Enhancing the Tree Awareness of a Relational DBMS

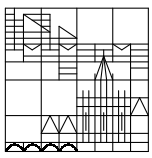
Adding Staircase Join to PostgreSQL

Online Appendix to Master Thesis

Sabine Mayer
Brüelstr. 7, 78462 Konstanz
mayers@inf.uni-konstanz.de

February 2004

Universität Konstanz
Fachbereich Informatik und
Informationswissenschaft



Appendix B

New Source Code in the Planner/Optimizer

B.1 Clause Preparation

B.1.1 check_strcsjoinable()

```
/*    check_strcsjoinable
 *
 *    .../src/backend/optimizer/plan/initsplan.c
 *
 * If the clause represented by the RestrictInfo can be used in a
 * staircase join, set the staircase join info fields in the
 * RestrictInfo accordingly.
 *
 * The staircase join is supported for binary operator clauses where both
 * operands are variables, the variables are of data type tree, and the
 * operator is a 'staircasejoinable' operator ('<', '>', '<=', or '>=').
 */
static void
check_strcsjoinable(RestrictInfo *restrictinfo)
{
    Expr       *clause = restrictinfo->clause;
    Var        *left, *right;
    Oid         opno,
                leftOp,
                rightOp;
    StrcsOpType strcs_optype;

    /* The clause must be an operator expression. */
    if (!is_opclause((Node *) clause))
        return;
```

```
    /* Get the left and right operands of the clause. */
    left = get_leftop(clause);
    right = get_rightop(clause);

                    :

    /* Get the operator of the clause. */
    opno = ((Oper *) clause->oper)->opno;

    /* Check in the system catalog, if this combination of operator
     * and operand types is listed as staircasejoinable. 'strcs_optype'
     * specifies the general operator type of the clause ('STRCS_LT',
     * 'STRCS_GT', 'STRCS_LE', or 'STRCS_GE'). 'leftOp' and 'rightOp'
     * specify the sort order required of the operands.
     */
    strcs_optype = op_strcsjoinable(opno,
                                    left->vartype,
                                    right->vartype,
                                    &leftOp,    /* req. sort orders */
                                    &rightOp);

    /* Set the staircase join info fields in the RestrictInfo. */
    if (strcs_optype)
    {
        restrictinfo->strcsjoinoperator = opno;
        restrictinfo->strcs_optype = strcs_optype;
        restrictinfo->strcs_left_sortop = leftOp;
        restrictinfo->strcs_right_sortop = rightOp;
    }
}
```

B.1.2 op_strcsjoinable()

```
/*    op_strcsjoinable
 *
 *    .../src/backend/utils/cache/lsyscache.c
 *
 * Look up the entry associated to the operator in the system catalog
 * and examine whether the given combination of operator ('opno')
 * and operand types ('ltype'/'rtype') is listed as staircasejoinable.
 *
 * Returns the operator type (STRCS_LT, STRCS_GT, STRCS_LE, or STRCS_GE),
 * if the input operator is 'staircasejoinable' or 0 if not. Apart from
 * that, the sort orders required of the left and right operand of the
 * staircase join are returned.
 *
 *      'opno'    : Oid of examined operator
 *      'ltype'   : data type of left operand
 *      'rtype'   : data type of right operand
 *      '*leftOp' : sort operator required of left operand
```

```
 *      '*rightOp': sort operator required of right operand
 */
StrcsOpType
op_strcsjoinable(Oid opno, Oid ltype, Oid rtype, Oid *leftOp, Oid *rightOp)
{
    HeapTuple   tp;
    StrcsOpType result = STRCS_UNDEF_OP;

    /* Search for the operator's entry in the system catalogue. */
    tp = SearchSysCache(OPEROID,
                        ObjectIdGetDatum(opno),
                        0, 0, 0);

    if (HeapTupleIsValid(tp))
    {
        Form_pg_operator optup = (Form_pg_operator) GETSTRUCT(tp);

        /* Determine the general operator type. */
        if (!strcmp(NameStr(optup->oprname), "<"))
            result = STRCS_LT;
        else if (!strcmp(NameStr(optup->oprname), ">"))
            result = STRCS_GT;
        else if (!strcmp(NameStr(optup->oprname), "<="))
            result = STRCS_LE;
        else if (!strcmp(NameStr(optup->oprname), ">="))
            result = STRCS_GE;

        /* Make sure that the fields specifying the required sort orders
         * are set and that the data type combination of the operands
         * ('ltype'/'rtype') conforms to the specifications in the entry.
         */
        if (result &&
            optup->oprlsortop &&
            optup->oprrsortop &&
            optup->oprleft == ltype &&    /* type casts */
            optup->oprright == rtype)
        {
            *leftOp = optup->oprlsortop;
            *rightOp = optup->oprrsortop;
            ReleaseSysCache(tp);
            return result;
        }

        ReleaseSysCache(tp);
    }
    return result;
}
```

B.2 Dynamic Programming

B.2.1 add_paths_to_joinrel()

```
/*    add_paths_to_joinrel
 *
 *    .../src/backend/optimizer/path/joinpath.c
 *
 * Given two relations that must be joined, consider all possible join
 * paths. If they survive cost-based comparisons with other paths, they
 * are added to the RelOptInfo that accommodates the join ('joinrel').
 *
 * 'joinrel'      represents the current join
 * 'outerrel'     outer relation in the join
 * 'innerrel'     inner relation in the join
 * 'jointype'     INNER, LEFT, FULL, or RIGHT
 * 'restrictlist' join clauses
 */
void
add_paths_to_joinrel(Query *root,
                     RelOptInfo *joinrel,
                     RelOptInfo *outerrel,
                     RelOptInfo *innerrel,
                     JoinType jointype,
                     List *restrictlist)
{
    List       *mergeclause_list = NIL;

    /* Find potential merge join clauses, i.e. equi-join clauses. */
    if (enable_mergejoin || jointype == JOIN_FULL)
        mergeclause_list = select_mergejoin_clauses(joinrel,
                                                    outerrel,
                                                    innerrel,
                                                    restrictlist,
                                                    jointype);

    /* Consider merge join paths where both relations must previously be
     * explicitly sorted.
     */
    sort_inner_and_outer(root, joinrel, outerrel, innerrel,
                         restrictlist, mergeclause_list, jointype);

    /* Consider paths where the outer relation is already suitably
     * sorted (nested-loop and merge joins).
     */
    match_unsorted_outer(root, joinrel, outerrel, innerrel,
                         restrictlist, mergeclause_list, jointype);

    /* Consider paths where both, outer and inner relation, must
     * be hashed before being joined.
```

```
     */
    if (enable_hashjoin)
        hash_inner_and_outer(root, joinrel, outerrel, innerrel,
                             restrictlist, jointype);

    /* Consider staircase join paths. */
    if (enable_strcsjoin && (jointype == JOIN_INNER))
        strcs_inner_and_outer(root, joinrel, outerrel, innerrel,
                              restrictlist, jointype);
}
```

B.2.2 strcs_inner_and_outer()

```
/*    strcs_inner_and_outer
 *
 *    .../src/backend/optimizer/path/joinpath.c
 *
 * Consider the creation of a staircase join path for the current join.
 * If its characteristics (e.g. execution cost, sort order) are superior
 * or equal to any other path created for the same join, the staircase
 * join path is a candidate for the final execution plan. So store it.
 */
static void
strcs_inner_and_outer(Query *root,
                      RelOptInfo *joinrel,
                      RelOptInfo *outerrel,
                      RelOptInfo *innerrel,
                      List *restrictlist,     /* join clauses */
                      JoinType jointype)
{
    List       *outerkey = NIL;
    List       *innerkey = NIL;
    List       *output_pathkey = NIL;
    List       *usable_outers = NIL;
    List       *o;
    Path       *path = NULL;
    List       *strcs_clauses = NIL;

    /* Determine the XPath axis represented by this join and return
     * a list which contains the pre and the post clause.
     */
    StrcsType strcs_type = get_strcs_type(root, outerrel, restrictlist,
                                          &strcs_clauses);

    /* Before a staircase join path is created, make sure that:
     * - the pre and the post clause have been successfully identified,
     * - the inner relation is a base relation (the document table),
     * - and the outer relation is either the initial context set or
     *   the result of the previous staircase join.
     *
```

```
 * This creates left-deep join trees only. The creation of right-
 * deep trees is redundant, because the staircase join's execution
 * module would internally execute them in precisely the same manner
 * at exactly the same execution cost.
 */
if ((strcs_type) && (innerrel->rtekind == RTE_RELATION) &&
    ((outerrel->is_start_context) || (outerrel->strcs_path != NULL)))
{
    /* Build the pathkeys that represent the sort order required
     * of the outer and the inner relation of the staircase join.
     * It is recorded in the pre clause (lfirst(strcs_clauses)).
     */
    outerkey = make_pathkeys_for_strcsclauses(root,
                                              lfirst(strcs_clauses),
                                              outerrel);
    innerkey = make_pathkeys_for_strcsclauses(root,
                                              lfirst(strcs_clauses),
                                              innerrel);

    /* Build path keys that represent the output sort order. The
     * result of the staircase join is guaranteed to be sorted on
     * the outerrels pre value AND the innerrels pre value, but
     * actually we only need the path key of the innerrel, because
     * this relation is the link to the staircase join next-in-line.
     */
    output_pathkey = innerkey;

    /* If the outer relation is the initial context set, we
     * consider the following access paths for it:
     */
    if (outerrel->is_start_context)
    {
        /* Path with cheapest total cost (independent of sorting). */
        usable_outers = lcons(outerrel->cheapest_total_path,
                              usable_outers);

        /* Suitably sorted path with lowest total cost. */
        path = get_cheapest_path_for_pathkeys(outerrel->pathlist,
                                              outerkey,
                                              TOTAL_COST);
        if (path)
            usable_outers = lcons(path, usable_outers);

        /* Suitably sorted path with lowest startup cost. */
        path = get_cheapest_path_for_pathkeys(outerrel->pathlist,
                                              outerkey,
                                              STARTUP_COST);
        if (path)
            usable_outers = lcons(path, usable_outers);
    }
```

```
        /* If the outer relation is the result of the previous
         * staircase join, consider only one path.
         */
        else if (outerrel->strcs_path != NULL)
            usable_outers = lcons((Path *)outerrel->strcs_path,
                                  usable_outers);

        /* Determine the access path to the inner relation. Only
         * the best inner-join index scan is considered.
         */
        path = best_innerjoin_for_strcs(innerrel->innerjoin,
                                        outerrel->relids,
                                        innerkey);

        /* Form all combinations of the outer paths and the inner path. */
        if (path)
        {
            foreach(o, usable_outers)
            {
                Path       *outerpath = (Path *) lfirst(o);

                /* Create a new staircase join path. It stores all
                 * the information required for the execution of the
                 * corresponding plan. If the estimated execution cost
                 * of the new staircase join path (always 0 for now)
                 * is cheaper than the cost of an already existing path
                 * for the same join and/or if the new path has a better
                 * sort order, add_path() stores it within 'joinrel'.
                 */
                add_path(joinrel, (Path *)
                         create_strcsjoin_path(root, joinrel,
                                               jointype, strcs_type,
                                               outerpath, path,
                                               restrictlist,
                                               strcs_clauses,
                                               output_pathkey,
                                               outerkey, innerkey));
            }
        }
    }
}
```

B.2.3 get_strcs_type()

```
/*    get_strcs_type
 *
 *    .../src/backend/optimizer/path/joinpath.c
 *
 * Return the XPath axis which is represented by the input list of
 * join clauses ('strcs_clauses') along with a separate list
 * containing only the pre and the post clause.
 *
 * First, iterate over the list of all join clauses and extract the
 * two staircase join clauses. Then use the two clauses to determine
 * the XPath axis (STRCS_PREC, STRCS_FOL, STRCS_DES,STRCS_DES_OR_SELF,
 * STRCS_ANC, or STRCS_ANC_OR_SELF).
 */
static StrcsType get_strcs_type(Query *root,
                                RelOptInfo *outerrel,
                                List *restrictlist,
                                List **strcs_clauses)
{
    StrcsType result = STRCS_UNDEF_TYPE;
    RestrictInfo *info;
    RestrictInfo *pre_info = NULL;
    RestrictInfo *post_info = NULL;
    Var          *leftop, *rightop;
    char         *left, *right;
    List         *rinfo;

    /* Extract the pre and post clause from the list of join clauses. */
    foreach(rinfo, restrictlist)
    {
       info = (RestrictInfo *) lfirst(rinfo);

       /* Get the left and right operand of the clause. */
       leftop = get_leftop(info->clause);
       rightop = get_rightop(info->clause);

       /* Make sure that both operands are variables. */
       if (IsA(leftop, Var) && IsA(rightop, Var))
       {
           /* Get the name of the attribute represented by the left and
            * right operand (must both be "pre" or "post", respectively).
            */
           left = get_attname(getrelid(leftop->varno, root->rtable),
                              leftop->varattno);
           right = get_attname(getrelid(rightop->varno, root->rtable),
                               rightop->varattno);

           /* Make sure:
```

```
         * - that both operands refer to the pre or post attribute,
         *   respectively,
         * - that the connecting operator is staircasejoinable, and
         * - that the left operand refers to the outer relation (i.e.
         *   the context set, which must always be refered to on the
         *   left side of an operator clause, so that we can deduce
         *   the correct axis represented by the clause).
         */
        if (!strcmp("pre", left) && !strcmp(left, right) &&
            info->strcsjoinoperator != InvalidOid &&
            intMember(leftop->varno, outerrel->relids))
        {
            /* We do not allow more than one pre clause. */
            if (pre_info != NULL)
                return result;
            pre_info = info;
        }

        else if (!strcmp("post", left) && !strcmp(left, right) &&
                 info->strcsjoinoperator != InvalidOid &&
                 intMember(leftop->varno, outerrel->relids))
        {
            /* We do not allow more than one post clause. */
            if (post_info != NULL)
                return result;
            post_info = info;
        }
    }
}

/* This cannot be a staircase join. */
if ((pre_info == NULL) || (post_info == NULL))
    return result;

/* Build a separate list of the staircase join clauses. */
*strcs_clauses = lappend(*strcs_clauses, pre_info);
*strcs_clauses = lappend(*strcs_clauses, post_info);

/* Check which XPath axis is represented by the pre and post
 * clause, if any.
 */

/* descendant, ancestor, following, and preceding */
if (pre_info->strcs_optype == STRCS_LT)
{
    if (post_info->strcs_optype == STRCS_GT)
        result = STRCS_DES;    /* descendant */
    else if (post_info->strcs_optype == STRCS_LT)
        result = STRCS_FOL;    /* following */
}
```

```
    else if (pre_info->strcs_optype == STRCS_GT)
    {
        if (post_info->strcs_optype == STRCS_GT)
            result = STRCS_PREC;    /* preceding */
        else if (post_info->strcs_optype == STRCS_LT)
            result = STRCS_ANC;    /* ancestor */
    }

    /* descendant-or-self and ancestor-or-self */
    if (pre_info->strcs_optype == STRCS_LE)
    {
        if (post_info->strcs_optype == STRCS_GE)
            result = STRCS_DES_OR_SELF;    /* descendant-or-self */
    }
    else if (pre_info->strcs_optype == STRCS_GE)
    {
        if (post_info->strcs_optype == STRCS_LE)
            result = STRCS_ANC_OR_SELF;    /* ancestor-or-self*/
    }

    return result;
}
```

B.2.4 make_pathkeys_for_strcsclauses()

```
/*    make_pathkeys_for_strcsclauses
 *
 *    .../src/backend/optimizer/path/pathkeys.c
 *
 * Retrieves a path key list representing the explicit sort order that
 * an input relation (a Path) must have in order to make it usable in
 * a staircase join.
 *
 * 'strcsclause' is the pre clause of the staircase join. The post
 *               clause must not be taken into account, because the
 *               staircase join's input is only required to be sorted
 *               on the pre value.
 * 'rel'         is the relation to which the pathkeys must be applied,
 *               either the inner or outer input relation of the join.
 */
List *
make_pathkeys_for_strcsclauses(Query *root,
                               RestrictInfo *strcsclause,
                               RelOptInfo *rel)
{
    Node       *key;
    List       *pathkey;

    /* Build the path keys for both operands of the pre clause. */
    cache_strcsclause_pathkeys(root, strcsclause);
```

```
    key = (Node *) get_leftop(strcsclause->clause);
    if (IsA(key, Var) &&
        VARISRELMEMBER(((Var *) key)->varno, rel))
    {
        /* 'rel' is the left (outer) input relation of the join. */
        pathkey = strcsclause->strcs_left_pathkey;
    }
    else
    {
        key = (Node *) get_rightop(strcsclause->clause);
        if (IsA(key, Var) &&
            VARISRELMEMBER(((Var *) key)->varno, rel))
        {
            /* 'rel' is the right (inner) input relation of the join. */
            pathkey = strcsclause->strcs_right_pathkey;
        }
        else
        {
            elog(ERROR, "make_pathkeys_for_strcsclauses: can't identify"
                        " which side of strcsclause to use");
            pathkey = NIL;
        }
    }

    /* Path keys must be a list of lists. */
    return makeList1(pathkey);
}
```

B.2.5 cache_strcsclause_pathkeys()

```
/*    cache_strcsclause_pathkeys
 *
 *    .../src/backend/optimizer/path/pathkeys.c
 *
 * Build the pathkeys that specify the sort order which is required of
 * the left (outer) and right (inner) input relation of a staircase join.
 * Store them in the respective RestrictInfo fields.
 *
 * A PathkeyItem consist of a 'key' and a sort operator. The key
 * specifies the ids of the relation and the attribute to which the sort
 * operator must apply.
 */
void
cache_strcsclause_pathkeys(Query *root, RestrictInfo *restrictinfo)
{
    Node        *key;
    PathKeyItem *item;

    Assert(restrictinfo->strcsjoinoperator != InvalidOid);
```

```
    if (restrictinfo->strcs_left_pathkey == NIL)
    {
        /* Build the pathkey for the outer relation. */
        key = (Node *) get_leftop(restrictinfo->clause);
        item = makePathKeyItem(key, restrictinfo->strcs_left_sortop);

        restrictinfo->strcs_left_pathkey = make_canonical_pathkey(root,
                                                                  item);
    }
    if (restrictinfo->strcs_right_pathkey == NIL)
    {
        /* Build the pathkey for the inner relation. */
        key = (Node *) get_rightop(restrictinfo->clause);
        item = makePathKeyItem(key, restrictinfo->strcs_right_sortop);

        restrictinfo->strcs_right_pathkey = make_canonical_pathkey(root,
                                                                   item);
        }
}
```

B.2.6 create_strcsjoin_path()

```
/*    create_strcsjoin_path
 *
 *    .../src/backend/optimizer/util/pathnode.c
 *
 * Creates a join path corresponding to a staircase join between
 * two relations.
 *
 * 'joinrel'          accomodates the join
 * 'jointype'         type of join required
 * 'strcs_type'       XPath axis represented by the join
 * 'outer_path'       outer input relation
 * 'inner_path'       inner input relation
 * 'restrict_clauses' all the RestrictInfo nodes to apply to the join
 * 'strcs_clauses'    the pre and post clause
 * 'pathkeys'         output path keys of the new join path
 * 'outersortkeys'    path keys of the outer relation
 * 'innersortkeys'    path keys of the inner relation
 */
StrcsPath *
create_strcsjoin_path(Query *root,
                      RelOptInfo *joinrel,
                      JoinType jointype,
                      StrcsType strcs_type,
                      Path *outer_path,
                      Path *inner_path,
                      List *restrict_clauses,
                      List *strcs_clauses,
```

```
                    List *pathkeys,
                    List *outersortkeys,
                    List *innersortkeys)
{
    int         costcmp;
    StrcsPath  *pathnode = makeNode(StrcsPath);

    /* If the outer path is already well enough sorted, we can skip
     * doing an explicit sort when the execution plan is created.
     */
    if (outersortkeys &&
        pathkeys_contained_in(outersortkeys, outer_path->pathkeys))
        outersortkeys = NIL;

    /* The inner path is suitably sorted in any case (inner-join
     * index). Otherwise, the creation of the join path would not
     * have been initiated.
     */
    innersortkeys = NIL;

    pathnode->jpath.path.pathtype = T_StrcsJoin;
    pathnode->jpath.path.parent = joinrel;
    pathnode->jpath.jointype = jointype;
    pathnode->jpath.outerjoinpath = outer_path;
    pathnode->jpath.innerjoinpath = inner_path;
    pathnode->jpath.joinrestrictinfo = restrict_clauses;
    pathnode->jpath.path.pathkeys = pathkeys;
    pathnode->outersortkeys = outersortkeys;
    pathnode->innersortkeys = innersortkeys;
    pathnode->type = strcs_type;
    pathnode->strcs_clauses = strcs_clauses;

    /* Calculate cost of staircase join path. Always 0 for the moment. */
    cost_strcsjoin(&pathnode->jpath.path, root,
                   outer_path, inner_path,
                   restrict_clauses,
                   outersortkeys, innersortkeys);

    /* Cache the cheapest staircase path separately. This is not the
     * cheapest overall path for the join, but only the cheapest
     * staircase join path. It is stored separately for easier
     * identification. The staircase join next-in-line will use it
     * as outer relation.
     * If old and new path have the same total cost, compare their
     * startup cost and choose the cheapest of these.
     */
    if (!joinrel->strcs_path)
        joinrel->strcs_path = pathnode;
```

```
    else
    {
        costcmp = compare_path_costs((Path *)pathnode,
                                     (Path *)joinrel->strcs_path,
                                     TOTAL_COST);
        /* The new path has cheaper total cost. */
        if (costcmp < 0)
            joinrel->strcs_path = pathnode;
        /* Old and new path have the same cost, so compare startup cost. */
        else if (costcmp == 0)
        {
            costcmp = compare_path_costs((Path *)pathnode,
                                         (Path *)joinrel->strcs_path,
                                         STARTUP_COST);
            if (costcmp < 0)
                joinrel->strcs_path = pathnode;
        }
    }

    return pathnode;
}
```

B.3 Conversion into an Execution Plan

B.3.1 create_strcsjoin_plan()

```
/*    create_strcsjoin_plan
 *
 *    .../src/backend/optimizer/plan/createplan.c
 *
 * Creates a staircase join plan. Prepares the join and index clauses
 * for the evaluation mechanisms of the executor and inserts explicit
 * sort nodes, if necessary.
 *
 * 'root'         the query's parse tree
 * 'best_path'    the staircase join path which was selected as optimal
 *                path for this join
 * 'tlist'        the target list of the join
 * 'joinclauses'  all the clauses refering to this join
 * 'otherclauses' clauses from OUTER joins, always NIL here
 * 'outer_plan'   plan of the outer input relation
 * 'outer_tlist'  target list of the outer plan
 * 'inner_plan'   plan of the inner input relation (inner-join index)
 * 'inner_tlist'  target list of the inner plan
 */
static StrcsJoin *
create_strcsjoin_plan(Query *root,
                      StrcsPath *best_path,
                      List *tlist,
```

```
                    List *joinclauses,
                    List *otherclauses,
                    Plan *outer_plan,
                    List *outer_tlist,
                    Plan *inner_plan,
                    List *inner_tlist)
{
    StrcsJoin *join_plan;

    /* Extract the pre and post clause from the staircase join path and
     * make a copy of them. They are needed in the executor to create
     * and evaluate predicates in addition to the actual join clauses.
     */
    List *strcs_clauses = get_actual_clauses(best_path->strcs_clauses);
    List *orig_clauses = (List *) copyObject((Node *) strcs_clauses);

    /* Prepare them to the evalaution mechanisms of the executor, i.e.
     * change the explicit references to relation ids into OUTER or INNER.
     */
    orig_clauses = join_references(orig_clauses, root->rtable,
                                   outer_tlist, inner_tlist, (Index) 0);

    /* To prevent duplicate evaluation, remove the pre and post clause
     * from the list of other join clauses, that may be present.
     */
    joinclauses = set_difference(joinclauses, strcs_clauses);

    /* Preparation of index clauses. */
    if (IsA(inner_plan, IndexScan)) /* only possibility */
    {
        IndexScan  *innerscan = (IndexScan *) inner_plan;
        List       *indxqualorig = innerscan->indxqualorig;

        /* If only one relation is referenced in the index clauses, the
         * index does not evaluate join clauses (but only selections).
         */
         if (NumRelids((Node *) indxqualorig) > 1)
         {
             Index           innerrel = innerscan->scan.scanrelid;

             /* The ancestor axes require special treatment. In this
              * case, the pre clause must be removed from the index
              * and evaluated as a join clause. Instead of this, the
              * index is supplied with a new clause which enables
              * ancestor axis skipping.
              *
              * What we really do is rewrite the index pre clause:
              * outer.pre  >  inner.pre     becomes
              * inner.post <= inner.pre.
              *
```

```
* This process will have to be carried out twice,
* because two versions of the index clauses are kept,
* a modified and an unmodified one.
*/
if ((best_path->type == STRCS_ANC) ||
    (best_path->type == STRCS_ANC_OR_SELF))
{
    List       *list;
    Expr       *o_qexpr = NULL;
    Expr       *qexpr = NULL;
    Expr       *pre_oexpr, *pre_expr, *post_expr, *e;
    Oper       *o_op, *op;
    Oid         o_opno, opno, new_opno;
    Var        *left, *right, *post_right;

    indxqualorig = (List*) copyObject(innerscan->indxqualorig);

    /* Prepare the index clauses to the evalaution mechanisms
     * of the executor, i.e. change the explicit references
     * to the outer relation into OUTER.
     */

    ⋮

    /* Now start to rewrite the pre clause. */
    pre_expr = (Expr *)lfirst(strcs_clauses);
    pre_oexpr = (Expr *)lfirst(orig_clauses);

    /* Get the post clause and extract its right operand
     * (inner.post). It will replace the outer.pre operand
     * of the pre clause.
     */
    post_expr = (Expr *)lsecond(orig_clauses);
    post_right = (Var *) get_rightop(post_expr);

    /* Extract the original pre clause from the index. */
    foreach(list, lfirst(innerscan->indxqualorig))
    {
        e = (Expr *) lfirst(list);

        if (equal(get_leftop(pre_oexpr), get_leftop(e)) &&
            equal(get_rightop(pre_expr), get_rightop(e)))
        {
            o_qexpr = (Expr *)lfirst(list);
            break;
        }
    }

    Assert(o_qexpr != NULL);
```

```
/* Get the left operand of the pre clause (outer.pre).
 * It is to be replaced by inner.post.
 */
left = (Var *) get_leftop(o_qexpr);

/* Get the connecting operator and oid (> or >=). */
o_op = (Oper *) o_qexpr->oper;
o_opno = o_op->opno;

/* Extract the modified pre clause from the index. */
foreach(list, lfirst(innerscan->indxqual))
{
    e = (Expr *) lfirst(list);

    if (equal(get_leftop(pre_oexpr), get_rightop(e)) &&
        equal(get_rightop(pre_expr), get_leftop(e)))
    {
        qexpr = (Expr *)lfirst(list);
        break;
    }
}

Assert(qexpr != NULL);

/* Get the right operand of the pre clause (outer.pre).
 * It is to be replaced by inner.post.
 */
right = (Var *) get_rightop(qexpr);

/* Get the connecting operator and oid (< or <=). */
op = (Oper *) qexpr->oper;
opno = op->opno;

/* Switch the operands. */
 *left = *post_right;
 *right = *post_right;

/* Now exchange the operators. In case of the ancestor
 * axis, 'o_qexpr' has the '>' operator and 'qexpr'
 * has the '<' operator. What we want is '<=' in
 * 'o_qexpr' and '>=' in 'qexpr'. That's why we get
 * the negators.
 */
if (best_path->type == STRCS_ANC)
{
    o_opno = get_negator(o_opno);
    opno = get_negator(opno);
}
```

```
            /* In case of the ancestor_or_self axis, 'o_qexpr' has
             * the '>=' operator and 'qexpr' has the '<=' operator
             * What we want is '<=' in 'o_qexpr' and '>=' in
             * 'qexpr'. That's why we swap the operators.
             */
            else if (best_path->type == STRCS_ANC_OR_SELF)
            {
                new_opno = opno;
                opno = o_opno;
                o_opno = new_opno;
            }

            /* Now, replace the operators. */
            o_op->opno = o_opno;
            op->opno = opno;
        }

        /* For all other axes, we must only remove from the join
         * clauses all predicates that are evaluated in the index.
         */
        else
        {
            joinclauses = set_difference(joinclauses,
                                         lfirst(indxqualorig));
            strcs_clauses = set_difference(strcs_clauses,
                                           lfirst(indxqualorig));

            /* Prepare the index clauses to the evalaution mechanisms
             * of the executor, i.e. change the explicit references
             * to the outer relation into OUTER.
             */

            ⋮

         }
}

/* Prepare the join clauses to the evalaution mechanisms, i.e.
 * change the references to relation ids into OUTER or INNER.
 */

          ⋮

/* Create explicit sort nodes for the outer and inner join paths,
 * if necessary. Should never be the case for the inner relation.
 */
if (best_path->outersortkeys)
    outer_plan = (Plan *)
                 make_sort_from_pathkeys(root, outer_tlist,
                                         outer_plan,
```

```
                                                best_path->outersortkeys);

        if (best_path->innersortkeys)
            inner_plan = (Plan *)
                        make_sort_from_pathkeys(root, inner_tlist,
                                                inner_plan,
                                                best_path->innersortkeys);


        /* Now we can build the staircase join plan node. */
        join_plan = make_strcsjoin(tlist, joinclauses, strcs_clauses,
                                   orig_clauses, otherclauses,
                                   outer_plan, inner_plan,
                                   best_path->jpath.jointype,
                                   best_path->type);

        copy_path_costsize(&join_plan->join.plan, &best_path->jpath.path);

        return join_plan;
}
```

Appendix D

New Source Code in the Executor

D.1 Initialization of Staircase Join Execution

D.1.1 ExecInitStrcsJoin()

```
/*
 *    ExecInitStrcsJoin
 *
 *    .../src/backend/executor/nodeStrcsjoin.c
 *
 * Initialize the private state information for this staircase
 * join plan node. This opens files, allocates storage and
 * leaves us ready to start processing tuples.
 */
bool
ExecInitStrcsJoin(StrcsJoin *node, EState *estate, Plan *parent)
{
    StrcsJoinState *strcsstate;

    /* Assign the node's execution state. It contains general
     * execution information, such as the overall scan direction
     * (forwards or backwards) and a pointer to the result tuple
     * table.
     */
    node->join.plan.state = estate;

    /* Create a new staircase join state for the current node whose
     * fields will be initialzed during execution. It will store
     * information on the current outer and inner tuple, the result
     * tuple and contains projection information.
     */
```

```
    strcsstate = makeNode(StrcsJoinState);
    node->strcsstate = strcsstate;

    /* Miscellaneous initialization.
     *
     * Create an expression context for the current node. Required
     * for nodes that involve projections and evaluate expressions
     * (e.g. join clauses).
     */
    ExecAssignExprContext(estate, &strcsstate->jstate);

    /* Recursively initialize the subplans. */
    ExecInitNode(outerPlan((Plan *) node), estate, (Plan *) node);
    ExecInitNode(innerPlan((Plan *) node), estate, (Plan *) node);

    /* This is the number of tuple slots required in the tuple table
     * to perform this join. In case of the staircase join, we will be
     * working with:
     * - one tuple slot for the result of the join (initialized below
     *   by ExecInitResultTupleSlot())
     * - two input tuples from the outer and inner relation (as these
     *   are the result tuples of another operation (e.g. a scan or
     *   another join, a slot was already reserved for them by the
     *   subplans)
     * - one tuple slot which is used to store a previously processed
     *   tuple (e.g. the second boundary of a partition; initialized
     *   below by ExecInitExtraTupleSlot())
     * - one special tuple slot used for the first rescan in
     *   connection with the ancestor axis.
     *
     * Makes 3 extra tuples.
     */
#define STRCSJOIN_NSLOTS 3

    /* Initiate the table slot for the result tuple. */
    ExecInitResultTupleSlot(estate, &strcsstate->jstate);

    /* Do the same for the other two extra table slots. */
    strcsstate->sc_PreviousTupleSlot = ExecInitExtraTupleSlot(estate);

    ExecSetSlotDescriptor(strcsstate->sc_PreviousTupleSlot,
                          ExecGetTupType(outerPlan((Plan *) node)), false);

    strcsstate->sc_RescanTupleSlot = ExecInitRescanTupleSlot(estate,
                            ExecGetTupType(innerPlan((Plan *) node)));

    /* Initialize tuple type and projection info. */

    /* Generate a tuple descriptor for the result tuple of the
     * staircase join node using the node's targetlist. Associate
```

```
     * the tuple descriptor with the result tuple slot created above.
     */
    ExecAssignResultTypeFromTL((Plan *) node, &strcsstate->jstate);

    /* Create the projection info using the node's targetlist. */
    ExecAssignProjectionInfo((Plan *) node, &strcsstate->jstate);

    /* Build up additional predicates that will be needed during
     * join execution to carry outer further comparisons (e.g. to
     * evaluate the second partition boundary in case of the descendant
     * axis or to apply pruning in case of the following axis).
     */
     SCFormSkipQuals(node->orig_joinquals,
                     &strcsstate->sc_LowerSkipQual,
                     &strcsstate->sc_GreaterSkipQual, node->type);

    /* Initialize the join state. */

    /* Switch to the initial state of staircase join execution. */
     strcsstate->sc_JoinState = EXEC_SC_INITIALIZE;

     strcsstate->jstate.cs_TupFromTlist = false;

     /* Table slots reserved for the outer and inner tuple of the join. */
     strcsstate->sc_OuterTupleSlot = NULL;
     strcsstate->sc_InnerTupleSlot = NULL;

     /* Initialization successful. */
        return TRUE;
}
```

D.1.2 SCFormSkipQuals()

```
/*    SCFormSkipQuals
 *
 *    .../src/backend/executor/nodeStrcsjoin.c
 *
 * Take the original pre and post clause and for both get the 'lower
 * than/greater than' operator associated with the tree data type.
 * The original operators of the clauses are then replaced by the
 * new operators, such that we obtain four more clauses:
 * - (pre < pre) and (post < post): ltQuals
 * - (pre > pre) and (post > post): gtQuals
 *
 * They are used to evaluate certain conditions in addition to the
 * staircase join clauses (e.g. for pruning).
 */
static void
SCFormSkipQuals(List *qualList, List **ltQuals, List **gtQuals, StrcsType type)
{
```

```
List   *ltcdr,
       *gtcdr;

/* Make two modifiable copies of the join clause list
 * (e.g. (pre < pre), (post < post)).
 */
*ltQuals = (List *) copyObject((Node *) qualList);
*gtQuals = (List *) copyObject((Node *) qualList);

/* Scan both lists in parallel, so that we can exchange the
 * operators with the minimum number of lookups.
 */
ltcdr = *ltQuals;
foreach(gtcdr, *gtQuals)
{
    Expr       *ltqual = (Expr *) lfirst(ltcdr);
    Expr       *gtqual = (Expr *) lfirst(gtcdr);
    Oper       *ltop = (Oper *) ltqual->oper;
    Oper       *gtop = (Oper *) gtqual->oper;

    /* The two operators ('ltop' and 'gtop') are identical, as we
     * are currently handling one and the same clause ('ltqual' =
     * 'gtqual'), so use either one for the lookup.
     */
    if (!IsA(ltop, Oper))
      elog(ERROR, "SCFormSkipQuals: op not an Operator!");

    /* Look up the 'lower/greater than' operators associated with
     * the tree data type and replace the operators in the original
     * clauses.
     */
    op_strcsjoin_crossops(ltop->opno,  /* the operator to be replaced */
                          &ltop->opno, &gtop->opno,
                          &ltop->opid, &gtop->opid);

            ltop->op_fcache = NULL;
            gtop->op_fcache = NULL;

            ltcdr = lnext(ltcdr);
    }

/* ANCESTOR AXIS: We need a further qualification which compares
 * the pre and the post value of the SAME (inner) tuple, i.e.
 * inner.pre < inner.post. It will make sure that skipping is not
 * directed backwards. So get the second clause from 'ltQuals'
 * (outer.post < inner.post) and replace the left operand by the
 * right operand of the first clause from 'ltQuals' (inner.pre).
 */
if ((type == STRCS_ANC) || (type == STRCS_ANC_OR_SELF))
{
```

```
        Expr       *post = (Expr *) copyObject(lsecond(*ltQuals));
        Expr       *pre  = (Expr *) lfirst(*ltQuals);
        Var        *out_post = (Var *) get_leftop (post);
        Var        *inn_post = (Var *) get_rightop (post);
        Var        *inn_pre  = (Var *) get_rightop (pre);

        *out_post = *inn_pre;

        /* Append it to the end of 'ltQuals'.*/
        *ltQuals = lappend(*ltQuals, post);
    }

    /* DESCENDANT AXIS: We need a further qualification which compares
     * the post values of two outer tuples (outer.post > outer.post).
     * It is required for pruning. So modify the second clause from
     * 'gtQuals' (outer.post > inner.post) accordingly.
     */
    else if ((type == STRCS_DES) || (type == STRCS_DES_OR_SELF))
    {
        Expr *gtqual = (Expr *) lsecond(*gtQuals);
        Expr *mod_gtqual;

        /* Replace inner.post by outer.post. */
        mod_gtqual = SCModSkipQual(gtqual);
        *gtqual = *mod_gtqual;
    }

    /* The same applies to the FOLLOWING AXIS. In this case, we need a
     * further qualification which compares the post values of two outer
     * tuples (outer.post < outer.post). It is also required for pruning.
     * So modify the second clause from 'ltQuals' (outer.post < inner.post)
     * accordingly.
     */
    else if (type == STRCS_FOL)
    {
        Expr *ltqual = (Expr *) lsecond(*ltQuals);
        Expr *mod_ltqual;

        /* Replace inner.post by outer.post. */
        mod_ltqual = SCModSkipQual(ltqual);
        *ltqual = *mod_ltqual;
    }
}
```

D.2 The Staircase Join's Execution Module

D.2.1 ExecStrcsJoin

```
/*    ExecStrcsJoin
 *
 *    .../src/backend/executor/nodeStrcsjoin.c
 *
 * Starts the execution of a staircase join plan. This routine is
 * called by the parent plan to request the next tuple.
 */
TupleTableSlot *
ExecStrcsJoin(StrcsJoin *node)
{
    switch (node->type)
    {
        case STRCS_DES:
        case STRCS_DES_OR_SELF:
            return ExecDescJoin((StrcsJoin *)node);
            break;
        case STRCS_ANC:
        case STRCS_ANC_OR_SELF:
            return ExecAncJoin((StrcsJoin *)node);
            break;
        case STRCS_FOL:
            return ExecFolJoin((StrcsJoin *)node);
            break;
        case STRCS_PREC:
            return ExecPrecJoin((StrcsJoin *)node);
            break;
        default:
            elog(WARNING, "ExecStrcsJoin: invalid staircase join type %d, "
                          "aborting", node->type);
            return NULL;
        }
}
```

D.2.2 ExecDescJoin()

```
/*     ExecDescJoin
 *
 *    .../src/backend/executor/nodeStrcsjoin.c
 *
 * Yields the next tuple from a descendant location step.
 */
static TupleTableSlot *ExecDescJoin(StrcsJoin *node)
{
    EState         *estate;
    StrcsJoinState *strcsstate;
    ScanDirection  direction;
```

```
List           *lowerSkipQual;
List           *greaterSkipQual;
List           *joinclause;        /* the post clause */
List           *more_clauses;      /* other join clauses */
List           *clause;
bool           qualResult;
Plan           *innerPlan;
TupleTableSlot *innerTupleSlot;
Plan           *outerPlan;
TupleTableSlot *outerTupleSlot;
ExprContext    *econtext;

/* Extract information from the staircase join plan node. */
strcsstate = node->strcsstate;
estate = node->join.plan.state;
direction = estate->es_direction;
innerPlan = innerPlan((Plan *) node);
outerPlan = outerPlan((Plan *) node);
econtext = strcsstate->jstate.cs_ExprContext;

/* There is only one staircase join clause left, namely the post
 * clause. The pre clause has become an index clause.
 */
joinclause = node->join.joinqual;

              ⋮

/* Extract the preprocessed additional clauses. */
lowerSkipQual = strcsstate->sc_LowerSkipQual;
greaterSkipQual = strcsstate->sc_GreaterSkipQual;

/* Reset the expression context. */
ResetExprContext(econtext);

/* Loop until we have the next joined tuple. */
for (;;)
{
    /* Get the current state of the join and do things accordingly. */
    switch (strcsstate->sc_JoinState)
    {
        /* Get first context tuple and get ready for pruning. */
        case EXEC_SC_INITIALIZE:

            /* Get the first tuple from the outer relation. */
            outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
            strcsstate->sc_OuterTupleSlot = outerTupleSlot;

            /* Proceed with pruning the context set. */
            strcsstate->sc_JoinState = EXEC_SC_STORE;
            break;
```

```
/* The descendant axis works on partitions that lie between
 * the pre values of two successive pruned tuples. This
 * state stores the upper partition boundary of the previous
 * partition as lower boundary of the current partition.
 */
case EXEC_SC_STORE:

    outerTupleSlot = strcsstate->sc_OuterTupleSlot;

    /* If there is no boundary to pass, end the join. */
    if (TupIsNull(outerTupleSlot))
        return NULL;


    /* Store the upper boundary of the previous partition
     * as lower boundary of the current partition.
     */
    CachePreviousTuple(strcsstate->sc_OuterTupleSlot, strcsstate);
    strcsstate->sc_JoinState = EXEC_SC_NEXT_OUTER;
    break;

/* Get the next pruned tuple. It must have a higher post
 * value than the previous one (i.e. the current lower
 * partition boundary) and will become the upper
 * boundary of the current partition.
 *
 * We remain in this state until a matching tuple is found
 * and then proceed with a rescan of the document table to
 * retrieve the first tuple within the new partition. In the
 * special case that there is no more outer tuple, we are
 * within the last partition.
 */
 case EXEC_SC_NEXT_OUTER:

     /* Get the next tuple from the outer relation. */
     outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
     strcsstate->sc_OuterTupleSlot = outerTupleSlot;

     /* Last partition reached. Proceed directly with a
      * rescan of the document table.
      */
     if (TupIsNull(outerTupleSlot))
     {
         strcsstate->sc_JoinState = EXEC_SC_RESCAN;
         break;
     }

      /* Otherwise, evaluate the pruning condition. */
      ResetExprContext(econtext);
```

```
    /* Set the tuples to be compared. */
    outerTupleSlot = strcsstate->sc_OuterTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;
    innerTupleSlot = strcsstate->sc_PreviousTupleSlot;
    econtext->ecxt_innertuple = innerTupleSlot;

    /* Extract pruning condition. */
    clause = makeList1((Expr *)nth(1, greaterSkipQual));
    qualResult = ExecQual(clause, econtext, false);

    /* If it is satisfied, the next partition is set.
     * Proceed with a rescan of the document table.
     */
    if (qualResult)
    {
        strcsstate->sc_JoinState = EXEC_SC_RESCAN;
        break;
    }

    /* Otherwise, remain in this state. */
    break;

/* Initiate the index rescan of the document table. It
 * uses the lower partition boundary as index search key
 * and guarantees that the retrieval of document nodes
 * starts directly at the first tuple within the new
 * partition.
 */
case EXEC_SC_RESCAN:

    ResetExprContext(econtext);

    /* Set the outer tuple that guides the index lookup. */
    outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;

    /* Initiate the rescan. */
    ExecReScan(innerPlan, econtext, (Plan *) node);

    strcsstate->sc_JoinState = EXEC_SC_NEXT_INNER;
    break;

/* Get the next inner tuple. It is guaranteed to satisfy
 * the pre clause. If it is NULL, there are no more
 * document nodes. (We can be sure about this, because
 * we do not allow that further join clauses (apart from
 * the pre clause) are evaluated in the index.)
 *
 * If an inner tuple is returned, verify that its pre
```

```
 * value does not exceed the upper partition boundary.
 */
 case EXEC_SC_NEXT_INNER:

     /* Get the next inner tuple.*/
     innerTupleSlot = ExecProcNode(innerPlan, (Plan *) node);
     strcsstate->sc_InnerTupleSlot = innerTupleSlot;

     /* If there is none, there are no more document nodes. */
     if (TupIsNull(innerTupleSlot))
         return NULL;

     strcsstate->sc_JoinState =  EXEC_SC_TEST_PARTITION;
     break;

/* Verify that the current inner tuple's pre value does
 * not exceed the upper partition boundary. If it does,
 * switch to the next partition. If not, proceed with
 * the evaluation of the post clause.
 *
 * If we are within the last partition, we can directly
 * proceed with post clause evaluation.
 */
case EXEC_SC_TEST_PARTITION:

    outerTupleSlot = strcsstate->sc_OuterTupleSlot;

    /* Verify that the current inner tuple's pre value
     * does not exceed the upper partition boundary.
     */
    if (!TupIsNull(outerTupleSlot))
    {
        ResetExprContext(econtext);

        /* Set the tuples to be tested. */
        outerTupleSlot = strcsstate->sc_OuterTupleSlot;
        econtext->ecxt_outertuple = outerTupleSlot;
        innerTupleSlot = strcsstate->sc_InnerTupleSlot;
        econtext->ecxt_innertuple = innerTupleSlot;

        /* Extract partition condition. */
        clause = makeList1(nth(0, greaterSkipQual));
        qualResult = ExecQual(clause, econtext, false);

        /* If it is false, switch to the next partition. */
        if (!qualResult)
        {
            strcsstate->sc_JoinState = EXEC_SC_STORE;
            break;
        }
```

```
        }

        /* If the partition condition is satisfied or if we
         * are within the last partition, proceed with the
         * evaluation of the post clause.
         */
        strcsstate->sc_JoinState = EXEC_SC_TEST_POST;
        break;

    /* Evaluate the post clause. If it is true, prepare for
     * joining the two tuples. Otherwise, proceed with the next
     * partition (skipping).
     */
    case EXEC_SC_TEST_POST:

        ResetExprContext(econtext);

        /* Set the tuples to be tested. */
        outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
        econtext->ecxt_outertuple = outerTupleSlot;
        innerTupleSlot = strcsstate->sc_InnerTupleSlot;
        econtext->ecxt_innertuple = innerTupleSlot;

        /* 'joinclause' contains only the post clause. */
        qualResult = ExecQual(joinclause, econtext, false);

        /* In case of true, proceed with the join. */
        if (qualResult)
            strcsstate->sc_JoinState = EXEC_SC_JOIN;

        /* Otherwise, go to next partition (skipping). */
        else
            strcsstate->sc_JoinState = EXEC_SC_STORE;

        break;

    /* Join the current outer and inner tuple and prepare for
     * getting the next inner tuple. Return the join result.
     */
    case EXEC_SC_JOIN:

      strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;

          ⋮

    default:
        elog(WARNING, "ExecDescJoin: invalid join state
                      %d, aborting", strcsstate->sc_JoinState);
        return NULL;
}
```

```
    }
}
```

D.2.3 ExecAncJoin()

```
/*    ExecAncJoin
 *
 *    .../src/backend/optimizer/util/pathnode.c
 *
 * Yields the next tuple from an ancestor location step.
 */
static TupleTableSlot *ExecAncJoin(StrcsJoin *node)
{
    EState     *estate;
    StrcsJoinState *strcsstate;
    ScanDirection direction;
    List       *lowerSkipQual;
    List       *greaterSkipQual;
    List       *pre_clause;        /* the pre clause */
    List       *post_clause;       /* the post clause */
    List       *more_clauses;      /* other join clauses */
    List        *clause;
    bool        qualResult;
    Plan       *innerPlan;
    TupleTableSlot *innerTupleSlot;
    Plan       *outerPlan;
    TupleTableSlot *outerTupleSlot;
    ExprContext *econtext;

    /* Extract information from the staircase join plan node. */
    strcsstate = node->strcsstate;
    estate = node->join.plan.state;
    direction = estate->es_direction;
    innerPlan = innerPlan((Plan *) node);
    outerPlan = outerPlan((Plan *) node);
    econtext = strcsstate->jstate.cs_ExprContext;

    /* In case of the ancestor axis, we will always have two staircase
     * join clauses (special case).
     */
    Assert(length(node->join.joinqual) == 2);

    /* Extract the pre clause for this staircase join. */
    pre_clause = makeList1(lfirst(node->join.joinqual));
    /* Extract the post clause for this staircase join. */
    post_clause = makeList1(lsecond(node->join.joinqual));

                  ⋮

    /* Extract the preprocessed additional clauses. */
```

```
lowerSkipQual = strcsstate->sc_LowerSkipQual;
greaterSkipQual = strcsstate->sc_GreaterSkipQual;

/* Reset the expression context. */
ResetExprContext(econtext);

/* Loop until we have one joined tuple. */
for (;;)
{
    /* Get the current state of the join and do things accordingly. */
    switch (strcsstate->sc_JoinState)
    {
       /* Get first context tuple. If it is NULL, return NULL
        * (empty table). Otherwise, get ready for a rescan of the
        * document table.
        */
        case EXEC_SC_INITIALIZE:

            /* Get the first tuple from the outer relation. */
            outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
            strcsstate->sc_OuterTupleSlot = outerTupleSlot;

            /* If there is none, end the join. */
            if (TupIsNull(outerTupleSlot))
                return NULL;

            /* Initiate the first rescan of the document table. */
            strcsstate->sc_JoinState = EXEC_SC_RESCAN;
            break;

        /* Initiate the index rescan of the document table. It
         * uses the rescan tuple as index search key and guarantees
         * that the retrieval of document nodes starts directly at
         * the first tuple within the new partition.
         *
         * In the first partition the rescan tuple is a dummy tuple
         * initialized with 0 values.
         */
        case EXEC_SC_RESCAN:
            /* Set the outer tuple that guides the index lookup. */
            econtext->ecxt_innertuple = strcsstate->sc_RescanTupleSlot;

            /* Initiate the rescan. */
            ExecReScan(innerPlan, econtext, (Plan *) node);

            strcsstate->sc_JoinState = EXEC_SC_NEXT_INNER;
            break;

        /* Get the next inner tuple. If it is NULL, there are no
         * more document nodes.
```

```
 *
 * If an inner tuple is returned, evaluate the pre clause.
 */
case EXEC_SC_NEXT_INNER:

    /* Get the next tuple from the inner relation. */
    innerTupleSlot = ExecProcNode(innerPlan, (Plan *) node);
    strcsstate->sc_InnerTupleSlot = innerTupleSlot;

    /* If there is none, there are no more document nodes. */
    if (TupIsNull(innerTupleSlot))
        return NULL;

    strcsstate->sc_JoinState = EXEC_SC_TEST_PRE;
    break;

/* Evaluate the pre clause. If it is true, proceed with the
 * post clause. If it is false, switch partitions.
 */
case EXEC_SC_TEST_PRE:

    ResetExprContext(econtext);

    /* Set the tuples to be tested. */
    outerTupleSlot = strcsstate->sc_OuterTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;
    innerTupleSlot = strcsstate->sc_InnerTupleSlot;
    econtext->ecxt_innertuple = innerTupleSlot;

    /* 'pre_clause' contains the pre qualification. */
    qualResult = ExecQual(pre_clause, econtext, false);

    /* In case of true, evaluate the post clause. */
    if (qualResult)
        strcsstate->sc_JoinState = EXEC_SC_TEST_POST;

    /* In case of false, switch partitions. */
    else
        strcsstate->sc_JoinState = EXEC_SC_NEXT_OUTER;
    break;

/* In case of the ancestor axis, pruning is not carried out,
 * because it doesn't offer any advantages. So just get the
 * next outer tuple. If there is one, the current inner
 * tuple is guaranteed to be the first tuple within the new
 * partition, so just proceed with evaluating the pre clause.
 * Otherwise, return NULL and end the join.
 */
case EXEC_SC_NEXT_OUTER:
```

```
    /* Get the next tuple from the outer relation. */
    outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
    strcsstate->sc_OuterTupleSlot = outerTupleSlot;

    /* If there is none, end the join. */
        return NULL;

    strcsstate->sc_JoinState = EXEC_SC_TEST_PRE;
    break;

/* Evaluate the post clause. If it is true, prepare for
 * joining the two tuples. Otherwise, skipping is called for.
 * We use the current inner tuple's post value to skip to the
 * next inner tuple with a pre value equal to or larger than
 * the post value. Note that, additionally, we have to make
 * sure that we will indeed skip forwards, i.e. that the cur-
 * rent inner tuple's post value is larger than its pre value.
 */
case EXEC_SC_TEST_POST:

    ResetExprContext(econtext);

    /* Set the tuples to be tested. */
    outerTupleSlot = strcsstate->sc_OuterTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;
    innerTupleSlot = strcsstate->sc_InnerTupleSlot;
    econtext->ecxt_innertuple = innerTupleSlot;

    /* 'post_clause' contains the post qualification. */
    qualResult = ExecQual(post_clause, econtext, false);

    /* In case of true, proceed with the join. */
    if (qualResult)
        strcsstate->sc_JoinState = EXEC_SC_JOIN;

    /* Otherwise, skip to the next inner tuple. */
    else
    {
        /* Before triggering the skipping, make sure that the
         * current inner tuple's post value is greater than its
         * pre value. Otherwise, we would skip backwards.
         */
        clause = makeList1((Expr *)nth(2, lowerSkipQual));

        /* Is skipping directed forwards? */
        qualResult = ExecQual(clause, econtext, false);

        /* Skip. */
        if(qualResult)
        {
```

```
                        CacheRescanTuple(strcsstate->sc_InnerTupleSlot,
                                         strcsstate);
                        strcsstate->sc_JoinState = EXEC_SC_RESCAN;
                        break;
                    }
                    /* Or retrieve the next inner tuple sequentially. */
                    strcsstate->sc_JoinState = EXEC_SC_NEXT_INNER;
                }
                break;

            /* Join the current outer and inner tuple and prepare for
             * getting the next inner tuple. Return the join result.
             */
            case EXEC_SC_JOIN:

              strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;

                 ⋮

            default:
                elog(WARNING, "ExecAncJoin: invalid join state %d, aborting",
                   strcsstate->sc_JoinState);
              return NULL;
        }
    }
}
```

D.2.4 ExecFolJoin()

```
/*    ExecFolJoin
 *
 *    .../src/backend/optimizer/util/pathnode.c
 *
 * Yields the next tuple from a following location step.
 */
static TupleTableSlot *ExecFolJoin(StrcsJoin *node)
{
    EState     *estate;
    StrcsJoinState *strcsstate;
    ScanDirection direction;
    List       *lowerSkipQual;
    List       *greaterSkipQual;
    List       *joinclause;        /* the post clause */
    List       *more_clauses;      /* other join clauses */
    List       *clause;
    bool        qualResult;
    Plan       *innerPlan;
    TupleTableSlot *innerTupleSlot;
    Plan       *outerPlan;
    TupleTableSlot *outerTupleSlot;
```

```
ExprContext *econtext;

/* Extract information from the staircase join plan node. */
strcsstate = node->strcsstate;
estate = node->join.plan.state;
direction = estate->es_direction;
innerPlan = innerPlan((Plan *) node);
outerPlan = outerPlan((Plan *) node);
econtext = strcsstate->jstate.cs_ExprContext;

/* There is only one staircase join clause left, namely the post
 * clause. The pre clause has become an index clause.
 */
Assert(length(node->join.joinqual) == 1);
joinclause = node->join.joinqual;

              ⋮

/* Extract the preprocessed comparison clauses. */
lowerSkipQual = strcsstate->sc_LowerSkipQual;
greaterSkipQual = strcsstate->sc_GreaterSkipQual;

/* Reset the expression context. */
ResetExprContext(econtext);

/* Loop until we have the next joined tuple. */
for (;;)
{
    /* Get the current state of the join and do things accordingly. */
    switch (strcsstate->sc_JoinState)
    {
       /* Get first context tuple. If it is NULL, return NULL (empty
        * table). Otherwise, get ready for pruning the context set.
        */
        case EXEC_SC_INITIALIZE:

            /* Get the first tuple from the outer relation. */
            outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
            strcsstate->sc_OuterTupleSlot = outerTupleSlot;

            /* If there is none, end the join. */
            if (TupIsNull(outerTupleSlot))
                return NULL;

            /* Remember the first outer tuple. It is a potential
             * candidate for the single context node.
             */
            CachePreviousTuple(strcsstate->sc_OuterTupleSlot, strcsstate);

            /* Proceed with pruning the context set. */
```

```
        strcsstate->sc_JoinState = EXEC_SC_NEXT_OUTER;
       break;

   /* In case of the following axis, pruning means to retrieve
    * the tuple with the minimum post value from the context set.
    * So, retrieve a tuple from the outer subplan and compare it
    * with the lowest post value found so far. If it is lower,
    * remember it. As long as we are not at the end of the context
    * set, remain in this state. Otherwise, initiate a rescan of
    * the document table.
    */
    case EXEC_SC_NEXT_OUTER:

        /* Get the next tuple from the outer relation. */
        outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
        strcsstate->sc_OuterTupleSlot = outerTupleSlot;

        /* If there is none more, initiate the rescan. */
        if (TupIsNull(outerTupleSlot))
        {
            strcsstate->sc_JoinState =  EXEC_SC_RESCAN;
            break;
        }

        /* Else, evaluate the pruning condition. */
        ResetExprContext(econtext);

        /* Set the tuples to be tested. */
        outerTupleSlot = strcsstate->sc_OuterTupleSlot;
        econtext->ecxt_outertuple = outerTupleSlot;
        innerTupleSlot = strcsstate->sc_PreviousTupleSlot;
        econtext->ecxt_innertuple = innerTupleSlot;

        /* Extract the pruning condition. */
        clause = makeList1((Expr *)nth(1, lowerSkipQual));
        qualResult = ExecQual(clause, econtext, false);

        /* If it is true, store the current tuple and remain
         * in this state.
         */
        if (qualResult)
            CachePreviousTuple(strcsstate->sc_OuterTupleSlot,
                               strcsstate);
        break;

    /* Initiate the index rescan of the document table. It
     * uses the single pruned tuple as index search key
     * and guarantees that the retrieval of document nodes
     * starts directly at the first tuple within the new
     * partition.
```

```
 */
case EXEC_SC_RESCAN:
    /* Set the pruned tuple that guides the index lookup. */
    outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;

    /* Initiate the rescan. */
    ExecReScan(innerPlan, econtext, (Plan *) node);

    strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;
    break;

/* Get the next inner tuple. It is guaranteed to satisfy
 * the pre clause. If it is NULL, there are no more
 * document nodes. (We can be sure about this, because
 * we do not allow that further join clauses (apart from
 * the pre clause) are evaluated in the index.)
 *
 * If an inner tuple is returned, proceed with the
 * evaluation of the post clause.
 */
case EXEC_SC_NEXT_INNER:

    /* Get the next inner tuple.*/
    innerTupleSlot = ExecProcNode(innerPlan, (Plan *) node);
    strcsstate->sc_InnerTupleSlot = innerTupleSlot;

    /* If there is none, there are no more document nodes. */
    if (TupIsNull(innerTupleSlot))
        return NULL;

    strcsstate->sc_JoinState =  EXEC_SC_TEST_POST;
    break;

/* Evaluate the post clause. If it is true, prepare for
 * joining the two tuples. Otherwise, proceed with the next
 * inner tuple.
 */
case EXEC_SC_TEST_POST:

    ResetExprContext(econtext);

    /* Set the tuples to be tested. */
    outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
    econtext->ecxt_outertuple = outerTupleSlot;
    innerTupleSlot = strcsstate->sc_InnerTupleSlot;
    econtext->ecxt_innertuple = innerTupleSlot;

    /* 'joinclause' contains only the post clause. */
    qualResult = ExecQual(joinclause, econtext, false);
```

```
                /* In case of true, proceed with the join. */
                if (qualResult)
                    strcsstate->sc_JoinState = EXEC_SC_JOIN;
                else
                /* Otherwise, fetch the next inner tuple. */
                    strcsstate->sc_JoinState = EXEC_SC_NEXT_INNER;
                break;

            /* Join the current outer and inner tuple and prepare for
             * getting the next inner tuple. Return the join result.
             */
            case EXEC_SC_JOIN:

                strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;

                    ⋮

            default:
                elog(WARNING, "ExecFolJoin: invalid join state %d, aborting",
                   strcsstate->sc_JoinState);
                return NULL;
        }
    }
}
```

D.2.5 ExecPrecJoin()

```
/*    ExecPrecJoin
 *
 *    .../src/backend/optimizer/util/pathnode.c
 *
 * Yields the next tuple from a preceding location step.
 */
static TupleTableSlot *ExecPrecJoin(StrcsJoin *node)
{
    EState     *estate;
    StrcsJoinState *strcsstate;
    ScanDirection direction;
    List       *joinclause;        /* the post clause */
    List       *more_clauses;      /* other join clauses */
    bool        qualResult;
    Plan       *innerPlan;
    TupleTableSlot *innerTupleSlot;
    Plan       *outerPlan;
    TupleTableSlot *outerTupleSlot;
    ExprContext *econtext;

    /* Extract information from the staircase join plan node. */
    strcsstate = node->strcsstate;
```

```
estate = node->join.plan.state;
direction = estate->es_direction;
innerPlan = innerPlan((Plan *) node);
outerPlan = outerPlan((Plan *) node);
econtext = strcsstate->jstate.cs_ExprContext;

/* There is only one staircase join clause left, namely the post
 * clause. The pre clause has become an index clause.
 */
Assert(length(node->join.joinqual) == 1);
joinclause = node->join.joinqual;

                ⋮

/* Reset the expression context. */
ResetExprContext(econtext);

/* Loop until we have the next joined tuple. */
for (;;)
{
    /* Get the current state of the join and do things accordingly. */
    switch (strcsstate->sc_JoinState)
    {
       /* Get the first context tuple. If it is NULL, return NULL (empty
        * table). Otherwise, get ready for pruning the context set.
        */
       case EXEC_SC_INITIALIZE:

            /* Get the first tuple from the outer relation. */
            outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);
            strcsstate->sc_OuterTupleSlot = outerTupleSlot;

            /* If there is none, end the join. */
            if (TupIsNull(outerTupleSlot))
                return NULL;

            /* Proceed with pruning the context set. */
            strcsstate->sc_JoinState = EXEC_SC_NEXT_OUTER;
            break;

       /* In case of the preceding axis, pruning means to retrieve the
        * tuple with the maximum pre value (i.e. the last tuple) from
        * the context set. So, sequentially retrieve all tuples from
        * the outer subplan. As long as we are not at the end of the
        * context set, remain in this state. Otherwise, initiate a
        * rescan of the document table.
        */
        case EXEC_SC_NEXT_OUTER:

            /* Remember the current outer tuple. It is a potential
```

```
         * candidate for our single context node.
         */
        CachePreviousTuple(strcsstate->sc_OuterTupleSlot, strcsstate);

        /* Get the next tuple from the outer relation. */
        outerTupleSlot = ExecProcNode(outerPlan, (Plan *) node);

        /* If there is none more, initiate the rescan. */
        if (TupIsNull(outerTupleSlot))
            strcsstate->sc_JoinState =  EXEC_SC_RESCAN;

        /* Else remain in this state. */
        break;

    /* Initiate the index rescan of the document table. It
     * uses the single pruned tuple as index search key
     * and guarantees that the retrieval of document nodes
     * starts directly at the first tuple within the new
     * partition.
     */
    case EXEC_SC_RESCAN:
        /* Set the pruned tuple that guides the index lookup. */
        outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
        econtext->ecxt_outertuple = outerTupleSlot;

        /* Initiate the rescan. */
        ExecReScan(innerPlan, econtext, (Plan *) node);

        strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;
        break;

    /* Get the next inner tuple. It is guaranteed to satisfy
     * the pre clause. If it is NULL, there are no more
     * document nodes. (We can be sure about this, because
     * we do not allow that further join clauses (apart from
     * the pre clause) are evaluated in the index.)
     *
     * If an inner tuple is returned, proceed with the
     * evaluation of the post clause.
     */
    case EXEC_SC_NEXT_INNER:

        /* Get the next inner tuple.*/
        innerTupleSlot = ExecProcNode(innerPlan, (Plan *) node);
        strcsstate->sc_InnerTupleSlot = innerTupleSlot;

        /* If there is none, there are no more document nodes. */
        if (TupIsNull(innerTupleSlot))
            return NULL;
```

```
                strcsstate->sc_JoinState =  EXEC_SC_TEST_POST;
                break;

            /* Evaluate the post clause. If it is true, prepare for
             * joining the two tuples. Otherwise, proceed with the next
             * inner tuple.
             */
            case EXEC_SC_TEST_POST:

                ResetExprContext(econtext);

                /* Set the tuples to be tested. */
                outerTupleSlot = strcsstate->sc_PreviousTupleSlot;
                econtext->ecxt_outertuple = outerTupleSlot;
                innerTupleSlot = strcsstate->sc_InnerTupleSlot;
                econtext->ecxt_innertuple = innerTupleSlot;

                /* 'joinclause' contains only the post clause. */
                qualResult = ExecQual(joinclause, econtext, false);

                /* In case of true, proceed with the join. */
                if (qualResult)
                    strcsstate->sc_JoinState = EXEC_SC_JOIN;
                else
                /* Otherwise, fetch the next inner tuple. */
                    strcsstate->sc_JoinState = EXEC_SC_NEXT_INNER;
                break;

              /* Join the current outer and inner tuple and prepare for
               * getting the next inner tuple. Return the join result.
               */
              case EXEC_SC_JOIN:

                strcsstate->sc_JoinState =  EXEC_SC_NEXT_INNER;

                    .
                    .
                    .

              default:
                  elog(WARNING, "ExecPrecJoin: invalid join state %d, aborting",
                     strcsstate->sc_JoinState);
                 return NULL;
        }
    }
}
```

D.3 Completing Staircase Join Execution

```
/*    ExecEndStrcsJoin
 *
 *    .../src/backend/executor/nodeStrcsjoin.c
 *
 * Deallocate memory and end the execution of the subplans.
 */
void
ExecEndStrcsJoin(StrcsJoin *node)
{
    StrcsJoinState *strcsstate;

    /* Get state information from the node. */
    strcsstate = node->strcsstate;

    /* Free the projection info and the scan attribute info. */
    ExecFreeProjectionInfo(&strcsstate->jstate);
    ExecFreeExprContext(&strcsstate->jstate);

    /* Shut down the subplans. */
    ExecEndNode((Plan *) innerPlan((Plan *) node), (Plan *) node);
    ExecEndNode((Plan *) outerPlan((Plan *) node), (Plan *) node);

    /* Clean out the tuple table. */
    ExecClearTuple(strcsstate->jstate.cs_ResultTupleSlot);
    ExecClearTuple(strcsstate->sc_PreviousTupleSlot);
    ExecClearTuple(strcsstate->sc_RescanTupleSlot);
}
```