1 Grammar Modifications

/* 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_repeateable_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_repeateable = true;
    n->repeat_seed = intVal($6);
  }

  $$ = (Node *) n;
  
  } /* EMPTY */
  

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

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

2 Parse Nodes

2.1 Modifications to RangeVar

prinmnodes.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;

3 Plan Node

planmnodes.h

typedef struct RangeVar
{
  NodeTag type;
  char *catalogname;
  char *schemaname;
  char *relname;
  InhOption inhOpt;
  bool istemp;
  Alias *alias;
  TableSampleInfo *sample_info;
} RangeVar;
4 Planner Modifications

4.1 Size Estimation
4.2 Cost Estimation

4.2.1 Cost Estimation

4.2.1.1 Cost Estimation

4.2.1.2 Cost Estimation

4.2.1.3 Cost Estimation

4.2.1.4 Cost Estimation

5 Executor Node

execnodes.h
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;

    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);

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

    // 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

6.3 Execution

/* Return the next tuple in the sample scan's result set */
TupleTableSlot *
ExecSampleScan(SampleScanState *node)
{
    /* Install our PRNG state */
    setstate(node->rand_state);

    /* ExecScan() provides generic infrastructure for "scan-like"
    executor nodes. It takes a ScanState describing the scan and
    a function pointer to an "access method". The access method
    is invoked repeatedly by ExecScan(); for each call, the
    access method should return the next tuple produced by the
    scan. ExecScan() then handles checking any relevant scan
    qualifiers, performing projection if necessary, and then
    stashing the result tuple in the appropriate TupleTableSlot.
    */
    return ExecScan((ScanState *) node,
                    (ExecScanAccessMtd) SampleGetNext);
}

static TupleTableSlot *
SampleGetNext(SampleScanState *node)
{
    EState     *estate ;
    TupleTableSlot *slot ;
    Relation    rel ;
    Index       scanrelid ;

    estate = node->ss.ps.state;
    slot   = node->ss.ss.ScanTupleSlot;
    rel    = node->ss.ss.currentRelation;
    scanrelid = ((SampleScan *) node->ss.ps.plan) -> scan.relid;

    while (true)
    {
        OffsetNumber  max_offset;
        Page          page ;

        /* If we don't have a valid buffer, choose the next block to
         * sample and load it into memory.
        */
        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 fill in 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. */
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->nblocs)
{
    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);