Handout: The Implementation of `TABLESAMPLE`

Neil Conway

May 21, 2007

Contents

1 Grammar Modifications

gram.y

```
/*
 * Pedagogical comment: the "relation_expr" production parses an
 * identifier name, optionally schema qualified, and including an
 * optional inheritance specification. It is used by the SELECT,
 * DELETE, and UPDATE productions, as well as several DDL commands.
 *
 * We want to allow the TABLESAMPLE clause to be specified for SELECT,
 * DELETE, and UPDATE, but not for DDL commands. Therefore, we add a
 * new production that is "relation_expr + optional TABLESAMPLE", and
 * use that anywhere we'd like to allow a TABLESAMPLE clause to be
 * specified.
 */
relation_expr_opt_sample:
            relation_expr  opt_table_sample
                {
                    $$ = $1;
                    $$->sample_info = (TableSampleInfo *) $2;
                }
        ;

relation_expr:
            qualified_name
                {
                    /* default inheritance */
                    $$ = $1;
                    $$->inhOpt = INH_DEFAULT;
                    $$->alias = NULL;
                }
/* The other variants are the same in principle; details ELIDED */
            | qualified_name '*'
            | ONLY qualified_name
            | ONLY '(' qualified_name ')'
        ;

opt_table_sample:
            TABLESAMPLE sample_method '(' Iconst ')' opt_repeatable_clause
            {
                TableSampleInfo *n = makeNode(TableSampleInfo);

                if (($2 == true)
                    n->sample_method = SAMPLE_BERNOULLI;
                else
```

```
                    n->sample_method = SAMPLE_SYSTEM;

                n->sample_percent = $4;
                if ($4 > 100)
                    ereport(ERROR,
                            (errcode(ERRCODE_INVALID_SAMPLE_SIZE),
                             errmsg("TABLESAMPLE percentage "
                                    "cannot exceed 100")));
                if ($4 <= 0)
                    ereport(ERROR,
                            (errcode(ERRCODE_INVALID_SAMPLE_SIZE),
                             errmsg("TABLESAMPLE percentage must "
                                    "be greater than 0" )));

                /* XXX: not supported yet */
                if (n->sample_method == SAMPLE_BERNOULLI)
                    ereport(ERROR,
                            (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                             errmsg("BERNOULLI sampling is not supported")));

                if ($6 != NULL)
                {
                    n->is_repeatable = true;
                    n->repeat_seed = intVal($6);
                }

                $$ = (Node *) n;
            }
            | /* EMPTY */                   { $$ = NULL; }
        ;

sample_method:
            BERNOULLI                       { $$ = true; }
            | SYSTEM_P                      { $$ = false; }
        ;

opt_repeatable_clause:
            REPEATABLE '(' Iconst ')'       { $$ = makeInteger($3); }
            | /* EMPTY */                   { $$ = NULL; }
        ;
```

2 Parse Nodes

primnodes.h

```
typedef enum TableSampleMethod
{
    SAMPLE_BERNOULLI,
    SAMPLE_SYSTEM
} TableSampleMethod;

typedef struct TableSampleInfo
{
    NodeTag             type;
    TableSampleMethod   sample_method;
    int                 sample_percent;
    bool                is_repeatable;
    int                 repeat_seed;
} TableSampleInfo;
```

2.1 Modifications to RangeVar

primnodes.h

```
/*
 * RangeVar - range variable, used in FROM clauses
 *
 * Also used to represent table names in utility statements; there,
 * the alias field is not used, and inhOpt shows whether to apply the
 * operation recursively to child tables. In some contexts it is also
 * useful to carry a TEMP table indication here.
 */
typedef struct RangeVar
{
    NodeTag     type;
    char        *catalogname;
    char        *schemaname;
    char        *relname;

    /* expand rel by inheritance? */
    InhOption   inhOpt;

    /* is this a temp relation? */
    bool        istemp;

    /* table alias and optional column aliases */
    Alias       *alias;

    /* TABLESAMPLE clause, if any */
    TableSampleInfo *sample_info;
} RangeVar;
```

3 Plan Node

plannodes.h

```
/*
```

```
 *        SampleScan node
 *
 * This is the information about a SampleScan that is fixed for a
 * given Plan. SampleScanState holds the run-time (executor-time)
 * state associated with a given ScanScan node.
 *
 * In addition to our parent class, we need only a single additional
 * piece of information: the information contained in the TABLESAMPLE
 * clause that corresponds to this SampleScan.
 */
typedef struct SampleScan
{
    Scan                scan;
    TableSampleInfo     *sample_info;
} SampleScan;
```

4 Planner Modifications

planner.c

```
/*
 *  set_plain_rel_pathlist
 *    Build access paths for a plain relation (no subquery, no inheritance)
 */
static void
set_plain_rel_pathlist (PlannerInfo *root, RelOptInfo *rel, RangeTblEntry *rte)
{
    /* Apply constraint exclusion: ELIDED */

    /* Mark rel with estimated output rows, width, etc */
    set_baserel_size_estimates (root, rel);

    /* Test any partial indexes of rel for applicability */
    check_partial_indexes(root, rel);

    /*
     * Check to see if we can extract any restriction conditions from join
     * quals that are OR-of-AND structures. If so, add them to the rel's
     * restriction list, and recompute the size estimates.
     */
    if (create_or_index_quals(root, rel))
        set_baserel_size_estimates (root, rel);

    /*
     * Generate paths and add them to the rel's pathlist.
     *
     * Note: add_path() will discard any paths that are dominated by another
     * available path, keeping only those paths that are superior along at
     * least one dimension of cost or sortedness.
     *
     * If there's a TABLESAMPLE clause, we ONLY consider using a
     * SampleScan. This could be improved: in some circumstances it
     * might make sense to do an IndexScan and then sample from the
     * index scan's result set, for instance.
     */
    if (rel->has_table_sample)
        add_path(rel, create_samplescan_path(root, rel));
    else
    {
        /* Consider sequential scan */
        add_path(rel, create_seqscan_path(root, rel));

        /* Consider index scans */
        create_index_paths(root, rel);

        /* Consider TID scans */
        create_tidscan_paths(root, rel);
    }

    /* Now find the cheapest of the paths for this rel */
    set_cheapest(rel);
}
```

4.1 Size Estimation

costsize.c

```
/*
 *  set_baserel_size_estimates
 *      Set the size estimates for the given base relation.
 *
 * The rel's targetlist and restrictinfo list must have been constructed
 * already.
 *
 * We set the following fields of the rel node:
 *  rows: the estimated number of output tuples (after applying
 *        restriction clauses and considering the effect of TABLESAMPLE).
 *  width: the estimated average output tuple width in bytes.
 *  baserestrictcost: estimated cost of evaluating baserestrictinfo clauses.
 */
void
set_baserel_size_estimates (PlannerInfo *root, RelOptInfo *rel)
{
    double              nrows;
    RangeTblEntry       *rte;
```

```
    /* Should only be applied to base relations */
    Assert(rel->relid > 0);

    nrows = rel->tuples *
        clauselist_selectivity (root,
                                rel->baserestrictinfo,
                                0,
                                JOIN_INNER);

    /*
     * Consider TABLESAMPLE, if any. We assume that the live heap rows
     * are uniformly distributed over the heap: this is a bogus
     * simplifying assumption. Note that the executor will apply the
     * TABLESAMPLE clause before applying any restrictions, we assume
     * that the restrictions have the same selectivity for the sampled
     * sub-relation as they do for the entire relation (which is
     * likely reasonable).
     */
    rte = planner_rt_fetch(rel->relid, root);
    if (rte->sample_info)
        nrows = nrows * rte->sample_info->sample_percent / 100;

    rel->rows = clamp_row_est(nrows);

    cost_qual_eval(&rel->baserestrictcost, rel->baserestrictinfo, root);

    set_rel_width(root, rel);
}
```

4.2 Cost Estimation

costsize.c

```
/*
 * cost_samplescan
 *     Determines and returns the cost of scanning a base relation with
 *     a TABLESAMPLE clause.
 */
void
cost_samplescan(Path *path, PlannerInfo *root, RelOptInfo *baserel)
{
    Cost        startup_cost = 0;
    Cost        run_cost     = 0;
    Cost        cpu_per_tuple;
    RangeTblEntry  *rte;
    int         sample_percent;

    /* Should only be applied to base relations */
    Assert(baserel->relid > 0);
    Assert(baserel->rtekind == RTE_RELATION);
    Assert(path->pathtype == T_SampleScan);

    rte = planner_rt_fetch(baserel->relid, root);
    sample_percent = rte->sample_info->sample_percent;

    /*
     * Disk costs. When the sample percentage is close to 100, we're
     * likely to be doing purely sequential I/O. Conversely, for small
     * percentage samples, we're doing random I/O. For now, just be
     * conservative and always assume that we need to do a random I/O
     * for each sampled block. Of course, this is quite bogus.
     */
    run_cost += random_page_cost * baserel->pages * sample_percent / 100;

    /* CPU costs */
    startup_cost += baserel->baserestrictcost.startup;
    cpu_per_tuple = cpu_tuple_cost + baserel->baserestrictcost.per_tuple;
    run_cost += cpu_per_tuple * baserel->tuples;

    path->startup_cost = startup_cost;
    path->total_cost = startup_cost + run_cost;
}
```

5 Executor Node

execnodes.h

```
/*
 * SampleScanState: the run-time state associated with a single
 * sample scan. This is the run-time dual of the SampleScan plan
 * node: for each SampleScan in the Plan tree, we create a
 * SampleScanState in the corresponding PlanState tree. A
 * PlanState's associated Plan can be found via ss.ps.plan.
 *
 * In addition to the fields of its parent class (ScanState), a
 * SampleScanState contains:
 *
 *      cur_buf: the current buffer/page being scanned, if any. The
 *               sample scan holds a pin on this buffer while it is
 *               executing, to ensure it isn't evicted from the
 *               buffer pool while we're using it. InvalidBuffer if
 *               we haven't started the scan yet, or the scan has
 *               finished (reached the end of the heap).
 *
 *      cur_offset: the current offset in the buffer being scanned.
 *
```

```
 *      cur_blkno: the BlockNumber of cur_buf -- that is, cur_buf's
 *                 position within the heap.
 *
 *      nblocks: the total # of blocks in the relation being scanned.
 *               Unless the sample percentage is 100, the scan
 *               likely won't visit this many blocks.
 *
 *      new_need_buf: have we run out of tuples on the current page?
 *
 *      cur_tup: current result tuple.
 */
typedef struct SampleScanState
{
    /* parent class; first field is NodeTag */
    ScanState       ss;
    Buffer          cur_buf;
    OffsetNumber    cur_offset;
    BlockNumber     cur_blkno;
    BlockNumber     nblocks;
    bool            need_new_buf;
    HeapTupleData   cur_tup;
} SampleScanState;
```

6 SampleScan Executor Implementation

6.1 Preamble

```
/*
 * nodeSamplescan.c
 *    Support routines for TABLESAMPLE-based scans of a relation
 *
 * Copyright (c) 2007, PostgreSQL Global Development Group
 *
 * IDENTIFICATION
 *    $PostgreSQL$
 */
#include "postgres.h"

#include <time.h>

#include "access/heapam.h"
#include "executor/executor.h"
#include "executor/nodeSamplescan.h"
#include "parser/parsetree.h"

static TupleTableSlot *SampleGetNext(SampleScanState *node);
static void           LoadNextSampleBuffer(SampleScanState *node);
static int            get_rand_in_range(int a, int b);
```

6.2 Initialization

```
/*
 * Initialize the run-time state of the sample scan on a single
 * relation. This requires setting up various executor machinery and
 * initializing the state of the PRNG.
 */
SampleScanState *
ExecInitSampleScan(SampleScan *node, EState *estate, int eflags)
{
    SampleScanState    *scanstate;
    Relation            rel;
    int                 seed;

    /* We don't expect to have any child plan nodes */
    Assert(outerPlan(node) == NULL);
    Assert(innerPlan(node) == NULL);

    scanstate                   = makeNode(SampleScanState);
    scanstate->ss.ps.plan       = (Plan *) node;
    scanstate->ss.ps.state      = estate;
    scanstate->cur_buf          = InvalidBuffer;
    scanstate->cur_offset       = FirstOffsetNumber;
    scanstate->cur_blkno        = InvalidBlockNumber;
    scanstate->need_new_buf = true;

    ExecAssignExprContext(estate, &scanstate->ss.ps);

    /*
     * Initialize the expression contexts required for evaluating the
     * target list and the scan's qualifiers, if any. We don't need to
     * do qual evaluation ourselves (ExecScan does it), but we do need
     * to do the required initialization.
     */
    scanstate->ss.ps.targetlist = (List *)
        ExecInitExpr((Expr *) node->scan.plan.targetlist,
                     (PlanState *) scanstate);
    scanstate->ss.ps.qual = (List *)
        ExecInitExpr((Expr *) node->scan.plan.qual,
                     (PlanState *) scanstate);

#define SAMPLESCAN_NSLOTS 2

    /*
     * Initialize the tuple table slots required by this scan: we need a
     * slot for the current result of the scan, and a slot for the
```

```
374         * current scan tuple.
375         */
376        ExecInitResultTupleSlot(estate, &scanstate->ss.ps);
377        ExecInitScanTupleSlot(estate, &scanstate->ss);
378
379        /*
380         * Open and lock the heap relation we're going to scan.
381         * ExecOpenScanRelation() will acquire the appropriate lock,
382         * depending on whether we're scanning this table with FOR UPDATE,
383         * FOR SHARE, or in normal mode.
384         */
385        rel = ExecOpenScanRelation(estate, node->scan.scanrelid);
386        scanstate->ss.ss_currentRelation = rel;
387
388        /*
389         * Determine the number of blocks in the relation. We need only do
390         * this once for a given scan: if any new blocks are added to the
391         * relation, they won't be visible to this transaction anyway.
392         */
393        scanstate->nblocks = RelationGetNumberOfBlocks(rel);
394
395        ExecAssignScanType(&scanstate->ss, RelationGetDescr(rel));
396
397        scanstate->ss.ps.ps_TupFromTlist = false;
398
399        /* Initialize result tuple type and projection info */
400        ExecAssignResultTypeFromTL(&scanstate->ss.ps);
401        ExecAssignScanProjectionInfo(&scanstate->ss);
402
403        /*
404         * Setup PRNG state; seed with the REPEATABLE clause, if any. We
405         * can't just use srandom(), since there could be multiple
406         * concurrent sample scans.
407         *
408         * XXX: using time() to seed the PRNG in the non-repeatable case
409         * could probably be improved. Different state array sizes could
410         * also be tried: do we need high-quality random numbers?
411         */
412        if (node->sample_info->is_repeatable)
413            seed = node->sample_info->repeat_seed;
414        else
415            seed = (int) time(NULL);
416
417    #define RAND_STATE_SIZE 128
418        scanstate->rand_state = palloc(RAND_STATE_SIZE);
419        initstate (seed, scanstate->rand_state, RAND_STATE_SIZE);
420
421        return scanstate;
422    }
```

6.3 Execution

```
422 /* Return the next tuple in the sample scan's result set. */
423 TupleTableSlot *
424 ExecSampleScan(SampleScanState *node)
425 {
426     /* Install our PRNG state */
427     setstate (node->rand_state);
428
429     /*
430      * ExecScan() provides generic infrastructure for "scan-like"
431      * executor nodes. It takes a ScanState describing the scan and
432      * a function pointer to an "access method". The access method
433      * is invoked repeatedly by ExecScan(); for each call, the
434      * access method should return the next tuple produced by the
435      * scan. ExecScan() then handles checking any relevant scan
436      * qualifiers, performing projection if necessary, and then
437      * stashing the result tuple in the appropriate TupleTableSlot.
438      */
439     return ExecScan((ScanState *) node,
440                     (ExecScanAccessMtd) SampleGetNext);
441 }
442
443 static TupleTableSlot *
444 SampleGetNext(SampleScanState *node)
445 {
446     EState          *estate;
447     TupleTableSlot  *slot;
448     Relation        rel;
449     Index           scanrelid;
450
451     estate    = node->ss.ps.state;
452     slot      = node->ss.ss_ScanTupleSlot;
453     rel       = node->ss.ss_currentRelation;
454     scanrelid = ((SampleScan *) node->ss.ps.plan)->scan.scanrelid;
455
456     while (true)
457     {
458         OffsetNumber   max_offset;
459         Page           page;
460
461         /*
462          * If we don't have a valid buffer, choose the next block to
463          * sample and load it into memory.
464          */
465         if (node->need_new_buf)
```

```
    {
        LoadNextSampleBuffer(node);
        node->need_new_buf = false;

        /* We're out of blocks in the rel, so we're done */
        if (!BufferIsValid(node->cur_buf))
            break;
    }

    /*
     * Iterate through the current block, checking for heap tuples
     * that are visible to our transaction. Return each such
     * candidate match: ExecScan() takes care of checking whether
     * the tuple satisfies the scan's quals.
     */
    LockBuffer(node->cur_buf, BUFFER_LOCK_SHARE);
    page = BufferGetPage(node->cur_buf);
    max_offset = PageGetMaxOffsetNumber(page);
    while (node->cur_offset <= max_offset)
    {
        /*
         * Postgres uses a somewhat unusual API for specifying the
         * location of the tuple we want to fetch. We've already
         * allocated space for a HeapTupleData; to indicate the TID
         * we want to fetch into the HeapTuple, we fillin its
         * "t_self" field, and then ask the heap access manager to
         * fetch the tuple's data for us.
         */
        ItemPointerSet(&node->cur_tup.t_self,
                       node->cur_blkno, node->cur_offset);

        node->cur_offset++;

        if (heap_release_fetch(rel, estate->es_snapshot,
                               &node->cur_tup, &node->cur_buf,
                               true, NULL))
        {
            LockBuffer(node->cur_buf, BUFFER_LOCK_UNLOCK);

            ExecStoreTuple(&node->cur_tup,
                           slot,
                           node->cur_buf,
                           false);

            return slot;
        }
    }
```

```
        /* Out of tuples on this page, so go on to the next one */
        LockBuffer(node->cur_buf, BUFFER_LOCK_UNLOCK);
        node->need_new_buf = true;
    }

    /* No more blocks to scan, so we're done: clear result slot */
    ExecClearTuple(slot);
    return NULL;
}
```

6.4 Support Routines

```
/*
 * Choose the next block from the relation to sample. This is called
 * when (a) we haven't sampled any blocks from the relation yet
 * (SampleScanState.cur_buf == InvalidBuffer) (b) we've examined every
 * tuple in the block we're currently sampling.
 *
 * If we've run out of blocks in the relation, we leave "cur_buf" as
 * InvalidBuffer.
 */
static void
LoadNextSampleBuffer(SampleScanState *node)
{
    SampleScan *plan_node = (SampleScan *) node->ss.ps.plan;

    while (true)
    {
        int rand_percent;

        /*
         * If this is the first time through, start at the beginning of
         * the heap.
         */
        if (BlockNumberIsValid(node->cur_blkno))
            node->cur_blkno++;
        else
            node->cur_blkno = 0;

        rand_percent = get_rand_in_range(0, 100);

        if (rand_percent >= plan_node->sample_info->sample_percent)
            continue;

        /*
         * If we've reached the end of the heap, we're done. Make sure
         * to unpin the current buffer, if any.
         */
```

```
        if (node->cur_blkno >= node->nblocks)
        {
            if (BufferIsValid(node->cur_buf))
            {
                ReleaseBuffer(node->cur_buf);
                node->cur_buf = InvalidBuffer;
            }

            break;
        }

        /*
         * Okay, we've chosen another block to read: ask the bufmgr to
         * load it into the buffer pool for us, pin it, and release the
         * pin we hold on the previous "cur_buf". For the case that
         * "cur_buf" == InvalidBuffer, ReleaseAndReadBuffer() is
         * equivalent to ReadBuffer().
         */
        node->cur_buf = ReleaseAndReadBuffer(node->cur_buf,
                                             node->ss.ss_currentRelation,
                                             node->cur_blkno);
        node->cur_offset = FirstOffsetNumber;
        break;
    }
}

/* Returns a randomly-generated integer x, such that a <= x < b. */
static int
get_rand_in_range(int a, int b)
{
    /*
     * XXX: Using modulus takes the low-order bits of the random
     * number; since the high-order bits may contain more entropy with
     * some PRNGs, we should probably use those instead.
     */
    return (random() % b) + a;
}
```

```
/*
 * Count the number of tuple table slots required by an instance of
 * the SampleScan.
 */
int
ExecCountSlotsSampleScan(SampleScan *node)
{
    return SAMPLESCAN_NSLOTS;
}
```

6.5 Shutdown

```
/*
 * Shutdown this scan. This function should generally be symmetric with
 * ExecInitSampleScan(): we ought to clean up after ourselves.
 */
void
ExecEndSampleScan(SampleScanState *node)
{
    ExecFreeExprContext(&node->ss.ps);

    ExecClearTuple(node->ss.ps.ps_ResultTupleSlot);
    ExecClearTuple(node->ss.ss_ScanTupleSlot);

    if (BufferIsValid(node->cur_buf))
    {
        ReleaseBuffer(node->cur_buf);
        node->cur_buf = InvalidBuffer;
    }

    /*
     * Note that ExecCloseScanRelation() does NOT release the lock we
     * acquired on the scan relation: it is held until the end of the
     * transaction.
     */
    ExecCloseScanRelation(node->ss.ss_currentRelation);
}
```