Chaoyu Li
Update: 5/8/2022

1. 课前**Big Data**视频问题

- **Where does Big Data come from?**
    1. **Retailers databases**
    2. **Logistics, financial services, healthcare data.**
    3. **Public social media**
    4. **Vision recognition**
    5. **Internet of Things**
    6. **New forms of scientific data**

- **What characteristics does Big Data have?**
    - **Volume, Velocity, Variety**

- **What big data technologies were mentioned?**
    1. **Hadoop**
    2. **Cloud-based big data solution(AWS)**
    3. **Quantum computing**

2. Firebase

a. CURD

- C – create PUT/POST
- R – retrieve/read GET
- U – update PATCH
- D – delete DELETE

b. JSON (Javascript Object Notation)

- value = string|number|object|array|true|false|null
- object = {} | { members }
    - members = pair | pair, members
    - pair = string : value
- array = [] | [elements]
    - elements = value | value, elements

- Example

Value	Valid?	Reason
{[25]}	No	You have the key but no value in JSON object
[25, {}, null]	Yes	
“name” : ”john”	No	Should be inside {}
[“name” : 25]	No	Should be either {“name”: 25} or [“name”, 25]
{“name” : []}	Yes	
{“address”: {“city”: “LA”}}	Yes	
{“25: 26”}	No	Should be [“25: 26”] or {“25”: “26”}
["foo", {"bar": ["baz", null, 1.0, 2]}]	Yes	
[25, False]	No	“Flase” should be “false”
{‘25’: ‘mary’}	No	Should be {“25”: “mary”}
True	No	Only “true” is acceptable
true	Yes	
TRUE	No	
Null	No	Only “null” is acceptable
null	Yes	
false	Yes	

c. Firebase REST API (all commands are sensitive(uppercases))

- **GET**
  - **curl URL or curl -X GET URL**
  - **Example:**

curl -X GET 'https://inf551-1b578.firebaseio.com/examples/phoneNumbers/0.json'

– {"number":"212 555-1234","type":"home"}

Note: refer to array element by index

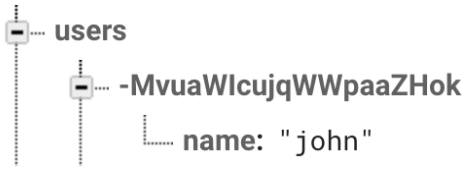
- **PUT (Overwrite if node already has value)**
  - **curl -X PUT URL**
  - **Example:**

1. **curl -X PUT** 'https://inf551-1b578.firebaseio.com/users/100.json' -d '{"name": "john"}'
2. **curl -X PUT** -d '{"100": {"name": "John"}}' 'https://inf551-1b578.firebaseio.com/users.json'

这两个command**的区别在于假设**users node**下面已经存在一些**values**，**command#1**只会覆盖“100”这个**child node**；而**command#2**会覆盖掉所有**users**下的**child node**只留下一个新加的“100”**

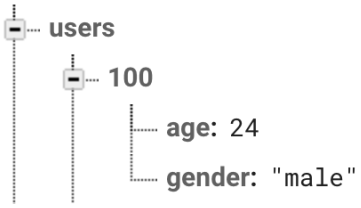
- **POST (Automatically generates a new key then stores the value)**
  - **curl -X POST URL**
  - **Example:**

    curl -X POST -d '{"name": "John"}' 'https://inf551-1b578.firebaseio.com/users.json'

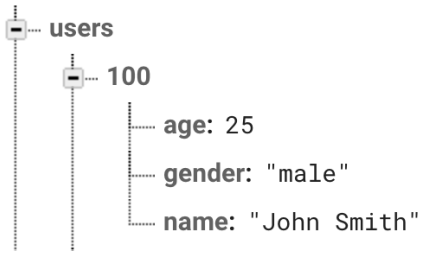


- **PATCH (Update the value(Overwrite the existing value))**

- **curl -X PATCH URL**
- **Example:**



curl -X PATCH -d '{"name": "John Smith","age": 25}' 'https://inf551-1b578.firebaseio.com/users/100.json'



- **DELETE**
  - **curl -X DELETE URL**

d. Firebase Filtering Data

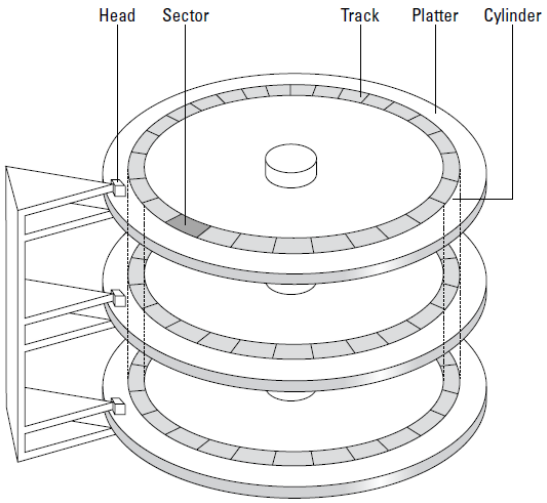
- **Ways (Order by Key; Order by Child Key; Order by Value)**
  - **orderBy="$key"; orderBy = "address/city"; orderBy="$value"**
- **Parameters**
  - **startAt="string"** =>
  - **endAt="string"** <=
  - **equalTo="string"** ==
  - **limitToFirst=num** 符合条件的前**num**项
  - **limitToLast=num** 符合条件的后**num**项

★ 参考资料

https://firebase.google.com/docs/database/rest/start?hl=zh-cn

3. Storage Systems

a. 磁盘基本单位



- **Head**:读写磁盘的磁头，整个硬盘的所有磁头位置是固定在一起的，移动的动作是基于所有磁头的。针对双面Disc，会有正反两个磁头对应。
- **Track**:每个磁盘上的多个同心圆。
- **Cylinder**:多个磁盘上（正反两面）相同同心圆组成的柱状，用于描述多个磁盘上相同Track位置的数据集。
- **Sector**:每个Track可以进一步划分为多个小的区域，定义磁头一次读写的数据量，一般为512 byte或4K byte。
- **Capacity = Cylinders × Heads × Sectors × sector_size**

b. 可能出现的计算题

1. 计算Completion Time和Actual Bandwidth

a. Sequentially

- **Completion Time = $T_{seek} + T_{rotation} + T_{transfer}$**

  T_{seek} = ⅓ * Maximun Seek Time

  $T_{rotation}$ = ½ * Maximum Rotation Time = $\frac{60000ms}{Rotational\ speed\ (RPM)}$

  $T_{transfer}$ = $\frac{Size\ of\ sector}{(Maximum)\ bandwidth}$**(通常size是4kb) * number of blocks**

- **Actual Bandwidth =** $\frac{number\ of\ blocks * size\ of\ sector * number\ of\ sector\ per\ block}{Completion\ Time}$

  $= \frac{传输的文件大小}{Completion\ Time}$

b. Randomly

- **Completion Time = ($T_{seek} + T_{rotation} + T_{transfer}$) * number of blocks**

  T_{seek} = ⅓ * Maximun Seek Time

  $T_{rotation}$ = ½ * Maximum Rotation Time = $\frac{60000ms}{Rotational\ speed\ (RPM)}$

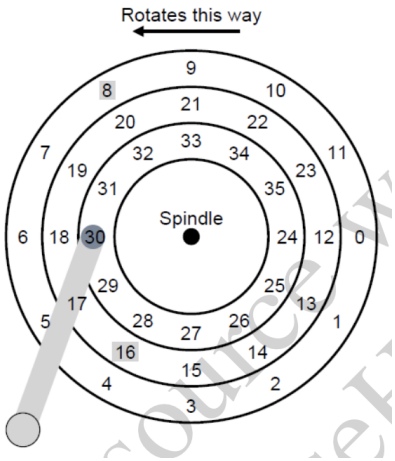
  $T_{transfer}$ = $\frac{Size\ of\ sector}{(Maximum)\ bandwidth}$**(通常size是4kb)**

- **Actual Bandwidth =** $\frac{number\ of\ blocks * size\ of\ sector * number\ of\ sector\ per\ block}{Completion\ Time}$

  $= \frac{传输的文件大小}{Completion\ Time}$

★ 区别：**Randomly**情况下通常$T_{rotation}$占花费时间的主导地位，**Sequentially**情况下通常$T_{transfer}$占花费时间的主导地位。

2. 计算在多**track**下的到某个**sector**的时间
   - **Example:** 计算转到16和8的时间(假设换一个track时间为1ms, 6000RPM)



- 到16: **Time = $T_{seek} + T_{rotation}$ = 1ms +** $\frac{10}{12}$***** $\frac{60000ms}{6000RPM}$ **= 9.33ms**
- 到8: **Time = $T_{seek} + T_{rotation}$ = 2ms +** $\frac{2}{12}$***** $\frac{60000ms}{6000RPM}$ **= 3.67ms**

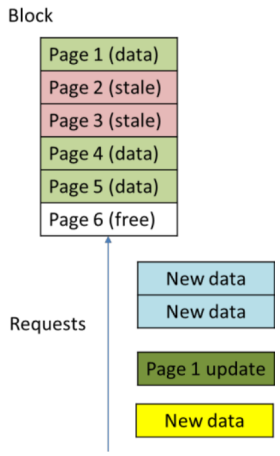
c. SSD

- **Writing to SSD is complicated (cannot overwrite, erase instead)**
- **(one)Chip -> (1,2,4)dies -> (1,2)planes -> (many)blocks -> (many)pages (rows) -> (many)cells**
- **Page is the smallest unit of data transfer between SSD and main memory(like block in hard disk)**

- **Page is the smallest unit that can be read, programmed/written**
- **Block is the smallest unit that can be erased**
- 时间比较（单**(s)**、双**(m)**、三**(t)**层存储单元）

	SLC	MLC	TLC	HDD	RAM
P/E cycles	100k	10k	5k	*	*
Bits per cell	1	2	3	*	*
Seek latency (µs)	*	*	*	9000	*
Read latency (µs)	25	50	100	2000-7000	0.04-0.1
Write latency (µs)	250	900	1500	2000-7000	0.04-0.1
Erase latency (µs)	1500	3000	5000	*	*
Notes	* metric is not applicable for that type of memory				
Sources	P/E cycles [20] SLC/MLC latencies [1] TLC latencies [23] Hard disk drive latencies [18, 19, 25] RAM latencies [30, 52] L1 and L2 cache latencies [52]				

- **Example:** 计多个requests下的read, write, erase次数。



➢ 第一个request: 插入两个new data

1. Read 6 pages, rearrange page order and remove stale pages
2. Erase all pages

3. Write 5 new pages

➢ 第二个request: 更新page1

1. Write new page1
2. Mark original page1 as stale

➢ 第三个request: 插入一个new data

1. Read 6 pages, rearrange page order and remove stale pages
2. Erase all pages
3. Write 6 new pages

➢ 因此，一共12次read，12次write，2次erase

4. File Systems

a. “stat” command

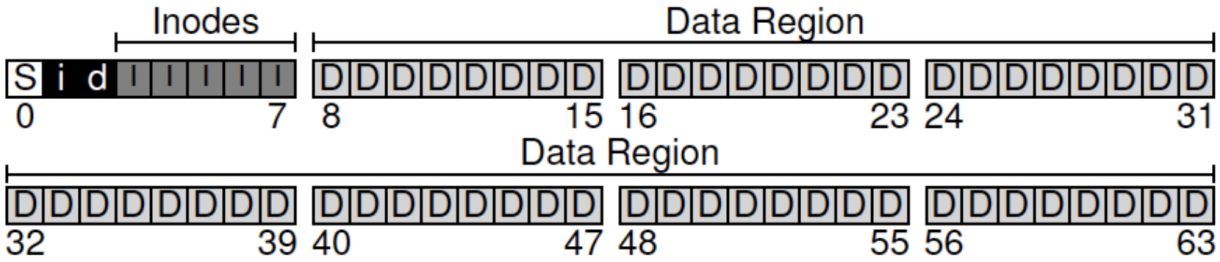
```
[root@TENCENT64 ~]# stat Changelog
  File: ‘Changelog’
  Size: 1598        Blocks: 8          IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 1579435     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-11-06 22:39:54.110931887 +0800
Modify: 2018-11-06 22:39:54.110931887 +0800
Change: 2018-11-06 23:07:14.428548887 +0800
```

Output	**Content**	存储内容
File: ‘Changelog’	File name	文件名称为Changelog
Size: 1598	The size of the file in bytes	文件大小1598字节
Blocks: 8	The number of blocks allocated to this file	文件占用的块数
IO Block: 4096	The preferred I/O block size	每个块的大小（字节）
regular file	File type	文件类型（普通文件）
Device: fd01h/64769d	Device ID	文件所在设备号，分别以十六进制和十进制显示
Inode: 1579435	File serial numbers	文件节点号
Links: 1	A link count telling how many hard links point to the	硬链接数

	inode	
Access: (0644/-rw-r--r--)	The file mode which determines the file type and how the file's owner, its group, and others can access the file.	访问权限
Uid	The User ID of the file's owner	所有者ID与名称
Gid	The Group ID of the file	所有者用户组ID与名称
Access	Timestamps telling when the last accessed (atime, access time)	最后访问时间
Modify	Timestamps telling when the file content last modified (mtime, modification time)	最后修改时间
Change	Timestamps telling when the inode itself was last modified (ctime, inode change time)	最后状态改变时间（chmod 可以改变状态）

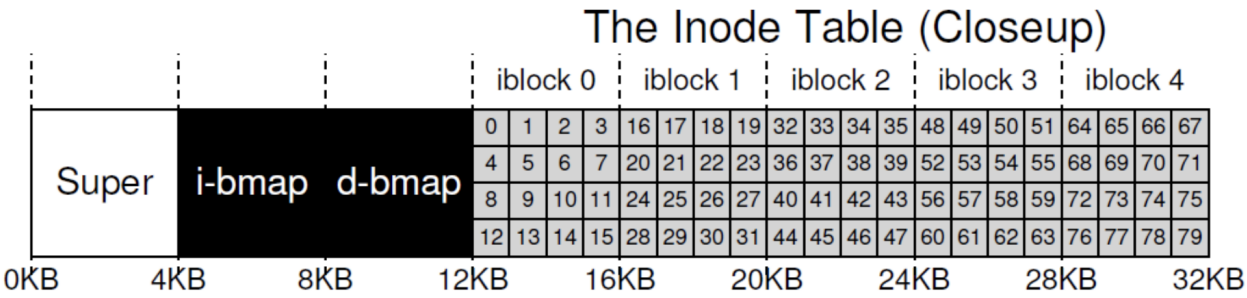
b. inode & inumber

1. inode



- Stores metadata/attributes about the file; Also stores locations of blocks holding the content of the file.
- 如上图，inode占5个block，假设一个inode的大小为256B，则总共有5*4KB/256B=80个inode

2. inumber



- Can figure out location of inode from inumber.
- 如上图，假设inumber=32，对应的inode起始位置（也叫offset）就是20KB

3. inode bitmap & data bitmap

- inode bitmap: keep track of which inodes in the inode table are available(用于track哪个inode是空的哪个有内容)
- data bitmap: Keep track of which blocks in data region are available

4. 计算题**example:** 假设**inode**大小**512B, sector**大小**512B, block**大小**4KB,** 一共有**56**个**datanode**和**5**个**inode.**

a. 求最多能存多少文件？

求最多能存多少文件实际上求的就是最多有多少inode。

Number of inode = $\frac{number\ of\ inode\ block * block\ size}{inode\ size} = \frac{5*4KB}{512B} = 40$

b. 求inode bitmap和data bitmap最多有多少bits？

40个inode就需要40bits, 所以**inode bitmap=40bits(5bytes)**

data bitmap track的是block, 所以**56**个**block**就是**56bits(7bytes)**

c. 求inumber=24时的offset，存在哪个sector里，哪个block里？

前三个block是superblock, inode bitmap, data bitmap,所以初始offset=12KB。

总**offset** = 12KB+23*512B=24KB

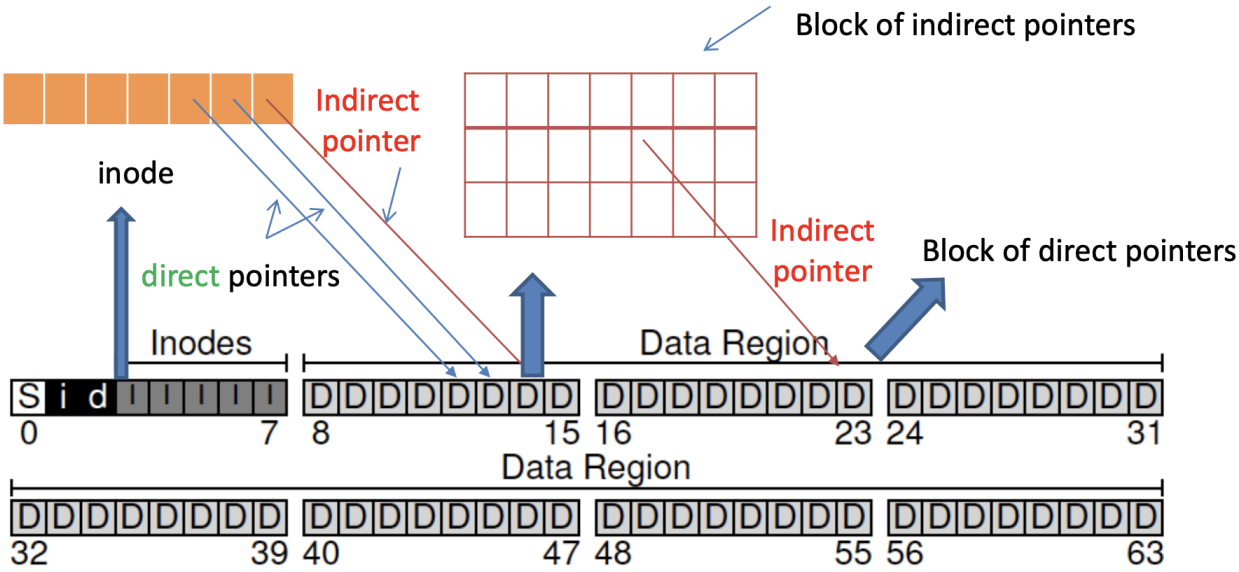
Sector number = $\frac{总offset}{sector\ size} = \frac{24KB}{512B} = 48$(有时候得到的不是整数需要向下取整)

Block number = $\frac{Sector\ number}{\frac{block\ size}{sector\ size}} = \frac{48}{\frac{4KB}{512B}} = 6$

d. 求这个系统能存下的最大文件大小？

Maximum file size = 4KB*56 = 224KB

5. direct pointer & indirect pointer



- 背景:每个block只能存4KB文件，一个inode存的是一个文件的信息。那么一个文件要是很大怎么办？就得分多个data block去存。
- 假设一个inode可以存8个pointer(pointer的作用就是指向对应的data block)，**direct pointer**就是指这个pointer直接指向存放文件内容的block。那么我们现在可以存一共8*4KB=32KB大小的文件。还是不够怎么办？
- 我们把8个pointer中的一个空间拿来变成**indirect pointer**。顾名思义indirect pointer就是指向另一个存放direct pointer的block。这样我们又多了整整一个block的空间来表达指向信息！(注意这里存放direct pointer的block是**data block**)
- 要是这么做还不够大怎么办？嵌套！将indirect pointer指向的block里direct pointer改成indirect pointer再指向一个新的存放direct pointer的block。这叫做**double indirect pointer**。
- 举一反三，有double自然就会有**triple indirect pointers**。

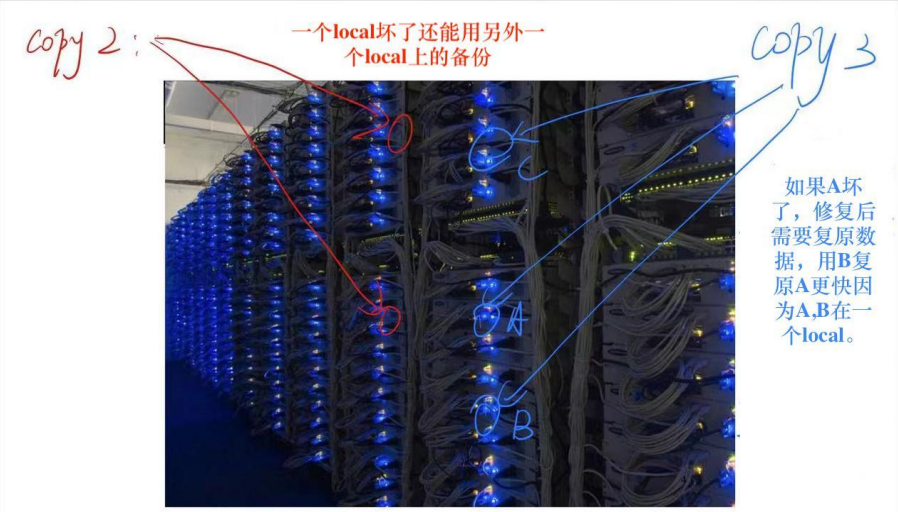
★ 参考资料

https://www.youtube.com/watch?v=ymYZPtrvgec

5. HDFS

a. HDFS architecture

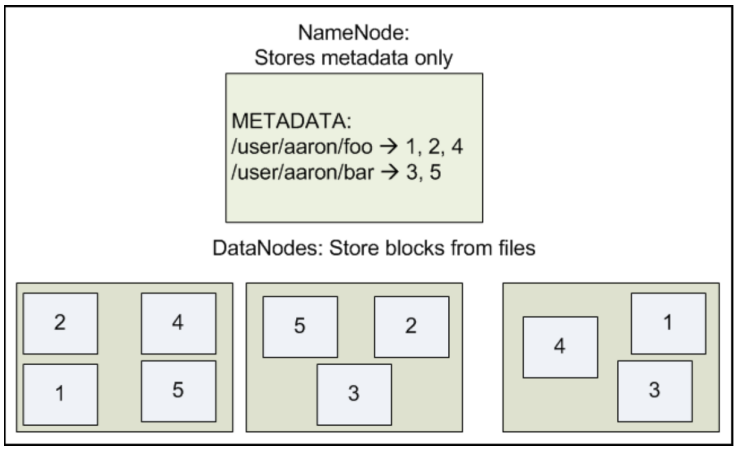
1. **Data are replicated to cope with node failure(replication factor: 2 or 3)**



2. **HDFS has a single NameNode and a number of DataNode**
3. **HDFS also has a SecondaryNameNode(Maintaining checkpoints/images of NameNode) for recovery**
4. **Block Size: 128MB(much BIGGER than disk block 4KB)**

- **Why larger size in HDFS?**
  - Reduce metadata required per file
  - Fast streaming read of data (since larger amount of data are sequentially laid out on disk)

b. NameNode & DataNode



1. **NameNode has an inode for each file and dir. It only stores metadata.**
2. 上图中的**replication factor=2**

c. Reading and Writing in HDFS

1. **Phase #1: Client asks NameNode for block locations**

- **Reading: Calling (sending request) *getBlockLocations()***
  - **Input:** (File name, Offset (to start reading), Length (how much data to be read))
  - **Output:** 一个list包含所有client请求的有效data node及其offset**(包括其replication的信息也会被返回)**。

★ 参考资料：http://shiyanjun.cn/archives/925.html

- **Writing: Calling addBlock() for allocating new blocks (one at a time), (need to call create()/append() first)**
- 课上问题**:** addBlock()和getBlockLocations()都会访问locatedBlock()返回一个包含blocks地址的list，他们有什么区别？

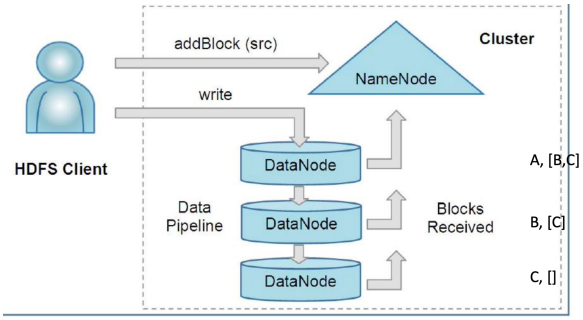
  A: addBlock()得到的list里的所有block都是空的，而getBlockLocations()得到的所有block都是有数据的。

2. **Phase #2: Client talks to DataNode for data transfer**

- **Reading blocks via readBlock() or writing blocks via writeBlock()**

★ 所有**writing**相关的函数调用次数需要***Replication Factor**

★ **Writing**详细过程：https://www.daimajiaoliu.com/daima/4870f9c4b9003e4



a. 调用**create()**生成一个新的**HDFS**文件空间。

i. 调用addBlock()访问新空间的NameNode为即将写入的文件分配block地址(返回的是一个包含空白的DataNode的地址的list)

OR调用**append()**打开一个已存在的文件。

i. 如果此文件的最后一个数据块没有被写满，就直接写在后面。

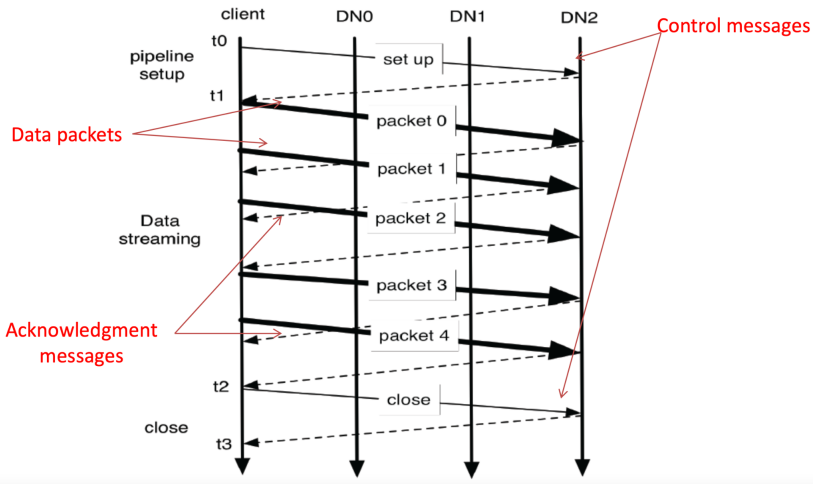
ii. 如果最后一个数据块已经写满了，就调用addBlock()。

b. 调用**writeBlock()**写入文件。

i. 假设Replication Factor=3，写入一个block X时addblock()会返回三个DataNode的地址，这里假设为A,B,C。

ii. 将X分成若干个packets(假设X是满的也就是128MB)，一个packet的大小是**64KB**，因此一共会有2048个packets(编号为0～2047)。

iii. 对于每一个packet，Client会先发给A，A收到后马上发给B，B收到后马上发给C。C也收到以后会返回一个信号给Client(但是！**Client**不会等待收到**C**的信号才发送下一个**packet**给**A**)



6. XML & XPath

a. 基本语法(来源https://www.runoob.com/xpath/xpath-syntax.html)

选取节点

XPath 使用路径表达式在 XML 文档中选取节点。节点是通过沿着路径或者 step 来选取的。 下面列出了最有用的路径表达式：

表达式	描述
nodename	选取此节点的所有子节点。
/	从根节点选取（取子节点）。
//	从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置（取子孙节点）。
.	选取当前节点。
..	选取当前节点的父节点。
@	选取属性。

在下面的表格中，我们已列出了一些路径表达式以及表达式的结果：

路径表达式	结果
bookstore	选取 bookstore 元素的所有子节点。
/bookstore	选取根元素 bookstore。 注释：假如路径起始于正斜杠(/)，则此路径始终代表到某元素的绝对路径！
bookstore/book	选取属于 bookstore 的子元素的所有 book 元素。
//book	选取所有 book 子元素，而不管它们在文档中的位置。
bookstore//book	选择属于 bookstore 元素的后代的所有 book 元素，而不管它们位于 bookstore 之下的什么位置。
//@lang	选取名为 lang 的所有属性。

谓语（Predicates）

谓语用来查找某个特定的节点或者包含某个指定的值的节点。

谓语被嵌在方括号中。

在下面的表格中，我们列出了带有谓语的一些路径表达式，以及表达式的结果：

路径表达式	结果
/bookstore/book[1]	选取属于 bookstore 子元素的第一个 book 元素。
/bookstore/book[last()]	选取属于 bookstore 子元素的最后一个 book 元素。
/bookstore/book[last()-1]	选取属于 bookstore 子元素的倒数第二个 book 元素。
/bookstore/book[position()<3]	选取最前面的两个属于 bookstore 元素的子元素的 book 元素。
//title[@lang]	选取所有拥有名为 lang 的属性的 title 元素。
//title[@lang='eng']	选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。
/bookstore/book[price>35.00]	选取 bookstore 元素的所有 book 元素，且其中的 price 元素的值须大于 35.00。
/bookstore/book[price>35.00]//title	选取 bookstore 元素中的 book 元素的所有 title 元素，且其中的 price 元素的值须大于 35.00。

注意：XPath中的index是从**1**开始的。

选取未知节点

XPath 通配符可用来选取未知的 XML 元素。

通配符	描述
*	匹配任何元素节点。
@*	匹配任何属性节点。
node()	匹配任何类型的节点。

在下面的表格中，我们列出了一些路径表达式，以及这些表达式的结果：

路径表达式	结果
/bookstore/*	选取 bookstore 元素的所有子元素。
//*	选取文档中的所有元素。
//title[@*]	选取所有带有属性的 title 元素。

选取若干路径

通过在路径表达式中使用"|"运算符，您可以选取若干个路径。

在下面的表格中，我们列出了一些路径表达式，以及这些表达式的结果：

路径表达式	结果
//book/title | //book/price	选取 book 元素的所有 title 和 price 元素。
//title | //price	选取文档中的所有 title 和 price 元素。
/bookstore/book/title | //price	选取属于 bookstore 元素的 book 元素的所有 title 元素，以及文档中所有的 price 元素。

b. Example:(注意如果题目只要求value,需要在最后加上 */text()*)

1. [6 points] Consider an XML document containing a catalog of CD's as shown on the right. Write an XPath expression for each of the following questions.

   a. Find all CDs (elements) not released in "UK".

   /catalog/cd[country != 'UK']

   b. Find the titles of all CDs with rating <= 3.

   /catalog/cd[@rating <= 3]/title

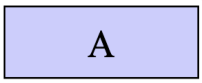
   c. Find all CDs (elements) by artists whose name contains "Bob".

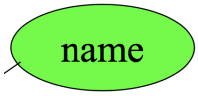
   /catalog/cd[contains(artist,'Bob')]

```
▼<CATALOG>
  ▼<CD rating="3">
     <TITLE>Empire Burlesque</TITLE>
     <ARTIST>Bob Dylan</ARTIST>
     <COUNTRY>USA</COUNTRY>
     <COMPANY>Columbia</COMPANY>
     <PRICE>10.90</PRICE>
     <YEAR>1985</YEAR>
   </CD>
  ▼<CD rating="2">
     <TITLE>Hide your heart</TITLE>
     <ARTIST>Bonnie Tyler</ARTIST>
     <COUNTRY>UK</COUNTRY>
     <COMPANY>CBS Records</COMPANY>
     <PRICE>9.90</PRICE>
     <YEAR>1988</YEAR>
   </CD>
```

7. ER and relational data models

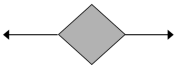
a. Basic Stuff

1. **Entities(Entity set: a collection of similar entities)** 

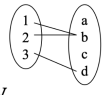
2. **Attributes(Necessary in Entities, Not necessary in Relationships)** 

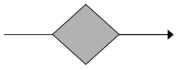
3. **Relationships: Multiplicity of E/R Relationships**

- one-one:
  - One = at most one

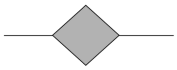


- many-one/one-many
  - Here left side = many





- many-many



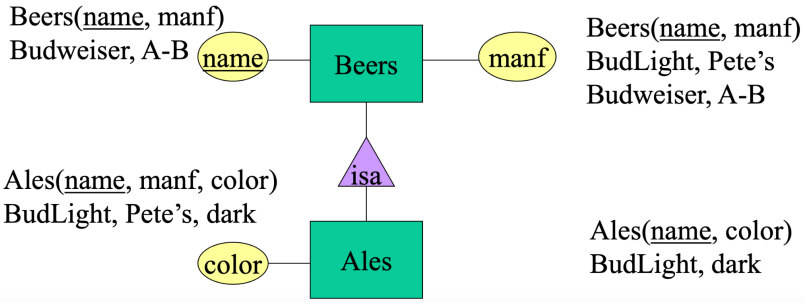
b. Subclasses(is-a) 

1. **Object Oriented(继承superclass的所有attributes)**, 省时间
2. **ER(只继承superclass的key attributes)**, 省空间

Example

OO:
Think about Java:
Ales a = new Ales(name, manf, color)

ER



3. **The Null Value(所有的class都装在一个table里)**, 会有很多**null**

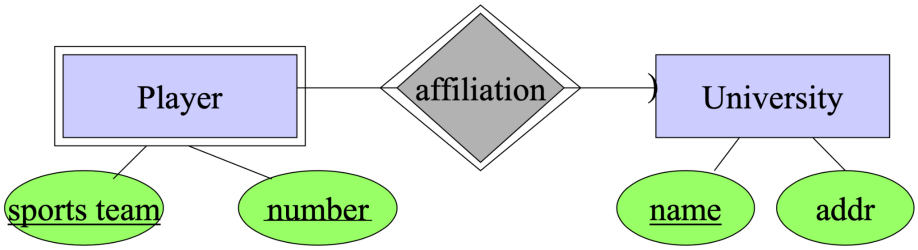
- **Comparisons:**
  - OO approach good for queries like "find the color of ales made by Pete's."

- E/R approach good for queries like "find all beers (including ales) made by Pete's."
- Using nulls might waste space if there are *lots of* attributes that are usually null.

c. Constraints

1. **Keys:** e.g. social security number uniquely identifies a person.
   - 每个enetities必须至少要有一个key attribute(来区分不同个体)
   - 每个subclass必须继承superclass的所有key attributes。
2. **Single-value constraints:** e.g. a person can have only one spouse. create table person(ssn, name ..., age int check(age <= 150))
3. **Referential integrity constraints:** e.g. if you work for a company, it must exist in the database.
4. **Domain constraints:** peoples' ages are between 0 and 150. (check)
5. **General constraints:** all others (e.g., at most 50 students can enroll in a class) // create assertion as SQL query

d. Weak entity set(依赖于其他entity set存在(无法避免自身重复))



e. Three Design principles

1. **Be faithful**
2. **Avoid redundancy (entity set vs attributes)**

   用**entity set**的条件：

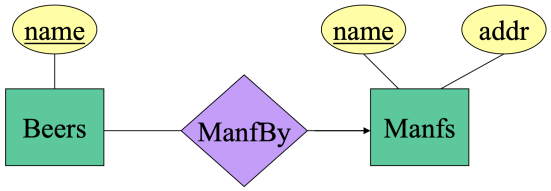
   - It has at least one nonkey attribute.

     OR

   - It is the “many” in a many-one or many-many relationship.

   Why? 因为如果是**“one”**的那一边且只有**key attibutes**，那么这个**entity set**里的所有元素都必定是独立的。也就意味着它可以被归纳为**relationship**另一边的一个

attibute来减少**redundancy**;而如果满足上面两个条件的其中一个，那么他作为**attibute**必定会造成**redundancy(**看例子**)**

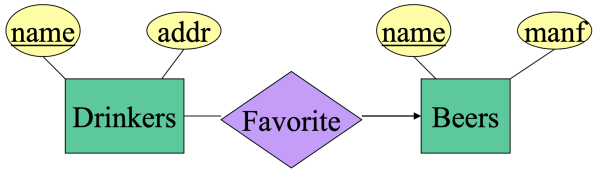


上面的例子里，如果**Manfs**作为**Beers**的一个**attibute**，那么不同的**entities**的**attibutes**里**addr**这个属性是可能重复的，造成了冗余。

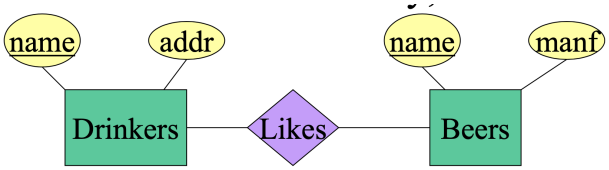
3. **KISS(Keep it Simple)**

f. Combining Relations

- 适用情况: m-1 relationship。
  - Example: combine Drinkers(name, addr) and Favorite(drinker, beer) => Drinkers(name, addr, favoriteBeer).



- 不适用情况: m-n relationship(会出现冗余)。
  - Example:



name	addr	beer
Sally	123 Maple	Bud
Sally	123 Maple	Miller

8. SQL(https://www.runoob.com/sql/sql-syntax.html)

a. 基本语句

语句	解释
SELECT **A** FROM **B**	从B数据集取A属性

SELECT DISTINCT **A**	只取不重复的A属性
WHERE **conditions**	给SELECT增加条件
AND & OR & NOT	在WHERE里串联多条件
LIKE	在WHERE里匹配某个模式
A IN **B**	多用于WHERE里筛选A是否在B里
LIMIT & OFFSET	限制输出的个数，LIMIT起始于1，OFFSET起始于0， LIMIT 1, OFFSET 0也可以写成LIMIT 0,1
ORDER BY **A** ASC|DESC	按照A排序 ASC是升序，DESC是降序。默认升序
通配符%, _, [charlist], [^charlist]|[!charlist](通常搭配LIKE使用)	1. %可以替代0个或多个字符，例如"%5"表示以5结尾，"%5%"表示中间有5。 2. _可以替代一个字符，例如"_5"就不能匹配125。 3. [charlist]可以匹配list中任意一个字符，例如^[GFs]就匹配以G或F或s开头的字符串(这里^是正则表达式)。 4. [^charlist]|[!charlist]简单理解就是上面的取反。
A AS its_new_name	在输出的表格里给A一个别名
GROUP BY **A**	根据一个或多个列对输出结果进行分组
HAVING **conditions**	用于在GROUP BY的时候添加条件，跟WHERE的用法基本相同但是HAVING是对分组后的结果进行筛选。
A INNER JOIN **B** ON **conditions**, **A** LEFT JOIN **B** ON **conditions**, **A** RIGHT JOIN **B** ON **conditions**, **A** FULL JOIN **B** ON **conditions**, **A** NATURAL JOIN **B** **(midterm 2不考据说)**	1. INNER JOIN返回A,B两个集合满足条件的行 2. LEFT JOIN返回A的所有行，对于满足条件的行和INNER JOIN一样，否则B的部分为NULL 3. RIGHT JOIN返回B的所有行，对于满足条件的行和INNER JOIN一样，否则A的部分为NULL 4. FULL JOIN对于满足条件的行和INNER JOIN一样，否则也会输出，相当于LEFT+RIGHT
SELECT … FROM … UNION/UNION ALL SELECT … FROM …	用于合并两个或多个 SELECT 语句的结果集。UNION 内部的每个 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每个 SELECT 语句中的列的顺序必须相同。UNION会自动排除重复值，UNION ALL则不会。
EXISTS, NOT EXISTS https://www.w3schools.com/sql/sql_exists.asp	用在WHERE后，后接subquery，EXISTS只要扫描到一条符合条件的记录就返回true(所以相当于自带DISTINCT)。NOT EXISTS要扫描到最后还没有符合条件的才返回true
UPDATE **B** SET **A**=new_value WHERE **conditions**	根据条件改数据集B中的A的值。

DELETE FROM **B** WHERE **conditions**	删除B中符合条件的行
INSERT INTO **B** (colname1, colname2…) VALUES (value1,value2…)	向B中插入列名为(colname1, colname2…)，值为(value1,value2…)的一行

SELECT A FROM (SELECT B FROM C) **as TEMP**;(嵌套的FROM后面需要定义一个别名不然编译不过)

- Example:

```
+------------+------------+-------+
| bar        | beer       | price |
+------------+------------+-------+
| Mary's bar | Bud        |  NULL |
| Mary's bar | Budweiser  |     2 |
| Bob's bar  | Bud        |     3 |
| Bob's bar  | Summerbrew |     3 |
| Joe's bar  | Bud        |     3 |
| Joe's bar  | Bud Lite   |     3 |
| Joe's bar  | Michelob   |     3 |
| Mary's bar | Bud Lite   |     3 |
| Joe's bar  | Summerbrew |     4 |
+------------+------------+-------+
9 rows in set (0.00 sec)

mysql> select beer from Sells where price >= all (select price from Sells);
Empty set (0.00 sec)
```

```
mysql> select beer from Sells where price >= all (select price from Sells where price is not null)
;
+------------+
| beer       |
+------------+
| Summerbrew |
+------------+
```

上面的例子显示了NULL值会影响all()函数的输出。

b. Constraints

1. **NOT NULL** - 指示某列不能存储 NULL 值。

- Example:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

这里ID, LastName, FirstName在插入或更新的时候不能给NULL值。

2. **UNIQUE** - 保证某列的每行必须有唯一的值(可以是NULL)。

- Example:

```
CREATE TABLE Persons (
P_Id int,
```

```
LastName varchar(255) NOT NULL,
UNIQUE (P_Id)
);
```

这里P_Id在插入或更新的时候不能给表格中已经存在的值。

3. **PRIMARY KEY** - NOT NULL 和 UNIQUE 的结合。确保某列(或两个列多个列的结合)有唯一标识，有助于更容易更快速地找到表中的一个特定的记录

4. **FOREIGN KEY** - 保证一个表中的数据匹配另一个表中的值的参照完整性。

- Example:

```
CREATE TABLE Orders(
O_Id int,
P_Id int,
FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
FOREIGN KEY (O_Id) REFERENCES Persons(O_Id) ON DELETE SET NULL ON
UPDATE CASCADE(表示在删除主键时修改O_Id为NULL，在更新主键值时同步更新O_Id的
值)
);
```

(1)匹配的主键不会改变FOREIGN KEY的变量的类型。(在这个例子里P_Id在添加和更新的时候仍然可以是**NULL**或者重复的值。)

(2)插入非空值时，如果主键表中没有这个值，则不能插入。

(3)更新时，不能改为主键表中没有的值。

(4)删除主键表记录时，你可以在建外键时选定外键记录一起级联删除还是拒绝删除，默认拒绝删除。

(5)更新主键记录时，同样有级联更新和拒绝执行的选择,默认拒绝。

```
mysql> create table T (a int primary key, foreign key (a) references R(a) on delete set null );
```

该语句不能执行因为a是primary key,不能为NULL。

5. **CHECK** - 保证列中的值符合指定的条件。

- Example:

```
CREATE TABLE Persons(
P_Id int NOT NULL,
CHECK (P_Id>0)
);
```

这里P_Id在插入或更新的时候不能给≤0的值。

6. **DEFAULT -** 规定没有给列赋值时的默认值。

- Example:

```
CREATE TABLE Persons(
    P_Id int NOT NULL,
    City varchar(255) DEFAULT 'Sandnes'
);
```

这里P_Id在插入的时候如果没有给定City的值就会自动赋值为Sandnes。

c. View

1. 和**Table**的区别：table是真实存有数据的，而**view**只是一个虚拟的表，通常是基于一个真实的table的select后的结果。 另外view通常只是可读的，但是一旦**view**关联到的表发生数值变化，**view**也会跟着变化
2. **View expansion(**又叫**view unfolding):**用view的definition代替它的名字。例子:上面的命令是view name，下面的命令是view expansion，它们是等价的。

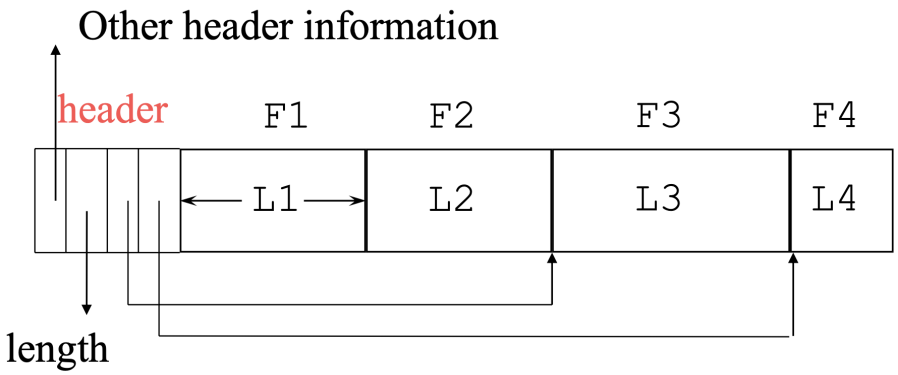
```
mysql> select * from BarCntBeer;
+------------+-----+
| bar        | cnt |
+------------+-----+
| Bob's bar  |   2 |
| Joe's bar  |   4 |
| Mary's bar |   3 |
+------------+-----+
3 rows in set (0.01 sec)

mysql> select * from (select bar, count(*) cnt from Sells group by bar) BarCntBeer;
+------------+-----+
```

3. **Virtual view, materialized view**和**table**的区别**(**这里假设数据都是从其他**table select**来的**)**:

名字	访问速度	更新频率refresh
Virtual view	最慢	Always up-to-date
Materialized view	比较快	可以自己设定更新频率(因此可能存在陈旧数据stale data)
Table	最快	永不随原表更新而更新

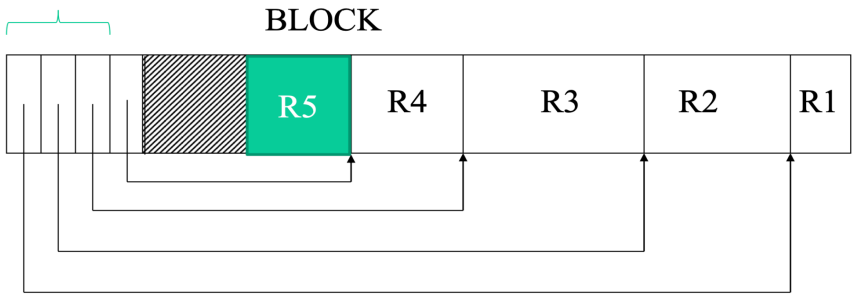
9. Data representation and External sorting

a. Data representation



1. 每个属性attribute被存在一个field中；fields被存在records中(上图表示的就是一个**record**)；records被存在blocks中；blocks被存在files中。(总结：**attributes=fields~records~blocks~files**)
2. 假设F_1, F_2是fixed field，也就是固定长度类型；F_3, F_4是variable length field，也就是可变长类型(e.g. VARCHAR)。**header**用来存储**metadata**。内容包括**record**的位置，**record**的长度，每个**field**的**offset**，**pointers**等等。
3. 指针**(Pointers)**只需要指向**F_4**，因为：
   a. F_1, F_2的长度L_1, L_2可以从header中得到(header中存储了他们的类型，因为是定长类型所以直到类型就知道长度)
   b. 知道了F_1和F_2的长度就知道了F_3的起始位置，所以F_3作为第一个可变长data也不需要指针指向。
4. 总结：所有定长field和第一个可变长field都不需要指针。

Offset table (slot directory)



5. 上图表示了一个block(4KB)存储records的方法。offset table相当于header，存储records的方式是从后往前依次存储。为什么？--> 因为如果是从前往后存

储，在最后加入新的**record**时需要给新的**record**分配指针，但是**header**后面紧接着就是$\mathbf{R_1}$，没有额外的空间新建指针了。

6. R_1不需要头指针因为它的头就是block的尾。
7. Offset table中对每个**record**除了存储它的指针信息还存储**block的位置(两个components合起来叫做record ID)**，因为指针存储的是record相对于block的offset，还得知道block的位置才能访问到record的物理地址。

b. External Sorting(https://zhuanlan.zhihu.com/p/343986766)

1. **Motivation**：我们要排序的数据量比内存还要大，存都存不下怎么排序？
2. 方法：归并排序(外部)，假设可以用的pages为M，数据量为N pages/blocks。
   a. Pass0：将数据分成⌈N/M⌉个比内存小的小块然后分别排序。
   b. Pass1：每次读入M-1个小块(还有一个page要用来存排序结果)，进行归并以后输出给外存。
   c. 重复Pass1直至最后一次输出到外存的一次性数据量跟原数据量一样

★ **Total cost** = $2N(\lceil log_{M-1}\frac{N}{M}\rceil + 1)$

3. 举例：Consider external-sorting a table R with **100 blocks**, using **4 pages** of memory.
   a. For each pass (sorting and merging), state the number of runs and the size of runs generated by the pass.

      A: Pass 0: Sort into 100/4=25 sorted runs of 4 blocks per run(4b/run).

      Pass 1: Merge 3 runs into 1 run each time. After merging 24 runs into 8 runs, only one run is left so we keep it. Output: 8 runs of 12 blocks per run and 1 run of 4 blocks per run.

      Pass 2: Merge 3 runs into 1 run each time. Output: 2 runs of 36 blocks per run and 1 run of 28 blocks per run.

      Pass 3: Merge 3 runs into 1 run. Output: 1 run of 100 blocks per run. The task is finished.

   b. What is the total cost (measured by the number of block I/O's) of this external-sorting?

A: 4 passes in total: **Total cost= 2*100 * (**$\lceil log_{4-1} \frac{100}{4} \rceil$**+1) = 2*100*4=800 blocks of I/O**. 或者 **Total cost = (#ofpasses)× 2× (#ofblocks)=4*2*100=800 blocks of I/O.**

##Or 792 blocks of I/O if we don't read and output the last run of length 4 in pass 1.

10. MongoDB

a. 基本语句

1. 和mySQL对比：https://www.mongodb.com/docs/v4.4/reference/sql-comparison/
2. db.xxx.update({multi: true}) = db.xxx.updateMany()
3. json格式找某个attribute的子attribute：

```
> db.person.find({"address.city": "LA"})
{ "_id" : ObjectId("624cce221c4a58745b1a7777"), "name" : "bill smith", "address" : { "city" : "LA"
, "state" : "CA" } }
```

4. scores这里是一个list，$in: [3,5]表示成绩里由3或者5的；$all: [3,5]表示成绩里有3和5的。

```
> db.person.find({scores: {$in: [3, 5]}})
{ "_id" : 1, "name" : "mary", "scores" : [ 2, 3, 5, 4, 2 ] }
{ "_id" : 4, "name" : "jennifer smith", "scores" : [ 3, 8, 9 ], "age" : 26 }
> db.person.find({scores: {$all: [3, 5]}})
{ "_id" : 1, "name" : "mary", "scores" : [ 2, 3, 5, 4, 2 ] }
```

5. db.person.find({"scores": {**$elemMatch**: {"midterm": "B", "score": {$gt: 90}}}}) 和db.person.find({"scores.midterm":"B", "scores.score": {$gt: 90}})的区别：后面的语句假设一个人的**final>90**也会输出，但是前面的一语句就不会。
6. Projection：**Can not mix 1 and 0 conditions (unless it is "_id")**

```
> db.person.find({age: {$eq: 25}}, {age: 1, _id: 0})
{ "age" : 25 }
{ "age" : 25 }
{ "age" : 25 }
> db.person.find({age: {$eq: 25}}, {age: 1, name: 0, _id: 0})
Error: error: {
        "ok" : 0,
        "errmsg" : "Cannot do exclusion on field name in inclusion projection",
        "code" : 31254,
        "codeName" : "Location31254"
```

b. 聚合语句(aggregate -> group by)

1. 和mySQL对比：

   https://www.mongodb.com/docs/manual/reference/sql-aggregation-comparison/

2. 例子：

```
> db.product.aggregate({$group: {_id: "$category", max_qty: {$max: "$qty"}}})
{ "_id" : "cell", "max_qty" : 20 }
{ "_id" : "laptop", "max_qty" : 40 }
> db.product.find()
{ "_id" : ObjectId("6223c6b54c5c0ad660391101"), "category" : "cell", "store" : 1, "qty" : 10 }
{ "_id" : ObjectId("6223c6b54c5c0ad660391102"), "category" : "cell", "store" : 2, "qty" : 20 }
{ "_id" : ObjectId("6223c6b54c5c0ad660391103"), "category" : "laptop", "store" : 1, "qty" : 10 }
{ "_id" : ObjectId("6223c6b54c5c0ad660391104"), "category" : "laptop", "store" : 2, "qty" : 30 }
{ "_id" : ObjectId("6223c6b54c5c0ad660391105"), "category" : "laptop", "store" : 2, "qty" : 40 }
> db.product.aggregate({$group: {_id: {cat: "$category", st: "$store"}, max_qty: {$max: "$qty"}}})

{ "_id" : { "cat" : "cell", "st" : 2 }, "max_qty" : 20 }
{ "_id" : { "cat" : "laptop", "st" : 2 }, "max_qty" : 40 }
{ "_id" : { "cat" : "cell", "st" : 1 }, "max_qty" : 10 }
{ "_id" : { "cat" : "laptop", "st" : 1 }, "max_qty" : 10 }
> db.product.aggregate({$group: {_id: "category", max_qty: {$max: "$qty"}}})
{ "_id" : "category", "max_qty" : 40 }
```

3. 更复杂的例子(mysql对照)（图里的_id: “category”）应该是_id: “$category”

```
> select category, count(*) from product where store = 2 group by category
uncaught exception: SyntaxError: unexpected token: identifier :
@(shell):1:7
> db.product.aggregate({$match: {store: 2}}, {$group: {_id: "category", stat: {$sum: 1}}})
{ "_id" : "category", "stat" : 3 }
> select category, count(*) from product where store = 2 group by category having count(*) > 1
uncaught exception: SyntaxError: unexpected token: identifier :
@(shell):1:7
> db.product.aggregate({$match: {store: 2}}, {$group: {_id: "category", stat: {$sum: 1}}},
...  {$match: {stat: {$gt: 1}}})
{ "_id" : "category", "stat" : 3 }
> select category, count(*) cnt from product where store = 2 group by category having count(*) > 1
  order by cnt desc limit 1 offset 1;
uncaught exception: SyntaxError: unexpected token: identifier :
@(shell):1:7
> db.product.aggregate({$match: {store: 2}}, {$group: {_id: "category", stat: {$sum: 1}}},  {$matc
h: {stat: {$gt: 1}}}, {$sort: {stat: -1}}, {$skip: 1}, {$limit: 1})
```

4. **join**合并两个表的例子：**localField**和**foreignField**部分相当于上面**mysql**语句里的**where**。

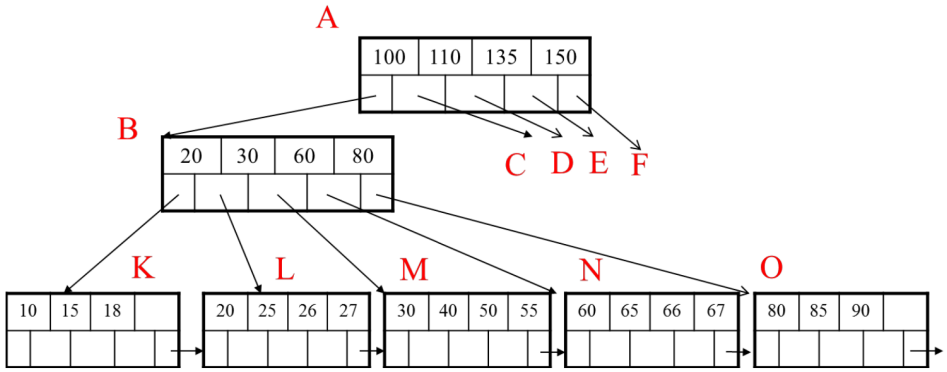
```
> db.person.find()
{ "_id" : ObjectId("6223bf604c5c0ad6603910fd"), "name" : "john", "age" : 31 }
{ "_id" : ObjectId("6223bf8e4c5c0ad6603910fe"), "name" : "john", "age" : 31 }
{ "_id" : ObjectId("6223c1f24c5c0ad6603910ff"), "name" : "john" }
{ "_id" : ObjectId("6223c20e4c5c0ad660391100"), "name" : "john", "age" : 30 }
{ "_id" : 1, "scores" : [ 3, 2, 5, 2, 4 ] }
{ "_id" : 2, "scores" : [ 2, 8, 5 ] }
{ "_id" : 3, "did" : 1 }
{ "_id" : 4, "did" : 2 }
{ "_id" : ObjectId("6244f20fe6721a6e371c8674"), "age" : 30, "gender" : "M" }
> db.department.find()
{ "_id" : 1, "name" : "CS" }
{ "_id" : 2, "name" : "ECE" }
> select * from person, department where person.did = department._id;
uncaught exception: SyntaxError: unexpected token: identifier :
@(shell):1:14
```

```
> db.person.aggregate({$lookup: {from: "department", localField: "did",  foreignField: "_id", as:
"result"}}, {$match: {result: {$ne: []}}})
{ "_id" : 3, "did" : 1, "result" : [ { "_id" : 1, "name" : "CS" } ] }
{ "_id" : 4, "did" : 2, "result" : [ { "_id" : 2, "name" : "ECE" } ] }
> db.person.aggregate({$match: {did: {$exists: true}}}, {$lookup: {from: "department", localField:
 "did",  foreignField: "_id", as: "result"}})
{ "_id" : 3, "did" : 1, "result" : [ { "_id" : 1, "name" : "CS" } ] }
{ "_id" : 4, "did" : 2, "result" : [ { "_id" : 2, "name" : "ECE" } ] }
```

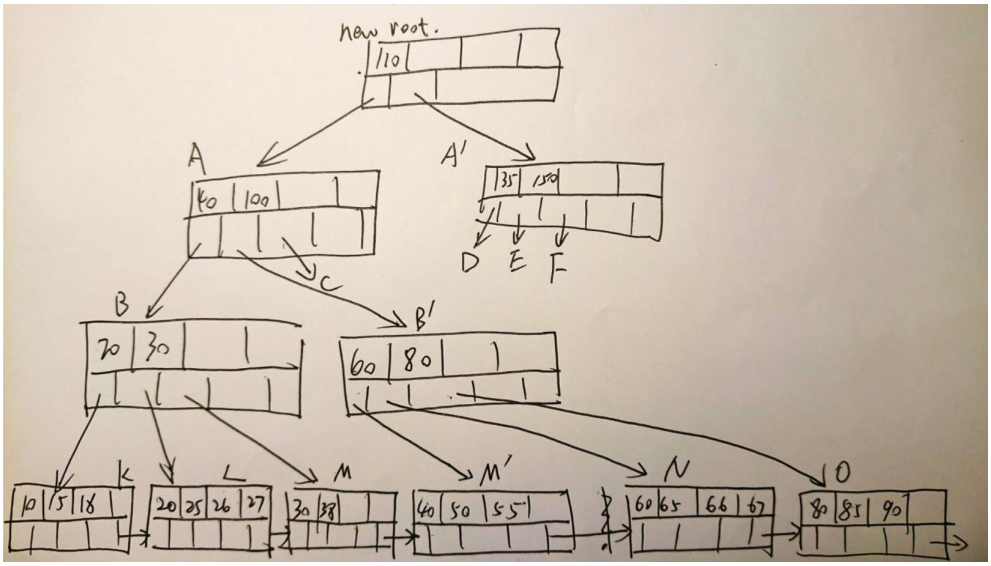
11. B+ Tree(https://zhuanlan.zhihu.com/p/149287061)

a. 插入操作(insert)

1. [8 points] Draw the updated B+-tree after inserting **38** into the following tree.

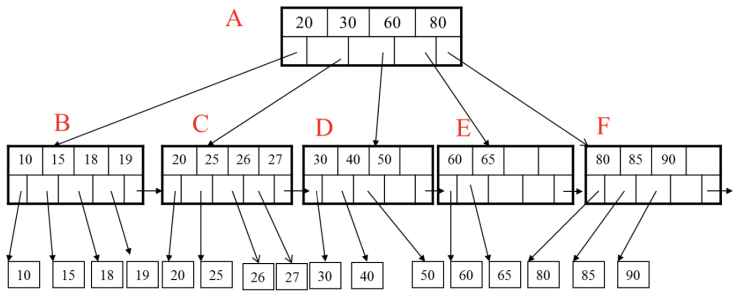


1. 能插直接插。
2. 插了以后发现节点数大于能存的最大节点数 → split，并将中间元素丢到parent去。
3. 如果parent加了以后也装不下了 → split，并将中间元素丢到parent的parent去
4. 一直到如果root节点也装不下了 → split，生成一个新的root，并将中间元素丢到这个新的root里去。



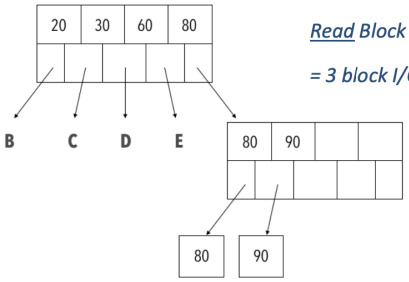
5. **cost (i.e., the number of block I/O's)** of this insertion：需要读**A,B,M**，需要写**New root, A, A', B, B', M, M'**。一共是3+7=10个block I/O's

b. 删除操作(delete)



1. 能删直接删。

1. [4 points] Draw the updated tree after deleting 85. What is the cost (# of block I/O's) of this deletion?



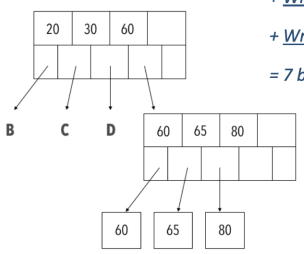
Cost of deleting 85:

Read Block A + Read Block F + Write Block F

= 3 block I/Os

2. 删了以后发现节点数小于能存的最大节点数的一半 → 向邻居借元素，并修改parent节点。
3. 如果邻居也没有余粮了 →跟邻居合并(生成一个新的)，再向parent借元素。
4. 如果parent被你借完以后也只剩一个了(因为只有root节点能只包含一个元素) → parent向它的邻居借元素，并修改它的parent。
5. 一直借到root节点为止，如果root节点最后一个也被借走了，那下面的就成为新的root节点。

2. [6 points] Draw the updated tree after further deleting 90 **from the tree produce in question 1**. What is the cost of this deletion?



Cost of deleting 90:

Read Block A + Read Block D + Read Block E + Read Block F

+ Write updated Block A + Write updated Block D

+ Write a new node for combined Block E&F

= 7 block I/Os

Note:

if you create a new node for the combined Block E&F

then you should also read/write D to connect the new block!

(5 block I/Os is acceptable if you write E&F in E)

6. 这里在计算cost的时候要注意3里面的标红部分很重要，因为生成一个新节点E'意味着这个节点前面的**D**节点虽然跟它没关系但是它要被读因为要修改它的节点指针指向新的**E'**所以它既有**read**也有**write**。

12. Query Execution

a. One-pass Algorithms

1. Tuple-based Nested Loop Joins(假设两个relationS和R都是**unclustered**的)

   R⋈S:

   for each tuple r in R do

   for each tuple s in S do

   if r and s join then output (r,s)

   Cost: T(S)*T(R)

2. Block-based Nested Loop Joins(假设两个relationS和R都是**clustered**的)

- **R⋈S:**

  For each (M-2) blocks b_r of R do:

  For each block b_s of S do:

  For each tuple t_s of S do:

  For each tuple t_r of R do:

  If "t_s and t_r join" then output(t_s, t_r)

  Cost: B(R) + B(R)/(M-2)*B(S)

  Assume B(R)≤B(S) & B(R)>M

- **S⋈R:**

  For each (M-2) blocks b_r of R do:

  For each block b_s of S do:

  For each tuple t_s of S do:

  For each tuple t_r of R do:

  If "t_s and t_r join" then output(t_s, t_r)

  Cost: B(S) + B(S)/(M-2)*B(R)

b. Two-pass Algorithms

1. Sort-Merge join

   a. Step 1: split R into runs of size M, then split S into runs of size M.

Cost: 2B(R) + 2B(S)

b. Step 2: merge M-1 runs from R and S; output a tuple on a case by cases basis

Total cost: 2B(R)+2B(S)+B(R)+B(S)=3B(R)+3B(S)

Assumption: B(R)+B(S) ≤ M^2

c. 如果不满足上面条件，就是merge的时候不能一个pass搞定，那就跟外部排序merge一样接着merge M-1 runs，每次merge增加2B(S or R)。

2. Partitioned Hash Join

R⋈S

a. Step 1: Hash S into M – 1 buckets → send all buckets to disk

Cost: 2B(S)

b. Step 2: Hash R into M – 1 buckets → Send all buckets to disk

Cost: 2B(R)

c. Step 3: Join every pair of corresponding buckets

Cost: B(S) + B(R)

Total cost: 2B(R)+2B(S)+B(R)+B(S)=3B(R)+3B(S)

Assumption: min(B(R), B(S))/(M-1) ≤ M-3

★ Step1和Step2中的hash function一定要和Step3中的hash function不同。如果相同的话Step3中所有经过hash function的数据都会被放在同一个**bucket里(end up with the same bucket)**。

13. NoSQL(*non-relational*, *distributed*, *open-source*, *horizontal scalable*)

c. Transactions – ACID Properties(RDBMS)

- **Atomicity, [*Consistency(strong)*], Isolation, Durability**
- By ***giving up*** ACID properties, one can ***achieve higher performance and scalability***.

d. CAP Theorem(consistency(strong consistency), availability, partitions)

- **Consistency:** Once a writer has written, all readers will see that write**.**
- **Availability:** System is available during software and hardware upgrades and node failures.

- **Partitions:** A system can continue to operate in the presence of a network partitions.
- **Theorem: You can have at most *two* of these properties for any shared-data system(In almost all cases, you would choose *availability > consistency*)**

e. Transactions – BASE Properties(NoSQL)

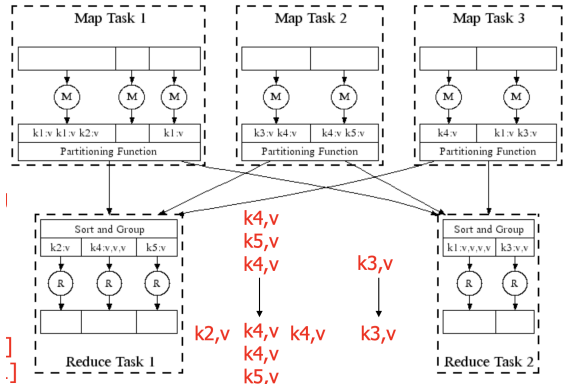
- **Basically Available:** Nodes in the a distributed environment can go down, but the whole system shouldn't be affected.
- **Soft State(scalable):** The state of the system and data changes over time.
- **Eventual Consistency(weak consistency):** Given enough time, data will be consistent across the distributed system.

14. MapReduce(https://zhuanlan.zhihu.com/p/55884610)

a. Suffle

1. Suffle例子（两个nodes），根据hash分配key给不同的reduce task。

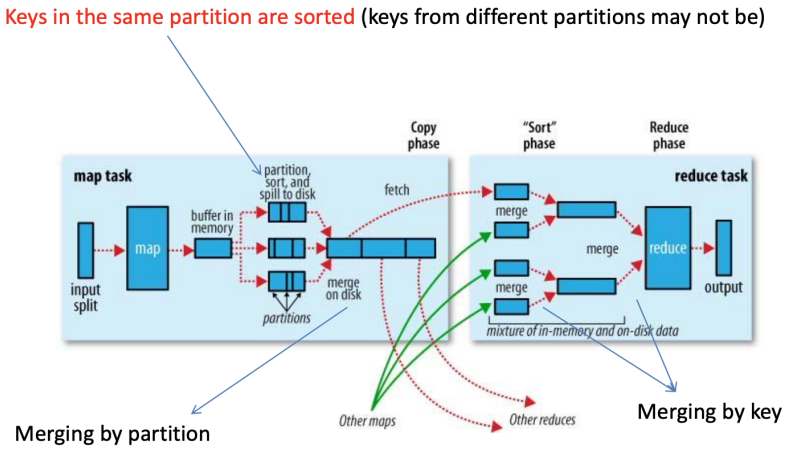
```
[ec2-user@ip-172-31-15-144 class]$ cat input/hello1.txt
hello world
hello this world
hello hello world
[ec2-user@ip-172-31-15-144 class]$ cat input/hello2.txt
hello world
hello this world
hello that world
dsci 351
dsci 352
[ec2-user@ip-172-31-15-144 class]$ cat output3/part-r-00000
351     1
dsci    2
that    1
[ec2-user@ip-172-31-15-144 class]$ cat output3/part-r-00001
352     1
hello   7
this    2
world   6
```



2. 为什么要排序？ → 因为要group，排序以后从头开始看，什么时候key跟上一个key不一样了什么时候就应该是新的group了。
3. **Suffle process(下图)**

- 这里的sort&merge用的是外部排序。要跟上面那个multi-tasks的图结合起来看。
    a. 整个过程就是首先Maps会各自map出一些(key,value)的对，然后根据hash分别将不同的key分开成不同的partitions各自排序。比如在Map Task2中这一步的结果应该是{(k4,v), (k4,v), (k5,v)}, {(k3,v)}，因为k4和k5是要给Reduce Task1的，而k3是要给Reduce Task2的。

b. 然后Reduce Task也会收到来自不同Map Task的包裹，包裹内各自是排好序的，因此Reduce Task只需要执行外部排序的**merge**部分就可以最终排好序的group了。整个过程叫做suffle。



b. Combiner

1. 可以理解为Map Tasks处理完各自的任务之后先进行一个中间操作，再给Reducer Tasks;这么做可以减小数据传输从而加快整体速度(但不是什么情况都可以用combiner的，比如求平均就不可以)。
2. 以统计单词个数为例：假设有两个Map Tasks和一个Reduce Task。其中：
   a. *Mapper 1*

      input: : [(0, “here”), (5, “here and there”)]

      output: [(“here”, 1), (“here”, 1), (“and”, 1), (“there”, 1)]
   b. *Mapper 2*

      input: : [(0, “here or there”), (14, “there”)]

      output: [(“here”, 1), (“or”, 1), (“there”, 1), (“there”, 1)]
   c. 经过**combiner**的处理以后：
   d. *Combiner 1:* **[(“here”, 2), (“and”, 1), (“there”, 1)]**
   e. *Combiner 2:* **[(“here”, 1), (“or”, 1), (“there”, 2)]**
   f. 可以看出来这里**combiners**先统统计了各自的答案，再给**Reducer**
   g. 最后**Reducer**整理出的结果：**[(“here”, 3), (“and”, 1), (“or”, 1), (“there”, 3)]**

15. aSpark Dataframe & RDD

```
>>> country.groupBy('continent').agg(fc.max('gnp').alias('max_gnp'), fc.count('*').alias('cnt')).show()
+-------------+---------+---+
|    continent|  max_gnp|cnt|
+-------------+---------+---+
|       Europe|2133367.0| 46|
|       Africa| 116729.0| 58|
|North America|8510700.0| 37|
|   Antarctica|      0.0|  5|
|South America| 776739.0| 14|
|      Oceania| 351182.0| 28|
|         Asia|3787042.0| 51|
+-------------+---------+---+

>>> spark.sql('select continent, max(gnp) max_gnp, count(*) cnt from country group by continent').show()
+-------------+---------+---+
|    continent|  max_gnp|cnt|
+-------------+---------+---+
|       Europe|2133367.0| 46|
|       Africa| 116729.0| 58|
|North America|8510700.0| 37|
|   Antarctica|      0.0|  5|
|South America| 776739.0| 14|
|      Oceania| 351182.0| 28|
|         Asia|3787042.0| 51|
+-------------+---------+---+
```

```
>>> spark.sql('select continent, max(gnp) max_gnp, count(*) cnt from country group by continent having count(*)
 > 5 order by cnt desc').show()
+-------------+---------+---+
|    continent|  max_gnp|cnt|
+-------------+---------+---+
|       Africa| 116729.0| 58|
|         Asia|3787042.0| 51|
|       Europe|2133367.0| 46|
|North America|8510700.0| 37|
|      Oceania| 351182.0| 28|
|South America| 776739.0| 14|
+-------------+---------+---+

>>> country.groupBy('continent').agg(fc.max('gnp').alias('max_gnp'), fc.count('*').alias('cnt')).where('cnt > 5
').orderBy('cnt', ascending = False).show()
+-------------+---------+---+
|    continent|  max_gnp|cnt|
+-------------+---------+---+
|       Africa| 116729.0| 58|
|         Asia|3787042.0| 51|
|       Europe|2133367.0| 46|
|North America|8510700.0| 37|
|      Oceania| 351182.0| 28|
|South America| 776739.0| 14|
+-------------+---------+---+
```

```
>>> country.join(city, country.Capital == city.ID).show(2)
+-------+----+-----+---------+--------+--------+--------------------+------------+---------+-------------+---
----------------+-----------+----------+--------------------+-----------+-----------+-------------+---+-------
--+----------+
|Capital|Code|Code2|Continent|     GNP|  GNPOld|      GovernmentForm|  HeadOfState|IndepYear|LifeExpectancy|
       LocalName|       Name|Population|              Region|SurfaceArea|CountryCode|     District| ID|     Na
me|Population|
+-------+----+-----+---------+--------+--------+--------------------+------------+---------+-------------+---
----------------+-----------+----------+--------------------+-----------+-----------+-------------+---+-------
--+----------+
|      1| AFG|   AF|     Asia|  5976.0|     0.0|     Islamic Emirate|Mohammad Omar|     1919|          45.9|Afg
anistan/Afqane...|Afghanistan|  22720000|Southern and Cent...|   652090.0|        AFG|        Kabol|  1|    Kab
ul|   1780000|
|      5| NLD|   NL|   Europe|371362.0|360478.0|Constitutional Mo...|      Beatrix|     1581|          78.3|
       Nederland|Netherlands|  15864000|      Western Europe|    41526.0|        NLD|Noord-Holland|  5|Amsterd
am|    731200|
+-------+----+-----+---------+--------+--------+--------------------+------------+---------+-------------+---
----------------+-----------+----------+--------------------+-----------+-----------+-------------+---+-------
--+----------+
only showing top 2 rows
```

RDD

```
>>> action: collect, take
  File "<stdin>", line 1
    action: collect, take
                   ^
SyntaxError: invalid syntax
>>> transformation: map filter
  File "<stdin>", line 1
    transformation: map filter
                          ^
```

```
>>> country_rdd.map(lambda r: (r['Name'], r['GNP'])).map(lambda r: r[1] > 10000).collect()
[False, False, False, False, False, False, False, True, True, False, False, False, False, False, True, True, Fa
lse, False, True, False, False, True, True, False, False, False, True, False, False, False, True, False, True,
False, False, False, False, True, False, True, True, True, True, False, False, False, False, False, True, False, False
, True, True, False, False, False, True, True, False, False, True, True, True, True, True, False, False, True,
False, False, True, False, False, True, False, False, False, True, False, False, False, False, False, False, Fa
lse, False, True, False, False, True, False, False, False, True, False, False, True, False, True, True, True, F
alse, True, True, True, False, True, True, False, False, False, False, False, True, True, True, False, True, T
rue, False, True, False, True, False, False, True, False, True, True, False, False, True, False, False, False,
False, True, False, False, False, False, True, False, False, False, False, False, False, False, False, True, Fa
lse, False, False, False, False, True, False, False, False, True, True, False, False, True, True, True, False,
True, True, False, False, True, True, False, True, False, False, False, False, False, True, True, False, True,
True, False, True, False, False, False, False, False, True, False, False, False, False, False, True, True, True
, False, False, True, False, False, False, True, False, False, False, False, False, False, True, True, False, T
rue, False, False, True, False, True, True, True, False, False, True, False, False, True, False, False, False,
False, True, True, False, False]
```

```
>>> country_rdd.map(lambda r: (r['Name'], r['GNP'])).filter(lambda r: r[1] > 10000).collect()
[('United Arab Emirates', 37966.0), ('Argentina', 340238.0), ('Australia', 351182.0), ('Austria', 211860.0), ('
Belgium', 249704.0), ('Bangladesh', 32852.0), ('Bulgaria', 12178.0), ('Belarus', 13714.0), ('Brazil', 776739.0)
, ('Brunei', 11705.0), ('Canada', 598862.0), ('Switzerland', 264478.0), ('Chile', 72949.0), ('China', 982268.0)
, ('CÃ´te dÃ'Ivoire', 11345.0), ('Colombia', 102896.0), ('Costa Rica', 10226.0), ('Cuba', 17843.0), ('Czech Rep
ublic', 55017.0), ('Germany', 2133367.0), ('Denmark', 174099.0), ('Dominican Republic', 15846.0), ('Algeria', 4
9982.0), ('Ecuador', 19770.0), ('Egypt', 82710.0), ('Spain', 553233.0), ('Finland', 121914.0), ('France', 14242
85.0), ('United Kingdom', 1378330.0), ('Greece', 120724.0), ('Guatemala', 19008.0), ('Hong Kong', 166448.0), ('
Croatia', 20208.0), ('Hungary', 48267.0), ('Indonesia', 84982.0), ('India', 447114.0), ('Ireland', 75921.0), ('
Iran', 195746.0), ('Iraq', 11500.0), ('Israel', 97477.0), ('Italy', 1161755.0), ('Japan', 3787042.0), ('Kazakst
an', 24375.0), ('South Korea', 320749.0), ('Kuwait', 27037.0), ('Lebanon', 17121.0), ('Libyan Arab Jamahiriya',
 44806.0), ('Sri Lanka', 15706.0), ('Lithuania', 10692.0), ('Luxembourg', 16321.0), ('Morocco', 36124.0), ('Mex
ico', 414972.0), ('Myanmar', 180375.0), ('Malaysia', 69213.0), ('Nigeria', 65707.0), ('Netherlands', 371362.0),
 ('Norway', 145895.0), ('New Zealand', 54669.0), ('Oman', 16904.0), ('Pakistan', 61289.0), ('Peru', 64140.0), (
'Philippines', 65107.0), ('Poland', 151697.0), ('Puerto Rico', 34100.0), ('Portugal', 105954.0), ('Romania', 38
158.0), ('Russian Federation', 276608.0), ('Saudi Arabia', 137635.0), ('Sudan', 10162.0), ('Singapore', 86503.0
), ('El Salvador', 11863.0), ('Slovakia', 20594.0), ('Slovenia', 19756.0), ('Sweden', 226492.0), ('Syria', 6598
4.0), ('Thailand', 116416.0), ('Tunisia', 20026.0), ('Turkey', 210721.0), ('Taiwan', 256254.0), ('Ukraine', 421
68.0), ('Uruguay', 20831.0), ('United States', 8510700.0), ('Uzbekistan', 14194.0), ('Venezuela', 95023.0), ('V
ietnam', 21929.0), ('Yugoslavia', 17000.0), ('South Africa', 116729.0)]
```

```
>>> spark.sql('select continent, max(gnp) from country group by continent').show()
+-------------+---------+
|    continent| max(gnp)|
+-------------+---------+
|       Europe|2133367.0|
|       Africa| 116729.0|
|North America|8510700.0|
|   Antarctica|      0.0|
|South America| 776739.0|
|      Oceania| 351182.0|
|         Asia|3787042.0|
+-------------+---------+
```

```
>>> country_rdd.map(lambda r: (r['Continent'], r['GNP'])).groupByKey().mapValues(list).collect()
[('North America', [828.0, 63.2, 1941.0, 612.0, 3527.0, 630.0, 2328.0, 2223.0, 598862.0, 10226.0, 17843.0, 1263
.0, 256.0, 15846.0, 3501.0, 318.0, 0.0, 19008.0, 5333.0, 3459.0, 6871.0, 299.0, 571.0, 414972.0, 109.0, 2731.0,
 1988.0, 9131.0, 34100.0, 11863.0, 0.0, 96.0, 6232.0, 8510700.0, 285.0, 612.0, 0.0]), ('Asia', [5976.0, 37966.0
, 1813.0, 4127.0, 32852.0, 6366.0, 11705.0, 372.0, 982268.0, 9333.0, 6064.0, 166448.0, 84982.0, 447114.0, 19574
6.0, 11500.0, 97477.0, 7526.0, 3787042.0, 24375.0, 1626.0, 5121.0, 320749.0, 27037.0, 1292.0, 17121.0, 15706.0,
 5749.0, 199.0, 180375.0, 1043.0, 69213.0, 4768.0, 16904.0, 61289.0, 65107.0, 5332.0, 4173.0, 9472.0, 137635.0,
 86503.0, 65984.0, 116416.0, 1990.0, 4397.0, 0.0, 210721.0, 256254.0, 14194.0, 21929.0, 6041.0]), ('Africa', [6
648.0, 903.0, 2357.0, 2425.0, 4834.0, 1054.0, 11345.0, 9174.0, 6964.0, 2108.0, 4401.0, 435.0, 382.0, 49982.0, 8
2710.0, 650.0, 60.0, 6353.0, 5493.0, 7137.0, 2352.0, 320.0, 293.0, 283.0, 0.0, 9217.0, 2012.0, 44806.0, 1061.0,
 36124.0, 3750.0, 2642.0, 2891.0, 998.0, 4251.0, 1687.0, 0.0, 3101.0, 1706.0, 65707.0, 8287.0, 2036.0, 10162.0,
 4787.0, 0.0, 746.0, 935.0, 6.0, 1206.0, 536.0, 1208.0, 1449.0, 20026.0, 8005.0, 6313.0, 116729.0, 3377.0, 5951
.0]), ('Europe', [3205.0, 1630.0, 211860.0, 249704.0, 12178.0, 2841.0, 13714.0, 264478.0, 55017.0, 2133367.0, 1
74099.0, 553233.0, 5328.0, 121914.0, 1424285.0, 0.0, 1378330.0, 258.0, 120724.0, 20208.0, 48267.0, 75921.0, 825
5.0, 1161755.0, 1119.0, 10692.0, 16321.0, 6398.0, 776.0, 1579.0, 1694.0, 3512.0, 371362.0, 145895.0, 151697.0,
105954.0, 38158.0, 276608.0, 0.0, 510.0, 20594.0, 19756.0, 226492.0, 42168.0, 9.0, 17000.0]), ('South America',
 [340238.0, 8571.0, 776739.0, 72949.0, 102896.0, 19770.0, 0.0, 681.0, 722.0, 64140.0, 8444.0, 870.0, 20831.0, 9
5023.0]), ('Oceania', [334.0, 351182.0, 0.0, 100.0, 0.0, 1536.0, 212.0, 1197.0, 40.7, 97.0, 0.0, 3563.0, 0.0, 0
.0, 197.0, 54669.0, 0.0, 105.0, 4988.0, 818.0, 182.0, 0.0, 146.0, 6.0, 0.0, 261.0, 0.0, 141.0]), ('Antarctica',
 [0.0, 0.0, 0.0, 0.0, 0.0])]
>>> country_rdd.map(lambda r: (r['Continent'], r['GNP'])).groupByKey().map(lambda kl: kl[0]).collect()
['North America', 'Asia', 'Africa', 'Europe', 'South America', 'Oceania', 'Antarctica']
>>> country_rdd.map(lambda r: (r['Continent'], r['GNP'])).groupByKey()
PythonRDD[279] at RDD at PythonRDD.scala:53
>>> country_rdd.map(lambda r: (r['Continent'], r['GNP'])).reduceByKey(max).collect()
[('North America', 8510700.0), ('Asia', 3787042.0), ('Africa', 116729.0), ('Europe', 2133367.0), ('South Americ
a', 776739.0), ('Oceania', 351182.0), ('Antarctica', 0.0)]
```

```
>>> country_rdd.map(lambda r: (r['Continent'], r['GNP'])).groupByKey().mapValues(lambda v: max(v)).collect()
[('North America', 8510700.0), ('Asia', 3787042.0), ('Africa', 116729.0), ('Europe', 2133367.0), ('South Americ
a', 776739.0), ('Oceania', 351182.0), ('Antarctica', 0.0)]
```