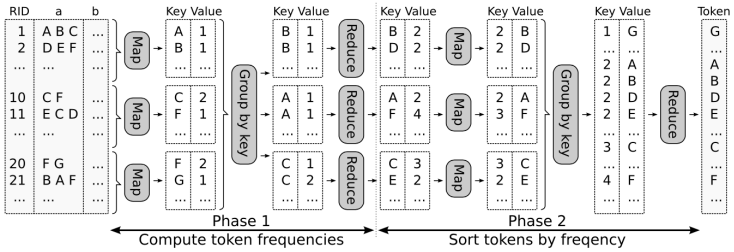
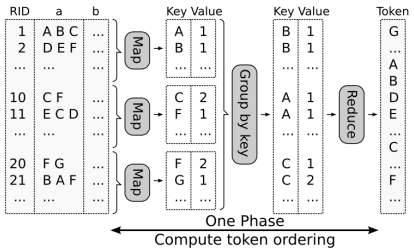
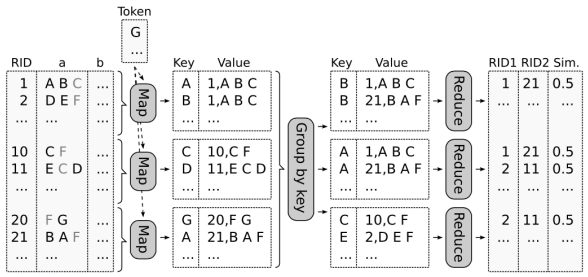


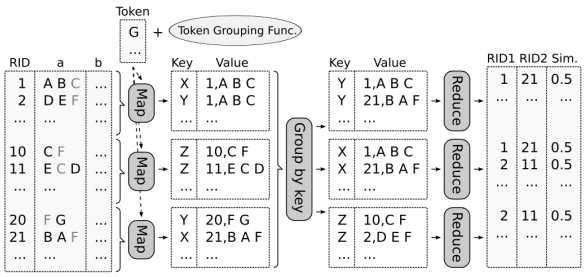
(a) Basic Token Ordering (BTO)



(b) One-Phase Token Ordering (OPTO)



(a) Basic Kernel (BK) using individual tokens for routing



(b) PPJoin+ Kernel (PK) using grouped tokens for routing

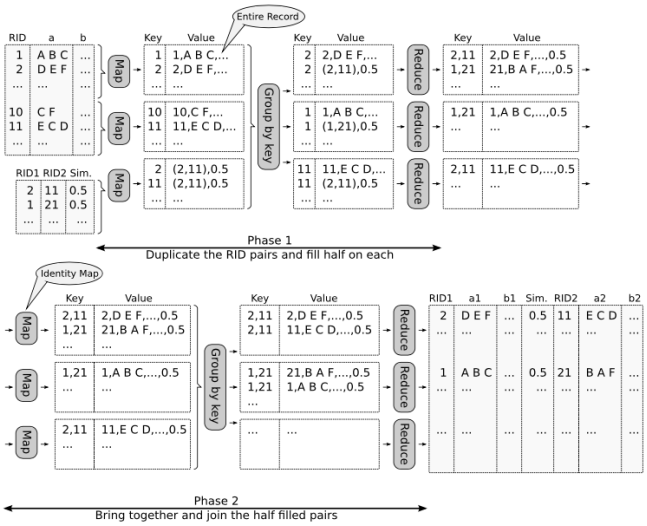


Figure 4: Example data flow of Stage 3 using Basic Record Join (BRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a", while "b1" and "b2" correspond to attribute "b".

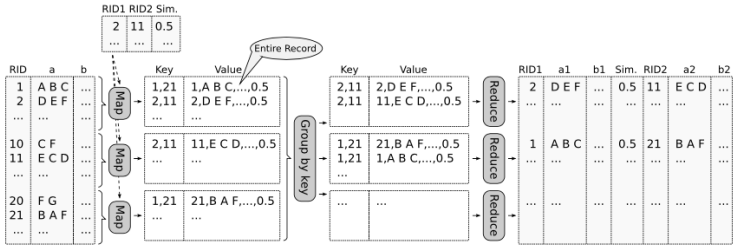


Figure 5: Example data flow of Stage 3 using One-Phase Record Join (OPRJ) for a self-join case. "a1" and "a2" correspond to the original attribute "a" while, "b1" and "b2" correspond to attribute "b".

the first stage alternatives, Basic Token Ordering (BTO) and One Phase Token Ordering (OPTO), respectively. The pseudo-code for the second stage alternatives is shown in Algorithm 3 and Algorithm 4, Basic Kernel (BK) and PPJoin+ Kernel (PK). Algorithm 5 and Algorithm 6 show the pseudo-code for the third stage, Basic Record Join (BRJ) and One-Phase Record Join (OPRJ).

Note: As an optimization for Algorithm 5, in the `reduce` of the first phase, we avoid looping twice over `list(v2)` in the following way: We use a composite key that includes a tag that indicates whether a line is a record line or a RID pair line. We then set the partitioning function so that the partitioning is done on RIDs and set the comparison function to sort the record line tags as low so that the line that contains the record will be the first element in the list.

Algorithm 1: Basic Token Ordering (BTO)

```
   // -- - Phase 1 - --
 1 map (k1=unused, v1=record)
 2 |   extract join attribute from record;
 3 |   foreach token in join attribute do
 4 |   |   output (k2=token, v2=1);
 5 combine (k2=token, list(v2)=list(1))
 6 |   partial_count ← sum of 1s;
 7 |   output (k2=token, v2=partial_count);
 8 reduce (k2=token, list(v2)=list(partial_count))
 9 |   total_count ← sum of partial_counts;
10 |   output (k3=token, v3=total_count);
   // -- - Phase 2 - --
11 map (k1=token, v1=total_count)
   |   // swap token with total_count
12 |   output (k2=total_count, v3=token);
   /* only one reduce task; with the total count as
      the key and only one reduce task, tokens will
      end up being totally sorted by token count */
13 reduce (k2=total_count, list(v2)=list(token))
14 |   foreach token do output (k3=token, v3=null);
```

Algorithm 2: One-Phase Token Ordering (OPTO)

```
   /* same map and combine functions as in Basic
      Token Ordering, Phase 1                    */
   /* only one reduce task; the reduce function
      aggregates the counts, while the reduce_close
      function sorts the tokens                  */
 1 reduce_configure
 2 |   token_counts ← [];
 3 reduce (k2=token, list(v2)=list(partial_count))
 4 |   total_count ← sum of partial_counts;
 5 |   append (token, total_count) to token_counts;
 6 reduce_close
 7 |   sort token_counts by total_count;
 8 |   foreach token in token_counts do
 9 |   |   output (k3=token, v3=null);
```

Algorithm 3: Basic Kernel (BK)

```
 1 map_configure
 2 |   load global token ordering, T;
 3 map (k1=unused, v1=record)
 4 |   RID ← record ID from record;
 5 |   A ← join attribute from record;
 6 |   reorder tokens in A based on T;
 7 |   compute prefix length, L, based on length(A) and
   |   similarity function and threshold;
 8 |   P ← tokens in prefix of length L from A;
 9 |   foreach token in P do
10 |   |   output (k2=token, v2=(RID, A));
11 reduce (k2=token, list(v2)=list(RID, A))
12 |   foreach (RID1, A1) in list(v2) do
13 |   |   foreach (RID2, A2) in list(v2) s.t. RID2 ≠ RID1
   |   |   do
14 |   |   |   if pass_filters(A1, A2) then
15 |   |   |   |   Sim ← similarity(A1, A2);
16 |   |   |   |   if Sim ≥ similarity threshold then
17 |   |   |   |   |   output (k3=(RID1, RID2, Sim),
   |   |   |   |   |   v3=null);
```

B. EXPERIMENTAL RESULTS

In this section we include additional figures for the experiments performed in Section 6 which we were not able to include in Section 6 due to lack of space. More specifically, we include figures for Table 1, Table 2, and data communication.

B.1 Self-Join Performance

B.1.1 Self-Join Speedup

Figure 15(a) shows the running time for the first stage

Algorithm 4: PPJoin+ Kernel (PK)

```
1  map_configure
2      load global token ordering, T;
3  map (k1=unused, v1=record)
       /* same computations for RID, A, and P as in
          lines 4-8 from Basic Kernel                  */
4      G ← [] // set of unique group IDs
5      foreach token in P do
6          groupID ← choose_group(token);
7          if groupID ∉ G then
8              output (k2=groupID, v2=(RID, A));
9              insert groupID to G;
10 reduce (k2=groupID, list(v2)=list(RID, A))
11     PPJoin_init(); // initialize PPJoin index
12     foreach (RID1, A1) in list(v2) do
13         R ← PPJoin_probe(A1);
           // returns a list of (RID, Sim) pairs
14         foreach (RID2, Sim) in R do
15             output (k3=(RID1, RID2, Sim), v3=null);
16         PPJoin_insert(RID1, A1)
```

Algorithm 5: Basic Record Join (BRJ)

```
   // -- - Phase 1 - --
1  map (k1=unused, v1=line)
2      if line is record then
3          RID ← extract record ID from record line;
4          output (k2=RID, v2=line);
5      else
           // line is a (RID1, RID2, Sim) tuple
6          output (k2=RID1, v2=line);
7          output (k2=RID2, v2=line);
8  reduce (k2=RID, list(v2)=list(line))
9      foreach line in list(v2) do
10         if line is record then
11             R ← line;
12             break;
13     foreach line in list(v2) do
14         if line is a (RID1, RID2, Sim) tuple then
15             output (k3=(RID1, RID2), v3=(R, Sim));
   // -- - Phase 2 - --
   /* identity map                                    */
16 reduce (k2=(RID1, RID2), list(v2)=list(R, Sim))
17     R1 ← extract R from first in list(v2);
18     Sim ← extract Sim from first in list(v2);
19     R2 ← extract R from second in list(v2);
20     output (k3=(R1, Sim, R2), v3=null);
```

Algorithm 6: One-Phase Record Join (OPRJ)

```
1  map_configure
2      load RID pairs and build a hash table, P, with the
       format RID1 → {(RID2, Sim), ...};
3  map (k1=unused, v1=record)
4      extract RID1 from record;
5      J ← probe P for RID1;
6      foreach (RID2, Sim) in J do
7          output (k3=(RID1, RID2), v3=(record, Sim));
   /* same reduce function as in Basic Data Join,
      Phase 2                                         */
```

Figure 16 shows the data-normalized communication in each stage. Each point shows the ratio between the total size of the data sent between `map` and `reduce` and the size of the original input data. Please note that only a fraction of that data is transferred through the network to other nodes while the rest is processed locally.

B.1.2 Self-Join Scaleup

Figure 17 shows the running time for self-joining the DBLP dataset, increased from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively on a "relative" scale. For each dataset size we plotted the ratio between the running time for that dataset size and the running time for the minimum dataset size. Figure 18 shows the data-normalized communication in each stage. Figure 19(a) shows the running time of the first stage (token ordering). The running time for the second stage (kernel) is plotted in Figure 19(b). Figure 19(c) shows the running time for the third stage (record join).

B.2 R-S Join Performance

B.2.1 R-S Join Speedup

Figure 20 shows the relative running time for joining the DBLP×10 and the CITESEERX×10 datasets on clusters of 2 to 10 nodes. We used the same three combinations for the three stages. We also show the ideal speedup curve (with a thin black line). Figure 21(a) shows the running time for the second stage of the join (kernel). In Figure 21(b) we plot the running time for the third stage (record join) of the join.

B.2.2 R-S Join Scaleup

Figure 22 shows the running time for joining the DBLP and the CITESEERX datasets, increased from 5 to 25 times, on a cluster with 2 to 10 nodes, respectively on a "relative" scale. Figure 23(a) shows the running time of the second stage (kernel). The running time for the third stage (record join) is plotted in Figure 23(b).