System Design - S3 + Quad trees (nearest neighbors)

Agenda:

How do we store large files?

In earlier classes, we discussed several problems, including how we dealt with ***metadata***, facebook's newsfeed, and many other systems.

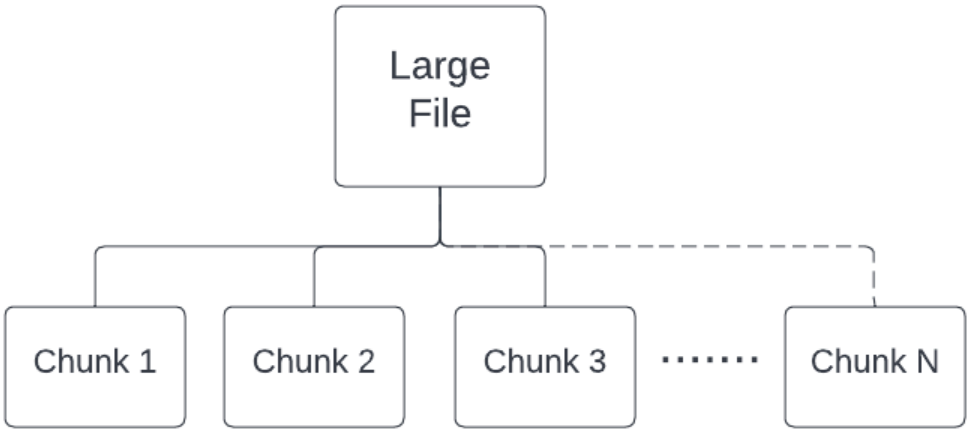
We discussed that for a post made by the user on Facebook with images(or some other media), we don't store the image in the database. We only store the metadata for the post (user_id, post_id, timestamp, etc.). The images/media are stored on different storage systems; from that particular storage system, we get a URL to access the media file. This URL is stored in the database file.

In this class, our main discussion is how to store these large files (not only images but very large files, say a 50 TB file). A large file can be a large video file or a log file containing the actions of the users (login, logout, and other interactions and responses), and it can keep increasing in size.

Conditions for building a large file system:

- Storage should be able to store large files
- Storage should be reliable and durable, and the files stored should not be lost.
- Downloading the uploaded file should be possible
- Analytics should be possible.

One way to store a large file is to divide it into chunks and store the chunks on different machines. So suppose a 50 TB file is divided into chunks. What will be the size of the chunks? If you divide a 50 TB file into chunks of 1 MB, the number of parts will be
$50TB/1MB = (50 * 10^6)$ MB / 1 MB = $5 * 10^7$ parts.



From this, we can conclude that if we keep the size of the chunk very small, then the number of parts of the file will be very high. It can result in issues like

1. **Collation of the parts:** concatenating too many files and returning them to the client will be overhead.
2. **Cost of entries:** We must keep metadata for the chunks,i.e., for a given chunk of a file, it is present on which machine. If we store metadata for every file, this is also an overhead.

HDFS

HDFS stands for Hadoop Distributed File System. Below are certain terminologies related to HDFS:

- The default chunk size is 128 MB in HDFS 2.0. However, in HDFS 1.0, it was 64 MB.
- The metadata table we maintain to store chunk information is known as the '**NameNode server**'. It keeps mapping that chunks are present on which machine(**data node**) for a certain file. Say, for File 1, chunk 1 is present on machine 3.
- In HDFS, there will be only one name node server, and it will be replicated.

You may wonder why the chunk size is 128 MB.
The reason is that large file systems are built for certain operations like storing, downloading large files, or doing some analytics. And based on the types of operations, benchmarking is done to choose a proper chunk size. It is like ‘what is the normal file size for which most people are using the system’ and keeping chunk size accordingly so that system's performance is best.

For example,
chunk size of X1 performance is P1
chunk size of X2 performance is P2,
Similarly, doing benchmarking for different chunk sizes.
And then choosing the chunk size that gives the best performance.

In a nutshell, we can say benchmarking is done for the most common operations which people will be doing while using their system, and HDFS comes up with a value of default chunk size.

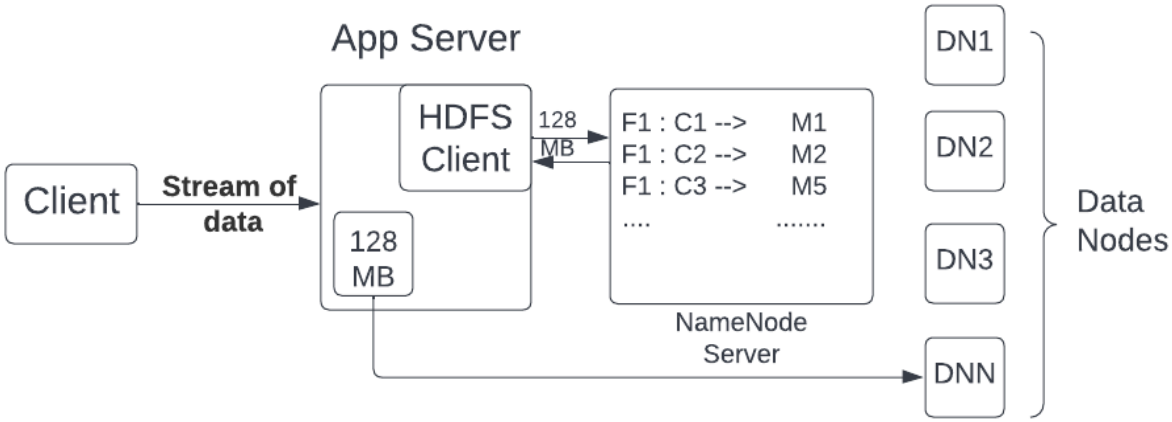
Making System reliable: We know that to make the distributed system reliable, we never store data on a single machine; we replicate it. Here also, a chunk cannot be stored on a single machine to make the system reliable. It needs to be saved on multiple machines. We will keep chunks on different data nodes and replicate them on other data nodes so that even if a machine goes down, we do not lose a particular chunk.

Rack Aware Algorithm: For more reliability, keep data on different racks so that we do not lose our data even if a rack goes down. We avoid replicating the chunks on the machines of the same rack. This is because if there comes an issue with the power supply, the rack will go down, and data won't be available anywhere else.

So this was about chunk divisions and storing them on HDD. Now comes the question of who does this division part.

The answer is it depends on the use case.

- Suppose there is a client who wants to upload a large file. The client requests the app server and starts sending the stream of data. The app server on the other side has a client (HDFS client) running on it.
- HDFS also has a NameNode server to store metadata and data nodes to keep the actual data.
- The app server will call the name node server to get the default chunk size, NameNode server will respond to it (say, the default chunk size is 128 MB).
- Now, the app server knows that it needs to make chunks of 128 MB. As soon as the app server collects 128 MB of data (equal to the chunk size) from the data stream, it sends the data to a data node after storing metadata about the chunk. Metadata about the chunk is stored in the name node server. For example, for a given file F1, nth chunk - Cn is stored in 3rd data node - D3.
- The client keeps on sending a stream of data, and again when the data received by the app server becomes equal to chunk size 128 MB (or the app server receives the end of the file), **metadata about the chunk is stored in the name node server first and then chunk it send to the data node.**



Briefly, the app server keeps receiving data; as soon as it reaches the threshold, it asks the name node server, 'where to persist it?', then it stores the data on the hard disk on a particular data node received from the name node server.

Few points to consider:

- For a file of 200MB, if the default chunk size is 128 MB, then it will be divided into two chunks, one of 128 MB and the other of 72 MB because it is the only data one will be receiving for the given file before the end of the data stream is reached.
- The chunks will not be saved on a single machine. We replicate the data, and we can have a master-slave architecture where the data saved on one node is replicated to two different nodes.
- We don't expect very good latency for storage systems with large files since there is only a single stream of data.

Downloading a file

Similar to upload, the client requests the app server to download a file.

- Suppose the app server receives a request for downloading file F1. It will ask the name node server about the related information of the file, how many chunks are present, and from which data nodes to get those chunks.
- The name node server returns the metadata, say for File 1, goto data node 2 for chunk 1, to data node 3 for chunk 2, and so on. The application server will go to the particular data nodes and will fetch the data.
- As soon as the app server receives the first chunk, it sends the data to the client in a data stream. It is similar to what happened during the upload. Next, we receive the subsequent chunks and do the same.

(More about data streaming will be discussed in the Hotstar case study)

Torrent example: Do you know how a file is downloaded very quickly from the torrent?

What is happening in the background is very similar to what we have discussed. The file is broken into multiple parts. If a movie of 1000MB is broken into 100 parts, we have 100 parts of 10 MB each.

If 100 people on torrent have this movie, then I can do 100 downloads in parallel. I can go to the first person and ask for part 1, the second person for part 2, and so forth. Whoever is done first, I can ask the person for the next part, which I haven't asked anybody yet. If a person is really fast and I have gotten a lot of parts, then I can even ask him for the remaining part, which I am receiving from someone, but the download rate is very slow.

https://www.explainthatstuff.com/howbittorrentworks.html

Nearest Neighbors

There are a lot of systems that are built on locations, and location-based systems are unique kinds of systems that require different kinds of approaches to design. Conventional database systems don't work for location-based systems.

We will start the discussion with a problem statement:
On google Maps, wherever you are, you can search for nearby businesses, like restaurants, hotels, etc. If you were to design this kind of feature, how would you design the feature that will find you nearest X number of neighbors(say ten nearby restaurants)?

Bruteforce

Well, the brute-force approach can simply get all restaurants along with their locations (latitude and longitude) and then find the distance from our current location (latitude and longitude). Simply euclidian distance between two points (x1, y1) and (x2, y2) in 2D space can be calculated with the formula $d = \sqrt{[(x2 – x1)^2 + (y2 – y1)^2}$.
This will calculate the distance around all points around the globe to get our X (say 10) nearest neighbors. The approach will take a lot of time.

Finding Locations Inside a Square

We cannot use the circle to get all the locations around the current position because there is no way to mark the region; therefore, using a square to do the same.

Another approach is to draw a square from our current location and then consider all the points/restaurants lying inside it to calculate X nearest ones. We can use the query:

```
SELECT * FROM places WHERE lat < x + k AND lat > x -k  AND long < y + k AND long > y-k
```

Here **‘x’** and **‘y’** are the coordinates of our current location, **‘lat’** is latitude, **‘long’** is longitude, and **‘k’** is the distance from the point **(x,y)**.

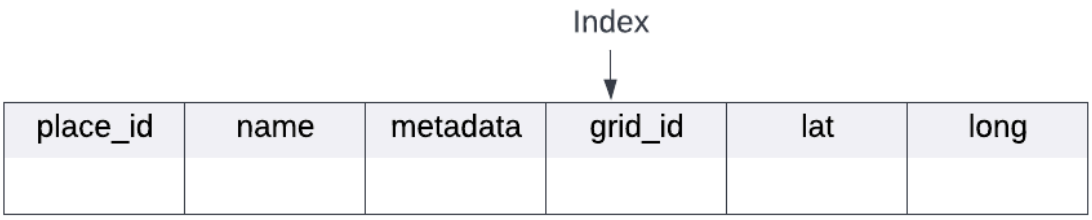
However, this approach has some issues:

1. Finding the right ‘**k**’ is difficult.
2. Query time will be high: Only one of lat or long index can be used in the above query and hence the query will end up spanning a lot of points. .

Grid Approach

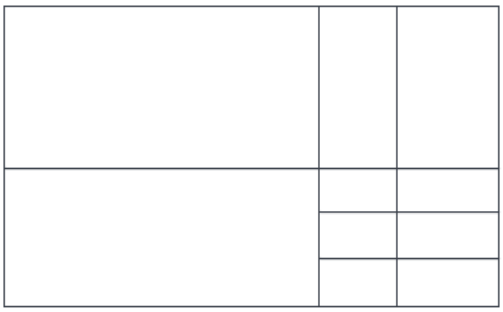
We can break the entire world into small grids (maybe 1 km sq. grids). Then to get all the points, we only need to consider the locations in the grid of our current location or the points in the adjacent grids. If there are enough points in these grids, then we can get all the nearest neighbors. The query and to get all the neighbors is depicted below:

```
SELECT * FROM places WHERE  grid_id IN (grid1, grid2, grid3……)
```



What should be the size of the grid?

It is not ideal to have a uniform grid size worldwide. The grid size should be small for dense areas and large for sparse areas. For example, the grid size needs to be very large for the ocean and very small for densely populated areas. The thumb rule is that size of the grid is to be decided based on the number of points it contains. We need to design variable-size grids so that they have just enough points. Points are also dynamically evolving sets of places.



Varible Size Grids

Dividing the entire world into variable-size grids so that every grid has approximately 100 points

So our problem statement reduces to preprocess all the places in the world so that we can get variable-size grids containing 100 points. We also need to have some algorithm to add or delete a point (a location such as a restaurant).

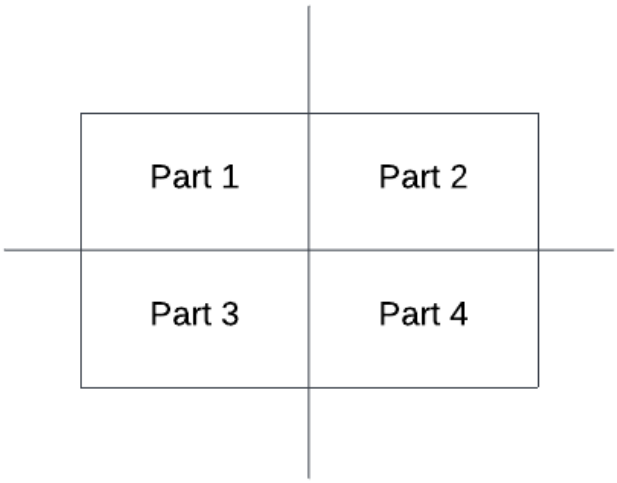
This can be achieved using **quadtrees**.

QuadTree

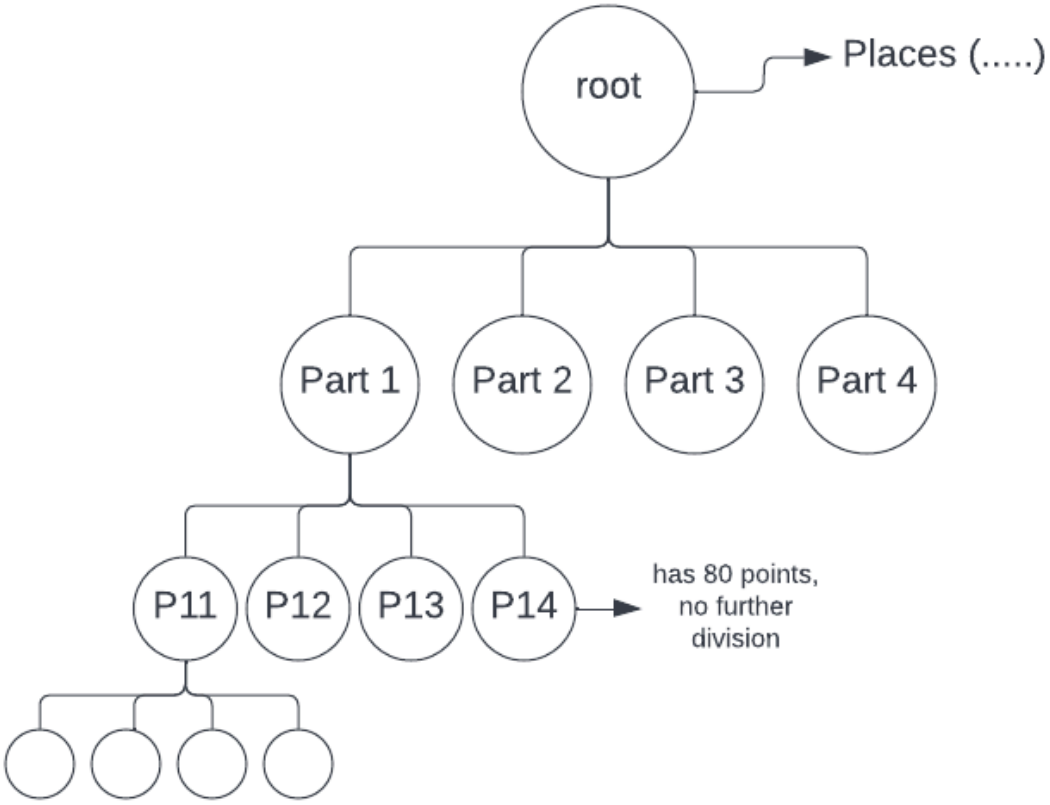
Creation

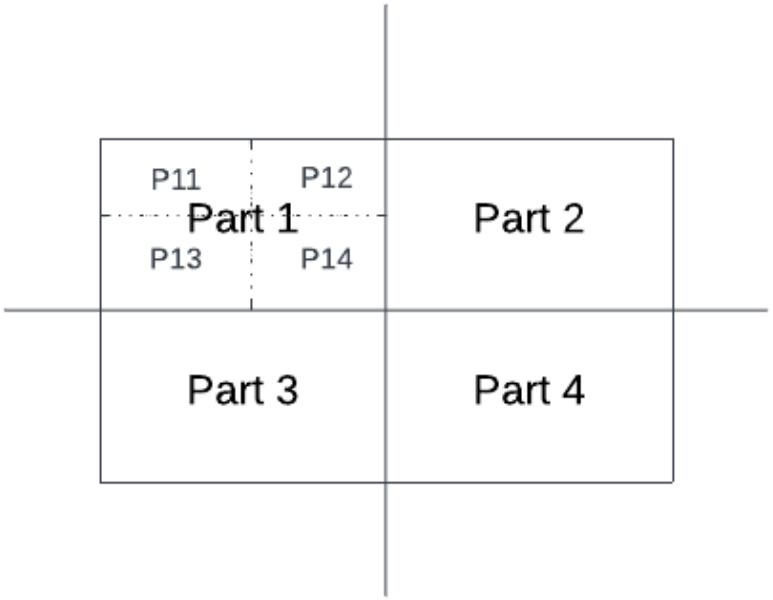
Imagine the entire world with billions of points (think of a world map, a rectangle with points all over).

- We can say that the entire world is a root of a tree and has all of the places of the world. We create a tree; if the current node has more than 100 points, then we need to create its four children (four because we need to create the same shape as a rectangle by splitting the bigger grid into four parts).



- We recursively repeat the process for the four parts as well. If any children have more than 100 points, it further divides itself into four children. Every child has the same shape as the parent, a rectangle.

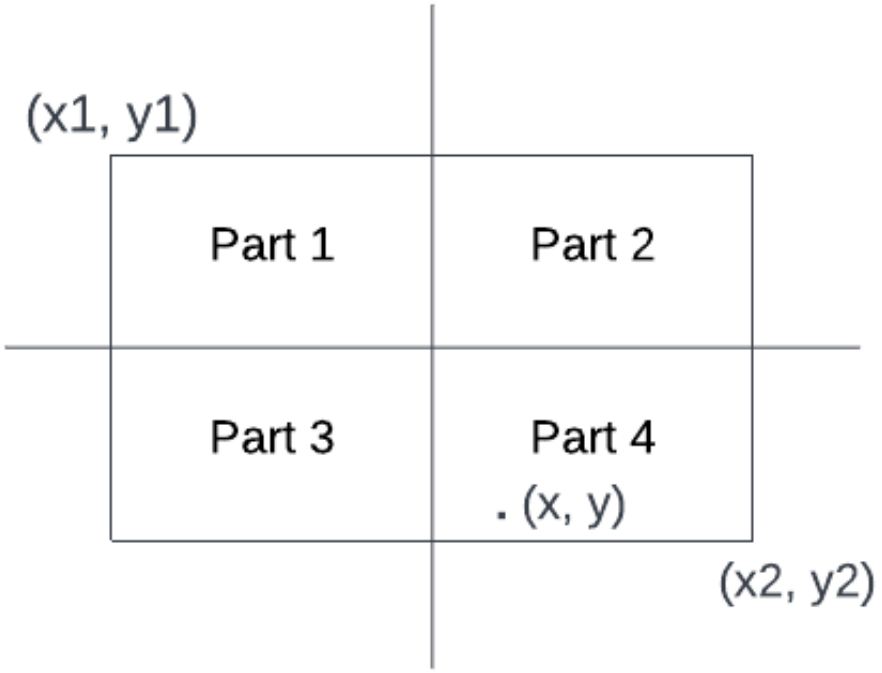

root
Places (.....)
Part 1
Part 2
Part 3
Part 4
P11
P12
P13
P14
has 80 points,
no further
division


P11
P12
Part 1
P13
P14
Part 2
Part 3
Part 4

- All the leaf nodes in the tree will have less than 100 points/places. And the tree's height will be **~log(N), N** being the number of places in the world.

Finding Grid ID

Now, suppose I give you my location **(x,y)** and ask you which grid/leaf I belong to. How will you do that? You can assume the whole world extends between coordinates (x1, y1) and (x2, y2).



What I can do is calculate the middle point for the x and y coordinates, X_{mid}= (x1+x2) / 2, Y_{mid}= (y1 + y2) / 2. And then, I can check if the x is bigger than X_{mid}. If yes, then the point will be present in either part 2 or 4, and if smaller than X_{mid}, the point will be in part 1 or 3. After that, I can compare y with Y_{mid} to get the exact quadrant.

This process will be used to get the exact grid/leaf if I start from the root node, every time choosing one part out of 4 by the above-described process as I know exactly which child we need to go to. Writing the process recursively:

```
findgrid(x, y, root):
    X1, Y1 = root.left.corner
    X2, Y2 = root.right.corner
    If root.children.empty():      // root is already a leaf node
            Return root.gridno      // returning grid number
```

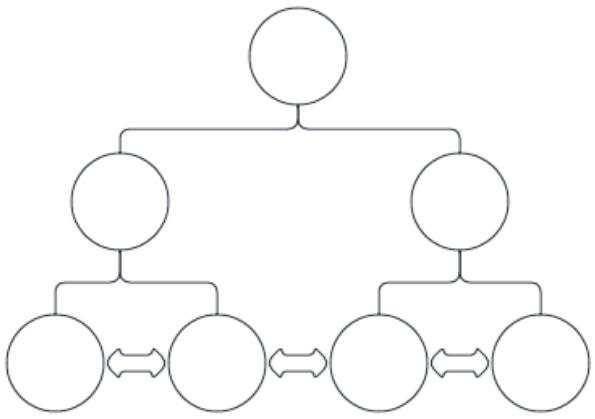
```
If x > (X1 + X2) / 2:
        If y > (Y1 + Y2) / 2:
                findgrid(x, y, root.children[1])
        Else:
                findgrid(x, y, root.children[3])

 Else  y > (Y1 + Y2) / 2:
       If x > (X1 + X2) / 2:
                findgrid(x, y, root.children[0])
        Else:
                findgrid(x, y, root.children[2])
```

What is the time complexity of finding the grid to which I belong by above-mentioned method? It will be equal to the height of the tree: log(N).

Once we find the grid, it becomes easy to calculate the nearby points. Every place in the world has been assigned a grid number and it is stored in MySQL DB. We can easily get all the required neighboring points. If neighbors are not enough, we also have to consider neighboring grids.

To find the neighboring grids:

- Next pointer Sibling: While creating the tree, if we also maintain the next pointer for the leaves, then we can easily get the neighbors. It becomes easy to find siblings. We can travel to the left or right of the leaf to get the siblings.



- Another way is by picking a point very close to the grid in all eight directions. For a point (X, Y) at the boundary, we can move X slightly, say X + 0.1, and check in which grid

point (X+ 0.1, Y) lies. It will be a log(N) search for all 8 directions, and we will get all the grid ids.

Add a new Place

If I have to add a point (x, y), first, I will check which leaf node/ grid it belongs to. (Same process as finding a grid_id). And I will try to add one more place in the grid. If the total size of points in the grid remains less than the threshold (100), then I simply add. Otherwise, I will split the grid into four parts/children and redivide the points into four parts. It will be done by going to MySQL DB and updating the grid id for these 100 places.

Delete an existing place

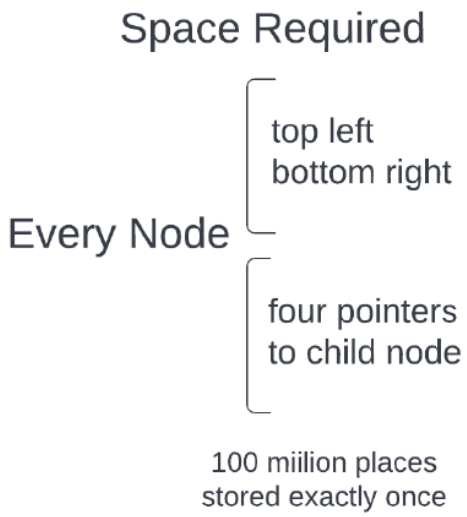
Deletion is exactly the opposite of addition. If I delete a point and the summation of all the points in the four children becomes less than 100, then I can delete the children and go back to the parent. However, the deletion part is not that common.

How to store a quad tree

For 100 million places, how can we store a quadtree in a machine?
What will we be storing in the machine, and how much space will it need?

So what do you think, whether it's possible to store data, i.e., 100 million places in a single machine or not?

Well, to store a quadtree, we have to store the **top-left** and **bottom-right** coordinates for each node. Apart from that, we will store **four pointers** to child nodes. The 100 million places will be stored in leaves only; every time a node contains more than X(say 100) places, we will split it into four children. In the end, all leaves will contain less than equal to 100 places, and every place will be present in exactly one leaf node.



Let's do some math for the number of nodes and space required.

Say every leaf stores one place (for 100 million places); there will be 100 million leaf nodes in the quadtree.

Number of parents of leaf nodes will be = (100 million) / 4
Parents of parents will be = (100 million) / 4
And so on.

So total number of nodes = 10^8 + 10^8/4 + 10^8/16 + 10^8/64 ……..
= 10^8 (1 + 1/4 + 1/16 + 1/64 …….)
The above series is an infinite G.P., and we can calculate the sum using formula 1/(1-r) { for series 1 + r + r^2 + r^4 + ……..}. In the above series, r = 1/4
*The sum will 10^8 * 1/(1-(1/4)) = 10^8 * (4/3) = 1.33 * 10^8*

*If we assume every leaf node has an average of 20 places, the number of nodes will be equal to (1.33 * 10^8) / (average number of places in a leaf) = (1.33 * 10^8) / 20*

*(1.33 * 10^8) / 20 = (1.33 * 5 * 10^6) =* ***6.5 million nodes***

So we have to store *100 million places + 6.5 million nodes*.

Now calculating space needed:

For every node, we need to store top-left and bottom-right coordinates and four pointers to children nodes. Top-left and bottom-right are location coordinates (latitude and longitude), and let's assume we need two doubles (16 bytes) to get the required amount of precision.

*For boundary, the space required will be 16 * 4 = 64 bytes.*

Every pointer is an integer since it points to a memory location,

*Storage required for 4 pointers = 4bytes * 4 = 16 bytes*

Every node requires 64 + 16 = 80 bytes

To store 100 million places, the storage required (say latitude and longitude each is 16 bytes)
*= 10^8 * 32*

Total space required = space required for nodes + space required for places
*= 6.5 million * 80 bytes + 100 million * 32 bytes*
*= 520 * 10^8 bytes + 3200 * 10^8 bytes*
*= ~ 4000 million bytes = **4GB***

So the total space required is 4GB to store a quadtree, and it can easily fit inside the main memory. All we need is to make sure that there are copies of this data is multiple machines so that even if a machine goes down, we have access to data.

A lot of production systems have 64GB RAM, and 4GB is not a problem to store.

Problem statements for the next class

1. How can we find the nearest taxi cabs and get matched to one of them? Note that taxis can move, and their location is not fixed :) {Uber case study}

Extra Reference Material

1. HDFS Architecture: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

2. HDFS Archival: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html
3. GeoHash
    a. https://www.youtube.com/watch?v=UaMzra18TD8
    b. https://www.youtube.com/watch?v=vGKs-c1nQYU
4. Comparison of Geohash, QuadTree, R-Tree https://tarunjain07.medium.com/geospatial-geohash-notes-15cbc50b329d