Assignment 1
Group 5: Qi Lin, Xiaoying Li, Ziying Yu
4/14/2022

Question 1

- Briefly Description:

The inode data structure stores information about any Linux file except its name and data. One inode represents one file or one directory. Inodes don't contain any of the file's actual data, it stores the file's metadata, including all the storage blocks on which the file's data can be found.

Extent-based allocation refers to the file as a collection of clusters of consecutive disk blocks(extents) where the collection is maintained using linked lists or indexes.

- Comparison:

Extent-based is suitable for sequential and random access to a large database since extent-based files are contiguously allocated. The next block to read is physically the next on the disk. No seek time to find the next block, and each block will be read sequentially as the disk head moves. However, inodes are uniform. Once a file exceeds the size of an inode, the original inode will have to point to another inode to encompass the file's metadata fully.

Inodes might run out on a device, such as a mail server database that contains many small files. In this case, new files cannot be created on the device, and even free space may be available. Essentially, you will have to rebuild the filesystem and include a larger limit for inodes. However, extent-based escapes this limitation by dynamically growing the file system.

Inodes are independent of filenames and support hard links. If multiple names hard link to the same inode, then the names are equivalent. However, Extent-based does not have the equivalent feature.

Extent-base can suffer heavily from external fragmentation, especially for large files in database relations. So it is not the best structure for getting the most bytes on the disk since lots of space will be wasted. At the same time, inode-based has more internal fragmentation because the block can still not be used entirely.

In the inode base, "lseek" is used to achieve non-sequential file I/O. Rather than O(n), this implementation is O(1) and is constant regardless of the size of things. Compared to inode-based, usage of linked lists on disk in extent-based may cause relatively poor performance in access time.

Finding free space in the database for the inodes data structure is fast. It only needs to check the block bitmap to see where something can be allocated. Extent-based file systems keep a free list of unused areas of the disk. In the beginning, the free list is sorted, and blocks in a file are allocated contiguously. But the free list quickly becomes scrambled, which results in a hard time finding free space since files are spread all over the disk.

Reference:
https://en.wikipedia.org/wiki/Extent_(file_systems)
https://en.wikipedia.org/wiki/Inode
https://www.scs.stanford.edu/10wi-cs140/notes/l12.pdf
https://inst.eecs.berkeley.edu/~cs162/fa15/static/sections/section10.pdf

Question 2

The ACID properties are atomicity, consistency, isolation, and durability.

- Example for atomicity (either all the effects of a transaction appear in the database or none of the effects of a transaction appears in the database):

Consider a transaction T of transferring $200 from account A to account B, which consists of two processes T1 and T2. Process T1 is Read(A) → Deduct $200 from account A → Write(A). Process T2 is Read(B) → Add $200 to account B → Write(B). Now if the transaction T fails after completion of process T1, but before completion of process T2. More specifically, the transaction T fails after Write(A), but before Write(B). Then if without the atomicity property, the amount would have been deducted from account A, but not added to account B, which would result in an inconsistent database state.

- Example for consistency (each transaction maps a database from a consistent state to another consistent state):

Consider the same transaction T in the last example, let's say originally account A's amount is $300, account B's amount is $100, and the total amount of account A and account B is $300 + $100 = $400. When the transaction T fails after completion of process T1, but before completion of process T2, i.e. the transaction T fails after Write(A), but before Write(B), then at this moment account A's amount is $300 - $200 = $100 since Write(A) completes, but account B's amount is still $100 since transaction T fails before Write(B). Now if without the consistency property, the total amount of account A and account B would be $100 + $100 = $200, which is not equal to the original total amount of account A and account B, and result in an inconsistent database state.

- Example for isolation (concurrent execution of a transaction is hidden from other concurrently executing transactions):

Consider two concurrent transactions T1 and T2. Transaction T1 is transferring $200 from Account A to account B, which process is Read(A) → Deduct $200 from account A → Write(A) for account A and Read(B) → Add $200 to account B → Write(B) for account B. Transaction T2 is transferring all amounts of account A and account B to account C. Let's say originally, account A's amount is $300, account B's amount is $100, and account C's amount is $400. Suppose transaction T2 starts when transaction T1 has been executed

till the process Read(B). As a result, interleaving of operations happens, and without isolation property, transaction T2 would read the correct value of account A ($300 - $200 = $100) since Write(A) is already completed, but the incorrect value of account B (still $100) since Write(B) has not been done. Thus, the total transferring amount calculated by transaction T2 is $100 + $100 = $200, which is not consistent with the total transferring amount at the end of the transaction of $100 + $300 = $400. And resulting in an inconsistent database state, due to a loss of $200.

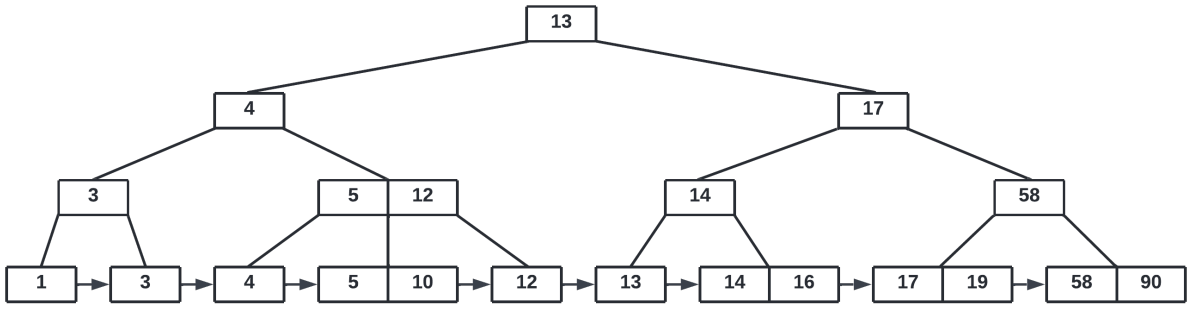
- Example for durability (if a transaction completes its effects are permanent and survive failures):

Consider a transaction T of withdrawing $100 from account A. And originally, account A's amount is $200. After transaction T is completed, the system fails before A's transaction is logged in the database, then when the system recovers, account A's amount would remain as $200 if without the durability property, which results in an inconsistent database state.

Reference:
https://www.geeksforgeeks.org/acid-properties-in-dbms/

Question 3

(a)



(b)

(i)

While finding the record with the key value 10 in the B+ tree constructed in part (a), we will access the following sequences of pages:

1. Page 1: the root record with key value 13, since 10 is smaller than 13, go to the left child;
2. Page 2: the record with key value 4, since 10 is greater than 4, go to the right child;
3. Page 3: the record with key value 5 and 12, since 10 is greater than 5 but smaller than 12, go to the middle child;
4. Page 4: the record with key value 5 and 10, since it's the leaf node, the record with key value 10 is found in the B+ tree.

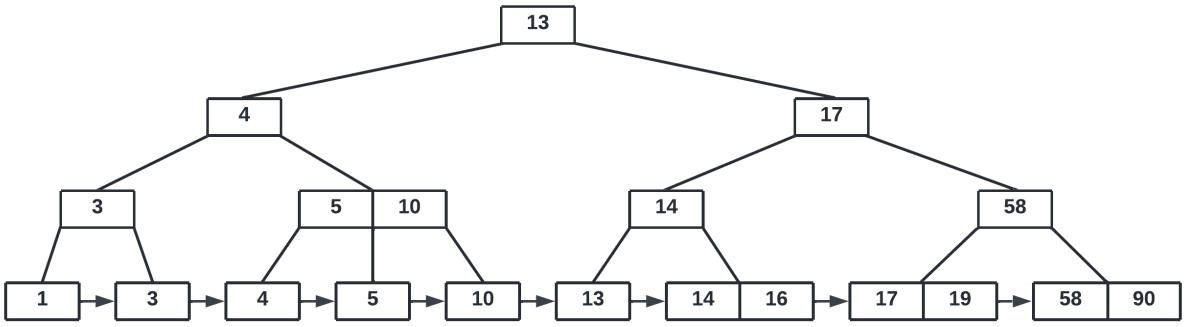
(ii)

While finding the record with key value in the range from 14 to 19 inclusive in the B+ tree constructed in part (a), we first need to find the record with the key value 14, and we will access the following sequences of pages:

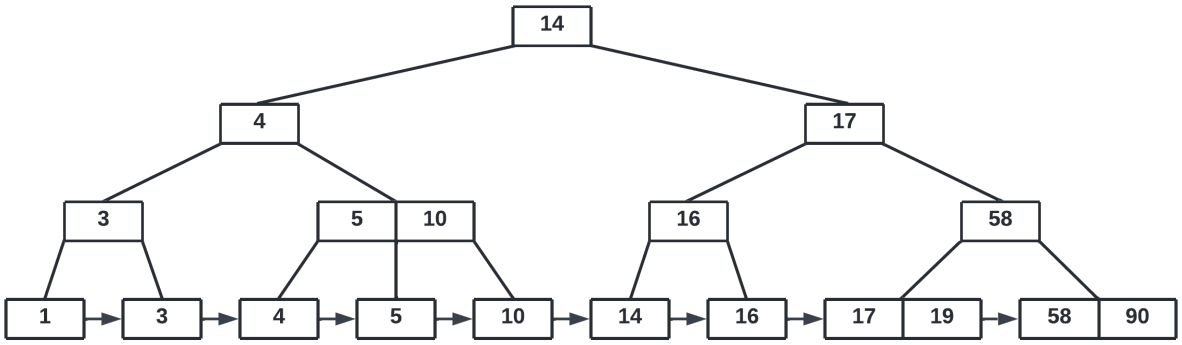
1. Page 1: the root record with key value 13, since 14 is greater than 13, go to the right child;
2. Page 2: the record with key value 17, since 14 is smaller than 17, go to the left child;
3. Page 3: the record with key value 14, since 14 is equal to 14, go to the right child;
4. Page 4 and 5: the record with key value 14 and 16, since it's a leaf node, use the right most pointer to find the right record with key value 17 and 19, which is also a leaf node. Thus, the record with the key values in the range from 14 to 19 inclusive is found in the B+ tree.

(c)

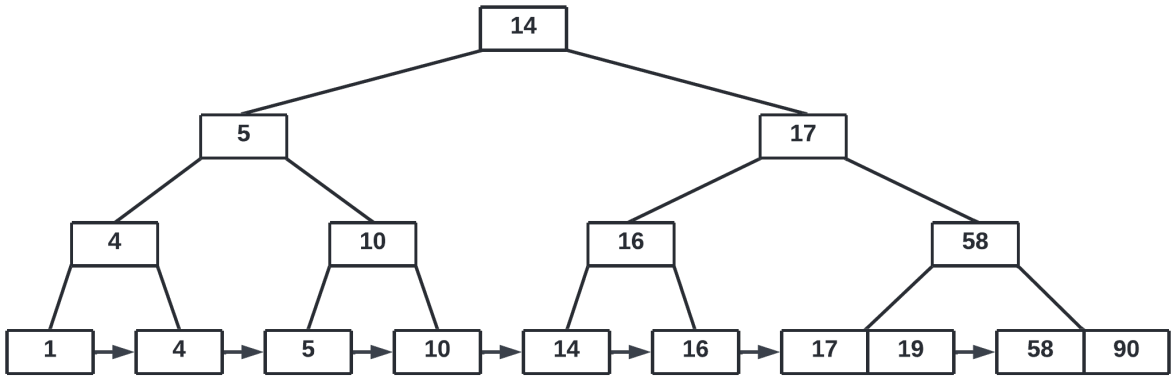
- After deleting 12:



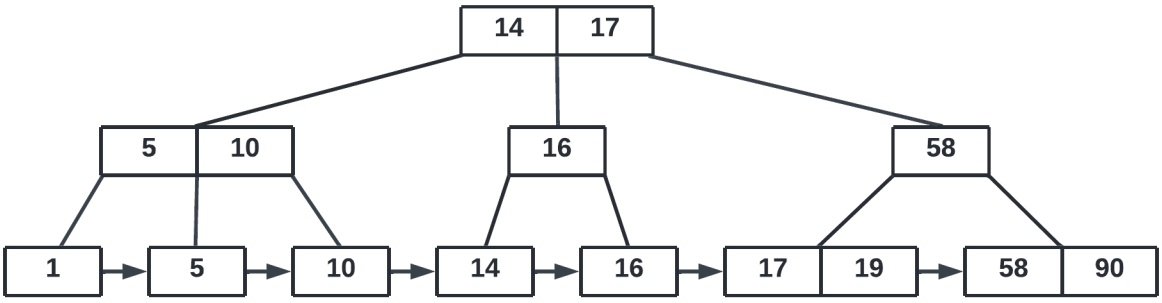
- After deleting 13:



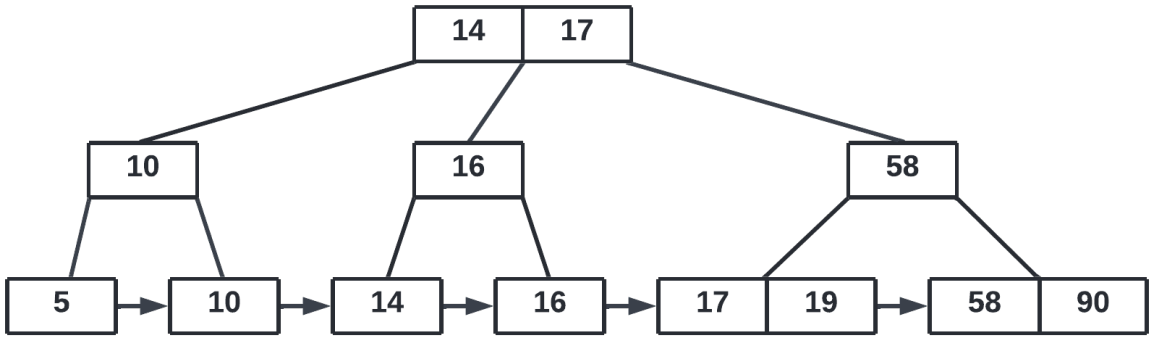
- After deleting 3:



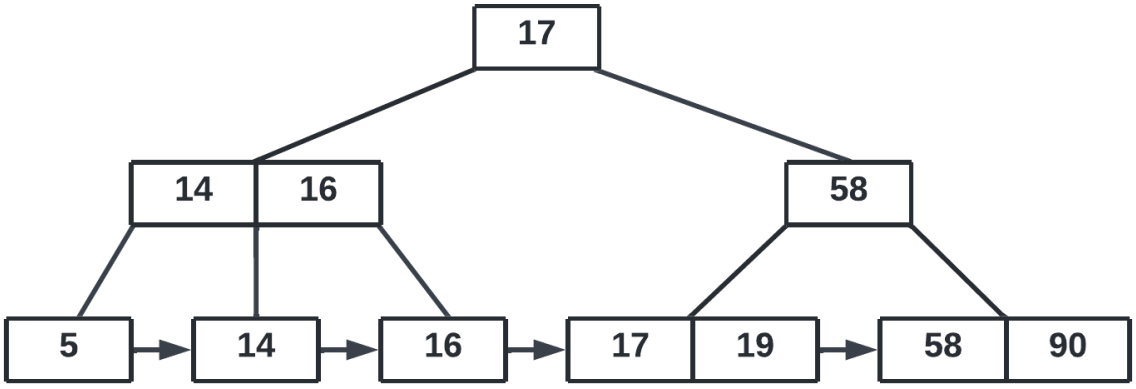
- After deleting 4:



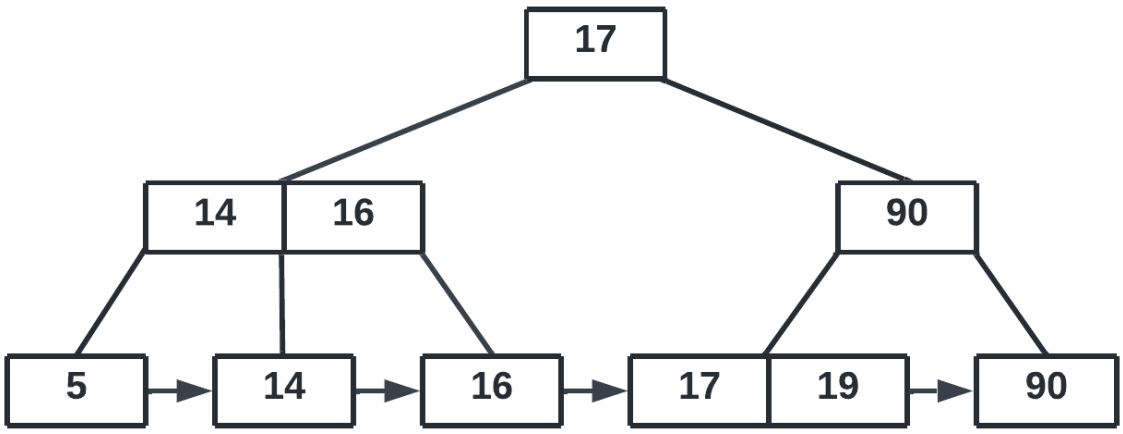
- After deleting 1:



- After deleting 10:



- After deleting 58 (final structure of the tree):



Question 4

Consider two transactions, T1 and T2, as following (Initially A=1000, B=1000):

T1: Move $100 from account A to account B	T2: Compute the total in all accounts and return it to the application.
Begin A=A-100 B=B+100 Commit	Begin Echo A+B Commit

Consider the correct scheduling: all lock events happen before any unlock events in T1 and T2:

T1	T2
Begin	Begin
x-lock(A)	
r(A)	s-lock(A)
A=A-100	
w(A)	
x-lock(B)	
unlock(A)	r(A)
s-lock(B)	
r(B)	
B=B+100	
w(B)	
unlock(B)	r(B)
Commit	unlock(A)
	unlock(B)
	Echo A+B
	Commit

Result: T2 output A+B = 2000, in which all the lock operations precede all the unlock operations and satisfied the serializability.

Now consider the following scheduling that violates the property:

T1	T2
Begin	Begin
x-lock(A)	
r(A)	
	s-lock(A)
A=A-100	
w(A)	
unlock(A)	
	r(A)
	unlock(A)
	s-lock(B)
x-lock(B)	
	r(B)
	unlock(B)
r(B)	Echo A+B
B=B+100	Commit
w(B)	
unlock(B)	
Commit	

Result: T2 output A+B =1900, which is an incorrect output led by the violation.

Analyze:

T1 first x-lock A in the beginning, it then reads A(1000), performs deduction(A=1000-100=900), and then writes it. At the same time, T2 first s-lock(a), it cannot read A until T1 unlocks A. After T1 unlocking A, T2 reads A(A=900). T2 then unlocks A and s-lock B. Since T2 s-lock B before T1 x-lock B, it reads B(B=1000) and then unlocks B. At this time, T2 unlocks B before T1's other operations(B=B+100=1000+100) are performed but the process to echo, which commits incorrectly to 1900 as opposed to 2000.

Reference: https://15445.courses.cs.cmu.edu/fall2018/slides/17-twophaselocking.pdf

Question 5

The value of X when the nested transaction T1 commits is 18.

Node	Action	Value of X
T1_C	commit	X = initial value = 10
T2_C	commit	X = 10
T3_C	commit	X = X + 3 = 10 + 3 = 13
T4_C	commit	X = 13
T5_C	commit	X = X + 4 = 13 + 4 = 17
T6_R	abort	/
T7_C	commit	X = 17
T8_C	commit	X = X - 3 = 17 - 3 = 14
T9_C	commit	X = 14
T10_C	commit	X = 14
T11_C	commit	X = X + 4 = 14 + 4 =18
T12_R	abort	/
T13_R	abort	/
T14_C	abort	/
T15_C	abort	/

Question 6

Write-ahead logging is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems. The changes are first recorded in the log, which must be written to stable storage before the changes are written to the database. In a system using WAL, all modifications are written to a log before they are applied. Usually, both redo and undo information is stored in the log.

Here is an example that if the system does not use write ahead logging. Imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Like a transaction (T) (read x; write x=4; write x=5; write x=6). When the second write operation was performed, which means the system committed and executed this operation but hasn't written to a log. After that, this machine crashed. Upon restart, the system might need to know whether the operation it was performing

succeeded, succeeded partially, or failed. If a write-ahead log is not used, the system cannot check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done. Since we haven't recorded the operation (write x=5) into the log, we don't know how to undo this operation if we decide to undo this transaction. Without the basis of this comparison, the program couldn't decide to undo what it had started, complete what it had started, or keep things as they are.

Reference: https://en.wikipedia.org/wiki/Write-ahead_logging

Question 7

2PC is formed of two phases, namely a voting phase and a decision phase. During the voting phase, the coordinator of a distributed system sends a prepare message to ask all participants in the transactions' execution to prepare to commit the transaction, whereas, during the decision phase, the coordinator either decides to commit the transaction if all the participants are prepared to commit or to abort if any participant has decided to abort.

When the coordinator fails, and at least one participant keeps waiting for the final decision of the coordinator, such a scenario depicts the 2PC as blocking, this means that participants cannot terminate the transaction (neither commit the transaction nor abort it) pending the recovery of the coordinator. if the coordinator crashes between the voting phase and the decision phase, a transaction can hold system resources for an unbounded period. Blocking means a non-failed process may have to wait for an unbounded time waiting for another process to recover.

Here is an example of blocking. Suppose a transaction (T)(read x; write x=4; read y; write y=4) holds locks (lock on x and y) on some participants during the execution, if the coordinator fails and no additional log record about the decision (commit or abort) is kept in these participants, these participants cannot determine what decision has been made on this transaction (T), whether commit or abort. So the final decision is delayed until the coordinator is restarted. During this delayed period, the locked data resource remains inaccessible for other transactions. This problem is known as the blocking problem.

Reference:
https://www.ijert.org/atomic-commit-in-distributed-database-systems-the-approaches-of-blocking-and-non-blocking-protocols
https://www.geeksforgeeks.org/two-phase-commit-protocol-distributed-transaction-management/?ref=rp