

Loading Based on Imperfect Data

Andreas Fredriksson

Sr Engine Programmer, Insomniac Games

GAME DEVELOPERS CONFERENCE®

SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29

2013



This talk is about the technology we had to develop to ship our most recent game, Fuse. Fuse is a third-person action game with heavy focus on Co-op play -- up to four players.



Fuse is coming out this spring on PS3 and Xbox 360.
So now that you know a little bit more about our game, let's get started.

Fuse & The Insomniac Engine

- Fuse is our first cross-platform title
 - PS3
 - Xbox 360
 - PC build for development & testing
- Engine focused on tools UX & iteration time
- This talk: How we got the game on disc!

Fuse is the first title to ship using Insomniac's new engine. No fancy engine name, sorry!

Supports Xbox, PS3 and PC for dev/testing.

Big shift away from runtime perf at any cost to focus on tools UX & iteration time

We try to avoid data baking at build time. Example tradeoff: Dynamic lighting only, no lightmaps.

I joined the Insomniac Core team in 2012 – we had a great engine for build new stuff, but it had never been used from optical media.

My group had the task of getting this thing ready for shipping a game.

This talk is the story on how we got the game loading well given the constraints we had.

Fuse Asset Stats

- 115 GB source data
- 10 GB runtime data
 - ~5.5 GB per SKU (unique files)
- 94 unique game regions in SP
 - Typical region = 6,000 assets out of 45k

Reason for high asset count in a game region: lots of small assets (actors, gameplay elements). Split up into small pieces for iteration speed purposes.

Data Building

- Built automatically in the background
- Assets mostly built 1:1 source:output
 - No global view of dependencies
- Goal: one asset change - one file rebuild

Novel feature of the tools – data building runs in the background without user interaction. No need to explicitly build a level to run the game. Needed a very fast build system to achieve this, which means, spend as little time as possible building each thing.

The data building is performed by LunaServer, a daemon type executable that always runs in the background on every machine. Besides coordinating data builds, it also provides a suite of REST-based web services for our tools (scene editor, undo queue, transparent asset version upgrades). We also use it as a file server to deliver built output files to the game over the network.

The enemy of build performance is dependencies. If we reduce them to a minimum we speed everything up. Most of our asset builds only need to look at their immediate dependencies and record them for later resolution. For example, a model refers to a material, but the model data does not rebuild when the material changes, and vice versa. The model only stores a reference to the material asset, not the material data itself.

We have a couple of exceptions to this rule for runtime performance reasons, region builds are allowed to read model data to aggregate their collision geometry for example.

Data Linking at Runtime

- The dark side of the simple build system
 - “Buy now, pay later!”
- Loose file loading
 - I/O and dependency detection run in lockstep

Paying for the simpler build logic at runtime.

Asset dependency graph unknown until last moment, when we actually load an asset into the engine. We act on dependencies as they become known to us. Compare to static/dynamic code linking; this is similar to an extreme case of DLL linking where every object file is its own DLL.

We call this loading approach “Loose loading”

Loose Loading Flow

Dependency Detection

I/O Ops

▼ = seek

Time →

Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

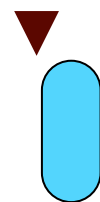
Loose Loading Flow

Dependency
Detection

Region

I/O Ops

▼ = seek



Time

Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

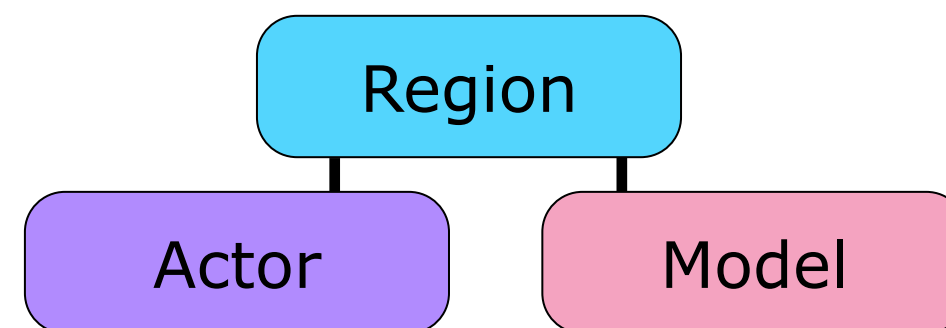
Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

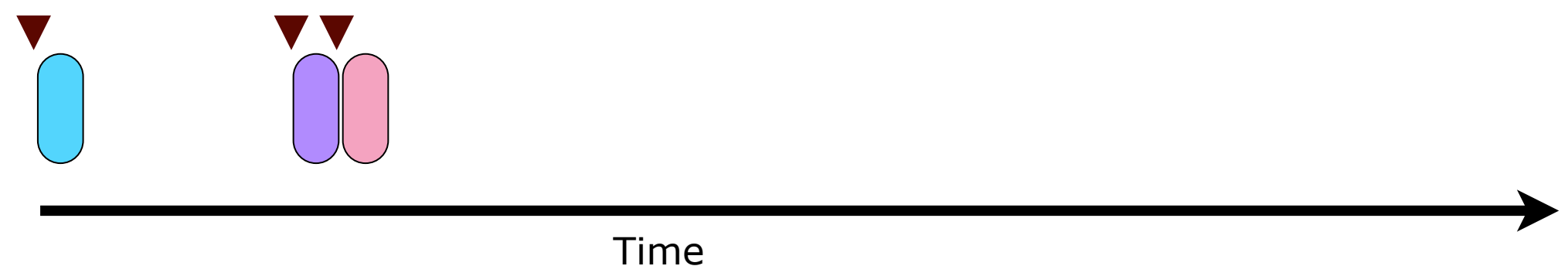
Loose Loading Flow

Dependency
Detection



I/O Ops

▼ = seek



Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

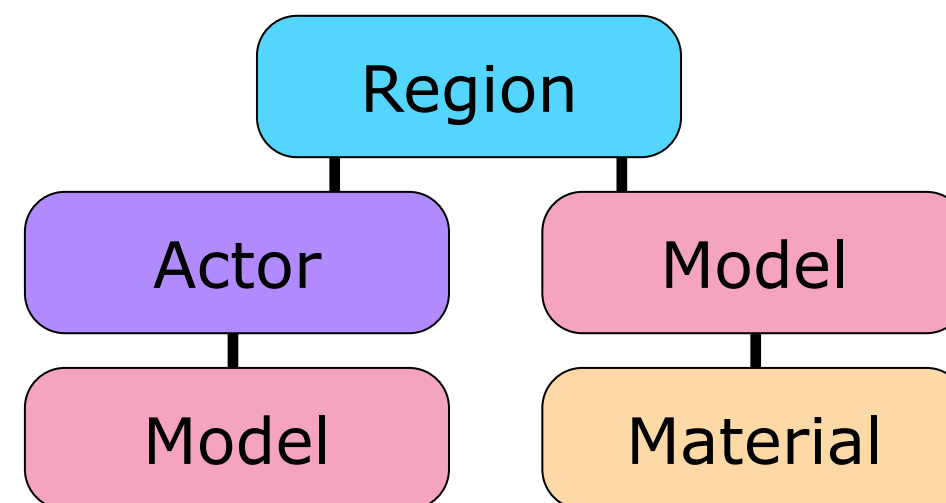
Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

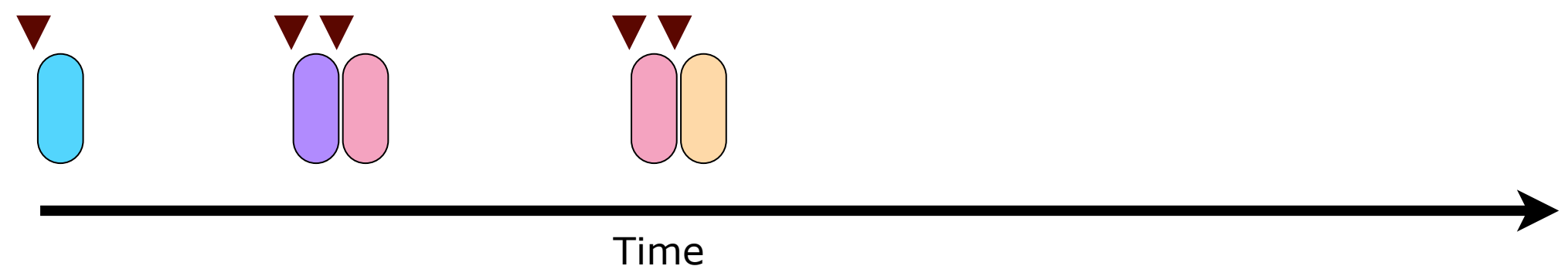
Loose Loading Flow

Dependency Detection



I/O Ops

▼ = seek



Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

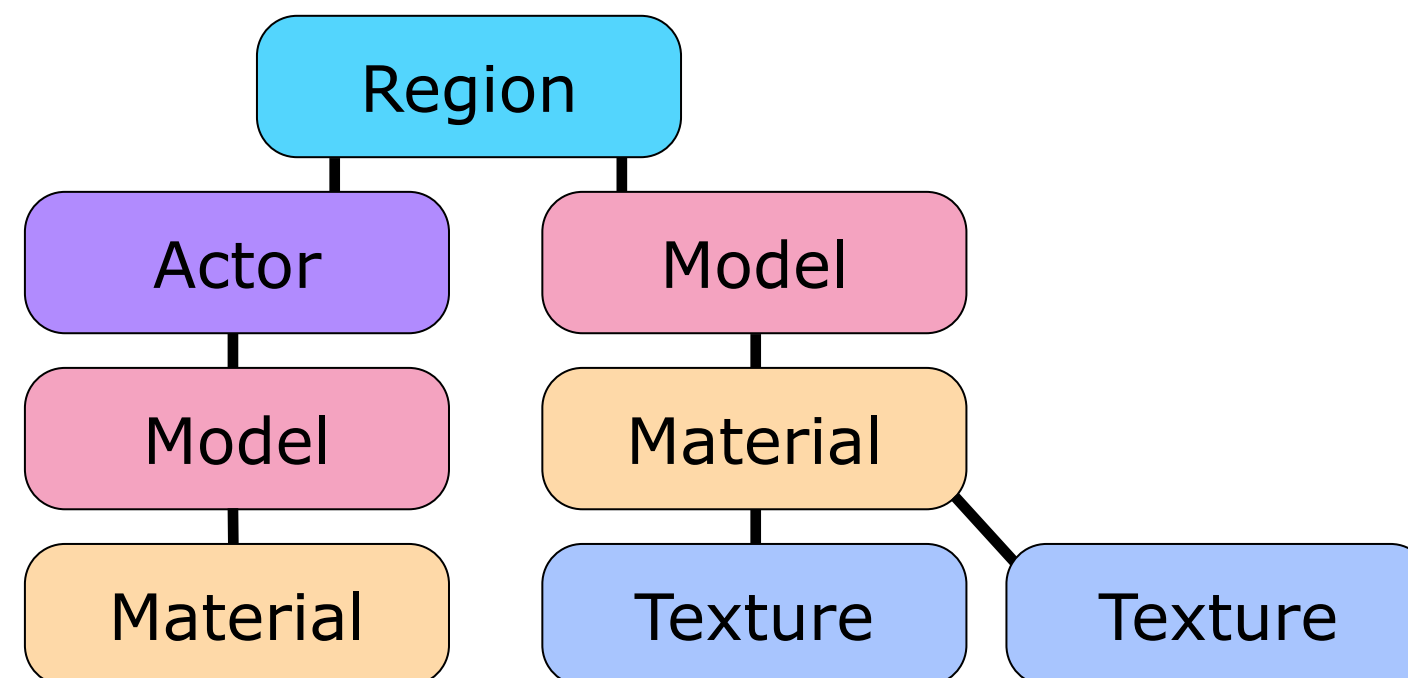
Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

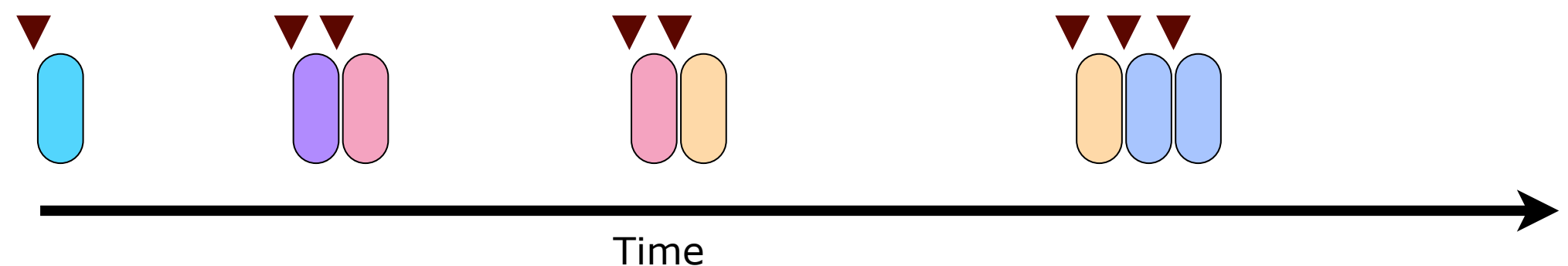
Loose Loading Flow

Dependency Detection



I/O Ops

▼ = seek



Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

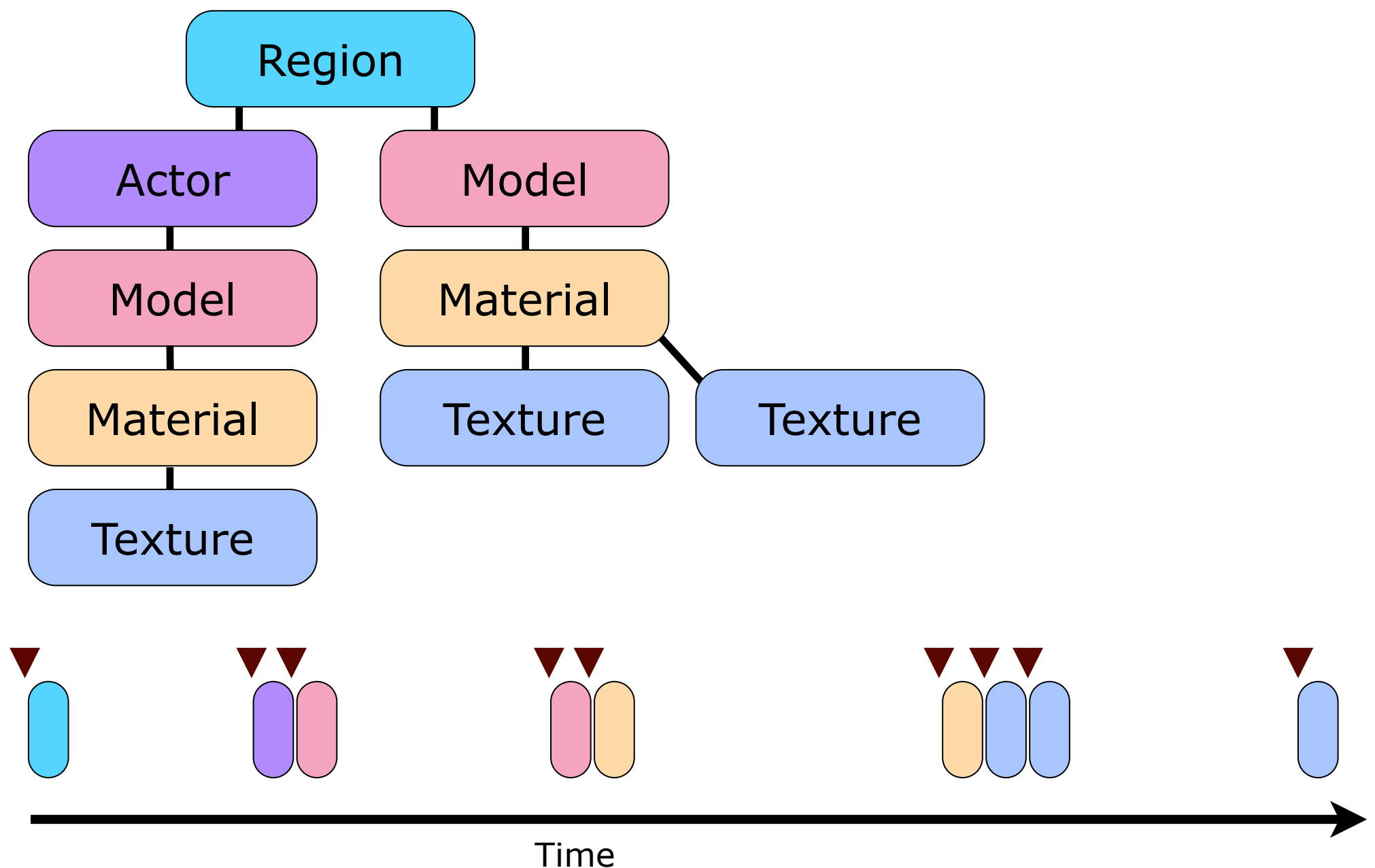
Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

Loose Loading Flow

Dependency Detection



I/O Ops

▼ = seek

Simplified model of the loose loading scheme. Consider a region with a static model and an actor (visually represented by a model).

Algorithm:

- * Load single asset data into memory
- * Construct asset + trigger loads for dependencies (in sync with frame loop)
- * Repeat for newly required assets

Gaps in I/O flow due to engine code resolving dependencies and issuing new loads in sync with frame and GPU usage. Can loose up to 33 ms (30 FPS game). (Alternative is to run multi-threaded, but we wanted simplicity.)

Loose Loading Benefits

- Load any asset at any time
 - Great for prototyping
- Assets only stored once in RAM
 - Reference counting
- Easy to reload assets at runtime

Loose loading is great for our level editing tools as they can just pull in any asset at any time. The level editor links with our engine code and uses the same basic loading infrastructure as the game.

The loading scheme also tried to reuse asset data that's already in RAM via reference counting. That means we only pay once for shared references to memory heavy assets like textures and models. The runtime memory pool for assets (excluding textures) is about 70 MB in Fuse.

The standalone nature of assets also means it becomes easier to support hot reloading of assets at runtime.

Optical Media

- Loose file loading on DVD = PAIN!
 - 8 minutes to load 160 MB production level
 - Typical seek time = ~80-200 ms per file
- Unable to schedule I/O operations
- Needed a 20x load time improvement

The loading time for loose loading on DVD are ridiculous as expected. A quick test showed that the load time was totally dominated by seeks (97%). This didn't come as a surprise, given that we were taking loading decisions at the very last moment, so we couldn't do any sort of intelligent scheduling against the drive.

Due to the design of our basic loading system, the order in which loads are issued can also differ from run to run (timing dependent, waiting for texture defragmentation, things like that), so it was not feasible to record loading patterns from game sessions as the data fluctuated quite a bit.

It was clear we needed a 20x load time improvement over this baseline to ship the game. But we didn't want to introduce a mandatory data linking step in the build system as we depended on fast iteration times to get the game done. We also didn't want two different loading pipelines if we could avoid it, as that would create different behavior (and bugs!) between dev and disc builds of the game.

At this point we considered using file caching libraries like FIOS, but we couldn't find something that would help our worst case (Xbox without hard drive cache). So we decided to try to solve the problem another way.

Fixing Optical Media

- Idea: Preload list of assets into RAM
 - “Load list”
 - Could also speed up dev loading
- Requires full dependency info to generate
 - Which we didn’t have (by design)

As our loose loader first considered assets already in RAM, our idea was to preload assets into RAM so the existing loader would become a no-op. If we had such a list of assets to preload, we could pre-load things we knew were going to be requested later. We call a these lists of asset references load lists – instructions for the loader.

To come up with a list like that we needed full dependency graph information from our assets however, which was something we didn’t have by design.

Imperfect Data? Yes, we can!

- Have game deal with stale data
- Pre-load an (imperfect) list of assets
 - Can schedule this bulk I/O *much* better
- Regular loose loading commences
 - Majority of data already in RAM
- Data only imperfect during dev iteration!

Loose loading algorithm is forgiving to any imperfections in the load list data:

– Missing assets brought in on demand, – Unwanted assets dropped via ref counting

Memory use can fluctuate a little during development, this was OK (devkit RAM)

For final data builds we can make sure the loadlist data is good to avoid loose load penalties.

During dev, we can gracefully support arbitrarily stale data, even not having any.

Drawback: Usability problem – People still waited for load lists to build. Important to communicate it was optional to the production team.

Building Load Lists

- Background optimization task
- Run only when build system is idle
 - Data not required to run game
- Produced by doing full dependency scans
 - Need to crawl through lots of data

As the load list data doesn't have to be perfectly up-to-date, we build it only when there's nothing else to do. The game will happily load stale load list data and cope with it accordingly. This covers the case where the asset database is being worked on and is in flux.

The load list data itself is stored as a regular output file from the build system. The game gracefully handles both the case when the load list is completely missing and when it is heavily out of date.

Load List Data

Filename Hash	Asset Type	File Size
fc8a137a	kModel	98760
7ae8912a	kModel	189012
1945fea8	kMaterial	890
...

The load list data itself is very compact and just lists the name hashes and asset types of the referenced items. The asset type is necessary so we know what asset manager to allocate memory from when we're processing the load list. It also includes the file size information so we can allocate memory without looking at file sizes.

Load List Flow

I/O Ops

▼ = seek

Time



Armed with the load lists data we can take another look at the loading flow.

All our files are now issued to the drive in bulk.

We're still seeking between most files, but at least we're able to issue all I/O requests in a much tighter grouping as we don't need to wait for a safe point in the frame.

In this example, the last texture reference was not part of the load list data so it was picked up in a loose load of its own. This will not happen in the retail game, but only during iteration.

Also notice that sometimes files can be loaded without seeks if they happen to sit next to each other on the disc, but that's just by chance at this point – we still hadn't optimized the layout.

Load List Flow

Load List I/O

Load List

I/O Ops

▼ = seek



Armed with the load lists data we can take another look at the loading flow.

All our files are now issued to the drive in bulk.

We're still seeking between most files, but at least we're able to issue all I/O requests in a much tighter grouping as we don't need to wait for a safe point in the frame.

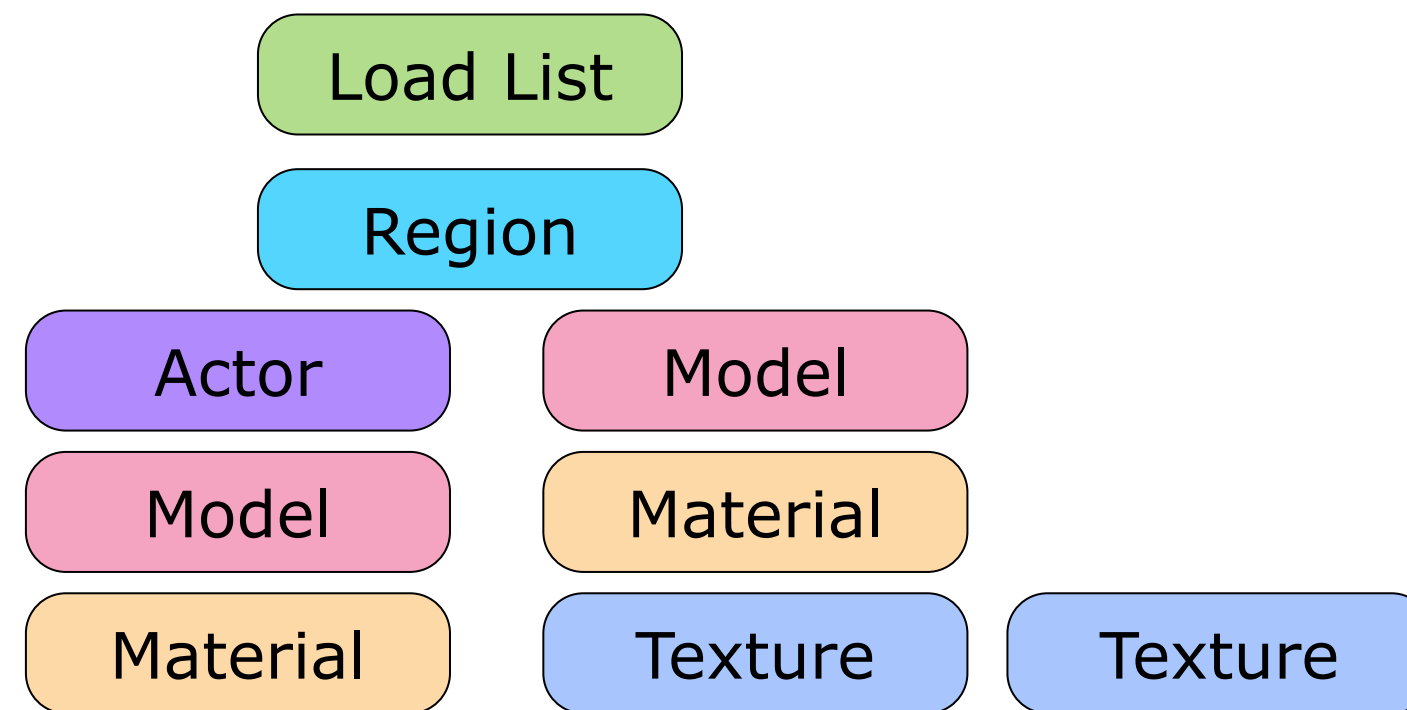
In this example, the last texture reference was not part of the load list data so it was picked up in a loose load of its own. This will not happen in the retail game, but only during iteration.

Also notice that sometimes files can be loaded without seeks if they happen to sit next to each other on the disc, but that's just by chance at this point – we still hadn't optimized the layout.

Load List Flow

Load List I/O

Asset Creation



I/O Ops

▼ = seek



Armed with the load lists data we can take another look at the loading flow.

All our files are now issued to the drive in bulk.

We're still seeking between most files, but at least we're able to issue all I/O requests in a much tighter grouping as we don't need to wait for a safe point in the frame.

In this example, the last texture reference was not part of the load list data so it was picked up in a loose load of its own. This will not happen in the retail game, but only during iteration.

Also notice that sometimes files can be loaded without seeks if they happen to sit next to each other on the disc, but that's just by chance at this point – we still hadn't optimized the layout.

Load List Flow

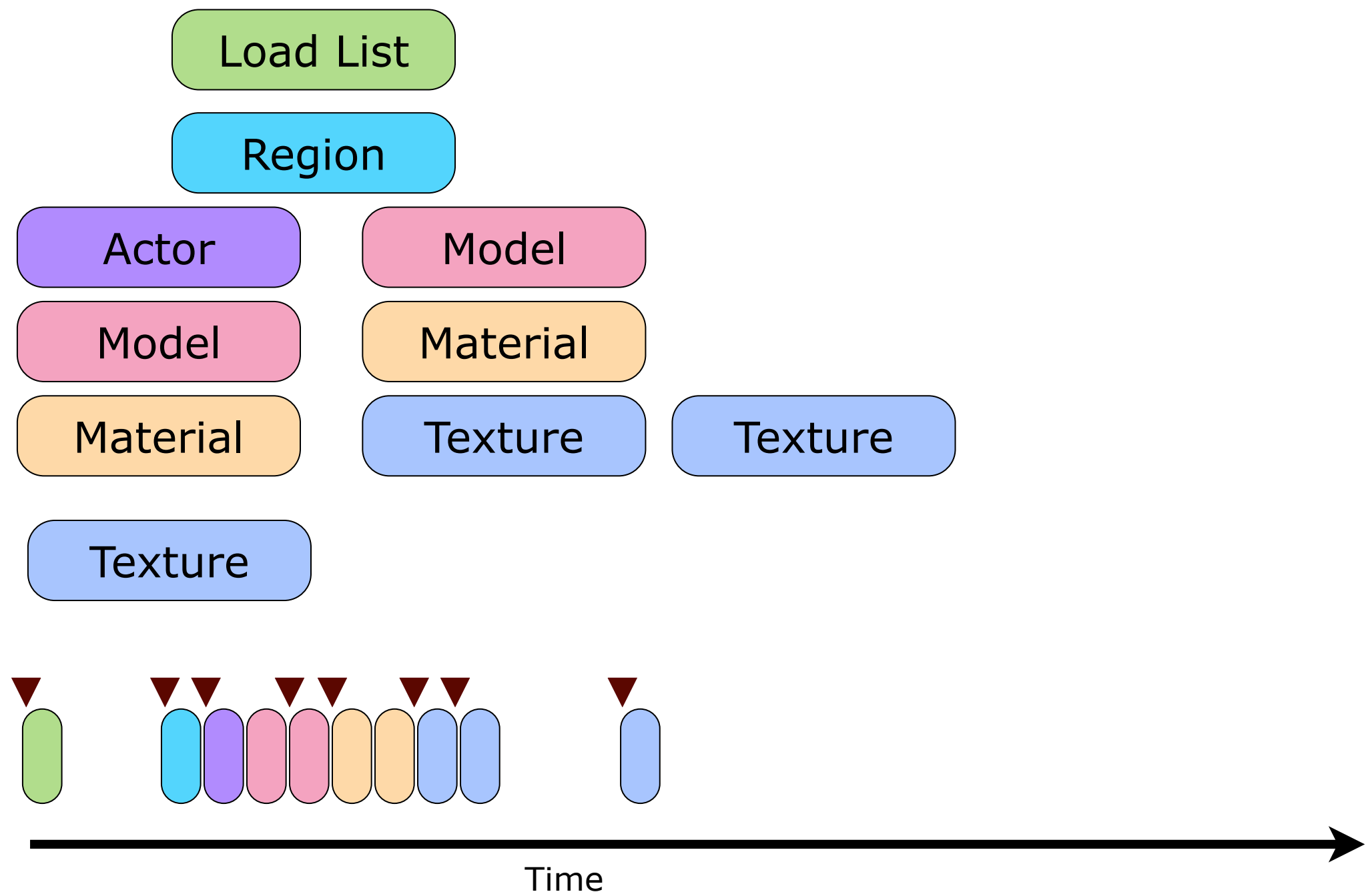
Load List I/O

Asset
Creation

Dep Detection
(+ Loose Load Fixup)

I/O Ops

▼ = seek



Armed with the load lists data we can take another look at the loading flow.

All our files are now issued to the drive in bulk.

We're still seeking between most files, but at least we're able to issue all I/O requests in a much tighter grouping as we don't need to wait for a safe point in the frame.

In this example, the last texture reference was not part of the load list data so it was picked up in a loose load of its own. This will not happen in the retail game, but only during iteration.

Also notice that sometimes files can be loaded without seeks if they happen to sit next to each other on the disc, but that's just by chance at this point – we still hadn't optimized the layout.

Load List Results

- Full level w/ load lists on DVD: 1 minute
 - Dev-time HTTP loading speedup: ~50%!
- Needed another 2-3x DVD speedup to ship
- Good: Stalls in I/O pipeline gone
- Bad: Still seek bound!

So now we have all our requests tightly grouped against the optical device, which is great. We saw an 8x improvement in performance by removing the bubbles in the I/O pipeline.

But even though our archive layer tries its best to sort a large array of I/O requests in the most optimal order to avoid seeks on the media, we were still seek bound in most scenarios. So it was clear we needed to eliminate seeks by a couple of orders of magnitude to ship the game.

Archives

- Discs store “archive” data - not files
- Nothing intelligent in the layout yet
- Archive builds not part of everyday workflow
 - Could afford to spend time to improve file layout
 - Could require fully accurate load list data

Optical file systems are shitty.

We create archives using a special path of the build pipeline that is not invoked by people iterating on the game. Typically a build master will kick off and produce a disc build. So we could afford this process to require fully up-to-date load lists information and take some extra time to build.

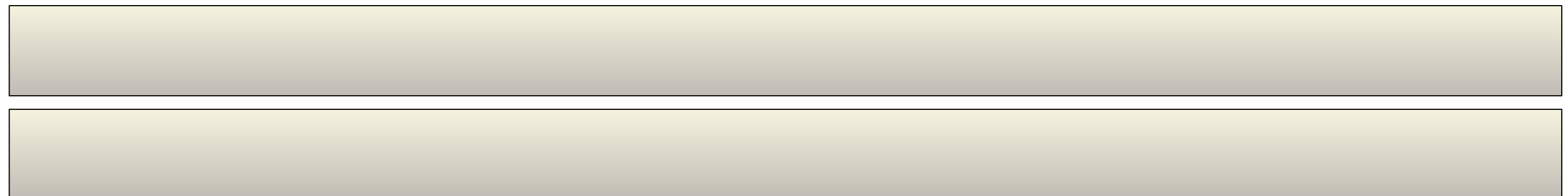
Archive Data

Table of Contents File (Memory Mapped) ~800 kb

Sorted by Hash ↓

Filename Hash	Compressed Size	Decompressed Size	Location
0001a234	1768	2896	0af82b60
0001a712	276193	358900	05fab88a
0001b682	57891	57891	1a58934c
...

Data Files (2 GB)



Here's a visual representation of our archive data.

The TOC is sorted by file hash and stores compressed/decompressed sizes for all files.

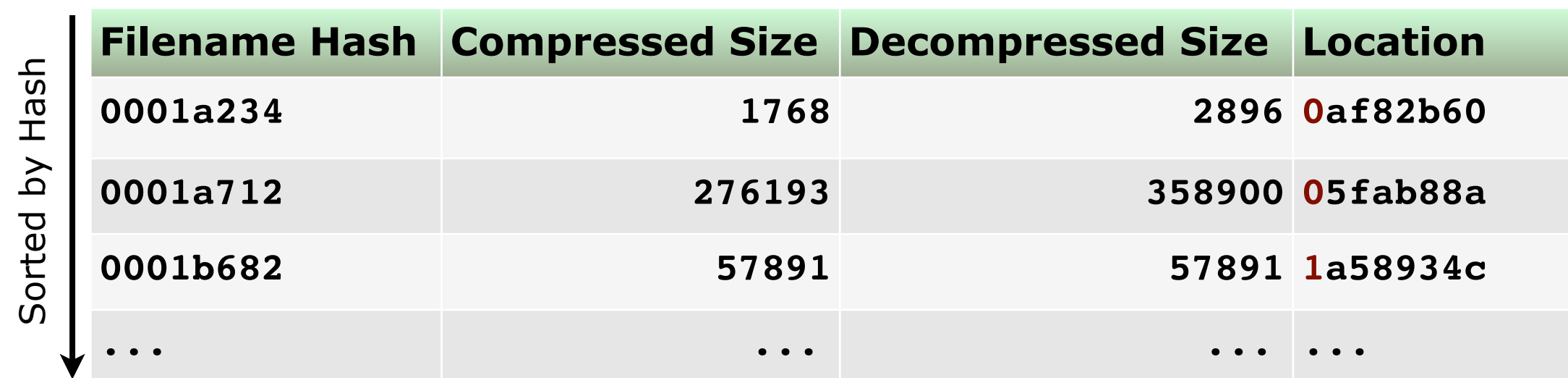
The location field stores 4 bits to indicate what segment the file is in, and a 28 bit offset within that segment where the file is stored. The 28 bit offset is taken to be 16-byte aligned, so we shift that up to get the full offset.

Basic mechanism used to order bulk I/O requests against the media:

* Look up all files. * Sort by location on disk. * Execute I/O ops in order.

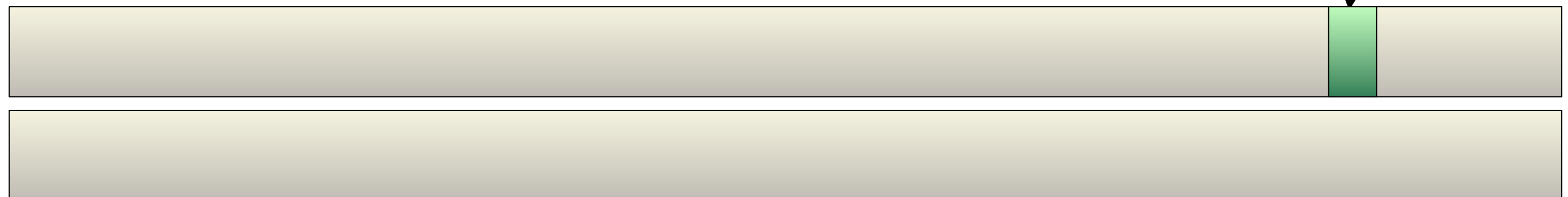
Archive Data

Table of Contents File (Memory Mapped) ~800 kb



Filename Hash	Compressed Size	Decompressed Size	Location
0001a234	1768	2896	0af82b60
0001a712	276193	358900	05fab88a
0001b682	57891	57891	1a58934c
...

Data Files (2 GB)



Here's a visual representation of our archive data.

The TOC is sorted by file hash and stores compressed/decompressed sizes for all files.

The location field stores 4 bits to indicate what segment the file is in, and a 28 bit offset within that segment where the file is stored. The 28 bit offset is taken to be 16-byte aligned, so we shift that up to get the full offset.

Basic mechanism used to order bulk I/O requests against the media:

* Look up all files. * Sort by location on disk. * Execute I/O ops in order.

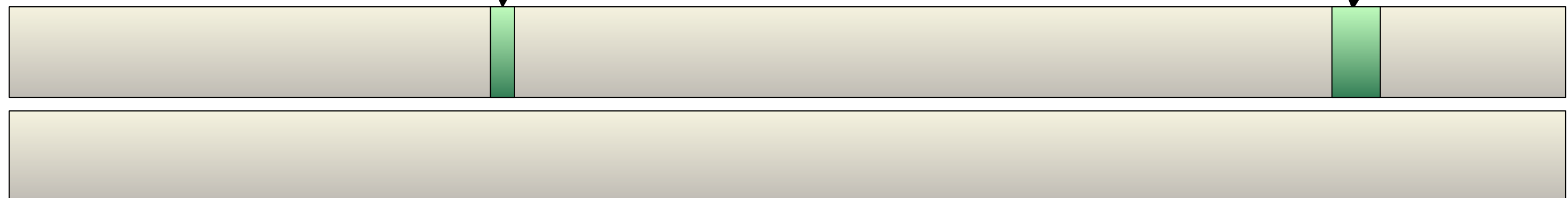
Archive Data

Table of Contents File (Memory Mapped) ~800 kb

Sorted by Hash ↓

Filename Hash	Compressed Size	Decompressed Size	Location
0001a234	1768	2896	0af82b60
0001a712	276193	358900	05fab88a
0001b682	57891	57891	1a58934c
...

Data Files (2 GB)



Here's a visual representation of our archive data.

The TOC is sorted by file hash and stores compressed/decompressed sizes for all files.

The location field stores 4 bits to indicate what segment the file is in, and a 28 bit offset within that segment where the file is stored. The 28 bit offset is taken to be 16-byte aligned, so we shift that up to get the full offset.

Basic mechanism used to order bulk I/O requests against the media:

* Look up all files. * Sort by location on disk. * Execute I/O ops in order.

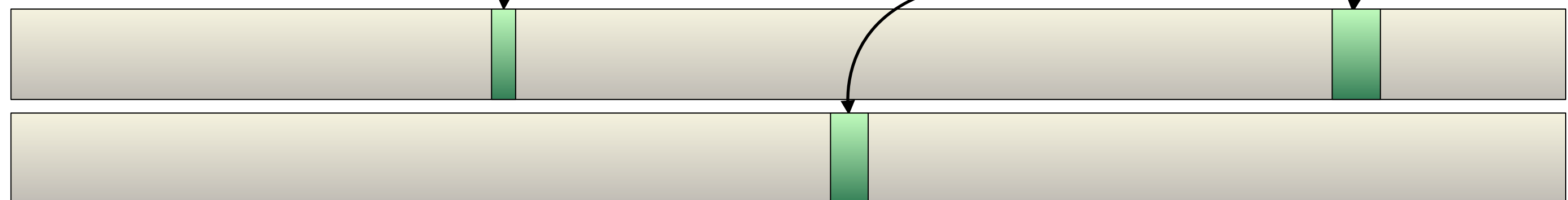
Archive Data

Table of Contents File (Memory Mapped) ~800 kb

Sorted by Hash ↓

Filename Hash	Compressed Size	Decompressed Size	Location
0001a234	1768	2896	0af82b60
0001a712	276193	358900	05fab88a
0001b682	57891	57891	1a58934c
...

Data Files (2 GB)



Here's a visual representation of our archive data.

The TOC is sorted by file hash and stores compressed/decompressed sizes for all files.

The location field stores 4 bits to indicate what segment the file is in, and a 28 bit offset within that segment where the file is stored. The 28 bit offset is taken to be 16-byte aligned, so we shift that up to get the full offset.

Basic mechanism used to order bulk I/O requests against the media:

* Look up all files. * Sort by location on disk. * Execute I/O ops in order.

Archive File Layout

- Goal: Seek time reduction
 - File order inside archives key
- Tried different approaches
 - Sort by histogram (# of referencing regions)
 - Sort by sampled game loading patterns
 - Brute force solving algorithms
- For the Fuse dataset, nothing helped

So to combat our seek times we had to find a way to reorganize the files inside our archives into an order that would allow all regions to load with a minimum of seeks.

There are what seems like an infinite number of ways to try to reorganize files, and we tried quite a few. But nothing we tried ended up helping us across the board. Lots of effort went in to it.

Some reasons for this:

- * Highly fragmented sharing patterns across regions. Not a lot of clear-cut groups that could be stashed away in one group.

- * Lots and lots of asset files.

Typically what helped one case penalized another.

As it turned out, asset sharing is great--faster/cheaper to build game--but not so great for disc loading.

So the next logical step was to consider duplicating files in archives to combat seek times.

File Duplication

- Store more copies of files to reduce seeks
 - Constraint: Only about 2 GB slack available!
- Naive approach: Duplicate on every use
 - Result: 6 GB over disc budget :(
- Needed selective duplication
 - No time to handhold the process
 - Wanted fully automatic builds

It's a simple idea: we store multiple copies of certain files in our archives. Good candidates for duplication are small outliers that cause excessive seeking. But how can we come up with this list of files to duplicate? And how many times do we duplicate them?

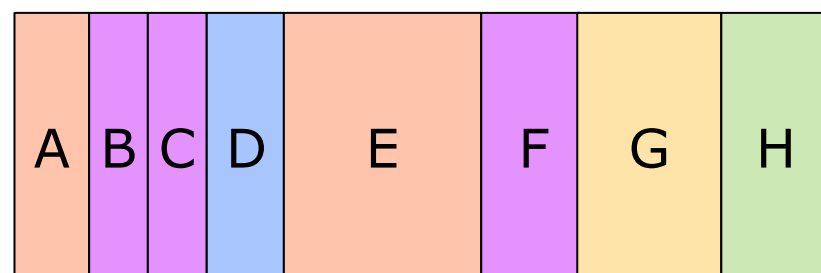
The naive approach is to duplicate every file on every reference. For example, if three regions all use an asset, that asset would be stored three times in the archives. Each of those regions could then be loaded with just one seek to find the first file in its group.

Trying this out blew our disc budget however, we ended up with a disc several GB larger than the physical media. So it was clear we needed a selective approach to the problem.

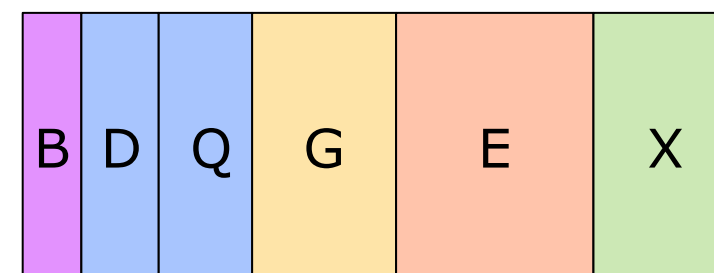
We also didn't want to introduce manual steps to help with the duplication process, and were looking for a fully automated disc building pipeline.

De-Duplication

Region A



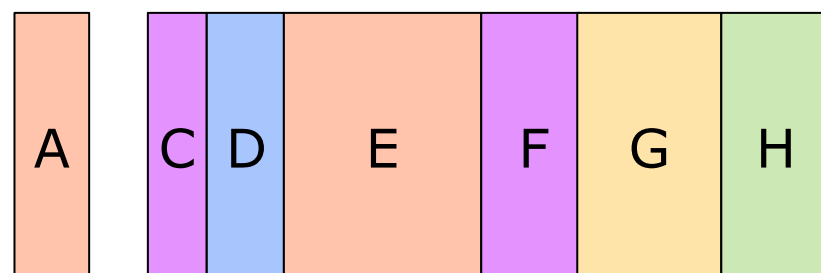
Region B



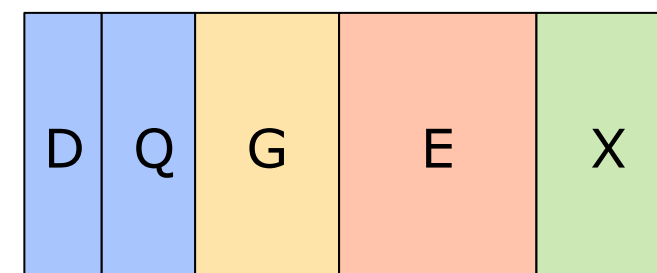
We ended up with something we call “de-duplication”. Here’s a visual representation of the basic algorithm. We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size. Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

Region A



Region B



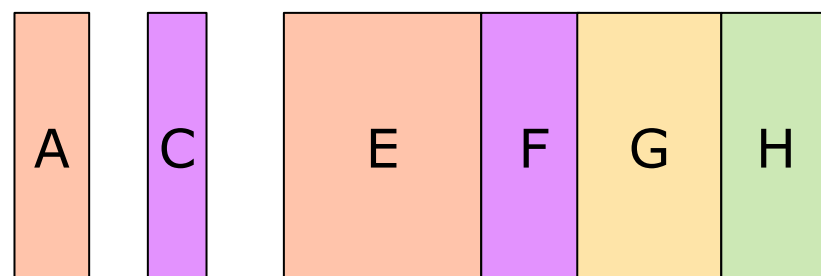
Shared Bucket



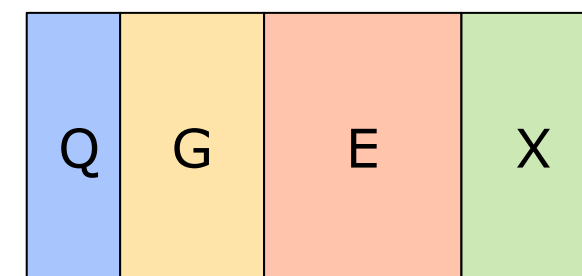
We ended up with something we call “de-duplication”. Here’s a visual representation of the basic algorithm. We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size. Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

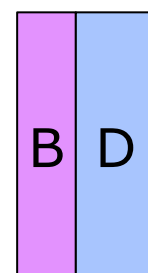
Region A



Region B



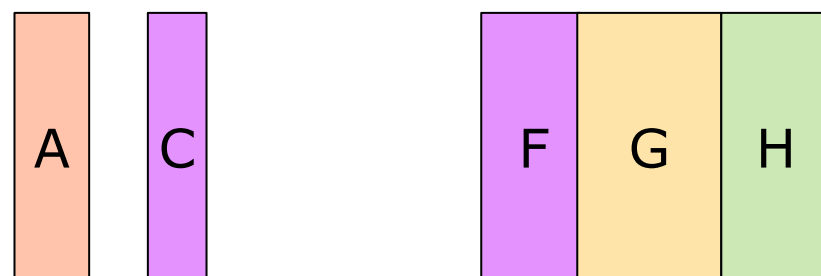
Shared Bucket



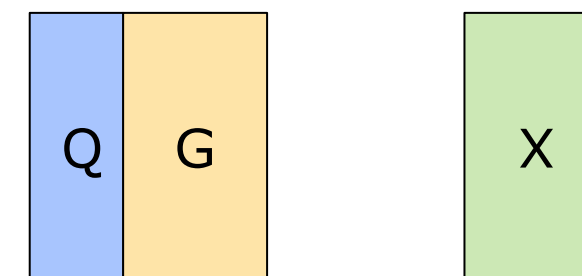
We ended up with something we call “de-duplication”. Here’s a visual representation of the basic algorithm. We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size. Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

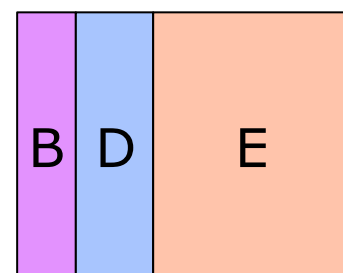
Region A



Region B



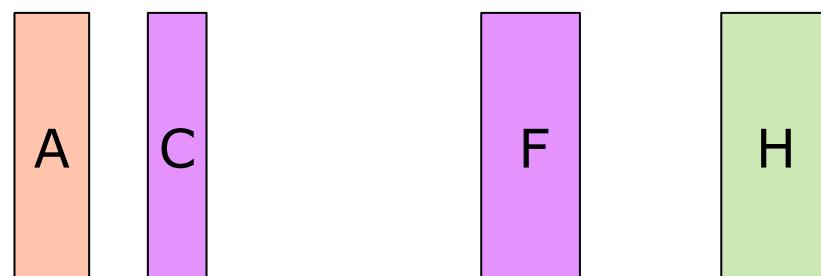
Shared Bucket



We ended up with something we call “de-duplication”. Here’s a visual representation of the basic algorithm. We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size. Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

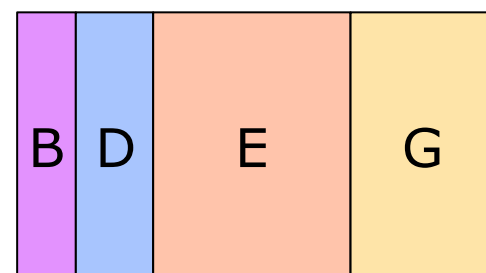
Region A



Region B



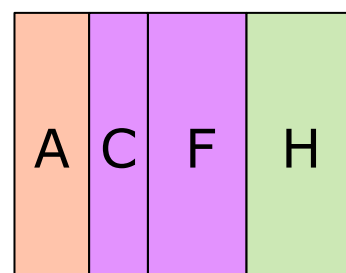
Shared Bucket



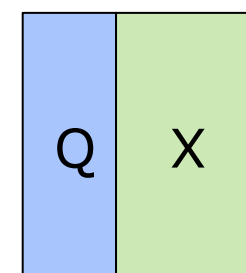
We ended up with something we call “de-duplication”. Here’s a visual representation of the basic algorithm. We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size. Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

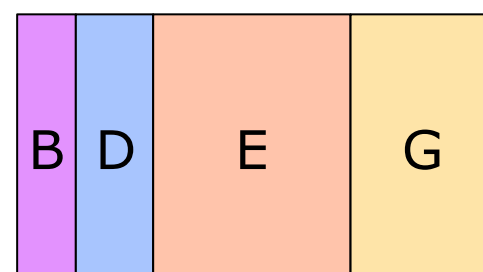
Region A



Region B



Shared Bucket



We ended up with something we call “de-duplication”.

Here’s a visual representation of the basic algorithm.

We start with two buckets of files. These contain all the files a region will read at runtime. If you were to load one of these regions now you would only have to make a single seek and then a linear read across the disc. But it’s too big. So we find common files and put them in their own bucket – that means instead of storing them twice they’re now stored once which saves us 50% of their size.

Now to load either Region A or B you also have to load this shared bucket, which costs you exactly one seek.

De-Duplication

- Start with duplicated buckets for each region
- Find largest (in bytes) shared cluster
 - Promote to new buckets & repeat
- Every new bucket adds exactly one seek
 - For every donor it de-duplicates files from

So to restate the algorithm in a bigger scope:
We start off with different file buckets for all regions, each one with a copy of any file it references.
In this state there's only one seek to load any region, as all files are tightly packed together.
But the disc footprint of this solution was too big, as we've already talked about.

De-duplication makes the total disc footprint smaller by finding clusters of shared files between other sets, and makes a single copy of those files. For every new set we introduce this way we save disc space proportional to the total file group size and the number of regions that donated files to the new set. We also introduce one seek to load any of the donor regions.

De-Duplication - Step 0

Region

Region

Region

Region

Region

Region

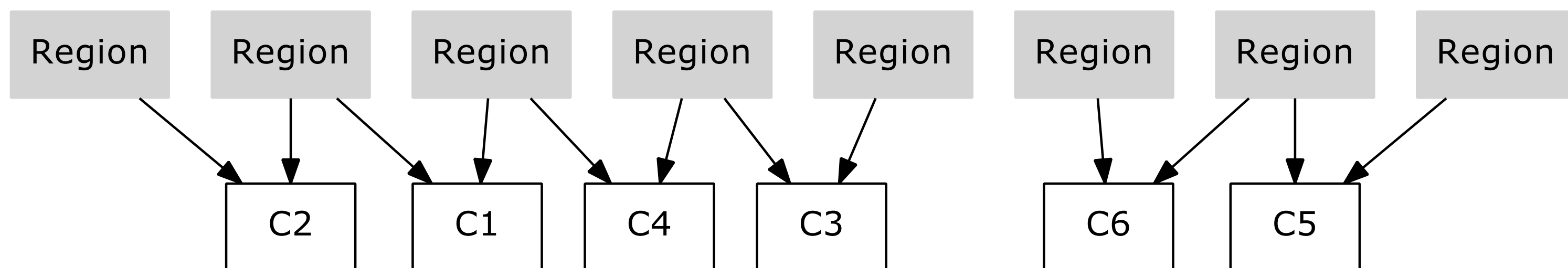
Region

Region

Total Data Size: 12 GB

Let's look at a bigger example. Here are a bunch of buckets of files representing regions with all their duplicated files. Remember that we had 94 of these in the actual game problem space.

De-Duplication - Step 1

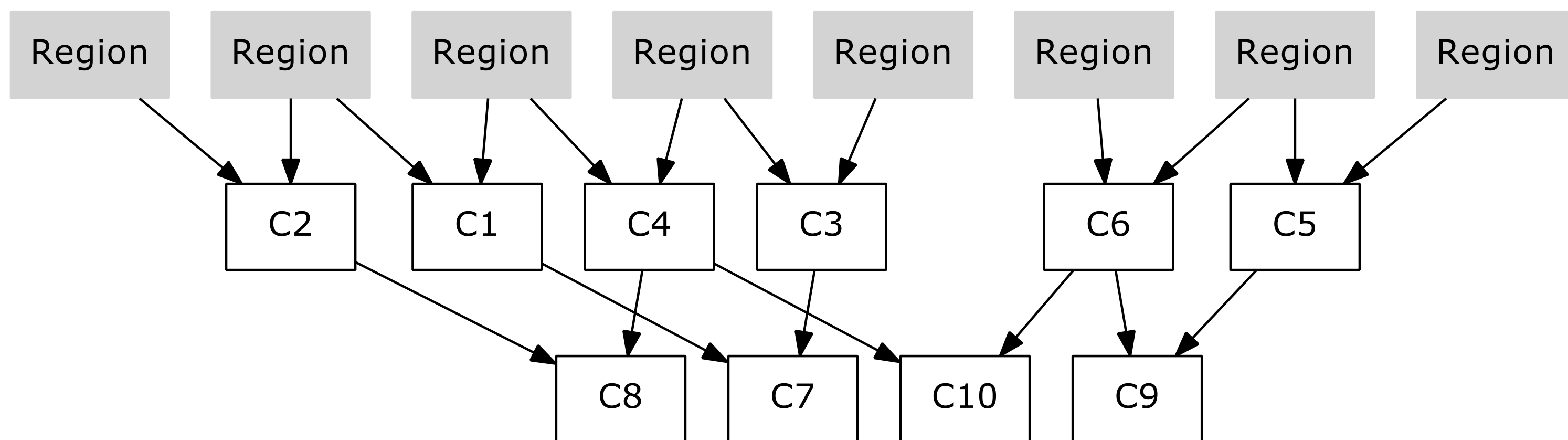


Total Data Size: 8 GB

We break out a few common clusters, sometimes more than once from each region. The total data size goes down as we de-duplicate files.

When you have more than two regions you have some choices to make – For example if you want regions to be donors more than once to these shared buckets. Check out the second guy here, it donates files to both C1 and C2.

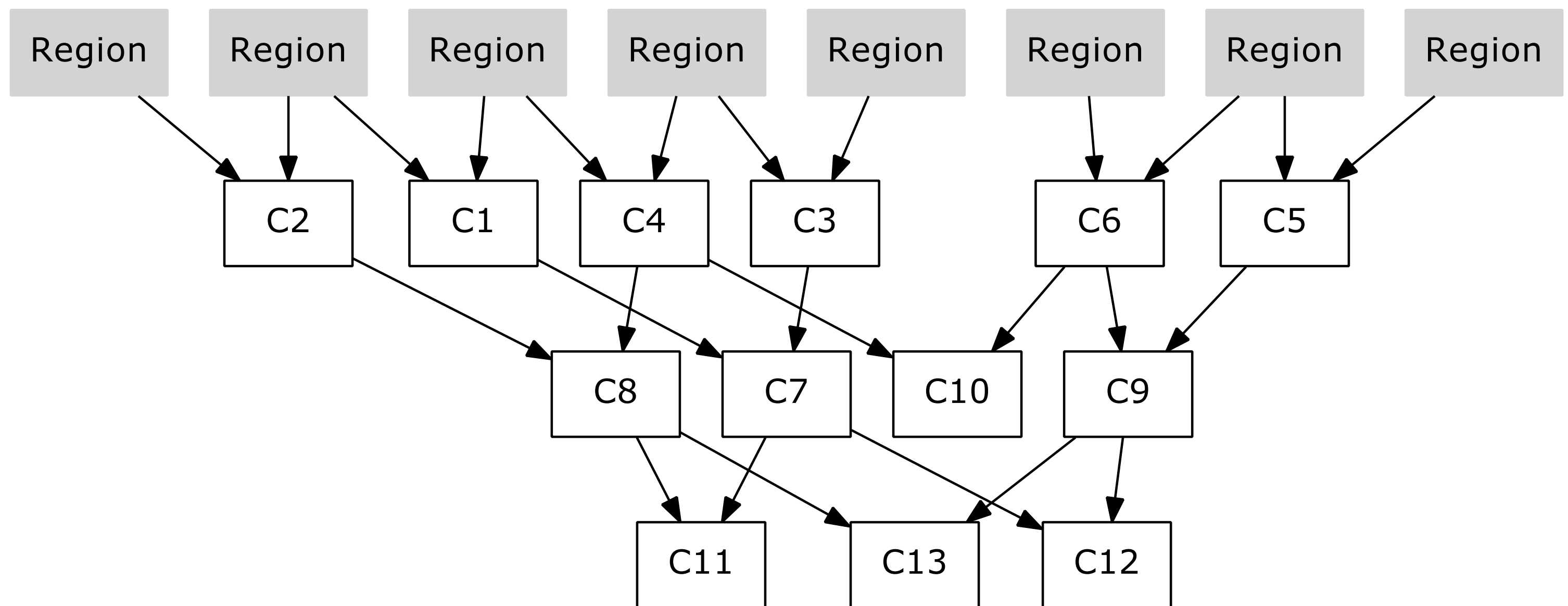
De-Duplication - Step 2



Total Data Size: 6 GB

We keep going. In this setup I'm constraining the de-duplication so it only considers the last tree level, but you definitely have the option to consider all buckets for every move. Just watch out for outliers that generate very long sharing chains. This will vary greatly with how your data is put together.

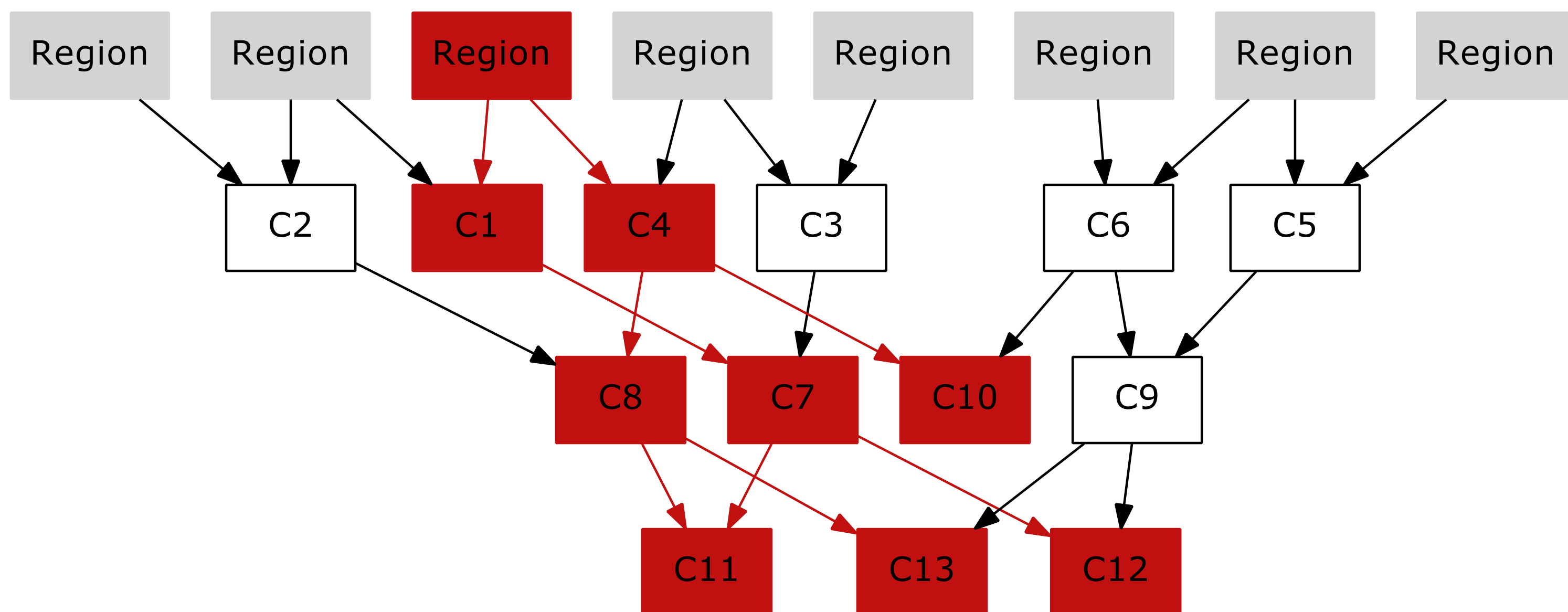
De-Duplication - Step 3



Total Data Size: 5 GB

In this case, we terminate at tree level 3. In this small example we've gone from 8 buckets to 21 buckets.

De-Duplication - Seeks



Seeks: 9

If we now want to load one of our regions, we have to load up all the buckets it donated files to as well. Visually this is equivalent to traversing all the edges of the DAG from that node. In the highlighted example here this process gives us 9 seeks in total for one of our regions.

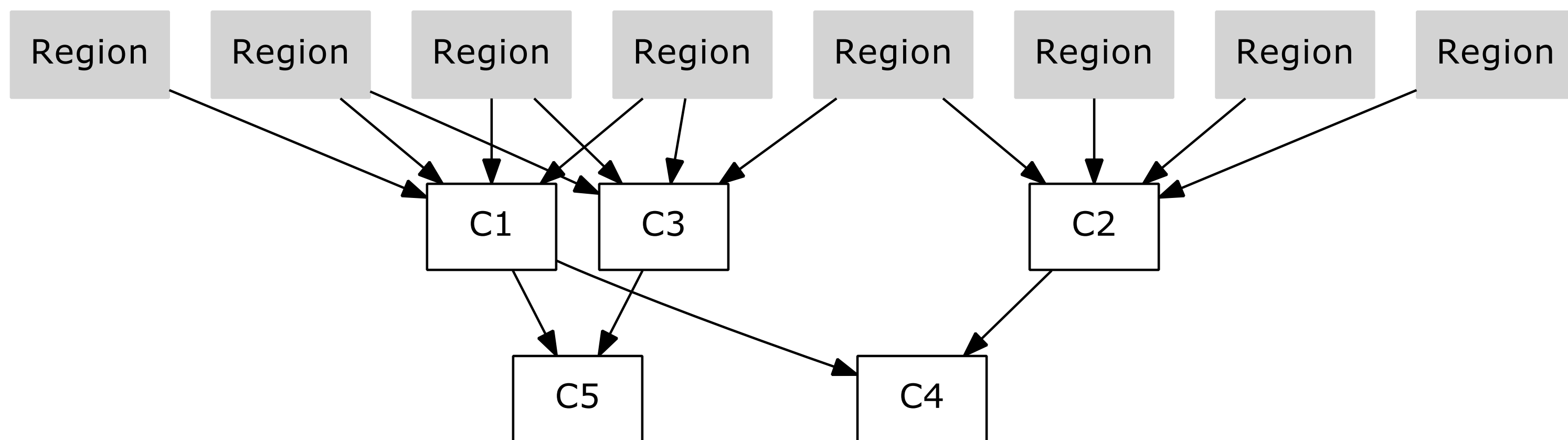
De-Duplication Results (Pairs)

- Required too many moves/splits
 - Either too big to fit on disc or too slow to load
- Wanted higher gains for each move
 - Instead of pairs, search 3 or more at once
 - Pairs => 50% savings
 - Triplets => 66% savings
 - Quads => 75% savings

Merging only bucket pairs like this was reasonably fast even for our Perl disc building pipeline. But we weren't getting good enough results. To fit in the 2 GB slack space we had available to us we had to split things too many times – the seek counts became too high which meant slow loading.

The next thing to try was to try to merge more than one set at a time. The reasoning here is pretty simple; if every move grabbed files from 4 sets instead of just 2, we would save 75% instead of 50% with the same number of seeks.

De-Duplication, Quads



If we could achieve this setup instead, we would have to seek way, way less and every duplication move saves 75% disc space instead of 50%.

So we tried it.. And our poor Perl disc pipeline ran overnight (12 hours!) until it completed. We were getting interesting data, but there was no way we could convince our build guys to wait 12 hours for a disc.

The Combinatorial Cliff

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}.$$

N	K	Combinations
94	2	4,371
94	3	134,044
94	4	3,049,501
94	5	54,891,018
94	6	814,216,767

N = Total number of buckets

K = Number of buckets to choose (each iteration)

It turns out there's one BIG problem with trying higher combination counts: performance.

Our poor Perl program was quickly overwhelmed when faced with number of combinations like this.

Remember we're searching for the best combination to de-duplicate *for every move* which means that you repeatedly hit the K-from-N number of combinations – instead of 4371 we were doing over 3 million.

De-Duplication on GPU

- Brute force search on GPU via OpenCL

K	Perl (secs)
2	15
3	3,500
4	42,000
5	Heat Death Of The Universe

CPU = 8x Xeon @ 2.93 GHz; GPU = Radeon HD 6900

Kept the brute force search algorithm, but moved it to OpenCL. The source for this tool is available, link at the end of the talk.

Embarrassingly parallel problem, maps very well to GPU.

Run 1000s of combination scorings in parallel.

Collate results on host CPU to perform dedupe moves.

De-Duplication on GPU

- Brute force search on GPU via OpenCL

K	Perl (secs)	CL/CPU (secs)
2	15	1
3	3,500	7
4	42,000	206
5	Heat Death Of The Universe	3851

CPU = 8x Xeon @ 2.93 GHz; GPU = Radeon HD 6900

Kept the brute force search algorithm, but moved it to OpenCL. The source for this tool is available, link at the end of the talk.

Embarrassingly parallel problem, maps very well to GPU.

Run 1000s of combination scorings in parallel.

Collate results on host CPU to perform dedupe moves.

De-Duplication on GPU

- Brute force search on GPU via OpenCL

K	Perl (secs)	CL/CPU (secs)	CL/GPU (secs)
2	15	1	1
3	3,500	7	1
4	42,000	206	25
5	Heat Death Of The Universe	3851	435

CPU = 8x Xeon @ 2.93 GHz; GPU = Radeon HD 6900

Kept the brute force search algorithm, but moved it to OpenCL. The source for this tool is available, link at the end of the talk.

Embarrassingly parallel problem, maps very well to GPU.

Run 1000s of combination scorings in parallel.

Collate results on host CPU to perform dedupe moves.

De-Duplication Results

- $K = 4$ (quads) was sweet spot for us
 - 25 seconds to run on commodity GPU
 - Resulting data set fits in 2 GB slack space
- Stats
 - Some files duplicated 31 times, average 6 times
 - Between 7-20 seeks required per region
 - Load times finally under control #winning

Sweet spot for us on commodity hw was $K=4$. 25 seconds is fine.

Looking at the results the splits made by this algorithm are non-obvious. If we had tried to wing this by hand it would have been a significant complexity to worry about every time the asset references and sharing patterns changed. And you'd be hard pressed to come up with the number 31 as your highest duplication number.

Final gold disc loading times between 10-20 seconds for any region.

Conclusion

- Hybrid loading approach worked great for us
 - Great iteration times during development
 - Great loading speed from DVD
- De-duplication saved our bacon
 - Automatic system saved time
 - Graceful performance scaling as asset pool grew

Great to have a single system for both dev and disc builds - one code path to debug

The End

Contact information

Twitter: @deplinenoise

Email: afredriksson@insomniacgames.com

Open-sourced de-duplication tool

<https://github.com/deplinenoise/ig-dedupe>