together, in order to obtain a cluster snapshot. For instance, in Fig. 3, we get several clusters for the snapshot at time 2, i.e., cluster 0: {0, 2, 3, 56}, cluster 1: {1, 3, 4}, and so forth.

Finally, when each cluster snapshot comes, ICPE utilizes Pattern Enumeration to obtain all co-movement patterns, to be covered in Section 6. For example, in Fig. 3, we get several patterns at time 2, i.e., {0, 2, 56}, {1, 3, 4}, etc.

During the first step, we also need to consider time synchronization for stream processing. Flink cannot ensure that trajectories are processed in time order. However, pattern detection requires that trajectories are processed in ascending time order. Hence, we add "last time" information to each trajectory in every snapshot, to be able to guarantee that trajectories are processed in time order. The "last time" denotes the time of the most recent snapshot for which the trajectory reported a location.

Using the "last time", we can determine whether the system needs to wait for the location of a particular time. As an example, for a stream trajectory $tr = \{r_1, r_2, r_3, r_5, ...\}$, where r_i denotes the GPS record at time i, the "last time" associated with r_3 is time 2, and the "last time" associated with r_5 is time 3. In case the system has only received r_1 and r_3, it must wait for r_2, because "last time" information of r_3 indicates that a location was reported by the trajectory for the snapshot at time 2 that has yet to be received. Next, in case the system has received r_1, r_2, r_3, and r_5, it does not need to wait for r_4, because the "last time" information of r_5 indicates that no position was reported for the snapshot at time 4.

5. INDEXED CLUSTERING

In this section, we first describe the GR-index, and then present the index based RangeJoin and DBSCAN methods.

5.1 GR-index

As stated in Section 4, clustering includes two steps, i.e., we first perform a range join on each snapshot S_t, and then, we use DBSCAN to cluster S_t. To accelerate the range join, a two-layered index, the **GR-index** [36], is built in Flink. GR-index is verified efficient for distributed platforms (e.g., Spark, Storm). It uses a grid index as a global index, and builds an R-tree [3] as a local index for each grid cell.

Fig. 4 illustrates a GR-index for a specific snapshot, in which Fig. 4(a) shows the global grid index and Fig. 4(b) shows the local R-trees. The GR-index has 16 grid cells, and an R-tree for grid cell g_6 is shown as an example. The R-tree leaf nodes N_1 and N_2 contain the real locations o_4, o_5, o_7, and o_8, and the minimum bounding rectangles of N_1 and N_2 are shown in g_6.

Key Computation. Each grid cell can be regarded as a partition in Flink. The key of the grid cell that a location $o = (x, y)$ belongs to can be computed as $\langle \lfloor o.x/l_g \rfloor, \lfloor o.y/l_g \rfloor \rangle$, where l_g is the grid cell width. For example, as shown in Fig. 4, given the location $o_5 = (4, 8)$ and the grid cell width $l_g = 3$, the key of g_6 that o_5 belongs to is $\langle 1, 2 \rangle$.

Note that, the GR-index is a primary index. We compute a key for each location, and locations with the same key will be distributed to the same subtask when building local R-trees.

5.2 GR-index Based Range Join

We develop three algorithms to compute the range join based on the GR-tree, i.e., GridAllocate, GridQuery, and GridSync. Fig. 5 depicts the processing of the ICPE-RangeJoin. GridAllocate builds the global grid index (i.e., computes the key for each location) to partition each snapshot into disjoint grid cells, and it transforms locations into data objects (i.e., locations contained in a specific grid cell) and query objects (i.e., locations whose range region intersects

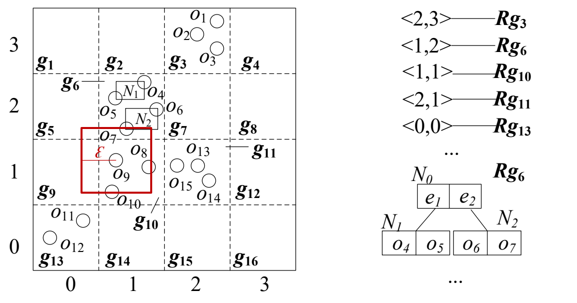

Figure 4: Example of a GR-index

with this grid cell). For each grid cell (key), a GridQuery method builds the local R-tree for the data objects, and performs a range query for each query object. Finally, GridSync collects the results from all grid cells. Here, a GR-index is built for each snapshot, and is deleted after querying. Thus, index maintenance is omitted.

According to the functioning of Flink, after building the grid index, the partitions are independent of each other, i.e., locations in other partitions cannot be accessed. However, during range-join processing, we have to access other partitions. As shown in Fig. 4, although o_9 is not located in grid cell g_6, the range query result *RQ*(o_9, ϵ) includes location o_7 that is located in grid cell g_6. To address this, we replicate each location into multiple GridObjects, and distribute them to the relevant grid cells.

DEFINITION 12. **(GridObject)**. *A GridObject go* $=$ (*key, flag, location*) *is a triple, where location is the actual position of go, flag indicates the type of go, and key records the grid cell that go belongs to.*

Specifically, if *flag* is false, then *go* is a data object, meaning that its *location* needs to be inserted into the corresponding R-tree of this grid cell; otherwise, if *key* is true, then *go* is a query object, indicating that the grid cell with *key* might contain the range query result for *go*.

Based on the definition of GridObject, a location can be represented as a data object (*key*(g), false, *location*), where g is the original grid cell that *location* belongs to; and several query objects (*key*(g_i), true, *location*), in which grid cells g_i intersect with the range region of *RQ*(*location*, ϵ).

For example, in Fig. 4, o_9 can be represented as a data object go_1 ($\langle 1, 1 \rangle$, false, o_9), indicating that o_9 needs to be inserted into the R-tree of grid cell g_{10}. In addition, according to the range region (i.e., the red square centered at o_9 with length 2ϵ), o_9 is represented by four query objects $go_2 = (\langle 0, 2 \rangle$, true, $o_9)$, $go_3 = (\langle 1, 2 \rangle$, true, $o_9)$, $go_4 = (\langle 0, 1 \rangle$, true, $o_9)$, and $go_5 = (\langle 1, 1 \rangle$, true, $o_9)$, because the range query result for o_9 is contained in grid cells g_5, g_6, g_9, and g_{10}. To improve the efficiency of the range join, we develop two lemmas below to avoid unnecessary verifications.

According to the definition of a range query, we need to verify all the grid cells that intersect with the range region. Nevertheless, for a range join on a single dataset, verifying all the grid cells might yield duplicated results [4]. For instance, in Fig. 4, o_9 needs to be distributed to grid cell g_6, as the range region of o_9 intersects with g_6, and thus, o_9 is represented by the GridObject ($\langle 1, 2 \rangle$, true, o_9) to find the result pair (o_9, o_7). In addition, o_7 needs to be distributed to grid cell g_{10}, and thus, it is represented by the GridObject ($\langle 1, 1 \rangle$, true, o_7) to find the result pair (o_7, o_9). The outcome is that pair (o_7, o_9) is duplicated in the result. The idea of the first lemma is that, instead of verifying all the grid cells intersected with the range region ($[o.x - \epsilon, o.x + \epsilon]$, $[o.y - \epsilon, o.y + \epsilon]$), we only verify half of those grid cells, i.e., the grid cells that intersect with the upper part