Primary Key and Indexes

This lesson explains the concept of an index and its utility in databases.

Primary Key and Indexes

Every time a query that contains a **WHERE** clause is issued, MySQL has to do a full scan of the table to find matching rows. A linear scan of the table isn't very efficient. Adding an *index* to the table can significantly speed-up search for matching rows. An index's primary purpose is to provide an ordered representation of indexed data. An index works similar to how a book index works. Say if you are looking for a particular word, you can scan the index and find the page numbers where the word appears. This saves you time as now you don't have to flip through every page in the book. Since a book index is sorted in alphabetical order, you can visually very quickly narrow down the portion of the index to inspect. If you are looking for a topic starting with the letter 'M', you can jump to the part of the index with all the topics starting with the letter 'M'. However, the trade-off is that the index itself takes up a few pages of the book.

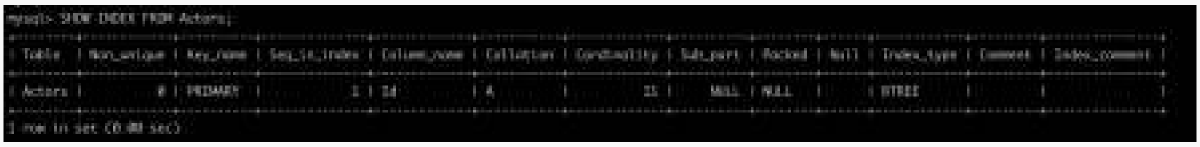
Connect to the terminal below by clicking in the widget. Once connected, the command line prompt will show up. Enter or copy and paste the command **./DataJek/Lessons/16lesson.sh** and wait for the MySQL prompt to start-up.

```
-- The lesson queries are reproduced below for convenient copy/paste into the terminal.

-- Query 1
SHOW INDEX FROM Actors;

-- Query 2
ANALYZE TABLE Actors;
ANALYZE TABLE Actors;
SHOW INDEX FROM Actors;

-- Query 3
INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions)
INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions)
INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions)
```

Terminal

1. We can use the following query to display the indexes on a table:

   ```
   SHOW INDEX FROM Actors;
   ```

   ```
   mysql> SHOW INDEX FROM Actors;
   +--------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
   | Table  | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
   +--------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
   | Actors |          0 | PRIMARY  |            1 | Id          | A         |          11 |     NULL | NULL   |      | BTREE      |         |               |
   +--------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
   1 row in set (0.00 sec)
   ```

   The cardinality shows the number of unique values for the primary key. It's also described as an estimate of the number of unique values in the index and may not be exact for smaller tables. Since we have eleven actors in the **Actors** table we have cardinality as 11. However, if you execute the above query in the console, the cardinality may not come out to be 11. Cardinality is counted based on statistics stored as integers, so the value is not necessarily exact even for small tables. However, if you execute the following command and then check for cardinality, it should come out to be exact.

   ```
   ANALYZE TABLE Actors;
   SHOW INDEX FROM Actors;
   ```

   All columns that make up the primary key must be non-null.

Index

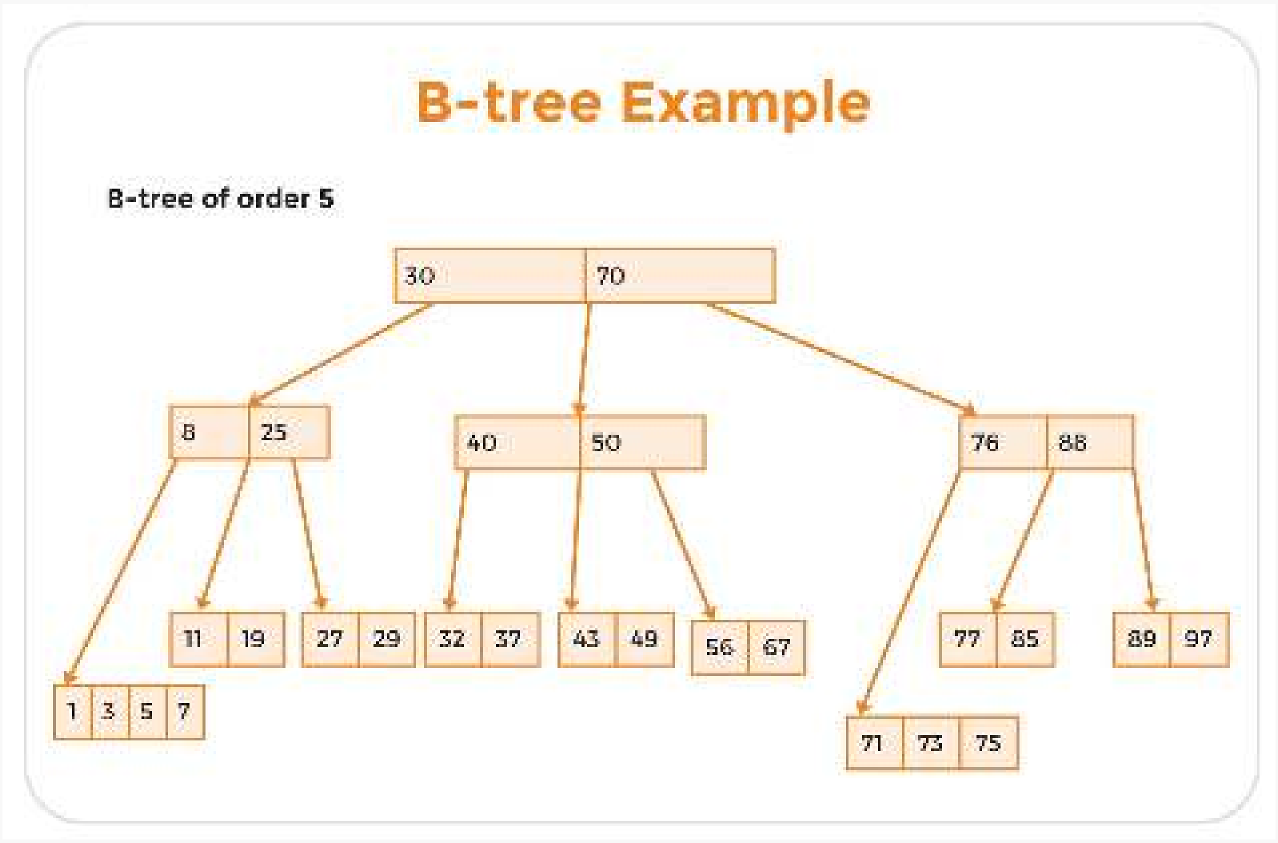
Before proceeding further let's understand the finer details and the general concept of an index. There are two kinds of indexes:

- Clustered
- Non-clustered

Clustered Index

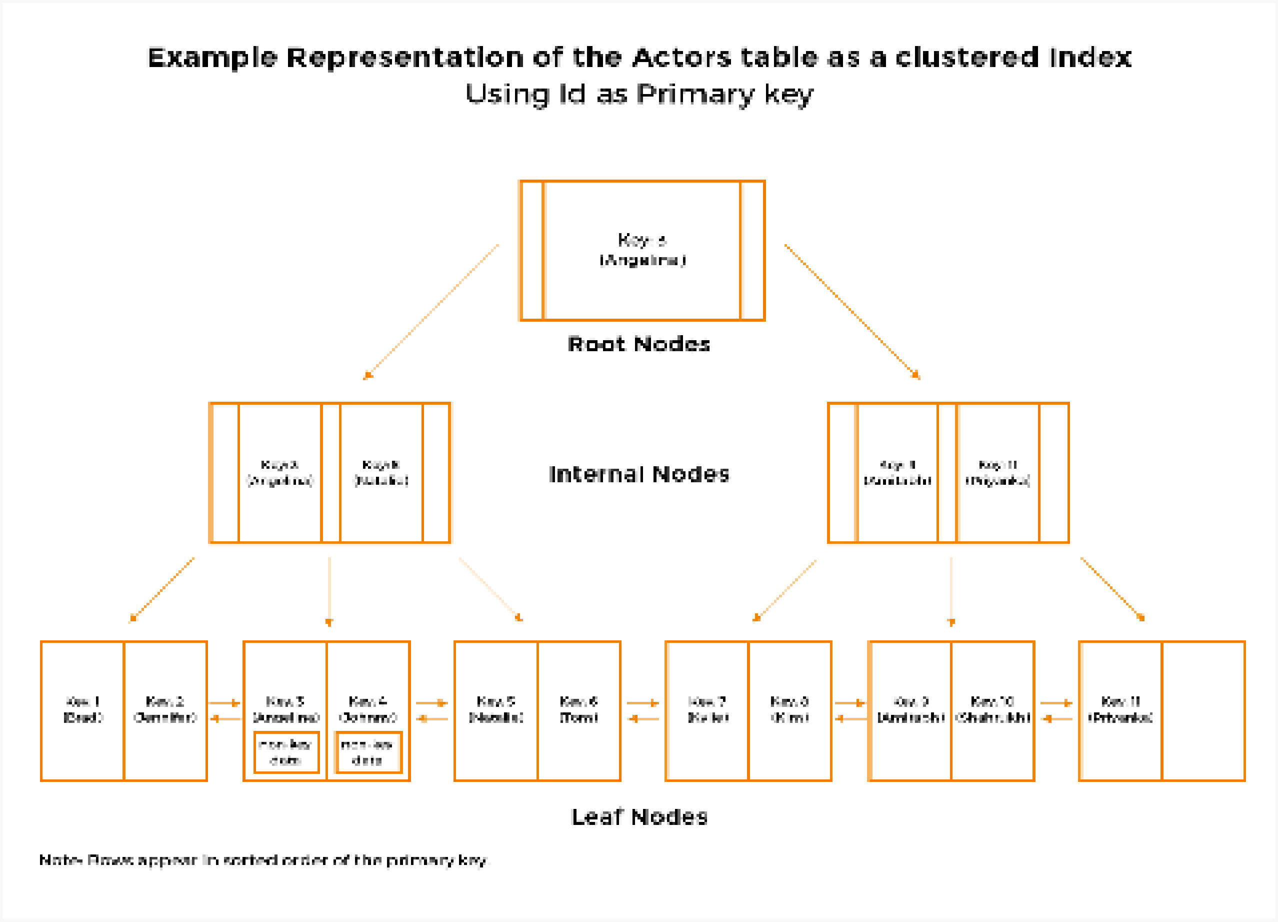
When we insert rows into a table they are written to the underlying physical storage medium such as the hard disk. In the case of a clustered index, the table rows are sorted and kept in a B-tree structure (or an R-tree in the case of a spatial index). The physical arrangement of the rows on the disk mimics the logical order defined by the index. There's no separate data-structure that holds the table's rows. It is a way to inform the database system to *cluster* values physically close to each other on the disk that are close to each other in the index order.

B-trees and their modified version, B+ trees, are common data-structures used to store data on disks. Very large tables can't fit into the main memory of the computer and only portions of such tables are brought from the hard disk to main memory. Searching for the required data becomes easy with B-trees as they guarantee a fixed number of disk reads since they are balanced. A B-tree consists of a root node, branch nodes, and then leaf nodes. An example B-tree looks like as follows:



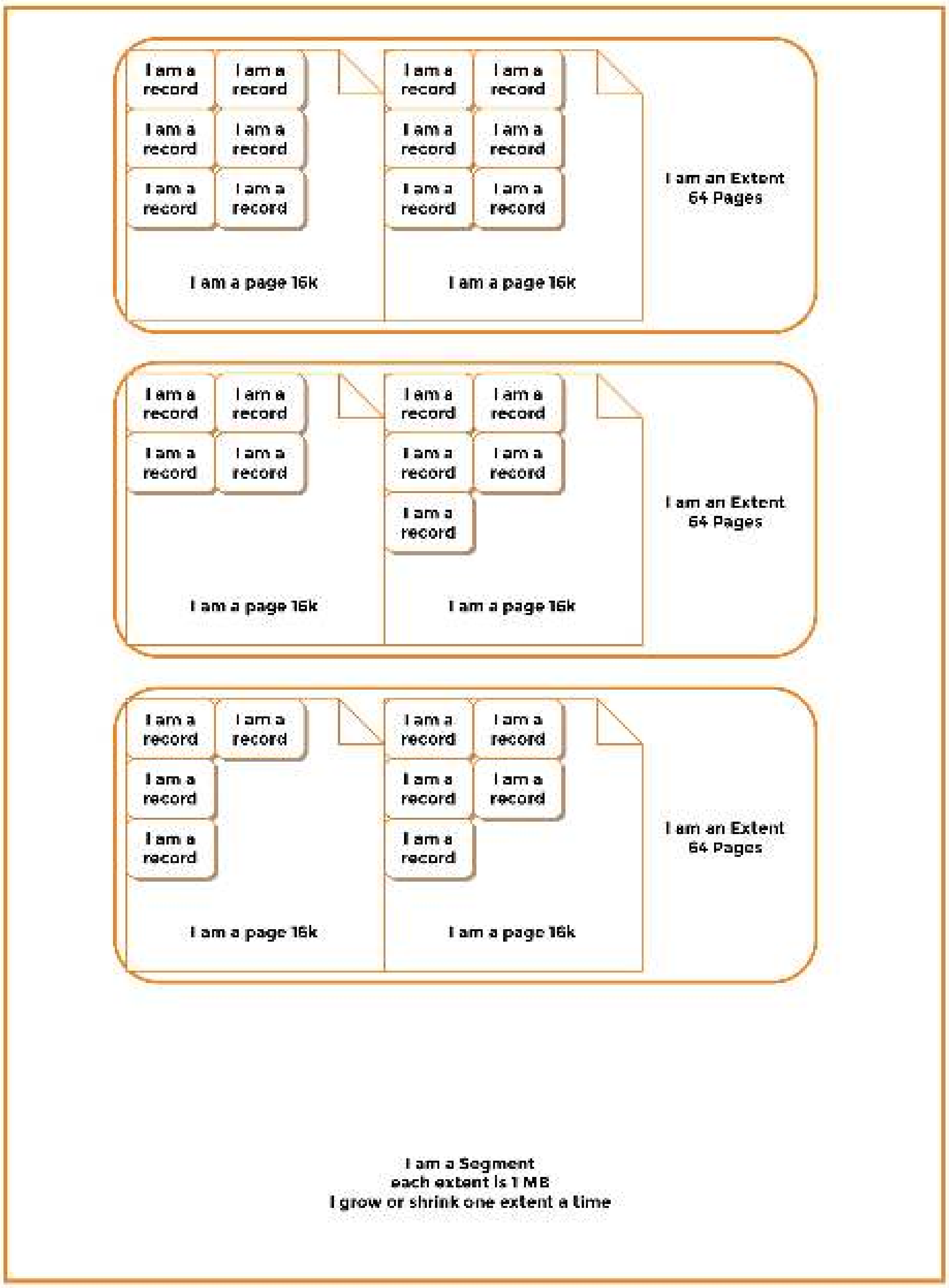
In a B+ tree, the data only lives in the leaf nodes. The root and the internal nodes contain the key on which the B+ tree is sorted. MySQL stores rows in an entity called a page. A page is the smallest unit of data that a database can write to or read from a disk. A page contains rows and forms the leaf node of the B+ tree. An approximation of how the rows forms the leaf node of the B+ tree. An approximation of how the rows

from the **Actors** table will appear as in a B+ tree with the first name as the key is shown below:



The above diagram shows the nodes are linked to each other via next and back pointers. Once we seek the desired data, we can easily navigate the index using these pointers. This is especially useful in range queries. For instance, if we wanted to select all the actors with names greater than Kylie in alphabetical order, we'll seek to the leaf node with the key-value Kylie, and then blindly follow the forward pointers until the end of the index to gather all the rows satisfying the query.

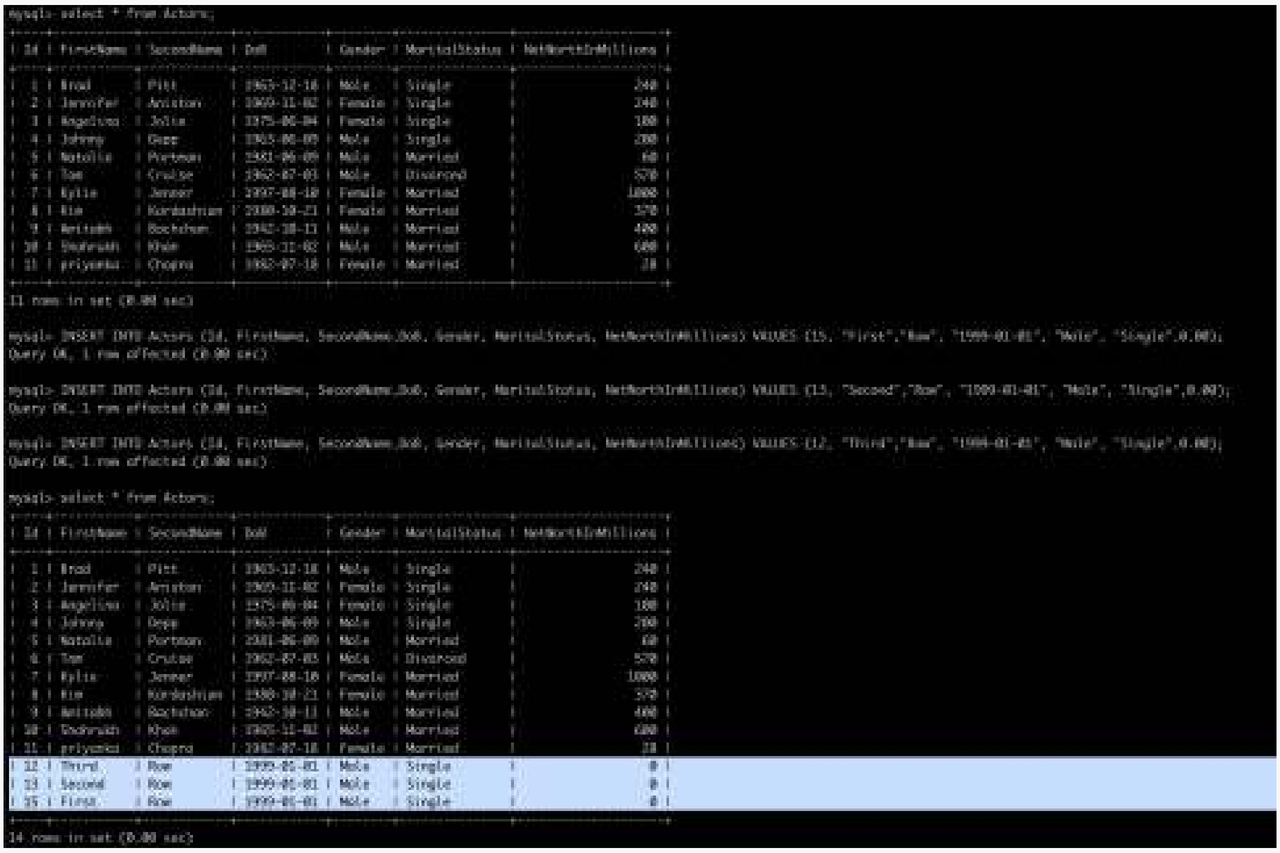
Remember the database works with pages and not rows. When a page is read into memory, all the rows in that page are loaded into memory. In MySQL the default size of a page is fixed at 16KB though it is configurable. A collection of pages forms an extent and a collection of extents forms a segment. Segments in turn form a tablespace. A tablespace consists of tables and their associated indexes. There is a tablespace called the **system tablespace**, and in older versions of MySQL, all user tables were also part of the system tablespace. With later MySQL versions a configuration can be specified to have a separate tablespace for each user table. Conceptually, the page, extent, and segment are related as shown below:



We'll skip the details on pages, extents, etc., as we learned enough to continue our discussion on indexes. Rows are physically written out in the order of the chosen primary key. A primary key uniquely identifies each row in a table. Let's take the example of our **Actors** table. We define the **ID** as the primary key constraint. The rows are stored in the order of the **ID** column. In other words, the clustered index is the table. As a test, we'll add three rows to our **Actors** table. The row we add first will have a

higher ID than the second and third rows. Next, we'll execute a select query and examine the order of the rows returned.

```
INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalSta
tus, NetWorthInMillions) VALUES (15, "First","Row", "1999-01-01", "Ma
le", "Single",0.00);

INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalSta
tus, NetWorthInMillions) VALUES (13, "Second","Row", "1999-01-01", "M
ale", "Single",0.00);

INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalSta
tus, NetWorthInMillions) VALUES (12, "Third","Row", "1999-01-01", "Ma
le", "Single",0.00);
```



```
mysql> select * from Actors;
+----+-----------+------------+------------+--------+---------------+--------------------+
| Id | FirstName | SecondName | DoB        | Gender | MaritalStatus | NetWorthInMillions |
+----+-----------+------------+------------+--------+---------------+--------------------+
|  1 | Brad      | Pitt       | 1963-12-18 | Male   | Single        |                240 |
|  2 | Jennifer  | Aniston    | 1969-11-02 | Female | Single        |                240 |
|  3 | Angelina  | Jolie      | 1975-06-04 | Female | Single        |                100 |
|  4 | Johnny    | Depp       | 1963-06-09 | Male   | Single        |                200 |
|  5 | Natalie   | Portman    | 1981-06-09 | Male   | Married       |                 60 |
|  6 | Tom       | Cruise     | 1962-07-03 | Male   | Divorced      |                570 |
|  7 | Kylie     | Jenner     | 1997-08-10 | Female | Married       |               1000 |
|  8 | Kim       | Kardashian | 1980-10-21 | Female | Married       |                370 |
|  9 | Amitabh   | Bachchan   | 1942-10-11 | Male   | Married       |                400 |
| 10 | Shahrukh  | Khan       | 1965-11-02 | Male   | Married       |                600 |
| 11 | priyanka  | Chopra     | 1982-07-18 | Female | Married       |                 28 |
+----+-----------+------------+------------+--------+---------------+--------------------+
11 rows in set (0.00 sec)

mysql> INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions) VALUES (15, "First","Row", "1999-01-01", "Male", "Single",0.00);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions) VALUES (13, "Second","Row", "1999-01-01", "Male", "Single",0.00);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Actors (Id, FirstName, SecondName,DoB, Gender, MaritalStatus, NetWorthInMillions) VALUES (12, "Third","Row", "1999-01-01", "Male", "Single",0.00);
Query OK, 1 row affected (0.00 sec)

mysql> select * from Actors;
+----+-----------+------------+------------+--------+---------------+--------------------+
| Id | FirstName | SecondName | DoB        | Gender | MaritalStatus | NetWorthInMillions |
+----+-----------+------------+------------+--------+---------------+--------------------+
|  1 | Brad      | Pitt       | 1963-12-18 | Male   | Single        |                240 |
|  2 | Jennifer  | Aniston    | 1969-11-02 | Female | Single        |                240 |
|  3 | Angelina  | Jolie      | 1975-06-04 | Female | Single        |                100 |
|  4 | Johnny    | Depp       | 1963-06-09 | Male   | Single        |                200 |
|  5 | Natalie   | Portman    | 1981-06-09 | Male   | Married       |                 60 |
|  6 | Tom       | Cruise     | 1962-07-03 | Male   | Divorced      |                570 |
|  7 | Kylie     | Jenner     | 1997-08-10 | Female | Married       |               1000 |
|  8 | Kim       | Kardashian | 1980-10-21 | Female | Married       |                370 |
|  9 | Amitabh   | Bachchan   | 1942-10-11 | Male   | Married       |                400 |
| 10 | Shahrukh  | Khan       | 1965-11-02 | Male   | Married       |                600 |
| 11 | priyanka  | Chopra     | 1982-07-18 | Female | Married       |                 28 |
| 12 | Third     | Row        | 1999-01-01 | Male   | Single        |                  0 |
| 13 | Second    | Row        | 1999-01-01 | Male   | Single        |                  0 |
| 15 | First     | Row        | 1999-01-01 | Male   | Single        |                  0 |
+----+-----------+------------+------------+--------+---------------+--------------------+
14 rows in set (0.00 sec)
```

Note that the first row we add appears last and the last row we add appears earlier than the first and the second rows. This is because the rows are retrieved in the order of the index. The leaf nodes of a clustered index contain the actual data called a page.