has two important features: (1) it is faster than *GSJoin* for very small datasets because it does not require any spatial partitioning methods and duplicate removal. (2) it may lead to system failure or very long execution time for large datasets because it shuffles an entire SRDD A to each partition of SRDD B.

Distance join algorithm The distance join algorithms can be seen as extensions to the range join algorithms, *GSJoin* and Broadcast join. The only difference is that we add a distance buffer to all objects in either Spatial RDD A or B at the very beginning (even before spatial partitioning) to extend their boundaries. The extended spatial objects can be used in both range join algorithms.

Algorithm 4 *GSJoin* algorithm for range join and distance join query

```
   Data: (repartitioned) SRDD A and (repartitioned) SRDD B
   Result: PairRDD in schema <Left object from A, right object from B>
   /* Step1: Zip partitions                                              */
1  foreach partition pair from SRDD A and B with the same grid ID i do
2  |  Merge two partitions to a bigger partition that has two sub-partitions;
3  Return the intermediate SRDD C;
   /* Step2: Run partition-level local join                              */
4  foreach partition P in the C do
5  |  foreach object O_A in the sub-partition from A do
6  |  |  if an index exists in the sub-partition from B then
   |  |  |  // Filter phase
7  |  |  |  Query the spatial index of this partition using the O_A's MBR;
   |  |  |  // Refine phase
8  |  |  |  Check the spatial relation using real shapes of O_A and candidate objects O_Bs;
   |  |  |  /* Step3: Remove duplicates                                  */
9  |  |  |  Report <O_A, O_B> pair only if the reference point of this pair is in P;
10 |  |  else
11 |  |  |  foreach object O_B in the sub-partition from B do
12 |  |  |  |  Check spatial relation between O_A and O_B;
   |  |  |  |  /* Step3: Remove duplicates                               */
13 |  |  |  |  Report <O_A, O_B> pair only if the reference point of this pair is in P;
14 Generate the result PairRDD;
```

Spark DAG The DAG and data flow of *GSJoin* are shown in Fig. 4. The join query introduces two transformations: zip partitions and partition-level local join. Both of them incur narrow dependencies. Therefore, they are pipelined to one stage with fast execution. On the contrary, the broadcast join algorithm (see the right part in Fig. 4) introduces two wide dependencies which lead to heavy network traffic and intensive computation. Until now, the effect of spatial partitioning is shown by degrading wide dependencies to narrow dependencies. For spatial analytics program that runs multiple join queries, the repartitioned A and B are cached into memory and recycled for each join query.

5.4 Spatial SQL query optimization

An important feature of GeoSpark is the Spatial SQL query optimizer. GeoSpark optimizer extends SparkSQL Catalyst optimizer to support Spatial SQL optimization. It takes as input the Spatial SQL statement written by the user and generates an efficient execution plan in terms of time cost. For advanced users who have the necessary Spark knowledge, they can use GeoSpark Scala /Java RDD APIs to achieve granular control over their applications (Fig. 6).