[ODT+91] G. Ozsoyoglu, K. Du, A. Tjahjana, W. Hou, and D. Y. Rowland. On estimating COUNT, SUM, and AVERAGE relational algebra queries. In D. Dimitris Karagiannis, editor, *Database and Expert Systems Applications, Proceedings of the International Conference in Berlin, Germany, 1991 (DEXA 91)*, pages 406–412. Springer-Verlag, 1991.

[OR86] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proc. 12th Intl. Conf. Very Large Data Bases*, pages 160–169, 1986.

[Pos97] PostgreSQL Home Page, 1997. URL http://www.postgresql.org.

[RSS94] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *Trans. Knowledge and Data Engrg.*, 6(4):501–517, 1994.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First Intl. Conf. Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, Miami Beach, December 1991.

A. Full Iterator Pseudo-Code

```
int curstep; // global to query plan: current sampling step

class RIPL {
  iterator left, right; // operands
  int lcount, rcount;   // our view of the operands´ positions
  int lbeta, rbeta;     // aspect ratio of left and right
  int lmax, rmax;       // maximum position seen so far
  int llimit, rlimit;   // maximum position for this sampling step
  bool leof, reof;      // operand reported end-of-file
  iterator currel;      // relation currently acting as ´inner´
  iterator otherrel;    // the other relation
  bool innerlooping;    // are we in midst of inner loop?
  relation starter;     // which relation (if any) is the starter relation


  // maintain max position so far, to distinguish ``old´´ and ``new´´ tuples
  incr_pos(Relation rel) {
  if (rel == left) {
          lcount++;
          if (lcount > lmax) lmax = lcount;
      } else {
          rcount++;
          if (rcount > rmax) rmax = rcount;
  }  }
  pos(Relation rel)    { if (rel == left) return lpos;  else return rpos;  }
  repos(Relation rel)  { if (rel == left) lpos = 0; else rpos = 0;  }
  max(Relation rel)    { if (rel == left) return lmax;  else return rmax; }
  eof(Relation rel)    { if (rel == left) return leof; else return reof; }
```

```
seteof(Relation rel) { if (rel == left) leof = TRUE; else reof = TRUE;  }
init() {
    lcount = 0; rcount = 0; // cursor positions in left and right
    lmax = rmax = 0;
    llimit = rlimit = 1; // set up to fetch 1,1
    leof = reof = false;
    curstep = 1;
    currel = left; otherrel = right;
    innerlooping = true;
    if (this node is the root of the tree) starter = right;
    else starter = NULL;
}
next() {
  do { // loop until return() is called
      if (innerlooping) { // scanning one side of a rectangle
        while (pos(currel) < max(currel)) {
          currel.next();  incr_pos(currel);
          if (!eof(currel)) {
             if (predicate(R[R.pos],S[S.pos]))
                 return(R[R.pos], S[S.pos])
          } }
        innerlooping = false; // no more matches for otherrel.pos
      }
      else { // done with one side of a rectangle, pos(currel)==max(currel)
          if (pos(currel) < limit(currel)
                   && !eof(currel))
              swap(currel,otherrel); // at corner, start a new layer
          else if (eof(currel) && !eof(otherrel) && starter==currel) {
              // time to hand off starter role
              if (otherrel is a join) {
                  starter = NULL;
                  otherrel.starter = otherrel.inner;
              } else starter = otherrel;
          }
          else if ((pos(currel) == limit(currel) || eof(currel))
                   && (pos(otherrel) == limit(otherrel) || eof(otherrel))
                   {
            // we are at end of step
            if (starter == currel) {
              updateStep();
              swap(currel,otherrel);
            }
            else return(NULL); // must be at eof for both rels
          }
          if (!otherrel.next()) {
             innerlooping = false;  // either pad side again, or step's done
             if (pos(otherrel) < limit(otherrel))
               seteof(otherrel);  // must have hit end-of-file
             swap(currel,otherrel); // always finish looping on currel
          }
          else {
```

```
                repos(currel) = 0;        // now proceed with a new layer
                incr_pos(otherrel);
                innerlooping = true;
            }
  } } }
  updateStep() {
    if (left is a join) {
       left.updateStep();
       llimit = left.llimit * left.rlimit;
    }
    else llimit += lbeta;
    rlimit += rbeta;
  }

}
```

B. An Updating Algorithm for $\tilde{T}_{n,1,1}(f,g)$

The algorithm in Figure 14 can be used to update the quantity $\tilde{T}_{n,1,1}(f,g)$ after a sampling step of a block ripple join. This quantity (with $f = uv$ and $g = u$ as in Section 4.2) is used in the computation of running confidence intervals for aggregation queries of the form (4.1) with *op* equal to `AVG`. Initialize the computation by setting $x_1^{(f)} = \cdots = x_K^{(f)} = 0$, $x_1^{(g)} = \cdots = x_K^{(g)} = 0$, $w_1 = \cdots = w_K = 0$, and $n = 0$. After each sampling step of the block ripple join, execute the algorithm given in Figure 14 with $\Delta n = 1$ and with $l_k = \alpha\beta_k(n-1)$ and $\delta_k = \alpha\beta_k$ for $1 \le k \le K$. As with the algorithm in Figure 10, l_k $(1 \le k \le K)$ denotes the number of tuples read from R_k prior to the current sampling step and δ_k denotes the number of tuples read from R_k during the current sampling step. The quantity J denotes the set of index vectors $(i_1, i_2, \ldots, i_K)$ such that

(i) $l_k < i_k \le l_k + \delta_k$ for some $1 \le k \le K$; and

(ii) $f(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}) \neq 0$ or $g(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}) \neq 0$.

Just after the algorithm has executed for the nth time, the variable z is equal to $\tilde{T}_{n,1,1}(f,g)$, the variable $y^{(f)}$ is equal to $\tilde{T}_n(f)$, the variable $y^{(g)}$ is equal to $\tilde{T}_n(g)$, each variable $w_k/(L'_k - 1)$ is equal to $Z_n(f,g;k)$, each variable $q_{k,j}^{(f)}/L'_k$ is equal to $\tilde{T}_n(f;k,j)$, and each variable $q_{k,j}^{(g)}/L'_k$ is equal to $\tilde{T}_n(g;k,j)$.

C. The I/O Cost of Block Ripple Joins

In this section we consider the number of I/O's required to execute n (≥ 1) sampling steps of a K-relation ripple join with blocking factor α and aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$. (More specifically, we consider a K-table query plan which consists of a left-deep tree of binary ripple joins, which in combination sweep out a sequence of K-dimensional hyper-rectangles with the appropriate aspect ratios.) As discussed in Section 5, we make