## 1 hbase的表模型 :

table: hbase的表 支持有多个表

rowkey: 行键 看做是 RDBMS中主键,

行键的类型都是字节的方式, 行键对应的这一列数据都自动按照字典序的升序排序

column family: 列族 看做是RDBMS中的列 一个列族下面可以有多个列(支持上百万个)

在构建一张表的时候, 列族一般不建议过多, 在能满足要求的情况下, 能写一个解决, 坚决不写两个

因为 列族越多, 在后续读写操作的时候, 经历的IO就会越大, 导致效率降低

注意: 在创建一张表的时候, 必须携带两个参数: 表名 + 列族

column : 列 (列的限定符号) 在hbase中, 一个列只能被一个列族所管理 一个列族下可以有多个列

列在插入数据的时候, 可以动态的话指定 需要有那些列即可, 并不需要创建时候指定

cell : 单元格

如何确定一个唯一单元格: rowkey + 列族 + 列名 + 列值

timeStamp : 时间戳

在向hbase操作据的时候, hbase默认会将当前操作的时间记录下来

versionNum : 版本号

hbase支持对每一条数据或者说每个列族下的数据, 定义其版本号, 版本号主要目的就是为了

表示是否需要存储数据历史变化信息, 已经存储多少个问题

## 2. 通过shell命令操作 hbase:

准备:

进入hbase的shell命令的编写模型下: hbase shell

2.1: 如何创建一张hbase表:

语法: create '表名' ,'列族1', '列族2' ....

2.2: 如何查看表:list

2.3: 如何向表中插入数据: put

语法:

put '表名' ,'行键' ,'列族:列名' ,'值'

例如:

put 'ORDER\_INFO' ,'rk0001' ,'C1:name' ,'zhangsan'

2.4: 如何查看某一条数据: get

语法:

get '表名' ,'行键' [,'列族' | '列族:列名' ....]

例如:

get 'ORDER\_INFO', 'rk0001'

get 'ORDER\_INFO', 'rk0001' ,'C1'

get 'ORDER\_INFO', 'rk0001' ,'C1','C2'

get 'ORDER\_INFO', 'rk0001' ,'C1:name','C2'

get 'ORDER\_INFO', 'rk0001' ,'C1:name','C2:birthday'

get 'ORDER\_INFO' ,'rk0002' ,{FORMATTER => 'toString' } --如果有中文 可以加入

2.5: 修改数据:

注意: 修改数据和 添加数据是一致的, 如果hbase中已经存在, 那就是修改了, 如果不存在就是添加操作

2.6: 如何删除数据: delete | deleteall

格式:

delete '表名' ,'行键','列族:列名'

deleteall '表名' , '行键' --删除一整行数据

例子:

delete 'ORDER\_INFO','rk0001' ,'C2:sex'

注意:

1. deleteall 是在 hbase 2.0版本后出现的, 在2.0版本之前, 只需要使用delete这个命令即可完成所有的删除数据工作

2. delete删除数据时候, 只会删除最新版本的数据, 而deleteall 直接将对应数据的所有的历史版本全部删除

2.7: 清空表:

格式: truncate "表名"

流程:

先禁用表, 然后删除表, 最后创建表

2.8: 如何删除表: drop

格式:

drop '表名'

例子:

drop 'ORDER\_INFO'

注意:

在删除表的时候, 一定要先禁用这个表, 否则无法删除

禁用表语法: disable '表名'

启动表语法: enable '表名'

2.9: 如何一次性执行多条hbase的命令 :

操作:

1) 将hbase命令放置到一个文本文件中

2) 将这个文本文件上传到Linux目录下

3) 使用: hbase shell 文本文件路径 即可将命令进行执行操作

如何查看表中有多少条数据:

第一种方式: 使用 count 命令 适合于查看数量较少的表 当数据量比较大的时候, 此种效率效率比较低

第二种方式: 使用hbase提供的MR程序 来进行统计计数

格式: hbase org.apache.hadoop.hbase.mapreduce.RowCounter '表名'

2.10 : 扫描查询: scan

需求一：查询订单所有数据

格式: scan '表名'

例子: scan 'ORDER\_INFO' , {FORMATTER => 'toString'}

注意:

此操作 慎用

需求二：查询订单数据（只显示3条）: limit

格式:

scan '表名', {LIMIT => N, FORMATTER => 'toString'}

例子:

scan 'ORDER\_INFO' ,{LIMIT => 1}

需求三: 查询订单状态、支付方式

格式:

scan '表名', {LIMIT => N, COLUMNS => '列族:列名', FORMATTER => 'toString'}

scan '表名', {LIMIT => N, COLUMNS => ['列族:列名' , '列族:列名' ...] , FORMATTER => 'toString'}

scan '表名', {LIMIT => N, COLUMNS => ['列族' , '列族:列名' ...] , FORMATTER => 'toString'}

例子:

scan 'ORDER\_INFO', {LIMIT => 3, COLUMNS => ['C1:STATUS', 'C1:PAYWAY'], FORMATTER => 'toString'}

scan 'ORDER\_INFO', {LIMIT => 3, COLUMNS => ['C1'], FORMATTER => 'toString'}

需求四：查询指定订单ID的数据并以中文展示

格式:

scan '表名', {ROWPREFIXFILTER => 'rowkey', COLUMNS => ['列族:列名', '列族:列名'], FORMATTER => 'toString'}

例子:

scan 'ORDER\_INFO', {ROWPREFIXFILTER => '02602f66-adc7-40d4-8485-76b5632b5b53', COLUMNS => ['C1:STATUS', 'C1:PAYWAY'], FORMATTER => 'toString'}

2.11: 过滤器

过滤器的使用格式:

scan '表名', { FILTER => "过滤器(比较运算符, '比较器表达式')" }

hbase中常用过滤器:

rowkey过滤器:

RowFilter : 实现行键字符串的比较和过滤

PrefixFilter : rowkey前缀过滤器

列族过滤器 :

FamilyFilter : 列簇过滤器

列名过滤器:

QualifierFilter : 列标识过滤器，只显示对应列名的数据

列值过滤器:

ValueFilter: 值过滤器，找到符合值条件的键值对

SingleColumnValueFilter : 在指定的列族和列中进行比较的值过滤器

SingleColumnValueExcludeFilter : 在指定的列族和列中进行比较的值过滤器 (排除匹配成功的值)

其他过滤器:

PageFilter : 用于实现分页查询的操作

比较运算符 : > < >= <= != =

比较器:

BinaryComparator 匹配完整字节数组

BinaryPrefixComparator 匹配字节数组前缀

NullComparator 匹配空值

SubstringComparator 模糊(包含)匹配子字符串

比较器表达式(比较器的使用方式):

BinaryComparator binary:值

BinaryPrefixComparator binaryprefix:值

NullComparator null

SubstringComparator substring:值

需求一：使用RowFilter查询指定订单ID的数据

只查询订单的ID为：02602f66-adc7-40d4-8485-76b5632b5b53 、订单状态以及支付方式

scan 'ORDER\_INFO', {FILTER =>"RowFilter(=,'binary:02602f66-adc7-40d4-8485-76b5632b5b53')", COLUMNS => ['C1:STATUS', 'C1:PAYWAY'], FORMATTER => 'toString'}

需求二: 查询状态为「已付款」的订单

scan 'ORDER\_INFO' ,{ FILTER => "SingleColumnValueFilter('C1','STATUS',=,'binary:已付款')" ,LIMIT => 1, FORMATTER => 'toString'} -- 找到 STATUS =已付款的整行数据

scan 'ORDER\_INFO' ,{ FILTER => "SingleColumnValueExcludeFilter('C1','STATUS',=,'binary:已付款')" ,LIMIT => 1, FORMATTER => 'toString'} -- 找到 STATUS =已付款的整行数据(不含条件字段)

scan 'ORDER\_INFO' ,{ FILTER => "ValueFilter(=,'binary:已付款')" ,LIMIT => 1, FORMATTER => 'toString'} --找到包含以付款的字段

需求三：查询支付方式为1，且金额大于3000的订单

scan 'ORDER\_INFO' , { FILTER => "SingleColumnValueFilter('C1','PAYWAY',=,'binary:1') AND SingleColumnValueFilter('C1','PAY\_MONEY',>,'binary:3000')" , LIMIT => 1, FORMATTER => 'toString' }

2.12: hbase中数据的自增操作: incr

格式:

incr '表名','rowkey','列蔟:列名',累加值（默认累加1）

注意:

累加值 如果不写, 表示自动+1 如果写正数 表示 +上具体的值 如果写 负数 表示 减少具体的值

所以如果想要实现-1操作

incr '表名','rowkey','列蔟:列名',-1

注意:

如果某一列要实现计数功能，必须要使用incr来创建对应的列

使用put创建的列是不能实现累加的

如果使用incr构建的列, 无法使用 get或者scan的方式查看到其内容, 需要使用 get\_counter 来查看数据

get\_counter '表名','行键','列族:使用incr构建列名'

2.13: 高级的shell命令:

describe : 查看表的信息

格式: describe '表名'

exists: 判断某个表是否存在

格式: exists '表名'

is\_enabled、is\_disabled: 判断某个表是否是启动状态 还是禁止状态

格式:

is\_enabled '表名'

is\_disabled '表名'

alter : 修改表结构信息

新增列族:

alter 'USER\_INFO', 'C3'

# 删除列蔟C3

alter 'USER\_INFO', 'delete' => 'C3'

总结 :

创建表: create '表名' ,'列族名1','列族名2' ...

添加数据: put '表名' ,'行键' ,'列族:列名' ,'列值'

查看某一个条数据: get '表名' , '行键' [,'列族' | '列族:列名' ....]

修改数据: 与添加数据是一致的, 如果表中存在, 那么就是修改操作

删除数据操作: delete '表名','行键' ,'列族:列名' | deleteall '表名','行键' [,'列族:列名'] | truncate '表名'

删除表: drop '表名'

注意: 删除表之前 先禁用表 disable '表名'

如果要启动表: enable '表名'

扫描数据: scan '表名'

查询表条数: count '表' | 使用hbase提供的MR程序

自增操作: incr '表名' ,'行键' ,'列族:列名', 累加值(默认为1)

查询自增字段数据: get\_counter '表名' ,'行键' ,'列族:列名'

过滤器的基本使用:

scan '表名' ,{FILTER =>过滤器(比较运算符,比较器表达式) }

查看列表: list

注意:

1) 如果有中文: 在查询语句的后面 加上 {FORMATTER => 'toString'}

2) 如果数据比较多, 使用limit : {LIMIT => N}

## 3. Java的API操作:

3.1: 如何创建hbase的表:

// 1) 如何创建表: WATER\_BILL 列族为C1 和 C2

@Test

public void test01() throws Exception{

//1. 创建 Java 连接 hbase的连接对象

Configuration conf = HBaseConfiguration.create();

conf.set("hbase.zookeeper.quorum","node1:2181,node2:2181,node3:2181");

Connection hBaseConn = ConnectionFactory.createConnection(conf);

//2. 根据连接对象获取相关的管理对象: admin | Table

Admin admin = hBaseConn.getAdmin();

//3. 执行相关的操作:

//3.1: 判断目标表是否存在, 如果不存在 才创建表 : 如果存在返回true 如果不存在 false

TableName tableName = TableName.valueOf("WATER\_BILL");

boolean flag = admin.tableExists(tableName);

if(!flag){

// 说明表不存在:

//3.3: 创建表的构建器

TableDescriptorBuilder descriptorBuilder = TableDescriptorBuilder.newBuilder(tableName);

List<ColumnFamilyDescriptor> families = new ArrayList<>();

families.add(ColumnFamilyDescriptorBuilder.newBuilder("C1".getBytes()).build());

families.add(ColumnFamilyDescriptorBuilder.newBuilder("C2".getBytes()).build());

//3.4: 在表构建器中添加 表的列族信息

descriptorBuilder.setColumnFamilies(families);

//3.5: 通过表构建器, 构建表信息对象

TableDescriptor desc = descriptorBuilder.build();

//3.2: 执行创建表的操作

admin.createTable(desc);

}

//4. 释放资源

admin.close();

hBaseConn.close();

}

3.2: 新增数据:

//2. 如何向表中添加数据操作: put

@Test

public void test02() throws Exception{

//1. 根据连接工厂构建hbase的连接对象

Configuration conf = HBaseConfiguration.create();

conf.set("hbase.zookeeper.quorum","node1:2181,node2:2181,node3:2181");

Connection hBaseConn = ConnectionFactory.createConnection(conf);

//2. 根据连接对象, 获取相关的管理对象: admin table

Table table = hBaseConn.getTable(TableName.valueOf("WATER\_BILL"));

//3. 执行相关的操作

Put put = new Put("4944191".getBytes());

put.addColumn("C1".getBytes(),"name".getBytes(),"登卫红".getBytes());

put.addColumn("C1".getBytes(),"address".getBytes(),"贵州省铜仁市德江县7单元267室".getBytes());

put.addColumn("C2".getBytes(),"sex".getBytes(),"男".getBytes());

table.put(put);

//4. 释放资源

table.close();

hBaseConn.close();

}

3.3 : 抽取公共代码:

private Connection hBaseConn;

private Admin admin;

private Table table;

private String tableName = "WATER\_BILL" ;

@Before

public void before() throws Exception {

//1. 根据连接工厂构建hbase的连接对象

Configuration conf = HBaseConfiguration.create();

conf.set("hbase.zookeeper.quorum", "node1:2181,node2:2181,node3:2181");

hBaseConn = ConnectionFactory.createConnection(conf);

//2. 根据连接对象, 获取相关的管理对象: admin table

admin = hBaseConn.getAdmin();

table = hBaseConn.getTable(TableName.valueOf(tableName));

}

@Test

public void test03() {

//3. 执行相关操作:

}

@After

public void after() throws Exception {

//4. 释放资源

table.close();

admin.close();

hBaseConn.close();

}

3.4: 删除数据:

//3. 删除数据: 4944191

@Test

public void test03() throws Exception {

//3. 执行相关操作:

Delete delete = new Delete("4944191".getBytes());

delete.addFamily("C2".getBytes());

delete.addColumn("C1".getBytes(),"name".getBytes());

table.delete(delete);

}

3.5: 删除表

@Test

public void test04() throws Exception {

//3. 执行相关操作

if(admin.isTableEnabled(TableName.valueOf(tableName))){

admin.disableTable(TableName.valueOf(tableName));

}

admin.deleteTable(TableName.valueOf(tableName));

}

3.6: 在hbase中 如何 进行导入和导出数据的操作:

导入操作:

hbase org.apache.hadoop.hbase.mapreduce.Import 表名 HDFS数据文件路径

导出操作:

hbase org.apache.hadoop.hbase.mapreduce.Export 表名 HDFS数据文件路径

3.6: 基于scan的扫描查询操作:

需求: 查询2020年6月份所有用户的用水量

@Test

public void test05() throws Exception{

//3. 执行相关的操作: scan 扫描

Scan scan = new Scan();

//3.1: 封装过滤条件

SingleColumnValueFilter singleColumnValueFilter1 =

new SingleColumnValueFilter("C1".getBytes(),"RECORD\_DATE".getBytes(), CompareOperator.GREATER\_OR\_EQUAL,new BinaryComparator("2020-06-01".getBytes()));

SingleColumnValueFilter singleColumnValueFilter2 =

new SingleColumnValueFilter("C1".getBytes(),"RECORD\_DATE".getBytes(), CompareOperator.LESS,new BinaryComparator("2020-07-01".getBytes()));

FilterList filterList = new FilterList();

filterList.addFilter(singleColumnValueFilter1);

filterList.addFilter(singleColumnValueFilter2);

scan.setFilter(filterList);

//3.2: 执行扫描操作 返回多行结果数据

ResultScanner results = table.getScanner(scan);

//3.3: 处理结果集

for (Result result : results) {

List<Cell> cellList = result.listCells(); // 获取 这一行所有的单元格

for (Cell cell : cellList) { // 单元格: rowkey + 列族 + 列名 + 列值

byte[] rowkeyBytes = CellUtil.cloneRow(cell);

byte[] familyBytes = CellUtil.cloneFamily(cell);

byte[] qualifierBytes = CellUtil.cloneQualifier(cell);

byte[] valueBytes = CellUtil.cloneValue(cell);

String rowkey = Bytes.toString(rowkeyBytes);

String family = Bytes.toString(familyBytes);

String qualifier = Bytes.toString(qualifierBytes);

if(qualifier.equals("NUM\_CURRENT") || qualifier.equals("NUM\_PREVIOUS") ||qualifier.equals("TOTAL\_MONEY") || qualifier.equals("NUM\_USAGE") ){

Double value2 = Bytes.toDouble(valueBytes);

System.out.println("rowkey为:"+rowkey +"; family:"+family +"; qualifier:"+qualifier+"; value:"+value2);

}else{

String value1 = Bytes.toString(valueBytes);

System.out.println("rowkey为:"+rowkey +"; family:"+family +"; qualifier:"+qualifier+"; value:"+value1);

}

}

System.out.println("---------------------------------");

}

}

//6. 查询 rowkey 为 9990611 的数据

@Test

@SuppressWarnings("ALL") // 去除警告信息

public void test06() throws Exception{

//3. 执行相关的操作

//3.1: 执行查询

Get get = new Get("9990611".getBytes());

Result result = table.get(get);

//3.2: 处理结果集

List<Cell> cellList = result.listCells(); // 获取 这一行所有的单元格

for (Cell cell : cellList) { // 单元格: rowkey + 列族 + 列名 + 列值

byte[] rowkeyBytes = CellUtil.cloneRow(cell);

byte[] familyBytes = CellUtil.cloneFamily(cell);

byte[] qualifierBytes = CellUtil.cloneQualifier(cell);

byte[] valueBytes = CellUtil.cloneValue(cell);

String rowkey = Bytes.toString(rowkeyBytes);

String family = Bytes.toString(familyBytes);

String qualifier = Bytes.toString(qualifierBytes);

if(qualifier.equals("NUM\_CURRENT") || qualifier.equals("NUM\_PREVIOUS") ||qualifier.equals("TOTAL\_MONEY") || qualifier.equals("NUM\_USAGE") ){

Double value2 = Bytes.toDouble(valueBytes);

System.out.println("rowkey为:"+rowkey +"; family:"+family +"; qualifier:"+qualifier+"; value:"+value2);

}else{

String value1 = Bytes.toString(valueBytes);

System.out.println("rowkey为:"+rowkey +"; family:"+family +"; qualifier:"+qualifier+"; value:"+value1);

}

}

}

## 4. hbase的读取数据的流程:

流程步骤:

1) 由客户端发起读取数据的请求, 先去连接zookeeper, 主要目的是用于获取hbase:meta表 所在的那个regionServer上

2) 连接meta对应的regionServer, 从meta表获取 对应目标表有那些region, 以及这些region被那些regionServer所管理

region1: node1

region2: node2

region3: node1

注意:

如果采用scan 此时一般要返回整个表每个region对应regionServer的地址

如果采用get 此时返回对应要读取rowkey所在region的regionServer的地址

3) 并发的连接对应regionServer, 开始进行读取数据操作, 首先先从memstore开始读取, 读取完成后接着到blockcache中读取

然后 再到 storeFile中读取, 最后再到大Hfile中读取数据

注意: 在此处读取的时候, 如果命令中存在过滤器 此时就会将过滤条件分发到各个region上, 进行过滤读取操作

4) 每个region可能都会有一定的数据返回, 将各个region读到数据都返回给客户端, 客户端对数据进行排序展示即可

hbase:meta表: 特殊的系统表, 此表只会有一个region 主要用于存储 hbase中各个表的元数据信息

元数据信息包含:

表对应有那些region, 每个region的范围是什么, 每个region被那些regionServer所管理 ....

## 5. 数据写入的流程:

客户端的流程:

1. 客户端发起写入数据的请求, 首先会先连接zookeeper, 从zookeeper

获取 hbase:meta表对应的regionServer的地址

2. 连接meta表对应的regionServer, 从meta表获取对应目标表有那些region, 以及从这些region中根据插入的rowkey确认要插入到那个region中, 以及这个region的regionServer

底层: 根据每个region的 region范围来确定

3. 连接对应的regionServer, 开始准备写入数据:

4. 首先先将数据写入这个regionServer的HLog的目录下, 当这个目录写入成功后,

然后将数据写入到对应的region的对应的memstore中

-----------当以上两个地方都插入成功后, 客户端写入流程就结束了-----------

5. 随着客户端不断的写入, 那么memstore中数据会变得越来越多了, 当memstore中数据达到一定的阈值(128M|1个小时)后, 就会启动flush刷新机制, 将这份数据最终刷新到磁盘上形成一个storeFile的文件

6) 随着不断的写入, 不断的刷新, 在磁盘(HDFS)上就会形成多个storeFile的文件,

当这个storeFile的文件达到一定的阈值(3个及以上 )后, 开始启动compact机制, 将多个storeFile的文件最终合并为一个大Hfile文件

7. 随着compact不断执行, 大的Hfile文件会越来越大, 当这个Hfile达到一定的阈值(10GB)后, 开始启动split机制, 对大的Hfile进行一分为二的操作, 将一个

Hfile形成两个新的Hfile, 对应的region, 也会被分成两个region, 每个

region管理其中一个Hfile的文件

系统: 在前期这两个region可能会同属于一个regionServer

## 6. hbase的批量装载操作:

Bulk load的流程主要分为两步：

1.通过MapReduce准备好数据文件（Store Files） : 参考代码即可

2.加载数据文件到HBase

格式:

hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles hfile的目录位置(HDFS位置) 加载到那个表中

示例

hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles /bulkLoad/output BANK:TRANSFER\_RECORD

## 7: hbase的协处理器:

两类协处理器:

observer协处理器: 类似于 Java中 拦截器(过滤器) 或者 类似于 数据库 触发器

在obServer协处理器上, 提供了各种监听的事件(钩子): 例如 监听 添加数据 删除数据, 创建表 删除表

在2.0接口中, 提供四大类obServer处理器:

RegionObserver：提供客户端的数据操纵事件钩子： Get、 Put、 Delete、 Scan 等

WALObserver：提供 WAL 相关操作钩子。

MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。

到 0.96 版本又新增一个 RegionServerObserver

有什么作用呢?

二级索引

记录日志

权限管理

....

endPoint协处理器: 类似于 数据库中存储过程 理解为 在Java中 封装方法的过程

过程:

通过endpoint协处理器, 将一段聚合代码封装在一起, 将这个代码从客户端发送到服务端, 有各个服务端来执行此操作,

各个服务端执行完成后, 将结果返回给客户端, 客户端在针对这个结果 作进行计算 就OK了....

主要应用场景:

聚集计算 : 求和 求最大 求最小 ....

如何将协处理器加载到hbase中:

静态配置: hbase-site.xml

<property>

<name>hbase.coprocessor.user.region.classes</name>

<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>

</property>

注意: 此种配置 是全局配置, 配置之后, 整个hbase 所有表 都具有这个协处理器

动态配置:

disable 'mytable'

alter 'mytable', METHOD => 'table\_att','coprocessor'=>

'|org.apache.Hadoop.hbase.coprocessor.AggregateImplementation|编号|'

enable 'mytable'

注意: 此种配置, 只针对当前表生效

卸载操作:

disable ‘test’

alter ‘test’, METHOD => ‘table\_att\_unset’, NAME => ‘coprocessor$1’

enable ‘test’

8: hbase的事务: hbase本身nosql型的数据库, 并不支持多行事务操作 仅支持单行事务, 仅仅可以保证在操作一行数据要不全部都成功 要不全部都失败

## 9. 常见的树结构:

跳表: 对列表建立多层结构, 实现稀疏索引, 从而加快检索速度

查询的时候: 是跳着查询

二叉查找树:

1)节点的左侧的数据比节点要小, 节点右侧的数据比节点大

2) 每个节点下只能有二个子节点

二叉查找树关键词:

高度 : 指定从下往上

深度 : 从上往下

层度 : 根节点为第一层，往下依次递增

查询的方式:

首先和根节点进行比较，如果等于根节点，则返回

如果小于根节点，则在根节点的左子树进行查找

如果大于根节点，则在根节点的右子树进行查找

存在的弊端:

由于二叉查找树每个节点下最多只能有二个节点, 如果数据量比较大, 会导致

树的深度 变多了, 从而导致查询的性能

有可能形成链表结构

平衡二叉查找树 :

特点:

任意节点左右两个子树的高度差绝对值不超过1

优点:

平衡二叉树很好地解决了二叉查找树退化成链表的问题

弊端: 写入效率比较差, 读取效率还可以

红黑树:

特点:

是一种弱平衡的二叉查找树, 任意一结点到每个叶子结点的路径都包含数量相同的黑结点

B树: 平衡多路搜索树

B树的节点可以有多个子节点，不限于最多两个节点

B树的查询方式:

从根节点的关键字开始比较，例如：上图为13，判断大于还是小于

继续往下查找，因为节点可能会有多个节点，所以需要判断属于哪个区间

不断往下查找，直到找到为止或者没有找到返回Null

弊端: 当获取树中所有数据, 整体查询效率比较慢, 经历的查询路径太长了

B+树: 平衡多路搜索树 是对B树一种升级版本

只有叶子节点包含数据（所有数据都是在叶子节点中出现）, 叶子也叶子也有关联

由于B+树数据都存储在叶子节点中,叶子节点指点指针关联, 在遍历所有的数据的时候, 直接到叶子依次来获取即可

B+ 数通常使用文件系统和数据库中