set the current outer tuple (`node->mj_OuterTupleSlot`, line 4) and then go to the initialize inner state where we set the current inner (`node->mj_CurrInnerTupleSlot`, line 11), the previous inner (`node->mj_PrevInnerTupleSlot`, line 12-13) and the next inner tuple slot (`node->mj_InnerTupleSlot`, line 14). We also initialize the nearest neighbor sets (`node->NNSet`, `node->localNNSet`, lines 17-18).

```
   case EXEC_MJ_INITIALIZE_OUTER:
 2
         outerTupleSlot = ExecProcNode(outerPlan);
 4       node->mj_OuterTupleSlot = outerTupleSlot;
         node->mj_JoinState = EXEC_MJ_INITIALIZE_INNER;
 6       break;

 8 case EXEC_MJ_INITIALIZE_INNER:

10       innerTupleSlot = ExecProcNode(innerPlan);
         ExecCopySlot(node->mj_CurrInnerTupleSlot, innerTupleSlot);
12       ExecCopySlot(node->mj_PrevInnerTupleSlot,
             node->mj_CurrInnerTupleSlot);
14       node->mj_InnerTupleSlot = innerTupleSlot;

16       if (!TupIsNull(node->mj_CurrInnerTupleSlot)) {
             node->NNSet=NIL;
18           node->localNNSet=NIL;
         }
20       node->mj_JoinState = EXEC_MJ_NEXTOUTER;
         break;
```

Listing 5. The initialize part of nodeNNJwoAGG.c

4.4.2. *Next Outer.* In this state we first return a tuple if there exists an equijoin on G and E as in line 7 of State 2. We do this with the help of the function `ExecProject`, which stores a tuple in a specified tuple table slot (lines 4-24). On lines 14 and 15 we get the tuples for the join and on line 17 we then create the join based upon the schema we decided on in the *parse_clause.c* file. We then get the next outer tuple with the help of `ExecProcNode`, which gets the next tuple of a table (lines 25-26).

```
 1      case EXEC_MJ_NEXTOUTER:

 3        while (/*line 2-3 of State 2*/) {
            if (RNNJWoAggCompareGroup(node, outerTupleSlot,
 5              node->mj_CurrInnerTupleSlot) == 0) {
              if(RNNJWoAggCompareSample(node, node->mj_OuterTupleSlot,
 7                    node->mj_CurrInnerTupleSlot)==0){
                MemoryContext oldContext;
 9              TupleTableSlot *result;
                ExprDoneCond isDone;
11              oldContext = MemoryContextSwitchTo(
                    node->mj_OuterTupleSlot->tts_mcxt);
13              MemoryContextSwitchTo(oldContext);
                econtext->ecxt_outertuple = node->mj_OuterTupleSlot;
15              econtext->ecxt_innertuple = node->mj_CurrInnerTupleSlot;

17              result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);

19              if (isDone != ExprEndResult) {
                  node->js.ps.ps_TupFromTlist = (isDone
21                      == ExprMultipleResult);
                  if(!TupIsNull(innerTupleSlot) && RNNJWoAggCompareSample(
        node, node->mj_OuterTupleSlot, node->mj_InnerTupleSlot)==0)
23                    node->mj_JoinState = EXEC_MJ_NEXTINNER;
                  else{
25                  outerTupleSlot = ExecProcNode(outerPlan);
                    node->mj_OuterTupleSlot = outerTupleSlot;
27                }
                  return result;
29              }
              }
31            ...
```

Listing 6. Lines 2-5 of State 2 in nodeRNNJwoAGG.c

If we add an outer tuple to the buffer, this is done with a copy of the tuple. For this we use the command `ExecCopySlotTuple`, which obtains a copy of a specific tuple (lines 3-4). Finally we point to the first elements of B_r (`node->outertupsBuffer`), N (`node->NNSet`) and N_L (`node->localNNSet`) (lines 11-16).

```
 1            ...
              else{
 3            node->outertupsBuffer = lappend(node->outertupsBuffer,
                  ExecCopySlotTuple(outerTupleSlot));
 5            }
            }
 7          outerTupleSlot = ExecProcNode(outerPlan);
            node->mj_OuterTupleSlot = outerTupleSlot;
 9        }

11        node->cont = list_head(node->outertupsBuffer);
          node->prev_cont = NULL;
13        node->nnCont=list_head(node->NNSet);
```

```
        node->prev_nnCont=NULL;
        node->localCont=list_head(node->localNNSet);
        node->prev_localCont=NULL;

        node->mj_JoinState = EXEC_MJ_JOINTUPLES;

        break;
```

Listing 7. Lines 6-9 of State 2 in nodeRNNJwoAGG.c

4.4.3. *Next Inner.* Here we set or reset the nearest neighbor sets and set the previous inner tuple with the help of the following commands:

- `list_copy`, which returns a copy of a given list
- `list_free`, which frees a given list and all of its cells
- `ExecCopySlot`, which obtains a copy of a specific tuple

```
  case EXEC_MJ_NEXTINNER:

      if (/* line 2 of State 3*/) {
        bool isNull;
        if (/* line 3 of State 3*/) {
          if (/* line 4 of State 3*/) {
            node->NNSet = list_copy(node->localNNSet);
          } else {
            list_free(node->NNSet);
            node->NNSet = NIL;
          }
          list_free(node->localNNSet);
          node->localNNSet=NIL;
          ExecCopySlot(node->mj_PrevInnerTupleSlot,
              node->mj_CurrInnerTupleSlot);
        }
        ...
```

Listing 8. Lines 2-9 of State 3 in nodeRNNJwoAGG.c

We then set the new values for the current and next inner tuple and add them to the nearest neighbor sets with the help of the following commands:

- `ExecProcNode`, which gets the next tuple of a table
- `ExecCopySlot`, which obtains a copy of a specific tuple
- `lappend`, which adds a given item to a specified list

```
        ...
        ExecCopySlot(node->mj_CurrInnerTupleSlot,
            node->mj_InnerTupleSlot);
        innerTupleSlot = ExecProcNode(innerPlan);
        node->mj_InnerTupleSlot = innerTupleSlot;

        if (TupIsNull(node->mj_OuterTupleSlot) || (
    RNNJWoAggCompareGroup(node, node->mj_OuterTupleSlot, node->
    mj_CurrInnerTupleSlot)==0)) {
          HeapTuple  sampleTupleSlot;
          sampleTupleSlot = ExecCopySlotTuple(node->
    mj_CurrInnerTupleSlot);
          node->NNSet = lappend(node->NNSet,sampleTupleSlot);
          node->localNNSet = lappend(node->localNNSet,
              sampleTupleSlot);
        }
        if (!TupIsNull(node->mj_OuterTupleSlot) &&
   RNNJWoAggCompareGroup(node, node->mj_OuterTupleSlot, node->
   mj_CurrInnerTupleSlot)<=0)
          node->mj_JoinState = EXEC_MJ_NEXTOUTER;
        else
          node->mj_JoinState = EXEC_MJ_JOINTUPLES;
      } else
        return NULL;
      break;
```

Listing 9. Lines 10-17 of State 3 in nodeRNNJwoAGG.c

4.4.4. *Join Tuples.* In this state we create the output tuples. With **ExecStoreTuple** we store a new tuple into a specified slot in the tuple table (**node->mj_InnerTempTupleSlot**, lines 20-21, **node->mj_OuterTempTupleSlot**, lines 26-27) and then we print the output tuple with the help of **ExecProject** (lines 31 and 40-43).Depending on the conditions this is done for the tuples in the local nearest neighbor set or the nearest neighbor set.

```
     case EXEC_MJ_JOINTUPLES:
 2     while (node->cont) { /*line 3 of State 4*/
         if (/* line 4 in State 4*/) {
 4
           MemoryContext oldContext;
 6         TupleTableSlot *result;
           ExprDoneCond isDone;
 8         HeapTuple new_tup;

10         ExecCopySlot(node->mj_InnerTempTupleSlot,
               node->mj_CurrInnerTupleSlot);
12
           if (/*line 5 of State 4*/) {
14           bool isNull;

16           while (node->nnCont){ /*line 7 of State 4*/
               oldContext = MemoryContextSwitchTo(
18                 node->mj_InnerTempTupleSlot->tts_mcxt);
               HeapTuple temp = heap_copytuple(lfirst(node->nnCont));
20             ExecStoreTuple(temp, node->mj_InnerTempTupleSlot,
                   InvalidBuffer, true);
22             MemoryContextSwitchTo(oldContext);
               oldContext = MemoryContextSwitchTo(
24                 node->mj_OuterTempTupleSlot->tts_mcxt);
               HeapTuple temporary =heap_copytuple(lfirst(node->cont));
26             ExecStoreTuple(temporary,
                   node->mj_OuterTempTupleSlot, InvalidBuffer, true);
28             MemoryContextSwitchTo(oldContext);
               econtext->ecxt_outertuple = node->mj_OuterTempTupleSlot;
30             econtext->ecxt_innertuple = node->mj_InnerTempTupleSlot;
               result = ExecProject(node->js.ps.ps_ProjInfo, &isDone);
32
               if ((node->nnSamples->tail == node->nnCont)){
34               node->nnCont = NULL;
               }
36             else{
                 node->prev_nnCont=node->nnCont;
```

```
              node->nnCont = node->prev_nnCont->next;
            }
            if (isDone != ExprEndResult) {
              node->js.ps.ps_TupFromTlist = (isDone
                  == ExprMultipleResult);
              return result;
            }
          }
        } else {
          bool isNull;
          while (node->localCont) {
          ...
            /* the same code as in lines 14-45, only with the local
    variables as in lines 9-12 from State 4*/
            ...
        }
```

LISTING 10. Lines 3-12 of State 4 in nodeRNNJwoAGG.c

We then remove the outer tuple form the buffer with `list_delete_cell`, which frees a cell from a list. If the buffer is not empty we fetch the next tuple from the buffer, otherwise we go to the next state. If the outer relation is empty we do not go to the next state, but stop processing tuples.

```
        ...
        node->outertupsBuffer = list_delete_cell(
            node->outertupsBuffer, node->cont, node->prev_cont);
        if (list_length(node->outertupsBuffer) == 0){
          node->cont = NULL;
        }
        else if (node->prev_cont == NULL){
          node->cont = list_head(node->outertupsBuffer);
          node->nnCont=list_head(node->NNSamples);
          node->prev_nnCont=NULL;
          node->localCont=list_head(node->localNNSamples);
          node->prev_localCont=NULL;
        }
        else{
          node->cont = node->prev_cont->next;
          node->nnCont=list_head(node->nnSamples);
          node->prev_nnCont=NULL;
          node->localCont=list_head(node->localNNSamples);
          node->prev_localCont=NULL;
        }

      } else {
        node->prev_cont = node->cont;
        node->cont = node->cont->next;
      }
    }

    if (list_length(node->outertupsBuffer) == 0
        && TupIsNull(node->mj_OuterTupleSlot))
      return NULL;
    else
      node->mj_JoinState = EXEC_MJ_NEXTINNER;
    node->cont = list_head(node->outertupsBuffer);
    node->prev_cont = NULL;
    node->nnCont =list_head(node->nnSamples);
    node->prev_nnCont = NULL;
    node->localCont =list_head(node->localNNSamples);
    node->prev_localCont = NULL;

    break;
```

Listing 11. Lines 13-16 of State 4 in nodeRNNJwoAGG.c

5. Implementation with backtracking

In a second approach we describe the previous algorithm without the use of buffering and aggregation. We will be able to query the PostgreSQL server as follows:

```
SELECT *
FROM R RNNJ S EQUAL ON E NN BY G USING T BACKTRACK
```

In order to do that, we use backtracking. Backtracking is the process of storing the position of a specific tuple and then returning to that position when needed. The algorithm is again split up in the four different states from the previous chapter, but they all behave differently. We use a running example to help understand the different steps in the algorithm.

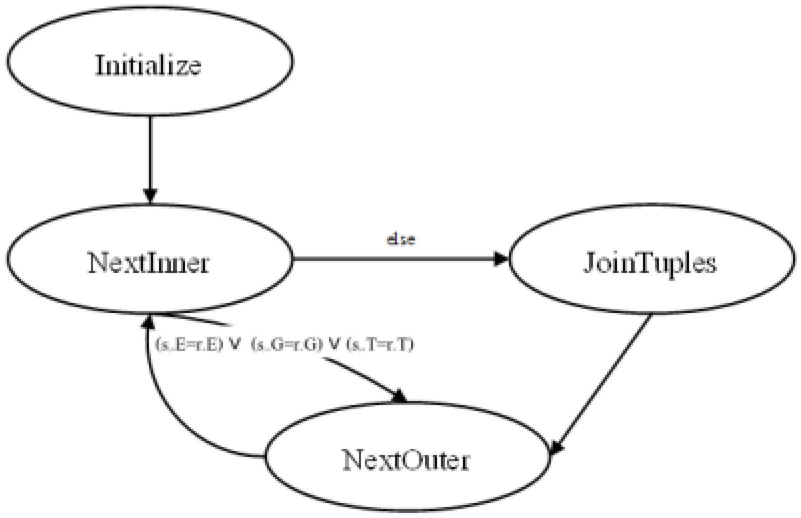

Figure 1. State diagram with backtracking

5.0.5. *Example 1.* In this example we have two tables, one is the outer table r and the inner table s.

Table 1. Example tables

r

G	T	E
Corn	9	18
Corn	13	27

s

G	T	E	M
Corn	4	10	7
Corn	8	14	11
Corn	8	17	9
Corn	10	20	6
Corn	13	24	10
Corn	13	27	8

Implementation of backtracking
The implementation of backtracking is done with the functions `ExecMarkPos` and `ExecRestrPos`. `ExecMarkPos` marks the position, that `ExecRestrPos` later returns to. Since `ExecRestrPos` does not return a tuple but only goes to the previously marked position, the first call of `ExecProcNode` returns the tuple after the one we wanted get. That is why we store a copy of the marked tuple every time we call `ExecMarkPos` and set s_c to said copy after the call of `ExecRestrPos` and s_n to the next tuple of the `ExecProcNode` function.

```
...
ExecCopySlot(node->mj_MarkedTempTupleSlot, node->mj_CurrInnerTupleSlot
    );
ExecMarkPos(innerPlan);
...
ExecRestrPos(innerPlan);
ExecCopySlot(node->mj_CurrInnerTupleSlot, node->mj_MarkedTempTupleSlot
    );
...
```

Listing 12. Example of backtracking in nodeRNNJwoBUFF.c

5.1. **Initialize.** In this state, we initialize r with the first tuple of $\mathbf{r}$ and s_c with the first tuple of $\mathbf{s}$ and s_n with the second tuple of $\mathbf{s}$. We also mark the first position in order to be able to backtrack to it later.

```
State 5: Initialize
Input: outer relation $r$, a set of inner partitions $\mathbf{s}$      1
begin                                                                  2
    $r \longleftarrow$ fetchRow($\mathbf{r}$)                          3
    $s_c \longleftarrow$ fetchRow($\mathbf{s}$)                        4
    markposition($s_c$)                                                5
    $s_n \longleftarrow$ fetchRow($\mathbf{s}$)                        6
    Go to NextInner                                                    7
end                                                                    8
```

5.1.1. *Example 2.* In our example we initialize the different tuples: r is the first tuple in $\mathbf{r}$ and s_c and s_n are the first and second tuple of $\mathbf{s}$. We also mark the position of the first tuple, which is s_c.

TABLE 2. Example tables after Initialize

r

G	T	E	
Corn	9	18	r
Corn	13	27	

s

G	T	E	M	
Corn	4	10	7	marked,s_c
Corn	8	14	11	s_n
Corn	8	17	9	
Corn	10	20	6	
Corn	13	24	10	
Corn	13	27	8	

5.2. **Next Inner.** In this state, we see which tuples to match with the selected outer tuple in the following order:

- Check if we have an equijoin match for the tuple s_c with r on E and G. If so we print an output tuple and go to the NextOuter state (lines 2-4 of State 6)
- If there is no equijoin match we search for the nearest neighbors of r. Whenever s_n is closer to r than s_c we mark the position of s_n, since s_c cannot be a nearest neighbor (lines 5-9 of State 6).
- Once the temporal distance between s_n and r is larger than between s_c and r we know all nearest neighbors are found and can restore the position of the first nearest neighbor and go to the next state (lines 10-19) of State6.
- For the case that there exists an equijoin with the last tuple of a subgroup of **s** with $s.T$ equal for all elements of the subgroup we check for an equijoin on s_c if s_n is null (lines 11-13 of State 6).

State 6: NextInner

```
begin                                                                1
    if $s_c.E = r.E \wedge s_c.G = r.G \wedge s_c.T = r.T$ then      2
        $z \longleftarrow z \cup r \circ s_c$                        3
        Go to NextOuter                                              4
    if $!Null(s_n)$ then                                             5
        if $d(r, s_c) > d(r, s_n) \wedge rGg = s_n.G$ then           6
            markposition($s_n$)                                      7
        $s_c \longleftarrow s_n$                                     8
        $s_n \longleftarrow$ fetchRow(**s**)                         9
    if $Null(s_n) \vee d(r, s_c) < d(r, s_n) \vee (r.G = s_c.G \wedge s_c.G \neq s_n.G)$ then   10
        if $s_c.E = r.E \wedge s_c.G = r.G \wedge s_c.T = r.T$ then  11
            $z \longleftarrow z \cup r \circ s_c$                    12
            Go to NextOuter                                          13
        else                                                         14
            $s_c \longleftarrow$ restoreposition(**s**)              15
            $s_n \longleftarrow$ fetchRow(**s**)                     16
            Go to JoinTuples                                         17
    else                                                             18
        Go to NextInner                                              19
```

5.2.1. *Example 3.* In the NextInner state we first check for an equijoin match between s_c and r without success. After that we go and compare the temporal distance between s_c and r and between s_n and r. Since s_n is closer than s_c we mark the position of s_n and then update s_c and s_n. Because the temporal distance between r and the next two tuples of s is also 1 we stay in NextInner until $s_c.E = 20$ and $s_n.E = 27$. Because $d(r, s_n)$ is larger than $d(r, s_c)$ we restore the marked position, update s_c and s_n and go to the JoinTuples state.

TABLE 3. Example tables after NextInner

r

G	T	E	
Corn	9	18	r
Corn	13	27	

s

G	T	E	M	
Corn	4	10	7	
Corn	8	14	11	marked,s_c
Corn	8	17	9	s_n
Corn	10	20	6	
Corn	13	24	10	
Corn	13	27	8	

5.3. **Join Tuples.** In this state we join the outer tuple r with all of its nearest neighbors. We know that we already restored the inner table to the first nearest neighbor and hence return r joined with s_c. After we return the tuple we check if s_n is also a nearest neighbor, if so we update s_c and s_n and stay in the JoinTuples state (lines 5-8 of State 7). If s_n is not a nearest neighbor we are done joining the tuples for r and go to the NextOuter state to get the next tuple from the outer table (lines 3-4 of State 7).

State 7: JoinTuples

```
begin                                                                                   1
    $z \longleftarrow z \cup r \circ s_c$                                                2
    if $Null(s_n) \vee d(r, s_c) < d(r, s_n) \vee ((r.G = s_c.G) \wedge (s_c.G \neq s_n.G))$ then   3
        Go to NextOuter                                                                 4
    else                                                                                5
        $s_c \longleftarrow s_n$                                                         6
        $s_n \longleftarrow$ fetchRow(s)                                                 7
        Go to JoinTuples                                                                8
```

5.3.1. *Example 4.* We now create the output tuples using the E, G and T values of r and the E, G, T and M values of s_c. Since we have three nearest neighbors we repeat JoinTuples twice and then go to the NextOuter state.

Table 4. Example tables after JoinTuples

r

G	T	E	
Corn	9	18	r
Corn	13	27	

s

G	T	E	M	
Corn	4	10	7	
Corn	8	14	11	marked
Corn	8	17	9	
Corn	10	20	6	s_c
Corn	13	24	10	s_n
Corn	13	27	8	

output

G	T	E	G	T	E	M
Corn	9	18	Corn	8	14	11
Corn	9	18	Corn	8	17	9
Corn	9	18	Corn	10	20	6

5.4. **Next Outer.** In this state we just get the next tuple from the outer table (line 2) and then check if the temporal distance to r of the current inner tuple is closer than the temporal distance to r of the next inner tuple (line 3-4 of State 8). If it is closer, we restore the inner table to the marked tuple and then go to the NextInner state (lines 5-6 of State8), if not we go directly to the NextInner state.

State 8: NextOuter

```
begin                                                  1
    $r \longleftarrow$ fetchRow($\mathbf{r}$)                     2
    if $!Null(r)$ then                                 3
        if $Null(s_n) \vee d(r, s_c) \leq d(r, s_n)$ then   4
            $s_c \longleftarrow$ restoreposition($\mathbf{s}$)    5
            $s_n \longleftarrow$ fetchRow($\mathbf{s}$)           6
        Go to NextInner                                7
```

5.4.1. *Example 5.* First we fetch the next tuple from r and then go directly to NextInner, since we do not have to restore the marked position.

TABLE 5. Example tables after NextOuter

r

G	T	E	
Corn	9	18	
Corn	13	27	r

s

G	T	E	M	
Corn	4	10	7	
Corn	8	14	11	marked
Corn	8	17	9	
Corn	10	20	6	s_c
Corn	13	24	10	s_n
Corn	13	27	8	

output

G	T	E	G	T	E	M
Corn	9	18	Corn	8	14	11
Corn	9	18	Corn	8	17	9
Corn	9	18	Corn	10	20	6

In the NextInner State we mark s_n and then update s_c and s_n, since $s_c.T = s_n.T$ we return to the NextInner state. There is no equijoin match on s_c, so we update s_c and s_n again with s_n being null. We then check for an equijoin match and since it exists we print the output tuple and then go to the NextOuter state. Since r is finished we are done.

TABLE 6. Example tables after NextInner

r

G	T	E	
Corn	9	18	
Corn	13	27	r

s

G	T	E	M	
Corn	4	10	7	
Corn	8	14	11	
Corn	8	17	9	
Corn	10	20	6	marked
Corn	13	27	8	s_c

output

G	T	E	G	T	E	M
Corn	9	18	Corn	8	14	11
Corn	9	18	Corn	8	17	9
Corn	9	18	Corn	10	20	6
Corn	13	27	Corn	13	27	8

6. MANUAL

In the following section we will have a look at how to compile, install and debug PostgreSQL on the Ubuntu operating system. The operation are working on PosgreSQL version 9.3.4.

6.1. **Requirements.**

The following packages are required in order for PostgreSQL to run on Ubuntu and can be installed via the `apt-get install` command in the terminal:

- libreadline-dev
- zlib1g-dev

- bison
- flex

6.2. **Compilation and installation.** The compilation and the installation is also done in the terminal, after navigating to the folder with the source code, with the following commands:

- `./configure --prefix=<PATH TO FOLDER>/server --enable debug`, where `<PATH TO FOLDER>` is the path name of the folder. The `--enable debug` option is needed to debug the program
- `make`, which compiles the programm
- `make install`, which installs the program

We have to set up a database in order to query it. This is done with the following commands:

- `./server/bin/initdb -D dbname`, where dbname can be changed to a desired database name
- `./server/bin/pg_ctl -D dbname -o "-F -i -p 5400" start`, which starts the server and where `-p 5400` can be changed to a free port
- `./server/bin/createdb -p 5400 dbname`
- `./server/bin/pg_ctl -D dbname -o "-F -i -p 5400" stop`, which stops the server

The database can now be started with the `./server/bin/psql -p 5400 dbname` command, once the server is running.

6.3. **Debugging.** To debug PostgreSQL we use the GNU debugger (gdb), which can be started in the terminal with `gdb`. We then have to find the process id (PID) of the PostgreSQL database running on Ubuntu. The quickest way to find it is to use the command `ps -ef | grep postgres`, `ps -ef` shows all processes running on the computer and `grep postgres` filters them to the processes containing postgres in the name. The output of the previous command gives us the specific PID of the database and we can now attach gdb to the database with `attach PID`, where PID is the process id of the database. Once we attached gdb to the programm we can now execute queries on the database and if a query causes the database to crash, we can see which functions were called and where they were called from with the help of the `continue` and `backtrace` commands.

7. Example queries

We now show the output of our queries on the following tables:

Table 7. Tables used for the queries

r

G	T	E
A	10	1
A	10	2
A	10	6
A	11	3
A	12	8
A	12	9
B	4	4

s

G	T	E	M
A	9	0	7
A	10	0	9
A	10	2	5
A	10	6	7
A	12	8	100
A	12	8	200
B	3	0	7
B	5	5	9

First the RNNJ operator with $\Phi = avg$:

G	T	E	M
A	10	2	5
A	10	6	7
A	10	1	7
A	12	8	100
A	11	3	64.2
A	12	9	150
B	5	5	9
B	4	4	8

Figure 2. `SELECT * FROM R RNNJ S EQUAL ON E NN BY G USING T AGG M`

The RNNJ operator with $\Phi = \emptyset$:

G	T	E	G	T	E	M
A	10	2	A	10	2	5
A	10	6	A	10	6	7
A	10	1	A	10	0	9
A	10	1	A	10	2	5
A	10	1	A	10	6	7
A	12	8	A	12	8	100
A	11	3	A	10	0	9
A	11	3	A	10	2	5
A	11	3	A	10	6	7
A	11	3	A	12	13	200
A	11	3	A	12	8	100
A	12	9	A	12	13	200
A	12	9	A	12	8	100
B	5	5	B	5	5	9
B	4	4	B	3	0	7
B	4	4	B	5	5	9

Figure 3. `SELECT * FROM R RNNJ S EQUAL ON E NN BY G USING T [BACKTRACK]`

8. Experiments

In this chapter we test the different implementations of the RNNJ operator and compare their computation times on tables of different sizes. First we compare the two implementations covered in this report without any aggregation, then we will compare the RNNJ with $\Phi = avg$ operator to the RNNJ operator with $\Phi = \emptyset$. The tests were run on eleven different outer tables (**r100, r1000, r2000,** $\ldots$, **r10000**) where the number in the name indicates the size of the table.

8.1. **RNNJ with $\Phi = \emptyset$.**

The results are shown in Figure 2. For $|\mathbf{r}| = 100$ the runtime of the different implementations are the same but the slope of the implementation with backtracking is higher,so the bigger **r** gets, the larger the time difference between the two implementation gets. For $|\mathbf{r}| = 10000$ we have a time difference of approximately 2.7 seconds, so the buffered implementation takes less than half the time than the one with backtracking.

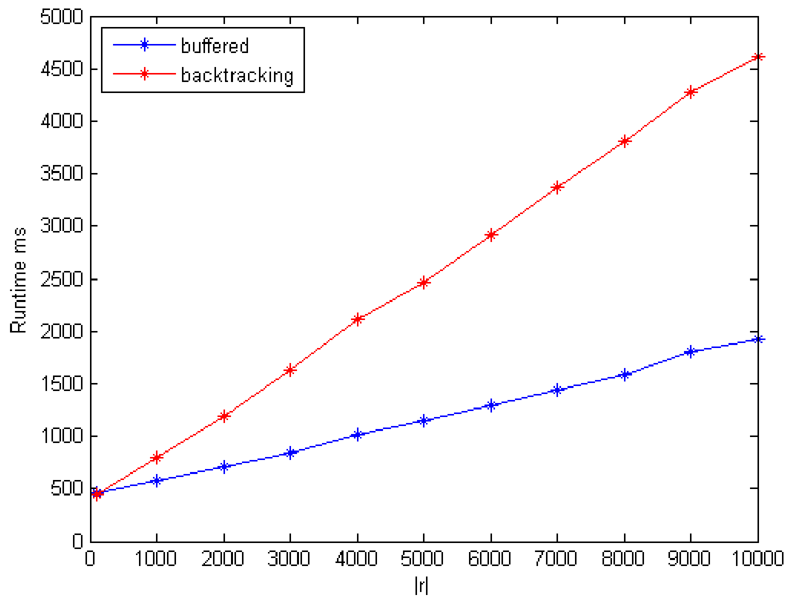

FIGURE 4. Runtime of $\mathbf{r} \bowtie_{E,G} \mathbf{s} \cup ((\mathbf{r} \triangleright_{E,G} \mathbf{s}) \overset{NN(T)}{\bowtie_G} \mathbf{s})$ using a buffferd implementation and one with backtracking

8.2. **RNNJ with $\Phi = avg$.** The results are shown in Figure 3. For $|\mathbf{r}| = 100$ the runtime of the different implementations are the same. The the slope of the implementation with backtracking is steeper than the slope of the implementation without backtracking. At $|\mathbf{r}| = 7000$ the time difference between the two implementations is approximately 2 seconds at $|\mathbf{r}| = 8000$ there is a large increase in runtime of the implementation with backtracking. The time difference at $|\mathbf{r}| = 10000$ is approximately 17 seconds. This difference occurs because at $|\mathbf{r}| = 8000$ the PostgreSQL server changes from hash aggregation, which sorts the tuples in the main memory, to group aggregation which sorts the tuples in the hard disk.

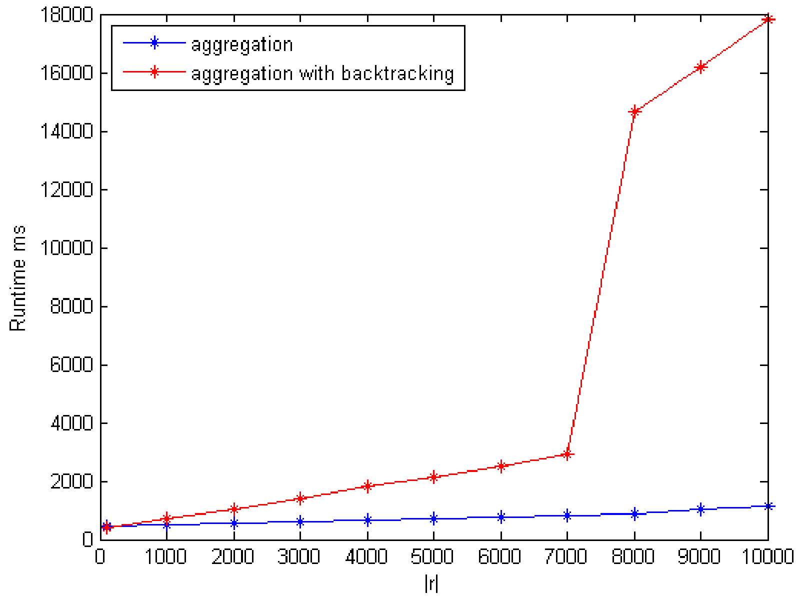

FIGURE 5. Runtime of ${}_{\mathbf{s}.*}\vartheta_{\Phi(\mathbf{s}.M)}(\mathbf{r} \bowtie_{E,G} \mathbf{s} \cup ((\mathbf{r} \rhd_{E,G} \mathbf{s}) \overset{NN(T)}{\bowtie_G} \mathbf{s})$ using a buffferd implementation and one with backtracking

9. Conclusion

In this report we had a look at two different implementations of the robust nearest neighbor join without aggregation. The first used the help of buffer and aggregation sets and was closely related to the original robust nearest neighbor join. We showed how we implemented the operator inside the kernel of PostgreSQL and which files we had to change in order for it to work. The second implementation used backtracking and the states were redefined in order to accomodate the challanges of the backtracking mechanism. The testing of the implementations showed that the implementation with backtracking needed more runtime calculate the joins if the size of the outer table was large.