6.033 L15

4/4

Transactions

Today: intro to transactions

Later: more details

(this is what I was missing on the hands on)

atomicity

2 types

all-or-nothing atomicity

before-or-after atomicity

Can bild fault tolerant, reliable system

- dual power supplies
- etc

- highly avilable

But can still fail!

How do you recover from this?

But what do you do when you turn the machine back on?

You have a bunch of writes that you sent to disk

They might not have all been written

Failure could happen at any point

And optimizer means that data could be written in any order
└ concurrency

Challenge

Concurrency   failure

Can solve both
using transactions

Example Bank accounts

Balances	A	B	C	D	etc

Do various transactions

Including transfers

- subtract x from A
- add x to B

} can't have failures in b/w

But could still have power failure in b/w

$50 could just disappear

Goal: Transfer $ or not at all
(whole process) - no intermediate state

↳ all-or-nothing

Also an audit fn

L might run while transfers happening

Can't have fn run inbetween (during) the transfer

Only before or after the complete transfer

L before or after

We saw how to do it w/ locks

- tricky

Can we do it w/o locks?

L So program does not need to worry about

So have **transaction**

Add some program calls for the programmer

- begin
- commit() or abort()

Runs complete audit as **single** txn

Locks - need global lock/unlock order

Not for txn

So **compose** much better

Can we do this?

(Demo) - works very badly

(Only 1 txn at a time today)

Since writing does not just work

- lots of overhead

Violates golden rule: never overwrite your only copy!

- So if failure - only some have been written

You have no hope at all or nothing atomicity

Instead write a shadow copy

└ a new Version, under a new name

When done writing → rename file back to original

It works!

(And I think he said it will still work)

How does Rename work?

Rename is like the commit point

If write is correct, we commit

But is actually a complex op

We always need to know where Commit works

File System is actually doing something special

Rename code (wrong)

```
unlink(bank)
 link(#bank, bank)
 unlink(#bank)
```

This is not all or nothing

So need to change it

What if we crash here

We've lost the account file

We don't know if shadow is complete

So can't just copy that

We need to look closer at FS

FS do rename differently now

Actually look at inode + file data

We could play around w/ ref count so never 0

Rename (x, y)   [correct]

```
    ni = lookup(x)
    oi = lookup(y)
    increase ref count (ni)
    y → ni
    decrement ref count (oi)
    remove (x)
    decrement ref (ni)
```

Note update ref counts very carefully

key point: ref count not at 0 before made changes

(for bank, not #bank, right???)

What was commit point

bank inode = 13

What happens if fail in middle

ref cant may be too high

On recovery could regen inode counts "salvage"

Will always delete shadow file

L since we know real file is always correct

Failure can't happen <u>during</u> a transfer

Transactions reduce places to think about to just before and after

[This is not just WAL - I think I was confused by this]

Shadow Copy

⊕ No overwrite

logging ⊖ Operates on single file only

- not 2 files or directory structure
- rename would not be 1 atomic commit pt

4

logging ⊖ Copy whole file - even if only changing one line

- very expensive if file is huge!

isolation ⊖ One at a time

- What if multiple txn at a time

2 phase commit ⊖ Only works on a single machine

Post More details on example

2 threads - write / ~~[illegible]~~ transfer

- reader / audit

Write is very slow at write

- not atomic
- line by line

Basically do it all at once

- but can't write since not atomic
- instead rename - as being atomic

Basically implemented atomic write

10

Locking would help stop reader from reading
in middle

But does not save during crash
- locks disapper

4/4

6.033 2011 Lecture 15: Atomicity

Plan: Atomicity
all-or-nothing atomicity
before-or-after atomicity
implementation using shadow copy
rename

Recall overall goal: build reliable system out of unreliable components.

Last lecture and recitation: use replication to tolerate disk failure.
Checksum used to detect faults.
If cannot read one disk, read data from replica(s) instead.
Weaker (but still very effective): error-correcting codes on disk platter.
Replication allows masking component failure.

What happens if some component is not (sufficiently) replicated?
Often cannot afford to replicate everything in a system.
Many examples: processor, power supply, network (Internet), ..
Programming mistake faults cause failures in software modules; difficult to replicate for fault-tolerance (n-version programming is tricky).
In these situations, we need to _reason_ about what happens due to failure.
Then write software that, based on that reasoning, deals with failure.
Hard to reason about arbitrary failures (without replication), so here we will generally assume fail-stop or fail-fast failures.
This can be tricky: have to consider all possible ways a failure can occur.
E.g., may need to consider all possible times when CPU or power can fail.
May also need to consider concurrent operations that were in progress..

Next several lectures, starting with this one: atomicity.
Enables sweeping simplifications in reasoning about failures, concurrency.
Makes (reasoning about) recovery, and concurrent execution, much simpler.

Example: bank account application.
Let's consider the operaton xfer(a, b, amt): transfer amt from a to b.
[slide: xfer code]
Assume that the bank array (balances a and b) are stored on disk somehow.
If a disk fails, but we replicated the disk, xfer still operates correctly.
If system crashes between two statements (e.g., power fails): lost amt.
[slide: xfer crash]
Once system comes back up, there's less total money in the system.
Removed amt from a's account, but never added it to b's account.
Not just hardware/power failures:
Disk device driver / controller might crash, even with replicated disk
Python interpreter might crash
OS kernel might crash

What do we want to happen if a crash occurs during xfer?
Transfer occurred completely.
Transfer did not occur at all.
Anything else is unexpected to xfer's caller:
- hard to reason about
- hard to recover from

We call this notion "all-or-nothing atomicity" (xfer doesn't have it, yet).
An all-or-nothing atomic action makes it easier to reason about failures.
No need to think about possible internal, intermediate failure states.
Easier to recover from failures by considering just two cases.

Why specifically do we care about the balance being wrong after a crash?
Presumably there's another function, like audit(), which reads balances later.
[slide: audit code]
Our example crash causes audit() to return 150 after crash, instead of 200.
Turns out, similar problems can often occur with just concurrency (no crash).

```
  [ slide: different audit values ]

What do we want to happen during concurrency?
  audit() ran before xfer()
  audit() ran after xfer()
  Don't want to see unexpected effects when audit, xfer run concurrently.
  We call this "before-or-after atomicity".
    Again, hiding intermediate states of each function.
    Enables reasoning about entire functions, not their internal details.
  Recall: locks can be used to achieve before-or-after atomicity.
    Unfortunately, locks are hard to get right, require global reasoning.
    We will look at better abstractions for before-or-after atomicity.

Ultimate goal: transactions (both all-or-nothing & before-or-after atomicity).
  [ slide: xfer, audit transactions ]
  Very powerful abstraction.
    "Just add begin and commit", no need for explicit locking.
    Will ensure any concurrent xfers, audits execute "as if" sequentially.
    Tricky to implement (4 lectures), but enables sweeping simplifications.

---

For the rest of this lecture, let's see how to get all-or-nothing atomic xfer.
  Suppose all accounts stored in one large file.
  [ slide: file xfer ]
  Write account balances one at a time.
  Problem: what if system crashes between the two write steps?
  Problem: what if an audit runs?

Demo: stresses inconsistent behavior due to concurrency (i.e., audit)

Golden rule of atomicity: never modify the only copy

Idea: write to a shadow copy of the file first, then rename file in one step.
  [ slide: file xfer with shadow ]
  Here, we assume rename() will replace existing bankfile with "newfile".
  If system crashes before rename, bank contains old balances.
  If system crashes after rename, bank contains new balances.
  Of course, we need to ensure that only one operation runs at a time,
    but we will talk about concurrency later.
  We call the rename a "commit point".
    Commit point: crash before gives old value, crash after gives new value.
    Commit point must be itself an all-or-nothing action.
    Need rename (our commit point) to have all-or-nothing atomicity..

  Demo: with rename things are good

How to implement all-or-nothing rename?
  Can we use existing Unix file system ops link and unlink?
    unlink(bankname)
    link("newfile", bankname)
    unlink("newfile")
  No good: if we crash before link, no file "bank" at all.

  [ slide: file system data structures ]
  What needs to happen during rename?
    Point "bank" directory entry at inode 13.
    Remove "newfile" directory entry.
    Remove refcount on inode 12.
  First try at rename(x, y):
    y's dirent gets x's inode #
    decref(y's original inode)
    remove x's dirent
    What happens if we crash after modifying y's dirent?
      Potentially problematic: two names point to inode 13, refcount is 1.
      What if we increment refcount before, and decrement it afterwards?
      rename(x, y):
        newino = lookup(x)
        oldino = lookup(y)

        incref(newino)
        change y's dirent to newino
        decref(oldino)
        remove x's dirent
        decref(newino)
      We never have too few refcounts, but still might have too many.
      [ slides: rename in action ]
    Q: What's the commit point?  Modifying y's dirent.
    What if we crash during the commit point writing to the dirent?
      Assume that writing to one sector on disk is all-or-nothing.
      An ideal disk saves enough energy to complete one sector write.
      Time spent writing a sector is small (remember, high sequential speed).
      Small capacitor suffices to power disk for a few microseconds.
      If write didn't start, no need to complete it: still all-or-nothing.

  Fixing up after a crash: salvage.
    If we crash, our commit point ensures we have all-or-nothing atomicity.
    But we still have a bit of a mess left over because of the crash.
    If we crashed before commit: extra refcount on inode 13.
    If we crashed after commit: extra refcount on inode 12.
    In both cases: directory entry for "newfile" can be removed.
    [ slide: salvage function ]

  Why does shadow copy give us all-or-nothing atomicity?
    Write to a copy of data, atomically switch to new copy.
    Switching can be done with one all-or-nothing operation (sector write).
    Requires a small amount of all-or-nothing atomicity from lower layer (disk).
    Main rule: only make one write to current/live copy of data.
      In our example, sector write for rename.
      Creates a well-defined commit point.

  Does the shadow copy approach work in general?
    +: Works well for a single file.
    -: Hard to generalize to multiple files or directories.
       Might have to place all files in a single directory, or rename subdirs.
    -: Requires copying the entire file for any (small) change.
    -: Only one operation can happen at a time.
    -: Only works for operations that happen on a single computer, single disk.

  Next lecture: more general techniques for all-or-nothing atomicity.

  Summary of this lecture:
    Not always possible to use replication to avoid failures.
    To recover from failure, need to reason about what happened to the system.
    Atomicity makes it much easier to reason about possible system states:
      All-or-nothing atomicity.
      Before-or-after atomicity.
    Today's approach to all-or-nothing atomicity: shadow copy.
      Make all modifications to a shadow copy of data.
      Replace old copy with new copy with an all-or-nothing write to one sector.
        -> commit point
      Rule: never modify live data (unless one sector write is all you need)
```

L15: Transactions

Frans Kaashoek & Nickolai Zeldovich
6.033 Spring 2012

Bank account transfer

```
xfer(bank, a, b, amt):
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt
```

Bank account transfer

xfer(bank, a, b, 50):

```
xfer(bank, a, b, amt):         ← a=100, b=100
   bank[a] = bank[a] – amt     ← a=50, b=100
   bank[b] = bank[b] + amt     ← a=50, b=150
```

Bank account transfer

```
xfer(bank, a, b, amt):
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt

audit(bank):
   sum = 0
   for acct in bank:
      sum = sum + bank[acct]
   return sum
```

4/4

Bank account transfer

```
                                     audit(bank):

xfer(bank, a, b, amt):             ← sum=200
   bank[a] = bank[a] – amt         ← sum=150
   bank[b] = bank[b] + amt         ← sum=200

audit(bank):
   sum = 0
   for acct in bank:
      sum = sum + bank[acct]
   return sum
```

Eventual goal: transactions
all-or-nothing & before-or-after atomicity

```
xfer(bank, a, b, amt):
   begin
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt
   commit

audit(bank):
   begin
   sum = 0
   for acct in bank:
      sum = sum + bank[acct]
   commit
   return sum
```

Strawman implementation

```
xfer(bankfile, a, b, amt):
   bank = read_accounts(bankfile)
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt
   write_accounts(bankfile)
```

Shadow copy

```
xfer(bankfile, a, b, amt):
   bank = read_accounts(bankfile)
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt
   write_accounts("#bankfile")
   rename("#bankfile", bankfile)
```

File system data structures

directory data blocks:
filename “bank” → inode 12
filename “#bank” → inode 13

inode 12:
data blocks: 3, 4, 5
refcount: 1

inode 13:
data blocks: 6, 7, 8
refcount: 1

rename(“#bank”, “bank”)

directory data blocks:
filename “bank” → inode 12
filename “#bank” → inode 13

inode 12:
data blocks: 3, 4, 5
refcount: 1

inode 13:
data blocks: 6, 7, 8
refcount: 1

rename(“#bank”, “bank”)

directory data blocks:
filename “bank” → inode 12
filename “#bank” → inode 13

inode 12:
data blocks: 3, 4, 5
refcount: 1

inode 13:
data blocks: 6, 7, 8
refcount: **2**

rename(“#bank”, “bank”)

directory data blocks:
filename “bank” → inode **13**
filename “#bank” → inode 13

inode 12:
data blocks: 3, 4, 5
refcount: 1

inode 13:
data blocks: 6, 7, 8
refcount: **2**

rename(“#bank”, “bank”)

directory data blocks:
  filename “bank” → inode **13**
  filename “#bank” → inode 13

inode 12:
  data blocks: 3, 4, 5
  refcount: **0**

inode 13:
  data blocks: 6, 7, 8
  refcount: **2**

rename(“#bank”, “bank”)

directory data blocks:
  filename “bank” → inode **13**
  ~~filename “#bank” → inode 13~~

inode 12:
  data blocks: 3, 4, 5
  refcount: **0**

inode 13:
  data blocks: 6, 7, 8
  refcount: **2**

rename(“#bank”, “bank”)

directory data blocks:
  filename “bank” → inode **13**
  ~~filename “#bank” → inode 13~~

inode 12:
  data blocks: 3, 4, 5
  refcount: **0**

inode 13:
  data blocks: 6, 7, 8
  refcount: **1**

Recovery after crash

```
salvage(disk):
   for inode in disk.inodes:
      inode.refcnt =
         find_all_refs(disk.root_dir, inode)

   if exists(“#bank”):
      unlink(“#bank”)
```

6.033: Computer Systems Engineering

Spring 2012

Home / News

Schedule

Submissions

General Information

Staff List

Recitations

TA Office Hours

Discussion / feedback

FAQ

Class Notes Errata

Excellent Writing Examples

2011 Home

Preparation for Recitation 15

Read *A case for redundant arrays of inexpensive disks (RAID)*, by Patterson et al. (Proceedings of the ACM SIGMOD Conference, 1988): (remote copy), (local copy). Read the paper and think of an answer to the following question:

> Modern RAID arrays use parity information and standby disks to provide a highly reliable storage medium even in the face of hardware failures. A highly reliable *system*, however, requires more than just a highly reliable *storage medium*. Consider a networked server handling network transactions (a web server or bank central computer, perhaps). Think about other components of this system whose failure could result in a loss of service. Pick out a few of these and explain how they might be made more reliable in the same way that RAID made disks more reliable. For example, multiple power supplies can be arranged in parallel to power a machine even if one fails. For your examples consider at least one hardware component and one software component of the system.

Read 3/31

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

Top // 6.033 home //

RAID Study

3/31

Raid WP article

RAID 0 Stripe -
splits data accross disks
w/ no parity/redundancy
(new level)
↑ peformance
Since read half from both
depends on data layout

RAID 1 Mirror - exact copy on 2 disks
↑ reliability geometrically (aka exponential)
Can also read half on each
does not depend on data layout
Writes like 1 disk
Can (somehow?) rebuild a mirror w/ changes
-I guess it just writes l → r

RAID 2 Stripes data at bit level
Hamming code for error correction
not used

RAID 3 bytelevel striping w/ dedicated parity ~~[illegible]~~ disk

Very rare in pratice/obselete

must read/write on all disks

so can only do 1 op at a time

good for long sequential reads/writes

all disks spin at same speed

(? so what errors can it fix

- whole disk break
- or a bit wrong)

RAID 4 block-level striping w/ dedicated parity disk

disks can act ind when 1 block needed

(perhaps I don't understand 2,3,4 since they are not actually used - no point)

Very high load on parity disk

RAID 5 - block level striping w/ data over all disk

Popular

4xTB drives -> 3TB + redundency

≥ 3 disks

Concurrent series of blocks = <u>stripe</u>

└ has a parity block

Calcs only the changes

Since parity is sum (xor) of bits

Can ~~rebuild~~ recover from failed disk

└ recover parity blocks and data blocks from disks

↑ (↑ so need to write carefully... ?)

Slow when writes not aligned w/ stripe

└ so some DBAs hate

Sync. angular orentation of disks

Same amt of parity data as RAID 4

limited to smallest sized drive

but as add disks prob of multiple failures ↑

How RAID 5 parity works

XOR

1 xor 1 = 0

1 0 = 1

0 1 = 1 } differences

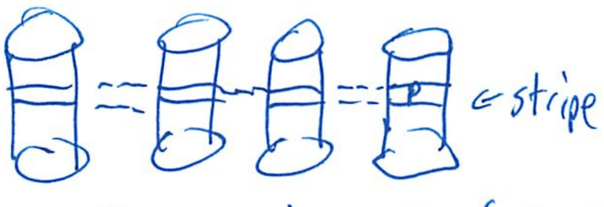
0 0 = 0

basically if sum = even

(Wait!) stripes are accross disks ?

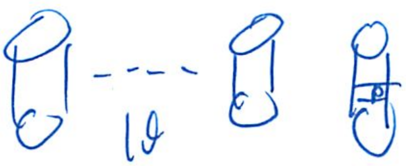
Yes!

Same cylinder accross disks



So can lose 1 of the 4

but can be more



↑ far less stored here $\frac{1}{20}$

but can always rebuild 1 piece

Standard RAID levels

From Wikipedia, the free encyclopedia

Main article: RAID

The **standard RAID levels** are a basic set of RAID configurations and employ striping, mirroring, or parity. The standard RAID levels can be modified for other benefits (*see Nested RAID levels for modes like 1+0 or 0+1*). Other, non-standard RAID levels and non-RAID drive architectures provide alternatives to RAID architectures. RAID levels and their associated data formats are standardised by SNIA (http://www.snia.org/) in the Common RAID Disk Drive Format (DDF) standard (http://www.snia.org/tech_activities/standards/curr_standards/ddf/) .

Contents

RAID 0

A **RAID 0** (also known as a **stripe set** or **striped volume**) splits data evenly across two or more disks (striped) with no parity information for redundancy. RAID 0 was not one of the original RAID levels and provides no data redundancy. RAID 0 is normally used to increase performance, although it can also be used as a way to create a small number of large logical disks out of a large number of small physical ones.

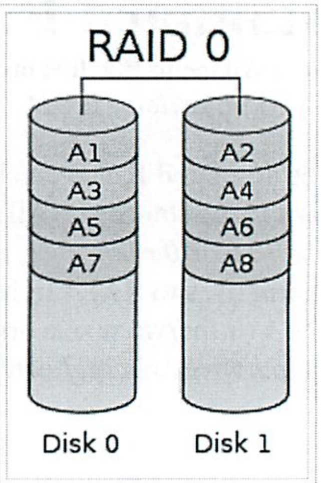


Diagram of a RAID 0 setup.

A RAID 0 can be created with disks of differing sizes, but the storage space added to the array by each disk is limited to the size of the smallest disk. For example, if a 100 GB disk is striped together with a 350 GB disk, the size of the array will be 200 GB.

$$\begin{aligned}\text{Size} &= 2 \cdot \min(100\,\text{GB}, 350\,\text{GB}) \\ &= 2 \cdot 100\,\text{GB} \\ &= 200\,\text{GB}\end{aligned}$$

RAID 0 failure rate

Although RAID 0 was not specified in the original RAID paper, an idealized implementation of RAID 0 would split I/O operations into equal-sized blocks and spread them evenly across two disks. RAID 0 implementations with more than two disks are also possible, though the group reliability decreases with member size.

Reliability of a given RAID 0 set is equal to the average reliability of each disk divided by the number of disks in the set:

$$\text{MTTF}_{\text{group}} \approx \frac{\text{MTTF}_{\text{disk}}}{\text{number}}$$

That is, reliability (as measured by mean time to failure (MTTF) or mean time between failures (MTBF)) is roughly inversely proportional to the number of members – so a set of two disks is roughly half as reliable as a single disk. If there were a probability of 5% that the disk would fail within three years, in a two disk array, that probability would be increased to $\mathbb{P}(\text{at least one fails}) = 1 - \mathbb{P}(\text{neither fails}) = 1 - (1 - 0.05)^2 = 0.0975 = 9.75\,\%$.

The reason for this is that the file system is distributed across all disks. When a drive fails the file system cannot cope with such a large loss of data and coherency since the data is "striped" across all drives (the data cannot be recovered without the missing disk). Data can be recovered using special tools; however, this data will be incomplete and most likely corrupt, and data recovery is typically very costly and not guaranteed.

RAID 0 performance

While the block size can technically be as small as a byte, it is almost always a multiple of the hard disk sector size of 512 bytes. This lets each drive seek independently when randomly reading or writing data on the disk. How much the drives act independently depends on the access pattern from the file system level. For reads and writes that are larger than the stripe size, such as copying files or video playback, the disks will be seeking to the same position on each disk, so the seek time of the array will be the same as that of a single drive. For reads and writes that are smaller than the stripe size, such as database access, the drives will be able to seek independently. If the sectors accessed are spread evenly between the two drives, the apparent seek time of the array will be half that of a single drive (assuming the disks in the array have identical access time characteristics). The transfer speed of the array will be the transfer speed of all the disks added together, limited only by the speed of the RAID controller. Note that these performance scenarios are in the best case with optimal access patterns.

RAID 0 is useful for setups such as large read-only NFS server where mounting many disks is time-consuming or impossible and redundancy is irrelevant.

RAID 0 is also used in some gaming systems where performance is desired and data integrity is not very important. However, real-world tests with games have shown that RAID-0 performance gains are minimal, although some desktop applications will benefit.[1][2] Another article examined these claims and concludes: "Striping does not always increase performance (in certain situations it will actually be slower than a non-RAID setup), but in most situations it will yield a significant improvement in performance." [3]

RAID 1

A **RAID 1** creates an exact copy (or **mirror**) of a set of data on two disks. This is useful when read performance or reliability is more important than data storage capacity. Such an array can only be as big as the smallest member disk. A classic RAID 1 mirrored pair contains two disks (see diagram), which increases reliability geometrically over a single disk. Since each member contains a complete copy of the data, and can be addressed independently, ordinary wear-and-tear reliability is raised by the power of the number of self-contained copies.

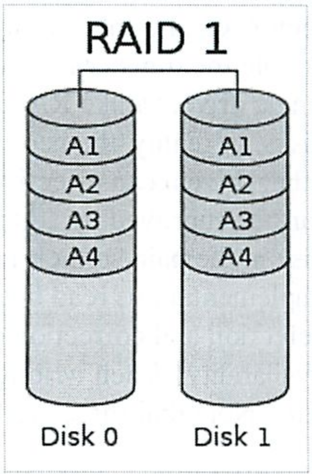


Diagram of a RAID 1 setup

RAID 1 failure rate

As a simplified example, consider a RAID 1 with two identical models of a disk drive with a 5% probability that the disk would fail within three years. Provided that the failures are statistically independent, then the probability of both disks failing during the three year lifetime is 0.25%. Thus, the probability of losing all data is 0.25% over a three year period if nothing is done to the array. If the first disk fails and is never replaced, then there is a 5% chance the data will be lost. If only one of the disks fails, no data would be lost. As long as a failed disk is replaced before the second disk fails, the data is safe.

However, since two identical disks are used and since their usage patterns are also identical, their failures cannot be assumed to be independent. Thus, the probability of losing all data, if the first failed disk is not replaced, may increase.

As a practical matter, in a well-managed system the above is irrelevant because the failed hard drive will not be ignored but will be replaced. The reliability of the overall system is determined by the probability the remaining drive will continue to operate through the repair period, that is the total time it takes to detect a failure, replace the failed hard drive, and for that drive to be rebuilt. If, for example, it takes one hour to replace the failed drive and 9 hours to repopulate it, the overall system reliability is defined by the probability the remaining drive will operate for ten hours without failure.

While RAID 1 can be an effective protection against physical disk failure, it does not provide protection against data corruption due to viruses, accidental file changes or deletions, or any other data-specific changes. By design, any such changes will be instantly mirrored to every drive in the array segment. A virus, for example, that damages data on one drive in a RAID 1 array will damage the same data on all other drives in the array at the same time. For this reason systems using RAID 1 to protect against physical drive failure should also have a traditional data backup process in place to allow data restoration to previous points in time. As this is also the case with other RAID levels, it would seem self-evident that any system critical enough to require disk redundancy also needs the protection of reliable data backups.

RAID 1 performance

Since all the data exist in two or more copies, each with its own hardware, the read performance can go up roughly as a linear multiple of the number of copies. That is, a RAID 1 array of two drives can be reading in two different places at the same time, though not all implementations of RAID 1 do this.[4] To maximize performance benefits of RAID 1, independent disk controllers are recommended, one for each disk. Some refer to this practice as **splitting** or **duplexing** (for two disk arrays) or **multiplexing** (for arrays with more than two disks). When reading, both disks can be accessed independently and requested sectors can be split evenly between the disks. For the usual mirror of two disks, this would, in theory, double the transfer rate when reading. The apparent access time of the array would be half that of a single drive. Unlike RAID 0, this would be for all access patterns, as all the data are present on all the disks. In reality, the need to move the drive heads to the next block (to skip blocks already read by the other drives) can effectively mitigate speed advantages for sequential access. Read performance can be further improved by adding drives to the mirror. Many older IDE RAID 1 controllers read only from one disk in the pair, so their read performance is always that of a single disk. Some older RAID 1 implementations read both disks simultaneously to compare the data and detect errors. The error detection and correction on modern disks makes this less useful in environments requiring normal availability. When writing, the array performs like a single disk, as all mirrors must be written with the data. Note that these are best case performance scenarios with optimal access patterns.

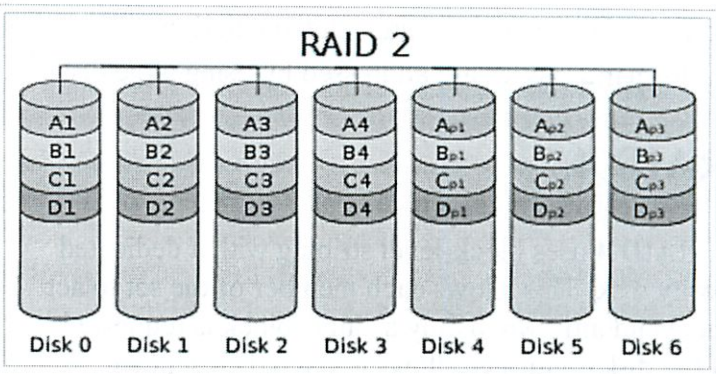
RAID 1 has many administrative advantages. For instance, in some environments, it is possible to "split the mirror," declare one disk as inactive, do a backup of that disk, then "rebuild" the mirror. This is useful in situations where the file system must be constantly available. This requires that the application supports recovery from the image of data on the disk at the point of the mirror split. This procedure is less critical in the presence of the "snapshot" feature of some file systems, in which some space is reserved for changes, presenting a static point-in-time view of the file system. Alternatively, a new disk can be substituted so that the inactive disk can be kept in much the same way as traditional backup. To maintain redundancy during the backup process, some controllers support adding a third disk to an active pair. After the third disk rebuild completes, it is made inactive and backed up as described above.

RAID 2

A **RAID 2** stripes data at the bit (rather than block) level, and uses Hamming code for error correction. The disks are synchronized by the controller to spin at the same angular orientation (they reach Index at the same time). Extremely high data transfer rates are possible. This is the only original level of RAID that is not currently used.[5][6]

The use of Hamming(7,4) code (four data bits plus three parity bits) also permits using seven disks in RAID 2, with four being used for data storage and three being used for error correction.

RAID 2 is the only standard RAID level, other than some implementations of RAID 6, which can automatically recover accurate data from single-bit corruption in data. Other RAID levels can detect single-bit corruption in data, or can sometimes reconstruct missing data, but cannot reliably resolve contradictions between parity bits and data bits without human intervention.



RAID Level 2

(Multiple-bit corruption is possible, though extremely rare. RAID 2 can detect, but not repair, double-bit corruption.)

All hard disks eventually implemented Hamming code error correction. This made RAID 2 error correction redundant and unnecessarily complex. Like RAID 3, this level quickly became useless and is now obsolete. There are no commercial applications of RAID 2.[5][6]

RAID 3

A **RAID 3** uses bit-level striping with a dedicated parity disk. RAID 3 is very rare in practice. One of the characteristics of RAID 3 is that it generally cannot service multiple requests simultaneously. This happens because any single block of data will, by definition, be spread across all members of the set and will reside in the same location. So, any I/O operation requires activity on every disk and usually requires synchronized spindles.

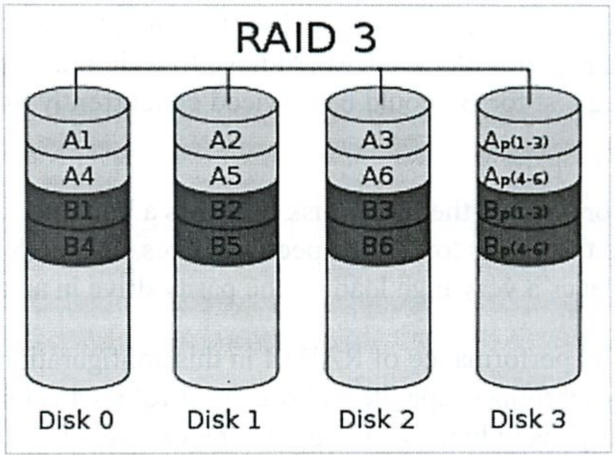


Diagram of a RAID 3 setup of 6-byte blocks and two parity bytes, shown are two blocks of data in different colors.

In our example, a request for block "A" consisting of bytes A1-A6 would require all three data disks to seek to the beginning (A1) and reply with their contents. A simultaneous request for block B would have to wait.

However, the performance characteristic of RAID 3 is very consistent, unlike that for higher RAID levels. The size of a stripe is less than the size of a sector or OS block. As a result, reading and writing accesses the entire stripe every time. The performance of the array is therefore identical to the performance of one disk in the array except for the transfer rate, which is multiplied by the number of data drives less the parity drives.

This makes it best for applications that demand the highest transfer rates in long sequential reads and writes, for example uncompressed video editing. Applications that make small reads and writes from random disk locations will get the worst performance out of this level.[6]

The requirement that all disks spin synchronously, aka lockstep, added design considerations to a level that didn't give significant advantages over other RAID levels, so it quickly became useless and is now obsolete.[5] Both RAID 3 and RAID 4 were quickly replaced by RAID 5.[7] However, this level has commercial vendors making implementations of it. It's usually implemented in hardware, and the performance issues are addressed by using large disks.[6]

RAID 4

A **RAID 4** uses block-level striping with a dedicated parity disk. This allows each member of the set to act independently when only a single block is requested. If the disk controller allows it, a RAID 4 set can service multiple read requests simultaneously. RAID 4 looks similar to RAID 5 except that it does not use distributed parity, and similar to RAID 3 except that it stripes at the block level, rather than the bit level. Generally, RAID 4 is implemented with hardware support for parity calculations, and a minimum of three disks is required for a complete RAID 4 configuration.

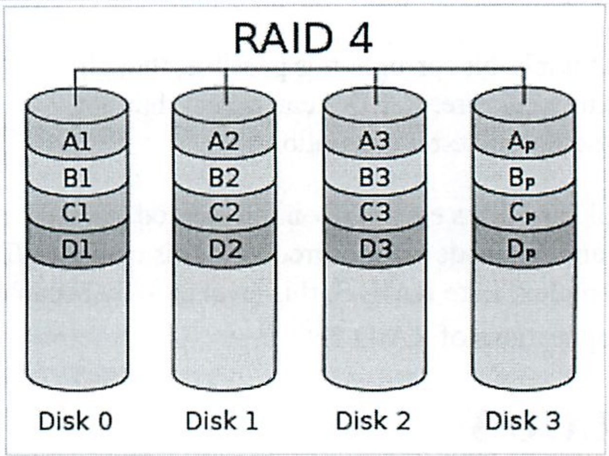


Diagram of a RAID 4 setup with dedicated parity disk with each color representing the group of blocks in the respective parity block (a stripe)

In the example on the right, a read request for block A1 would be serviced by disk 0. A simultaneous read request for block B1 would have to wait, but a read request for B2 could be serviced concurrently by disk 1.

For writing, the parity disk becomes a bottleneck, as simultaneous writes to A1 and B2 would, in addition to the writes to their respective drives, also both need to write to the parity drive. In this way RAID 4 places a very high load on the parity drive in an array.

The performance of RAID 4 in this configuration can be very poor, but unlike RAID 3 it does not need synchronized spindles. However, if RAID 4 is implemented on synchronized drives and the size of a stripe is reduced below the OS block size a RAID 4 array then has the same performance pattern as a RAID 3 array.

Currently, RAID 4 is only implemented at the enterprise level by one company, NetApp. The aforementioned performance problems were solved with their proprietary Write Anywhere File Layout (WAFL), an approach to writing data to disk locations that minimizes the conventional parity RAID write penalty. By storing system metadata (inodes, block maps, and inode maps) in the same way application data is stored, WAFL is able to write file system metadata blocks anywhere on the disk. This approach in turn allows multiple writes to be "gathered" and scheduled to the same RAID stripe—eliminating the traditional read-modify-write penalty prevalent in parity-based RAID schemes. http://partners.netapp.com/go/techontap/matl/NetApp_DNA.html

Both RAID 3 and RAID 4 were quickly replaced by RAID 5.[7]

RAID 5

A **RAID 5** uses block-level striping with parity data distributed across all member disks. RAID 5 has achieved popularity because of its low cost of redundancy. This can be seen by comparing the number of drives needed to achieve a given capacity. For an array of n drives, with S_{min} being the size of the smallest disk in the array, other RAID levels that yield redundancy give only a storage capacity of S_{min} (for RAID 1), or $S_{\mathrm{min}} \times (n/2)$ (for RAID 1+0). In RAID 5, the yield is $S_{\mathrm{min}} \times (n-1)$. For example, four 1 TB drives can be made into two separate 1 TB redundant arrays under RAID 1 or 2 TB under RAID 1+0, but the same four drives can be used to build a 3 TB array under RAID 5. Although RAID 5 may be implemented in a disk controller, some have hardware support for parity calculations (hardware RAID cards with onboard processors) while some use the main system processor (a form of software RAID in vendor drivers for inexpensive controllers). Many operating systems also provide software RAID support independently of the disk controller, such as Windows *Dynamic Disks*, Linux mdadm, or RAID-Z. In most implementations, a minimum of three disks is required for a complete RAID 5 configuration. In some implementations a degraded RAID 5 disk set can be made (three disk set of which only two are online), while mdadm supports a fully functional (non-degraded) RAID 5 setup with two disks - which functions as a slow RAID-1, but can be expanded with further volumes.

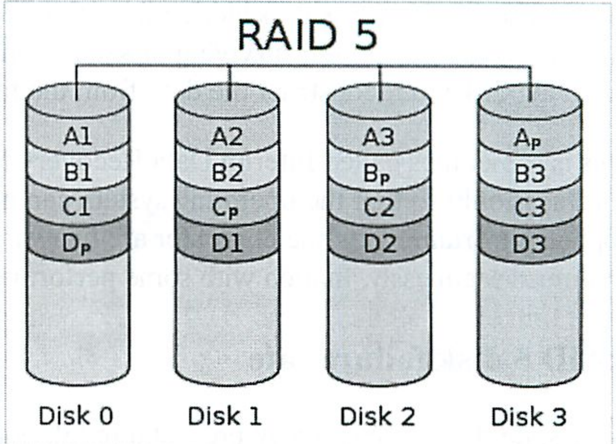


Diagram of a RAID 5 setup with distributed parity with each color representing the group of blocks in the respective parity block (a stripe). This diagram shows left asymmetric algorithm

In the example, a read request for block A1 would be serviced by disk 0. A simultaneous read request for block B1 would have to wait, but a read request for B2 could be serviced concurrently by disk 1.

RAID 5 parity handling

A concurrent series of blocks - one on each of the disks in an array - is collectively called a **stripe**. If another block, or some portion thereof, is written on that same stripe, the parity block, or some portion thereof, is recalculated and rewritten. For small writes, this requires:

- Read the old data block
- Read the old parity block
- Compare the old data block with the write request. For each bit that has flipped (changed from 0 to 1, or from 1 to 0) in the data block, flip the corresponding bit in the parity block
- Write the new data block
- Write the new parity block

The disk used for the parity block is staggered from one stripe to the next, hence the term **distributed parity blocks**. RAID 5 writes are expensive in terms of disk operations and traffic between the disks and the controller.

The parity blocks are not read on data reads, since this would add unnecessary overhead and would diminish performance. The parity blocks are read, however, when a read of blocks in the stripe fails due to failure of any one of the disks, and the parity block in the stripe are used to reconstruct the errant sector. The CRC error is thus hidden from the main computer. Likewise, should a disk fail in the array, the parity blocks from the surviving disks are combined mathematically with the data blocks from the surviving disks to reconstruct the data from the failed drive on-the-fly.

This is sometimes called Interim Data Recovery Mode. The computer knows that a disk drive has failed, but this is only so that the operating system can notify the administrator that a drive needs replacement; applications running on the computer are unaware of the failure. Reading and writing to the drive array continues seamlessly, though with some performance degradation.

RAID 5 disk failure rate

Solid-state drives (SSDs) may present a revolutionary instead of evolutionary way of dealing with increasing RAID-5 rebuild limitations. With encouragement from many flash-SSD manufacturers, JEDEC is preparing to set standards in 2009 for measuring UBER (uncorrectable bit error rates) and "raw" bit error rates (error rates before ECC, error correction code).[8]

But even the economy-class Intel X25-M SSD claims an unrecoverable error rate of 1 sector in 10^{15} bits and an MTBF of two million hours.[9] Ironically, the much-faster throughput of SSDs (STEC claims its enterprise-class Zeus SSDs exceed 200 times the transactional performance of today's 15k-RPM, enterprise-class HDDs)[10] suggests that a similar error rate (1 in 10^{15}) will result a two-magnitude shortening of MTBF.

In the event of a system failure while there are active writes, the parity of a stripe may become inconsistent with the data. If this is not detected and repaired before a disk or block fails, data loss may ensue as incorrect parity will be used to reconstruct the missing block in that stripe. This potential vulnerability is sometimes known as the **write hole**. Battery-backed cache and similar techniques are commonly used to reduce the window of opportunity for this to occur. The same issue occurs for RAID-6.

RAID 5 performance

RAID 5 implementations suffer from poor performance when faced with a workload that includes many writes that are not aligned to stripe boundaries, or are smaller than the capacity of a single stripe. This is because parity must be updated on each write, requiring read-modify-write sequences for both the data block and the parity block. More complex implementations may include a non-volatile write back cache to reduce the performance impact of incremental parity updates. Large writes, spanning an entire stripe width, can however be done without read-modify-write cycles for each data + parity block but *only if they are stripe aligned*, by simply overwriting the parity block with the computed parity since the new data for each data block in the stripe is known in its entirety at the time of the write. This is sometimes called a *full stripe write*.

Random write performance is poor, especially at high concurrency levels common in large multi-user databases. The read-modify-write cycle requirement of RAID 5's parity implementation penalizes random writes by as much as an order of magnitude compared to RAID 0.[11]

Performance problems can be so severe that some database experts have formed a group called BAARF — the Battle Against Any Raid Five.[12]

The read performance of RAID 5 is almost as good as RAID 0 for the same number of disks. Except for the parity blocks, the distribution of data over the drives follows the same pattern as RAID 0. The reason RAID 5 is slightly slower is that the disks must skip over the parity blocks.

RAID 5 latency

When a disk record is randomly accessed there is a delay as the disk rotates sufficiently for the data to come under the head for processing. This delay is called *latency*. On average, a single disk will need to rotate 1/2 revolution. Thus, for a 7200 RPM disk the average latency is 4.2 milliseconds. In RAID 5 arrays all the disks must be accessed so the latency can become a significant factor. In a RAID 5 array, with n randomly oriented disks, the average latency becomes 1-(2^{-n}) revolutions.[*citation needed*] In order to mitigate this problem, well-designed RAID systems will synchronize the angular orientation of their disks. In this case the random nature of the angular displacements goes away, the average latency returns to 1/2 revolution, and a savings of up to 50% in latency is achieved. Since solid state drives do not have disks, their latency does not follow this model.

Effect of Angular Synchronization

Number of Disks	Average Randomly Oriented Latency (Revolutions)
1	0.500
2	0.750
3	0.875
4	0.938
5	0.969
6	0.984
7	0.992
8	0.996

RAID 5 usable size

Parity data uses up the capacity of one drive in the array. (This can be seen by comparing it with RAID 4: RAID 5 distributes the parity data across the disks, while RAID 4 centralizes it on one disk, but the amount of parity data is the same.) If the drives vary in capacity, the smallest one sets the limit. Therefore, the usable capacity of a RAID 5 array is $(N-1) \cdot S_{\mathrm{min}}$, where N is the total number of drives in the array and S_{min} is the capacity of the smallest drive in the array.

The number of hard disks that can belong to a single array is limited only by the capacity of the storage controller in hardware implementations, or by the OS in software RAID. One caveat is that unlike RAID 1, as the number of disks in an array increases, the probability of data loss due to multiple drive failures also increases. This is because there is a reduced ratio of "losable" drives (the number of drives that can fail before data is lost) to total drives.[*citation needed*]

RAID 6

Redundancy and data loss recovery capability

RAID 6 extends RAID 5 by adding an additional parity block; thus it uses block-level striping with two parity blocks distributed across all member disks.

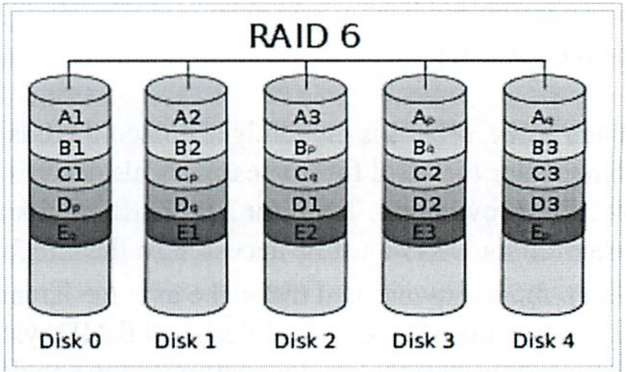


Diagram of a RAID 6 setup, which is identical to RAID 5 other than the addition of a second parity block

Performance (speed)

RAID 6 does not have a performance penalty for read operations, but it does have a performance penalty on write operations because of the overhead associated with parity calculations. Performance varies greatly depending on how RAID 6 is implemented in the manufacturer's storage architecture – in software, firmware or by using firmware and specialized ASICs for intensive parity calculations. It can be as fast as a RAID-5 system with one fewer drive (same number of data drives).[13]

Efficiency (potential waste of storage)

RAID 6 is no more space inefficient than RAID 5 with a hot spare drive when used with a small number of drives, but as arrays become bigger and have more drives, the loss in storage capacity becomes less important, although the probability of data loss is greater with larger arrays. RAID 6 provides protection against data loss during an array rebuild, when a second drive is lost, a bad block read is encountered, or when a human operator accidentally removes and replaces the wrong disk drive when attempting to replace a failed drive.

The usable capacity of a RAID 6 array is $(N-2) \cdot S_{\text{min}}$, where N is the total number of drives in the array and S_{min} is the capacity of the smallest drive in the array.

Implementation

According to the Storage Networking Industry Association (SNIA), the definition of RAID 6 is: "Any form of RAID that can continue to execute read and write requests to all of a RAID array's virtual disks in the presence of any two concurrent disk failures. Several methods, including dual check data computations (parity and Reed-Solomon), orthogonal dual parity check data and diagonal parity, have been used to implement RAID Level 6."[14]

Computing parity

Two different *syndromes* need to be computed in order to allow the loss of any two drives. One of them, **P** can be the simple XOR of the data across the stripes, as with RAID 5. A second, independent syndrome is more complicated and requires the assistance of field theory.

To deal with this, the Galois field $GF(m)$ is introduced with $m = 2^k$, where

$GF(m) \cong F_2[x]/(p(x))$ for a suitable irreducible polynomial $p(x)$ of degree k. A chunk of data can be written as $d_{k-1}d_{k-2}...d_0$ in base 2 where each d_i is either 0 or 1. This is chosen to correspond with the element $d_{k-1}x^{k-1} + d_{k-2}x^{k-2} + ... + d_1x + d_0$ in the Galois field. Let $D_0, ..., D_{n-1} \in GF(m)$ correspond to the stripes of data across hard drives encoded as field elements in this manner (in practice they would probably be broken into byte-sized chunks). If g is some generator of the field and $\oplus$ denotes addition in the field while concatenation denotes multiplication, then $\mathbf{P}$ and $\mathbf{Q}$ may be computed as follows (n denotes the number of data disks):

$$\mathbf{P} = \bigoplus_i D_i = \mathbf{D}_0 \oplus \mathbf{D}_1 \oplus \mathbf{D}_2 \oplus ... \oplus \mathbf{D}_{n-1}$$

$$\mathbf{Q} = \bigoplus_i g^i D_i = g^0\mathbf{D}_0 \oplus g^1\mathbf{D}_1 \oplus g^2\mathbf{D}_2 \oplus ... \oplus g^{n-1}\mathbf{D}_{n-1}$$

For a computer scientist, a good way to think about this is that $\oplus$ is a bitwise XOR operator and g^i is the action of a linear feedback shift register on a chunk of data. Thus, in the formula above,[15] the calculation of **P** is just the XOR of each stripe. This is because addition in any characteristic two finite field reduces to the XOR operation. The computation of **Q** is the XOR of a shifted version of each stripe.

Mathematically, the *generator* is an element of the field such that g^i is different for each nonnegative i satisfying $i < n$.

If one data drive is lost, the data can be recomputed from **P** just like with RAID 5. If two data drives are lost or the drive containing **P** is lost the data can be recovered from **P** and **Q** using a more complex process. Working out the details is extremely hard with field theory, and the lack of explanation here isn't making it any easier. Suppose that D_i and D_j are the lost values with $i \neq j$. Using the other values of D, constants A and B may be found so that $D_i \oplus D_j = A$ and $g^i D_i \oplus g^j D_j = B$. Multiplying both sides of the latter equation by g^{n-i} and adding to the former equation yields $(g^{n-i+j} \oplus 1)D_j = g^{n-i}B \oplus A$ and thus a solution for D_j, which may be used to compute D_i.

The computation of **Q** is CPU intensive compared to the simplicity of **P**. Thus, a RAID 6 implemented in software will have a more significant effect on system performance, and a hardware solution will be more complex.

Non-standard RAID levels and non-RAID drive architectures

Main articles: Non-standard RAID levels and Non-RAID drive architectures

There are other RAID levels that are promoted by individual vendors, but not generally standardized. The non-standard RAID levels 5E, 5EE and 6E extend RAID 5 and 6 with hot-spare drives.

Other non-standard RAID levels include:

- RAID 1.5,
- RAID 7 (a hardware-supported, proprietary RAID developed by Storage Computer Corp. of Nashua, NH),
- RAID-DP,

- RAID S or parity RAID,
- Matrix RAID,
- RAID-K,
- RAID-Z,
- RAIDn,
- Linux MD RAID 10,
- IBM ServeRAID 1E,
- unRAID,
- ineo Complex RAID,
- Drobo BeyondRAID, and
- Microsoft Storage Spaces.

There are also non-RAID drive architectures, which are referred to by similar acronyms, notably SLED, Just a Bunch of Disks, SPAN/BIG, and MAID.

See also

- RAID
- Nested RAID levels
- Non-standard RAID levels
- Non-RAID drive architectures

References

1. ^ "Western Digital's Raptors in RAID-0: Are two drives better than one?" (http://www.anandtech.com/storage/showdoc.aspx?i=2101) . AnandTech. July 1, 2004. http://www.anandtech.com/storage/showdoc.aspx?i=2101. Retrieved 2007-11-24.
2. ^ "Hitachi Deskstar 7K1000: Two Terabyte RAID Redux" (http://www.anandtech.com/storage/showdoc.aspx?i=2974) . AnandTech. April 23, 2007. http://www.anandtech.com/storage/showdoc.aspx?i=2974. Retrieved 2007-11-24.
3. ^ "RAID 0: Hype or blessing?" (http://tweakers.net/reviews/515/1/raid-0-hype-or-blessing-pagina-1.html) . Tweakers.net. August 7, 2004. http://tweakers.net/reviews/515/1/raid-0-hype-or-blessing-pagina-1.html. Retrieved 2008-07-23.
4. ^ "Mac OS X, Mac OS X Server: How to Use Apple-Supplied RAID Software" (http://docs.info.apple.com/article.html?artnum=106594) . Apple.com. http://docs.info.apple.com/article.html?artnum=106594. Retrieved 2007-11-24.
5. ^ a b c Derek Vadala (2003). *Managing RAID on Linux. O'Reilly Series* (http://books.google.com/?id=RM4tahggCVcC&pg=PA6&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation) (illustrated ed.). O'Reilly. p. 6. ISBN 1565927303, 9781565927308. http://books.google.com/?id=RM4tahggCVcC&pg=PA6&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation.
6. ^ a b c d Evan Marcus, Hal Stern (2003). *Blueprints for high availability* (http://books.google.com/?id=D_jYqFoJVEAC&pg=RA2-PA167&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation) (2, illustrated ed.). John Wiley and Sons. p. 167. ISBN 0471430269, 9780471430261. http://books.google.com/?id=D_jYqFoJVEAC&pg=RA2-PA167&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation.
7. ^ a b Michael Meyers, Scott Jernigan (2003). *Mike Meyers' A+ Guide to Managing and Troubleshooting PCs* (http://books.google.com/?id=9vfQKUT_BjgC&pg=PT348&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation) (illustrated ed.). McGraw-Hill Professional. p. 321. ISBN 0072231467, 9780072231465. http://books.google.com/?id=9vfQKUT_BjgC&pg=PT348&dq=raid+2+implementation#v=onepage&q=raid%202%20implementation.

6.033 Recitation

4/5

RAID

Does anyone use it? Oh yes!

MIT Startup: Thinking Machines

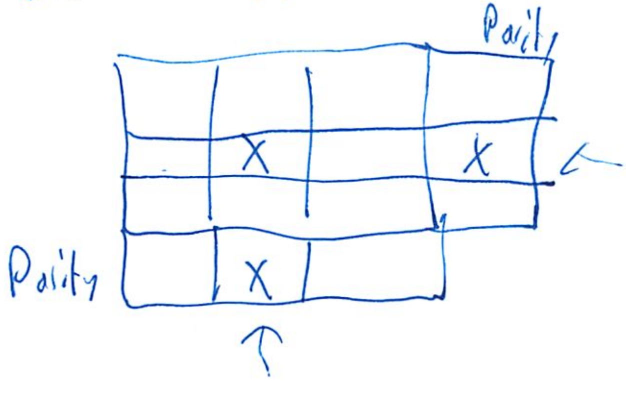
30 years ago

Very parallel machines

Explaining xor to the trial

Parity, ECC, etc

Bits of data in matrix



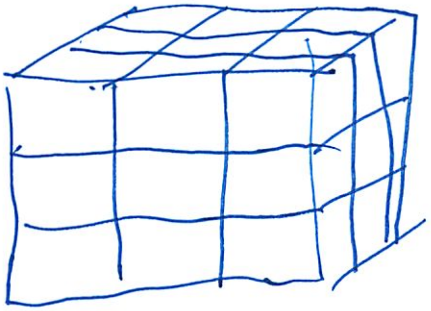
2 mistakes

shows where it is

2 dimension - can detect, can't fix

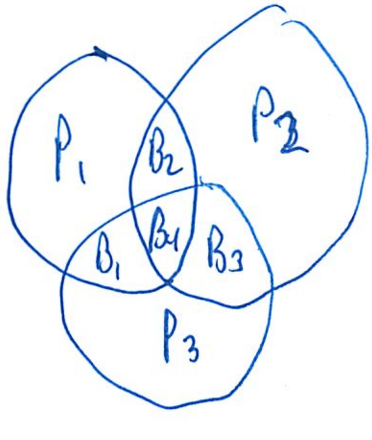
Double Error Correction

3D Matrix



and 3 rows of parity bits

Hamming Code



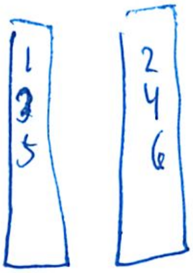
3 data bits
7 bits total

Single error correction
Hamming distance

(3)

Types of RAID

RAID 0



~~Do [illegible] reading in parallel~~

Way drives work

bits per track same

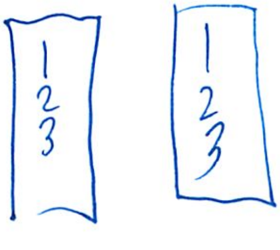
every cyl has same data

So angular read is same

Wasteful, but want fast

Newer tech might write more to outside

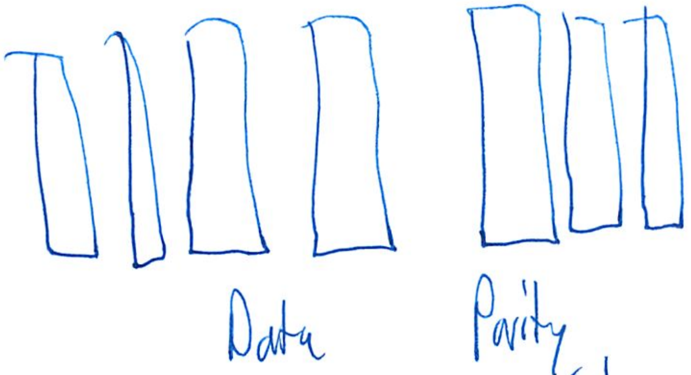
RAID 1



do reading in parallel
(take the 1st response

(4)

RAID 2



Corect any poblem

≈ log n # of parity disks ±1

bit striped

└ bitwise

Could do bit
block
Sector

better Large # of disks

→ parity covers every power of 2

must do this one in HW

Synced spindles

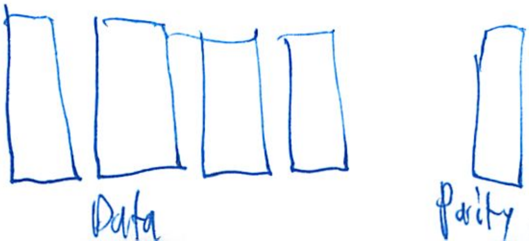
No one anymore does it

RAID 3

byte ~~[illegible]~~ ~~higher~~ striping

You assume you know which disk fails

1 parity disk for any # data disks



Synced spindals

Reliability ↓ as # disks ↑

MTTF

MTTR

Where did they get that?

1 hr - need a maint person w/ spare disks

but it also needs to rebuild!

Rebuilding

He did internship at NetApp

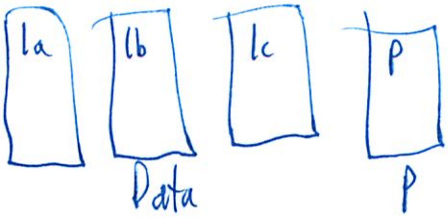
1000 or 10,000 disks

Every 20 min something repairing

RAID 4

Now by sectors

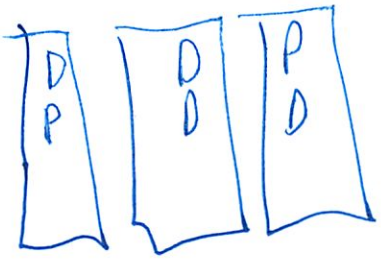
No synced spindals



6

But parity disk always being accessed

RAID 5



← normalized, known where parity bits are

Doesn't matter which data goes where
But parity bit does compute parity for
certain sectors

If you can batch writes, so always fill stripes
RAID 4 is just as good
Eaiser to add disks to RAID 4

①

What's new?

Google found Actual MTTF lower than claimed

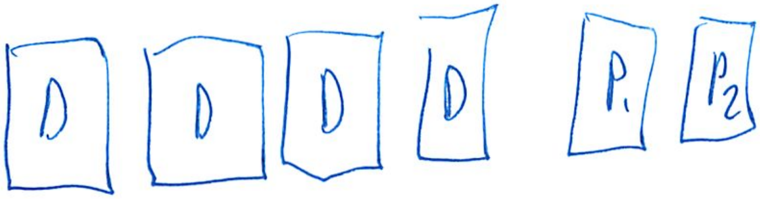
Not much diff for ent. discs

What happens if 2 drives fail
- problem!

Buying disks from same manuf risky

¿Could add 1 more parity disk to deal w/ 2 failures
└ No

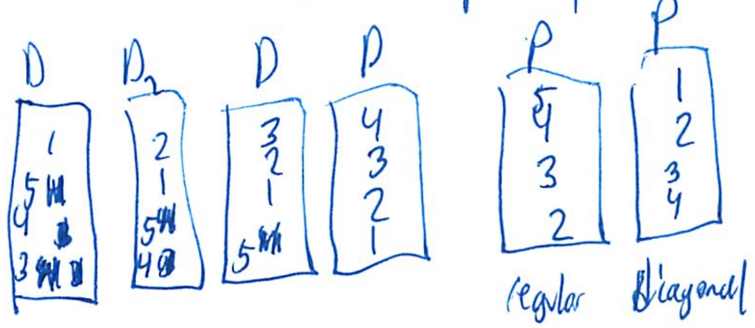
Want



took people years to figre this out

8

Compute a diagonal parity



↑ each bit present 4 times

If 2 disks fail
Can use combo of 2 disks to help

Recovery 2 on D_2 first, Then Recover 1 on D_1

Just 1 way to do this

each vendor does it diff

RAID 6

Can easily add P_2 when needed

Otherwise completly vunerable while rebuilding

Redundent array of inexpensive —— usually works very well

6.033 DP Tutorial

4/6

Need to pick a team member for DP2

DP1 Debrief

Really good

Much better than memo

But have better captions

Cross reference to image

- not below
- instead Figure 1
- figure could still update
- don't hash the figure #s

Travis

Larry's grading

People liked it

- better than past

Put some #s in perfomace section

Dave

~~Ana~~ All reports are above average

Figures in tech documents are like cartoons in the New Yorker

Needs a Punch Line

Guides how see picture

Lots of problems placing pictures

CI Trajectory

You have to come find Linda + Dave

6.UAT - oral presentation

Can revise DP1

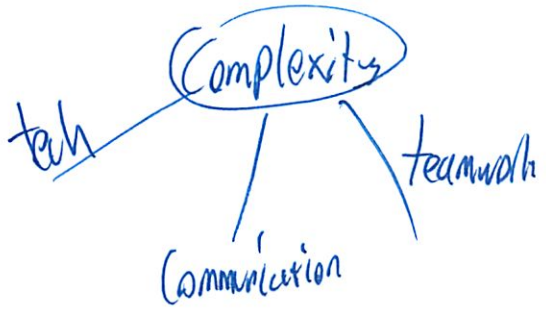
Writing component only

Max grade B+

③

Teamwork

3 legged stool



How do you handle document management?

Don't just staple together

Will do it a lot in real life

Appoint Someone combiner, and editor — Detail oriented indv.

— makes it look like 1 doc

— Spell words the same way

— hyphinate the same way

— (ch searchs for that word

— All use same software

—

Problems w/ teams in the past

(I'm on 4 group projects - don't need a lecture on team work)

Ind effort → Collaberation

teams are built on ind effort

Problems

1. People too busy - work on diff time frames
2. don't communicate
3. don't take serious
   - └ no indv effort
4. Comm ~~[illegible]~~ diff understandings

Teams that don't work

Push teams until they break

Can rebuild stronger

Isreli army - teams can't just not perform

Relt

Some people don't read email on Sat for religious reasons

↓

they need to tell their team that

Scheduling issues

└ #1 problem in academia

Working style

- procrastination brinkmenship

Quality expectation

6

"Leadership"

Lack of accountability

- assign stuff to people

Trust

Freeloaders

Unwilling to compromise

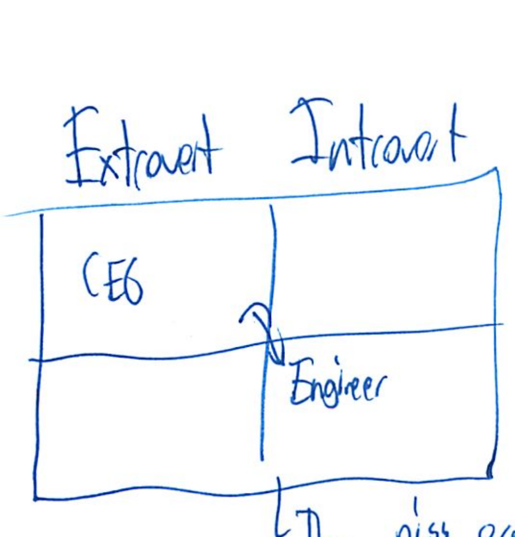
- Heartburn
- Ego
- Control Freak

Division of work

Compartmentalize

Abstraction

Don't micro manage other teams

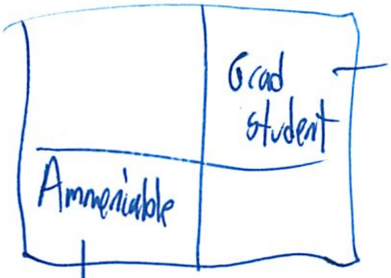


Extrovert	Introvert	
CEO		Product/Goal
	Engineer	Process

They piss each other off

Both people need to be made happy

- avoid ad hominen attacks



Works on own

Want things to go well

Smooths over conflict

Don't go to your boss about problems

Travis does not want to hear

Team Peer Eval Questionaire

As a team do get to know you meetings

Make little decisions

eat ice creame

Pratice

Where are we going to eat ice cream

Address problems pronto

Things to Know About Working in Teams

Choosing teammates:

Choose teams to avoid these failure modes:

- Orthogonal schedules—members do not have common times to meet; consider both weekly and semester schedules. Meeting times are more challenging in the academic setting than in industry.
- Orthogonal objectives and expectations—members do not share expectations of effort and quality. Your best friend or roommate may make a poor teammate.
- Orthogonal text editors—this constraint seems trivial, but can be a barrier to success. It is OK for members to work in different text editors as long as the collation of individual work is arranged well in advance.

A variety of backgrounds and working styles is an advantage in a team in terms of the quality of the final product though such variety can generate additional heartburn. Chances are that the "variety" range in a 6.033 recitation is narrow. Expect the process oriented introvert and the product oriented extrovert to give each other heartburn; but both product and process must be attended to in order to succeed.

Starting out:

- Don't allow yourself to be left out at the start; make sure to attend the first team meetings/activities. Corollary: Don't schedule first team meetings at times that might be difficult for some members, for example 8:00 AM on Friday morning or "I think I can make it if we get back from the track meet in time."
- Consider a "get to know yah" activity or two before actually addressing any technical/design issues—not the first inclination of many an MIT student. (Build a fort; go bowling; visit the MFA, take the CATME sample survey; do something that requires you to make decisions as a team.)
- Identify the amenables.
- Nurture trust.

Contract:

As one of the first team activities, consider arriving at consensus on a mission statement—what your team's objective is and how you will accomplish this objective—and a contract—an outline of the expectations for conduct. These elements have counterparts in the corporate world; the need is similar—if expectations are clearly identified and shared then there is a higher probability that reality will match the expectation.

Meetings:

Schedule meetings long before you hold them; announce an agenda before the meeting. Prepare for meetings. During meetings, make sure 3 roles are embodied: time keeper, moderator, & note-taker. Consider trading these roles in different meetings. Make sure everyone speaks—continually. Be aware of "pause duration"—the amount of time different participants wait until feeling comfortable speaking. At the end of meetings, write and distribute the deliverables expected of each participant; allow all to have 2nd thoughts (for a brief period) after a meeting.

Forming, Storming, Norming, and Performing (and Adjourning)

Teams generally progress through stages. High performing teams rarely skip the storming stage—in which individual members find themselves outside of their comfort zones. If things go well, the heartburn of the storming stage is more than compensated for by the quality of the final product and the development of individuals. To reduce the symptoms of heartburn, separate people from their ideas; criticize the ideas, not the teammate.

5 Dimensions of Team Performance:

There are five dimensions along which you can contribute to your team:

- Skills and experience
- Doing work
- Expecting quality
- Keeping the team on track
- Playing well with others

Consider taking the CATME sample instrument survey together as a team:
https://www.catme.org/login/survey_instructions

compiled by dave custer draft of 4/6/2012
sources include: http://en.wikipedia.org/wiki/Tuckman%27s_stages_of_group_development, https://engineering.purdue.edu/CATME, http://www4.ncsu.edu/unity/lockers/users/f/felder/public/Papers/Oakley-paper%28JSCL%29.pdf

6.033, Spring 2012

6 April, 1:00/2:00 (circle one)

Final Writing Tutorial

Please leave this with your TA. Neither your name nor any other identifying information should be written on this sheet.

My writing program lecturer has been most helpful to me in: ____________________________________

____

____

It would have been helpful to me if s/he had also: ______________________________________

____

____

Tear Here----------------------------------Tear Here-------------------------------------Tear Here-------------------------------------Tear Here

KEEP THIS INFORMATION ON REVISING DP1

You may revise DP1 if your grade was under a B+. Revision grade will not be higher than B+.

If you wish to revise, make an appointment to speak with your writing program lecturer no later than one week after you receive your writing grade for DP1.

1. You will agree on a section to revise; the grade for the revision of that section will replace your writing grade for DP1.

2. You will agree on a due date for that revision. Our aim is to provide enough time so that you can have the revised grade before drop date.

3. You will not receive much written feedback on the revision, but you are encouraged to discuss any questions you have with your writing program lecturer.

4. Your recitation instructor *must* approve the completed revision (to make sure no technical content has been broken in the process). Only your writing grade is changed by the revision, however.

Managing Collaborative Documents

With their potential to optimize people's skills, collaborative documents can have greater depth. It takes management, though, to keep individual skills from showing up as assorted writing, formatting and graphic styles. The most professional collaborative documents embody the myth -- the myth of the single author. Here's a way to approach making that happen.

Step 1: Centralize

Appoint an Editorial Czar, empowered to decide how the document will treat matters with the potential for inconsistency such as whether to capitalize the name of the design, whether to spell a word like trade-off with or without the hyphen, and whether to use *Webster's* or *American Heritage Dictionary* for spelling. Rely on a dictionary; do not take a poll for spelling. The Editorial Czar needs to be someone who is very detail oriented (it's where the devil resides).

Step 2: Construct

Everyone needs to reread The 6.033 Style Guide and direct questions to the Editorial Czar. You may want to create an Addendum to that guide for your document. For example, you may want to create a table showing the spec for your document's body text (margin, font and size), headings (font, size and placement by level), paragraphs (how much to indent and what amount of space to use between paragraphs), figure and table labels, numbers, titles and captions (Arabic numbers, capitalization of titles, font size, style, color) and even table characteristics (font, column justification, fonts for headings, use of lines). Will you use pseudocode and if so how will you spec the resulting figure? Will you have any graphs and if so, how will you spec the graphic field, the axes, their labels, their legends?

Step 3: Agree

Agree on software and set up any functionality it may have so everyone uses the same style.

LaTeX Example: Consider using the LaTex template created by a 6.033 student for DP1: http://benjamin.barenblat.name/projects/6033dp1/. This template closely mirrors the 6.033 style guide. To use the class file, download it and include \documentclass[strict]{6033dp1} at the start of your TeX file.

Word Example: Word lets you spec font for the body text, headings, figure and table numbers, labels, titles and captions. It also lets you specify that your references are to be cited using IEEE style. If you use the full functionality of that software, you can reorder figures and tables and it will update your label numbers and any cross references to them in your text. The citation software will similarly let you insert a reference and it will renumber both your in-text citations and your reference list.

None of that is to say you do not have to double check. Remember Murphy's Law.

Step 4: Empower

Provide the Editorial Czar with a copy of the file as you work on it so that person can use the comment and track changes function to modify your work as you go. By doing that, the Editorial Czar can make and maintain a list of emerging inconsistencies, resolve them as much as possible, and at the very end run searches for them. For example, it is likely that "data" will be misused and the phrase "this data is" appear. By running a search for "data," the Czar can find and repair so that only the expression "these data are" lands in the final document.

6.033 Tutorial

Design Reports and Other Advice

Attendees

Travis Grusecki, TA for 6.033 Sections 1 and 2

Section 1 students and their writing instructor: Dave Custer

Section 2 students and their writing instructor, Linda Sutliff

Friday, April 6, 2012, 1:00 p.m. to 2:00 p.m. and 2:00 p.m. to 3:00 p.m., Room 26-153

Agenda

Topic	Lead
A. **Debriefing on Design Reports**	Custer, Grusecki and Sutliff (5 minutes)
B. **Questions on Design Report Revisions**[1]	Custer and Sutliff (2 minutes)
C. **Questions on Schedule and CI involvement thru May**[2]	Custer and Sutliff (2 minutes)
D. **Collaborative documents**	Sutliff (5 minutes)
E. **Team work**	Custer (Remainder of time)

[1] Students may revise the design reports and increase in the writing grade up to a B+. The technical grade will not change. To revise, students must meet with their CI instructor to develop a revision plan within a week of receiving their paper's grade. The student must obtain a sign off on the revision from the technical instructor before submitting the document to the CI instructor. No revision can be accepted after April 19, 2012.

[2] The CI instructional team does not have a formal role in the second design report. It is a team-written document. Students may arrange to meet with their CI instructor to work on their writing of any part of it or to get some help on the final oral presentation

Graded by Dave

The Plaz Provenance File System (PPFS)

Michael Plasmeier
theplaz@mit.edu
Rudolph 10AM
March 22, 2012

Introduction

A provenance file system is a file system which stores the history and source of files edited on a local computer system. For example, when one is editing a slide deck, one might want to know from which slide deck a slide was copied from. This paper builds upon the basic file system in the early versions of Unix and introduces a provenance file system known as the *Plaz Provenance File System (PPFS)*. In particular, the PPFS introduces another layer, called the log layer, which maintains pointers to past versions and ancestors of the file.

The PPFS stores a full, verbose set of provenance information. The PPFS also stores the complete data of old versions of files, giving it many of the features of a versioning file system. The PPFS also supports seeing which files are based off a specific file. This is called reverse lookup. These characteristics make the PPFS well suited for businesses that face regulatory or legal requirements to log all changes to files. The PPFS is also well suited to businesses which frequently create derivative works from previous works. For example, a consulting company may what to know which project a slide in a slide deck was copied from. ⇒ conclude w/ several functional requirements?

→ you can more closely couple these sentences.

The system aims for a simple design and it attempts to use a minimal amount of disk space for maintaining provenance information and a de minimis amount of Random Access Memory (RAM) space for the tracking of provenance information. However, it currently uses up a lot of disk space to store old versions of files. Additional disk space could be saved by de-duplication algorithms. Data is laid out so that lookups from both directions (the ancestors of a file and the children of a file) can be performed relatively quickly. As part of the operating system, the PPFS extends the usual (`read()`, `write()`) operations. For more complicated or novel functionality, new API calls are introduced.

how quickly?

provenance-specific functionality?

At the moment, the PPFS operates only on one computer. It is not optimized to work over a network, nor does not track provenance information from files copied from other computers, such as web servers.

The current design? The design described in this report,

The Log Layer

The PPFS introduces a new layer into the Unix file system called the log layer. This layer is inserted between the file name layer and the inode layer, as shown in Figure 1. The file name layer is modified by redefining the inode number in the directory table to the log entry number.

A- Minor issues in the performance & conclusion sections dilute the impact of the whole

A

A

A

A

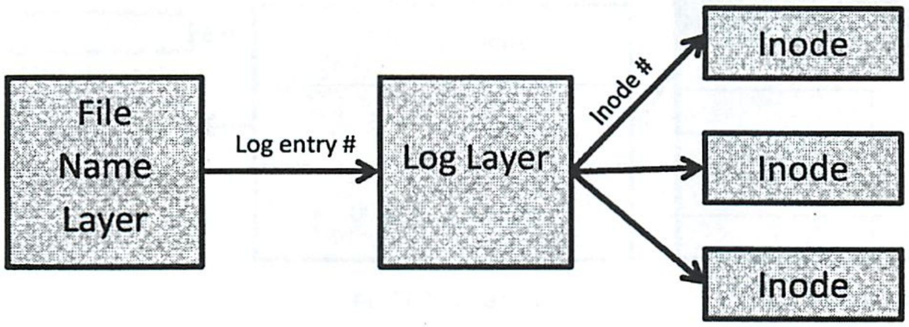


Figure 1 The Log Layer is inserted between the file name layer and the inode layer

The log layer contains a table of information about the history of each file. Each file has its own table, which is stored on disk in the same way as the inode layer. The log layer table is stored at the beginning of the disk, in a fixed position on the disk, after the inode table. The table consists of a list of log entries, each pointing to the inode number of a *version* of the file, as shown in Figure 2. A version is created automatically each time the file is saved. The last entry in the log layer entry contains a reference to the log layer entry that the file was created from (the *ancestor)*. A bit in each entry designates a row as a version/inode pointer or an ancestor /log layer pointer. If a file was created from scratch (ie using `touch`) then the last entry in the log layer table will be `0`. Although the inode stores the time the file was modified, that information would not be changed if the file was copied, so PPFS also stores that information in the log table. Additional log information, such as the current user's username and the application that made the change could is also stored here.

The number of incoming links that was stored in the inode in the original Unix file system is redefined to count the number of log layer entries pointing to the inode. The log layer table includes a count of the number of incoming links that was traditionally found in the inode. These counts are manifested in the reverse tables, described below.

Directories are ignored by the PPFS and function as usual. This behavior? may differ from certain versioning file systems.

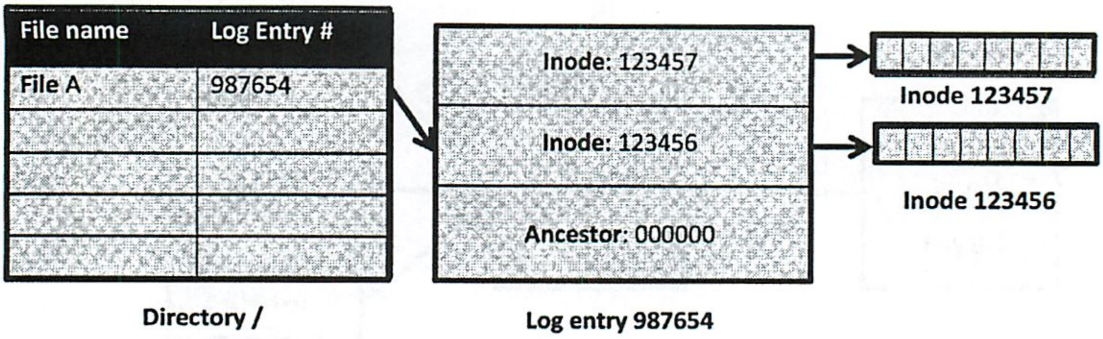


Figure 2 File A is created with content and is then edited.

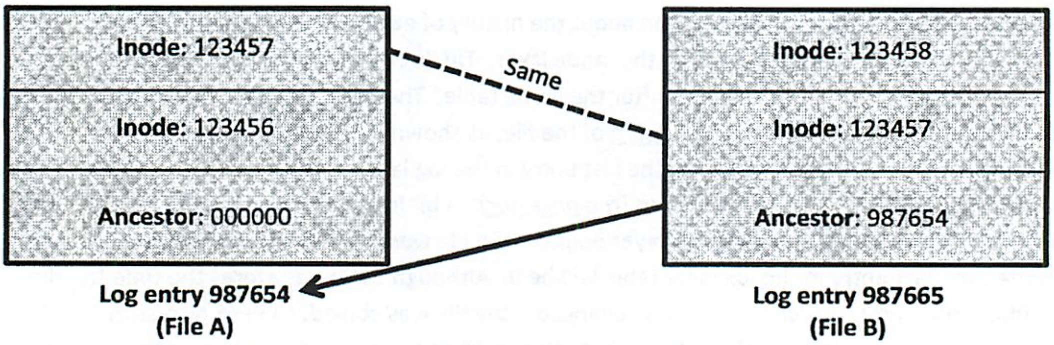


Figure 2?

Figure 3 File A, from above, is then copied to be File B. File B is then edited. Notice that File B rev 0 shares an inode or version with file A rev 1.

Should these figures be combined? Otherwise, reference figure 3 in the text.

When provenance information is queried (via `read_prov()`) for File B, the log entry for File B will be retrieved. Provenance information will then be recursively queried (to A in this example) until an ancestor of 0 is reached.

Reverse Lookup

One requirement of a provenance file system is to know all of the files which originated from a particular file. This information can be accessed using the `search_prov()` system call. In order to support the reverse search case, the log entry and inode tables are modified. A reverse inode table is added for each inode, which contains the list of log entry tables which point to that inode. A reverse log entry table is added to each leg entry to retrieve the files names which each log entry represents.

reference Figure 4 in the text?

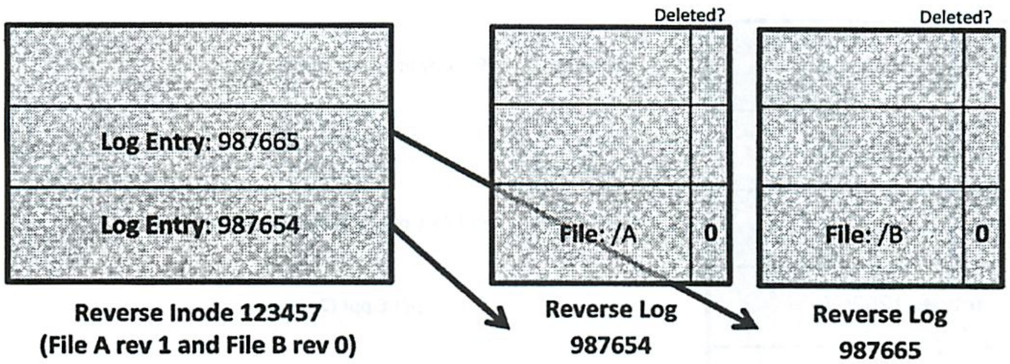


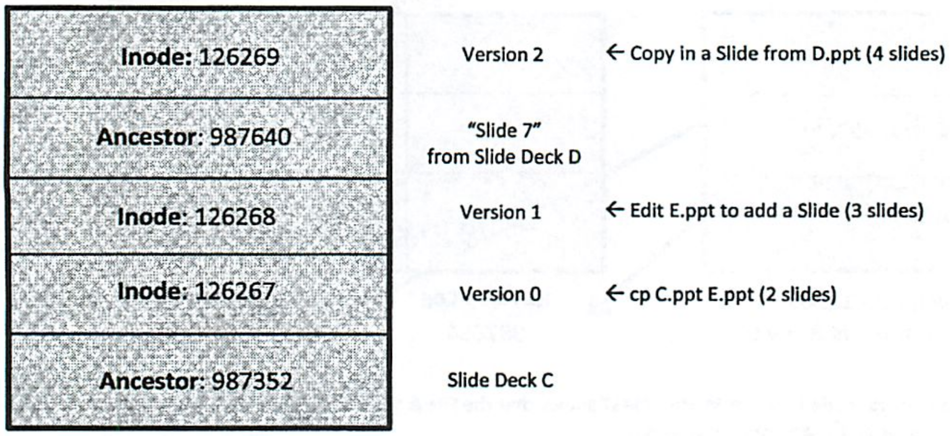
Figure 4 The Reverse Lookup Inode table for Inode 123457 shows that the File A and File B shared the same data at some point in time (i.e. one must be the ancestor of another)

In a `search_prov()` query, the system first looks up the log layer table for a particular file. For every inode mentioned in the log layer table, the system retrieves the reverse inode table. The system then retrieves the reverse log tables for each file. The system then outputs the list of files referenced. If queried? needed, the system could also lookup the log tables themselves to find the revision number and timestamp for each file.

Parts of a File

Provenance information can be stored about the *parts* of a file (for example, the slides in a slide deck). This information is stored by having multiple ancestors in the log layer, as shown in Figure 5. In this example, the difference between Slide Deck E version 2 and version 1 came from Slide Deck D. The pointer to Slide Deck D and a name for this piece would be set by the `write_prov()` call. The data that is different would be inferred by looking at the difference between the inode before and after the ancestor entry. The name data is stored in a separate table, as seen in Figure 6.

Applications wishing to take advantage of the *provenance by parts* functionality would need to implement this API call.



Log entry 987636
(Slide Deck E)

Figure 5 Slide Deck E was copied from Slide Deck C with 2 slides; a slide was added from scratch; and then a fourth slide was copied in from Slide Deck D (which was previously called slide 7 in Slide Deck D)

Log entry #	Name
4	"Slide 7"

Piece names 987636
(Slide Deck E)

Figure 6 Piece names for the various pieces for Slide Deck E. In this example, the 4th entry (from bottom) of the log entry for Slide Deck E were previously called "Slide 7" by the application.

Compilations of Files

Information about the source in compiled binary files can be stored in in a similar way. Multiple ancestor entries are stored at the bottom of the table between the first ancestor and the inode of the newly compiled file, as shown in Figure 7. The first entry will be 0, since the file was created new. Normal log entries will accumulate on top of this information, as before.

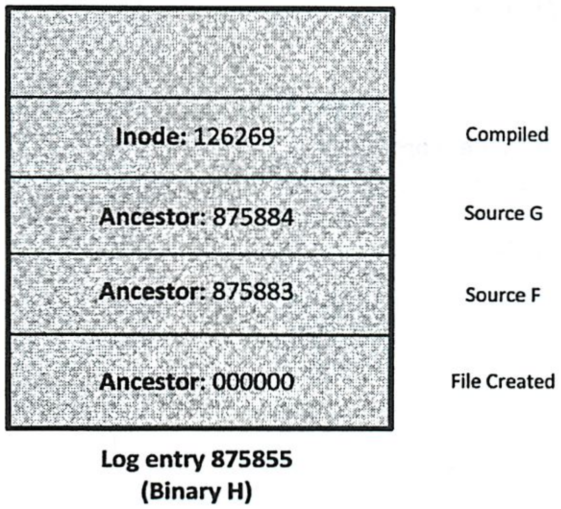


Figure 7 Binary H is compiled from Source F and Source G

File Archives

File archives present a particular challenge. File archives read information off the disk and then store it in their own proprietary format. In order to be truly portable, this requires all of the provenance information, along with all of the past versions and ancestors of a file to be stored in the file archive. This information would be retrieved using a special call, such as `read_full_provenance()`, which would store the provenance information and past versions in a flat format. This format is a XML format which mirrors the tables in the file system, as shown in Figure 8. The file would then be compressed using normal ZIP or TAR algorithms.

```
<xml schema=”ppfs-portable”>
      <log-entries>
            <log-entry id=”875855”>
                  <ancestor>000000</ancestor>
                  <ancestor>875883</ancestor>
                  <ancestor>875884</ancestor>
                  <inode>126269</inode>
                  <reverse>
                        <file>/H</file>
                  </reverse>
                  //Additional metadata (i.e. name, date) removed
            </log-entry>
            <log-entry id=”875883”>
                  <ancestor>000000</ancestor>
                  <inode>126267</inode>
                  <reverse>
```

```
                <file>/F</file>
            </reverse>
        </log-entry>
        <log-entry id="875884">
            <ancestor>000000</ancestor>
            <inode>126268</inode>
            <reverse>
                <file>/G</file>
            </reverse>
        </log-entry>
    </log-entries>
    <inodes>
        <inode id="126269">
            <data>(Binary data)</data>
            <reverse>
                <log-entry>875855</log-entry>
            </reverse>
        <inode>
        <inode id="126268">
            <data>(Binary data)</data>
            <reverse>
                <log-entry>875883</log-entry>
            </reverse>
        <inode>
        <inode id="126267">
            <data>(Binary data)</data>
            <reverse>
                <log-entry>875884</log-entry>
            </reverse>
        <inode>
    </inodes>
</xml>
```

Can you make figure 8 fit on one page?

Figure 8 The flat file XML for the scenario in Figure 6

When the file archive is extracted, the provenance information is recreated using a special `write_full_provenance()` call. The inode and table entry numbers will change, but the same structure will be created. Those that are interested in preserving the authenticity of the provenance information should disable this feature, because it allows anyone to write provenance information (including old time stamps) to disk.

Deletion and Thinning

↓ orphan/widowing

When unlink(filename) is called, the filename to log entry link is removed, and the deleted bit in the reverse log table is flipped (decrementing the traditional link count in the log layer entry). When the count of incoming links in the log entry table reaches 0, the file is no longer accessible. However, the log table and versions are kept in order to preserve provenance information.

In order to save space some intermediate versions can be removed according to a thinning schedule. This schedule is user-settable, but the default values are shown in Table 1. The thinning process is accomplished by a "garbage collection"-style program. A revision will is? be kept if more than one log entry is present in the reverse inode table – i.e. when a file was copied and is now provenance information for a different file. When versions are thinned, the actual inode/data is removed from the disk, and all references to that version are removed from the log layer.

Technical thought: Your "thinning" is a little bit like how people forget details over time. I find it curious that the computer suffers the same amnesia as the user.

Days after revision created	Target number of revisions kept
< 7 days	1 / minute
> 7 days and < 30 days	1 / hour
> 30 days and < 1 year	1 / day
> 1 year	1 / week

Table 1 Intermediate revisions can be thinned after a certain amount of time after their creation. These are the default values.

Performance

It's hard to tell whether this is a functional requirement or you have successfully such a requirement.

Minor errors make this paragraph difficult to understand.

PPFS should be not appreciably slower when adding many files to the disk. Principally, the disk must make one additional write (the log layer table) in addition to its other writes. Generally, the non-sequential disk accesses slow a hard drive down. PPFS adds one additional non-sequential access. Thus the system should be no more than 33% slower (adding the log layer to the file system pointer, inode, and file data). Thus the system should be able to easily handle writing 10 files to disk per second. PPFS scales with the size of the disk and is linear with regard to the number of items added to disk per second. The garbage collection process is optional, and can be postponed until the system is relatively idle.

Set this structure up?

In addition, the system can quickly search for the children of a file (files that are based on that file) by using the reverse inode table. Such lookups should not depend on the number of files on the disk. PPFS scales well with regard to the number of files on disk.

how well?

For a file with many ancestors, PPFS handles reads and writes to a file the same as a file without an ancestor. Retrieving the full list of provenance information scales with the number of ancestors. It is envisioned that this will not be a large bottleneck, since the number of ancestors is envisioned to be relatively low and pulling a full list of provenance information is an infrequent operation. Caching could be added to the system to improve this time, but the additional step to update or invalidate the cache would slow the copying of files.

One of the most significant performance impacts is the time to update a file. PPFS rewrites the entire file each time it is saved, in order to maintain a version history of the file. PPFS is not optimized for large files, such as media files, and is likely unsuitable for those use cases. ✓

Could you "fix" this by breaking such files into parts?

support your qualitative claims with quantitative information?

word choice?

PPFS uses a significant amount of disk space. PPFS is designed to provide verbosity and maintain provenance information at the expense of disk space. Thus PPFS is best suited to organizations that require comprehensive and persistent logging. ✓

Conclusion

no need for the subjunctive mood here.

PPFS is a provenance file system designed to provide a comprehensive and reliable log of the history of each file. PPFS modifies the basic Unix file system to add a log layer that preserves the provenance and version history of each file on the disk. PPFS should add minimal overheard, beyond the keeping of multiple versions. PPFS should be able to easily handle lookups from both directions (the ancestors of a file and the children of a file) in a short amount of time. PPFS should scale well to the size of the disk, the number of files on disk, the number of ancestors of a file. PPFS can do additional work to reduce the disk space that old versions take up.

Again, it is difficult to discern between what your design does & doesn't do.

Implementation Issues

Modern file systems have advanced beyond the basic Unix file system that PPFS is based on. Care should be taken to maintain the current features of file systems while implementing PPFS.

Where additional API calls have been added, developers must be recruited to update their applications to support the new APIs.

Future Work

PPFS suffers from a number of limitations, which could be addressed by modifications to the system.

PPFS currently rewrites each file when it is edited, using a lot of disk space. A de-duplication algorithm which, for example, only stored the changes to a file, could save a significant amount of disk space.

PPFS currently only works on a local computer. PPFS could be expanded to work across a network. Provenance information is particularly helpful when there are multiple people working on a group of documents.

Instead, show how your log layer can be used across a network

PPFS is currently designed to operate on a single disk. Because of the large amount of disk space used by PPFS, it could be expanded to work across multiple disks. For example, the RAID system allows multiple disks to be seen as one disk by a computer. This would allow users to add storage to the system as needed.

Can you rearrange so that you end on a conclusive note (declare victory) rather than on the details of implementation & future work?

Acknowledgements

Reviewers

- Dave Custer
- Travis Grusecki

References

[1] J. Saltzer, M. F. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA: Morgan Kaufmann, 2009.

Word Count

2,358 words, including captions

M.I.T. DEPARTMENT OF EECS

6.033 - Computer System Engineering **Database Hands-On Assignment**

Hands-on 6: Databases

This hands-on assignment will introduce you to databases and transactions. Many applications store data in a database similar to the one you will be using in this exercise. For example, it is common for web applications to store data in an SQL database, such as MySQL or Postgres, and in this hands-on, we will be using Postgres in particular. One of the benefits of using a database is that it can provide serializability and durability guarantees that an ordinary file system does not. have not used before

Once you are done with this hands-on assignment, submit your answers, along with the commands you ran and the output you got, using the online submission site before the **beginning of recitation**.

Intro to databases

For the purposes of this exercise, we have set up a Postgres database server on the machine `ud0.csail.mit.edu`. We have created an account for every student registered for 6.033 this term; check the online submission web site for a short text file (under the database hands-on assignment) that contains your username and password. Please use our Postgres server for this hands-on assignment, rather than some other Postgres server, since we made some changes to defaults on our server. In particular, we changed the default transaction isolation level for all user accounts to `SERIALIZABLE`. ?

You can access your database by running the following commands on a Linux-based Athena workstation or dialup. To find your assigned username and password, check the online submission site for a comment under the database hands-on assignment.

```
athena% psql -h ud0.csail.mit.edu -U username
Password for user username: password
psql (8.4.10)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

username=>
```

The `psql` command allows you to issue SQL queries to the database. To exit from this shell, type `\q`.

SQL databases allow users (such as application developers) to organize data in *tables*. Each table consists of record called *rows*. Each row, in turn, consists of several attributes called *columns*. All rows in the table share the same set of columns, and these columns must be defined ahead of time by the user when the table is created.

As an example, MIT's TechCash system, which handles payments using MIT ID cards, might construct a table that stores information about accounts, such as the account holder's Athena username, their full name, and the total amount of money in their account. You can visualize the table as containing the following information, with 4 rows and 3 columns:

username	fullname	balance
jones	Alice Jones	82
bitdiddl	Ben Bitdiddle	65

mike	Michael Dole	73
alyssa	Alyssa P. Hacker	79

In this hands-on, you will use the SQL language to issue queries to the database to perform operations on tables. Postgres has a good tutorial on SQL, as well as a more detailed SQL language manual that you can refer to.

Using SQL

Exercise 1. As a first step, use the SQL `CREATE` command to create the table shown above, and then execute several `INSERT` commands to insert each of the rows into the resulting table, as follows:

```
username=> create table accounts (username varchar(8), fullname varchar(128), balance int);
CREATE TABLE
username=> insert into accounts values ('jones', 'Alice Jones', 82);
INSERT 0 1
...
```

If you make a mistake, you can delete the `accounts` table by issuing a `DROP TABLE accounts;` command, and then starting over.

Exercise 2. Now, examine the table using the special `\d` command:

```
username=> \d accounts
                Table "public.accounts"
  Column  |          Type          | Modifiers
----------+------------------------+-----------
 username | character varying(8)   |
 fullname | character varying(128) |
 balance  | integer                |

username=>
```

We can now read data from the table using the `SELECT` command. The `SELECT` command allows you to choose which rows to select (by specifying a predicate in a `WHERE` clause), and what data to return from each row (e.g., individual columns or aggregates such as `SUM()`):

```
username=> select username, fullname, balance from accounts;
...
username=> select fullname from accounts where balance > 75;
...
username=> select sum(balance) from accounts;
...
username=>
```

Exercise 3. Run the above three commands. Also construct and run a command to display the full name of the person with username `bitdiddl`. Also construct and run a command to display the average account balance of people that have at least $70 in their account. You may want to refer to the Postgres SQL manual to find a suitable function for computing the average.

Exercise 4. Transfer $10 from `jones` to `mike`. You will find the `UPDATE` command useful; consult the Postgres SQL manual for more details. For this exercise, it will suffice to perform two updates: one to deduct $10 from Alice's balance, and another to add $10 to Michael's balance.

Transactions

To deal with concurrent operations, many SQL databases, including Postgres, support transactions, which allow several SQL statements to execute atomically (often meaning both *before-or-after atomicity* and *all-or-nothing* *atomicity*). To execute statements as a single transaction, SQL clients first issue a `BEGIN` command, then execute some SQL commands, and finally issue either a `COMMIT` command, which makes the transaction's changes permanent, or a `ROLLBACK` command, which reverts the changes from all of the commands in the transaction.

In this assignment, you will simulate concurrent database queries by issuing SQL statements over two different connections to the database. Open up two terminals, and run `psql` in each of them, to create two connections to the database. We will use two colors (blue and red) to indicate the two database sessions.

Start a transaction in the first (blue) terminal, and display a list of all accounts:

B

```
username=> begin;
BEGIN
username=> select * from accounts;
 username |      fullname      | balance
----------+-------------------+---------
 bitdiddl | Ben Bitdiddle      |      65
 alyssa   | Alyssa P. Hacker |      79
 jones    | Alice Jones        |      72
 mike     | Michael Dole       |      83
(4 rows)

username=>
```

Now, in the second (red) terminal, also start a transaction and add an account for Chuck:

R

```
username=> begin;
BEGIN
username=> insert into accounts values ('chuck', 'Charles Robinson', 55);
INSERT 0 1
username=>
```

Exercise 5. Generate a list of all accounts in the first (blue) terminal, and in the second (red) terminal. What output do you get? Are they the same or not? Why?

Now, commit the transaction in the second (red) terminal, by issuing the `COMMIT` statement:

R

```
username=> commit;
COMMIT
username=>
```

Exercise 6. Generate a list of all accounts from the first (blue) terminal again. Does it include Chuck? Why or why not?

Exercise 7. Commit the transaction in the first (blue) terminal, start a new transaction, and generate a list of all accounts in the new transaction:

B

```
username=> commit;
COMMIT
username=> begin;
BEGIN
username=> select * from accounts;
...
```

What output do you get? Is it different than the output you received in exercise 6? Why or why not?

Now, let's try to modify the same account from two different transactions. In the first (blue) terminal, start a transaction and deposit $5 into Mike's account:

B

```
username=> begin;
BEGIN
username=> update accounts set balance=balance+5 where username='mike';
```

```
    UPDATE 1
```

In the second (red) terminal, start a transaction and withdraw $10 from Mike's account:

R

```
    username=> begin;
    BEGIN
    username=> update accounts set balance=balance-10 where username='mike';
    UPDATE 1
```

Exercise 8. What happens to the second update? Why?

Let's try aborting a transaction: enter the ABORT command in the first (blue) terminal, undoing the $5 deposit:

B

```
    username=> abort;
    ROLLBACK
    username=>
```

Exercise 9. What happens to the second (red) terminal's transaction?

Exercise 10. If you now commit the transaction in the second terminal, what is the resulting balance in Mike's account?

Now let's perform an atomic transfer of $15 from Ben to Alyssa, using two UPDATE statements in a single transaction. In the first (blue) terminal, list the balances of all accounts. Then start a transaction and do a part of a transfer in the second (red) terminal:

R

```
    username=> begin;
    BEGIN
    username=> update accounts set balance=balance-15 where username='bitdiddl';
    UPDATE 1
    username=>
```

Exercise 11. If you now look at the list of all account balances in the first (blue) terminal, have the results changed compared to before you started the transaction in the second (red) terminal?

Exercise 12. Finish the transfer by executing the following commands in the second (red) terminal. After each command, list all of the account balances in the first (blue) terminal, to see at what point the effects of the second terminal's transaction become visible. What is that point, and why?

R

```
    username=> update accounts set balance=balance+15 where username='alyssa';
    UPDATE 1
    username=> commit;
    COMMIT
    username=>
```

Transaction isolation levels

Postgres supports different transaction isolation levels. So far, you have been using SERIALIZABLE transactions, meaning that the transactions appear to execute in some serial order. Postgres also has a READ COMMITTED isolation level, which allows one transaction to immediately see the results of any other committed transaction; you can read more about it in the Postgres manual here.

Exercise 13. By default, Postgres uses the READ COMMITTED isolation level (and for all of the above exercises, we explicitly configured our Postgres server to use the SERIALIZABLE isolation level instead). Why do you think Postgres developers chose to make the default isolation level READ COMMITTED, which allows non-serializable schedules?

Challenge exercise 14 (optional). Come up with two transactions that, when run concurrently (with some interleaving of commands that you get to determine) produce a non-serializable schedule. That is, the result that you obtain should be impossible to produce with SERIALIZABLE isolation, but is possible with READ COMMITTED.

To set the transaction isolation level to READ COMMITTED, you can issue the following command:

```
username=> set session characteristics as transaction isolation level read committed;
SET
username=>
```

Keep in mind that the effects of this command last for the duration of a single session. This means you have to issue it every time you re-start psql, and issue it in *every* session of psql that you start, to get READ COMMITTED isolation between your transactions.

Go to 6.033 Home Page

Hands On 6

4/8

User Theplaz

pass 58d2ae08

Searalizable?

No dirty reads = read data from Concurrent uncommitted txn

No nonrepeatable read = reread data fond that modified

No phantom read = reexecutes a query and found that stuff changed

Basically very strict isolation

Emulates no concurrency

(Lots of complicatedness ...)

tables, rows, cols, etc

lots of manual ~~label~~ labor ...

ld to view table

(I know like all of this stuff ...)

②

Transfer w/ 2 updates

Now trasactions

both types of atomacy

BEGIN

COMMIT or ROLLBACK

Opened 2 terminals

Ahh select is different in trasactions

Does not show up

(? How does it do this?)

Read Committed

Dirty read → Not possible

Non repeatable read → possible

Phantom read → possible

③

Why?

IDK

To make it go faster

But what exactly are non repetable read + Phantom read

like balance=+15 is wrong

Or select data and then it changes

I get dirty read..

Found WP some article w/ the ans

Isolation (database systems)

From Wikipedia, the free encyclopedia

In database systems, **isolation** is a property that defines how/when the changes made by one operation become visible to other concurrent operations. Isolation is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties.

Contents

Read
4/8

Isolation levels

Of the four ACID properties in a DBMS (Database Management System), the isolation property is the one most often relaxed. When attempting to maintain the highest level of isolation, a DBMS usually acquires locks on data or implements multiversion concurrency control, which may result in a loss of concurrency. This requires adding additional logic for the application to function correctly.

Most DBMS's offer a number of *transaction isolation levels*, which control the degree of locking that occurs when selecting data. For many database applications, the majority of database transactions can be constructed to avoid requiring high isolation levels (e.g. SERIALIZABLE level), thus reducing the locking overhead for the system. The programmer must carefully analyze database access code to ensure that any relaxation of isolation does not cause software bugs that are difficult to find. Conversely, if higher isolation levels are used, the possibility of deadlock is increased, which also requires careful analysis and programming techniques to avoid.

The isolation levels defined by the ANSI/ISO SQL standard are listed as follows.

Serializable

This is the *highest* isolation level.

With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the ***phantom reads*** phenomenon (see below).

When using non-lock based concurrency control, no locks are acquired; however, if the system detects a *write collision* among several concurrent transactions, only one of them is allowed to commit. See *snapshot isolation* for more details on this topic.

Repeatable reads

In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. However, *range-locks* are not managed, so the ***phantom reads*** phenomenon can occur (see below).

Read committed

In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the ***non-repeatable reads*** phenomenon can occur in this isolation level, as discussed below). As in the previous level, *range-locks* are not managed.

Read uncommitted

This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed (see below), so one transaction may see *not-yet-committed* changes made by other transactions.

the 4 levels

Default isolation level

The *default isolation level* of different DBMS's varies quite widely. Most databases that feature transactions allow the user to set any isolation level. Some DBMS's also require additional syntax when performing a SELECT statement to acquire locks (e.g. *SELECT ... FOR UPDATE* to acquire exclusive write locks on accessed rows).

However, the definitions above have been criticised in the paper A Critique of ANSI SQL Isolation Levels (http://www.cs.umb.edu/~poneil/iso.pdf) as being ambiguous, and as not accurately reflecting the isolation provided by many databases:

> This paper shows a number of weaknesses in the anomaly approach to defining isolation levels. The three ANSI phenomena are ambiguous. Even their broadest interpretations do not exclude anomalous behavior. This leads to some counter-intuitive results. In particular, lock-based isolation levels have different characteristics than their ANSI equivalents. This is disconcerting because commercial database systems typically use locking. Additionally, the ANSI phenomena do not distinguish among several isolation levels popular in commercial systems.

There are also other criticisms concerning ANSI SQL's isolation definition, in that it encourages implementors to do "bad things":

> ... it relies in subtle ways on an assumption that a locking schema is used for concurrency control, as opposed to an optimistic or multi-version concurrency scheme. This implies that the proposed semantics are *ill-defined*.[1]

I don't get at 1st glance

the problem

Read phenomena

The ANSI/ISO standard SQL 92 refers to three different *read phenomena* when Transaction 1 reads data that Transaction 2 might have changed.

In the following examples, two transactions take place. In the first, Query 1 is performed. Then, in the second transaction, Query 2 is performed and committed. Finally, in the first transaction, Query 1 is performed again.

The queries use the following data table:

users

id	name	age
1	Joe	20
2	Jill	25

Dirty reads (Uncommitted Dependency)

A dirty read occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.

very bad

Dirty reads work similarly to non-repeatable reads; however, the second transaction would not need to be committed for the first query to return a different result. The only thing that may be prevented in the READ UNCOMMITTED isolation level is updates appearing out of order in the results; that is, earlier updates will always appear in a result set before later updates.

In our example, Transaction 2 changes a row, but does not commit the changes. Transaction 1 then reads the uncommitted data. Now if Transaction 2 rolls back its changes (already read by Transaction 1) or updates different changes to the database, then the view of the data may be wrong in the records of Transaction 1.

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
/* No commit here */
```

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Transaction 2

```
ROLLBACK; /* lock-based DIRTY READ */
```

But in this case no row exists that has an id of 1 and an age of 21.

Non-repeatable reads

A *non-repeatable read* occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.

Non-repeatable reads phenomenon may occur in a lock-based concurrency control method when read locks are not acquired when performing a SELECT, or when the acquired locks on affected rows are released as soon as the SELECT operation is performed. Under the multiversion concurrency control method, *non-repeatable reads* may occur when the requirement that a transaction affected by a commit conflict must roll back is relaxed.

Transaction 1 **Transaction 2**

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```

L reads 20

```
/* Query 2 */
UPDATE users SET age = 21 WHERE i
COMMIT; /* in multiversion concu
   control, or lock-based READ C(
```

id = 1;

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
COMMIT; /* lock-based REPEATABLE READ */
```

L reads 21

In this example, Transaction 2 commits successfully, which means that its changes to the row with id 1 should become visible. However, Transaction 1 has already seen a different value for *age* in that row. At the SERIALIZABLE and REPEATABLE READ isolation levels, the DBMS must return the old value for the second SELECT. At READ COMMITTED and READ UNCOMMITTED, the DBMS may return the updated value; this is a non-repeatable read.

The How To

There are two basic strategies used to prevent non-repeatable reads. The first is to delay the execution of Transaction 2 until Transaction 1 has committed or rolled back. This method is used when locking is used, and produces the serial schedule **T1, T2**. A serial schedule does not exhibit *non-repeatable reads* behaviour.

Where it paused

In the other strategy, as used in *multiversion concurrency control*, Transaction 2 is permitted to commit first, which provides for better concurrency. However, Transaction 1, which commenced prior to Transaction 2, must continue to operate on a past version of the database — a snapshot of the moment it was started. When Transaction 1 eventually tries to commit, the DBMS checks if the result of committing Transaction 1 would be equivalent to the schedule **T1, T2**. If it is, then Transaction 1 can proceed. If it cannot be seen to be equivalent, however, Transaction 1 must roll back with a serialization failure.

So if its not thes same - for update

? how - i leads to error message?

Using a lock-based concurrency control method, at the REPEATABLE READ isolation mode, the row with ID = 1 would be locked, thus blocking Query 2 until the first transaction was committed or rolled back. In READ COMMITTED mode, the second time Query 1 was executed, the age would have changed.

Read the MVCC article

Under multiversion concurrency control, at the SERIALIZABLE isolation level, both SELECT queries see a snapshot of the database taken at the start of Transaction 1. Therefore, they return the same data. However, if Transaction 1 then attempted to UPDATE that row as well, a serialization failure would occur and Transaction 1 would be forced to roll back.

how - from log - or a rollback log - Don't update only copy

At the READ COMMITTED isolation level, each query sees a snapshot of the database taken at the start of each query. Therefore, they each see different data for the updated row. No serialization failure is possible in this mode (because no promise of serializability is made), and Transaction 1 will not have to be retried.

Phantom reads

A *phantom read* occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first.

This can occur when *range locks* are not acquired on performing a *SELECT* ... *WHERE* operation. The *phantom reads* anomaly is a special case of *Non-repeatable reads* when Transaction 1 repeats a ranged *SELECT* ... *WHERE* query and, in the middle of both operations, Transaction 2 creates (i.e. INSERT) new rows (in the target table) which fulfill that *WHERE* clause.

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users VALUES ( 3, 'Bob', 27 );
COMMIT;
```

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

↳ shows new row

Note that Transaction 1 executed the same query twice. If the highest level of isolation were maintained, the same set of rows should be returned both times, and indeed that is what is mandated to occur in a database operating at the SQL SERIALIZABLE isolation level. However, at the lesser isolation levels, a different set of rows may be returned the second time.

In the SERIALIZABLE isolation mode, Query 1 would result in all records with age in the range 10 to 30 being locked, thus Query 2 would block until the first transaction was committed. In REPEATABLE READ mode, the range would not be locked, allowing the record to be inserted and the second execution of Query 1 to include the new row in its results.

Isolation Levels, Read Phenomena and Locks

Isolation Levels vs Read Phenomena

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

"may occur" means that the isolation level suffers that phenomenon, while "-" means that it does not suffer it.

Isolation Levels vs Locks

Isolation level	Write Lock	Read Lock	Range Lock
Read Uncommitted	-	-	-
Read Committed	V	-	-
Repeatable Read	V	V	-
Serializable	V	V	V

"V" indicates that the method locks for that operation, keeping that lock till the end of the transaction containing that operation.

Note: Read (i.e. SELECT) operations can acquire read (shared) locks in the *Read Committed* isolation level, but they are released immediately after the read operation is performed.

References

1. ^ salesforce (2010-12-06). "Customer testimonials (SimpleGeo, CLOUDSTOCK 2010)" (http://www.youtube.com/v/7J61pPG9j90?version=3) . www.DataStax.com: DataStax. http://www.youtube.com/v/7J61pPG9j90?version=3. Retrieved 2010-03-09. "(see above at about 13:30 minutes of the webcast!)"

See also

- Atomicity
- Consistency
- Durability
- Lock (database)
- Optimistic concurrency control
- Relational Database Management System
- Snapshot isolation

Retrieved from "http://en.wikipedia.org/w/index.php?title=Isolation_(database_systems)&oldid=481823552"
Categories: Data management | Transaction processing

- This page was last modified on 14 March 2012 at 09:51.
-

Multiversion concurrency control

From Wikipedia, the free encyclopedia

How it does updates

Multiversion concurrency control (abbreviated **MCC** or **MVCC**), in the database field of computer science, is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.[1]

For instance, a database will implement updates not by deleting an old piece of data and overwriting it with a new one, but instead by marking the old data as obsolete and adding the newer "version." Thus there are multiple versions stored, but only one is the latest. This allows the database to avoid overhead of filling in holes in memory or disk structures but requires (generally) the system to periodically sweep through and delete the old, obsolete data objects. For a document-oriented database such as CouchDB it also allows the system to optimize documents by writing entire documents onto contiguous sections of disk—when updated, the entire document can be re-written rather than bits and pieces cut out or maintained in a linked, non-contiguous database structure.

MVCC also provides potential "point in time" consistent views. In fact read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the DB to read, and read these "versions" of the data. This avoids managing locks for read transactions because writes can be isolated by virtue of the old versions being maintained, rather than through a process of locks or mutexes. Writes affect future "version" but at the transaction ID that the read is working at, everything is guaranteed to be consistent because the writes are occurring at a later transaction ID.

In other words, MVCC provides each user connected to the database with a "snapshot" of the database for that person to work with. Any changes made will not be seen by other users of the database until the transaction has been committed.

Contents

Implementation

MVCC uses timestamps or increasing transaction IDs to achieve transactional consistency. MVCC ensures a transaction never has to wait for a database object by maintaining several versions of an object. Each version would have a write timestamp and it would let a transaction (T_i) read the most recent version of an object which precedes the transaction timestamp ($TS(T_i)$).

If a transaction (T_i) wants to write to an object, and if there is another transaction (T_k), the timestamp of T_i must precede the timestamp of T_k (i.e., $TS(T_i) < TS(T_k)$) for the object write operation to succeed. Which is to say a write cannot complete if there are outstanding transactions with an earlier timestamp.

Every object would also have a read timestamp, and if a transaction T_i wanted to write to object P, and the timestamp of that transaction is earlier than the object's read timestamp ($TS(T_i) < RTS(P)$), the transaction T_i is aborted and restarted. Otherwise, T_i creates a new version of P and sets the read/write timestamps of P to the timestamp of the transaction $TS(T_i)$.

The obvious drawback to this system is the cost of storing multiple versions of objects in the database. On the other hand reads are never blocked, which can be important for workloads mostly involving reading values from the database. MVCC is particularly adept at implementing true snapshot isolation, something which other methods of concurrency control frequently do either incompletely or with high performance costs.

Example

At Time = "t1", the state of a database could be:

timing

Time	Object 1	Object 2
t1	"Hello"	"Bar"
t0	"Foo"	"Bar"

value changed

This indicates that the current set of this database (perhaps a key-value store database) is Object 1="Hello", Object 2="Bar". Previously, Object 1 was "Foo" but that value has been superseded. It is not deleted because the database holds multiple versions, but it will be deleted later.

If a long running transaction starts a read operation, it will operate at transaction "t1" and see this state. If there is a concurrent update (during that long-running read transaction) which deletes Object 2 and adds Object 3="Foo-Bar", the database state will look like:

Time	Object 1	Object 2	Object 3
t2	"Hello"	(deleted)	"Foo-Bar"
t1	"Hello"	"Bar"	
t0	"Foo"	"Bar"	

Now there is a new version as of transaction ID "t2". Note, critically, that the long-running read transaction *still has access to a coherent snapshot of the system at "t1"*, even though the write transaction added data as of "t2", so the read transaction is able to run in isolation from the update transaction that created the "t2" values. This is how MVCC allows isolated, ACID reads without any locks. (Note, however, that the write transaction does need to use locks.)

the old data still there

History

thats how I thought it worked - but is high overhead

Multiversion concurrency control is described in some detail in the 1981 paper "Concurrency Control in

Distributed Database Systems" [2] by Philip Bernstein and Nathan Goodman—then employed by the Computer Corporation of America. Bernstein and Goodman's paper cites a 1978 dissertation[3] by David P. Reed which quite clearly describes MVCC and claims it as an original work.

Databases with MVCC

- Altibase
- Berkeley DB[4]
- Bigdata[5]
- CouchDB
- IBM DB2 since IBM DB2 9.7 LUW ("Cobra") under CS isolation level - in CURRENTLY COMMITTED mode[6]
- IBM Cognos TM1 in versions 9.5.2 and up. [7]
- Drizzle
- eXtremeDB[8]
- Firebird[9]
- FLAIM
- GE Smallworld Version Managed Data Store
- H2 Database Engine (experimental since Version 1.0.57 (2007-08-25)) [10]
- MDB (http://gitorious.org/mdb)
- Hawtdb (http://hawtdb.fusesource.org)
- HSQLDB (starting with version 2.0)
- Ingres[11]
- InterBase (all versions)[12]
- MarkLogic Server
- MDB (http://gitorious.org/mdb)
- Meronymy SPARQL Database Server
- Microsoft SQL Server (starting with SQL Server 2005)
- MySQL when used with InnoDB,[13] Falcon,[14] or Archive storage engines.
- Netezza
- ObjectStore
- Oracle database all versions since Oracle 3[15]
- OrientDB[16]
- PostgreSQL[17]
- Rdb/ELN [18]
- RDM Embedded[19]
- REAL Server
- ScimoreDB
- sones GraphDB [20]
- Sybase SQL Anywhere
- Sybase IQ
- ThinkSQL
- Zope Object Database[21]

Other software with MVCC

- JBoss Cache (v. 3.0) MVCC has landed (http://jbosscache.blogspot.com/2008/07/mvcc-has-landed.html)
- Infinispan [22]
- EHcache (v. 1.6.0-beta4) [23]
- Clojure language software transactional memory
- Subversion and many other source code repositories.
- pojo-mvcc - a lightweight MVCC implementation written in the Java programming language [24]

See also

- Timestamp-based concurrency control
- Clojure
- Read-copy-update
- Vector clock

References

1. ^ http://clojure.org/refs
2. ^ Philip Bernstein and Nathan Goodman (1981). "Concurrency Control in Distributed Database Systems" (http://portal.acm.org/citation.cfm?id=356842.356846) . *ACM Computing Surveys*. http://portal.acm.org/citation.cfm?id=356842.356846.
3. ^ Reed, D.P. (September 21, 1978). "Naming and Synchronization in a Decentralized Computer System" (http://www.lcs.mit.edu/publications/specpub.php?id=773) . *MIT dissertation*. http://www.lcs.mit.edu/publications/specpub.php?id=773.
4. ^ Berkeley DB Reference Guide: Degrees of Isolation (http://download.oracle.com/docs/cd/E17076_02/html/programmer_reference/transapp_read.html)
5. ^ Bigdata Blog (http://www.bigdata.com/blog)
6. ^ DB2 Version 9.7 LUW Information Center, Currently committed semantics improve concurrency (http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.perf.doc/doc/c0053760.html)
7. ^ TM1 9.5.2 Information Center, Parallel Interaction (http://publib.boulder.ibm.com/infocenter/ctm1/v9r5m0/topic/com.ibm.swg.im.cognos.tm1_nfg.9.5.2.doc/tm1_nfg_id90tm1_952_nfg_CubeVersioning_N38274.html#tm1_952_nfg_CubeVersioning_N38274)
8. ^ Graves, Steve (May 1, 2010). "Multi-Core Software: To Gain Speed, Eliminate Resource Contention" (http://www.rtcmagazine.com/articles/view/101612) . *RTC Magazine*. http://www.rtcmagazine.com/articles/view/101612.
9. ^ White paper by Roman Rokytsky Firebird and Multi Version Concurrency Control (http://www.firebirdsql.org/doc/whitepapers/fb_vs_ibm_vs_oracle.htm)
10. ^ Multi-Version Concurrency Control in the H2 Database Engine (http://www.h2database.com/html/advanced.html#mvcc)
11. ^ http://community.ingres.com/wiki/MVCC
12. ^ Todd, Bill (2000). "InterBase: What Sets It Apart" (http://web.archive.org/web/20060226083331/http://www.dbginc.com/tech_pprs/IB.html) . Archived from the original (http://dbginc.com/tech_pprs/IB.html) on 26 February 2006. http://web.archive.org/web/20060226083331/http://www.dbginc.com/tech_pprs/IB.html. Retrieved 4 May 2006.
13. ^ MySQL 5.1 Reference Manual, Section 14.2.12: Implementation of Multi-Versioning (http://dev.mysql.com/doc/refman/5.1/en/innodb-multi-versioning.html)
14. ^ or Maria MySQL 5.1 Reference Manual, Section 14.6.1: Falcon Features (http://mysql.org/doc/refman/5.1/en/se-falcon-features.html)
15. ^ Oracle Database Concepts: Chapter 13 Data Concurrency and Consistency Multiversion Concurency Control

(http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14220/consist.htm#i17881)

16. ^ OrientDb Documentation[1] (http://code.google.com/p/orient/wiki/Transactions)
17. ^ PostgreSQL 9.1 Documentation, Chapter 13: Concurrency Control (http://postgresql.org/docs/9.1/static/mvcc.html)
18. ^ h18000.www1.hp.com/info/SP2803/SP2803PF.PDF
19. ^ RDM Embedded 10.1 Reference Manual, d_trrobegin (http://docs.raima.com/rdme/10_1/Content/RM/d_trrobegin.htm)
20. ^ [2] (http://www.sones.com)
21. ^ Proposal for MVCC in ZODB (http://wiki.zope.org/ZODB/MultiVersionConcurrencyControl)
22. ^ [3] (http://www.jboss.org/infinispan)
23. ^ ehcache site (http://ehcache.sourceforge.net/)
24. ^ pojo-mvcc project home (http://code.google.com/p/pojo-mvcc/)

Further reading

- Gerhard Weikum, Gottfried Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann, 2002, ISBN 1-55860-508-8

Retrieved from "http://en.wikipedia.org/w/index.php?title=Multiversion_concurrency_control&oldid=480767248"

Categories: Concurrency control | Concurrency control algorithms | Transaction processing

- This page was last modified on 8 March 2012 at 02:04.
-

Hands-on 6: Databases

Michael Plasmeier

1.
```
theplaz=> create table accounts (username varchar(8), fullname
varchar(128), balance int);
CREATE TABLE
theplaz=> insert into accounts values ('jones', 'Alice Jones', 82);
INSERT 0 1
theplaz=> insert into accounts values ('bitdiddl', 'Ben Bitdiddle',
65);
INSERT 0 1
theplaz=> insert into accounts values ('mike', 'Michael Dole', 73);
INSERT 0 1
theplaz=> insert into accounts values ('alyssa', 'Alyssa P. Hacker',
79);
INSERT 0 1
```

2.
```
theplaz=> \d accounts
            Table "public.accounts"
  Column  |          Type          | Modifiers
----------+------------------------+-----------
 username | character varying(8)   |
 fullname | character varying(128) |
 balance  | integer                |
```

3.
```
theplaz=> select username, fullname from accounts;
 username |     fullname
----------+------------------
 jones    | Alice Jones
 bitdiddl | Ben Bitdiddle
 mike     | Michael Dole
 alyssa   | Alyssa P. Hacker
(4 rows)

theplaz=> select sum(balance) from accounts;
 sum
-----
 299
(1 row)

theplaz=> select fullname from accounts where balance > 75;
     fullname
------------------
```

```
 Alice Jones
 Alyssa P. Hacker
(2 rows)

theplaz=> select fullname from accounts where username = 'bitdiddl';
   fullname
---------------
 Ben Bitdiddle
(1 row)

theplaz=> select avg(balance) from accounts where balance > 70;
avg
---------------------
 78.0000000000000000
(1 row)
```

4.
```
theplaz=> update accounts set balance = balance-10 where username =
'jones';
UPDATE 1
theplaz=> update accounts set balance = balance+10 where username =
'mike';
UPDATE 1
```

5. They are different

A

```
theplaz=> select * from accounts;
 username |     fullname      | balance
----------+-------------------+---------
 bitdiddl | Ben Bitdiddle     |      65
 alyssa   | Alyssa P. Hacker  |      79
 jones    | Alice Jones       |      72
 mike     | Michael Dole      |      83
(4 rows)

theplaz=> insert into accounts values ('chuck', 'Charlie Robinson',
55);
INSERT 0 1
```

B

```
theplaz=> select * from accounts;
username |     fullname      | balance
----------+-------------------+---------
bitdiddl | Ben Bitdiddle     |      65
alyssa   | Alyssa P. Hacker  |      79
jones    | Alice Jones       |      72
mike     | Michael Dole      |      83
```

```
chuck     | Charlie Robinson |       55
(5 rows)
```

6. No, it does not include chuck. It only includes the rows that were in place on `BEGIN.`

7. Yes, the output now includes chuck, since a new transaction has started.

8. The command does not return anything nor create a new command line. This is different than the example in the P-Set where it displays UPDATE `1.`

9. The other transaction now completes and displays UPDATE `1.`

10. Mike's balance is now 73, which is 10 less than before.

11. No, the results do not change until a transaction is committed.

12. The transaction becomes visible when `COMMIT` is issued;

13. SERIALIZABLE has a performance implication. All of that locking makes it hard to get stuff done. In addition, having SERIALIZABLE correctness may not be important for the particular use. Thus, Postgres sets a default that is good for performance, and allows users to change it if they want.

6.033 L16 <span>4/9</span>

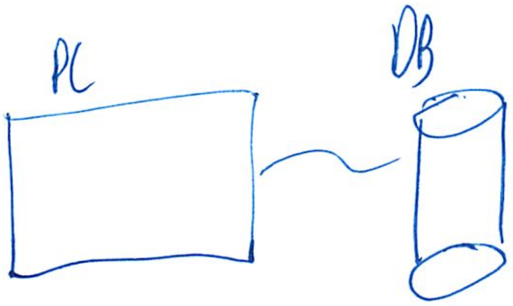
Logging

(Watching 4/15 on video - lecture recorded from this year)

Start simple + build up

1. Simple log	log
2. Write ahead	log + cell
3. Redo/undo logging	log + cell + cache

Setting



Goal: all or nothing
not really availability

Last week: did w/ shadow copy
- don't overwrite original till end (commit pt)
- Could add abort function

Can add transaction terms

- begin
- abort
- commit

Not everything is transactional

- like launching a rocket (can't abort)

Shadow copy downsides

- Single file only
- Slow performance (rewrite file)

Instead -> just write deltas

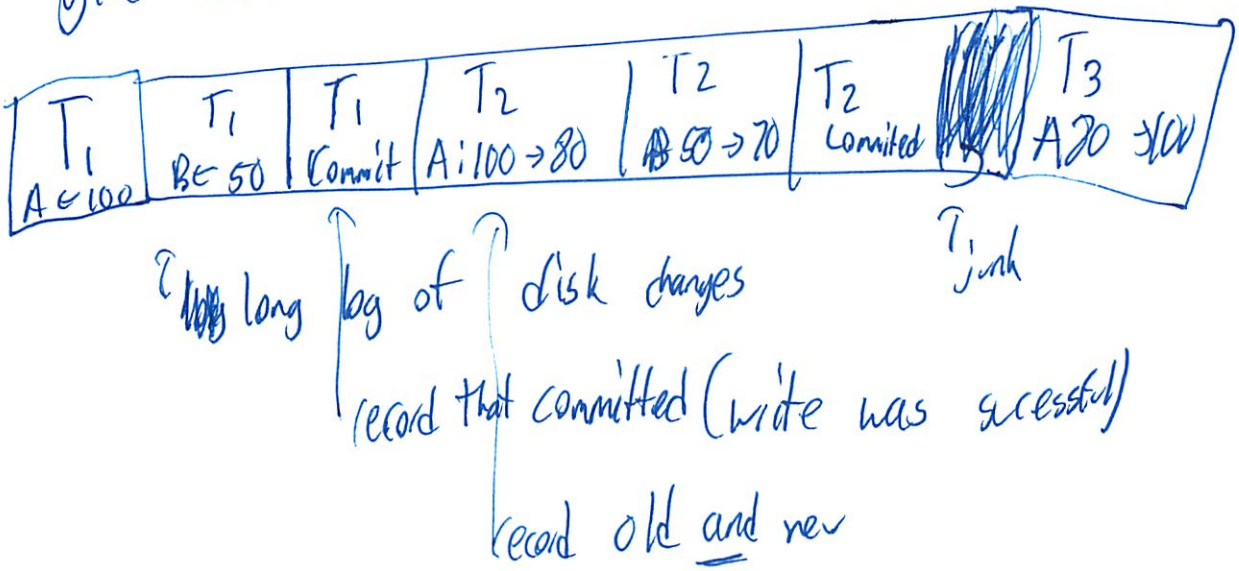
Example

```
Begin
    A ← 100
    B ← 50
End

Begin
    A ← A - 20
    B ← B + 20
End

Begin
    A ← A+30
Crash
```

3

Log of tranx

give each txn a #



Begin()

returns tranx id

Write()

appends to log

commit()

appends a commit record

read()

will just use log for now

*Log has enough info to recreate state of the world

Changes become visable on commit

L see read code on slide

ignores uncommited code

Changes must be applied atomically

L will say writing single disk block is atomic

~512 bytes

most disks actually do this -little capacitor

abort()

(never erase histore

L bad pratie

L can do -will talk later..)

Just leave

Since read will never Return

Later we will do something special

recovery()

don't need to do <u>anything</u>

just start running

other schemes different...

<u>But</u> read performance really sucks!

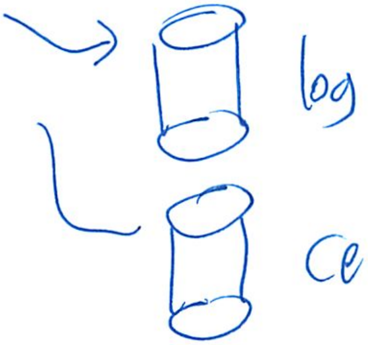
must scan whole disk!

Write is pretty good

↳ just append

How to fix → <u>Write Ahead Log</u>

Now 2 disks



cell stores the latest version

(this is helpful overview)

Note cell is just "slave" info

-not authoitative!

"logging" → appending to log

"installing" → writing to cell

read() from cells

write() → log first

then update cell storage

recover()

∟what is most nasty place to fail?

∟b/w logging + installing

just undo everything that has not been committed

?I don't fully get

Note log first

otherwise would update only copy (the cell)

recovery () code on slide

L if commit record → we know cell was updated

if no commit → roll back cell value

(since don't keep uncommited code)

[Demo from hands on]

[there it writes value immed - are all like that?

I think

But read history when other txn look

Thats what makes this ~~the~~ section hard...)

Note it records in log that aborted

L is very important

Since if
- T_3 → abort
- T_4 → 80→120, commit
- fail/crash

8

So then when we restore

~~We lost that T4 commit~~

T4 did sucessfully commit -> 120

T is correct

But then ~~undoing~~ restoring - it would see T3 and tried to roll it back

(wait what if didn't crash

↳ oh 2 open

↳ usually blocks)

Need to ignore sucessfully aborted code on Failure

So is this safe if we crash during recovery

If rewrite abort message -> doesn't matter

↳ just takes up space

↳ still idempotent

9 reads -> much better

Writes -> slightly worse off

But can we do better?

Add a <u>cache</u>

- write into cache, not cell or log
- <u>Then</u> write ~~all~~ to cell + log at commit point

(ahh... thats it)

So only write once to disk

<u>but</u> could cache fill up

L values can be flushed to cell <u>before</u> commit

```
read()
    L check cache
      then cell

write()  write log   <- log 1st!
         write cache
```

But can we have missing changes?

↳ write commit log entry

↳ but cell might not be up to date

  ↳ since only cache written

 * must fix during recovery

Also must undo info on cell disk as before

recovery()

  ↳ scans disk twice

  1. Undo uncommitted changes

  2. Redo committed changes

  (what was this called in the reading?)

  Undo/Redo (thought it was something else...)

But log is growing forever

- After data is in cell storage we don't really need

(kinda weird how you need to do all this ...
but they showed theirs no other way!)

Truncate()

- no pending transaction (must be committed or abborted)
  - cell storage ~~[illegible]~~
    - up to date

Flush cache

So write a checkpoint

- See when scanning backwards
  when we hit one we're done
  (well it tells you what is still open...)
  Only search those

(12)

Then delete the final stuff (non-pending)
↑ truncate

(He was describing if ~~all~~ ~~was~~ everything was done)

~~Could we~~ L not going to talk about partial stuff

People tried to do lots of other stuff
L just do logging!

L16: Logging

Frans Kaashoek & Nickolai Zeldovich
6.033 Spring 2012

All-or-nothing using shadow copy

```
xfer(bank, a, b, amt):
   copy(bank, tmp)
   tmp[a] = tmp[a] – amt
   tmp[b] = tmp[b] + amt
   rename(tmp, bank)
```

Shadow copy abort vs. commit

```
xfer(bank, a, b, amt):
   copy(bank, tmp)
   tmp[a] = tmp[a] – amt
   tmp[b] = tmp[b] + amt
   if tmp[a] < 0:
      print "Not enough funds"
      unlink(tmp)
   else:
      rename(tmp, bank)
```

Transaction terminology

```
xfer(bank, a, b, amt):
   begin
   bank[a] = bank[a] – amt
   bank[b] = bank[b] + amt
   if bank[a] < 0:
      print "Not enough funds"
      abort
   else:
      commit
```

4/1[illegible]2

Read with a log

```
read(log, var):
    commits = { }
    for record r in log[len(log)-1] .. log[0]:
        if (r.type == commit):
            commits = commits + r.tid
        if (r.type == update) and
           (r.tid in commits) and
           (r.var == var):
            return r.new_val
```

Read / write with cell storage

```
read(var):
    return cell_read(var)

write(var, value):
    log.append(cur_tid, update,
                   var, read(var), value)
    cell_write(var, value)
```

Recovering cell storage from log

```
recover(log):
    done = { }
    for record r in log[len(log)-1] .. log[0]:
        if r.type == commit or r.type == abort:
            done = done + r.tid
        if r.type == update and r.tid not in done:
            cell_write(r.var, r.old_val)     # undo
```

Recovering cell storage from log

```
recover(log):
    done = { }, aborted = { }
    for record r in log[len(log)-1] .. log[0]:
        if r.type == commit or r.type == abort:
            done = done + r.tid
        if r.type == update and r.tid not in done:
            cell_write(r.var, r.old_val)     # undo
            aborted = aborted + r.tid
    for tid in aborted: log.append(tid, abort)
```

Cached read / write

```
read(var):
   if var not in cache:
      # may evict others from cache to cell store
      cache[var] = cell_read(var)
   return cache[var]

write(var, value):
   log.append(cur_tid, update,
                  var, read(var), value)
   cache[var] = value
```

Recovery for cached writes

```
recover(log):
   done = { }
   for record r in log[len(log)-1] .. log[0]:
      if r.type == commit ~~or r.type == abort~~:
         done = done + r.tid
      if r.type == update and r.tid not in done:
         cell_write(r.var, r.old_val)     # undo

   for record r in log[0] .. log[len(log)-1]:
      if r.type == update and r.tid in done:
         cell_write(r.var, r.new_val)   # redo
```

4/9

```
6.033 2012 Lecture 16: Logging

Recall from last time:
  Two kinds of atomicity: all-or-nothing, before-or-after
  Shadow copy can provide all-or-nothing atomicity
    [ slide: shadow copy ]
    Golden rule of atomicity: never modify the only copy!
    Typical way to achieve all-or-nothing atomicity.
    Works because you can fall back to the old copy in case of failure.
  Software can also use all-or-nothing atomicity to abort in case of error.
    [ slide: shadow copy abort/commit ]
  Drawbacks of shadow file approach:
    - only works for single file (annoying but maybe fixable with shadow dirs)
    - copy the entire file for every all-or-nothing action (harder to avoid)
  Still, shadow copy is a simple and effective design when it suffices.
    Many Unix applications (e.g., text editors) use it, owing to rename.

Today, more general techniques for achieving all-or-nothing atomicity.
  [ slide: transaction syntax ]
  Idea: keep a log of all changes, and whether each change commits or aborts.
  We will start out with a simple scheme that's all-or-nothing but slow.
  Then, we will optimize its performance while preserving atomicity.

  Consider our bank account example again.
    Two accounts: A and B.
    Accounts start out empty.
  Run these all-or-nothing actions:
    begin
    A = 100
    B = 50
    commit

    begin
    A = A - 20
    B = B + 20
    commit

    begin
    A = A + 30
    --CRASH--

  What goes into the log?
    We assign every all-or-nothing action a unique transaction ID.
      Need to distinguish multiple actions in progress at the same time.
    Two kinds of records in the log:
      UPDATE records: both new and old value of some variable.
        (we'll see in a bit why we need the old values..)
      COMMIT/ABORT records: specify whether that action committed or aborted.
```

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

```
  What happens when a program runs now?
    begin: allocate a new transaction ID.
    write variable: append an entry to the log.
    read variable: scan the log looking for last committed value.
      [ slide: read with a log ]
      As an aside: how to see your own updates?
      Read uncommitted values from your own tid.
    commit: write a commit record.
      Expectedly, writing a commit record is the "commit point" for action,
        because of the way read works (looks for commit record).
      However, writing log records better be all-or-nothing.
      One approach, from last time: make each record fit within one sector.
    abort: do nothing (could write an abort record, but not strictly needed).
    recover from a crash: do nothing.

  Quick demo:
    rm DB LOG
    cat l16-demo.txt
    ./wal-sys < l16-demo.txt
    cat LOG

What's the performance of this log-only approach?
  Write performance is probably good: sequential writes, instead of random.
    (Since we aren't using the old values yet, we could have skipped the read.)
  Read performance is terrible: scan the log for every read!
  Crash recovery is instantaneous: nothing to do.

How can we optimize read performance?
  Keep both a log and "cell storage".
    Log is just as before: authoritative, provides all-or-nothing atomicity.
    Cell storage: provides fast reads, but cannot provide all-or-nothing.
    [ board: log, cell storage; updates going to both, read from cell storage ]
    We will say we "log" an update when it's written to the log.
    We will say we "install" an update when it's written to cell storage.
    [ slide: read/write with cell storage ]
   Two questions we have to answer now:
     - how to update both the log and cell storage when an update happens?
     - how to recover cell storage from the authoritative log after crash?

 Let's look at the above example in our situation.
   Log still contains the same things.
   As we're running, maintain cell storage for A and B.
   Except one problem: after crash, A's value in cell storage is wrong.
   Last action aborted (due to crash), but its changes to A are visible.
   We're going to have to repair this in our recovery function.
   Good thing we have the log to provide authoritative information.

 Ordering of logging and installing.
   Why does this matter?
     Because the two together don't have all-or-nothing atomicity.
     Can crash inbetween, so just one of the two might have taken place.
   What happens if we install first and then log?
     If we crash, no idea what happened to cell storage, or how to fix it.
     Bad idea, violates the golden rule ("Never modify the only copy").
   The corresponding rule for logging is the "Write-ahead-log protocol" (WAL).
     ==> Log the update before installing it. <==
   If we crash, log is authoritative and intact, can repair cell storage.
     (You can think of it as not being the only copy, once it's in the log.)

 Recovering cell storage.
   What happens if we log an update, install it, but then abort/crash?
   Need to undo that installed update.
   Plan: scan log, determine what actions aborted ("losers"), undo them.
   [ slide: recover cell storage from log ]
   Why do we have to scan backwards?
     Need to undo newest to oldest.
     Also need to find outcome of action before we decide whether to undo.
   In our example: done will be {1, 2}, we will set cellStorage[A] to 80.

   Quick demo:
     rm DB LOG
     cat l16-demo.txt
     ./wal-sys -undo < l16-demo.txt
     cat LOG
     cat DB
     ./wal-sys -undo
       show_state

 What if the programmer decides to abort explicitly, without crashing?
   Use the log to find all update records that were part of this action.
   Reset cell storage to old values from those records.
   Do we need to write an abort record, or can we skip & pretend we crashed?
     What if our example had a software abort instead of a crash?
     We might access A again later, and write an update record for it.
     After a crash, A will be undone to value before aborted action!
     So, need an abort record, to indicate that no undo is necessary.
   For the same reason, we need to record abort records after recovery.
     [ slide: recover with abort logging ]
     Otherwise, will keep rolling back to point before crashed action.

 What if we crash during recovery?
   Idempotent: can keep recovering over and over again.
   Crash during the undo phase: restarting is OK, will perform same undo.
   Crash during logging aborts: restarting is OK, duplicate aborts.

 What's the performance going to be like now?
   Writes might still be OK (but we do write twice: log & install).
   Reads are fast: just look up in cell storage.
   Recovery requires scanning the entire log

 Remaining performance problems:
   We have to write to disk twice.
   Scanning the log will take longer and longer, as the log grows.

 Optimization 1: defer installing updates, by storing them in a cache.
   [ slide: read/write with a cache ]
   writes can now be fast: just one write, instead of two.
     the hope is that variable is modified several times in cache before flush.
   reads go through the cache, since cache may contain more up-to-date values.
   atomicity problem: cell storage (on disk) may be out-of-date.
     is it possible to have changes that should be in cell storage, but aren't?
       yes: might not have flushed the latest commits.
     is it possible to have changes that shouldn't be in cell storage, but are?
       yes: flushed some changes that then aborted (same as before).
   during recovery, need to go through and re-apply changes to cell storage.
     undo every abort (even if it had an explicit record).
     redo every commit.
   [ slide: recovery for caching ]
     Don't treat actions with an abort record as "done".
       -> there might be leftover changes from them in cell storage.
     Re-do any actions that are committed (in the "done" set now).

 Optimization 2: truncate the log.
   Current design requires log to grow without bound: not practical.
```

```
  What part of the log can be discarded?
    We must know the outcome of every action in that part of log.
    Cell storage must reflect all of those log records (commits, aborts).
  Truncating mechanism (assuming no pending actions):
    Flush all cached updates to cell storage.
    Write a checkpoint record, to save our place in the log.
    Truncate log prior to checkpoint record.
    (Often log implemented as a series of files, so can delete old log files.)
  With pending actions, delete before checkpoint & earliest undecided record.

Back to the log records: why do we need all of those parts?
  ID: might need to distinguish between multiple actions at the same time.
  Undo: roll back losers, in case we wrote to cell storage before abort/crash.
  Redo: apply commits, in case we didn't write to cell storage before commit.

Summary.
  Logging is a general technique for achieving all-or-nothing atomicity.
  Widely used: databases, file systems, ..
  Can achieve reasonable performance with logging.
    Writes are always fast: sequential.
    Reads can be fast with cell storage.
    Key idea 1: write-ahead logging, makes it safe to update cell storage.
    Key idea 2: recovery protocol, undo losers / redo winners.

What's coming?
  Wednesday: dealing with concurrent actions, before-or-after atomicity.
  Cannot deal with external actions as part of an all-or-nothing action.
    E.g., dispensing money from an ATM: cannot undo or redo during recovery.
    Some hope: we'll talk about distributed transactions next Monday.
```

6.033: Computer Systems Engineering

Read 4/9

Spring 2012

Home / News
Schedule
Submissions
General Information
Staff List
Recitations
TA Office Hours
Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples
2011 Home

Preparation for Recitation 17

Read *The Design and Implementation of a Log-Structured File System*.

In today's reading, the authors depart radically from UNIX File System (presented in the UNIX paper and in Section 2.5 of the notes) and propose, instead, a file system in which all of the data is stored in *an append-only log*, that is, a flat file that can be modified only by having data added to the end of it. In Chapter 9, we also hear about logs, specifically how they help achieve *reliability*. For today's reading, the purpose of the log is to achieve good *performance*. hmm

like DP

To understand why an append-only log might lead to high performance, we quickly review the basics of disks (which you should recall from class). Data on a disk is stored on a constantly spinning platter and read by a head that floats above the platter. The platter is broken down into concentric rings called tracks; each track is divided into sectors. The time required to read or write data to a disk can be broken down into three components: moving the head to the correct track, waiting for the correct sector to rotate under the head, and transferring the data off the disk. The first two operations (known collectively as a "seek") are very expensive (and aren't getting cheaper very fast): together, they can take 10 or more milliseconds. Once the disk head is at the correct location, it can transfer data relatively quickly (at around 40 MB/s on today's high-end disks or 10 MB/s on lower-quality disks).

Now, given this description, the goal of a log-structured file system is to minimize seeks by treating the disk as an infinite append-only log. For example, the file system software simply appends new files to the end of the log. For this strategy to translate into higher performance, the file system software must:

- Write in large batches to amortize the cost of a seek (if any).
- Always start writing from the last place it stopped writing.
- Avoid reading; reads are likely to cause a seek.

but how read?

As you read the paper, consider how the authors achieve these three goals. Keep in mind that the "infinite log" is actually on a finite disk and ask how that constraint affects their goals. Also, ask yourself whether certain file access patterns by applications might make it hard for a log-structured file system to avoid seeks.

yes

haha

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

Top // **6.033 home** //

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720
mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

Abstract

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-structured file system called Sprite LFS; it outperforms current Unix file systems by an order of magnitude for small-file writes while matching or exceeding Unix performance for reads and large writes. Even when the overhead for cleaning is included, Sprite LFS can use 70% of the disk bandwidth for writing, whereas Unix file systems typically can use only 5-10%.

FB uses to store photos

recompact

?

Picking + choosing stats

1. Introduction

Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly. This trend is likely to continue in the future and it will cause more and more applications to become disk-bound. To lessen the impact of this problem, we have devised a new disk storage management technique called a *log-structured file system*, which uses disks an order of magnitude more efficiently than current file systems.

Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. A log-structured file system writes all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by eliminating almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log.

I dont think so

The notion of logging is not new, and a number of recent file systems have incorporated a log as an auxiliary structure to speed up writes and crash recovery[2, 3]. However, these other systems use the log only for temporary storage; the permanent home for information is in a traditional random-access storage structure on disk. In contrast, a log-structured file system stores data permanently in the log: there is no other structure on disk. The log contains indexing information so that files can be read back with efficiency comparable to current file systems.

but I guess could be better than random

For a log-structured file system to operate efficiently, it must ensure that there are always large extents of free space available for writing new data. This is the most difficult challenge in the design of a log-structured file system. In this paper we present a solution based on large extents called *segments*, where a *segment cleaner* process continually regenerates empty segments by compressing the live data from heavily fragmented segments. We used a simulator to explore different cleaning policies and discovered a simple but effective algorithm based on cost and benefit: it segregates older, more slowly changing data from young rapidly-changing data and treats them differently during cleaning.

lots of fragmentation

We have constructed a prototype log-structured file system called Sprite LFS, which is now in production use as part of the Sprite network operating system[4]. Benchmark programs demonstrate that the raw writing speed of Sprite LFS is more than an order of magnitude greater than that of Unix for small files. Even for other workloads, such

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

This paper will appear in the *Proceedings of the 13th ACM Symposium on Operating Systems Principles* and the February 1992 *ACM Transactions on Computer Systems*.

as those including reads and large-file accesses, Sprite LFS is at least as fast as Unix in all cases but one (files read sequentially after being written randomly). We also measured the long-term overhead for cleaning in the production system. Overall, Sprite LFS permits about 65-75% of a disk's raw bandwidth to be used for writing new data (the rest is used for cleaning). For comparison, Unix systems can only utilize 5-10% of a disk's raw bandwidth for writing new data; the rest of the time is spent seeking.

The remainder of this paper is organized into six sections. Section 2 reviews the issues in designing file systems for computers of the 1990's. Section 3 discusses the design alternatives for a log-structured file system and derives the structure of Sprite LFS, with particular focus on the cleaning mechanism. Section 4 describes the crash recovery system for Sprite LFS. Section 5 evaluates Sprite LFS using benchmark programs and long-term measurements of cleaning overhead. Section 6 compares Sprite LFS to other file systems, and Section 7 concludes.

2. Design for file systems of the 1990's

File system design is governed by two general forces: technology, which provides a set of basic building blocks, and workload, which determines a set of operations that must be carried out efficiently. This section summarizes technology changes that are underway and describes their impact on file system design. It also describes the workloads that influenced the design of Sprite LFS and shows how current file systems are ill-equipped to deal with the workloads and technology changes.

2.1. Technology

Three components of technology are particularly significant for file system design: processors, disks, and main memory. Processors are significant because their speed is increasing at a nearly exponential rate, and the improvements seem likely to continue through much of the 1990's. This puts pressure on all the other elements of the computer system to speed up as well, so that the system doesn't become unbalanced.

Disk technology is also improving rapidly, but the improvements have been primarily in the areas of cost and capacity rather than performance. There are two components of disk performance: transfer bandwidth and access time. Although both of these factors are improving, the rate of improvement is much slower than for CPU speed. Disk transfer bandwidth can be improved substantially with the use of disk arrays and parallel-head disks[5] but no major improvements seem likely for access time (it is determined by mechanical motions that are hard to improve). If an application causes a sequence of small disk transfers separated by seeks, then the application is not likely to experience much speedup over the next ten years, even with faster processors.

The third component of technology is main memory, which is increasing in size at an exponential rate. Modern file systems cache recently-used file data in main memory, and larger main memories make larger file caches possible. This has two effects on file system behavior. First, larger file caches alter the workload presented to the disk by absorbing a greater fraction of the read requests[1, 6]. Most write requests must eventually be reflected on disk for safety, so disk traffic (and disk performance) will become more and more dominated by writes.

The second impact of large file caches is that they can serve as write buffers where large numbers of modified blocks can be collected before writing any of them to disk. Buffering may make it possible to write the blocks more efficiently, for example by writing them all in a single sequential transfer with only one seek. Of course, write-buffering has the disadvantage of increasing the amount of data lost during a crash. For this paper we will assume that crashes are infrequent and that it is acceptable to lose a few seconds or minutes of work in each crash; for applications that require better crash recovery, non-volatile RAM may be used for the write buffer.

2.2. Workloads

Several different file system workloads are common in computer applications. One of the most difficult workloads for file system designs to handle efficiently is found in office and engineering environments. Office and engineering applications tend to be dominated by accesses to small files; several studies have measured mean file sizes of only a few kilobytes[1, 6-8]. Small files usually result in small random disk I/Os, and the creation and deletion times for such files are often dominated by updates to file system "metadata" (the data structures used to locate the attributes and blocks of the file).

Workloads dominated by sequential accesses to large files, such as those found in supercomputing environments, also pose interesting problems, but not for file system software. A number of techniques exist for ensuring that such files are laid out sequentially on disk, so I/O performance tends to be limited by the bandwidth of the I/O and memory subsystems rather than the file allocation policies. In designing a log-structured file system we decided to focus on the efficiency of small-file accesses, and leave it to hardware designers to improve bandwidth for large-file accesses. Fortunately, the techniques used in Sprite LFS work well for large files as well as small ones.

2.3. Problems with existing file systems

Current file systems suffer from two general problems that make it hard for them to cope with the technologies and workloads of the 1990's. First, they spread information around the disk in a way that causes too many small accesses. For example, the Berkeley Unix fast file system (Unix FFS)[9] is quite effective at laying out each file sequentially on disk, but it physically separates different files. Furthermore, the attributes ("inode") for a file are separate from the file's contents, as is the directory entry containing the file's name. It takes at least five separate disk I/Os, each preceded by a seek, to create a new file in

Unix FFS: two different accesses to the file's attributes plus one access each for the file's data, the directory's data, and the directory's attributes. When writing small files in such a system, less than 5% of the disk's potential bandwidth is used for new data; the rest of the time is spent seeking.

The second problem with current file systems is that they tend to write synchronously: the application must wait for the write to complete, rather than continuing while the write is handled in the background. For example even though Unix FFS writes file data blocks asynchronously, file system metadata structures such as directories and inodes are written synchronously. For workloads with many small files, the disk traffic is dominated by the synchronous metadata writes. Synchronous writes couple the application's performance to that of the disk and make it hard for the application to benefit from faster CPUs. They also defeat the potential use of the file cache as a write buffer. Unfortunately, network file systems like NFS[10] have introduced additional synchronous behavior where it didn't used to exist. This has simplified crash recovery, but it has reduced write performance.

Throughout this paper we use the Berkeley Unix fast file system (Unix FFS) as an example of current file system design and compare it to log-structured file systems. The Unix FFS design is used because it is well documented in the literature and used in several popular Unix operating systems. The problems presented in this section are not unique to Unix FFS and can be found in most other file systems.

3. Log-structured file systems

The fundamental idea of a log-structured file system is to improve write performance by buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially in a single disk write operation. The information written to disk in the write operation includes file data blocks, attributes, index blocks, directories, and almost all the other information used to manage the file system. For workloads that contain many small files, a log-structured file system converts the many small synchronous random writes of traditional file systems into large asynchronous sequential transfers that can utilize nearly 100% of the raw disk bandwidth.

Although the basic idea of a log-structured file system is simple, there are two key issues that must be resolved to achieve the potential benefits of the logging approach. The first issue is how to retrieve information from the log; this is the subject of Section 3.1 below. The second issue is how to manage the free space on disk so that large extents of free space are always available for writing new data. This is a much more difficult issue; it is the topic of Sections 3.2-3.6. Table 1 contains a summary of the on-disk data structures used by Sprite LFS to solve the above problems; the data structures are discussed in detail in later sections of the paper.

3.1. File location and reading

Although the term ''log-structured'' might suggest that sequential scans are required to retrieve information from the log, this is not the case in Sprite LFS. Our goal was to match or exceed the read performance of Unix FFS. To accomplish this goal, Sprite LFS outputs index structures in the log to permit random-access retrievals. The basic structures used by Sprite LFS are identical to those used in Unix FFS: for each file there exists a data structure called an *inode*, which contains the file's attributes (type, owner, permissions, etc.) plus the disk addresses of the first ten blocks of the file; for files larger than ten blocks, the inode also contains the disk addresses of one or more *indirect blocks*, each of which contains the addresses of more data or indirect blocks. Once a file's inode has been found, the number of disk I/Os required to read the file is identical in Sprite LFS and Unix FFS.

In Unix FFS each inode is at a fixed location on disk; given the identifying number for a file, a simple calculation

Data structure	Purpose	Location	Section
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log	3.1
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log	3.1
Indirect block	Locates blocks of large files.	Log	3.1
Segment summary	Identifies contents of segment (file number and offset for each block).	Log	3.2
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log	3.6
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed	None
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed	4.1
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log	4.2

Table 1 — Summary of the major data structures stored on disk by Sprite LFS.
For each data structure the table indicates the purpose served by the data structure in Sprite LFS. The table also indicates whether the data structure is stored in the log or at a fixed position on disk and where in the paper the data structure is discussed in detail. Inodes, indirect blocks, and superblocks are similar to the Unix FFS data structures with the same names. Note that Sprite LFS contains neither a bitmap nor a free list.

still inodes

new inode map

yields the disk address of the file's inode. In contrast, Sprite LFS doesn't place inodes at fixed positions; they are written to the log. Sprite LFS uses a data structure called an *inode map* to maintain the current location of each inode. Given the identifying number for a file, the inode map must be indexed to determine the disk address of the inode. The inode map is divided into blocks that are written to the log; a fixed checkpoint region on each disk identifies the locations of all the inode map blocks. Fortunately, inode maps are compact enough to keep the active portions cached in main memory: inode map lookups rarely require disk accesses.

Figure 1 shows the disk layouts that would occur in Sprite LFS and Unix FFS after creating two new files in different directories. Although the two layouts have the same logical structure, the log-structured file system produces a much more compact arrangement. As a result, the write performance of Sprite LFS is much better than Unix FFS, while its read performance is just as good.

3.2. Free space management: segments

The most difficult design issue for log-structured file systems is the management of free space. The goal is to maintain large free extents for writing new data. Initially all the free space is in a single extent on disk, but by the time the log reaches the end of the disk the free space will have been fragmented into many small extents corresponding to the files that were deleted or overwritten.

From this point on, the file system has two choices: threading and copying. These are illustrated in Figure 2. The first alternative is to leave the live data in place and thread the log through the free extents. Unfortunately, threading will cause the free space to become severely fragmented, so that large contiguous writes won't be possible and a log-structured file system will be no faster than traditional file systems. The second alternative is to copy live data out of the log in order to leave large free extents for writing. For this paper we will assume that the live data is written back in a compacted form at the head of the log; it could also be moved to another log-structured file system to form a hierarchy of logs, or it could be moved to some totally different file system or archive. The disadvantage of copying is its cost, particularly for long-lived files; in the simplest case where the log works circularly across the disk and live data is copied back into the log, all of the long-lived files will have to be copied in every pass of the log across the disk.

Sprite LFS uses a combination of threading and copying. The disk is divided into large fixed-size extents called *segments*. Any given segment is always written sequentially from its beginning to its end, and all live data must be copied out of a segment before the segment can be rewritten. However, the log is threaded on a segment-by-segment basis; if the system can collect long-lived data together into segments, those segments can be skipped over so that the data doesn't have to be copied repeatedly. The segment size is chosen large enough that the transfer time to read or write a whole segment is much greater than the cost of a seek to the beginning of the segment. This allows whole-segment operations to run at nearly the full bandwidth of the disk, regardless of the order in which segments are accessed. Sprite LFS currently uses segment sizes of either 512 kilobytes or one megabyte.

3.3. Segment cleaning mechanism

so

The process of copying live data out of a segment is called *segment cleaning*. In Sprite LFS it is a simple three-step process: read a number of segments into memory, identify the live data, and write the live data back to a smaller number of clean segments. After this

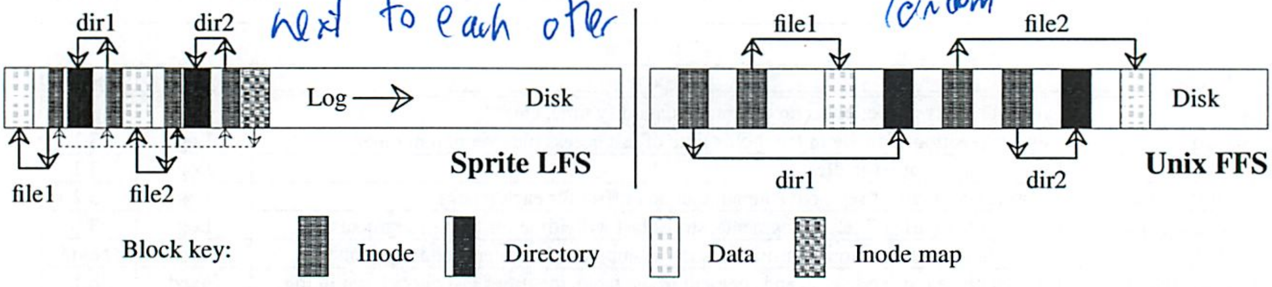


Figure 1 — A comparison between Sprite LFS and Unix FFS.
This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named `dir1/file1` and `dir2/file2`. Each system must write new data blocks and inodes for `file1` and `file2`, plus new data blocks and inodes for the containing directories. Unix FFS requires ten non-sequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations.

operation is complete, the segments that were read are marked as clean, and they can be used for new data or for additional cleaning.

As part of segment cleaning it must be possible to identify which blocks of each segment are live, so that they can be written out again. It must also be possible to identify the file to which each block belongs and the position of the block within the file; this information is needed in order to update the file's inode to point to the new location of the block. Sprite LFS solves both of these problems by writing a *segment summary block* as part of each segment. The summary block identifies each piece of information that is written in the segment; for example, for each file data block the summary block contains the file number and block number for the block. Segments can contain multiple segment summary blocks when more than one log write is needed to fill the segment. (Partial-segment writes occur when the number of dirty blocks buffered in the file cache is insufficient to fill a segment.) Segment summary blocks impose little overhead during writing, and they are useful during crash recovery (see Section 4) as well as during cleaning.

Seems too complex

Sprite LFS also uses the segment summary information to distinguish live blocks from those that have been overwritten or deleted. Once a block's identity is known, its liveness can be determined by checking the file's inode or indirect block to see if the appropriate block pointer still refers to this block. If it does, then the block is live; if it doesn't, then the block is dead. Sprite LFS optimizes this check slightly by keeping a version number in the inode map entry for each file; the version number is incremented whenever the file is deleted or truncated to length zero. The version number combined with the inode number form an unique identifier (uid) for the contents of the file. The segment summary block records this uid for each block in the segment; if the uid of a block does not match the uid currently stored in the inode map when the segment is cleaned, the block can be discarded immediately without examining the file's inode.

This approach to cleaning means that there is no free-block list or bitmap in Sprite. In addition to saving memory and disk space, the elimination of these data structures also simplifies crash recovery. If these data structures existed, additional code would be needed to log changes to the structures and restore consistency after crashes.

3.4. Segment cleaning policies

Given the basic mechanism described above, four policy issues must be addressed:

(1) When should the segment cleaner execute? Some possible choices are for it to run continuously in background at low priority, or only at night, or only when disk space is nearly exhausted.

(2) How many segments should it clean at a time? Segment cleaning offers an opportunity to reorganize data on disk; the more segments cleaned at once, the more opportunities to rearrange.

(3) Which segments should be cleaned? An obvious choice is the ones that are most fragmented, but this turns out not to be the best choice.

(4) How should the live blocks be grouped when they are written out? One possibility is to try to enhance the locality of future reads, for example by grouping files in the same directory together into a single output segment. Another possibility is to sort the blocks by the time they were last modified and group blocks of similar age together into new segments; we call this approach *age sort.*

This is just silly...

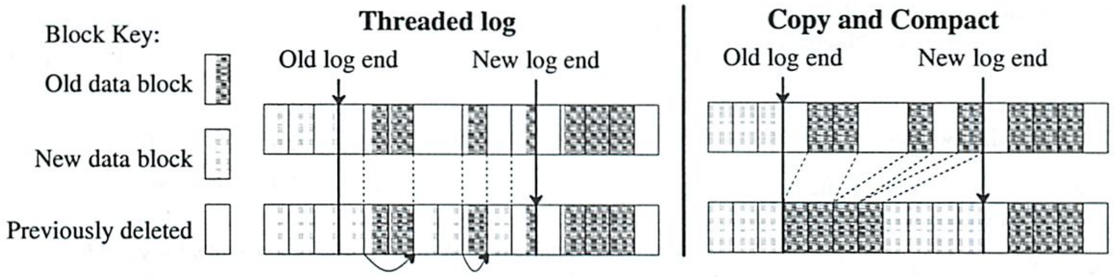


Figure 2 — Possible free space management solutions for log-structured file systems.
In a log-structured file system, free space for the log can be generated either by copying the old blocks or by threading the log around the old blocks. The left side of the figure shows the threaded log approach where the log skips over the active blocks and overwrites blocks of files that have been deleted or overwritten. Pointers between the blocks of the log are maintained so that the log can be followed during crash recovery. The right side of the figure shows the copying scheme where log space is generated by reading the section of disk after the end of the log and rewriting the active blocks of that section along with the new data into the newly generated space.

In our work so far we have not methodically addressed the first two of the above policies. Sprite LFS starts cleaning segments when the number of clean segments drops below a threshold value (typically a few tens of segments). It cleans a few tens of segments at a time until the number of clean segments surpasses another threshold value (typically 50-100 clean segments). The overall performance of Sprite LFS does not seem to be very sensitive to the exact choice of the threshold values. In contrast, the third and fourth policy decisions are critically important: in our experience they are the primary factors that determine the performance of a log-structured file system. The remainder of Section 3 discusses our analysis of which segments to clean and how to group the live data.

We use a term called *write cost* to compare cleaning policies. The write cost is the average amount of time the disk is busy per byte of new data written, including all the cleaning overheads. The write cost is expressed as a multiple of the time that would be required if there were no cleaning overhead and the data could be written at its full bandwidth with no seek time or rotational latency. A write cost of 1.0 is perfect: it would mean that new data could be written at the full disk bandwidth and there is no cleaning overhead. A write cost of 10 means that only one-tenth of the disk's maximum bandwidth is actually used for writing new data; the rest of the disk time is spent in seeks, rotational latency, or cleaning.

For a log-structured file system with large segments, seeks and rotational latency are negligible both for writing and for cleaning, so the write cost is the total number of bytes moved to and from the disk divided by the number of those bytes that represent new data. This cost is determined by the utilization (the fraction of data still live) in the segments that are cleaned. In the steady state, the cleaner must generate one clean segment for every segment of new data written. To do this, it reads N segments in their entirety and writes out N*u segments of live data (where u is the utilization of the segments and $0 \le u < 1$). This creates N*(1−u) segments of contiguous free space for new data. Thus

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u} \qquad (1)$$

In the above formula we made the conservative assumption that a segment must be read in its entirety to recover the live blocks; in practice it may be faster to read just the live blocks, particularly if the utilization is very low (we haven't tried this in Sprite LFS). If a segment to be cleaned has no live blocks ($u = 0$) then it need not be read at all and the write cost is 1.0.

Figure 3 graphs the write cost as a function of u. For reference, Unix FFS on small-file workloads utilizes at most 5-10% of the disk bandwidth, for a write cost of 10-20 (see [11] and Figure 8 in Section 5.1 for specific measurements). With logging, delayed writes, and disk request sorting this can probably be improved to about 25% of the bandwidth[12] or a write cost of 4. Figure 3 suggests that the segments cleaned must have a utilization of less than .8 in order for a log-structured file system to outperform the current Unix FFS; the utilization must be less than .5 to outperform an improved Unix FFS.

It is important to note that the utilization discussed above is not the overall fraction of the disk containing live data; it is just the fraction of live blocks in segments that are cleaned. Variations in file usage will cause some segments to be less utilized than others, and the cleaner can choose the least utilized segments to clean; these will have lower utilization than the overall average for the disk.

Even so, the performance of a log-structured file system can be improved by reducing the overall utilization of the disk space. With less of the disk in use the segments that are cleaned will have fewer live blocks resulting in a lower write cost. Log-structured file systems provide a cost-performance tradeoff: if disk space is underutilized, higher performance can be achieved but at a high cost per usable byte; if disk capacity utilization is increased, storage costs are reduced but so is performance. Such a tradeoff

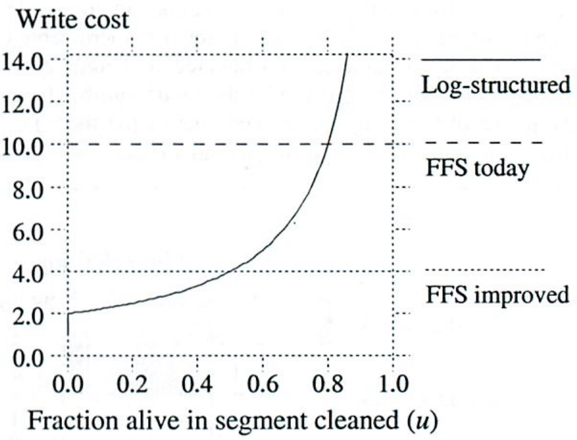


Figure 3 — Write cost as a function of *u* for small files. In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned the more disk bandwidth that is needed for cleaning and not available for writing new data. The figure also shows two reference points: "FFS today", which represents Unix FFS today, and "FFS improved", which is our estimate of the best performance possible in an improved Unix FFS. Write cost for Unix FFS is not sensitive to the amount of disk space in use.

between performance and space utilization is not unique to log-structured file systems. For example, Unix FFS only allows 90% of the disk space to be occupied by files. The remaining 10% is kept free to allow the space allocation algorithm to operate efficiently.

The key to achieving high performance at low cost in a log-structured file system is to force the disk into a bimodal segment distribution where most of the segments are nearly full, a few are empty or nearly empty, and the cleaner can almost always work with the empty segments. This allows a high overall disk capacity utilization yet provides a low write cost. The following section describes how we achieve such a bimodal distribution in Sprite LFS.

3.5. Simulation results

We built a simple file system simulator so that we could analyze different cleaning policies under controlled conditions. The simulator's model does not reflect actual file system usage patterns (its model is much harsher than reality), but it helped us to understand the effects of random access patterns and locality, both of which can be exploited to reduce the cost of cleaning. The simulator models a file system as a fixed number of 4-kbyte files, with the number chosen to produce a particular overall disk capacity utilization. At each step, the simulator overwrites one of the files with new data, using one of two pseudo-random access patterns:

Uniform Each file has equal likelihood of being selected in each step.

Hot-and-cold Files are divided into two groups. One group contains 10% of the files; it is called *hot* because its files are selected 90% of the time. The other group is called *cold*; it contains 90% of the files but they are selected only 10% of the time. Within groups each file is equally likely to be selected. This access pattern models a simple form of locality.

In this approach the overall disk capacity utilization is constant and no read traffic is modeled. The simulator runs until all clean segments are exhausted, then simulates the actions of a cleaner until a threshold number of clean segments is available again. In each run the simulator was allowed to run until the write cost stabilized and all cold-start variance had been removed.

Figure 4 superimposes the results from two sets of simulations onto the curves of Figure 3. In the ''LFS uniform'' simulations the uniform access pattern was used. The cleaner used a simple greedy policy where it always chose the least-utilized segments to clean. When writing out live data the cleaner did not attempt to re-organize the data: live blocks were written out in the same order that they appeared in the segments being cleaned (for a uniform access pattern there is no reason to expect any improvement from re-organization).

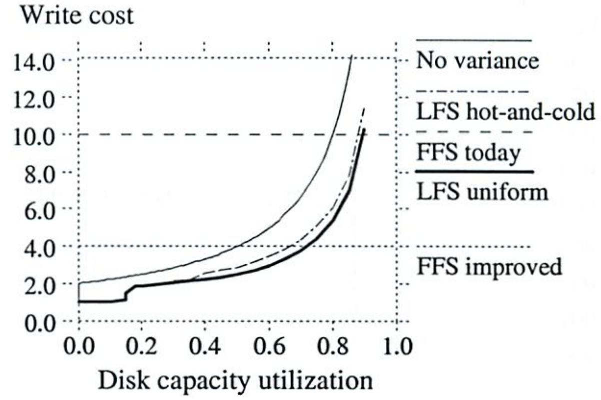


Figure 4 — Initial simulation results.
The curves labeled ''FFS today'' and ''FFS improved'' are reproduced from Figure 3 for comparison. The curve labeled ''No variance'' shows the write cost that would occur if all segments always had exactly the same utilization. The ''LFS uniform'' curve represents a log-structured file system with uniform access pattern and a greedy cleaning policy: the cleaner chooses the least-utilized segments. The ''LFS hot-and-cold'' curve represents a log-structured file system with locality of file access. It uses a greedy cleaning policy and the cleaner also sorts the live data by age before writing it out again. The x-axis is overall disk capacity utilization, which is not necessarily the same as the utilization of the segments being cleaned.

Even with uniform random access patterns, the variance in segment utilization allows a substantially lower write cost than would be predicted from the overall disk capacity utilization and formula (1). For example, at 75% overall disk capacity utilization, the segments cleaned have an average utilization of only 55%. At overall disk capacity utilizations under 20% the write cost drops below 2.0; this means that some of the cleaned segments have no live blocks at all and hence don't need to be read in.

The ''LFS hot-and-cold'' curve shows the write cost when there is locality in the access patterns, as described above. The cleaning policy for this curve was the same as for ''LFS uniform'' except that the live blocks were sorted by age before writing them out again. This means that long-lived (cold) data tends to be segregated in different segments from short-lived (hot) data; we thought that this approach would lead to the desired bimodal distribution of segment utilizations.

Figure 4 shows the surprising result that locality and ''better'' grouping result in *worse* performance than a system with no locality! We tried varying the degree of locality (e.g. 95% of accesses to 5% of data) and found that performance got worse and worse as the locality increased. Figure 5 shows the reason for this non-intuitive result. Under the greedy policy, a segment doesn't get cleaned until it becomes the least utilized of all segments. Thus every segment's utilization eventually drops to the cleaning threshold, including the cold segments. Unfortunately, the

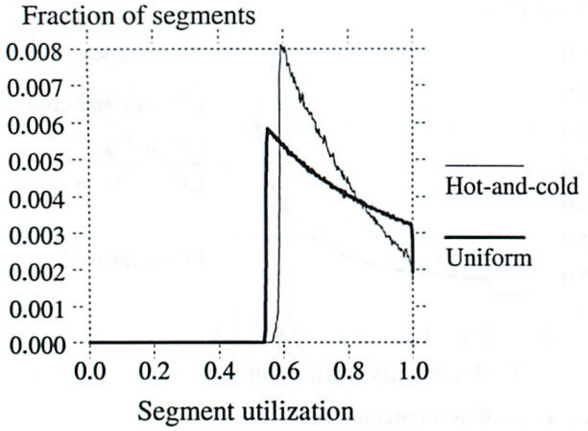


Figure 5 — Segment utilization distributions with greedy cleaner.
These figures show distributions of segment utilizations of the disk during the simulation. The distribution is computed by measuring the utilizations of all segments on the disk at the points during the simulation when segment cleaning was initiated. The distribution shows the utilizations of the segments available to the cleaning algorithm. Each of the distributions corresponds to an overall disk capacity utilization of 75%. The ''Uniform'' curve corresponds to ''LFS uniform'' in Figure 4 and ''Hot-and-cold'' corresponds to ''LFS hot-and-cold'' in Figure 4. Locality causes the distribution to be more skewed towards the utilization at which cleaning occurs; as a result, segments are cleaned at a higher average utilization.

utilization drops very slowly in cold segments, so these segments tend to linger just above the cleaning point for a very long time. Figure 5 shows that many more segments are clustered around the cleaning point in the simulations with locality than in the simulations without locality. The overall result is that cold segments tend to tie up large numbers of free blocks for long periods of time.

After studying these figures we realized that hot and cold segments must be treated differently by the cleaner. Free space in a cold segment is more valuable than free space in a hot segment because once a cold segment has been cleaned it will take a long time before it re-accumulates the unusable free space. Said another way, once the system reclaims the free blocks from a segment with cold data it will get to ''keep'' them a long time before the cold data becomes fragmented and ''takes them back again.'' In contrast, it is less beneficial to clean a hot segment because the data will likely die quickly and the free space will rapidly re-accumulate; the system might as well delay the cleaning a while and let more of the blocks die in the current segment. The value of a segment's free space is based on the stability of the data in the segment. Unfortunately, the stability cannot be predicted without knowing future access patterns. Using an assumption that the older the data in a segment the longer it is likely to remain unchanged, the stability can be estimated by the age of data.

To test this theory we simulated a new policy for selecting segments to clean. The policy rates each segment according to the benefit of cleaning the segment and the cost of cleaning the segment and chooses the segments with the highest ratio of benefit to cost. The benefit has two components: the amount of free space that will be reclaimed and the amount of time the space is likely to stay free. The amount of free space is just $1-u$, where u is the utilization of the segment. We used the most recent modified time of any block in the segment (ie. the age of the youngest block) as an estimate of how long the space is likely to stay free. The benefit of cleaning is the space-time product formed by multiplying these two components. The cost of cleaning the segment is $1+u$ (one unit of cost to read the segment, u to write back the live data). Combining all these factors, we get

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated * age of data}}{\text{cost}} = \frac{(1-u)\text{*age}}{1+u}$$

We call this policy the *cost-benefit* policy; it allows cold segments to be cleaned at a much higher utilization than hot segments.

We re-ran the simulations under the hot-and-cold access pattern with the cost-benefit policy and age-sorting

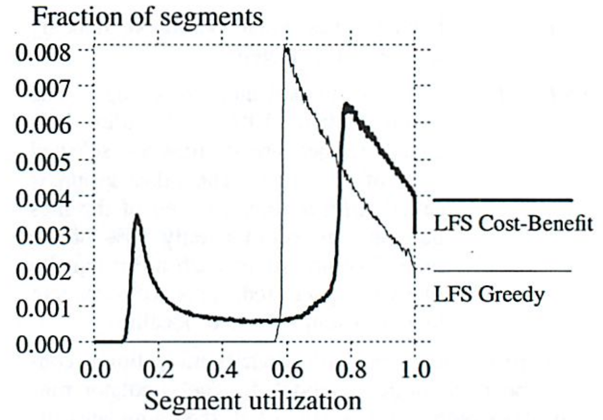


Figure 6 — Segment utilization distribution with cost-benefit policy.
This figure shows the distribution of segment utilizations from the simulation of a hot-and-cold access pattern with 75% overall disk capacity utilization. The ''LFS Cost-Benefit'' curve shows the segment distribution occurring when the cost-benefit policy is used to select segments to clean and live blocks grouped by age before being re-written. Because of this bimodal segment distribution, most of the segments cleaned had utilizations around 15%. For comparison, the distribution produced by the greedy method selection policy is shown by the ''LFS Greedy'' curve reproduced from Figure 5.

on the live data. As can be seen from Figure 6, the cost-benefit policy produced the bimodal distribution of segments that we had hoped for. The cleaning policy cleans cold segments at about 75% utilization but waits until hot segments reach a utilization of about 15% before cleaning them. Since 90% of the writes are to hot files, most of the segments cleaned are hot. Figure 7 shows that the cost-benefit policy reduces the write cost by as much as 50% over the greedy policy, and a log-structured file system out-performs the best possible Unix FFS even at relatively high disk capacity utilizations. We simulated a number of other degrees and kinds of locality and found that the cost-benefit policy gets even better as locality increases.

The simulation experiments convinced us to implement the cost-benefit approach in Sprite LFS. As will be seen in Section 5.2, the behavior of actual file systems in Sprite LFS is even better than predicted in Figure 7.

3.6. Segment usage table

In order to support the cost-benefit cleaning policy, Sprite LFS maintains a data structure called the *segment usage table*. For each segment, the table records the number of live bytes in the segment and the most recent modified time of any block in the segment. These two values are used by the segment cleaner when choosing segments to clean. The values are initially set when the segment is written, and the count of live bytes is decremented when files are deleted or blocks are overwritten. If the count falls to zero then the segment can be reused without cleaning. The blocks of the segment usage table are written to the log, and the addresses of the blocks are stored in the checkpoint regions (see Section 4 for details).

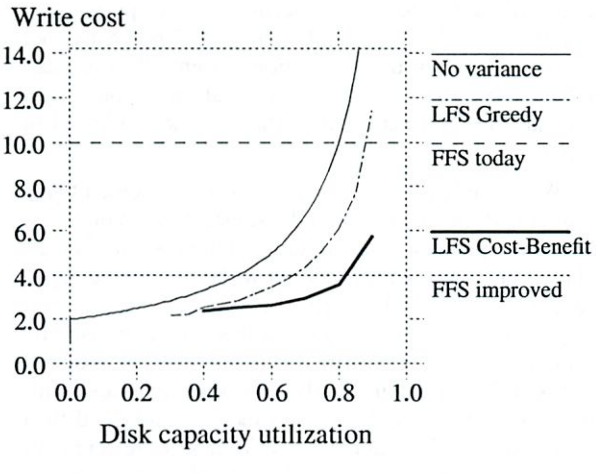


Figure 7 — Write cost, including cost-benefit policy.
This graph compares the write cost of the greedy policy with that of the cost-benefit policy for the hot-and-cold access pattern. The cost-benefit policy is substantially better than the greedy policy, particularly for disk capacity utilizations above 60%.

In order to sort live blocks by age, the segment summary information records the age of the youngest block written to the segment. At present Sprite LFS does not keep modified times for each block in a file; it keeps a single modified time for the entire file. This estimate will be incorrect for files that are not modified in their entirety. We plan to modify the segment summary information to include modified times for each block.

4. Crash recovery

When a system crash occurs, the last few operations performed on the disk may have left it in an inconsistent state (for example, a new file may have been written without writing the directory containing the file); during reboot the operating system must review these operations in order to correct any inconsistencies. In traditional Unix file systems without logs, the system cannot determine where the last changes were made, so it must scan all of the metadata structures on disk to restore consistency. The cost of these scans is already high (tens of minutes in typical configurations), and it is getting higher as storage systems expand.

by looking for discrepancies

In a log-structured file system the locations of the last disk operations are easy to determine: they are at the end of the log. Thus it should be possible to recover very quickly after crashes. This benefit of logs is well known and has been used to advantage both in database systems[13] and in other file systems[2, 3, 14]. Like many other logging systems, Sprite LFS uses a two-pronged approach to recovery: *checkpoints*, which define consistent states of the file system, and *roll-forward*, which is used to recover information written since the last checkpoint.

4.1. Checkpoints

A checkpoint is a position in the log at which all of the file system structures are consistent and complete. Sprite LFS uses a two-phase process to create a checkpoint. First, it writes out all modified information to the log, including file data blocks, indirect blocks, inodes, and blocks of the inode map and segment usage table. Second, it writes a *checkpoint region* to a special fixed position on disk. The checkpoint region contains the addresses of all the blocks in the inode map and segment usage table, plus the current time and a pointer to the last segment written.

During reboot, Sprite LFS reads the checkpoint region and uses that information to initialize its main-memory data structures. In order to handle a crash during a checkpoint operation there are actually two checkpoint regions, and checkpoint operations alternate between them. The checkpoint time is in the last block of the checkpoint region, so if the checkpoint fails the time will not be updated. During reboot, the system reads both checkpoint regions and uses the one with the most recent time.

Sprite LFS performs checkpoints at periodic intervals as well as when the file system is unmounted or the system is shut down. A long interval between checkpoints reduces the overhead of writing the checkpoints but increases the time needed to roll forward during recovery; a short checkpoint interval improves recovery time but increases the cost of normal operation. Sprite LFS currently uses a checkpoint interval of thirty seconds, which is probably much too short. An alternative to periodic checkpointing is to perform checkpoints after a given amount of new data has been written to the log; this would set a limit on recovery time while reducing the checkpoint overhead when the file system is not operating at maximum throughput.

4.2. Roll-forward

In principle it would be safe to restart after crashes by simply reading the latest checkpoint region and discarding any data in the log after that checkpoint. This would result in instantaneous recovery but any data written since the last checkpoint would be lost. In order to recover as much information as possible, Sprite LFS scans through the log segments that were written after the last checkpoint. This operation is called *roll-forward*.

During roll-forward Sprite LFS uses the information in segment summary blocks to recover recently-written file data. When a summary block indicates the presence of a new inode, Sprite LFS updates the inode map it read from the checkpoint, so that the inode map refers to the new copy of the inode. This automatically incorporates the file's new data blocks into the recovered file system. If data blocks are discovered for a file without a new copy of the file's inode, then the roll-forward code assumes that the new version of the file on disk is incomplete and it ignores the new data blocks.

The roll-forward code also adjusts the utilizations in the segment usage table read from the checkpoint. The utilizations of the segments written since the checkpoint will be zero; they must be adjusted to reflect the live data left after roll-forward. The utilizations of older segments will also have to be adjusted to reflect file deletions and overwrites (both of these can be identified by the presence of new inodes in the log).

The final issue in roll-forward is how to restore consistency between directory entries and inodes. Each inode contains a count of the number of directory entries referring to that inode; when the count drops to zero the file is deleted. Unfortunately, it is possible for a crash to occur when an inode has been written to the log with a new reference count while the block containing the corresponding directory entry has not yet been written, or vice versa.

To restore consistency between directories and inodes, Sprite LFS outputs a special record in the log for each directory change. The record includes an operation code (create, link, rename, or unlink), the location of the directory entry (i-number for the directory and the position within the directory), the contents of the directory entry (name and i-number), and the new reference count for the inode named in the entry. These records are collectively called the *directory operation log*; Sprite LFS guarantees that each directory operation log entry appears in the log before the corresponding directory block or inode.

During roll-forward, the directory operation log is used to ensure consistency between directory entries and inodes: if a log entry appears but the inode and directory block were not both written, roll-forward updates the directory and/or inode to complete the operation. Roll-forward operations can cause entries to be added to or removed from directories and reference counts on inodes to be updated. The recovery program appends the changed directories, inodes, inode map, and segment usage table blocks to the log and writes a new checkpoint region to include them. The only operation that can't be completed is the creation of a new file for which the inode is never written; in this case the directory entry will be removed. In addition to its other functions, the directory log made it easy to provide an atomic rename operation.

The interaction between the directory operation log and checkpoints introduced additional synchronization issues into Sprite LFS. In particular, each checkpoint must represent a state where the directory operation log is consistent with the inode and directory blocks in the log. This required additional synchronization to prevent directory modifications while checkpoints are being written.

5. Experience with the Sprite LFS

We began the implementation of Sprite LFS in late 1989 and by mid-1990 it was operational as part of the Sprite network operating system. Since the fall of 1990 it has been used to manage five different disk partitions, which are used by about thirty users for day-to-day computing. All of the features described in this paper have been implemented in Sprite LFS, but roll-forward has not yet been installed in the production system. The production disks use a short checkpoint interval (30 seconds) and discard all the information after the last checkpoint when they reboot.

When we began the project we were concerned that a log-structured file system might be substantially more complicated to implement than a traditional file system. In reality, however, Sprite LFS turns out to be no more complicated than Unix FFS[9]: Sprite LFS has additional complexity for the segment cleaner, but this is compensated by the elimination of the bitmap and layout policies required by Unix FFS; in addition, the checkpointing and roll-forward code in Sprite LFS is no more complicated than the *fsck* code[15] that scans Unix FFS disks to restore consistency. Logging file systems like Episode[2] or Cedar[3] are likely to be somewhat more complicated than either Unix FFS or Sprite LFS, since they include both logging and layout code.

In everyday use Sprite LFS does not feel much different to the users than the Unix FFS-like file system in Sprite. The reason is that the machines being used are not fast enough to be disk-bound with the current workloads. For example on the modified Andrew benchmark[11],

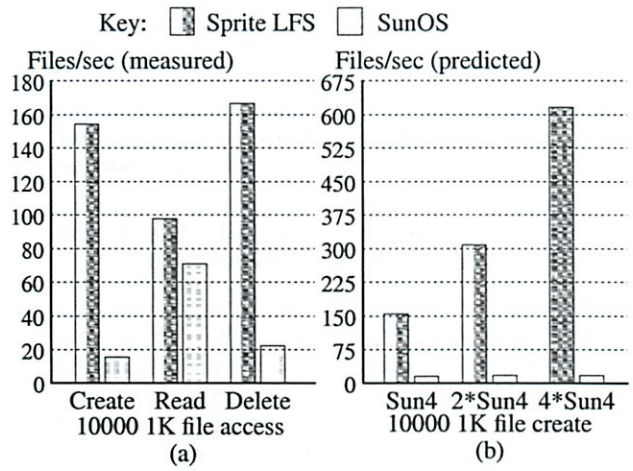


Figure 8 — Small-file performance under Sprite LFS and SunOS.
Figure (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. Figure (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

Sprite LFS is only 20% faster that SunOS using the configuration presented in Section 5.1. Most of the speedup is attributable to the removal of the synchronous writes in Sprite LFS. Even with the synchronous writes of Unix FFS, the benchmark has a CPU utilization of over 80%, limiting the speedup possible from changes in the disk storage management.

5.1. Micro-benchmarks

We used a collection of small benchmark programs to measure the best-case performance of Sprite LFS and compare it to SunOS 4.0.3, whose file system is based on Unix FFS. The benchmarks are synthetic so they do not represent realistic workloads, but they illustrate the strengths and weaknesses of the two file systems. The machine used for both systems was a Sun-4/260 (8.7 integer SPECmarks) with 32 megabytes of memory, a Sun SCSI3 HBA, and a Wren IV disk (1.3 MBytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both LFS and SunOS, the disk was formatted with a file system having around 300 megabytes of usable storage. An eight-kilobyte block size was used by SunOS while Sprite LFS used a four-kilobyte block size and a one-megabyte segment size. In each case the system was running multiuser but was otherwise quiescent during the test. For Sprite LFS no cleaning occurred during the benchmark runs so the measurements represent best-case performance; see Section 5.2 below for measurements of cleaning overhead.

Figure 8 shows the results of a benchmark that creates, reads, and deletes a large number of small files. Sprite LFS is almost ten times as fast as SunOS for the create and delete phases of the benchmark. Sprite LFS is also faster for reading the files back; this is because the files are read in the same order created and the log-structured file system packs the files densely in the log. Furthermore, Sprite LFS only kept the disk 17% busy during the create phase while saturating the CPU. In contrast, SunOS kept the disk busy 85% of the time during the create phase, even though only about 1.2% of the disk's potential bandwidth was used for new data. This means that the performance of Sprite LFS will improve by another factor of 4-6 as CPUs get faster (see Figure 8(b)). Almost no improvement can be expected in SunOS.

Although Sprite was designed for efficiency on workloads with many small file accesses, Figure 9 shows that it also provides competitive performance for large files. Sprite LFS has a higher write bandwidth than SunOS in all cases. It is substantially faster for random writes because it turns them into sequential writes to the log; it is also faster for sequential writes because it groups many blocks into a single large I/O, whereas SunOS performs

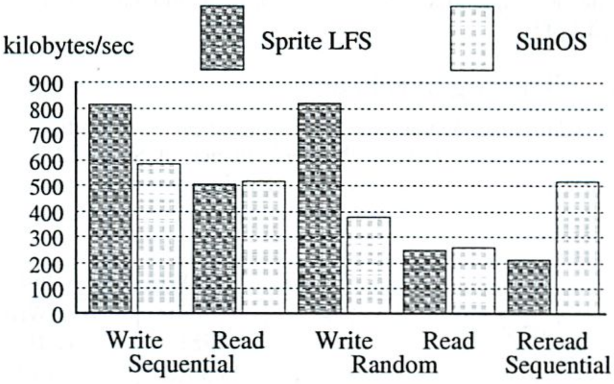


Figure 9 — Large-file performance under Sprite LFS and SunOS.
The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

individual disk operations for each block (a newer version of SunOS groups writes [16] and should therefore have performance equivalent to Sprite LFS). The read performance is similar in the two systems except for the case of reading a file sequentially after it has been written randomly; in this case the reads require seeks in Sprite LFS, so its performance is substantially lower than SunOS.

Figure 9 illustrates the fact that a log-structured file system produces a different form of locality on disk than traditional file systems. A traditional file system achieves *logical locality* by assuming certain access patterns (sequential reading of files, a tendency to use multiple files within a directory, etc.); it then pays extra on writes, if necessary, to organize information optimally on disk for the assumed read patterns. In contrast, a log-structured file system achieves *temporal locality*: information that is created or modified at the same time will be grouped closely on disk. If temporal locality matches logical locality, as it does for a file that is written sequentially and then read sequentially, then a log-structured file system should have about the same performance on large files as a traditional file system. If temporal locality differs from logical locality then the systems will perform differently. Sprite LFS handles random writes more efficiently because it writes them sequentially on disk. SunOS pays more for the random writes in order to achieve logical locality, but then it handles sequential re-reads more efficiently. Random reads have about the same performance in the two systems, even though the blocks are laid out very differently. However, if the nonsequential reads occurred in the same order as the nonsequential writes then Sprite would have been much faster.

5.2. Cleaning overheads

The micro-benchmark results of the previous section give an optimistic view of the performance of Sprite LFS because they do not include any cleaning overheads (the write cost during the benchmark runs was 1.0). In order to assess the cost of cleaning and the effectiveness of the cost-benefit cleaning policy, we recorded statistics about our production log-structured file systems over a period of several months. Five systems were measured:

`/user6` Home directories for Sprite developers. Workload consists of program development, text processing, electronic communication, and simulations.

`/pcs` Home directories and project area for research on parallel processing and VLSI circuit design.

`/src/kernel`
Sources and binaries for the Sprite kernel.

`/swap2` Sprite client workstation swap files. Workload consists of virtual memory backing store for 40 diskless Sprite workstations. Files tend to be large, sparse, and accessed nonsequentially.

`/tmp` Temporary file storage area for 40 Sprite workstations.

Table 2 shows statistics gathered during cleaning over a four-month period. In order to eliminate start-up effects we waited several months after putting the file systems into use before beginning the measurements. The behavior of the production file systems has been substantially better than predicted by the simulations in Section 3. Even though the overall disk capacity utilizations ranged from 11-75%, more than half of the segments cleaned were totally empty. Even the non-empty segments have utilizations far less than the average disk utilizations. The overall write costs ranged from 1.2 to 1.6, in comparison to write costs of 2.5-3 in the corresponding simulations. Figure 10 shows the distribution of segment utilizations, gathered in a recent snapshot of the `/user6` disk.

We believe that there are two reasons why cleaning costs are lower in Sprite LFS than in the simulations. First,

Write cost in Sprite LFS file systems

File system	Disk Size	Avg File Size	Avg Write Traffic	In Use	Segments		*u*	Write
					Cleaned	Empty	Avg	Cost
`/user6`	1280 MB	23.5 KB	3.2 MB/hour	75%	10732	69%	.133	1.4
`/pcs`	990 MB	10.5 KB	2.1 MB/hour	63%	22689	52%	.137	1.6
`/src/kernel`	1280 MB	37.5 KB	4.2 MB/hour	72%	16975	83%	.122	1.2
`/tmp`	264 MB	28.9 KB	1.7 MB/hour	11%	2871	78%	.130	1.3
`/swap2`	309 MB	68.1 KB	13.3 MB/hour	65%	4701	66%	.535	1.6

Table 2 - Segment cleaning statistics and write costs for production file systems.
For each Sprite LFS file system the table lists the disk size, the average file size, the average daily write traffic rate, the average disk capacity utilization, the total number of segments cleaned over a four-month period, the fraction of the segments that were empty when cleaned, the average utilization of the non-empty segments that were cleaned, and the overall write cost for the period of the measurements. These write cost figures imply that the cleaning overhead limits the long-term write performance to about 70% of the maximum sequential write bandwidth.

all the files in the simulations were just a single block long. In practice, there are a substantial number of longer files, and they tend to be written and deleted as a whole. This results in greater locality within individual segments. In the best case where a file is much longer than a segment, deleting the file will produce one or more totally empty segments. The second difference between simulation and reality is that the simulated reference patterns were evenly distributed within the hot and cold file groups. In practice there are large numbers of files that are almost never written (cold segments in reality are much colder than the cold segments in the simulations). A log-structured file system will isolate the very cold files in segments and never clean them. In the simulations, every segment eventually received modifications and thus had to be cleaned.

If the measurements of Sprite LFS in Section 5.1 were a bit over-optimistic, the measurements in this section are, if anything, over-pessimistic. In practice it may be possible to perform much of the cleaning at night or during other idle periods, so that clean segments are available during bursts of activity. We do not yet have enough experience with Sprite LFS to know if this can be done. In addition, we expect the performance of Sprite LFS to improve as we gain experience and tune the algorithms. For example, we have not yet carefully analyzed the policy issue of how many segments to clean at a time, but we think it may impact the system's ability to segregate hot data from cold data.

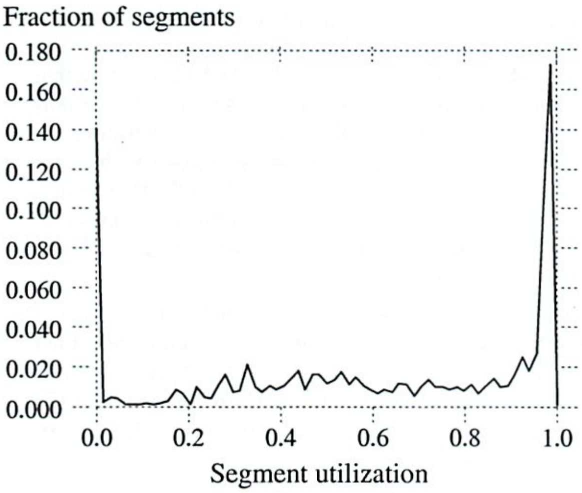


Figure 10 — Segment utilization in the /user6 file system
This figure shows the distribution of segment utilizations in a recent snapshot of the `/user6` disk. The distribution shows large numbers of fully utilized segments and totally empty segments.

5.3. Crash recovery

Although the crash recovery code has not been installed on the production system, the code works well enough to time recovery of various crash scenarios. The time to recover depends on the checkpoint interval and the rate and type of operations being performed. Table 3 shows the recovery time for different file sizes and amounts of file data recovered. The different crash configurations were generated by running a program that created one, ten, or fifty megabytes of fixed-size files before the system was crashed. A special version of Sprite LFS was used that had an infinite checkpoint interval and never wrote directory changes to disk. During the recovery roll-forward, the created files had to be added to the inode map, the directory entries created, and the segment usage table updated.

Table 3 shows that recovery time varies with the number and size of files written between the last checkpoint and the crash. Recovery times can be bounded by limiting the amount of data written between checkpoints. From the average file sizes and daily write traffic in Table 2, a checkpoint interval as large as an hour would result in average recovery times of around one second. Using the maximum observered write rate of 150 megabytes/hour, maximum recovery time would grow by one second for every 70 seconds of checkpoint interval length.

5.4. Other overheads in Sprite LFS

Table 4 shows the relative importance of the various kinds of data written to disk, both in terms of how much of the live blocks they occupy on disk and in terms of how much of the data written to the log they represent. More than 99% of the live data on disk consists of file data blocks and indirect blocks. However, about 13% of the information written to the log consists of inodes, inode map blocks, and segment map blocks, all of which tend to be overwritten quickly. The inode map alone accounts for more than 7% of all the data written to the log. We suspect that this is because of the short checkpoint interval currently used in Sprite LFS, which forces metadata to disk

Sprite LFS recovery time in seconds			
File	File Data Recovered		
Size	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

Table 3 — Recovery time for various crash configurations
The table shows the speed of recovery of one, ten, and fifty megabytes of fixed-size files. The system measured was the same one used in Section 5.1. Recovery time is dominated by the number of files to be recovered.

more often than necessary. We expect the log bandwidth overhead for metadata to drop substantially when we install roll-forward recovery and increase the checkpoint interval.

6. Related work

The log-structured file system concept and the Sprite LFS design borrow ideas from many different storage management systems. File systems with log-like structures have appeared in several proposals for building file systems on write-once media[17, 18]. Besides writing all changes in an append-only fashion, these systems maintain indexing information much like the Sprite LFS inode map and inodes for quickly locating and reading files. They differ from Sprite LFS in that the write-once nature of the media made it unnecessary for the file systems to reclaim log space.

The segment cleaning approach used in Sprite LFS acts much like scavenging garbage collectors developed for programming languages[19]. The cost-benefit segment selection and the age sorting of blocks during segment cleaned in Sprite LFS separates files into generations much like generational garbage collection schemes[20]. A significant difference between these garbage collection schemes and Sprite LFS is that efficient random access is possible in the generational garbage collectors, whereas sequential accesses are necessary to achieve high performance in a file system. Also, Sprite LFS can exploit the fact that blocks can belong to at most one file at a time to use much simpler algorithms for identifying garbage than used in the systems for programming languages.

The logging scheme used in Sprite LFS is similar to schemes pioneered in database systems. Almost all database systems use write-ahead logging for crash recovery and high performance[13], but differ from Sprite LFS in how they use the log. Both Sprite LFS and the database systems view the log as the most up to date ''truth'' about the state of the data on disk. The main difference is that database systems do not use the log as the final repository for data: a separate data area is reserved for this purpose. The separate data area of these database systems means that they do not need the segment cleaning mechanisms of the Sprite LFS to reclaim log space. The space occupied by the log in a database system can be reclaimed when the logged changes have been written to their final locations. Since all read requests are processed from the data area, the log can be greatly compacted without hurting read performance. Typically only the changed bytes are written to database logs rather than entire blocks as in Sprite LFS.

Sprite LFS `/user6` file system contents		
Block type	Live data	Log bandwidth
Data blocks*	98.0%	85.2%
Indirect blocks*	1.0%	1.6%
Inode blocks*	0.2%	2.7%
Inode map	0.2%	7.8%
Seg Usage map*	0.0%	2.1%
Summary blocks	0.6%	0.5%
Dir Op Log	0.0%	0.1%

Table 4 — Disk space and log bandwidth usage of /user6
For each block type, the table lists the percentage of the disk space in use on disk (Live data) and the percentage of the log bandwidth consumed writing this block type (Log bandwidth). The block types marked with '*' have equivalent data structures in Unix FFS.

The Sprite LFS crash recovery mechanism of checkpoints and roll forward using a ''redo log'' is similar to techniques used in database systems and object repositories[21]. The implementation in Sprite LFS is simplified because the log is the final home of the data. Rather than redoing the operation to the separate data copy, Sprite LFS recovery insures that the indexes point at the newest copy of the data in the log.

Collecting data in the file cache and writing it to disk in large writes is similar to the concept of group commit in database systems[22] and to techniques used in main-memory database systems[23, 24].

7. Conclusion

The basic principle behind a log-structured file system is a simple one: collect large amounts of new data in a file cache in main memory, then write the data to disk in a single large I/O that can use all of the disk's bandwidth. Implementing this idea is complicated by the need to maintain large free areas on disk, but both our simulation analysis and our experience with Sprite LFS suggest that low cleaning overheads can be achieved with a simple policy based on cost and benefit. Although we developed a log-structured file system to support workloads with many small files, the approach also works very well for large-file accesses. In particular, there is essentially no cleaning overhead at all for very large files that are created and deleted in their entirety.

The bottom line is that a log-structured file system can use disks an order of magnitude more efficiently than existing file systems. This should make it possible to take advantage of several more generations of faster processors before I/O limitations once again threaten the scalability of computer systems.

8. Acknowledgments

Diane Greene, Mary Baker, John Hartman, Mike Kupfer, Ken Shirriff and Jim Mott-Smith provided helpful comments on drafts of this paper.

References

1. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James

G. Thompson, ''A Trace-Driven Analysis of the Unix 4.2 BSD File System,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 15-24 ACM, (1985).

2. Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas, ''DEcorum File System Architectural Overview,'' *Proceedings of the USENIX 1990 Summer Conference*, pp. 151-164 (Jun 1990).

3. Robert B. Hagmann, ''Reimplementing the Cedar File System Using Logging and Group Commit,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 155-162 (Nov 1987).

4. John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, ''The Sprite Network Operating System,'' *IEEE Computer* **21**(2) pp. 23-36 (1988).

5. David A. Patterson, Garth Gibson, and Randy H. Katz, ''A Case for Redundant Arrays of Inexpensive Disks (RAID),'' *ACM SIGMOD 88*, pp. 109-116 (Jun 1988).

6. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, ''Measurements of a Distributed File System,'' *Proceedings of the 13th Symposium on Operating Systems Principles*, ACM, (Oct 1991).

7. M. Satyanarayanan, ''A Study of File Sizes and Functional Lifetimes,'' *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 96-108 ACM, (1981).

8. Edward D. Lazowska, John Zahorjan, David R Cheriton, and Willy Zwaenepoel, ''File Access Performance of Diskless Workstations,'' *Transactions on Computer Systems* **4**(3) pp. 238-268 (Aug 1986).

9. Marshall K. McKusick, ''A Fast File System for Unix,'' *Transactions on Computer Systems* **2**(3) pp. 181-197 ACM, (1984).

10. R. Sandberg, ''Design and Implementation of the Sun Network Filesystem,'' *Proceedings of the USENIX 1985 Summer Conference*, pp. 119-130 (Jun 1985).

11. John K. Ousterhout, ''Why Aren't Operating Systems Getting Faster As Fast as Hardware?,'' *Proceedings of the USENIX 1990 Summer Conference*, pp. 247-256 (Jun 1990).

12. Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout, ''Disk Scheduling Revisited,'' *Proceedings of the Winter 1990 USENIX Technical Conference*, (January 1990).

13. Jim Gray, ''Notes on Data Base Operating Systems,'' in *Operating Systems, An Advanced Course*, Springer-Verlag (1979).

14. A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, ''Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors,'' *IBM Journal of Research and Development* **34**(1) pp. 105-109 (Jan 1990).

15. Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry, ''Fsck - The UNIX File System Check Program,'' *Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version*, USENIX, (Apr 1986).

16. Larry McVoy and Steve Kleiman, ''Extent-like Performance from a UNIX File System,'' *Proceedings of the USENIX 1991 Winter Conference*, (Jan 1991).

17. D. Reed and Liba Svobodova, ''SWALLOW: A Distributed Data Storage System for a Local Network,'' *Local Networks for Computer Communications*, pp. 355-373 North-Holland, (1981).

18. Ross S. Finlayson and David R. Cheriton, ''Log Files: An Extended File Service Exploiting Write-Once Storage,'' *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 129-148 ACM, (Nov 1987).

19. H. G. Baker, ''List Processing in Real Time on a Serial Computer,'' A.I. Working Paper 139, MIT-AI Lab, Boston, MA (April 1977).

20. Henry Lieberman and Carl Hewitt, ''A Real-Time Garbage Collector Based on the Lifetimes of Objects,'' *Communications of the ACM* **26**(6) pp. 419-429 (1983).

21. Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler, ''Reliable Object Storage to Support Atomic Actions,'' *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 147-159 ACM, (1985).

22. David J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, Mike R. Stonebraker, and David Wood, ''Implementation Techniques for Main Memory Database Systems,'' *Proceedings of SIGMOD 1984*, pp. 1-8 (Jun 1984).

23. Kenneth Salem and Hector Garcia-Molina, ''Crash Recovery Mechanisms for Main Storage Database Systems,'' CS-TR-034-86, Princeton University, Princeton, NJ (1986).

24. Robert B. Hagmann, ''A Crash Recovery Scheme for a Memory-Resident Database System,'' *IEEE Transactions on Computers* **C-35**(9)(Sep 1986).

6.033 Recitation

9/10

(2 min late)

DP1 Being Graded

did you handel loops, esp if not versioning

Log File System

SSD

no moving parts

non volatile store like disks

Much more expensive

Can't get as close to disks

Not clear if will take over $\frac{\text{capacity}}{\text{price}}$

Must charge up capacitor to erase a block

- slower
- or it erases around it

Only erase a block 100,000 times

Bitmap to show which blocks are good + bad

- But only 0->1 which is allowed
  - 1->0 is expensive/limited

Similar to a log based system

- But we don't care about contigeous free space
  - or large writes at all time (a track)

LFS

- Minimal read/write head movement
- Write to a track or cyl
- Writes are more common

Prof: Reads occur 10x often as a write

- Caching is a false assumption
  - 99.9% hit rate on microprocessor caches
  - ↳ a lot more regular

Adds iNode Map to point to which is live

Where do you store it?

- in memory
  - but then need to rebuild on crash
  - was too big in the 90s
- So do start from the begining "roll forward" and recover
  - ~~even slower~~ much faster than Unix file system's fsck recovery
- write checkpoints occasionally

~2 writes for each block: directory + inode

Buffer the writes

If crash, lose updates, but still consistent

How much to buffer in mem?

- Depends on your prob of crashing
- And optimal amt to write

Called a segment

Size depends on disk

Flash - doesn't matter

But you might run out of space!

Mac + Windows use journal file systems

Log the metadata, but not the data

- inode map
- ts

Garbage Collection

traditional mark everything that is valid → Threading

but that does not give Contigeas space

But where copy the stuff to?

They do threading and segments.

linked list
of free segments

Go through segments + compact

If can combine/compact segments

Paper says look ~ 10 segments at once

At the end of each segment is segment summary table

- which blocks are in use and which arent
- looks at inode versions in inode map
- So table says which inodes point to which segment block

When segment is 95% good - trying to compact is waste of time

So look for segments that are mostly garbage

They picked a threshhold to ~~clear sto~~ compact statements

Threshholds are always a bit sketchy

So LFS ended w/ a lot of blocks just over the threshhold

So look at cost benefit threshhold

- How full is the disk?
- They didn't look at this

Checkpoint - Write the iNode table and other structures to disk

Write in 2 fixed known locations

Write alt times

- even, odd, w/ ts

So can recover if crash during checkpoint

Roll forward from there

Was kinda implemented once

L had trouble w/ recovery

But journals are a good idea

So why do we read this paper?

Since we talk about logs

Prof "Which are awesome"

But don't over use logs

Logs

This was 1st use of logs for performance

6.033

4/11

L 17 Transactions

(still need to watch last lecture)
└ on all or nothing

Today: Before or after

└ If $ transfer, whole thing runs before
or aftter an another entire txn

Did w/ locks before spring break
- but do it auto - ~~[illegible]~~ txn system
programer doesn't think about it

Could just do in serial order
└ lock whole system
- lecture over!

But we want some parallelism

②

for example $A \xrightarrow{Txn} B$

$D \xrightarrow{Txn} E$) can run concurrently

Read set and write sets

- Shared write set → worry
  → watch for approriate order
- Read set is in write set of other Txn ?
  - You have a decision
  - Diff ways to solve

Lets try for correctness 1st

ex) A = 100 B = 50

Txn 1: XFR(A, B, 10)

int (.1)

One must run before the other

So either

XFR 1st	Int 1st
A = 99	A = 100
B = 66	B = 65

③

So any system we builds should have 1 of 2 outcomes

Schedule

XfR	RA 100	RW 90			RB 50	WB 60		
int			RA 90	WA 99			RB 60	WB 66

✓ Works good schedule

Schedule 2

XfR	RA 100		WA 90		RB 50	WB 60		
int		RA 100		WA 110			RB 60	WB 66

⊗ Not legal bad schedule

So whats' the problem here?

- intersecting write sets
- Conflicting updates of A W ↔ W
  - └ also possible (not here) R ↔ W

So we need to avoid these conflicting updates

Conflict serializable schedule

∀pairs T_i, T_j

∀pairs T_{io}, T_{jo} ← objects Transaction manipulates

Must be ordered in the same way in $T_i \underset{\uparrow \text{and}}{*} T_j$

Conflicting write sets - shouldn't precede

Postgres pauses 2nd Txn

Various Semantics

1. Strict Seralizablity
   - easy to understand
   - Slow

Willing to give up strictness for performance

2. Snapshot isolation
   - Our Hands On

3. Read - Committed

- ^{Txn}Started later but committed
- ~~[illegible]~~
- better performance

Depends on the property of your program

Clear inside of Txn system needs locks

When release lock?

Xfr RA WA ReleaseA RB WB
90 65

⊗ wrong

int RAWA RBWB
99 55

Correct → when you commit

Do we still have parallelism?

(missed)

Also a problem w/ deadlocks

if look at accounts in wrong order

Xfr lock A → can't percede

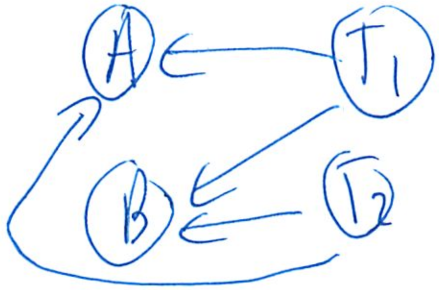
deadlock

int lock B → can't procede

Possible sols

- always acquire locks in fixed order
- hirarchical lock
  - 1 big lock
- don't take locks till commit
  - destroy before + after symantics
- abort one
  - frees up its lock

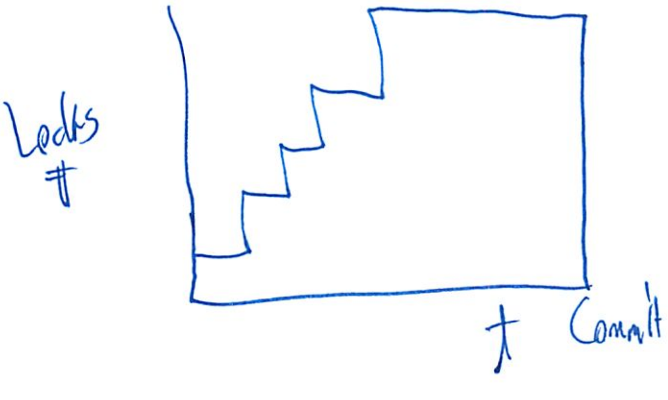
Detect: wait dependency graph



See a cycle → possibility of deadlock

Or time out
- abort some txn
- only works if short txn

Can we do better? better parallism



Care about read/write conflicts?
└ if all we do is read, we can do that in parallel

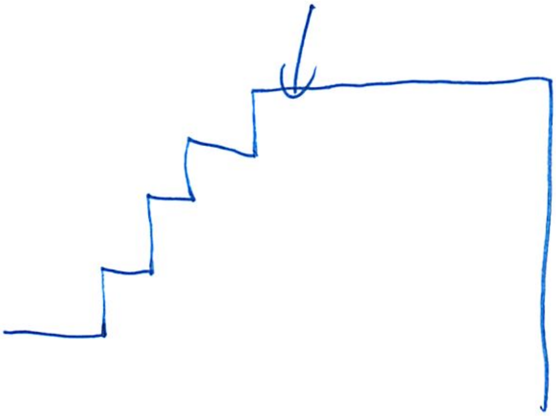
⑧

So split into read/write locks

So can we release read lock before commit?

L since not doing anything else w/ A

So we can release A



L since using value from when lock acq

(need to think about more...)

Can't release write lock

L since if rollback/abort

would undo the update

other people could have tried to update it

it would see uncommitted values

key point: Ok to release read, but not write at commit pt

L> Straight two phase locking

L17: Isolation

Frans Kaashoek & Nickolai Zeldovich
6.033 Spring 2012

Concurrent actions

```
xfer(a, b, amt):
    begin
    a = a – amt
    b = b + amt
    commit

interest(rate):
    begin
    for each account x:
        x = x * (1+rate)
    commit
```

Locking protocol

```
read(var):
    if var.lock not held:
        acquire(var.lock)
    return var.value

write(var, newval):
    if var.lock not held:
        acquire(var.lock)
    var.value = newval
```

Locking protocol with release

```
read(var):
    if var.lock not held:
        acquire(var.lock)
    return var.value

write(var, newval):
    if var.lock not held:
        acquire(var.lock)
    var.value = newval

commit():
    write commit record
    release all locks
```

11/4

Locking with reader-writer locks

```
read(var):
    if var.lock not held:
        acquire_reader(var.lock)
            # block if any writers
    return var.value

write(var, newval):
    if var.lock not held as writer:
        acquire_writer(var.lock)
            # block if any readers or writers
    var.value = newval
```

4/11

6.033 2012 Lecture 17: Isolation

```
Recall: goal is transactions: all-or-nothing and before-or-after atomicity.
  Both forms are about hiding implementation details.
  Last lectures: techniques for achieving all-or-nothing atomicity.
    Shadow copy, logging.
  Today: how to achieve before-or-after atomicity.
    Much easier to reason about what happens when multiple actions run.
    Allows concurrent actions to have all-or-nothing atomicity.
    (If actions see each other's intermediate states, then not all-or-nothing..)
    In database terminology, called "serializability" or "serial equivalence".

One simple way to achieve before-or-after atomicity: serial execution.
  Run all transactions in some serial order.
  Use one system-wide lock to ensure no concurrent operations occur.
  Why do we want anything more?  Performance
  But manually inserting fine-grained locks is error-prone,
    requires significant effort.
  So we're going to come up with a more automated plan to acquire locks.

Goal: higher throughput and parallelism
  Exploit multiple CPUs and I/O overlapping
  read-set = objects read by transactions
  write-set = objects written by transactions
  if read and write set are not intersecting, run them in parallel
  if read sets intersecting, but write sets don't, run them in parallel
  When intersecting, more tricky: what semantics do we want?

Consider two concurrent operations in a bank application:
  [ slide: concurrent actions ]
  xfer(A, B, amt):
    begin
    A = A - amt
    B = B + amt
    commit

  int(rate):
    begin
    for each account x:
      x = x*(1+rate)
    commit

What happens if we have A=100, B=50, and concurrent xfer(A,B,10) & int(0.1)?
  Serial executions:
    .. -> [xfer] -> A=90,  B=60 -> [int]  -> A=99,  B=66
    .. -> [int]  -> A=110, B=55 -> [xfer] -> A=100, B=65
  Unless there's some apriori ordering of xfer and int, either outcome is OK
    if xfer() & int() are executed concurrently.

What could happen if we run these operations concurrently?
  xfer:       int:
   RA [100]
   WA [90]
               RA [90]
               WA [99]
   RB [50]
   WB [60]
               RB [60]
               WB [66]
  This is called a "schedule" for these two transactions.
  The two transactions didn't run serially, but is this schedule serializable?
  Yes: outcome is the same as if they ran sequentially.

  xfer:       int:
   RA [100]
               RA [100]
   WA [90]
               WA [110]
   RB [50]
   WB [60]
               RB [60]
               WB [66]
  Not serializable: could not have possibly gotten (A=110,B=66) serially.

How to tell if a schedule is equivalent to some serial schedule?
  (In other words, has "serial equivalence"?)
  Problem: both transactions read & wrote A; both read before the other wrote.
    In a serial execution, one reads & writes after another reads & writes.
  So, clearly not serializable; can we formalize this intuition?

[XXX skip this part next year; not very important for 6.033. we are happy
to look just at simple examples as above; for which we can compute what
the possible serial outcomes are, and then check if a schedule computes
one of the possible serial outcomes.]
Automatic determining if schedules are serial equivalent or not
  Formally: two read/write operations T1.o1 and T2.o2 are conflicting if:
    - both access the same object
    - at least one is a write [i.e., order of two reads doesn't matter]
  "Conflict" here just means a pair of operations whose order matters.
```

```
 A schedule is "conflict serializable" if:
   - for all pairs of transactions T1, T2,
   - for all conflicting operation pairs T1.o1 and T2.o2,
   - they are ordered in the same way (either T1->T2 or T2->T1)
  If this condition isn't met, could have a problem:
   for some ops, T1 appears to be first, and for others, T2 appears first.
  (Technically, could be serializable even if not conflict-serializable,
   but typically not used in practice.)

Example:
  [ draw conflicts as bi-directional arrows in above non-serializable schedule ]
   RA[100] of xfer must precede WA[110] of int, otherwise xfer would have read 110
   RA[100] of int must precede WA[90] of xfer, othewise infer would have read 90
   No way to order two dependencies in a serial order

Demo: waiting transactions
   psql -h ud0.csail.mit.edu -U kaashoek in two different terminals
   one terminal (blue):
      begin; update accounts set balance=balance+5 where username='mike';
   other terminal (red):
      begin; update accounts set balance=balance-10 where username='mike';
   transaction in red terminal hangs:
      why?  because it conflicts with T1
      would result in non-serializable behavior if it didnt; must block
   abort blue transaction
      2nd one commits
      cool!

Hands-on: several isolation levels in Postgres
   Snapshot isolation (which is slightly different from serializability as defined above)
   Read committed isolation  (which is weaker than snapshot and serializability)
   I'll pretent that Postgres is using locking, even though it isn't for reads
   (it use an optimistic multiversion scheme; see textbook 9.4.3).

Achieving serializability: locking protocol.
  We don't want to rely on programmers to get locking correct.
  Need to devise a plan that guarantees serial-equivalent execution.
  Plan: Associate a lock with every variable; grab lock before accessing var.
  [ slide: locking protocol ]
  Q: When should we release?
    Should not release right after read/write operation.
    Can we release after txn is done with some variable (e.g., A or B)?
    xfer:       int:
     RA [100]              \ T1 holds A.lock
     WA [90]               /
                RA [90]       \ T2 holds A.lock
                WA [99]       /
                RB [50]       \ T2 holds B.lock
                WB [55]       /
     RB [55]               \ T1 holds B.lock
     WB [65]               /
    No good: (99,65) is not serial-equivalent.
  One solution that works is to release upon commit (or abort).
  [ slide: locking protocol with release ]
  Once T1 grabs A.lock, no other transaction that touches A can run before T1.

Deadlock
  What if the int() function accessed B first, then A?
  T1 (xfer) grabs A.lock, T2 (int) grabs B.lock.
  Now both want each other's locks, and neither can make progress.
  What are the solutions for deadlock?
    - Lock ordering: not a great approach if set of locks not known apriori.
    - Abort one of the deadlocked txns: make use of all-or-nothing atomicity.
  How to detect deadlocks?
    - Form a wait dependency graph: what each transaction has & wants.
        T1: holds A, wants B (-> T2)
        T2: holds B, wants A (-> T1)
      If a cycle is found, then we have deadlock.
      Requires a fair amount of book-keeping, analysis algorithm.
    - Assume that any acquire that takes more than threshold is deadlock.
      Simple, but doesn't deal well with long-running transactions.

Optimization 1: reader-writer locks.
  Recall that conflict required at least one operation to be a writer.
  Multiple readers should be able to operate on same data concurrently.
  We can use the idea of reader-writer locks to implement this.
  [ slide: locking with rw locks ]
  Instead of one acquire function, we have two: acquire_reader() and _writer().
  A reader can acquire lock at the same time as other readers (but no writers).
  A writer can acquire lock only if no other readers or writers.
  One possible problem: fairness.
    What if readers keep acquiring the lock, and a writer is waiting?
    A naive impl might never allow the writer to proceed, if always 1+ reader.
    Typical solution: if a writer is waiting, new readers wait too.
  This is an instance of the more general lock compatibility idea.
    Determine what pairs of operations are safe to execute concurrently.
    Allow concurrent locks by those ops (& disallow any that are unsafe).
    In general, need to understand operation semantics to determine compat.
    Reader/writer locks don't require understand semantics.
```

```
Optimization 2: releasing read-locks early.
  Observation: once a txn has acquired all of its locks, it will not block.
    The point at which all locks have been acquired is called the "lock point".
  Once T1 reaches lock point, any conflicting transaction T2 will run later.
    Intuition: if any other txn wants one of T1's locks, it will have to run
      later, because T1 has already acquired all of its locks.
  The serial order we get is the order in which txns reach their lock points.
  If txn reached lock point and will no longer access x, can release x.lock.
    Will not affect serialization order.
    Will not affect x.
  Our earlier example:
    xfer:       int:
     RA [100]                \ T1 holds A.lock
     WA [90]                 /
      ...                    > T1 acquires B.lock, lock point, releases A.lock
                RA [90]         \ T2 holds A.lock
                WA [99]         /
                lock B: wait
     RB [50]                 \ T1 holds B.lock
     WB [60]                 /
                RB [60]         \ T2 holds B.lock, lock point, had to wait for T1
                WB [66]         /
  This is called "two-phase locking" (2PL).
    Phase 1: acquire read and write locks, until transaction reaches lock point.
    Phase 2: release read-locks, after lock point & done with object.
  To make this optimization work, need to know when we're done with object.
    Requires input from programmer, or program analysis.

Why hold write locks till commit?
  Problem arises if some transaction doesn't commit (i.e., abort or crash).
  If T1 releases a read lock and then aborts, not a problem.
    Even if T2 acquired read or write lock, and committed, still serializable.
  If T1 releases a write lock and then aborts, could reveal uncommitted values.
    E.g., in our example above, if T1 (xfer) aborts, we would get (A=99, B=55).
    Big problem: not all-or-nothing / before-or-after.
    T2 saw intermediate effects of concurrent execution; T1's abort incomplete.
  Two possible solutions:
    - Hold write locks until commit (strict 2PL): do not reveal new values
      until we know the transaction commits.  Typical.
    - Abort any transaction that saw uncommitted results (e.g., T2).
      This strategy is known as cascading aborts.
      Requires bookkeeping to track readers,
      Commit sometimes blocks, if txn read values that weren't committed yet.
      Complicates crash recovery, ...

[XXX made it to here]
Isolation and recovery:
  The log contains a serializable trace
  Recovery is fine

More isolation
    Lecture scratches the surface of different isolation schemes
    You saw other one in hands-on: read-committed.
    Can it result in non-serializable schedules?  Yes.
      a committed write from trans 1 will be visible to read in trans 2, even if
      trans 2 started before trans 1.  if trans 2 performs 2 reads, they may
      return two different results: one without the effects of trans 1 and one
      with the effects fo trans 1. a serial schedule wouldn't allow this.
    Implementation: reads don't take a read-lock
    Why does Postgres allow it?
       It is ok for some transactions. For example:
         BEGIN;
          UPDATE accounts SET balance = balance + 100.00 WHERE username = 'mike';
          UPDATE accounts SET balance = balance - 100.00 WHERE username = 'jones';
         COMMIT;
       More concurrency, more performance
       But, difficult to reason about!
   Default isolation in postgres: snapshot isolation
       More serialization than read-commit, but still anomalies
       T1
          R(A)
          R(B)
          if R(A) and R(B):
             W(A)<-0
        END
       T1
          R(A)
          R(B)
          if R(A) and R(B):
             W(B)<-0
        END
        write sets are independent, and are allowed to run in parallel
        but now both A and B could be written 0, which is not serializable
   Dan Port's version is in postgres now
       Serializable snapshot isolation

Summary.
  Technique for reasoning about concurrent actions: before-or-after atomicity,
    or serializability.
  Can check for serializability by considering ordering of conflicting ops.
```

```
2PL can ensure serializability by using locks to order conflicting ops.
Can recover from deadlock by aborting one of the deadlocked transactions.
```

6.033: Computer Systems Engineering

Spring 2012

Home / News

Schedule

Submissions

General Information

Staff List

Recitations

TA Office Hours

Discussion / feedback

FAQ

Class Notes Errata

Excellent Writing Examples

2011 Home

Preparation for Recitation 18

Read the paper *The Recovery Manager of the System R Database Manager*.

This is a long and difficult paper. The introduction summarizes the whole of System R, which provides a number of features beyond the recovery features focused on in this paper. You should read this section, but do not worry if there are details you do not understand. Sections 1 and 2 are the meat of the paper, and you should try to understand them as much as possible. Because this paper is quite dense, you may wish to skim these sections first, trying to understand the basic approach to recovery. Section 3 is a retrospective on the recovery features of System R; you do not need to focus on the details of this section, but you should read it quickly as it provides an interesting discussion of things that were problematic in the design presented in the previous two sections.

If there are terms or concepts you did not understand while reading the paper, be sure to ask about them in recitation!

As you are reading the paper, think about the different mechanisms the system provides. For example, do shadow files and logging solve the same problem in System R, or different problems? Why are both mechanisms there?

Read

4/11

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

Top // 6.033 home //

IBM System R

From Wikipedia, the free encyclopedia

IBM **System R** is a database system built as a research project at IBM San Jose Research (now IBM Almaden Research Center) in the 1970s. System R was a seminal project: it was a precursor of SQL, which has since become the standard relational data query language. It was also the first system to demonstrate that a relational database management system could provide good transaction processing performance. Design decisions in System R, as well as some fundamental algorithm choices (such as the dynamic programming algorithm used in query optimization[1]), influenced many later relational systems.

System R's first customer was Pratt & Whitney in 1977. [1] (http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95-System.html)

See also

- SQL
- System/38

1st real SQL
+ Concurrent Txn

References

1. ^ Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979), "Access Path Selection in a Relational Database Management System", *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 23–34, doi:10.1145/582095.582099 (http://dx.doi.org/10.1145%2F582095.582099)

External links

- "A History and Evaluation of System R (http://www.cs.berkeley.edu/~brewer/cs262/SystemR.pdf) "
- System R website (http://www.mcjones.org/System_R/)
- The 1995 SQL Reunion: People, Projects, and Politics (http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1997-018.pdf)

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

Retrieved from "http://en.wikipedia.org/w/index.php?title=IBM_System_R&oldid=441307576"
Categories: Proprietary database management systems | IBM software | Database software stubs

6.033 Recitation

4/12

He has office hrs in Ridigi

Complaining about Stata

- leaks, but doesn't collapse
- we're not there w/ s/w yet

System R

(he doesn't fully understand)

LFS is a good design project

-requirements

-step through

System R

is not really it

changed mind/req half way through project

Database

↳ so why talk about files

↳ underlying files

Modify content via transactions

Added logging half way through

- Couldn't get it to work w/ shadow files

Writing file + crash

We've solved w/ Journaling File System

(I'm complaining about 6.033 class)

L Why learn about this before fully understanding ~~the~~ System A

Prof: You have to read papers to avoid mistakes of he past

Prof: Database world was seperate from OS in 70s

<u>Copy on write</u> - whenever you write, copy whole file

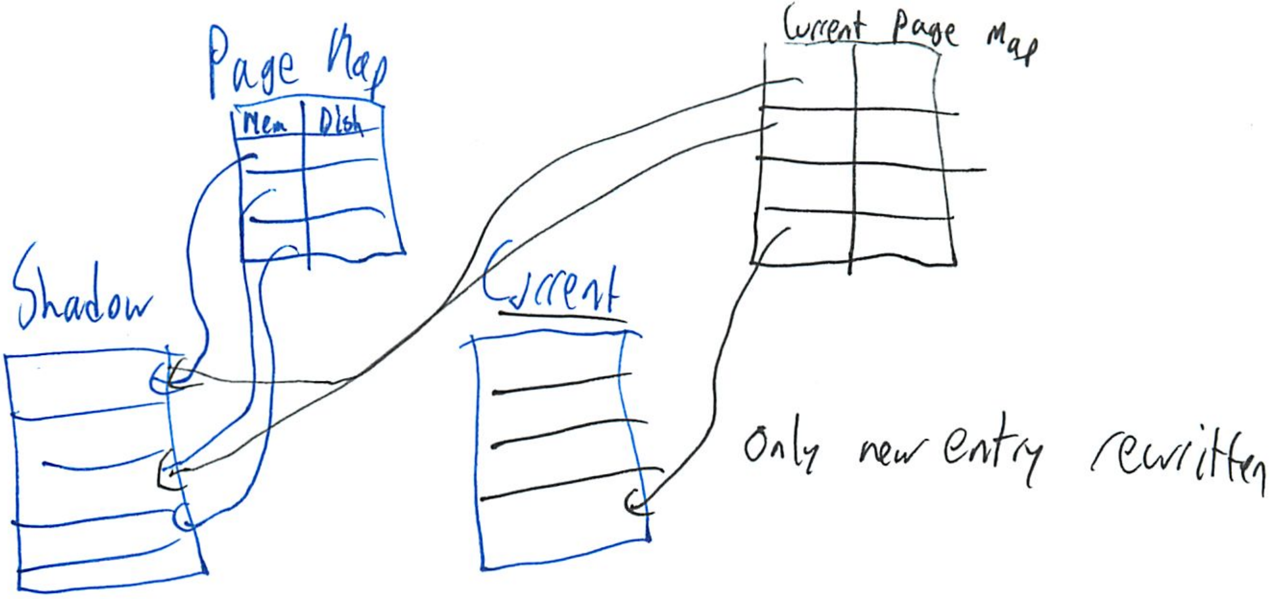
Can mark in page table

get a trap

Make own copy

<u>Don't</u> copy before hand

(3)



Performance: Read + Write from File Cache

Write Back vs Write Through Cache

So Don't have LRU - control write back

Or System R: Use shadow file

Only shadow what is on disc
- Consistent
- Correct
- duplexed

System crash -> shadow is always correct
don't trust current

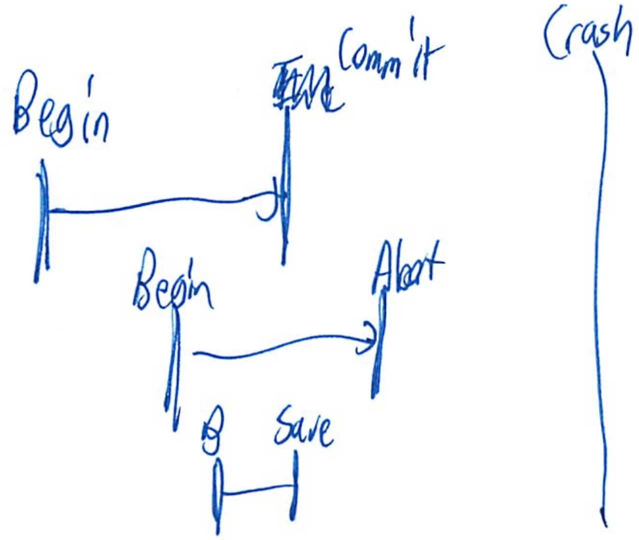
4

File Save → Push out all blocks of current

Shadow ← current

But ~~file~~ can't do unrolling and can't handel txn
couldn't get all properties
-had to do log

?~~crash before txn~~
Can ~~only~~ do file save on 1
in txn - update multiple files
now get to get all file
saves in a txn to happen
atomically



No nested txn anymore - not useful

Save vs Commit

└ atomic txn

it happens or it doesn't

hard to show old data before commit

May have to write to disk before commit

└ since txn must be bigger than cache

Save is like a bookmark

Go back to this point

For that txn

Not exactly Hand's On Checkpoint

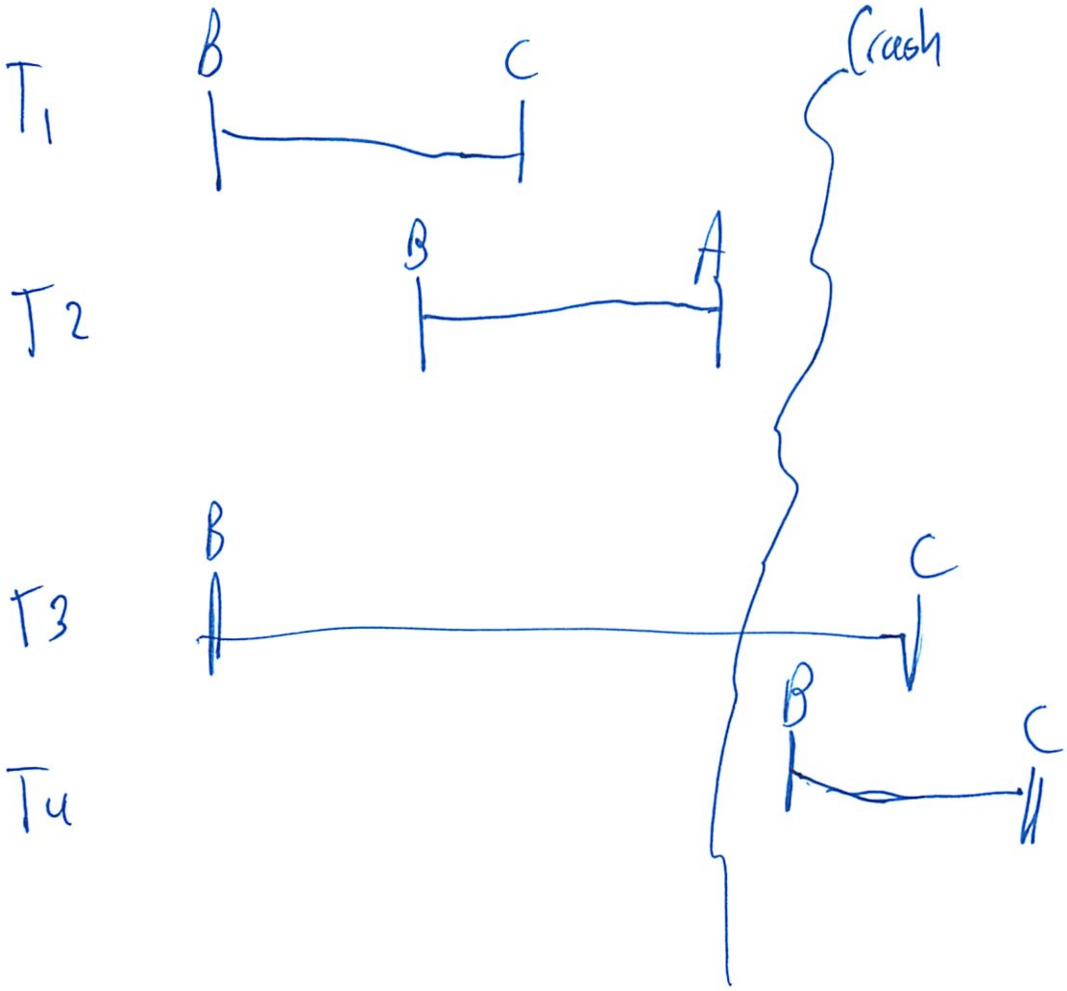
Partial undo

Kinda nested txn

The multi-hop airplane example

6

Paper does not talk about locking

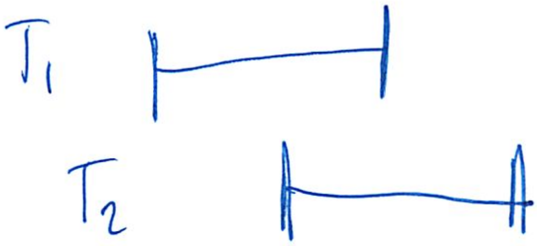


T_1 commits fine

T_2 doesn't exist

T_3 not there - since never committed

T_4 after recovered



Serailzable : T_2 never sees T_1

but very hard to commit

thats versioning works

normally T_n calls all the stuff I might use

System R does not do this Prof believes

They didn't have the distinction clear

We know now - see Postgress reading

Redo after crash

Q: How does RAID change this? SSD? SRAM? Web?

- diff disk charastics

Name: Michael Plasmeir

Section: 10

A

	Does Paper Address Cycles in Graph?	Filename? Inode? By Value/ Ref?	Where & How Stored. iNode xattr, block metadata,	Can it handle multiple parents. What happens on parent or child rename
Data Structures 30%	?	File name iNode	Files + Tables Reverse index	Yes (I think) ~~a little confused~~ Paper is not clear on This
	How refer to past versions.	Discuss False dependencies	iNode reused	
Versioning Or Self Versioning (bonus/minus)	Log table	?	✓	
	how r they named? by whom?	File part operations	Part Prov Conservation	Rename parts, rename file
File Parts 10%	App	Ok		✗
	Modifications to Open, Read / Write, Close	Write_Prov, Read_Prov	Search_Prov	What does search return and how can it be used
System Calls 20%	✓	✓	✓	
	Unlink (hard/soft)	Copy Costs	How to handle and cost of continuous copying	Real Numbers
Performance Analysis 20%		✓	✓	✓
	Zip/unzip iNode<->File name	PowerPoint	Make/Compile Case	Unmodified Programs
Workloads / Use cases 20%	✓	✓		

The Plaz Provenance File System (PPFS)

Michael Plasmeier
theplaz@mit.edu
Rudolph 10AM
March 22, 2012

Introduction

A provenance file system is a file system which stores the history and source of files edited on a local computer system. For example, when one is editing a slide deck, one might want to know from which slide deck a slide was copied from. This paper builds upon the basic file system in the early versions of Unix and introduces a provenance file system known as the *Plaz Provenance File System (PPFS).* In particular, the PPFS introduces another layer, called the log layer, which maintains pointers to past versions and ancestors of the file.

The PPFS stores a full, verbose set of provenance information. The PPFS also stores the complete data of old versions of files, giving it many of the features of a versioning file system. The PPFS also supports seeing which files are based off a specific file. This is called reverse lookup. These characteristics make the PPFS well suited for businesses that face regulatory or legal requirements to log all changes to files. The PPFS is also well suited to businesses which frequently create derivative works from previous works. For example, a consulting company may what to know which project a slide in a slide deck was copied from.

or copy-on-write Blocks

The system aims for a simple design and it attempts to use a minimal amount of disk space for maintaining provenance information and a de minimis amount of Random Access Memory (RAM) space for the tracking of provenance information. However, it currently uses up a lot of disk space to store old versions of files. Additional disk space could be saved by de-duplication algorithms. Data is laid out so that lookups from both directions (the ancestors of a file and the children of a file) can be performed relatively quickly. As part of the operating system, the PPFS extends the usual (`read()`, `write()`) operations. For more complicated or novel functionality, new API calls are introduced.

At the moment, the PPFS operates only on one computer. It is not optimized to work over a network, nor does not track provenance information from files copied from other computers, such as web servers.

The Log Layer

The PPFS introduces a new layer into the Unix file system called the log layer. This layer is inserted between the file name layer and the inode layer, as shown in Figure 1. The file name layer is modified by redefining the inode number in the directory table to the log entry number.

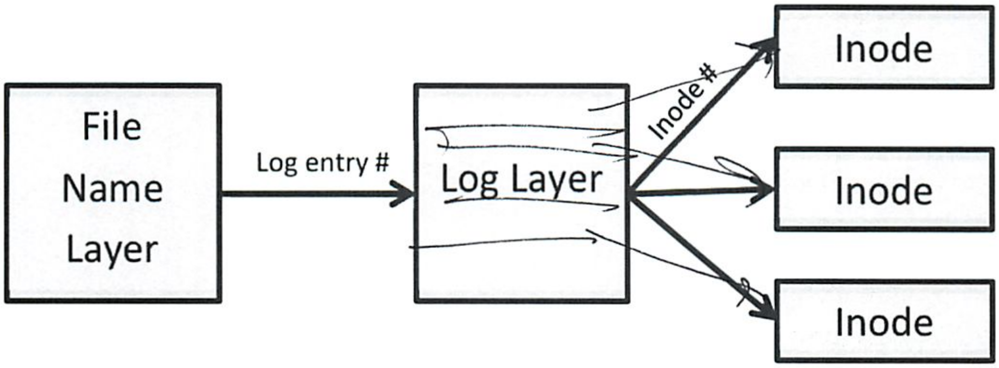


Figure 1 The Log Layer is inserted between the file name layer and the inode layer

is log

The log layer contains a table of information about the history of each file. Each file has its own table, which is stored on disk in the same way as the inode layer. The log layer table is stored at the beginning of the disk, in a fixed position on the disk, after the inode table. The table consists of a list of log entries, each pointing to the inode number of a *version* of the file, as shown in Figure 2. A version is created automatically each time the file is saved. The last entry in the log layer entry contains a reference to the log layer entry that the file was created from (the *ancestor)*. A bit in each entry designates a row as a version/inode pointer or an ancestor /log layer pointer. If a file was created from scratch (ie using `touch`) then the last entry in the log layer table will be `0`. Although the inode stores the time the file was modified, that information would not be changed if the file was copied, so PPFS also stores that information in the log table. Additional log information, such as the current user's username and the application that made the change could is also stored here.

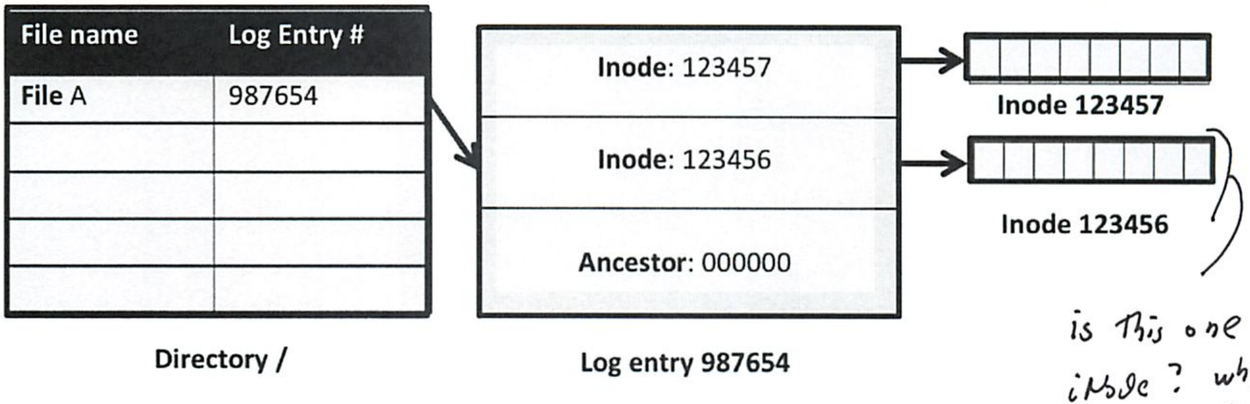
maybe, seems overkill

The number of incoming links that was stored in the inode in the original Unix file system is redefined to count the number of log layer entries pointing to the inode. The log layer table includes a count of the number of incoming links that was traditionally found in the inode. These counts are manifested in the reverse tables, described below.

Directories are ignored by the PPFS and function as usual. This may differ from certain versioning file systems.

Only one ancestor? why is this the last entry?

versions



is this one inode? why the array of cells?

Figure 2 File A is created with content and is then edited.

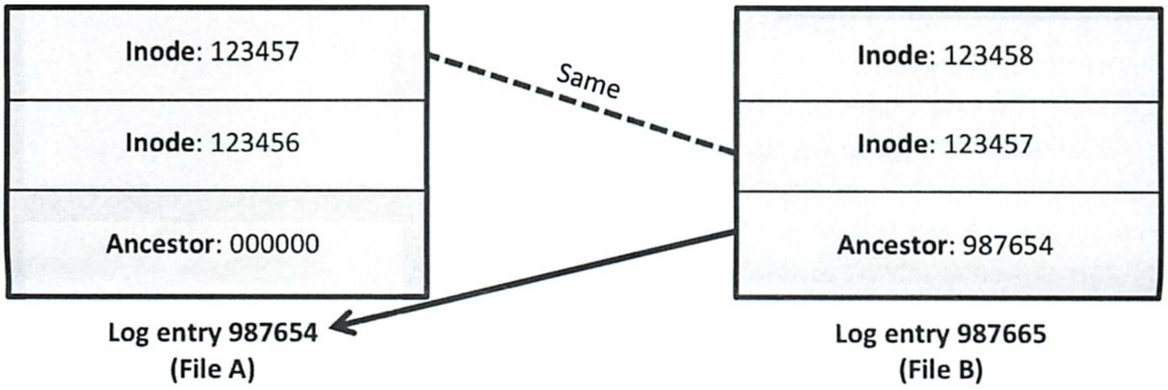


Figure 3 File A, from above, is then copied to be File B. File B is then edited. Notice that File B rev 0 shares an inode or version with file A rev 1.

When provenance information is queried (via read_prov()) for File B, the log entry for File B will be retrieved. Provenance information will then be recursively queried (to A in this example) until an ancestor of 0 is reached.

Does it make sense for read-prov to do rec search each time even if app just needs immediate parent?

Reverse Lookup

One requirement of a provenance file system is to know all of the files which originated from a particular file. This information can be accessed using the search_prov() system call. In order to support the reverse search case, the log entry and inode tables are modified. A reverse inode table is added for each inode, which contains the list of log entry tables which point to that inode. A reverse log entry table is added to each leg entry to retrieve the files names which each log entry represents.

log

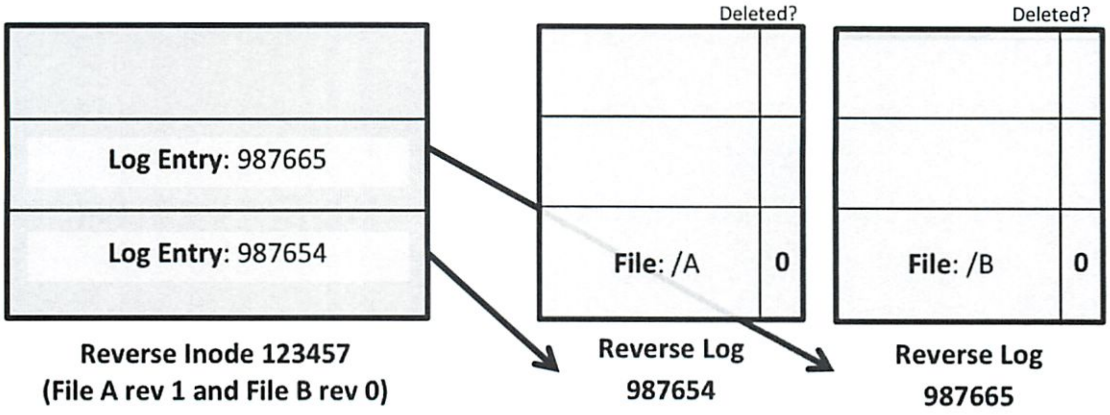


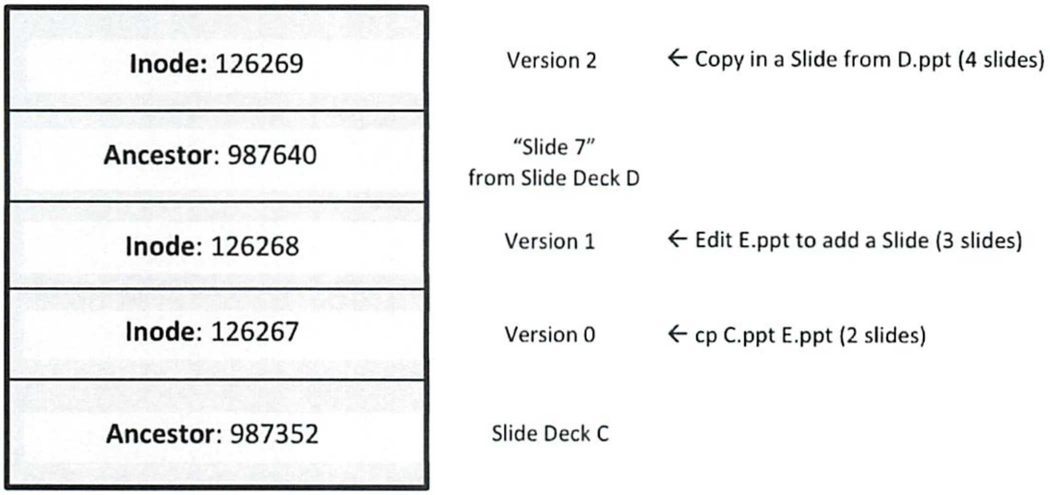
Figure 4 The Reverse Lookup Inode table for Inode 123457 shows that the File A and File B shared the same data at some point in time (i.e. one must be the ancestor of another)

In a `search_prov()` query, the system first looks up the log layer table for a particular file. For every inode mentioned in the log layer table, the system retrieves the reverse inode table. The system then retrieves the reverse log tables for each file. The system then outputs the list of files referenced. If needed, the system could also lookup the log tables themselves to find the revision number and timestamp for each file.

Parts of a File

Provenance information can be stored about the *parts* of a file (for example, the slides in a *slide deck).* This information is stored by having multiple ancestors in the log layer, as shown in Figure 5. In this example, the difference between Slide Deck E version 2 and version 1 came from Slide Deck D. The pointer to Slide Deck D and a name for this piece would be set by the `write_prov()` call. The data that is different would be inferred by looking at the difference between the inode before and after the ancestor entry. The name data is stored in a separate table, as seen in Figure 6.

Applications wishing to take advantage of the *provenance by parts* functionality would need to implement this API call.



Log entry 987636
(Slide Deck E)

Figure 5 Slide Deck E was copied from Slide Deck C with 2 slides; a slide was added from scratch; and then a fourth slide was copied in from Slide Deck D (which was previously called slide 7 in Slide Deck D)

Log entry #	Name
4	"Slide 7"

Piece names 987636
(Slide Deck E)

Figure 6 Piece names for the various pieces for Slide Deck E. In this example, the 4th entry (from bottom) of the log entry for Slide Deck E were previously called "Slide 7" by the application.

Compilations of Files

Information about the source in compiled binary files can be stored in in a similar way. Multiple ancestor entries are stored at the bottom of the table between the first ancestor and the inode of the newly compiled file, as shown in Figure 7. The first entry will be 0, since the file was created new. Normal log entries will accumulate on top of this information, as before.

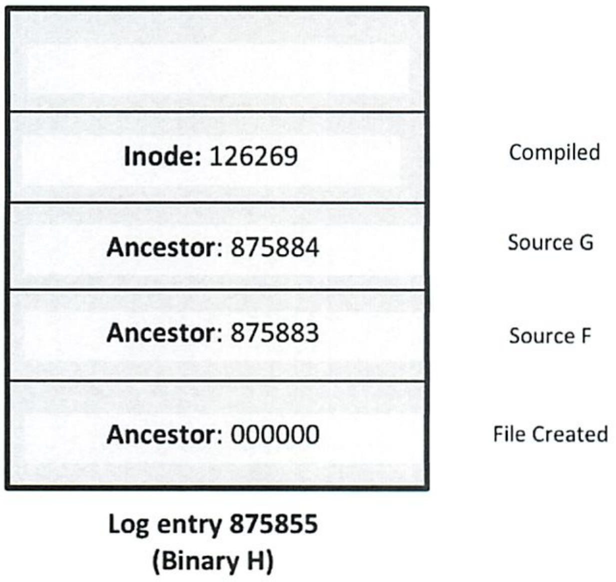


Figure 7 Binary H is compiled from Source F and Source G

File Archives

File archives present a particular challenge. File archives read information off the disk and then store it in their own proprietary format. In order to be truly portable, this requires all of the provenance information, along with all of the past versions and ancestors of a file to be stored in the file archive. This information would be retrieved using a special call, such as `read_full_provenance()`, which would store the provenance information and past versions in a flat format. This format is a XML format which mirrors the tables in the file system, as shown in Figure 8. The file would then be compressed using normal ZIP or TAR algorithms.

```
<xml schema=”ppfs-portable”>
      <log-entries>
            <log-entry id=”875855”>
                  <ancestor>000000</ancestor>
                  <ancestor>875883</ancestor>
                  <ancestor>875884</ancestor>
                  <inode>126269</inode>
                  <reverse>
                        <file>/H</file>
                  </reverse>
                  //Additional metadata (i.e. name, date) removed
            </log-entry>
            <log-entry id=”875883”>
                  <ancestor>000000</ancestor>
                  <inode>126267</inode>
                  <reverse>
```

```
                        <file>/F</file>
                    </reverse>
                </log-entry>
                <log-entry id=”875884”>
                    <ancestor>000000</ancestor>
                    <inode>126268</inode>
                    <reverse>
                        <file>/G</file>
                    </reverse>
                </log-entry>
            </log-entries>
            <inodes>
                <inode id=”126269”>
                    <data>(Binary data)</data>
                    <reverse>
                        <log-entry>875855</log-entry>
                    </reverse>
                <inode>
                <inode id=”126268”>
                    <data>(Binary data)</data>
                    <reverse>
                        <log-entry>875883</log-entry>
                    </reverse>
                <inode>
                <inode id=”126267”>
                    <data>(Binary data)</data>
                    <reverse>
                        <log-entry>875884</log-entry>
                    </reverse>
                <inode>
            </inodes>
        </xml>
```

Figure 8 The flat file XML for the scenario in Figure 6

When the file archive is extracted, the provenance information is recreated using a special `write_full_provenance()` call. The inode and table entry numbers will change, but the same structure will be created. Those that are interested in preserving the authenticity of the provenance information should disable this feature, because it allows anyone to write provenance information (including old time stamps) to disk.

— some ancestors are in zip file + some are not. when unzipping what happens to references to files not in zip file?

Deletion and Thinning

When `unlink(filename)` is called, the filename to log entry link is removed, and the deleted bit in the reverse log table is flipped (decrementing the traditional link count in the log layer entry). When the count of incoming links in the log entry table reaches `0`, the file is no longer accessible. However, the log table and versions are kept in order to preserve provenance information.

In order to save space some intermediate versions can be removed according to a thinning schedule. This schedule is user-settable, but the default values are shown in Table 1. The thinning process is accomplished by a "garbage collection"-style program. A revision will be kept if more than one log entry is present in the reverse inode table – i.e. when a file was copied and is now provenance information for a different file. When versions are thinned, the actual inode/data is removed from the disk, and all references to that version are removed from the log layer.

Days after revision created	Target number of revisions kept
< 7 days	1 / minute
> 7 days and < 30 days	1 / hour
> 30 days and < 1 year	1 / day
> 1 year	1 / week

Table 1 Intermediate revisions can be thinned after a certain amount of time after their creation. These are the default values

Performance

PPFS should be not appreciably slower when adding many files to the disk. Principally, the disk must make one additional write (the log layer table) in addition to its other writes. Generally, the non-sequential disk accesses slow a hard drive down. PPFS adds one additional non-sequential access. Thus the system should be no more than 33% slower (adding the log layer to the file system pointer, inode, and file data). Thus the system should be able to easily handle writing 10 files to disk per second. PPFS scales with the size of the disk and is linear with regard to the number of items added to disk per second. The garbage collection process is optional, and can be postponed until the system is relatively idle. ✓

In addition, the system can quickly search for the children of a file (files that are based on that file) by using the reverse inode table. Such lookups should not depend on the number of files on the disk. PPFS scales well with regard to the number of files on disk.

For a file with many ancestors, PPFS handles reads and writes to a file the same as a file without an ancestor. Retrieving the full list of provenance information scales with the number of ancestors. It is envisioned that this will not be a large bottleneck, since the number of ancestors is envisioned to be relatively low and pulling a full list of provenance information is an infrequent operation. Caching could be added to the system to improve this time, but the additional step to update or invalidate the cache would slow the copying of files.

One of the most significant performance impacts is the time to update a file. PPFS rewrites the entire file each time it is saved, in order to maintain a version history of the file. PPFS is not optimized for large files, such as media files, and is likely unsuitable for those use cases.

Is there a difference between these two write cases in overhead of PPFS? ~~Every wrn~~

```
Buf ← Read file A
For b in Buf:
    Append to file B
```

```
Buf ← Read file A
Write file B (buf)
```

PPFS uses a significant amount of disk space. PPFS is designed to provide verbosity and maintain provenance information at the expense of disk space. Thus PPFS is best suited to organizations that require comprehensive and persistent logging.

Conclusion

PPFS is a provenance file system designed to provide a comprehensive and reliable log of the history of each file. PPFS modifies the basic Unix file system to add a log layer that preserves the provenance and version history of each file on the disk. PPFS should add minimal overheard, beyond the keeping of multiple versions. PPFS should be able to easily handle lookups from both directions (the ancestors of a file and the children of a file) in a short amount of time. PPFS should scale well to the size of the disk, the number of files on disk, the number of ancestors of a file. PPFS can do additional work to reduce the disk space that old versions take up.

Implementation Issues

Modern file systems have advanced beyond the basic Unix file system that PPFS is based on. Care should be taken to maintain the current features of file systems while implementing PPFS.

Where additional API calls have been added, developers must be recruited to update their applications to support the new APIs.

Future Work

PPFS suffers from a number of limitations, which could be addressed by modifications to the system.

PPFS currently rewrites each file when it is edited, using a lot of disk space. A de-duplication algorithm which, for example, only stored the changes to a file, could save a significant amount of disk space.

PPFS currently only works on a local computer. PPFS could be expanded to work across a network. Provenance information is particularly helpful when there are multiple people working on a group of documents.

PPFS is currently designed to operate on a single disk. Because of the large amount of disk space used by PPFS, it could be expanded to work across multiple disks. For example, the RAID system allows multiple disks to be seen as one disk by a computer. This would allow users to add storage to the system as needed.

Acknowledgements

Reviewers

- Dave Custer
- Travis Grusecki

References

[1] J. Saltzer, M. F. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA: Morgan Kaufmann, 2009.

Word Count

2,358 words, including captions

6.033 Tutorial

4/13

Today

Overview
Requirements
Conflict Scenario
Central Server Design

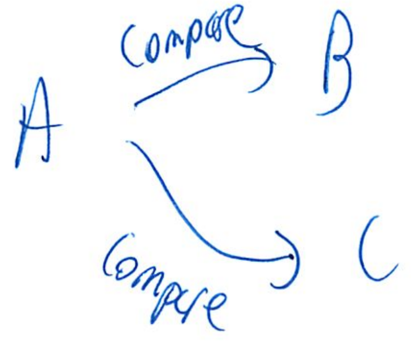
(He has assigned teams)
(He doesn't think this is all that ~~all~~ similar
String merging tricky — lots of edge cases)

Basically building Google Docs but P2P
- P2P
- Offline
- Various Scenarios
- Static group membership
  - no users come in/leave
  - could be disconnected

- known, publically accessible IP addresses
- support direct connectivity
    - prob won't change anything
- notion of commitpoints
    - mark as "draft 1"
    - everyone must agree

<u>Commits</u>

- Failures at any time
- two phase committ



"Do you all agree this is draft 1"

Once B says yes

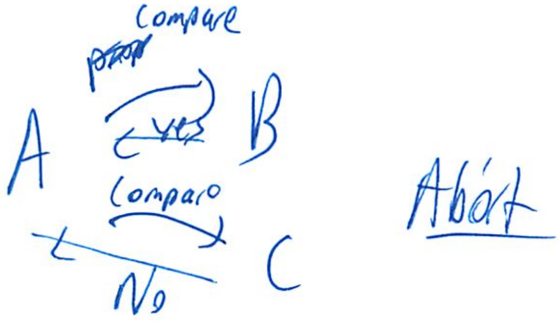
- committing to commit if he is asked to

③

← text editor

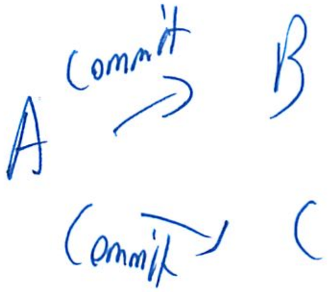
But if C says no

A aborts



otherwise If C says yes

A will commit



General idea: anyone can fail at any point

B will commit even if he crashes.

Since its in the log

Shows protocal basically works

Wed lecture: will prove it

4

Connection scenarios

- online (everyone)
- offline

Can elect someone the master

But could ~~[illegible]~~ always work offline
Only something happens when hit save
easier

~~direct Connection~~

- part online / part offline
  A—B C D
- partitioned networks

A C
| |
B D

Depends how you design merging

5

- **NOT** Bridging

$$\begin{tikzcd} A \arrow[r, leftrightarrow] \arrow[rr, leftrightarrow, bend right, "\times" marking] & B \arrow[r, leftrightarrow] & C \end{tikzcd}$$

Can bild on mecurial, etc
  L if you can get it to do everything

Merging

Don't have to support multi-way merging
  L can do pairwise merging

Basically the standard is eventual consistency

How do you order changes?

Next week: version vectors
  L keep your version
  and what you think everyone else's state is

$V_a = [\underset{A}{5} \quad \underset{B}{10} \quad \underset{C}{3}]$

(6)

Say A makes a change

$V_a = [6 \; 10 \; 3]$

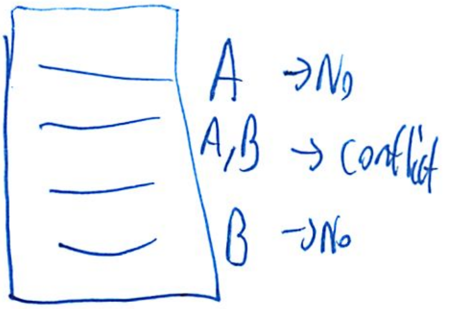
When merging → take max of version vectors

Only that client can update their section in tuple

Max over each col

Conflicts

(1)



2 users disconnected

— work seperatly

(2) A, B | C

"abc" | "abd"

If A ↔ C now syncing

②

A,C

↳ decide "abcd"

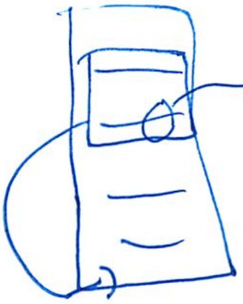
Could be a user interface on 1 screen

but now no one should ever see

a Conflict here

L like when Bob Comes in → no conflict

③



Seperatly/offline

B fixed spelling

A moves text

? note move not copy

* Shold pick both (w/o error UI)

* How data structure set is critical

L the unit is very important

letter

word

sentance

line

para

(8)

Version vector for every word ??

Note: Performance +Efficency is not important!

Can take 2GB for a 2 sentance doc

Thats fine

Correctness is key

(9) Two users fix the same change

↳ No conflict

ie both run the same spellcheck ind.

(or manually fixed

<u>Draft Central Server Design</u>

from 2 years ago

Much more of a Google Docs design

easier - can order based on central server

(9)

[Can assume all clocks correct — but must say]

Google Docs never has conflicts

Cursors are ~~[illegible]~~ like characters

Always ordered

So know what comes 1st

Google does not do locking

↳ you can do

Or lock after you say Yes to compare

until commit

↳ ~~users could keep~~ Could allow users to keep typing

not required

Could do everyone is the server

↳ primary/backup scenario like in class

6.033 Reading

4/15

(Makes much more sense now - after lecture...)

blind writes - just write the value w/o concern to whats there

rollforward/redo logging - ~~da~~ ~~att~~ update cell after
recording outcome
perform all logged installs w/o END
(so both Outcome and End message)
Can be fast

both backwards + forwards

diff than rollback/~~ad~~ undo - where ~~[illegible]~~ write cell
then log Outcome
here must undo incomplete entries

Cache is write through

②

both all or nothing

before + after

here serial concurrency → strictly one after the other

L 9.4 adds more concurrency

L this is very conservative

though it only really matters when executing a txn

txn 3 uses values after all txn 2 has run or committed

(i not when it started — which is what it was before i's)

Can loosen up timing → as long as result ~~it change~~ are same

Mark point — behavior of before + after

4/17

later tranx need to read pending before comm

So read — waits for pending values that < txn id

~~ass~~ \ Skips " > txnid

(i so same as in class)

Seperate read-my-value for = txnid

③

Also must ~~the~~ save the values when a txn starts to use as inputs

↳ So mark which txns to use by making pending versions

? this points to later txns is called the mark point

(why did Postgress not use)
emailed in

So new txn can't read until prev txn reached mark point, or committed/ aborted

Can delay setting mark point

↳ or other displine (descibed later)

(No deadlocks - since wait for earlier ...)

So mark point lets you go faster than simple serialization

Both Serialization + mark point are pessimistic

- presume inteference, conservativly block

Optimistic allow inference and then recover

require to re run txn

^ so we can rely on app to do this?

read capture - don't need to tell in advance

when read mark threads position

high water mark - higest #ed txn that

has ever read from this point

so lower serial #s know its too late

to read this

Must abort txn and try again later

6.033 48

4/18

Quiz Fri

Distributed Txns

Talked about Txn a lot

└ But this is the last lecture

Before Recovering from crashes

Can reconstruct state of the system

- WAL
- undo/redo
- 2 phase locking (review)

Today Distributed Transactions

Isolation (wrap-up)

Isolation

Serializibility

└ its like concurrent txns ~~have schedule~~ are exected in serial order

same outcome!

Saw last week can do w/ strict 2-phase locking
keep lock mark point → lock point

Other txns must wait

Weaker Forms of Isolation

Saw in Postgres

Allows more concurrency → thus higher performance

But less programmer friendly

examples

Snapshot isolation

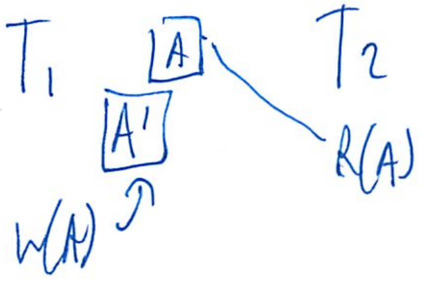
- implemented w/ multiversion memory
- keep diff versions of same amt

T1 □ T2
~~R(A)~~ R(A)
W(A)

- before ^{T_2} would have lock on A

③

there W(A) writes a new version A'



But <u>not</u> same outcome as serial
(example w/ doctors in slides)
doctors going off call if ≥1 on call
but if run concurrently?

<u>Serial outcomes</u>

$T_1 \to T_2$
or   Bob still on call
$T_2 \to T_1$
                Alice still on call

<u>Snapshot isolation</u>

each txn uses data from start of txn

that is not seralizability

(I was confused on that...)

4

Basically read set is frozen on begin
↳ no read locks

Here read sets overlap, but write sets don't

Most txns don't have this problem

(I thought this was gold standard - but I guess serializability is gold standard)

Default in Postgress: Read committed isolation

- even weaker
- no read locks
- no multi-version memory

T1	T2
	R(A)
W(A) Commit	
	R(A)

↳ gets the recently committed value

~~Not from~~

The 2 reads are different

Seralize + snapshot would have diff ans

Must carefully consider how your DB is set up

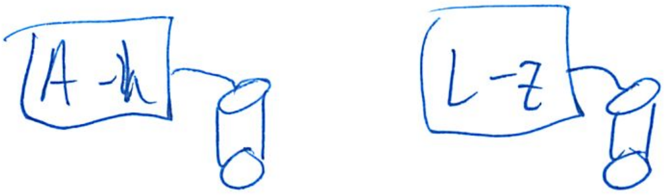
Have DB do it, since harder to do in app

6.033: Only right way is serializability

Distibuted Txns

Run txns accross machines

Higher throughput so split accounts



If txn within one, its fine

xfr(A,B)   xfr(Q;Z)

But between dbs?

xfr(A,Z)

Cool solution

```
begin
    begin
        A ← A-10
    end
    begin
       z ← z + 10
    end
end
```

↙ like a priest asking each person if they will marry

Nested txn

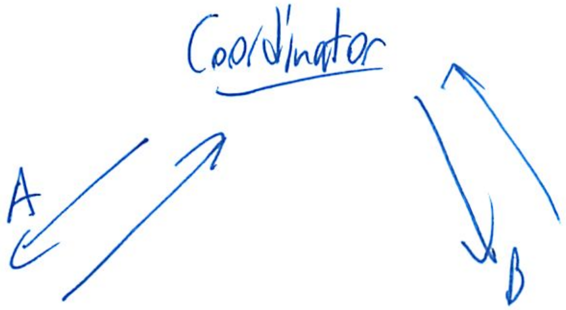
Outer only suceeds if both inner suceedes

inner only commits if outer commits

But where do you commit?

So introduce a 3rd party → coordinator

like the priest in our weding story



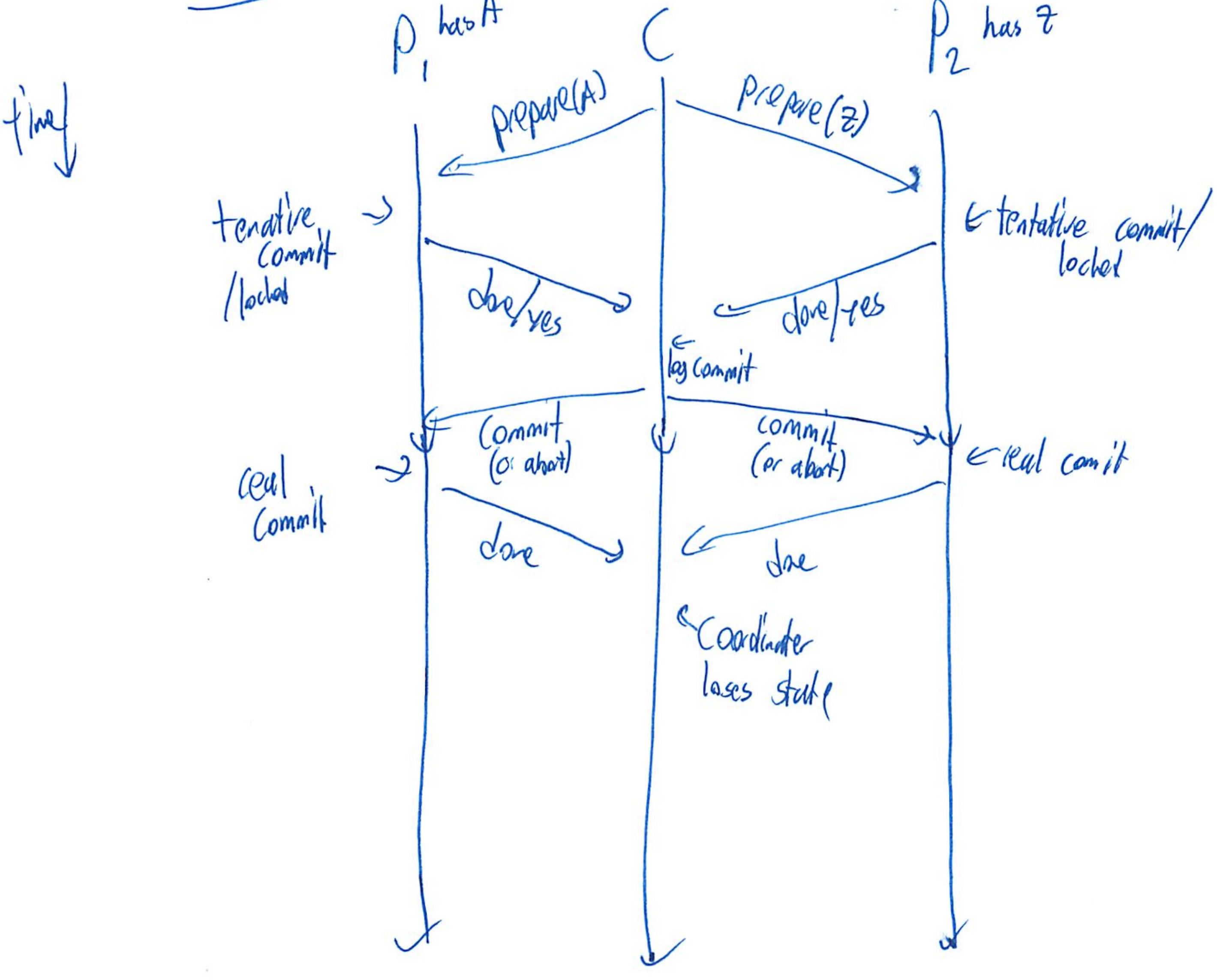
commit

2-phase protocol

- ask each to commit
- Then tells everyone to commit

but nothing to do w/ 2-phase locking!

Basic Version



What is the commit point?
When coordinator say commit

Since we can always recover from there

Postgres kinda supports

must also be an isolation

PREPARE TRANSACTIONS

(i didn't get how this was working)

Select * from pg_prepared_xacts

Fails if can never prepare serializable

What if scenarios

- packet drops?
  - can re-transmit?

    Must check if prepare message recieved but not processed
- participants can fail ~~(missed)~~ → - before prepare important?
  - before real commit - never really explained
- Coordinator can fail

  Other guys can't individually decide

  Must wait for coordinator to come back up ← always a case where this happens

  Then look at their log
- can get messy fast

CS033 Review

4/18

(Half pay attention

Map Reduce

$$\begin{array}{lcccccc} & ((k_1, v_1), & (k_2, v_2) & \dots & (k_n, v_n)) & & m(k,v) \\ \text{map} \hookrightarrow & \downarrow m & \downarrow m & & \downarrow m & & \\ & ((q_1, w_1), & (q_2, w_2) & \dots & (q_n, w_n)) & & v(Q,w)) \end{array}$$

reduce $\hookrightarrow (Q_1, v_1)(Q_1, w_2) \dots)$

$\downarrow X$

So easily parallizable

Sum the 1s

Map can be done multiple times

Throw away dupes

Use Google FS to dedupe

Write temp copy so don't overwrite

Duplicate towards the end...

2

End-to end

apps know best

don't have lower levels do that stuff

apps might not want a feature

UDP vs TCP

Ethernet does an best effort

Carrier detection

RAID

Used to be big price difference

0 = striping

- spread data across discs

1 = n-drive mirroring

no faster writing

but it tries to retrieve from each

so get lucky → fast

2 - bit level striping

Hamming scheme

So protect 1 of the disks

So protect all disks

- detect 1 or 2 bit errors
- Correct 1 bit errors
  - only thing that can fix errors

Rare in practice

3 - bit level + parity disk

- Sync spindal
- not too popular

4 - block level + parity disk

- good if can write whole disk
- easy to add disks
- ; NetApp relies on it

5 - Raid 4 + distributed parity

Normally raid (except 2) for disk failure - not #2 which is for bit failures

(4)

Internet Rating

Topological addressing

Lad address based on geo

18.31.*

Tier 3 - consmer

2 - regional

1 - national - are 9

transit = paying for connection

peeing = free forwarding

BGP

order ↓
- customers
- peers
- providers

Look at BGP AS Path

Filtering - which routes to advertise outward

Not sure how to prevent malicious use

P2P/CDN

- no server
- big problem - where to return resources?
- DHTs

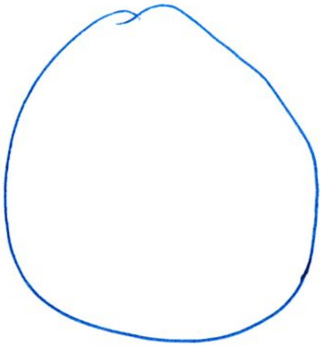
i. keyspace partitioning

Chord

(Frans wrote)

keyid = hash(filename)

node id = has (node's IP addr



key y

↓

key ID y

↓

node ID ≥ key ID

Now move to id - not popping

Just immed. successor is linear

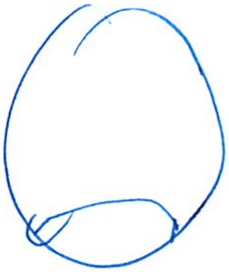
↳ only 1 neighbor passes around

dhash get (y)

↓

SHA1 (y) = key_Id = overlay network

↑ you know of <u>some</u> nodes in the system



tries to get as close as possible to segment

7

BitTorrent

Upload →

download →

BitTyrant

Dynamically changes the active set size

And breaks download/upload ratio

- uses greedy strategy
- who has highest ration

Coral CDN

1. Coral DNS - tells you which proxy to talk to
2. Coral CDN HTTP Proxies
   - if don't have go to other Coral caches
3. Coral Indexing Inf
   - DHT

8

TCP

reliable transport

Congestion control

On top of (missed)

Sliding window

RTT

Latency

Throughput

6.033 Study
Quiz 2

4/19

Best way:

- hard topic review
- reread stuff confused on
- practice exams

Map Reduce

(review recitation!)

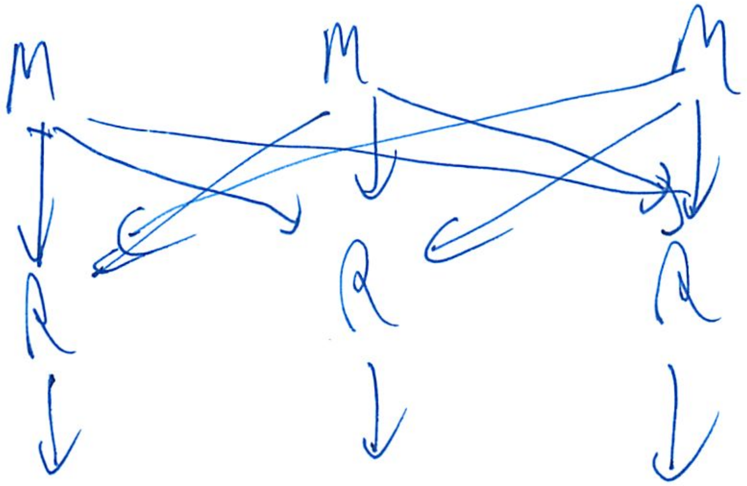
Example in class: Mapper splits names into 10 sets by 1st digit

Gives each digit to a reducer

Which sorts it

Sorted list

2



For names
- hash name then give to right group

Google sends keyword to reduce - who has a section of the alphabet
↳ indexer has loaded index to Reduce

Split # Mappers + Reducers well

Bad at boundry cases
↳ didn't get context...

(3)

key/value → Mapper → intermediate key/value → Reducer
- merges all vals w/ same key

auto paral'isible

typically 0 or 1 output

So word count

map | gets part of doc
emits (w, 1) for each word

reduce | each reducer has diff alpha range
adds up all the words

~~Comb~~

Results ~~return~~ combined together + returned

5

(So this output 1 for each word is kinda weird - but makes sense...)

Handles worker failure

L schedule elsewhere

Aborts on master failure

GFS -> copies (3) on diff machines

Schedule tasks on same machine if possible

Schedule some backups at the end

L actually fairly significant

- slow disk, etc

<u>Networks</u> enforce modularity

economical, org, physical

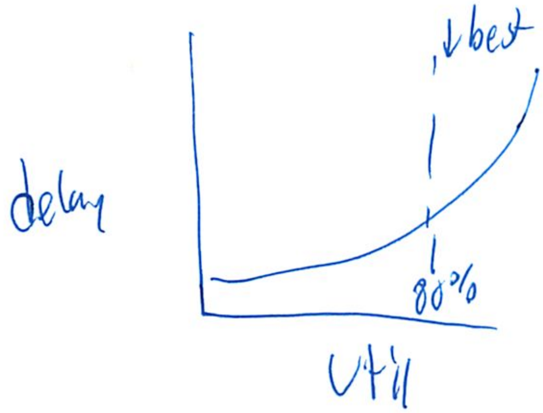
Routing -> by ID

best effort

have a buffer

Utilization

$C = \sum_i \text{max send rate}$



Throughput avg rate of sucessful deliveries

bits/sec

latency time delay

End-to-end ~~system~~ app knows best

but NN has been attacking it

(wish there will be a qu on this!)

7

NAT routers also break
- Lno public IP
- complex routing around

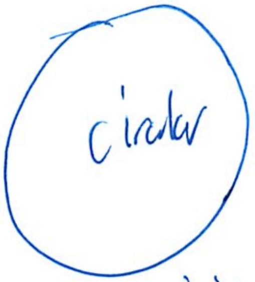
Link → Network → TCP

BGP
- Lbut paths can be suboptimal
  - loop free
  - AS = monolithic
  - update to share routes
    - except local_pref
  - AS Path
  - but hijack w/ shorter route
  - takes time to converge

8

P2P

DHT - tracker distributed to peers

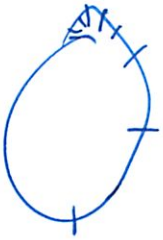


Chord

visit $\frac{n}{2}$ on avg

So instead finger that points to
sucessor of $n+2^i$
Look for highest node that preceds
log(n) hop as get closer to closer

fingers to 1/2
1/4 around the circle
1/8
⋮ etc



Read about joining - pops itself in - transfers keys

if fall off - others have it

put ?

~~remembers~~ puts data at r sucessors
L of its node id
so redundent to failure

Chord has keys to sucessors

Bit Tyrant

I still can't remember what this does exactly

Normal - everyone tit-for-tat

BitTyrant gives data to my top peers

Normally optimistic unchoking

BT_1: Upload min amt that still downloads

BT_2: Good for indv

10

TCP flow control

window ≤ recieve buffer

Congestion

Tx Rate ≤ Bottleneck Capaity

$$\text{Tx Rate} = \frac{\text{window}}{\text{RTT}}$$

Logging

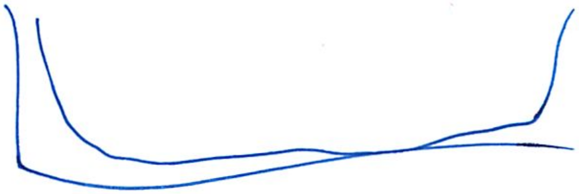
I read all this stuff - think I get it

Fault tolerant

reliable systems from unreliable components

Modules

$$\left[\frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \right.$$



rate failure high

④

Coral CDN

(I had forgotten about Choard when I read this)

Basically overkill

Transactions

before-or-after - wait for it to be finished

all or nothing - atomic whole

1. Log lot
2. Never overwrite only copy
   ↳ if fails → problem!

So instead atomic rename

RAID

I think I got this..

12

Isolation levels

printed out

MVCC

how some isolation works

Types of logging

1. Simple log
2. Write ahead log + cell
3. Redo/undo log + cell + cache

Begin()

Commit()

Recovery procedure

Log File System

Logs all of the writes

So faster to write

But reads are more common

So disaster

1. Strict serializibility
2. Snapshot isolation

   - our hands on

3. Read committed

   Started later but then committed reflected

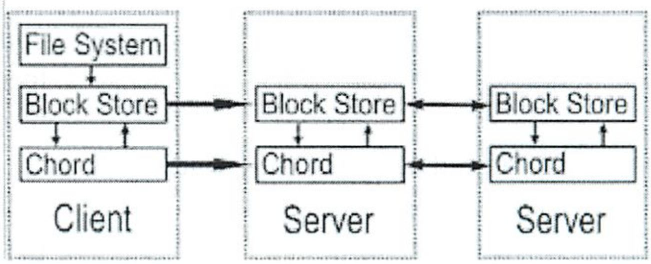
Chord (peer-to-peer)

From Wikipedia, the free encyclopedia

In computing, **Chord** is a protocol and algorithm for a peer-to-peer distributed hash table. A distributed hash table stores key-value pairs by assigning keys to different computers (known as "nodes"); a node will store the values for all the keys for which it is responsible. Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key.[1]

Chord is one of the four original distributed hash table protocols, along with CAN, Tapestry, and Pastry. It was introduced in 2001 by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, and was developed at MIT.[2]

Contents



Overview

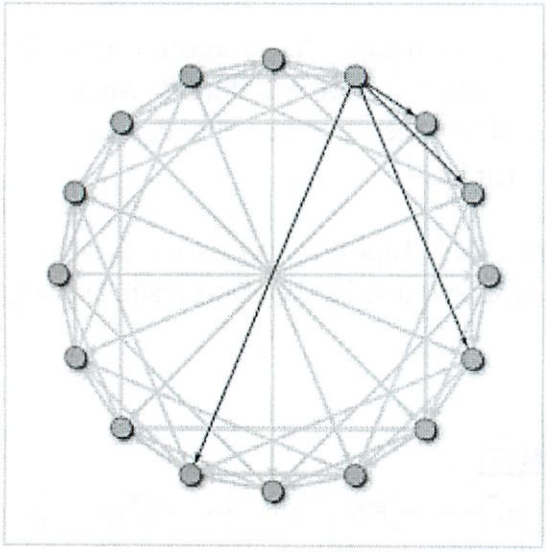
Using the Chord lookup protocol, node keys are arranged in a circle that has at most 2^m nodes. The circle can have IDs/keys ranging from 0 to $2^m - 1$.

IDs and keys are assigned an m-bit identifier using *consistent hashing*. The SHA-1 algorithm is the base hashing function for consistent hashing. Consistent hashing is integral to the robustness and performance of Chord because both keys and IDs (IP addresses) are uniformly distributed and in the same identifier space. Consistent hashing is also necessary to let nodes join and leave the network without disruption.

Each node has a *successor* and a *predecessor*. The successor to a node (or key) is the next node (key) in the identifier circle in a clockwise direction. The predecessor is counter-clockwise. If there is a node for each possible ID, the successor of node 2 is node 3, and the predecessor of node 1 is node 0; however, normally there are holes in the sequence. For example, the successor of node 153 may be node 167 (and nodes from 154 to 166 will not exist); in this case, the predecessor of node 167 will be node 153.

Since the successor (or predecessor) node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e. the r nodes preceding it and the r nodes following it. This list results in a high probability that a node is able to correctly locate its successor or predecessor, even if the network in question suffers from a high failure rate.

Chord protocol



A 16-node Chord network. The "fingers" for one of the nodes are highlighted.

The Chord protocol is one solution for connecting the peers of a P2P network. Chord consistently maps a key onto a node. Both keys and nodes are assigned an m-bit identifier. For nodes, this identifier is a hash of the node's IP address. For keys, this identifier is a hash of a keyword, such as a file name. It is not uncommon to use the words "nodes" and "keys" to refer to these identifiers, rather than actual nodes or keys. There are many other algorithms in use by P2P, but this is a simple and common approach.

A logical ring with positions numbered 0 to $2^m - 1$ is formed among nodes. Key *k* is assigned to node successor(*k*), which is the node whose identifier is equal to or follows the identifier of *k*. If there are *N* nodes and *K* keys, then each node is responsible for roughly K/N keys.

When a new node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands.

If each node knows only the location of its successor, a linear search over the network could locate a particular key. This is a naive method for searching the network, since any given message could potentially have to be relayed through most of the network. Chord implements a faster search method.

Chord requires each node to keep a "finger table" containing up to m entries. The i^{th} entry of node n will contain the address of successor$((n + 2^{i-1}) \, mod \, 2^m)$.

With such a finger table, the number of nodes that must be contacted to find a successor in an *N*-node network is $O(\log N)$. (See proof below.)

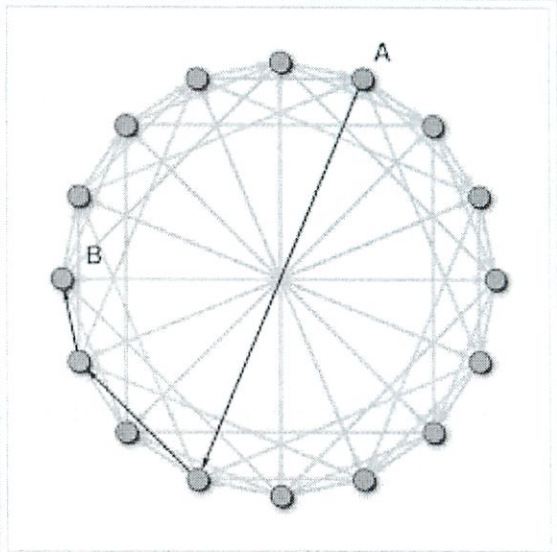
Potential uses

- Cooperative Mirroring: A load balancing mechanism by a local network hosting information available to computers outside of the local network. This scheme could allow developers to balance the load between many computers instead of a central server to ensure availability of their product.
- Time-shared storage: In a network, once a computer joins the network its available data is distributed throughout the network for retrieval when that computer disconnects from the network. As well as other computers' data is sent to the computer in question for offline retrieval when they are no longer connected to the network. Mainly for nodes without the ability to connect full time to the network.
- Distributed Indices: Retrieval of files over the network within a searchable database. e.g. P2P file transfer clients.
- Large scale combinatorial searches: Keys being candidate solutions to a problem and each key mapping to the node, or computer, that is responsible for evaluating them as a solution or not. e.g. Code Breaking

Proof sketches

With high probability, Chord contacts $O(\log N)$ nodes to find a successor in an N-node network.

Suppose node n wishes to find the successor of key k. Let p be the predecessor of k. We wish to find an upper bound for the number of steps it takes for a message to be routed from n to p. Node n will examine its finger table and route the request to the closest predecessor of k that it has. Call this node f. If f is the i^{th} entry n's finger table, then both f and p are at distances between 2^{i-1} and 2^i from n along the identifier circle. Hence, the distance between f and p along this circle is at most 2^{i-1}. Thus the distance from f to p is less than the distance from n to f: the new distance to p is at most half the initial distance.



The routing path between nodes A and B. Each hop cuts the remaining distance in half (or better).

This process of halving the remaining distance repeats itself, so after t steps, the distance remaining to p is at most $2^m/2^t$; in particular, after $\log N$ steps, the remaining distance is at most $2^m/N$. Because nodes are distributed uniformly at random along the identifier circle, the expected number of nodes falling within an interval of this length is 1, and with high probability, there are fewer than $\log N$ such nodes. Because the message always advances by at least one node, it takes at most $\log N$ steps for a message to traverse this remaining distance. The total expected routing time is thus $O(\log N)$.

If Chord keeps track of r = O(log N) predecessors/successors, then with high probability, if each node has probability of 1/4 of failing, find_successor (see below) and find_predecessor (see below) will return the correct nodes

Simply, the probability that all r nodes fail is $\left(\frac{1}{4}\right)^r = O\left(\frac{1}{N}\right)$, which is a low probability; so with high probability at least one of them is alive and the node will have the correct pointer.

Pseudocode

Definitions for pseudocode:

- finger[k]: first node that succeeds $(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
- successor: the next node from the node in question on the identifier ring
- predecessor: the previous node from the node in question on the identifier ring

The pseudocode to find the *successor* node of an id is given below:

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor] ) // Yes, that should be a closing square bracket to match the opening
    return successor;
  else
    // forward the query around the circle
    n0 = closest_preceding_node(id);
    return n0.find_successor(id);
```

```
// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i]∈(n,id))
      return finger[i];
  return n;
```

The pseudocode to stabilize the chord ring/circle after node joins and departures is as follows:

```
// create a new Chord ring.
n.create()
  predecessor = nil;
  successor = n;
```

```
// join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);
```

```
// called periodically. n asks the successor
// about its predecessor, verifies if n's immediate
// successor is consistent, and tells the successor about n
n.stabilize()
  x = successor.predecessor;
  if (x∈(n, successor))
    successor = x;
  successor.notify(n);
```

```
// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n'∈(predecessor, n))
    predecessor = n';
```

```
// called periodically. refreshes finger table entries.
// next stores the index of the finger to fix
n.fix_fingers()
  next = next + 1;
  if (next > m)
    next = 1;
  finger[next] = find_successor(n+$2^{next-1}$);
```

```
// called periodically. checks whether predecessor has failed.
n.check_predecessor()
  if (predecessor has failed)
    predecessor = nil;
```

See also

- CAN
- Kademlia
- Pastry (DHT)
- Tapestry (DHT)
- Koorde

- OverSim - the overlay simulation framework
- SimGrid - a toolkit for the simulation of distributed applications -

References

1. ^ "Chord - frequently asked questions" (http://pdos.csail.mit.edu/chord/faq.html) . Chord Project. http://pdos.csail.mit.edu/chord/faq.html.
2. ^ Stoica, Ion et al. (2001). "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". *Proceedings of SIGCOMM'01* (ACM Press New York, NY, USA).

External links

- The Chord Project (http://www.pdos.lcs.mit.edu/chord)
- Paper proposing Chord: "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications" (http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01/)
- Updated version of the above paper (http://pdos.csail.mit.edu/papers/ton:chord/)
- Open Chord - An Open Source Java Implementation (http://open-chord.sourceforge.net/)
- Chordless - Another Open Source Java Implementation (http://chordless.wiki.sourceforge.net/)
- jDHTUQ- An open source java implementation. API to generalize the implementation of peer-to-peer DHT systems. Contains GUI in mode data structure (http://sourceforge.net/projects/jdhtuq//)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Chord_(peer-to-peer)&oldid=447444452"

Categories: Distributed data storage | Software using the MIT license

- This page was last modified on 30 August 2011 at 06:28.
-

```
6.033 2012 Lecture 18: Multi-site atomicity

Administrivia
  Quiz 2 this Friday.
  Quiz review today (Wednesday) 7:30-9:30pm in 32-123

Last time: transactions provide all-or-nothing & before-or-after atomicity
  All-or-nothing: shadow copy or logging.
  Before-or-after: serializability using the 2PL locking protocol.
  Works when each atomic action is wrapped in one begin/commit block,
    on a single machine.

Today:
  Wrap-up isolation
  Distributed transactions: spanning multiple machines.

Isolation last lecture: serializability
  the schedule produced by interleaving of concurrent trans == a schedule of serial trans
     if not, then transaction system is incorrectly implemented
  implementation: strict two-phase locking
     acquire and hold lock until lock point
         various techniques to handle deadlock
     release read locks after lock point
     hold write locks until commit/abort
  isolation & recoverability
     The log must have a serializable schedule
      --> no special action require for isolation during recovery

Weaker forms of isolation:
  Goal: get more concurrency
  Approach: sacrifice serializability
  Example from hands-on
    Snap-shot isolation
    Read-committed

Snap-shot isolation: avoid read locks
  2 concurrent transfers
  T1 each write creates a new version of account
  T2 reads version from when its transaction starts

Read-committed isolation:
   No read locks and no multi-version memory

Behavior:
  see demo-17.txt
  Good way of understanding why serializability is important!

Distributed (multi-site) transactions
  Assumption so far:  all transactions run on a single machine
  But what if you want to use several machines?
     1 machine holds accounts A-K
     1 machine holds accounts K-Z
     transactions for that effects accounts on one machine can run in parallel
     but what to do about a transfer(A, Z)?
```

```
What makes this harder?
  Nodes communicate over unreliable network.
    Each node can fail independently.
    Messages can be lost or re-ordered.
    [ diagram of new scenario: internet connecting travel agent, bank, airline ]
  All-or-nothing atomicity: how do we ensure that everyone commits or aborts?
    Problem is, different components can fail independently.
    - Contrast: in a single machine, entire system crashes if power goes out.
    Problem: if one part crashes, other parts might not know about it.
    - Is the network just broken, or is the other machine really crashed?
    - Contrast: in a single computer, if disk is broken, get an error back.

Approach: distributed transactions
  Each site has its own transaction machinery: logging, recovery, etc.
  Remaining problem: ensuring that either both commit or both abort.
  We'll call the machine that started the transaction the coordinator,
    it will be in charge of ensuring agreement between nodes.

Distributed transaction steps:
  1. Coordinator sends tasks to workers (xfer to bank, issue_tkt to airline).
     Think of this as the PREPARE message in the 2PC protocol.
  2. Nodes run txn, report status to coordinator (abort or tentative commit).
     VOTE message in the 2PC protocol.
     Tentatively committed: node has no choice on whether to commit anymore!
       If coordinator says commit, it _must_ commit.
       If coordinator says abort, it similarly _must_ abort.
       In particular, must log to disk before sending tentative commit.
  3. Coordinator waits for all replies, decides whether to commit or abort.
     (e.g., if anyone aborted, then entire txn aborts; otherwise commit).
     Send message to all nodes informing them of the outcome.
     COMMIT message in the 2PC protocol (or ABORT).
  4. Nodes acknowledge the commit (or abort) with ACK message.

do l18-demo.txt

Why wait for VOTEs, instead of just sending COMMIT to finished nodes?
  If some node isn't in tentative commit, might crash and refuse to commit.
  Now we're in trouble: some nodes have committed, others have aborted.

What happens with packet loss?
  Use retransmission and duplicate suppression (keep old responses).

What happens if a worker node crashes?
  Crash before it wrote tentative-commit record.
    Coordinator resends PREPARE, worker will respond with VOTE ABORT.
  Crash after it wrote tentative-commit, before it sent VOTE.
    Coordinator resends PREPARE, worker will respond with VOTE COMMIT.
  Crash after it sent VOTE.
    Coordinator resends COMMIT, worker will turn tentative commit into commit.

What happens if the coordinator crashes?
  Need to remember what happened to the parent transaction!
  Coordinator maintains a log with each distributed transaction's outcome.
  Crash before coordinator sends PREPARE.
    No workers have promised to commit, can timeout.
  Crash after coordinator sent some (but not all) PREPAREs.
    Some workers promised to commit.
```

```
      Will resend VOTE to coordinator, coordinator can reply to abort.
    Crash after coordinator got all VOTEs and wrote its commit record.
      Workers will resend VOTEs, coordinator will send out COMMIT.

  When can we garbage-collect the logs?
    Coordinator must hear ACK from every worker before removing commit log.
      Otherwise, crashed worker might need to re-ask if some txn committed.
      Coordinator writes a "done" record when it receives all ACKs.
    Workers must keep around log in tentative-commit until coordinator decides.
      Worker not allowed to abort, even if the coordinator has crashed..
      After all, it may be the network just being slow: cannot tell.
    2PC guarantees that _eventually_ everyone will agree on outcome.
      But cannot guarantee when this will occur: might take arbitrarily long..
      In the meantime, nodes must keep their logs, and maybe keep locks held.

  Summary
    Distributed transactions
    Two-phase commit can eventually agree on commit/abort (but no time bound).
    2PC coordinator, workers may have to keep state for arbitrarily long time.

  --- Unused ---

  Why nested transactions?
    Want to build a large transaction out of smaller components.
    For example: buying an airline ticket, via a travel agency.
    [ slide: buying a ticket with transactions ]
    Existing programs might have transactions for xfer() and issue_tkt().
    Travel agency wants to combine them into one transaction (purchase a ticket).
    Might actually want xfer() and issue_tkt() to run concurrently, so still
      want before-or-after atomicity for them, just in case.

  Why are nested transactions tricky?
    Hard questions: when to log, commit, release locks, etc.
    For example, xfer may commit, but computer crashes before purchase commits.
    The transfer should not happen in this case.
      Outer transaction supposedly has all-or-nothing atomicity.
      Since it didn't commit, none of its parts should appear to have run.

  Easy strawman: ignore the begin and commit records for any sub-transaction.
    Sort-of works: all-or-nothing atomicity for outer transaction.
    Problem with aborts: causes entire outer txn to abort, not just sub-txn.
    Would like xfer() to abort if customer doesn't have enough money, and
      allow them to retry with a different account, without losing the ticket
      (i.e., aborting the issue_tkt transaction).
    Problem with concurrency: xfer() & issue_tkt() might not be before-or-after.
      Need locking between sub-transactions to work.

  Nested transactions with independent aborts require several changes:
    Modified locking protocol.
    Modified logging/recovery.
    New state for a transaction: tentatively committed.
      Means that sub-transaction committed, but parent is still undecided.

  Nested locking protocol
    When can we release locks held by issue_tkt() or xfer()?
      Must hold on to locks until parent txn commits.
    How can one sub-transaction read another sub-txn's data?
```

```
      Check if lock is held by a tentatively-committed sub-txn with same parent.
      If so, transfer lock ownership to the new sub-txn.

  Nested logging/recovery:
    Need separate transaction IDs for each sub-txn.
    If a sub-txn aborts, need to undo the change records for that sub-txn.
    If a sub-txn commits, need to write a tentative commit record,
      indicating it's part of some parent txn.
    During recovery, need to figure out if parent txn committed.
    If parent didn't commit, will need to abort the tentatively-committed txn.
    Otherwise, approximately same recovery as before:
      scan to determine winners, losers (take into account parent txns).
      undo losers (backward scan).
      redo winners (forward scan).

  Often, there are two more phases upfront:
    - Send tasks to workers, but without the PREPARE message.
    - Workers report back to coordinator their status, without tentative commit.
    Only then does 2PC start (PREPARE, VOTE, COMMIT, ACK).
    Reason is that tentative-commit state is expensive to maintain
      Forced log writes and many messages
      Node can't decide to abort, even if it would be convenient (e.g., crash).
```

Search Documentation: Search Search
Text Size: Normal / Large
Home → Documentation → Manuals → PostgreSQL 8.4

PostgreSQL 8.4.11 Documentation

Prev Fast Backward Chapter 13. Concurrency Control Fast Forward Next

13.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty read

> A transaction reads data written by a concurrent uncommitted transaction.

nonrepeatable read

> A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

> A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

The four transaction isolation levels and the corresponding behaviors are described in Table 13-1.

Table 13-1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

In PostgreSQL, you can request any of the four standard transaction isolation levels. But internally, there are only two distinct isolation levels, which correspond to the levels Read Committed and Serializable. When you select the level Read Uncommitted you really get Read Committed, and when you select Repeatable Read you really get Serializable, so the actual isolation level might be stricter than what you select. This is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen. The reason that PostgreSQL only provides two isolation levels is that this is the only sensible way to map the standard isolation levels to the multiversion concurrency control architecture. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command SET TRANSACTION.

13.2.1. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction uses this isolation level, a `SELECT` query (without a `FOR UPDATE/SHARE` clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. In effect, a `SELECT` query sees a snapshot of the database as of the instant the query begins to run. However, `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes during execution of the first `SELECT`.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the `WHERE` clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation using the updated version of the row. In the case of `SELECT FOR UPDATE` and `SELECT FOR SHARE`, this means it is the updated version of the row that is locked and returned to the client.

Because of the above rule, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions; however, it is just right for simpler cases. For example, consider updating bank balances with transactions like:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start with the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

More complex usage can produce undesirable results in Read Committed mode. For example, consider a `DELETE` command operating on data that is being both added and removed from its restriction criteria by another command, e.g., assume `website` is a two-row table with `website.hits` equaling `9` and `10`:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

The `DELETE` will have no effect even though there is a `website.hits = 10` row before and after the `UPDATE`. This occurs because the pre-update row value `9` is skipped, and when the `UPDATE` completes and `DELETE` obtains a lock, the new row value is no longer `10` but `11`, which no longer matches the criteria.

Because Read Committed mode starts each command with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue above is whether or not a single command sees an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use; however, it is not sufficient for all cases. Applications that do complex queries and updates might require a more rigorously consistent view of the database than Read Committed mode provides.

13.2.2. Serializable Isolation Level

The *Serializable* isolation level provides the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is using the serializable level, a `SELECT` query only sees data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is different from Read Committed in that a query in a serializable transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. Thus, successive `SELECT` commands within a single transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started. (This behavior can be ideal for reporting applications.)

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the serializable transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the serializable transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the serializable transaction will be rolled back with the message

```
ERROR:  could not serialize access due to concurrent update
```

because a serializable transaction cannot modify or lock rows changed by other transactions after the serializable transaction began.

When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning. The second time through, the transaction will see the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

The Serializable mode provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution. Since the cost of redoing complex transactions can be significant, serializable mode is recommended only when updating transactions contain logic sufficiently complex that they might give wrong answers in Read Committed mode. Most commonly, Serializable mode is necessary when a transaction executes several successive commands that must see identical views of the database.

13.2.2.1. Serializable Isolation versus True Serializability

The intuitive meaning (and mathematical definition) of "serializable" execution is that any two successfully committed concurrent transactions will appear to have executed strictly serially, one after the other — although which one appeared to occur first might not be predictable in advance. It is important to realize that forbidding the undesirable behaviors listed in Table 13-1 is not sufficient to guarantee true serializability, and in fact PostgreSQL's Serializable mode does not guarantee serializable execution in this sense. As an example, consider a table `mytab`, initially containing:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Suppose that serializable transaction A computes:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

and then inserts the result (30) as the `value` in a new row with `class = 2`. Concurrently, serializable transaction B computes:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

and obtains the result 300, which it inserts in a new row with `class = 1`. Then both transactions commit. None of the listed undesirable behaviors have occurred, yet we have a result that could not have occurred in either order serially. If A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

To guarantee true mathematical serializability, it is necessary for a database system to enforce *predicate locking*, which means that a transaction cannot insert or modify a row that would have matched the `WHERE` condition of a query in another concurrent transaction. For example, once transaction A has executed the query `SELECT ... WHERE class = 1`, a predicate-locking system would forbid transaction B from inserting any new row with class 1 until A has committed. [1] Such a locking system is complex to implement and extremely expensive in execution, since every session must be aware of the details of every query executed by every concurrent transaction. And this large expense is mostly wasted, since in practice most applications do not do the sorts of things that could result in problems. (Certainly the example above is rather contrived and unlikely to represent real software.) For these reasons, PostgreSQL does not implement predicate locking.

In cases where the possibility of non-serializable execution is a real hazard, problems can be prevented by appropriate use of explicit locking. Further discussion appears in the following sections.

Notes

[1] Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas MVCC prevents them by restricting what is read.

Prev	Home	Next
Introduction	Up	Explicit Locking

Privacy Policy | About PostgreSQL

6.033: Computer Systems Engineering Spring 2012

Home / News
Schedule
Submissions

General Information
Staff List
Recitations
TA Office Hours

Discussion / feedback
FAQ
Class Notes Errata
Excellent Writing Examples

2011 Home

Preparation for Quiz 2

Quiz 2 will last 50 minutes. The quiz will cover material from lecture 9 through lecture 18 (inclusive). The quiz will be "open book." That means you can bring along any printed or written materials that you think might be useful. Calculators are allowed, though not necessary. You may also bring a laptop to view PDF versions of papers, but you may not connect to any network. 6.033 quizzes typically begin with half a dozen or so independent questions that focus on material in the readings followed by one or two longer, multi-part questions that deal with concepts of computer systems covered in the notes and lecture. The quiz will be, at least in part, multiple choice.

Warning: the lectures touch on only a modest subset of the totality of 6.033 concepts. Anything in the class notes assigned from LEC 9 through LEC 18 may show up on the quiz.

You can find old quiz questions to practice on in the Problem Sets section of the class notes. Although there has been a little rearrangement of material over the years, you should be able to answer most questions relevant to the subjects we have covered.

Old quizzes:

- 2011 - Quiz 2
- 2010 - Quiz 2
- 2009 - Quiz 2
- 2008 - Quiz 2
- 2007 - Quiz 2
- 2006 - Quiz 2

Old quizzes w/ solutions

- 2011 - Quiz 2 Solutions
- 2010 - Quiz 2 Solutions
- 2009 - Quiz 2 Solutions
- 2008 - Quiz 2 Solutions
- 2007 - Quiz 2 Solutions
- 2006 - Quiz 2 Solutions
- 2005 - Quiz 2 Solutions
- 2004 - Quiz 2 Solutions

Questions or comments regarding 6.033? Send e-mail to the 6.033 staff at 6.033-staff@mit.edu or to the 6.033 TAs at 6.033-tas@mit.edu.

Top // 6.033 home //

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2011

Quiz 2

There are 11 questions and 8 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

For true/false questions, you will receive 0 points for no answer, and negative points for an incorrect answer. Do not guess; if you are unsure about your answer, consult your notes. The total score for each numbered question (1 through 11) will be at least 0 (i.e., those scores cannot go negative).

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below. Write your initials at the bottom of each page.

THIS IS AN OPEN BOOK, OPEN NOTES, OPEN LAPTOP QUIZ, BUT DON'T USE YOUR LAPTOP FOR COMMUNICATION WITH OTHERS.

CIRCLE your recitation section number:

10:00	1. Lampson/Pesterev	
11:00	2. Lampson/Pesterev	3. Ports/Mutiso
12:00		4. Ports/Mutiso
1:00	5. Katabi/Raza	6. Strauss/Narula
2:00	7. Katabi/Raza	8. Strauss/Narula

Do not write in the boxes below

1-2 (xx/20)	3-4 (xx/20)	5-6 (xx/20)	7-9 (xx/20)	10-11 (xx/20)	Total (xx/100)

Name:

I Reading Questions

1. [10 points]: Based on the paper "Do incentives build robustness in BitTorrent?", which of the following statements are correct?

(Circle True or False for each choice.)

? **A.** [✓ **True**] **/ False** A reference BitTorrent client will attempt to split its outgoing bandwidth equally between every peer it can connect to.

B. [✓ **True**] **/ False** BitTorrent permits content providers to shift bandwidth costs from their own ISP to those paid for by their content consumers.

C. True / [✓ **False**] The *torrent* file that a BitTorrent client downloads before joining a swarm contains the IP addresses of the *seed* nodes.

D. [✓ **True**] **/ False** The *BitTyrant* client uses unequal outgoing bandwidth allocations as one strategy to improve its download performance.

2. [10 points]: Based on the paper "A case for redundant arrays of inexpensive disks (RAID)" which of the following statements are correct?

(Circle True or False for each choice.)

A. [✓ **True**] **/ False** A RAID array can improve reliability when individual disks fail independently from each other, but is less useful in the face of highly correlated failures.

B. True / [✓ **False**] The RAID paper predicted that disk seek times would improve at an annual rate similar to improvements seen in transistor density in microprocessors.

C. True / [✓ **False**] If average disk capacities grow proportionally faster over time than sequential data transfer rates between disks and disk controllers, the MTTR (mean time to repair) of a RAID array will decrease.

D. True / [✓ **False**] If solid-state disks entirely replace spinning magnetic disks, none of the approaches described in the RAID paper will be needed anymore, since SSDs do not need to seek between different portions of the disk.

II Border Gateway Protocol (BGP)

These questions are based on reading "An Introduction to Wide-Area Internet Routing".

3. [10 points]: Ben Bitdiddle was asked to debug a BGP problem. Help Ben to brush up his information about BGP. Which of the following statements are correct?

(Circle True or False for each choice.)

A. [✓ **True**] / **False** To make money, an AS should announce the AS routes it learned from its customers to its peers.

(, **B.** [✓ **True**] / [✓ **False**] To make money, an AS should not announce the AS routes it learned from its peers to its provider.

C. [✓ **True**] / **False** In BGP, an AS may learn many AS routes to an IP prefix.

D. [✓ **True**] / [✓ **False**] In BGP, the customer-provider relationships imply that if an AS has announced a route for a particular prefix to a neighbor, and later it learned a new route to that prefix, the AS will announce the new route to the neighbor.

(, maybe not if Hops ?

but I think anyway

Won't annonce worse routes

Not clear

└ Shares routing table

but does this have old info

```
        Sprint
      /        \
  MIT AS        AS1
                  \
                   AS2
                     \
                      ...
                        \
                         AS10
                           \
                            AS EVIL
```

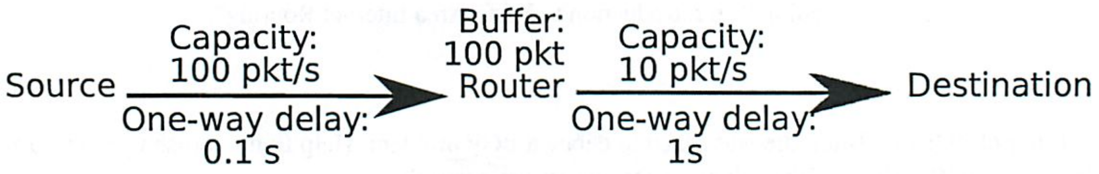
4. [10 points]: Consider the topology shown above. For any link, assume the higher node is a provider and the lower is a customer. MIT advertises prefix 18.* to Sprint. How can AS EVIL hijack most traffic destined to MIT from Sprint? (Describe briefly.)

By publishing a 1 hop route to MIT to all ASes 1 → 10

⊗ Longest prefix match

(, so can't fake hop cang

III Congestion control



Consider the above topology and assume the following:

did not study

- Link between Source and Router has a capacity of 100 packets/second and a propagation delay of 0.1 second in the forward direction.
- Link between Router and Destination has a capacity of 10 packets per-second and a propagation delay of 1 second in the forward direction. The transmission delays are 0.
- Acknowledgments from the destination to the source are delivered on a different path which has infinite capacity and zero delay, i.e., this means that acknowledgments will be delivered immediately from the destination to the source and will not suffer any drops.
- The Source and Destination can process packets very fast and the destination does not have any buffering limitations.
- The Router has a queue that can store a maximum of 100 packets.

5. [10 points]: Assume the Source uses a fixed window whose size is set to W. What is the maximum throughput in packet/sec that the source can deliver to its destination, if W=5, and then if W=20? (You can round your answer to the nearest integer.)

W = 5: ____________ pkt/s

W = 20: ____________ pkt/s

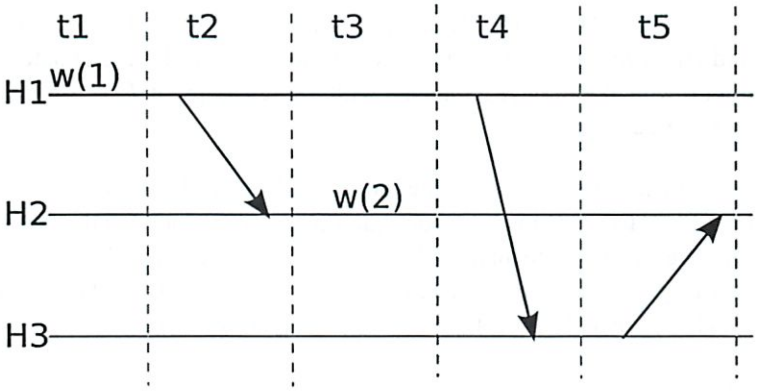
6. [10 points]: Assume that the source uses TCP. Focus on how TCP adapts its congestion window using AIMD as described in class. Ignore other details of TCP (e.g., slow start, fast retransmission and fast recovery). What is the maximum window size that the source's TCP will experience during AIMD?

Maximum window size: ____________ packets

IV File reconciliation and time vectors

Ben designs a new tool for reconciling copies of a file on different computers. The scenario is a single user who has a number of computers, each of which may have a copy of a file f, and the user updates the copies of f independently from each other. For example, when the user is on an airplane, he can update only the copy on his laptop, and when the user gets to his office but forgets his laptop at home, he can update only the copy on his office machine. Ben's goal is that the different copies of f should behave as if there is a single copy of f.

To develop a design, he considers two update and reconciliation patterns. The first pattern for file f is as follows:



It shows 3 hosts: H1, H2, and H3. H1 writes (w) the value 1 to the file f, and then reconciles to H2 (i.e., changes on H1 are propagated to H2 and reconciled with changes on H2). Later H2 writes 2 to the file f and H1 reconciles to H3. Then, H3 reconciles to H2. To make it easy to talk about the different events, we have labeled them with real times t1 through t5, but you should assume that the clocks of the different computers are not synchronized and the computers have no agreement on this global time.

7. [5 points]: For the ideal outcome (i.e., f should behave as if there is a single copy), what value should f contain at H1, H2, and H3 after the above update and reconciliation pattern completes (i.e., at the end of t5)?

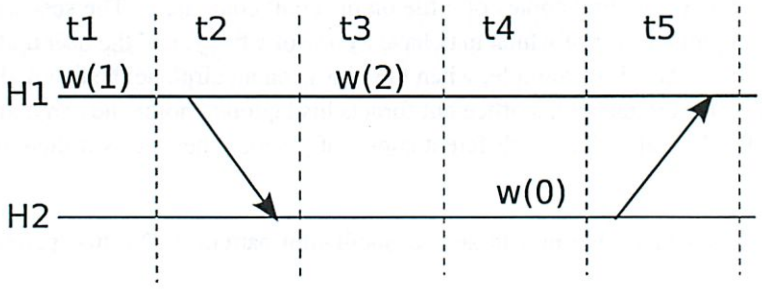
(Circle True or False for each choice.)

A. True / False H1: 1, H2: 2, H3: 2

B. True / False H1: 1, H2: 2, H3: 1

Initials:

After reading the Unison paper, Ben realizes that it isn't always possible to come to agreement between nodes what the value of f should be. Consider the following second pattern:



H1 writes 1 into f and reconciles to H2. Then, H1 and H2 modify f in different ways, and then H2 reconciles to H1 at t5. At t5, H1 and H2 have a *conflict* because there is no way to order the updates at H1 and H2 at t3 and t4 that is consistent with what could have happened with a single copy of f. A user needs to get involved to determine what the right value is.

Ben's goal is to design a tool that reconciles updates to a file f correctly when it is possible, but raises a conflict when it is not possible. His design is based on time vectors (introduced in the lecture on time, but you don't need to recall the lecture to be able to answer this question). A time vector is a vector of integers, where entry i counts the updates to f on computer i. For example, the vector (0, 0) on host 2 signals that it hasn't seen updates for f from H1, and that H2 itself hasn't done any updates either.

8. [10 points]: For the pattern above (the second pattern), what is the value of the time vector for f at each node at the end of t1 through t4? (complete the table)

time	H1	H2
t1	(1, 0)	(0, 0)
t2	—	
t3		—
t4	—	

9. [5 points]: How can Ben's design detect that pattern 2 is a conflict? Be specific, explain using the time vectors in the examples above.

V Write-Ahead Logging

Ben hasn't solved enough problems in the last hour. Inspired by the lectures on atomicity and hands-on #5, he decides to build a database system for his bank. The database is a simple one: its cell storage is a table of account numbers and balances stored on disk, and it has only one operation: `credit(account, delta)` adjusts the balance of the specified account by the (possibly negative) amount `delta`.

Ben wants the database to support atomic transactions, so he adds the usual `begin` and `commit` operations, and stores a log on a separate disk. The operations are implemented as follows:

- `begin` appends a ⟨BEGIN, *transactionid*⟩ record to the log.
- `credit(account, delta)` updates the balance of the appropriate account in memory, but does *not* write the new balance to cell storage. It also appends to the log the record:
  ⟨UPDATE, *transactionid*, *account*, *delta*, *oldbalance*, *newbalance*⟩
- `commit` first appends to the log ⟨OUTCOME COMMITTED, *transactionid*⟩. It then writes out to cell storage any account balances modified by the transaction. When this is done, it appends to the log ⟨END, *transactionid*⟩

Ben knows that this protocol can support all-or-nothing atomicity across system crashes. He sketches a recovery procedure. After a crash, the system will scan the log and *redo* the effects of any transaction that logged an OUTCOME COMMITTED record but not an END record. But rather than implementing the recovery procedure, he puts the system into production immediately. After all, Worse Is Better, and he can always write that recovery procedure later...

Inevitably, Ben's system crashes before he gets around to writing that recovery procedure. The bank's customers are not amused. Panicked, Ben turns to you to help him recover the state of his database.

Initials:

The cell storage of the database at the time of the crash, and the last few entries of the log appear below. All earlier entries in the log correspond to transactions known to have completed in their entirety.

Cell Storage

AccountID	Balance
account1	\$890
account2	\$648
account3	\$32
account4	\$1500

Log

⟨BEGIN, T_1⟩
⟨UPDATE, T_1, *account*2, 200, 448, 648⟩
⟨OUTCOME COMMITTED, T_1⟩
⟨END, T_1⟩
⟨BEGIN, T_2⟩
⟨BEGIN, T_3⟩
⟨UPDATE, T_3, *account*1, 500, 390, 890⟩
⟨UPDATE, T_3, *account*4, −500, 1500, 1000⟩
⟨UPDATE, T_2, *account*3, 100, 32, 132⟩
⟨OUTCOME COMMITTED, T_3⟩

10. [10 points]: Fill in the values that will appear in the cell storage once recovery is completed using Ben's intended recovery scheme:

AccountID	Balance
account1	
account2	
account3	
account4	

11. [10 points]: Ben is worried that the write-ahead log will take up too much space. Given Ben's recovery procedure, which of the following options would reduce the size of the log, while still being able to persistently store data, and recover from crashes with all-or-nothing atomicity?

(Circle True or False for each choice.)

A. True / False eliminating the BEGIN record.

B. True / False removing both the *oldbalance* and *delta* fields from the UPDATE record

C. True / False removing both the *oldbalance* and *newbalance* fields from the UPDATE record

D. True / False periodically writing a CHECKPOINT record to the log which contains the ID of each uncommitted transaction, then discarding all log entries before the CHECKPOINT record.

End of Quiz I

Please double check that you wrote your name on the front of the quiz,
and circled your recitation section number.

Initials:

Department of Electrical Engineering and Computer Science

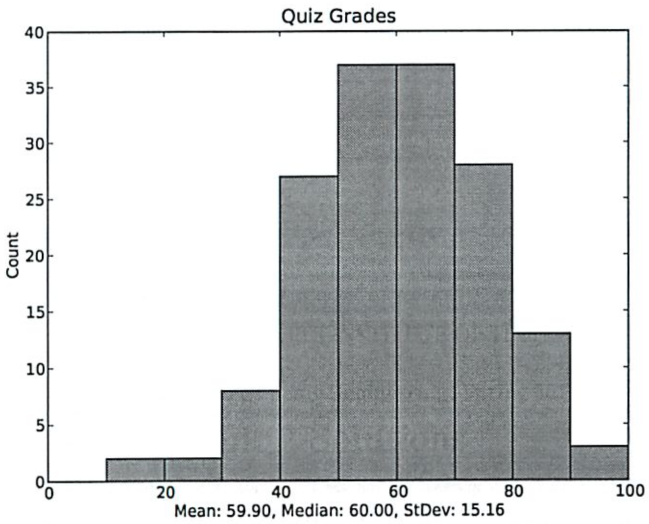
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2011

Quiz 2 Solutions

There are 11 questions and 12 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Grade distribution histogram:



I Reading Questions

1. [10 points]: Based on the paper "Do incentives build robustness in BitTorrent?", which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False A reference BitTorrent client will attempt to split its outgoing bandwidth equally between every peer it can connect to.

Answer: False. A BitTorrent client C only sends data to a peer P if P sends data to C at a rate fast enough to merit inclusion in the C's active set, or if C has optimistically unchoked P.

B. True / False BitTorrent permits content providers to shift bandwidth costs from their own ISP to those paid for by their content consumers.

Answer: True. A content provider publishes a torrent file and might initially seed the data, content consumers use BitTorrent's peer-to-peer protocol to further share data.

C. True / False The *torrent* file that a BitTorrent client downloads before joining a swarm contains the IP addresses of the *seed* nodes.

Answer: False. A torrent file contains the name and size of the file to be downloaded as well as SHA-1 fingerprints of the content blocks.

D. True / False The *BitTyrant* client uses unequal outgoing bandwidth allocations as one strategy to improve its download performance.

Answer: True. A BitTyrant client sets its upload contribution to a specific peer just high enough so that the peer reciprocates.

2. [10 points]: Based on the paper "A case for redundant arrays of inexpensive disks (RAID)" which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False A RAID array can improve reliability when individual disks fail independently from each other, but is less useful in the face of highly correlated failures.

Answer: True. If multiple disks fail at the same time, a RAID array will not be able to prevent data loss. However, if multiple disks fail over time, an administrator may be able to replace one failed disk (and allow RAID time to rebuild it) before the next one fails.

B. True / False The RAID paper predicted that disk seek times would improve at an annual rate similar to improvements seen in transistor density in microprocessors.

Answer: False. Seek times are limited by physical movement of the disk arm, and have not been improving much.

C. True / False If average disk capacities grow proportionally faster over time than sequential data transfer rates between disks and disk controllers, the MTTR (mean time to repair) of a RAID array will decrease.

Answer: **False.** Repairing a RAID array after a disk failure requires writing a disk's worth of data to the new disk. If the ratio of disk capacity to transfer speed grows, this will require more time, not less.

D. True / False If solid-state disks entirely replace spinning magnetic disks, none of the approaches described in the RAID paper will be needed anymore, since SSDs do not need to seek between different portions of the disk.

Answer: **False.** The RAID paper deals with failures of disks, and SSDs can still fail (albeit some of the reasons for failure are different).

II Border Gateway Protocol (BGP)

These questions are based on reading "An Introduction to Wide-Area Internet Routing".

3. [10 points]: Ben Bitdiddle was asked to debug a BGP problem. Help Ben to brush up his information about BGP. Which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False To make money, an AS should announce the AS routes it learned from its customers to its peers.

Answer: **True.** These customers run services on their own servers and expect that any user on the Internet is able to reach their servers. Customers pay an AS to spread the reachability information of these servers to all routers on the Internet.

B. True / False To make money, an AS should not announce the AS routes it learned from its peers to its provider.

Answer: **True.** Two ASes establish a peering link to exchange traffic at no cost to each other. An AS, on the other hand, needs to pay its provider for any traffic the AS send to the provider or the provider sends to the AS. If an AS announces the routes it hears from its peers to its provider, the provider might route packets to the peers via the AS. The AS will in fact have to pay the provider for this incoming data.

C. True / False In BGP, an AS may learn many AS routes to an IP prefix.

Answer: **True.** One example is multi-homing. A customer may want to have two links to the Internet from two different providers. These different providers will announce routes to the same IP prefix up to their respective providers. At some point these announcements will reach a Tier 1 provider which will hear two routes to the same IP prefix.

D. True / False In BGP, the customer-provider relationships imply that if an AS has announced a route for a particular prefix to a neighbor, and later it learned a new route to that prefix, the AS will announce the new route to the neighbor.

Answer: **False.** Here is one scenario: the next hop in the old route uses a peer link, the new route's next hop uses a transit link. Financially, it is best for the AS to choose the old route over the new route; the AS will not advertise the new route.

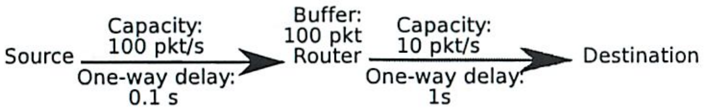
```
        Sprint
      /        \
   MIT AS       AS1
                  \
                   AS2
                    \
                     ...
                      \
                       AS10
                        \
                         AS EVIL
```

4. [10 points]: Consider the topology shown above. For any link, assume the higher node is a provider and the lower is a customer. MIT advertises prefix 18.* to Sprint. How can AS EVIL hijack most traffic destined to MIT from Sprint? (Describe briefly.)

Answer: It is no sufficient for AS EVIL to announce the 18.* prefix because Sprint will choose the shorter path to MIT and will not use AS EVIL's path. Instead, AS EVIL can take advantage of Longest Prefix Match to advertise prefixes that are more specific than 18.*. One scheme would announce 256 prefixes from 18.1.* to 18.256.*. Sprint's routers will choose the more specific prefix over MIT's 18.* prefix and send all traffic to AS EVIL.

Initials:

III Congestion control



Consider the above topology and assume the following:

- Link between Source and Router has a capacity of 100 packets/second and a propagation delay of 0.1 second in the forward direction.
- Link between Router and Destination has a capacity of 10 packets per-second and a propagation delay of 1 second in the forward direction. The transmission delays are 0.
- Acknowledgments from the destination to the source are delivered on a different path which has infinite capacity and zero delay, i.e., this means that acknowledgments will be delivered immediately from the destination to the source and will not suffer any drops.
- The Source and Destination can process packets very fast and the destination does not have any buffering limitations.
- The Router has a queue that can store a maximum of 100 packets.

5. [10 points]: Assume the Source uses a fixed window whose size is set to W. What is the maximum throughput in packet/sec that the source can deliver to its destination, if W=5, and then if W=20? (You can round your answer to the nearest integer.)

W = 5: ____________ pkt/s

Answer: The main point of this question is to realize that when the window is too small, it limits the throughput, but when the window is too large the throughput is limited by the capacity of the bottleneck link (given the system has enough buffering). Thus, when W = 5 packets, the delay inside the network is due to the propagation delay which is 0.1+1 = 1.1 second. The transmission rate = W/RTT = 5/1.1 packets/s. Since this is smaller than the bottleneck capacity of 10 packets/s, the throughput is 5/1.1 packets/s. A more accurate answer includes in the RTT computation the time to process a packet on each of the links, which is 0.01 seconds on the first link and 0.1 second on the second link. Hence, the RTT is 0.1+0.01+1+0.1 = 1.21 seconds. The throughput will be 5/1.21 packets/second. We accepted both answers since some routers can start forwarding the packet on the next link even before they receiver the last bit from the previous link.

W = 20: ____________ pkt/s

Answer: When W=20 packets, the throughput is 10 packets/second, which can be found immediately by noticing that in this scenario you are limited by the bottleneck capacity. The transmission rate is still given by W/RTT, however the throughput can never exceed the bottleneck capacity of 10 packets/second. In this case, the RTT will increase due to increased queue size at the router and the accompanying increase in delay, so that W/RTT is 10 packet/s.

Initials:

6. [10 points]: Assume that the source uses TCP. Focus on how TCP adapts its congestion window using AIMD as described in class. Ignore other details of TCP (e.g., slow start, fast retransmission and fast recovery). What is the maximum window size that the source's TCP will experience during AIMD?

Maximum window size: ____________ packets

Answer: The correct answer is 111 packets. Specifically, the TCP window will grow until the first drop, which will occur once the router's queue is full, and the bottleneck link has 10 packets along the link. This is total of 110 packets inside the network. Say the network has that number of packets in the router's queue and on the bottleneck link. The next RTT, when the TCP tries to increase its window by 1 more packet, i.e., W=111, the new packet will reach the router before the router can empty a spot in its queue and the drop will occur.

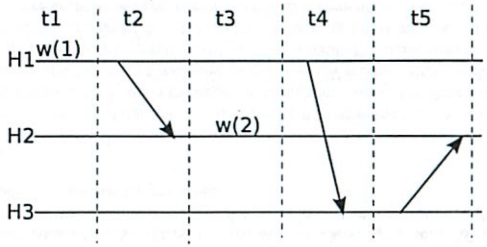
We accepted also 110 because this is the window at which the drop occurs. We also accepted 121 as a good answer because a TCP may get to 121 packets before it sees a drop if the window is initialized to 120. Specifically, the first link which has a capacity of 100 packets/s and a delay of 0.1 second, can keep 10 additional packets inside the network for a total of 120 packets in the network. This is however conditioned on starting with a large window and sending the packets over the first link in a burst.

We gave partial credit to students who realized that the router's queue has to be full before there is a drop and hence the TCP window will grow to 100 packets. These students missed the fact that there are some packets on the bottleneck link but got the buffer part. We also gave partial credits to students who have a correct argument but thought that TCP increases its window by 1 packet per ack (rather than 1 packet per RTT).

IV File reconciliation and time vectors

Ben designs a new tool for reconciling copies of a file on different computers. The scenario is a single user who has a number of computers, each of which may have a copy of a file f, and the user updates the copies of f independently from each other. For example, when the user is on an airplane, he can update only the copy on his laptop, and when the user gets to his office but forgets his laptop at home, he can update only the copy on his office machine. Ben's goal is that the different copies of f should behave as if there is a single copy of f.

To develop a design, he considers two update and reconciliation patterns. The first pattern for file f is as follows:



It shows 3 hosts: H1, H2, and H3. H1 writes (w) the value 1 to the file f, and then reconciles to H2 (i.e., changes on H1 are propagated to H2 and reconciled with changes on H2). Later H2 writes 2 to the file f and H1 reconciles to H3. Then, H3 reconciles to H2. To make it easy to talk about the different events, we have labeled them with real times t1 through t5, but you should assume that the clocks of the different computers are not synchronized and the computers have no agreement on this global time.

7. [5 points]: For the ideal outcome (i.e., f should behave as if there is a single copy), what value should f contain at H1, H2, and H3 after the above update and reconciliation pattern completes (i.e., at the end of t5)?

(Circle True or False for each choice.)

A. True / False H1: 1, H2: 2, H3: 2

Answer: False.

B. True / False H1: 1, H2: 2, H3: 1

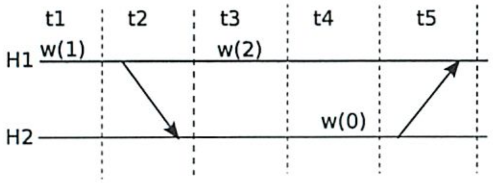
Answer: True.

Reconciliations are unidirectional, as stated in the question and as indicated by the arrows.

At t2, H2 sees that H1 has a more recent value of f and set its own value of f to equal 1 because of the reconciliation. At t3, H2 changes its own local value of f to 2. At t4, H3 sees that H1 has a more recent value of f and set its own value of f to equal 1 because of the reconciliation. At t5, H2 ignores the value of f from H3 despite the reconciliation, because it has a more recent value of f. H2 has seen H1's write, and later made a change to f, so H2's value should not by overwritten by H3.

We end up with H1: 1, H2: 2, H3: 1.

After reading the Unison paper, Ben realizes that it isn't always possible to come to agreement between nodes what the value of f should be. Consider the following second pattern:



H1 writes 1 into f and reconciles to H2. Then, H1 and H2 modify f in different ways, and then H2 reconciles to H1 at t5. At t5, H1 and H2 have a *conflict* because there is no way to order the updates at H1 and H2 at t3 and t4 that is consistent with what could have happened with a single copy of f. A user needs to get involved to determine what the right value is.

Ben's goal is to design a tool that reconciles updates to a file f correctly when it is possible, but raises a conflict when it is not possible. His design is based on time vectors (introduced in the lecture on time, but you don't need to recall the lecture to be able to answer this question). A time vector is a vector of integers, where entry i counts the updates to f on computer i. For example, the vector (0, 0) on host 2 signals that it hasn't seen updates for f from H1, and that H2 itself hasn't done any updates either.

8. [10 points]: For the pattern above (the second pattern), what is the value of the time vector for f at each node at the end of t1 through t4? (complete the table)

time	H1	H2
t1	(1, 0)	(0, 0)
t2	—	
t3		—
t4	—	

Answer:

time	H1	H2
t1	(1, 0)	(0, 0)
t2	—	*(1, 0)*
t3	*(2, 0)*	—
t4	—	*(1, 1)*

When H2 and H1 reconcile at t2, H2 will compare its vector (0, 0) with H1's vector (1, 0). Because H1's vector is more recent – (1, 0) $\geq$ (0, 0) – H2 will fetch the recent value of f from H1, and set its own vector to be (1, 0).

At t3, H1 makes a change to its copy of f, so it updates its vector to mark a write. H1's vector becomes (2, 0).

At t4, H2 makes a change to its copy of f, and it also updates its vector to mark a write. H2's vector becomes (1, 1).

9. [5 points]: How can Ben's design detect that pattern 2 is a conflict? Be specific, explain using the time vectors in the examples above.

Answer: When H1 and H2 reconcile at t5, they will notice a conflict because of concurrent updates to f. Specifically, pattern 2 is a conflict because vectors $v = (2,0)$ and $w = (1,1)$ cannot be ordered with respect to one another. We can order v and w if $v[i] \geq w[i]$ for all i (because then $v \geq w$), or if $w[i] \geq v[i]$ for all i (because then $w \geq v$). Here, $v[0] > w[0]$ but $v[1] < w[1]$.

V Write-Ahead Logging

Ben hasn't solved enough problems in the last hour. Inspired by the lectures on atomicity and hands-on #5, he decides to build a database system for his bank. The database is a simple one: its cell storage is a table of account numbers and balances stored on disk, and it has only one operation: `credit(account, delta)` adjusts the balance of the specified account by the (possibly negative) amount `delta`.

Ben wants the database to support atomic transactions, so he adds the usual `begin` and `commit` operations, and stores a log on a separate disk. The operations are implemented as follows:

- `begin` appends a ⟨BEGIN, *transactionid*⟩ record to the log.
- `credit(account, delta)` updates the balance of the appropriate account in memory, but does *not* write the new balance to cell storage. It also appends to the log the record:
  ⟨UPDATE, *transactionid*, *account*, *delta*, *oldbalance*, *newbalance*⟩
- `commit` first appends to the log ⟨OUTCOME COMMITTED, *transactionid*⟩. It then writes out to cell storage any account balances modified by the transaction. When this is done, it appends to the log ⟨END, *transactionid*⟩

Ben knows that this protocol can support all-or-nothing atomicity across system crashes. He sketches a recovery procedure. After a crash, the system will scan the log and *redo* the effects of any transaction that logged an OUTCOME COMMITTED record but not an END record. But rather than implementing the recovery procedure, he puts the system into production immediately. After all, Worse Is Better, and he can always write that recovery procedure later...

Inevitably, Ben's system crashes before he gets around to writing that recovery procedure. The bank's customers are not amused. Panicked, Ben turns to you to help him recover the state of his database.

The cell storage of the database at the time of the crash, and the last few entries of the log appear below. All earlier entries in the log correspond to transactions known to have completed in their entirety.

Cell Storage

AccountID	Balance
account1	$890
account2	$648
account3	$32
account4	$1500

Log

```
⟨BEGIN, T1⟩
⟨UPDATE, T1, account2, 200, 448, 648⟩
⟨OUTCOME COMMITTED, T1⟩
⟨END, T1⟩
⟨BEGIN, T2⟩
⟨BEGIN, T3⟩
⟨UPDATE, T3, account1, 500, 390, 890⟩
⟨UPDATE, T3, account4, −500, 1500, 1000⟩
⟨UPDATE, T2, account3, 100, 32, 132⟩
⟨OUTCOME COMMITTED, T3⟩
```

10. [10 points]: Fill in the values that will appear in the cell storage once recovery is completed using Ben's intended recovery scheme:

AccountID	Balance
account1	$890
account2	$648
account3	$32
account4	$1000

Answer: $T1$ has logged both a OUTCOME record and an END record, so its changes have already been installed into cell storage and the recovery procedure can ignore it.

$T3$ has logged an OUTCOME record but not an END record, so it is a *winner* and needs to be redone. At the time of the crash, its update to *account*1 had already been installed to cell storage, but its update to *account*4 had not. So the recovery procedure will change account4's balance to $1000.

$T2$ has not logged an OUTCOME record, so it had not reached the commit point at the time of the crash. It is deemed a *loser*, and the recovery procedure should not redo its changes. *account*3's balance remains unchanged.

11. [10 points]: Ben is worried that the write-ahead log will take up too much space. Given Ben's recovery procedure, which of the following options would reduce the size of the log, while still being able to persistently store data, and recover from crashes with all-or-nothing atomicity?

(Circle True or False for each choice.)

A. True / False eliminating the BEGIN record.

Answer: True. The BEGIN record is redundant. The existence of a transaction can be inferred from the first UPDATE record for that transaction.

B. True / False removing both the *oldbalance* and *delta* fields from the UPDATE record

Answer: True. As long as the *newbalance* field is present, we can redo changes made by committed transactions. Ben's recovery procedure does not require undoing changes.

C. True / False removing both the *oldbalance* and *newbalance* fields from the UPDATE record

Answer: False. The *delta* field alone is not enough to redo the effects of committed transactions, because adjusting the account balance by *delta* is not an idempotent operation. A transaction might have committed but only written part of its changes to cell storage at the time of the crash (like $T3$ from the previous question). Without either the old or new balance, the recovery procedure has no way to know whether it needs to redo the change or not.

D. True / False periodically writing a CHECKPOINT record to the log which contains the ID of each uncommitted transaction, then discarding all log entries before the CHECKPOINT record.

Answer: False. Consider a transaction that begins before the checkpoint, and commits after the checkpoint. If the system crashes between when that transaction logs its OUTCOME and END records, the recovery procedure will need to redo its updates. But any UPDATE records made before the checkpoint will have been discarded.

End of Quiz II

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2010

Quiz II

There are 13 questions and 11 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below. Write your initials at the bottom of each page.

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
You may use a computer to look at PDFs and notes but not for any other purpose.

CIRCLE your recitation section number:

10:00	1. Lampson/Kushman	
11:00	2. Jones/Rieb	3. Rudolph/Kushman
12:00		4. Rudolph/Rieb
1:00	5. Gifford/Post	6. Jones/Spicer
2:00	7. Gifford/Spicer	8. Lampson/Post

Do not write in the boxes below

1-6 (xx/36)	7-9 (xx/28)	10-11 (xx/20)	12-13 (xx/16)	Total (xx/100)

Name:

I Reading Questions

1. [8 points]: Based on the description of TCP in the paper "TCP Congestion Control with a Misbehaving Receiver" by Savage *et al.* mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False A TCP transmitter normally interprets three duplicate ACKs to mean that, while data packets have been received out of order, all data is successfully being delivered.

B. True / False Acknowledging every TCP packet with a separate ACK is useful because it keeps the transmitters' congestion window size constant.

C. True / False A TCP transmitter avoids wasting network bandwidth by only resending a packet when it has not received an ACK that encompasses the packet within a time-out interval.

D. True / False Transmitter-randomized TCP segment boundaries would allow a receiver to optimistically ACK data to improve performance.

2. [6 points]: Given the context of the paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)" by Patterson *et al.*, mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False RAID 1 gets slower as you add more drives because for every read one must wait for the slowest disk in a group to respond.

B. True / False In RAID 5 parity sectors are rotated across disks, and thus independent writes are less likely to require operations on the same physical devices in a group.

C. True / False Sector interleaving means that byte i of a file is written to disk drive $(i + k) \bmod N$, where N is the number of drives in the array, and k is a constant.

3. [8 points]: With respect to the description of the BitTyrant protocol in the paper "Do incentives build robustness in BitTorrent?" by Piatek *et al.* mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False A client connecting to a peer for the first time is allowed to download data until the peer has had a chance to determine if the client is reciprocating.

B. True / False When a node "reciprocates", it provides data to a peer at a rate that is the same or higher than than the rate at which it is receiving data from that peer.

C. True / False The swarm is robust to tracker failures: if the tracker fails and restarts, existing downloads will eventually complete.

D. True / False One difference between the BitTyrant system and the reference implementation of BitTorrent is that a node in BitTyrant may attempt to upload at different rates to different peers.

4. [6 points]: Based on the description of the experimental Ethernet system in the 1976 paper by Metcalfe and Boggs (reading #9) mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False If there are 20 hosts sending traffic as fast as they can and all the packets sent are the same size, the fraction of the time that the network is successfully sending packet data does not depend on the packet size.

B. True / False If t is the maximum time in microseconds for a packet to get from one host to another, r is the number of bytes transmitted per microsecond, and p is the length of the packet preamble in bytes, then the minimum number of data bytes that must be in a packet for the Ethernet to work properly is $2 * r * t - p$.

C. True / False Upcoming versions of Ethernet, which will run at 100 gigabits/second, could use the same MAC protocol and minimum packet size as the experimental Ethernet.

5. [6 points]: Based on the description of the Internet Border Gateway Protocol (BGP) in the paper on wide-area routing (reading #12), mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False A packet always needs to pass through a tier 1 router on its way from source to destination.

B. True / False A tier 1 router needs a table with a separate entry for each host in the Internet that tells it how to route to that host.

C. True / False A router sends a packet on the shortest path to its destination.

6. [2 points]: Answer this question with reference to reading #15, "How to Build a File Synchronizer," by Jim *et al.*. Alice and Bob each have a computer, and each of them has a directory called Papers, which they synchronize with Unison. Since the last time Unison was run, Alice and Bob both add the same paper to their directories. Alice loves the paper and annotates it with many praises. Bob hates it and not only deletes this paper but also accidently deletes his Unison archive. Alice runs Unison. What happens? (Circle the best answer.)

A. Unison causes the file to re-appear in Bob's directory.

B. Unison deletes the file from Alice's directory.

C. Unison leaves the directories with different contents.

II Link-level flow control

David DoGood is troubled by the fact that Internet routers intentionally discard packets when they receive more input than they can forward. He designs a new kind of router and inter-router link designed to never discard a packet due to queue overflow. David's design adds one new wire in each direction on each link that conveys a "flow-control" signal from the router that is receiving packets from a link to the router that is sending packets on the link. A receiving router can *assert* or *deassert* the signal on a link to control whether the sender at the other end of the link sends packets: when a sender sees an asserted flow-control signal on a link, it is not allowed to send packets on that link; when it sees a de-asserted signal on a link, it is allowed to send packets on that link.

Here's how David's routers use the flow-control signals. Each router has a separate queue of packets for each outgoing link, containing packets waiting to be sent on that link. A router only sends packets on a link if that link's incoming flow-control signal is deasserted. Each router has two fixed parameters T_1 and T_2, which you can think of as the flow-control threshold and the maximum queue length, respectively. A router asserts the flow-control signal on all links when any queue in the router contains T_1 or more packets; the router de-asserts all flow-control signals when all its queues have fewer than T_1 packets. If a packet arrives and the queue of the link that the packet should be sent out on has T_2 or more packets, the router discards the packet (hopefully this never happens). T_2 should be greater than T_1. Every router in a network uses the same T_1 and T_2 values, set by the network manager.

You should assume that a packet never enters and leaves a router by the same link, and that the routers' CPUs and memories are infinitely fast.

7. [8 points]: Suppose router R2 has two links, one to R1 and another to R3.

```
--R1---R2---R3--
```

The link between R2 and R3 has capacity 1 packet/second (that's very slow!). All other links have capacity 1000 packets/second. All links have a one-way speed-of-light delay of 0.1 seconds.

T_1 is set to 1000. What is the minimum value for T_2 that will ensure that R2 need never discard a packet from R1 due to a queue exceeding length T_2? (Circle the answer that is closest to the correct value.)

A. 1000

B. 1100

C. 1200

D. 2000

E. 2100

Initials:

8. [10 points]: In the following topology, all links have capacity 1000 packets/second except the link between R2 and R3, which has capacity 100 packets/second.

```
--R1---R2---R3--
```

All routers have T_1 set to 2 and T_2 set to 1000. All links have one-way speed-of-light delays of 0.1 seconds. There is one long-running file transfer flowing through R1, R2, and R3. There is no other traffic in the network. David's flow-control scheme will limit the rate at which packets flow over the link from R2 to R3 to be no more than a certain rate. What is that rate? (Circle the answer that is closest to the correct value.)

A. 10 packets/second

B. 20 packets/second

C. 50 packets/second

D. 90 packets/second

E. 100 packets/second

9. [10 points]: Consider this configuration:

```
    R2
    |
R1--R3--R4
     \  |
      \ |
       \|
        R5--R6
        |
        R7
```

There are two long-running file transfers, one along path R1-R3-R4-R5-R7, and the other along path R6-R5-R3-R2. The first transfer's direction is from R1 to R7, the second transfer's direction is from R6 to R2. Both transfers have plenty of data to send, and there is no other traffic in the network. All links have capacity 1000 packets/second and speed-of-light delays of 0.1 seconds. T_1 is 100 and T_2 is 1000. Imagine that for a few seconds R2 asserts the flow control signal to R3, and R7 asserts the flow control signal to R5, so that both transfers pause. At the end of this period all the relevant output queues contain more than T_1 packets (i.e. the queues feeding links R1-R3, R3-R4, R4-R5, R5-R7, R6-R5, R5-R3, and R3-R2). At some point both R2 and R7 simultaneously de-assert their flow-control signals. How long will it take before R6 sees a deasserted flow-control signal from R5? (Circle the answer that is closest to the correct value.)

A. 0.1 seconds

B. 0.3 seconds

C. 0.6 seconds

D. 0.8 seconds

E. Never

III Reliability and Atomicity

10. [8 points]:

Suppose you have a computer with a single hard drive. The expected lifetime of the drive is about five years, and it takes 10 hours to swap out a failed drive and restore from a tape backup (during which time the system is unavailable).

Now suppose you add a second drive (of the same model) to your computer and replicate data across the drives. For the following two replication schemes, indicate whether it will improve, decrease, or keep availability of the disk subsystem about the same versus the single-drive system. Also indicate whether it will increase, decrease, or not change the time to perform a write of a single block to a file.

A. Mirror every write on both hard drives. When a drive fails, perform all reads and writes to the other drive, and repair the failed drive by copying contents from the other drive. Assume this copying takes 10 hours, and the other drive can service application reads and writes during this time.

(Circle one availability and one write time option.)

Availability: Improves Decreases Stays the Same

Single block write time: Increases Decreases Stays the Same

B. Interleave file system blocks between the two hard drives (e.g., place blocks 1, 3, 5, and 7, . . . on Disk 1 and blocks 2, 4, 6, and 8, . . . on Disk 2.) When a hard drive fails, replace it with a spare and recover its contents from a backup (taking 10 hours).

(Circle one availability and one write time option.)

Availability: Improves Decreases Stays the Same

Single block write time: Increases Decreases Stays the Same

Initials:

11. [12 points]: For each of the following transaction schedules, indicate whether it could be generated by 2-phase locking (where read and write locks on an object can be released immediately after the lock point and the last access to the object) and if it could be, what an equivalent serial order is (e.g., T1, T2). Suppose there are three data items, A, B, and C, and the schedules record BEGIN, COMMIT, ABORT, READ, and WRITE operations. The value of any item that is written may depend on previously read values. Assume that a transaction that performs a READ operation on a data item has already acquired a shared lock on that item, and that a transaction that performs a WRITE operation has already acquired an exclusive lock on that item.

A.
```
T1 BEGIN
     T2 BEGIN
T1 READ A
     T2 READ A
T1 WRITE B
     T2 WRITE C
T1 COMMIT
     T2 COMMIT
```

Possible under two phase locking? Yes No

Serial Equivalent Order:____________

B.
```
T1 BEGIN
     T2 BEGIN
T1 READ A
     T2 READ B
T1 WRITE A
     T2 WRITE A
T1 COMMIT
     T2 COMMIT
```

Possible under two phase locking? Yes No

Serial Equivalent Order:____________

C.

```
T1 BEGIN
     T2 BEGIN
          T3 BEGIN
T1 READ A
     T2 READ A
          T3 READ A
T1 READ B
     T2 WRITE C
          T3 WRITE B
T1 COMMIT
     T2 WRITE A
          T3 WRITE A
     T2 WRITE B
          T3 READ B
     T2 COMMIT
          T3 COMMIT
```

Possible under two phase locking? Yes No

Serial Equivalent Order:_____________

IV Logging

Suppose you are running a system that uses logging and two phase locking where **all locks are held until after a transaction commits** (this is different than in the previous question). Log records are immediately written to disk when a write occurs. In addition to a log, the system maintains a cell store, though it buffers writes to the cell store in memory (so the cell store may not immediately reflect all of the operations in the log.) Writes to the cell store may happen before a transaction commits. The log may contain transaction BEGIN, UPDATE, COMMIT, and ABORT records.

After running a few transactions, the system crashes, and unfortunately, the log file is garbled such that some of the log records cannot be read. The system contains three objects, A, B, and C, each containing an integer value. Before the transactions were run all objects had value 0. After the crash the cell storage on disk records A = 50, B = 90, and C = 100.

Here is the corrupted log on disk (assume that there are no additional missing records besides those labeled with ?).

	Transaction ID	Operation	Before Value	After Value
	1	BEGIN		
	1	UPDATE A	0	75
	2	BEGIN		
	2	UPDATE C	0	100
	1	UPDATE B	0	125
Q1:	?	?	?	?
	2	UPDATE A	0	50
	3	BEGIN		
	3	UPDATE B	?	?
	3	COMMIT		
Q2:	2	UPDATE B	?	75
	2	COMMIT		

12. [8 points]: What must the log record labeled Q1: have contained?

13. [8 points]: Which of the following are possible values for the Before value of B in the log line labeled Q2:? (Circle all that apply)

A. 0

B. 90

C. 100

D. 125

End of Quiz II

Please double check that you wrote your name on the front of the quiz,
and circled your recitation section number.

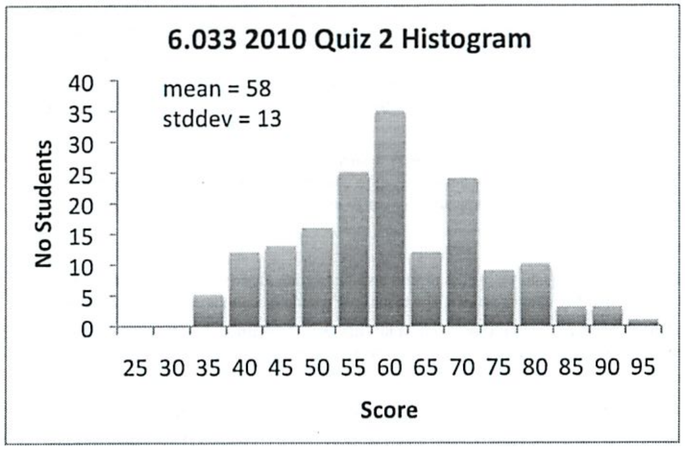
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2010

Quiz II Solutions

Grade distribution:



I Reading Questions

1. [8 points]: Based on the description of TCP in the paper "TCP Congestion Control with a Misbehaving Receiver" by Savage *et al.* mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False A TCP transmitter normally interprets three duplicate ACKs to mean that, while data packets have been received out of order, all data is successfully being delivered.

Answer: False. A TCP transmitter interprets three duplicate ACKs to mean that the network dropped a packet, and that the transmitter should re-send it.

B. True / False Acknowledging every TCP packet with a separate ACK is useful because it keeps the transmitters' congestion window size constant.

Answer: False. The transmitter's congestion window usually grows with each received ACK.

C. True / False A TCP transmitter avoids wasting network bandwidth by only resending a packet when it has not received an ACK that encompasses the packet within a time-out interval.

Answer: False. The transmitter also re-sends when it receives three duplicate ACKs.

D. True / False Transmitter-randomized TCP segment boundaries would allow a receiver to optimistically ACK data to improve performance.

Answer: False. Randomizing the segment boundaries makes it harder for a receiver to optimistically ACK.

2. [6 points]: Given the context of the paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)" by Patterson *et al.*, mark each of the following statements true or false.

(Circle True or False for each choice.)

A. True / False RAID 1 gets slower as you add more drives because for every read one must wait for the slowest disk in a group to respond.

Answer: False. RAID 1 can read any data from any disk, so it is not forced to wait for the slowest disk.

B. True / False In RAID 5 parity sectors are rotated across disks, and thus independent writes are less likely to require operations on the same physical devices in a group.

Answer: True.

C. True / False Sector interleaving means that byte i of a file is written to disk drive $(i + k) \bmod N$, where N is the number of drives in the array, and k is a constant.

Answer: False. The scheme described is byte interleaving.

3. [8 points]: With respect to the description of the BitTyrant protocol in the paper "Do incentives build robustness in BitTorrent?" by Piatek *et al.* mark each of the following statements true or false.
(Circle True or False for each choice.)

A. True / False A client connecting to a peer for the first time is allowed to download data until the peer has had a chance to determine if the client is reciprocating.

Answer: False. It must wait to be "optimistically unchoked" by some peer.

B. True / False When a node "reciprocates", it provides data to a peer at a rate that is the same or higher than than the rate at which it is receiving data from that peer.

Answer: False. Reciprocation can be at a lower rate.

C. True / False The swarm is robust to tracker failures: if the tracker fails and restarts, existing downloads will eventually complete.

Answer: True. Nodes will reconnect to the tracker after it restarts and eventually complete their download.

D. True / False One difference between the BitTyrant system and the reference implementation of BitTorrent is that a node in BitTyrant may attempt to upload at different rates to different peers.

Answer: True. BitTyrant tries to upload at a rate that is just fast enough to get the peer to reciprocate, whereas the reference BitTorrent implementation tries to send at an equal data rate to all peers.

4. [6 points]: Based on the description of the experimental Ethernet system in the 1976 paper by Metcalfe and Boggs (reading #9) mark each of the following statements true or false.
(Circle True or False for each choice.)

A. True / False If there are 20 hosts sending traffic as fast as they can and all the packets sent are the same size, the fraction of the time that the network is successfully sending packet data does not depend on the packet size.

Answer: False. Shorter packets have worse utilization because the cost of acquiring the ether is fixed, and a longer packet keeps it for longer.

B. True / False If t is the maximum time in microseconds for a packet to get from one host to another, r is the number of bytes transmitted per microsecond, and p is the length of the packet preamble in bytes, then the minimum number of data bytes that must be in a packet for the Ethernet to work properly is $2 * r * t - p$.

Answer: True. The minimum packet must be at least that long to ensure that the sender detects a collision before it is done sending.

C. True / False Upcoming versions of Ethernet, which will run at 100 gigabits/second, could use the same MAC protocol and minimum packet size as the experimental Ethernet.

Answer: False. The maximum network diameter would have to be tiny in order to allow the minimum packet size to be unchanged.

5. [6 points]: Based on the description of the Internet Border Gateway Protocol (BGP) in the paper on wide-area routing (reading #12), mark each of the following statements true or false.
(Circle True or False for each choice.)

A. True / False A packet always needs to pass through a tier 1 router on its way from source to destination.

Answer: False. If source and destination have a common provider below tier 1, the packet will only go through that provider. If each has a provider below tier 1 and there is a peering path that connects these providers, the packet will take that path.

B. True / False A tier 1 router needs a table with a separate entry for each host in the Internet that tells it how to route to that host.

Answer: False. A tier 1 provider needs to know how to route to every endpoint, but lots of endpoint IP addresses are aggregated into a set of IP addresses with the same prefix, and a tier 1 provider only needs one entry for each prefix.

C. True / False A router sends a packet on the shortest path to its destination.

Answer: False. BGP routers are typically configured to prefer to send each packet on a peer path, rather than on a path to a provider, because that is cheaper. In fact, a BGP router may not even know the shortest path to the destination, because that path may go through one of its customers who didnt advertise the path in order to avoid paying the provider.

6. [2 points]: Answer this question with reference to reading #15, "How to Build a File Synchronizer," by Jim *et al.*. Alice and Bob each have a computer, and each of them has a directory called Papers, which they synchronize with Unison. Since the last time Unison was run, Alice and Bob both add the same paper to their directories. Alice loves the paper and annotates it with many praises. Bob hates it and not only deletes this paper but also accidently deletes his Unison archive. Alice runs Unison. What happens? (Circle the best answer.)

A. Unison causes the file to re-appear in Bob's directory.

B. Unison deletes the file from Alice's directory.

C. Unison leaves the directories with different contents.

Answer: A. Bob's Unison has no idea the paper ever existed (both because Bob deleted it before running Unison again, and because Bob deleted his archive), so Bob's unison will treat the paper as a new file created by Alice, and copy it from Alice's computer.

II Link-level flow control

David DoGood is troubled by the fact that Internet routers intentionally discard packets when they receive more input than they can forward. He designs a new kind of router and inter-router link designed to never discard a packet due to queue overflow. David's design adds one new wire in each direction on each link that conveys a "flow-control" signal from the router that is receiving packets from a link to the router that is sending packets on the link. A receiving router can *assert* or *deassert* the signal on a link to control whether the sender at the other end of the link sends packets: when a sender sees an asserted flow-control signal on a link, it is not allowed to send packets on that link; when it sees a de-asserted signal on a link, it is allowed to send packets on that link.

Here's how David's routers use the flow-control signals. Each router has a separate queue of packets for each outgoing link, containing packets waiting to be sent on that link. A router only sends packets on a link if that link's incoming flow-control signal is deasserted. Each router has two fixed parameters T_1 and T_2, which you can think of as the flow-control threshold and the maximum queue length, respectively. A router asserts the flow-control signal on all links when any queue in the router contains T_1 or more packets; the router de-asserts all flow-control signals when all its queues have fewer than T_1 packets. If a packet arrives and the queue of the link that the packet should be sent out on has T_2 or more packets, the router discards the packet (hopefully this never happens). T_2 should be greater than T_1. Every router in a network uses the same T_1 and T_2 values, set by the network manager.

You should assume that a packet never enters and leaves a router by the same link, and that the routers' CPUs and memories are infinitely fast.

7. [8 points]: Suppose router R2 has two links, one to R1 and another to R3.

```
--R1---R2---R3--
```

The link between R2 and R3 has capacity 1 packet/second (that's very slow!). All other links have capacity 1000 packets/second. All links have a one-way speed-of-light delay of 0.1 seconds.

T_1 is set to 1000. What is the minimum value for T_2 that will ensure that R2 need never discard a packet from R1 due to a queue exceeding length T_2? (Circle the answer that is closest to the correct value.)

A. 1000

B. 1100

C. 1200

D. 2000

E. 2100

Answer: 1200. It takes 0.1 seconds for R2's flow-control signal to reach R1, and another 0.1 seconds for the last packet R1 sends to reach R2. Thus R2 will receive 200 packets from R1 after R2 asserts the flow-control signal, and R2 must be prepared to queue a total of 1200 packets.

8. [10 points]: In the following topology, all links have capacity 1000 packets/second except the link between R2 and R3, which has capacity 100 packets/second.

```
--R1---R2---R3--
```

All routers have T_1 set to 2 and T_2 set to 1000. All links have one-way speed-of-light delays of 0.1 seconds. There is one long-running file transfer flowing through R1, R2, and R3. There is no other traffic in the network. David's flow-control scheme will limit the rate at which packets flow over the link from R2 to R3 to be no more than a certain rate. What is that rate? (Circle the answer that is closest to the correct value.)

A. 10 packets/second

B. 20 packets/second

C. 50 packets/second

D. 90 packets/second

E. 100 packets/second

Answer: 90. R2 will repeat the following cycle. When R2's queue length falls to 2, it will de-assert the flow-control signal to R1. 0.2 seconds later, R2 will start receiving packets from R1. R1 will also almost instantly assert the flow-control signal back to R1, so R2 will receive a total of 200 packets from R1. It takes R2 two seconds to send these 200 packets to R3, after which R2 will again de-assert the flow-control signal to R1. The total cycle takes 2.2 seconds, during which time R2 sends 200 packets; $200/2.2 = 90.9$.

9. [10 points]: Consider this configuration:

```
    R2
    |
R1--R3--R4
     \  |
      \ |
       \|
        R5--R6
        |
        R7
```

There are two long-running file transfers, one along path R1-R3-R4-R5-R7, and the other along path R6-R5-R3-R2. The first transfer's direction is from R1 to R7, the second transfer's direction is from R6 to R2. Both transfers have plenty of data to send, and there is no other traffic in the network. All links have capacity 1000 packets/second and speed-of-light delays of 0.1 seconds. T_1 is 100 and T_2 is 1000. Imagine that for a few seconds R2 asserts the flow control signal to R3, and R7 asserts the flow control signal to R5, so that both transfers pause. At the end of this period all the relevant output queues contain more than T_1 packets (i.e. the queues feeding links R1-R3, R3-R4, R4-R5, R5-R7, R6-R5, R5-R3, and R3-R2). At some point both R2 and R7 simultaneously de-assert their flow-control signals. How long will it take before R6 sees a deasserted flow-control signal from R5? (Circle the answer that is closest to the correct value.)

A. 0.1 seconds

B. 0.3 seconds

C. 0.6 seconds

D. 0.8 seconds

E. Never

Answer: Never. R5 will only de-assert flow-control to R4 when R3's queue to R3 drains; R3 will only de-assert flow-control to R5 when R3's queue to R4 drains; and R4 will only de-assert flow-control to R3 when R4's queue to R5 drains. However, none of those queues will drain, because all three flow-control signals are asserted.

III Reliability and Atomicity

10. [8 points]:

Suppose you have a computer with a single hard drive. The expected lifetime of the drive is about five years, and it takes 10 hours to swap out a failed drive and restore from a tape backup (during which time the system is unavailable).

Now suppose you add a second drive (of the same model) to your computer and replicate data across the drives. For the following two replication schemes, indicate whether it will improve, decrease, or keep availability of the disk subsystem about the same versus the single-drive system. Also indicate whether it will increase, decrease, or not change the time to perform a write of a single block to a file.

A. Mirror every write on both hard drives. When a drive fails, perform all reads and writes to the other drive, and repair the failed drive by copying contents from the other drive. Assume this copying takes 10 hours, and the other drive can service application reads and writes during this time.

(Circle one availability and one write time option.)

Availability: Improves Decreases Stays the Same

Single block write time: Increases Decreases Stays the Same

Answer: Availability improves, because the other drive can be used during the ten hours required to repair a broken drive. Single block write time increases, since each write must wait for the slower (longest rotation) of the two drives.

B. Interleave file system blocks between the two hard drives (e.g., place blocks 1, 3, 5, and 7, ... on Disk 1 and blocks 2, 4, 6, and 8, ... on Disk 2.) When a hard drive fails, replace it with a spare and recover its contents from a backup (taking 10 hours).

(Circle one availability and one write time option.)

Availability: Improves Decreases Stays the Same

Single block write time: Increases Decreases Stays the Same

Answer: Availability decreases, since both drives must work. Single block write time stays the same.

11. [12 points]: For each of the following transaction schedules, indicate whether it could be generated by 2-phase locking (where read and write locks on an object can be released immediately after the lock point and the last access to the object) and if it could be, what an equivalent serial order is (e.g., T1, T2). Suppose there are three data items, A, B, and C, and the schedules record BEGIN, COMMIT, ABORT, READ, and WRITE operations. The value of any item that is written may depend on previously read values. Assume that a transaction that performs a READ operation on a data item has already acquired a shared lock on that item, and that a transaction that performs a WRITE operation has already acquired an exclusive lock on that item.

A.

```
T1 BEGIN
     T2 BEGIN
T1 READ A
     T2 READ A
T1 WRITE B
     T2 WRITE C
T1 COMMIT
     T2 COMMIT
```

Possible under two phase locking? **Answer:** Yes

Serial Equivalent Order: **Answer:** Either T1,T2 or T2,T1

B.

```
T1 BEGIN
     T2 BEGIN
T1 READ A
     T2 READ B
T1 WRITE A
     T2 WRITE A
T1 COMMIT
     T2 COMMIT
```

Possible under two phase locking? **Answer:** Yes

Serial Equivalent Order: **Answer:** T1,T2

C.

```
T1 BEGIN
     T2 BEGIN
          T3 BEGIN
T1 READ A
     T2 READ A
          T3 READ A
T1 READ B
     T2 WRITE C
          T3 WRITE B
T1 COMMIT
     T2 WRITE A
          T3 WRITE A
     T2 WRITE B
          T3 READ B
     T2 COMMIT
          T3 COMMIT
```

Possible under two phase locking? **Answer:** No, because T2 and T3 would deadlock rather than both getting write-locks on A.

Serial Equivalent Order: **Answer:** Doesn't exist.

IV Logging

Suppose you are running a system that uses logging and two phase locking where **all locks are held until after a transaction commits** (this is different than in the previous question). Log records are immediately written to disk when a write occurs. In addition to a log, the system maintains a cell store, though it buffers writes to the cell store in memory (so the cell store may not immediately reflect all of the operations in the log.) Writes to the cell store may happen before a transaction commits. The log may contain transaction BEGIN, UPDATE, COMMIT, and ABORT records.

After running a few transactions, the system crashes, and unfortunately, the log file is garbled such that some of the log records cannot be read. The system contains three objects, A, B, and C, each containing an integer value. Before the transactions were run all objects had value 0. After the crash the cell storage on disk records A = 50, B = 90, and C = 100.

Here is the corrupted log on disk (assume that there are no additional missing records besides those labeled with ?).

	Transaction ID	Operation	Before Value	After Value
	1	BEGIN		
	1	UPDATE A	0	75
	2	BEGIN		
	2	UPDATE C	0	100
	1	UPDATE B	0	125
Q1:	?	?	?	?
	2	UPDATE A	0	50
	3	BEGIN		
	3	UPDATE B	?	?
	3	COMMIT		
Q2:	2	UPDATE B	?	75
	2	COMMIT		

12. [8 points]: What must the log record labeled Q1: have contained?

Answer: Abort transaction 1

13. [8 points]: Which of the following are possible values for the Before value of B in the log line labeled Q2:? (Circle all that apply)

A. 0

B. 90

C. 100

D. 125

Answer: 90. The fact that the on-disk cell-storage value of B after the crash was 90 means that some log record must have UPDATEd B with an After Value of 90. That record must be the corrupted UPDATE B between Q1 and Q2. Thus the before value for Q2 must be 90.

End of Quiz II

Please double check that you wrote your name on the front of the quiz,
and circled your recitation section number.

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2012

Quiz 2

There are 12 questions and 9 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

For true/false questions, you will receive 0 points for no answer, and negative points for an incorrect answer. Do not guess; if you are unsure about your answer, consult your notes. We will round up the score for every *numbered* question to 0 if it's otherwise negative (i.e., you cannot get less than 0 on a numbered question).

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below. Write your initials at the bottom of each page.

THIS IS AN OPEN BOOK, OPEN NOTES, OPEN LAPTOP QUIZ, BUT DON'T USE YOUR LAPTOP FOR COMMUNICATION WITH OTHERS.

CIRCLE your recitation section number:

10:00	[✓ 1. Rudolph/Grusecki]		
11:00	2. Rudolph/Grusecki	3. Abelson/Gokce	4. Katabi/Joshi
12:00		5. Abelson/Gokce	6. Katabi/Joshi
1:00	7. Shavit/Moll	8. Szolovits/Fang	
2:00	9. Shavit/Moll	10. Szolovits/Fang	

Do not write in the boxes below

1-3 (xx/26)	4-7 (xx/26)	8 (xx/12)	9 (xx/18)	10-12 (xx/18)	Total (xx/100)
10 KJ	18 16	0 KF	12 FL	12	52 46

avg 59
st dev 16.

Name: Michael Plasmeier

I Reading Questions

1. [10 points]: Answer the following question based on the paper "Do incentives build robustness in BitTorrent?". The main feature that distinguishes BitTyrant from BitTorrent is:

(Circle the BEST answer)

A. Data transfers with BitTyrant clients are all transactions. WTF

B. BitTyrant data transfers can be rolled back. WTF

C. BitTyrant clients find their peers with the use of distributed hash tables.

[✓ D.] BitTyrant clients choose their peers in a way that leads them towards downloading more information than they upload.

E. BitTyrant's algorithms model altruism by using cumulative distribution functions (CDF).

✓

easy

2. [8 points]: Choose the correct statement about System R's recovery process, based on the paper you read in recitation.

Check

undo/redo logging

(Circle the BEST answer)

A. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is an undo of the parts that happened before the last checkpoint.

B. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is a redo of the parts that happened before the last checkpoint.

C. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is an undo of the parts that happened after the last checkpoint.

/

[✓ D.] During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and a complete undo of all uncommitted transactions.

backwards

1. undo the losers

but other way around

X A

2. forwards redo the winners

Initials: MEP

Check

3. [8 points]: In the LFS system, as described in the paper that you read in recitation, what is the commit point, meaning the point at which a single data block overwritten in a file will be available after a crash, assuming that there are no software bugs and the contents of the disk survive the crash?

(Circle the BEST answer)

A. The application's `write()` system call returns.

B. The application calls `close()` on the file.

[✓ **C.**] The file's data block is flushed to the log. ← log 1st ?

Check

D. The file's updated inode is flushed to the log.

E. The segment summary for the segment containing the file's data and updated inode is flushed to the log.

F. The checkpoint region is updated to include the location of the updated inode.

? from lecture - not LFS paper ?

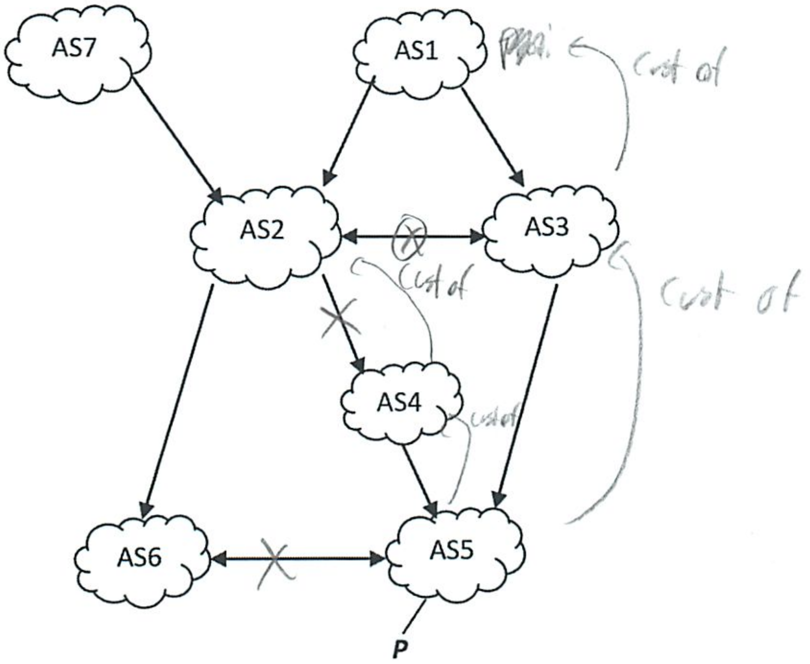
lecture i y dirent to new info
- wrong failure pt

X E

Initials: MEP

II BGP

Consider the topology below, where arrows point from provider to customer and bi-directional arrows refer to a peering links. Assume that the ASes follow the import and export rules described in the required reading titled "Wide-Area Internet Routing". The figure shows that the address prefix *P* belongs to AS 5.



makes sens

Answer the questions on the next page.

Initials: MEP

4. [5 points]: Which AS path does AS1 follow to reach prefix P in AS5? You only need to list the ASes on the path in order, starting with AS1 and ending with AS5.

AS1 → AS 3 → AS5 as5

5. [5 points]: How many paths does AS2 learn for prefix P? Which of these AS paths does it use to deliver packets to prefix P? Again, write the path as an ordered list of ASes.

providers cost

Number of paths: 3

AS2 → AS6 AS4 ← uses AS3 backup → AS5

Since wants to send data to its customers

6. [8 points]: Say that the link between AS2 and AS4 and the link between AS6 and AS5 both fail. Would any of the ASes in the figure lose the ability to reach prefix P (i.e., will have no routes to P even after the routing stabilizes)? If "yes", tell us which AS or ASes?

No, AS6 would use AS2 as a provider and would pay for that service

7. [8 points]: Say that the topology is as in the Figure except that the link between AS2 and AS3 fails (this is the only link that fails). Let the routes stabilize after the failure. Does any of the ASes change the route it uses to reach prefix P before and after the failure? If "yes", which AS, and what is the new route?

No. AS1 would know of the change and sends to AS 3 assuming = local prefs

Initials: MEP

should read closer

III Fault-tolerance

RAID?

0

8. [12 points]: Alyssa P. Hacker stores her data on a fault-tolerant storage system, which has 5 1 TB disks that together provide 4 TB of aggregate space, and can tolerate any one (but no more) disk failure. Each disk has a MTTF of 10^6 hours, and you can assume that all disk failures are independent. When a disk fails, Alyssa orders a replacement from Amazon which arrives in approximately 4 days (100 hours). Once Alyssa plugs in the replacement disk, assume that its content is reconstructed instantaneously. What is the MTTF for Alyssa's storage system? yeah

(Circle the BEST answer)

└complete failure?

non-recoverable?

A. 10^6 hours

B. 4×10^7 hours

check

C. 5×10^8 hours

D. 2×10^9 hours

[✓ **E.** 10^{10} hours]

?but prob fail is on the 4 blocks

prob 2 failures within 4 days

¿what was

$$\text{availability} = \frac{MTTF}{MTTF + MTTR} \leftarrow = ~~MTBF~~$$

$$\text{failure rate} = \frac{\text{failures}}{hr} = \longrightarrow \frac{1}{10^6}$$

$$MTTF = \frac{1}{\text{failure rate}} = \frac{hr}{\text{failures}} = \frac{10^6}{1}$$

$$MTTF + MTTR = MTBF$$

$P(\frac{1}{10^6})$ each hr it will fail

$$P\left(\frac{1}{10^6} \,\middle|\, \frac{100}{10^6}\right) = \frac{P(A \cap B)}{P(B)}$$

?B ?given ?A

¿wrong to multiply?

$$\frac{\frac{1}{10^6} \cdot \frac{100}{10^6}}{\frac{1}{10^6}} = \frac{100}{10^6}$$

$P(\frac{1}{10^6}$ happens in 10^6 hrs$)$

10^{10}

should be higher

Initials: MEP

$\frac{100}{10^6}$ - will fail in any of the 100 hrs before

so $\frac{1}{10^6}$ that another failure will happen

IV Isolation

0

9. [18 points]: Alyssa P. Hacker switches her transactional system to a weaker isolation mode, where one transaction that's still running can observe the effects of any other committed transaction, even if that committed transaction started later. (In Postgres, this is called `READ COMMITTED`.) Alyssa runs two transactions, `xfer(A, B, 10)` and `audit(50)`, whose code is below, both of which eventually commit but can run concurrently:

```
xfer(a, b, amt):
  accounts[a].balance = accounts[a].balance - amt
  accounts[b].balance = accounts[b].balance + amt

audit(thresh):
  for a in accounts:
    if a.balance > thresh:
      sum += a.balance
  return sum
```

So can start any order
and when one finishes can see

Every reference to the `balance` field of an account is a separate read or write. Do not make any assumptions about the order in which `audit` iterates over the accounts. The database starts in the following state:

Account	*Balance*
A	55
B	45

55 45

A -> B $10 - is atomic
Audit ($50) - is not

Which of the following tuples of values (A, B, audit), representing the final state of the database and the return value of the `audit` function, can result after running these two transactions in Alice's weaker isolation mode?

(Circle True or False for each choice.)

12

A. [✓ **True**] / **False** A=45, B=55, audit=0

B. **True** / [✓ **False**] A=45, B=55, audit=45

C. [✓ **True**] / **False** A=45, B=55, audit=55

D. **True** / [✓ **False**] A=45, B=55, audit=90

E. **True** / [✓ **False**] A=45, B=55, audit=100

F. [✓ **True**] / **False** A=45, B=55, audit=110

XFR
audit

55 45
45 55
audit = 55

audit
XFR

55 45
audit = 55
45 55

audit
A
B
XFR

55 45
audit=55
45 55
audit=110

audit
B
A
XFR

55 45
audit=0
45 55
audit=0

Initials: MEP

V Performance

Ben Bitdiddle runs the database for a credit card processing company. Ben's clients are online stores that each have a single program running the following pseudocode to charge customer credit cards:

```
while True:
  ccnum, amount = get_next_order()
  send(ben, {ccnum, amount})
  status = recv(ben)
```

Ben's server looks like the following pseudocode, where write_log() prepares the log records, but only flush_log() writes to disk. Assume that every log flush requires seeking to a new location. Assume that reply() does not block.

```
while True:
  ccnum, amount = recv()
  status = check(ccnum, amount)
  if status == OK:
    write_log(ccnum, amount)
    flush_log()
  reply(status)
```

Must respond before replying

Ben's database must commit each credit card charge operation before responding to the client. Ben uses a typical rotational disk, with a 10 msec average seek time, and 100 MB/sec sequential throughput. Each log record is 512 bytes. The round-trip latency between each client and the server is 100 msec. Assume that the check() function is instantaneous.

Assuming clients have a lot of txns

4

10. [4 points]: How many credit cards can Ben's system successfully charge per second, if Ben has a large number of clients?

- just disk which is just seek - disk band

$\frac{1 sec}{10 milliseconds} = 100$ custs

0

11. [6 points]: How many credit cards can Ben's system successfully charge per second, if Ben has 5 clients, the clients spend almost no time in get_next_order(), and get_next_order() never blocks?

does it matte - matters how many txns

- oh for network latency

- assume again clients have lots of CCs to charge

- txn take 110 ms to client, send 1 every 110 ms

So max 1sec/110 ms = 9.091 txns/sec per cust

9.091·5 = 45,455 txns since Ben's system can fully process them

Initials: MEP

Ben decides that he wants to process more transactions per second, and changes the server code to flush the log after a batch of records have been written to the log:

```
while True:
  for i = 1..N:
    ccnum[i], amount[i] = recv()
  for i = 1..N:
    status[i] = check(ccnum[i], amount[i])
    if status[i] == OK:
      write_log(ccnum[i], amount[i])
  flush_log()
  for i = 1..N:
    reply(status[i])
```

So wait till have N txns

8

12. [8 points]: How many credit cards can Ben's modified system successfully charge per second, if Ben has a large number of clients? Assume the best choice of N.

Depends on N
- don't want too high or clients must wait a while and can't process many txns per second (not optimal for his business, but I will ignore)

N will be very large
L the ~~all~~ sequential ~~time~~ throughput is bound

$$\frac{100MB}{512 \text{ bytes}} = 195313 \text{ Txn/sec}$$

N should be much bigger than this so seek does not matter
Of course this is useless for a biz (see above)
and vunerable to crashes - could say not allowed
- So just prev answer 100 Txn/sec
since he does not log elsewhere

End of Quiz II

Double-check that you wrote your name on the front of the quiz, and circled your recitation.

Perminently (if you want to assume this condition)

Initials: MEP

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

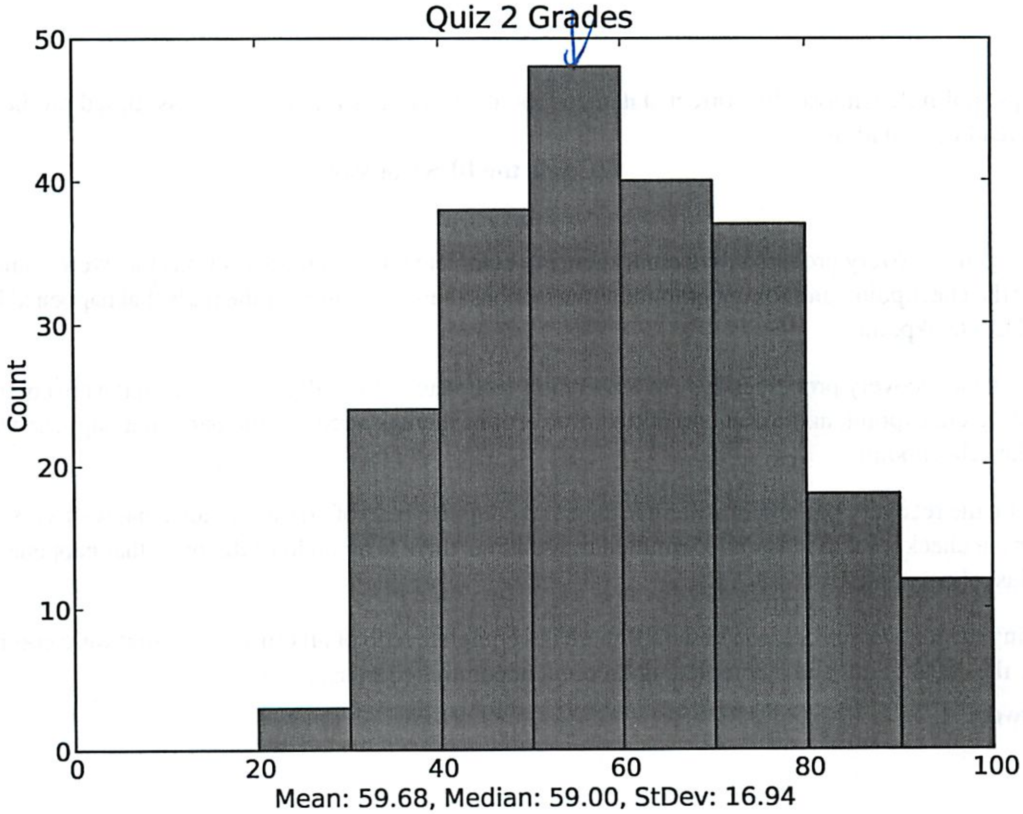
6.033 Computer Systems Engineering: Spring 2012

Quiz 2 Solutions

There are 12 questions and 10 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

52

Grade distribution histogram:



I Reading Questions

1. [10 points]: Answer the following question based on the paper "Do incentives build robustness in BitTorrent?". The main feature that distinguishes BitTyrant from BitTorrent is:

(Circle the BEST answer)

A. Data transfers with BitTyrant clients are all transactions.

B. BitTyrant data transfers can be rolled back.

C. BitTyrant clients find their peers with the use of distributed hash tables.

D. BitTyrant clients choose their peers in a way that leads them towards downloading more information than they upload.

E. BitTyrant's algorithms model altruism by using cumulative distribution functions (CDF).

Answer: D.

2. [8 points]: Choose the correct statement about System R's recovery process, based on the paper you read in recitation.

(Circle the BEST answer)

A. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is an undo of the parts that happened before the last checkpoint.

B. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is a redo of the parts that happened before the last checkpoint.

C. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and for uncommitted transactions there is an undo of the parts that happened after the last checkpoint.

D. During the recovery process in system R there is a complete redo of all transactions that were committed after the checkpoint, and a complete undo of all uncommitted transactions.

Answer: A.

3. [8 points]: In the LFS system, as described in the paper that you read in recitation, what is the commit point, meaning the point at which a single data block overwritten in a file will be available after a crash, assuming that there are no software bugs and the contents of the disk survive the crash?

(Circle the BEST answer)

A. The application's `write()` system call returns.

B. The application calls `close()` on the file.

C. The file's data block is flushed to the log.

D. The file's updated inode is flushed to the log.

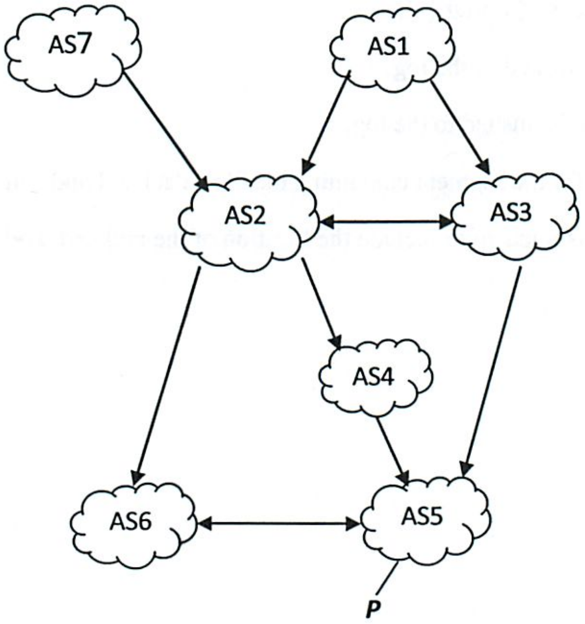
E. The segment summary for the segment containing the file's data and updated inode is flushed to the log.

F. The checkpoint region is updated to include the location of the updated inode.

Answer: E.

II BGP

Consider the topology below, where arrows point from provider to customer and bi-directional arrows refer to a peering links. Assume that the ASes follow the import and export rules described in the required reading titled “Wide-Area Internet Routing”. The figure shows that the address prefix *P* belongs to AS 5.



Answer the questions on the next page.

Answer note: During the quiz many students overlooked the definition of arrows as pointing from provider to customer in the text of the question, and instead assumed it was the opposite. We graded this question under each of the two interpretations, and gave each student the higher of the two scores.

4. [5 points]: Which AS path does AS1 follow to reach prefix P in AS5? You only need to list the ASes on the path in order, starting with AS1 and ending with AS5.

Answer (for both arrow interpretations): AS1 → AS3 → AS5.

5. [5 points]: How many paths does AS2 learn for prefix P? Which of these AS paths does it use to deliver packets to prefix P? Again, write the path as an ordered list of ASes.

Answer (arrows from provider to customer): Number of paths: 3
Answer (arrows from customer to provider): Number of paths: 2

Answer (arrows from provider to customer): AS2 → AS4 → AS5
Answer (arrows from customer to provider): AS2 → AS4 → AS5 is a valid answer, AS2 → AS6 → AS5 as well (not both)

6. [8 points]: Say that the link between AS2 and AS4 and the link between AS6 and AS5 both fail. Would any of the ASes in the figure lose the ability to reach prefix P (i.e., will have no routes to P even after the routing stabilizes)? If "yes", tell us which AS or ASes?

Answer (arrows from provider to customer): Yes. AS7 loses the ability to reach prefix P. All other ASes will have routes.

Answer (arrows from customer to provider): Yes. AS2, AS6, and AS7 all lose connectivity to P.

7. [8 points]: Say that the topology is as in the Figure except that the link between AS2 and AS3 fails (this is the only link that fails). Let the routes stabilize after the failure. Does any of the ASes change the route it uses to reach prefix P before and after the failure? If "yes", which AS, and what is the new route?

Answer (both interpretations): No AS changes routes.

Initials:

III Fault-tolerance

8. [12 points]: Alyssa P. Hacker stores her data on a fault-tolerant storage system, which has 5 1 TB disks that together provide 4 TB of aggregate space, and can tolerate any one (but no more) disk failure. Each disk has a MTTF of 10^6 hours, and you can assume that all disk failures are independent. When a disk fails, Alyssa orders a replacement from Amazon which arrives in approximately 4 days (100 hours). Once Alyssa plugs in the replacement disk, assume that its content is reconstructed instantaneously. What is the MTTF for Alyssa's storage system?

(Circle the BEST answer)

A. 10^6 hours

B. 4×10^7 hours

C. 5×10^8 hours

D. 2×10^9 hours

E. 10^{10} hours

Answer: $\frac{1}{\frac{5}{10^6} \times \frac{4 \cdot 100}{10^6}} = \frac{1}{\frac{20}{10^{10}}} = 5 \times 10^8$. C. You can find similar calculations in the RAID paper.

IV Isolation

9. [18 points]: Alyssa P. Hacker switches her transactional system to a weaker isolation mode, where one transaction that's still running can observe the effects of any other committed transaction, even if that committed transaction started later. (In Postgres, this is called `READ COMMITTED`.) Alyssa runs two transactions, `xfer(A, B, 10)` and `audit(50)`, whose code is below, both of which eventually commit but can run concurrently:

```
xfer(a, b, amt):
  accounts[a].balance = accounts[a].balance - amt
  accounts[b].balance = accounts[b].balance + amt

audit(thresh):
  for a in accounts:
    if a.balance > thresh:
      sum += a.balance
  return sum
```

Every reference to the `balance` field of an account is a separate read or write. Do not make any assumptions about the order in which `audit` iterates over the accounts. The database starts in the following state:

Account	*Balance*
A	55
B	45

Which of the following tuples of values (A, B, audit), representing the final state of the database and the return value of the `audit` function, can result after running these two transactions in Alice's weaker isolation mode?

(Circle True or False for each choice.)

A. True / False A=45, B=55, audit=0

Answer: True. Audit reads B=45, fails if check. Then xfer commits. Then audit reads A=45, fails if check.

B. True / False A=45, B=55, audit=45

Answer: True. Audit reads B=45, fails if check. Then audit reads A=55, passes if check. Then xfer commits. Then audit reads A=45, adds to sum.

C. True / False A=45, B=55, audit=55

Answer: True. xfer commits. Then audit reads A=45, fails if check. Then audit reads B=55, passes if check. Then audit reads B=55, adds it to sum.

D. True / False A=45, B=55, audit=90

Answer: False. The only way to get 90 would be to get audit() to add 45 twice. audit() only adds 45 if it first sees 55, then xfer commits, and then it reads 45 when adding to sum. Since there is only one xfer that can commit, there's only one 45 that audit can add.

E. True / False A=45, B=55, audit=100

Answer: True. Audit reads A=55, passes if check. Then xfer commits. Then audit reads A=45, adds it to sum. Then audit reads B=55, passes if check. Then audit reads B=55, adds it to sum.

F. True / False A=45, B=55, audit=110

Answer: True. Audit reads A=55, passes if check. Then audit reads A=55, adds it to sum. Then xfer commits. Then audit reads B=55, passes if check. Then audit reads B=55, adds it to sum.

V Performance

Ben Bitdiddle runs the database for a credit card processing company. Ben's clients are online stores that each have a single program running the following pseudocode to charge customer credit cards:

```
while True:
  ccnum, amount = get_next_order()
  send(ben, {ccnum, amount})
  status = recv(ben)
```

Ben's server looks like the following pseudocode, where `write_log()` prepares the log records, but only `flush_log()` writes to disk. Assume that every log flush requires seeking to a new location. Assume that `reply()` does not block.

```
while True:
  ccnum, amount = recv()
  status = check(ccnum, amount)
  if status == OK:
    write_log(ccnum, amount)
    flush_log()
  reply(status)
```

Ben's database must commit each credit card charge operation before responding to the client. Ben uses a typical rotational disk, with a 10 msec average seek time, and 100 MB/sec sequential throughput. Each log record is 512 bytes. The round-trip latency between each client and the server is 100 msec. Assume that the `check()` function is instantaneous.

10. [4 points]: How many credit cards can Ben's system successfully charge per second, if Ben has a large number of clients?

Answer: approximately 100 per second: limited by the server only being able to issue 1 `flush_log` per 10 msec seek.

11. [6 points]: How many credit cards can Ben's system successfully charge per second, if Ben has 5 clients, the clients spend almost no time in `get_next_order()`, and `get_next_order()` never blocks?

Answer: approximately 50 per second: limited by the requests that a client can issue: one request per 100 msec roundtrip to the server.

Initials:

Ben decides that he wants to process more transactions per second, and changes the server code to flush the log after a batch of records have been written to the log:

```
while True:
  for i = 1..N:
    ccnum[i], amount[i] = recv()
  for i = 1..N:
    status[i] = check(ccnum[i], amount[i])
    if status[i] == OK:
      write_log(ccnum[i], amount[i])
  flush_log()
  for i = 1..N:
    reply(status[i])
```

12. [8 points]: How many credit cards can Ben's modified system successfully charge per second, if Ben has a large number of clients? Assume the best choice of N.

Answer: about 204,800 per second: limited by the bandwidth of the disk: 100MB/sec / 512 bytes per record = 204,800 records per second.

End of Quiz II

Double-check that you wrote your name on the front of the quiz, and circled your recitation.

Initials: