Reinventing the Mempool

A cluster based solution

Overview

1. Current problems with the mempool
2. Design of a solution
3. Open questions

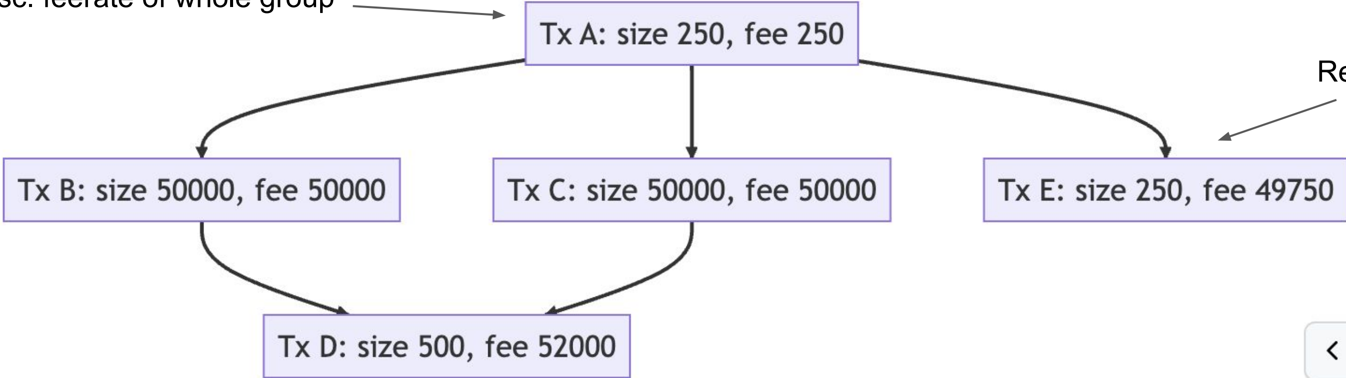
Current problems

1. Eviction is broken
2. Our mining algorithm is suboptimal.
3. RBF rules are broken: sometimes we replace things we shouldn’t, and sometimes we don’t replace things we should
   a. Probably won’t completely solve this one, but hopefully we’ll make it substantially better.

Eviction is broken

Sometimes we evict things that would be mined in the next block:

Lowest desc. feerate of whole group



Really good tx!

Mining has problems

The most important issue is that mining and eviction are not doing symmetrically opposite things: ancestor feerate-based mining is not practical to run in reverse.

But also the mining algorithm is not optimal: ancestor feerate based mining is "quick" ($O(n^2)$, probably?) but fails to find optimal blocks, eg in "children-pay-for-parents" scenarios. Would be nice if we had a way to do better.

RBF is broken

- Incentive **incompatible** replacements **can** happen
  - Because we only compare feerates of the replacement with feerates of the direct conflicts, it's possible that we are evicting some child that has a very high ancestor feerate
  - Fixing this in current paradigm is hard (see #26451)
- Many incentive **compatible** replacements **cannot** happen
  - "Pinning" - but this will be solved by v3 relay
  - No new unconfirmed parents
  - Proposals such as #26451 (to solve first problem) would make this problem worse

Frequently we've discussed that if we had a total ordering on all transactions, we could make this problem much better.

A modest proposal

We can solve these problems if we just had a total ordering on all transactions:

- Eviction and mining could be inverses of each other.
  - Improvements towards an optimal total ordering would benefit both eviction and mining.
- RBF would be much better:
  - Can eliminate uncertainty in comparing when transactions will be mined and at what feerate, for all transactions in the mempool and for a new transaction.
  - Pinning may still be an issue – but that will be solved by v3.

The main problem, and a solution?

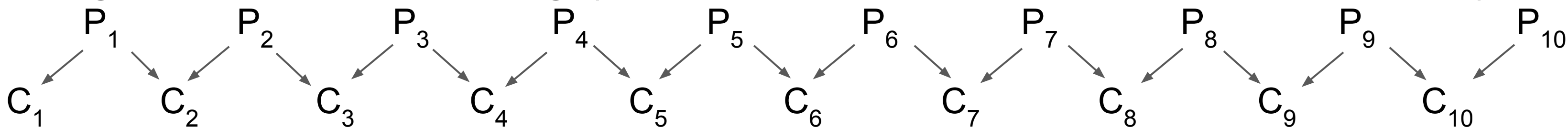
1. “Obviously correct” solution to mining/total ordering the mempool is to just look at all possible candidate sets from mempool and select the highest feerate one, and repeat.
   a. But… exponential runtime. We can’t even run ancestor-feerate-based mining on the whole mempool.

2. Idea of a solution: limit size of connected components (“clusters”) in mempool
   a. Unrelated transactions (not connected, in the graph theory sense, when looking at transactions as vertices in a graph and ancestor/descendant relationships as edges) can be evaluated separately in ANY mining algorithm
   b. Quadratic algorithms like ancestor-feerate-based transaction selection could then be feasible to maintain a total (imperfect) ordering on mempool.
   c. If we can limit cluster sizes enough, the exponential runtime of optimal sorting may be feasible.

Caveat

This would be a new policy limit.

- Existing mempool ancestor/descendant limits do **not** serve as a cap on cluster size.
- Eg consider a “trellis” transaction graph, where each ancestor has 2 children, and each child has 2 parents:



Would users accept some kind of cluster limit?

- Weird – but not so different in flavor from descendant limits.
- Unclear what the proposed value might be. Probably more than 20, and likely less than 100. More investigation needed.

Some terminology

Cluster: set of transactions that are connected to each other

Linearization: a topologically valid sorting of transactions (usually referring to transactions in a cluster, but we can refer to block construction as well)

Given a **cluster**, we can produce a **linearization** in many different ways, eg using the ancestor-feerate-based algorithm (like we use for mining today), or ideally by using the optimal algorithm, where we find the subset of transactions that have highest feerate, sort those topologically and add to our linearization, and then repeat.

A mining algorithm

Given a list of linearized clusters, what should our mining algorithm be?

Intuition:

1. The hard part – sorting each cluster, taking into account dependencies and maximizing feerate – has been done.
   a. All we want to do is “merge-sort” the clusters.
   b. Because there are no dependencies between clusters (by definition), we can mix transactions from different clusters arbitrarily.

2. Idea: look at best “package” from each cluster, and pick the one with highest feerate to add to block. Advance that cluster to its next highest feerate “package” and repeat until block is full.

More terminology

Given a linearization of a cluster [tx_1, tx_2, …, tx_n], we can calculate the **chunks** that represent where we should break up the linearization when deciding how to merge sort with other clusters (intuitively, these are packages).

Although chunks arise from the concept of packages (or “candidate sets”), we don’t need to dig into the linearization algorithm’s internals in order to define this concept generically.

Chunk example

Consider a cluster of ten transactions, with the following (fee, size) for each, which has been linearized as shown.

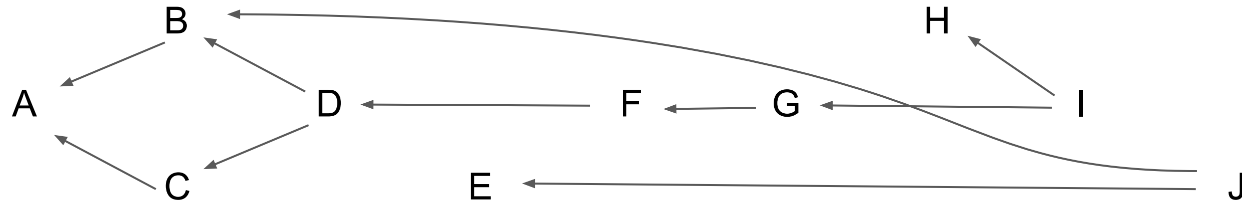
We want to figure out what the best set of transactions from this cluster would be if we were mining.

Tx A (100, 100)	Tx B (100, 100)	Tx C (300, 100)	Tx D (2000, 1000)	Tx E (500, 400)	Tx F (500, 500)	Tx G (1000, 900)	Tx H (200, 200)	Tx I (250, 240)	Tx J (100, 100)

Step 1: Figure out the maximum feerate achievable if we start at the beginning and advance through the linearization.

Where could this linearization have come from?

Note: many possible transaction graphs are compatible with this linearization. Some possibilities are:



Tx A (100, 100)	Tx B (100, 100)	Tx C (300, 100)	Tx D (2000, 1000)	Tx E (500, 400)	Tx F (500, 500)	Tx G (1000, 900)	Tx H (200, 200)	Tx I (250, 240)	Tx J (100, 100)

Chunk example (II)

Figure out the maximum feerate achievable if we were to select the first N transactions, for N=1, 2, ….

Cumulative feerates for this linearization, starting at the beginning:

1	1	1.67	**1.92**	1.76	1.59	1.45	1.42	1.40	1.39

Gives rise to this first chunk:

Tx A (100, 100)	Tx B (100, 100)	Tx C (300, 100)	Tx D (2000, 1000)	Tx E (500, 400)	Tx F (500, 500)	Tx G (1000, 900)	Tx H (200, 200)	Tx I (250, 240)	Tx J (100, 100)

Chunk example (III)

Continue in the same way, starting from after the first chunk:

Cumulative feerates for this linearization, starting from the first chunk:

-	-	-	-	**1.25**	1.11	1.11	1.1	1.093	1.089

Gives rise to this second chunk:

Tx A (100, 100)	Tx B (100, 100)	Tx C (300, 100)	Tx D (2000, 1000)	Tx E (500, 400)	Tx F (500, 500)	Tx G (1000, 900)	Tx H (200, 200)	Tx I (250, 240)	Tx J (100, 100)

Chunk example (IV)

Repeat!

Cumulative feerates for this linearization, starting from the second chunk:

-	-	-	-	-	1	**1.07**	1.0625	1.060	1.057

Gives rise to this third chunk:

Tx A (100, 100)	Tx B (100, 100)	Tx C (300, 100)	Tx D (2000, 1000)	Tx E (500, 400)	Tx F (500, 500)	Tx G (1000, 900)	Tx H (200, 200)	Tx I (250, 240)	Tx J (100, 100)

And we can continue to the end of the linearization in the same way.

Last thing about chunks

The **linearization** algorithm does the hard work of figuring out how to order transactions in a way that maximizes feerate and respects the transaction graph topology.

Chunking is a post-processing step we perform that is independent of the linearization algorithm, to produce groups of transactions that will be our building blocks of mining and eviction.

Chunking is also something that runs in linear time, so this separation of concerns between linearizing a cluster and producing these chunks doesn't really cost anything from a complexity standpoint (linearization will be far more expensive).

Back to mining

Armed with linearizations of each cluster and the corresponding chunks for each cluster, our mining algorithm is now trivial:

1. Throw the first chunk from each cluster into a heap.
2. Remove the highest feerate element from the heap and add to block.
3. If the cluster from which the highest feerate element came has more chunks remaining, add its next best chunk to the heap and repeat until the block is full.

(There are some more details about what to do when the block approaches being full, and if a chunk doesn’t fit into a block.)

And now for eviction

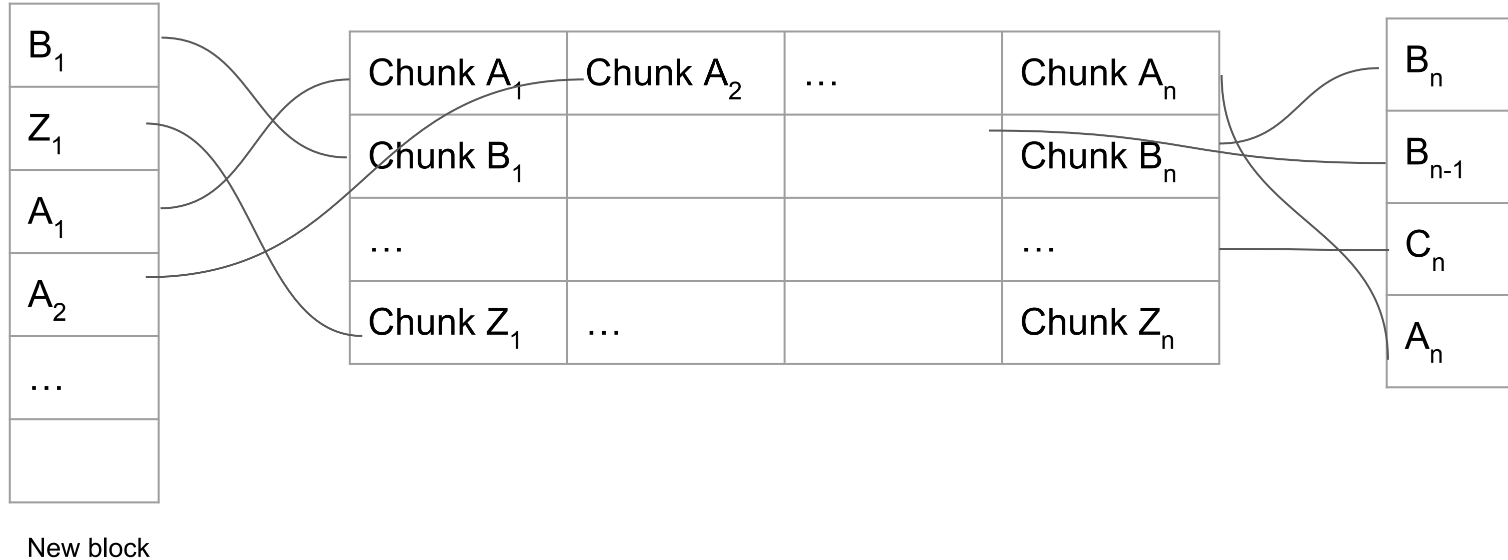
Eviction is exactly opposite of mining.

While the mempool is too big:

1. Throw the last chunk of each cluster into a heap.
2. Remove the lowest feerate element of the heap and evict those transactions from the mempool.
3. If the selected cluster in step 2 is not empty, select its next worst chunk and add to the heap.
4. Repeat until the mempool is small enough.

Claim: This process will evict the very last transactions that would be mined by the algorithm described on the last slide.

Mining versus eviction



RBF policy

This mempool design produces a total ordering of transactions. At any time, we can score a transaction's suitability for mining as exactly the **feerate of the chunk** to which it belongs.

So with this design, our RBF rules can be that a replacement transaction:

1. Would need a higher chunk feerate than the chunk feerates of every transaction that would be evicted.
2. Must pay a higher total fee than all transactions that would be evicted (anti-DoS).
3. We will also need to address CPU DoS with a limit on how many transactions can be evicted at once (as we have today), as evicted transactions can all be from different clusters, which will all need to be re-linearized when a replacement happens.

Revisiting our problems

1. Eviction
   a. No longer have an asymmetry between mining and eviction (so won't ever evict something that should be mined)
2. Mining
   a. Optimal mining may be possible, if we limit cluster sizes sufficiently (rough estimate: maybe 20 or 25 transactions can be optimally sorted in 10-15ms?)
3. RBF
   a. Incentive incompatible replacements can be eliminated without heuristics that would massively worsen pinning.
   b. Pinning – due to large, low feerate children – is still possible, as that arises from anti-free-relay DoS rule that requires new transactions to pay for all evicted transactions. Still need v3!

Open questions (I)

- Are we ok with an even-more-opaque “black box” that decides when an RBF can take place, or what the mining score of a transaction is?
    a. BIP 125 feerate rules would be going out the window
- Are users/developers ok with a cluster size limit at all?
    a. Currently there is no limit
    b. However the descendant count limit is sort of like a cluster size limit, in that it’s out of your control and can prevent you from creating a transaction spending unconfirmed coins.
    c. What range of values are users likely to accept?

Open questions (II)

- What do we do if a transaction comes in that violates the cluster size limit?
    a. Easiest thing is to just reject it. Figuring out which sibling to evict to make room seems hard.
    b. This will be a potentially new pinning vector – but not so different from descendant count limit either.
- Non-determinism and/or mixing of linearization algorithms:
    a. For performance reasons we may want to mix use of fast algorithms (eg ancestor-feerate-mining, or a bounded search) with slower algorithms (like higher bounded search or optimal search).
    b. You could imagine allowing users to tune these parameters too (how much to bound a bounded search, at what cluster size to switch from optimal search to ancestor-feerate-based sort, etc)
    c. This all could lead to non-deterministic behavior, both for a single node and for the network, when looking up a given transaction's chunk feerate – and therefore non-deterministic RBF behavior.
- Are there ways to relay linearizations and their chunks on the p2p network?
    a. Maybe we can distribute the work of calculating optimal sorts to the whole network someday.

Open questions (III)

- Chunk sizes should be bounded, but how?
  a. Need some bound for chunk sizes, because if they're the smallest thing we evict, then we can have a free relay problem if they get too big.
  b. Does this imply that we bound clusters to the same limit? Or can we devise a new way of thinking about clusters and their linearized chunks where we allow ourselves to bound the chunk sizes that we consider?
     i. This would imply that chunks within a cluster are no longer going to be in decreasing feerate order.
     ii. But maybe this doesn't matter, if we can reason about what the mining scores are for transactions in such "split chunks" anyway, somehow.

Further work to be done

1. First draft implementation in the Bitcoin Core codebase is essentially done.
    a. Just uses ancestor feerate mining of each cluster for now.
    b. Implements clustering/sorting/chunking, supports adding transactions during regular relay as well as from disconnected blocks, mining, eviction, RBF.
    c. Overall I think we have a decent handle on algorithmic complexity of our operations at this point.
    d. Few missing things (no cluster size limit yet, memory accounting, more tests, etc)
2. Simulate the effect of cluster size limits on historical transaction data, to see how much of an effect they would have had, were they implemented in the past.
3. Performance optimize and try to determine biggest cluster sizes we can tolerate (while managing worst-case behavior).
4. Consider implications for other projects including package validation/package relay and fee estimation.