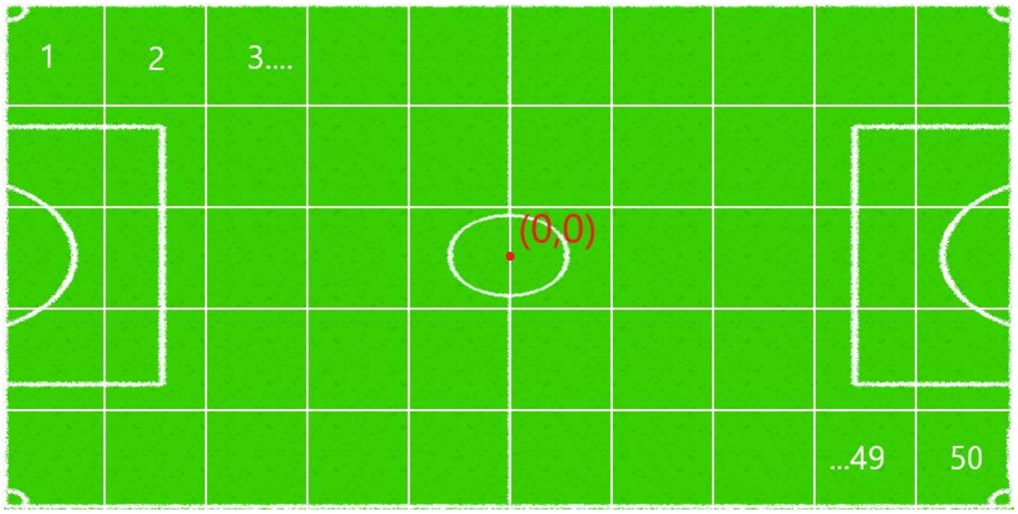
Assignment 2 - CT3532 Database Systems 2

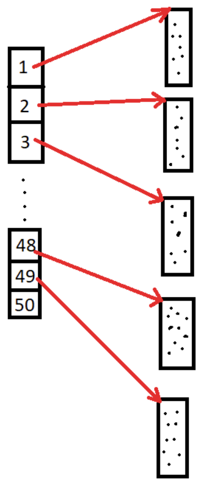
Cathal Lawlor - 21325456 | Liam Holland - 21386331

1. Indexing the locations of a single player

For this type of query, where you might want to build a heat map of a player's location across the course of a game, we believe the most optimal approach is to build a grid-based index to group blocks of pointers. The index itself will be based on dividing the pitch into a grid. Taking the centre of the pitch to be (0,0), the length of the pitch to be 600 units and the height of the pitch to be 300 units. This way we can divide the pitch into squares of 60 x 60 units. This will give us a 10 x 5 grid, 50 squares in total. We can then number the grid squares as below:



To construct the index, we will then sift through the database to find the tuples which match the player id we are constructing the index for; this will be fairly inefficient as for every timestamp, there is exactly one relevant tuple. When we find the relevant tuple, we can check the x and y position to check which grid square it falls under. For example, the player is in square 1 when their x is less than -240 and their y is less than -90. A pointer to the that tuple can then be stored in a block which the index value of 1 points to.



This will sort all the player's tuples into buckets of when they were in a particular grid square. A potential issue here is that the player could be in a position that sees them remain in a small portion of the pitch for the entire game; the goalkeeper, for example. This could result in one block of pointers being massive while all the others are practically or totally empty. To address this, we would make these buckets dynamically sized, where they able to house 10 pointers at the beginning, but as the buckets overflow, we can allocate more memory. If there is no space available sequentially, we can have a pointer to the beginning of the next block, essentially fragmenting the data on disk.

To find the amount of time the player spent in a particular square, all you must do is query the index for the bucket and count the number of pointers, then multiply that number by the interval between each timestamp. To construct a heatmap from that data, you could simply relatively increase the colour of a square based on the amount of time, or even just the number of pointers.

2. Parallelising the above approach

While the approach in question 1 will work, the construction of the index will take some time and not run especially efficiently. This is due to the fact that only 1/22 tuples are actually relevant to the query. Searching through thousands of tuples will take a long time and knowing that most of those tuples are not relevant is not ideal. This is why parallelisation is so useful here; it will vastly decrease the amount of time it takes to construct the index.

To parallelise the construction of the index, we believe the best approach is to divide the entries across N processors using range partitioning based on the timestamps in the tuples. This should ensure that the tuples are divided evenly across the processors, as we can simply use a modulo N operation on the timestamp to determine the processor it should be dispatched to.

We use range partitioning over a round robin approach as every 22nd entry will be the player we are looking for, where with a round robin approach, we might encounter the case where the first processor is doing all the indexing for the specified player e.g., it is doing every second entry in the file, with 22 entries per time stamp, dividing by 2 it will always encounter the player. This will mean a much more fairly divided workload, with a fairer execution time for each processor, with each one having the approximately the same amount of relevant data.

Likewise, we would not want to use a hash function here, as keeping the data temporally sorted is to our advantage. A hash function would also likely just involve hashing on the timestamp anyway, as it is really the only unique value. With no collisions, the result would still be the same as the range partition approach.

3. **Indexing the locations of when any given player is in a specified rectangle in a 3x3 grid pitch.**

The query in this question is similar to that in the first, only this time we need to be able to do the task for any given player and find the specific times when they were in the square. To do this, we believe the best approach is to create a hash map to store pointers to tuples. With only 9 squares, the ranges on positions included in each square will be much larger. The approach removes the need for a grid-based index by including the grid number as part of the hash function along with the player id and the time. Because each player can only be in one square at each time, this will provide unique parameters for each hash.

To construct the index, we will start the same, going through each entry on by one, only this time we are going to create a pointer to *every* tuple in the index. When we find the square the tuple was recorded from (1 – 9), we hash that number along with the player id and the timestamp to create a pointer, and write it into the hash map. This method will allow us to avoid collisions while also keeping data sorted in a meaningful way.

We can also implement dynamic hashing in this approach, which will make it far more space conscious. Dynamic hashing will allow the size of the hash map to grow as needed, wasting less space.

To build a heat map for a given player id across the field, you could run an algorithm like this:

```
For each gridSquare:
    time = 0
    while (time < maxMatchTime):
        if(hash(gridSquare, p_id, time) != null):
            numInstances++
        time += interval
gridSquare.colourIntensity *= numInstances
```

This algorithm goes through each grid square, hashing every timestamp with the player id that you are looking for in the grid square you are currently checking. If there is a hit in the hash map, the program will increment a count of instances that that player is in that grid square. When all of the pointers have been checked, the colour intensity can be modified to reflect how much time the player spent in that square.

4. **Indexing the locations of when any two players are both in the same grid square.**

Our approach to this is very similar to the one in question 3. The main difference is that we build the index to include pointers to tuples which we *know* represent a player that is in a grid square at a time that another player is also in that square. In other words, every pointer in the index needs to be relevant to the query, so we need to do some operation to check that a tuple in the database is in the same x and y bounds as another tuple at that timestamp before we enter pointers to each into the index.

To achieve this, we decided to use a temporary hash map when sorting the tuples initially. For each timestamp (for each 22 entries, as the tuples are in order of time), we hash each tuple into the temporary map using the grid square they are in. If there is a collision, *then* we write pointers to the tuples involved in the collision to the main index. In this case, however, we will exclude the player ids. We will instead just allow for collisions in the database, inserting into the next free space with a pointer to each tuple. The reason for this is to save time when querying the index. If there is no entry for a grid square at a time, we can just skip straight to the next time. If there *is* an entry, we don't have to check if there is one for every player id, we just keep printing out until we get to a pointer that is not at that timestamp.

So, to receive data from these tuples, where two or more players are in a grid square, we can use a similar algorithm to that which is in the previous question, but with the above mentioned changes for efficiency:

```
For each gridSquare:
    time = 0
    while (time < maxMatchTime):
        if(hash(gridSquare, time) != null):
            pointer = hash(gridSquare, time)
            do:
                print(pointer.value.timestamp)
                pointer += pointerSize
            while(pointer.timestamp == time)
        time += interval
```

This will go through each square and time. If there is a single entry for a gridSquare at a time, then we know there were multiple players in that grid square at that time. This means there are sequential pointers to the players in that same grid square at that time in the index, so we just keep going through the pointers until one doesn't have the correct timestamp for the one we are on, indicating we have printed out all the players in the same grid square at the same time.