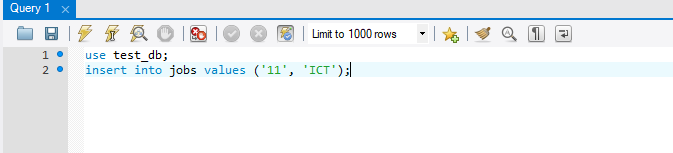
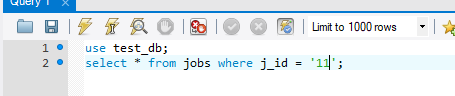
***Extra DB***

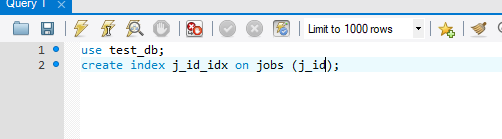
1. **What is a database index? Explain in your own words?  
     
   Ans.** A database index is a very powerful mechanism to optimize query execution time.

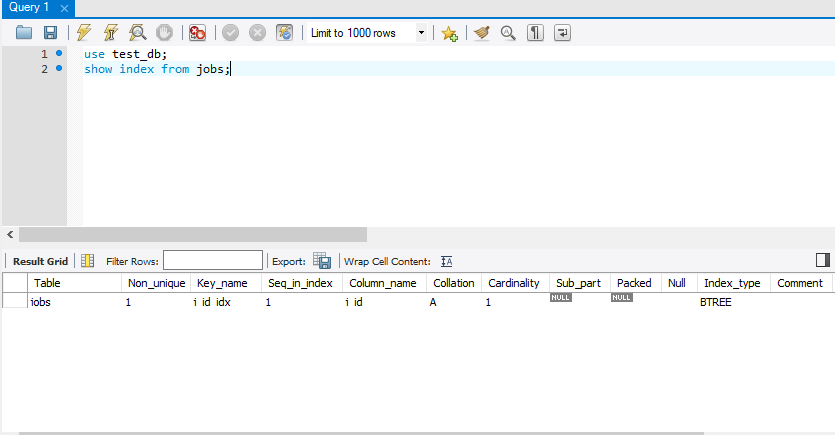
By example:   
here I have created a database (test\_db) with one table inside (jobs) and no index on it.

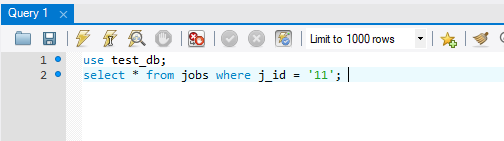
Then, I have inserted a record as shown in the photo …

  
the query executed time is 0.157 sec …  
now I create select query to select job by ID like :

The time required to do this select query is 0.047 sec …  


Then I created index on jobs table on field (j\_id) as:

Here is the description of index:

  
Then I rerun last select and the executed time is 0 sec …



We can say that the index is very important especially when we work with huge database records.  
And we can say:  database is a set of tables, and table is a set of indexes. And each index is, essentially,   
B-tree storing its data.  
  
Note:

Primary and unique keys automatically have indexes, but you might want to create an index on a [foreign key](https://docs.oracle.com/cloud/latest/db112/CNCPT/glossary.htm#i996900)***.***

***----------------------------------------------------------------------------------------------------------------------***

1. **Considering the world database used in the assignments. Here's a query:  
   SELECT \* FROM city WHERE Population > 5000000;**

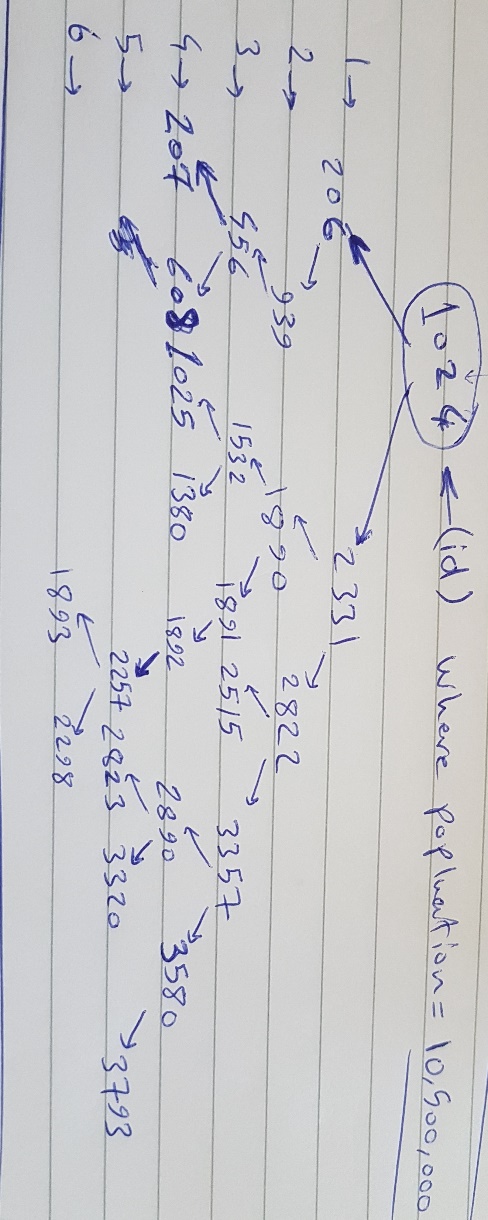
**Which rows in the city table will the DBMS look at to answer your query?**

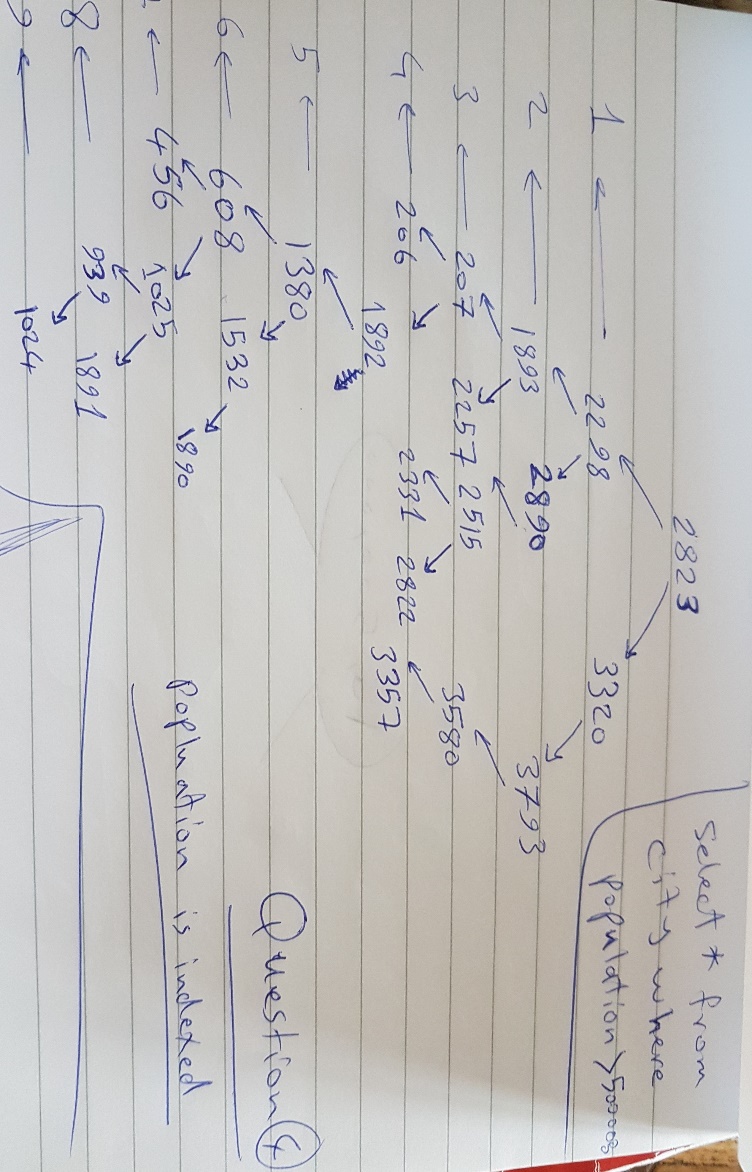
**Ans.** I think the DBMS will fetch all records in a table then apply (where) condition to select match ones.

1. **Indices can be implemented with a binary search tree. Draw a binary tree for the Population of the top 24 largest cities.**  
   **Ans.** The query is:  
   use world;

SELECT \* FROM city ORDER BY population DESC LIMIT 24;

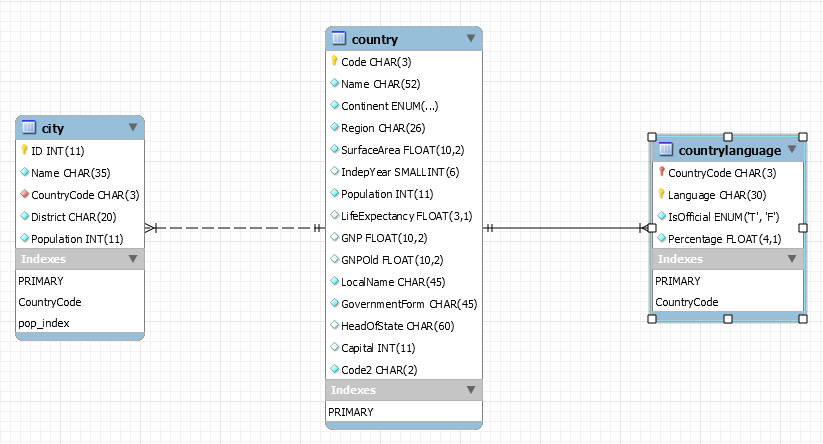
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **CountryCode** | **District** | **Population** |
| 1024 | Mumbai (Bombay) | IND | Maharashtra | 10500000 |
| 2331 | Seoul | KOR | Seoul | 9981619 |
| 206 | SÃ£o Paulo | BRA | SÃ£o Paulo | 9968485 |
| 1890 | Shanghai | CHN | Shanghai | 9696300 |
| 939 | Jakarta | IDN | Jakarta Raya | 9604900 |
| 2822 | Karachi | PAK | Sindh | 9269265 |
| 3357 | Istanbul | TUR | Istanbul | 8787958 |
| 2515 | Ciudad de MÃ©xico | MEX | Distrito Federal | 8591309 |
| 3580 | Moscow | RUS | Moscow (City) | 8389200 |
| 3793 | New York | USA | New York | 8008278 |
| 1532 | Tokyo | JPN | Tokyo-to | 7980230 |
| 1891 | Peking | CHN | Peking | 7472000 |
| 456 | London | GBR | England | 7285000 |
| 1025 | Delhi | IND | Delhi | 7206704 |
| 608 | Cairo | EGY | Kairo | 6789479 |
| 1380 | Teheran | IRN | Teheran | 6758845 |
| 2890 | Lima | PER | Lima | 6464693 |
| 1892 | Chongqing | CHN | Chongqing | 6351600 |
| 3320 | Bangkok | THA | Bangkok | 6320174 |
| 2257 | SantafÃ© de BogotÃ¡ | COL | SantafÃ© de BogotÃ¡ | 6260862 |
| 207 | Rio de Janeiro | BRA | Rio de Janeiro | 5598953 |
| 1893 | Tianjin | CHN | Tianjin | 5286800 |
| 2298 | Kinshasa | COD | Kinshasa | 5064000 |
| 2823 | Lahore | PAK | Punjab | 5063499 |

Binary search tree to implement this is like:

1. **Explain how the following query SELECT \* FROM city WHERE Population > 5000000; will work when we set an index on the Population column. How does the database search the binary tree to answer the query?**

**Ans.** Since we apply index on population column so the column will be order ascending.

1. **Pick a database schema and explain which columns you think should get an index and explain for each column why you think that's a good idea.**

**Ans.**

For city table:   
1. ID as PK.   
2. countryCode as Foreign Key.  
3. Population.

For country table:  
1. Code as PK.  
2. surfaceArea.  
3. IndepYear.

For countryLangyage table:

1. Language as PK.  
2. Percentage.

I prefer integer or numeral data type to be indexed.

1. **Implement a binary search tree that stores numbers in NodeJS.  
     
   Ans.**

'use strict';

function Node(data) {

this.data = data;

this.left = null;

this.right = null;

};

function BinarySearchTree() {

this.root = null;

};

BinarySearchTree.prototype.add = function (data) {

const node = new Node(data);

if (!this.root) {

this.root = node;

} else {

let current = this.root;

while (current) {

if (node.data < current.data) {

if (!current.left) {

current.left = node;

break;

}

current = current.left;

} else if (node.data > current.data) {

if (!current.right) {

current.right = node;

break;

}

current = current.right;

} else {

break;

}

}

}

};

BinarySearchTree.prototype.remove = function (data) {

let that = this;

const removeNode = function (node, data) {

if (!node) {

return null;

}

if (data === node.data) {

if (!node.left && !node.right) {

return null;

}

if (!node.left) {

return node.right;

}

if (!node.right) {

return node.left;

}

// 2 children

let temp = that.getMin(node.right);

node.data = temp;

node.right = removeNode(node.right, temp);

return node;

} else if (data < node.data) {

node.left = removeNode(node.left, data);

return node;

} else {

node.right = removeNode(node.right, data);

return node;

}

};

this.root = removeNode(this.root, data);

};

BinarySearchTree.prototype.contains = function (data) {

let current = this.root;

while (current) {

if (data === current.data) {

return true;

}

if (data < current.data) {

current = current.left;

} else {

current = current.right;

}

}

return false;

};

BinarySearchTree.prototype.print = function () {

if (!this.root) {

return console.log('No root node found');

}

let newline = new Node('|');

let queue = [this.root, newline];

let string = '';

while (queue.length) {

const node = queue.shift();

string += node.data.toString() + ' ';

if (node === newline && queue.length) {

queue.push(newline);

}

if (node.left) {

queue.push(node.left);

}

if (node.right) {

queue.push(node.right);

}

}

console.log(string.slice(0, -2).trim());

};

BinarySearchTree.prototype.getMin = function (node) {

if (!node) {

node = this.root;

}

while (node.left) {

node = node.left;

}

return node.data;

};

BinarySearchTree.prototype.getMax = function (node) {

if (!node) {

node = this.root;

}

while (node.right) {

node = node.right;

}

return node.data;

};

BinarySearchTree.prototype.getDepth = function () {

let node = this.root;

let maxDepth = 0;

let traverse = function (node, depth) {

if (!node) return null;

if (node) {

maxDepth = depth > maxDepth ? depth : maxDepth;

traverse(node.left, depth + 1);

traverse(node.right, depth + 1);

}

};

traverse(node, 0);

return maxDepth;

};

const binarySearchTree = new BinarySearchTree();

binarySearchTree.add(5);

binarySearchTree.add(3);

binarySearchTree.add(7);

binarySearchTree.add(2);

binarySearchTree.add(4);

binarySearchTree.add(4);

binarySearchTree.add(6);

binarySearchTree.add(8);

binarySearchTree.add(22);

binarySearchTree.add(25);

binarySearchTree.add(30);

binarySearchTree.add(29);

binarySearchTree.add(23);

binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8

console.log('min is: ', binarySearchTree.getMin()); // => 2

console.log('max is: ', binarySearchTree.getMax()); // => 8

console.log('Depth of BST: ', binarySearchTree.getDepth());

console.log('tree contains 30 is:', binarySearchTree.contains(30)); // => true

console.log('tree contains 9 is: ', binarySearchTree.contains(9)); // => false

binarySearchTree.remove(11); // remove non existing node

binarySearchTree.print(); // => 5 | 3 7 | 2 4 6 8

binarySearchTree.remove(5); // remove 5, 6 goes up

binarySearchTree.print(); // => 6 | 3 7 | 2 4 8

binarySearchTree.remove(7); // remove 7, 8 goes up

binarySearchTree.print(); // => 6 | 3 8 | 2 4

binarySearchTree.remove(8); // remove 8, the tree becomes unbalanced

binarySearchTree.print(); // => 6 | 3 | 2 4

binarySearchTree.remove(4);

binarySearchTree.remove(2);

binarySearchTree.remove(3);

binarySearchTree.remove(6);

binarySearchTree.print(); // => 'No root node found'