**Distance Match:**

1. **Pre-computing the distances of every sky object from all other objects in the dataset:**

For example, in an astronomy dataset with n = 3 million sky objects, (doing a sort-merge-join) needs to pre-compute the distances of every sky object from all other objects and store an ordered list of **m** - least distances results in **m**-nearest points to be able to answer the future **m**-points pattern queries. This operation needs O (m^2\*n^2) comparisons. Well, maybe we can do a little better. We could use an octree as a pre-filtering step and then look only a certain likely radius from each point. Furthermore, we need to store this data structure in a file; Using this data structure, make it possible to answer every pattern query (**k** the number of points in query < **m**) with complexity of O(k\*m\*n). Here, is the result of this step:

|  |
| --- |
| **Astronomy dataset** |
| **star\_id** | **…** | **m-partners** **(in ascending distance order)** | **partners\_distances** **(ascending order)** |
| **…** | **…** |  |  |
| **s12** | **s15, s18, s19, s21, s33, s35, …, sm** | **d1, d2,d3,…,dm** |
| **s13** | **…** | **d1, d2,d3,…,dm** |
| **…** | **…** | **d1, d2,d3,…,dm** |
| **s46** | **s41, s42, …, sm** | **d1, d2,d3,…,dm** |
| **…** | **…** | **d1, d2,d3,…,dm** |
| **s67** | **s15, s65, …, sm** | **d1, d2,d3,…,dm** |
| **…** | **…** | **…** |

1. **Compute the distances between all points in the query:**

In a sample query with k = 3 points, we compute the distances between all points in the query and then sort them in ascending order as following:

d0

P0

P1

d2

d1

P2

**P0 (d0, d1) P0\_partners (P1, P2)**

**P1 (d0, d2) P1\_partners (P0, P2)**

**P2 (d1, d2) P2\_partners (P0, P1)**

|  |
| --- |
| **Query Distances** |
| point\_index | partners (in ascending distance order) | distances (ascending order) |
| 0 | **P1, P2** | **d0, d1** |
| 1 | **P0, P2** | **d0, d2** |
| 2 | **P0, P1** | **d1, d2** |

1. **Merge join:**

Now, performing marge join on tables **Astronomy dataset** and **Query Distances,** we find all points that look like every query point in terms of their distances. I may have been unclear on this point: a star point may have more distances than the query point, but every distance relative to the query point should be present (ignoring measurement errors for now) for the candidate star point. Here, we have table of matches:

|  |
| --- |
| **Candidates (result of merge-join)** |
| **star\_id** | **query\_matched\_point\_id** | **partners (in ascending distance order)** | **partners\_distances** **(ascending order)** |
| **s12** | **0** | **s15, s33** | **d\_1, d\_2** |
| **s12** | **0** | **s21, s18** | **d\_1, d\_2** |
| **s12** | **1** | **s21, s19** | **d\_1, d\_2** |
| **s35** | **1** | **s54, s70** | **d\_1, d\_2** |
| **s46** | **0** | **s41, s42** | **d\_1, d\_2** |
| **s67** | **0** | **s15, s65** | **d\_1, d\_2** |
| **…** | **…** | **…** | **…** |

1. Finally, we run the following query to find the solutions:

for each c in C (for each group of candidate stars)
  form a 1-1 correspondence between the stars in c and the k+1 query points based on the

 distances from point0, thus renaming the stars in c point0, point1, .... pointk
    consistently.

 For every pair pointu and pointv in point1, … pointk

 See whether dist(pointu, pointv) corresponds to dist(qu, qv) where qu is the uth point fartheset away from q0 and qv is the vth point farthest away from q0

 If false, then c is not a match
  ~~for each r in c after renaming
    if r.i = q and r.partners != pointkpartners
        then c is not a match~~  c is a match

for each value t in c
       let j = t.q
       jstar = select s' from pointj where s = t.s'