

Libbitcoin Version 4 Node

Version 8

November 6, 2021

Summary

This is a major revision to the design of the libbitcoin-database and libbitcoin-blockchain, with some less significant changes to libbitcoin-network. There are no material impacts on other repositories. There are two major objectives in Version 4 node architecture.

The **parallel block download objective** aims for a significant reduction in bootstrap time. Two sequential chasers follow independent and concurrent downloading and validating operations, initiating validation and confirmation respectively. Validation beyond headers is bypassed by consensus checkpoints or (externally-validated) milestones.

Second, the **reliable shutdown objective** was significantly complicated by metadata updates, such as spend marking and state changes. This proposal limits store writes to either appended-only and small indexes that can be quickly copied. It also provides a simple and reliable design to acquiesce store write (for recovery checkpointing) and table read/write (for memory map expansion), as well as reliable shutdown termination.

Storage Tradeoffs

The storage tradeoffs are a very large new **spender hashmap**, a very small (typically empty) **candidate spender hashmap** moderate new **transaction and block confirmed hashmaps**, **block and transaction validated multimaps**, and a moderate net increase from factoring transactions into an **input hashmap and output hashmap**. Other storage increases arise from conversion to a transaction **association hashmap** (vs. array) and the use of an input/output **association array**. Each of these are offset somewhat by the elimination of updateable metadata.

Computational Tradeoffs

The main computational tradeoff is in the traversal of natural key hash tables to query information required for block/transaction download/storage and/or validation, and update of those tables. Those that accrue to client-server queries and are not material.

Advantages

Advantages are elimination of update code complexity and locking, opportunity to implement reliable shutdown, and improved performance of parallel download and of post-validation (especially under milestone) spend marking. Despite the addition of several new tables and the conversion of one to a hash table (and another from slab to record), code complexity is dramatically reduced. Each of the store tables are trivially implemented using existing database primitives. The change to append-only necessarily isolates neutrino and utreexo caching to independent files, no longer impacting header storage.

Candidate Chain

There is an in-memory header tree managed by the network that dynamically reorganizes itself against the candidate chain, ensuring that the candidate chain is the strongest visible chain competing with the confirmed chain. Only one very long competing chain is ever expected, as this can reasonably only be the result of an ongoing 50% mining conflict. For this reason the candidate index is maintained as a file system table and is always as long the confirmed index, allowing for efficient reorganization of any depth and supporting efficient process restart in the midst of a 50% conflict. The in-memory tree is therefore transitional and almost always empty.

The candidate chain introduces a second header index and independent metadata. Spends are independently tracked for the candidate chain from above the top common block (fork point). The fork point is determined at startup by scanning both height indexes for the first header foreign key deviation.

Any given output may be spent in both the candidate and confirmed chains (as well as by pooled transactions). Unconfirmed double spends are otherwise valid transactions. During candidate validation, any spend by a transaction in a block above the fork point is disregarded unless it is a candidate spend. As such, candidate spends must be retained in an independent spends hash table.

As candidate spends will reorganize to empty with organization of the candidate into the strong chain, the candidate spends table is typically empty. Candidate (unconfirmed) neutrino and utreexo records are not computed. These pertain to payment state, which requires at least one confirmation. These are created and pushed upon block confirmation (organization) and popped upon block reorganization.

A previously validated transaction remains valid under the same validation context (forks, minimum time, and minimum height). If context differs then the transaction is revalidated under the candidate context and both contexts are retained in the transaction validated table. This storage of more than one set of transaction validation flags within a multimap is made possible by removing the state from a fixed-width updateable transaction field. While the multimap is less efficient than a simple hash table, it is a tradeoff only to implement this potential for validation to be accelerated (in some cases).

Store

Archives

The **store** consists of archives, indexes, caches and buffers – collectively referred to as **tables**.

- An **archive table** stores primary blockchain data, sometimes augmented with 1:1 cached information that is available at the time the object is stored (e.g. block height and median time past).
- An **index table** provides reverse navigation across table relations (e.g. address to output, output to input, height to block, transaction to block).
- A **cache table** provides summary data derived from archives (e.g. block confirmation and validation state, transaction confirmation and validation state/context, neutrino proof, utreexo filter).
- A **buffer table** stores work for later execution (e.g. .transaction buffer).

Data Structures

The logical data structures persisted by the store are **array**, **hashmap** and **multimap**. Each uses a **key** to store and recover an **element**. Elements are typed as either **records** (fixed-size values) or **slabs** (variably-size values).

An element consist of a key, **link**, and **value** (which can contain multiple sub-values). Links are pointers implemented as file **offsets** (in bytes for slabs, element count for records). Value size (in bytes) is a record template parameter. Link size is a template parameter for records and slabs.

A **natural key** (NK) is a unique identifier for an element, integral to the value. A **foreign key** (FK) is a unique identifier for an element, private to the store instance. Search keys are used to locate an element via the interface, and may be either a natural key or a foreign key (of another table)¹. Elements links are used as foreign keys.

- An **array** is an ordered set of elements, each keyed by position within the array. Search is $O(1)$.
 - record type is a template parameter
- A **hashmap** is an unordered set of elements, each keyed by DJB2 hash. Search is $\sim O(1)$ ².
 - pointer size and element type are template parameters
 - privately maintains a linked list of elements per hash bucket (hash conflict list)³
 - interface can retrieve an element by its search key or its foreign key
 - interface exposes each element's search key and foreign key
 - storing a value with the same search key as one previously stored logically orphans the previous
 - orphaned elements remain reachable from foreign keys, ensuring data consistency

¹ Keying on the FK of table's own elements would be pointless, as search for a table's element is not required given the FK to that element.

² Hashmap searches can be as costly as $O(n)$, depending upon the configured header size (bucket count) and the evenness of the DJB2 hash distribution.

³ The search index is a size-configurable array of hash buckets, with each retaining an offset to the first element of the hash conflict list of elements. Given a single hash bucket, search would be $O(n)$ in the number of elements.

- A **multimap** is a hashmap that retains an ordered set of elements for a given search key. Search is $\sim O(1)$.⁴
 - maintains a value stack, implemented as a linked list of empty-valued elements (**entries**)
 - the hashmap element value of a multimap is an offset to the head of a multimap value stack
 - retrieval is an ordered traversal of entries within the stack of the hash table element
 - storing a value, with the same search key as previously stored, pushes it onto the value stack

All elements (and entries) are appended to the table and then added to the data structure's search index. In the case of an array, indexation is merely an increment (or decrement) to the record count property. In the case of hashmaps (and multimap stacks) indexation is a swap of the element's head value, with the offset of the new element. In this manner of insertion, all elements are logically stacked, and all writes (except to the search index) append the table.

The element stack of a multimap is implemented in the same manner. However, in order to append all new entries, a new hashmap element is appended for each new multimap entry.⁵ The previous multimap stack head element is orphaned from the hashmap and its first entry is linked from the new stack entry. In this case append comes at the cost of orphaning one empty-valued element (a key and a link) for every multimap value stored after the first.⁶

Because writing a new element does not disrupt any existing structure, the table remains both readable and writable during a write. There are two atomicity guards in place. Allocation of new space (an addition operation to increment the logical file size, by the underlying memory manager) and swap of the conflict list head pointer (i.e. the bucket value).

These are implemented as thread safe critical sections and are of course necessary to implement atomic swaps. Because we use upgrade mutexes, reads of the index are only blocked during the write of the swap. Contending writes are blocked slightly longer, as they must wait to read what is being modified. Note that these have no impact on writing the element, these are limited to indexation. Nor is logical update of the multimap value stack affected, as it is just a new element in a hashmap, pointed to the previous head. Writes proceed concurrently, only contending for position, as the stack is an ordered shared resource.

Enumerability

All tables are enumerable. This allows for full table scans at startup, used for:

- candidate fork point identification
- parallel block download gap population
- transaction pooled index population

⁴ Multimaps don't yet support slab elements, though this is not currently required.

⁵ Multimap space efficiency could be improved (and complexity reduced) at the cost of search performance by using the multimap's hashmap conflict list to store each value, but probably not worth the cost.

⁶ This is a simple (as well as simplifying and contention-reducing) design change to change to append-only multimaps.

- transaction pool graph population

Each table interface exposes an enumerator. Tables are enumerable stack write order (bottom up, oldest first).⁷

Files

Each type of table consists of a **header** and a **body**, where the header is an index of the body. Header and body are persisted in independent operating system **files**, identified and associated to each other by naming convention.

- Headers
 - All table headers contain the logical body size (in bytes or elements).
 - Hashmap headers contain a fixed bucket count and array of buckets.
 - Multimap headers are hash table headers.
- Bodies
 - Array bodies contain only values.
 - Hashmap bodies contain only keys, pointers, and values.
 - Multimap bodies are hashmap bodies with a pointer per value.

This file system split between header and body exists because headers are small and randomly written, while hashmap bodies are large and append only. A consistent set of copied headers and two array tables (small files in relative terms) makes up a checkpoint for failure recovery, with no touch to bodies whatsoever (more below).

Memory Maps

Each file is memory mapped. The file is paged in and out of available contiguous virtual memory by the operating system. This provides a simple and independent contiguous memory space in which each data structure operates. With sufficient resources the operating system may end up load the entire store into RAM. Otherwise it retains the pages of memory that are accessed most frequently.

Optimization of the tradeoff between fast RAM and slower disk is left entirely to the operating system, as that is one of its primary functions. As a scalable implementation, the store makes optimal use of all resources offered to it by the operating system while allowing the operating system, and thereby the person who runs the process, to manage resource consumption tradeoffs.

To avoid unnecessary memory map resizing while not wasting resources, each file grows by an independently configurable rate and starts with an independently configurable size. With sufficient sizing all resizing can be avoided. To avoid a failure due to insufficient disk space, files can be preconfigured to exceed the known size of store files. Otherwise the process will self-terminate due to a resize failure when it occurs.

During a resize the memory map may need to be moved (remapped) by the operating system, to obtain sufficient contiguous space. The database manages this even by suspending reads and writes against the file, then allowing the remap to proceed. This (slightly) pauses operations. This can be avoided entirely by proper initial sizing or can be minimized by higher growth rates, though this will consume memory

⁷ Multimap value stacks are enumerable in reverse write order (top down, newest first).

unnecessarily. Proper initial sizing with slower growth rates is generally preferred for both performance and avoidance of a late stage disk full error.

Keying

All data that is identifiable with a natural key is indexed to that key. The foreign key to that data can be obtained via the data structure interface and is used as the key in all other queries. Archive data is written (including its natural key) first, and then indexes and caches are related to it through the foreign key. This ensures that all data is navigable only after it is indexed by a natural key, while dramatically reducing storage space over comprehensive use of natural keys. Hashmaps are keyed by either the natural key of the data or the hash of a foreign key of relating data.

Append Only

All data structures are implemented as memory stacks, and so can be popped until logically empty. However we do not actually reduce the size of the file when popping, with the exception of arrays (and we only pop from the two height arrays). Pushing a new element of the same key to a hashmap logically hides the preceding element from search, though upon pop the preceding element is restored. Popping reverts the hashmap head pointer to the previous element of the hash conflict list, but does not move the end of file pointer, so the popped memory becomes permanently orphaned (never overwritten).⁸

All hashmap headers are stored in independent memory maps. This allows all data structures with the exception of arrays to be append-only in their independent memory maps. This allows the entire store to be checkpointed (more below) by copying only headers and the two array tables. This provides for full recovery to the last checkpoint in the case of an uncontrolled process termination.

- **Arrays** do not exhibit orphaning, so while they are stacks, they are also fully mutable. Popped arrays are limited to height indexes and are very small. As such checkpointing simply copies them entirely. Alternatively these may be regenerated.
- Given the orphaning behavior of **hashmaps**, a pop does not recover space, but it avoids wasting space by pushing a competing value to hide the preceding value. However parallel operations preclude popping all but confirmed/candidate height.

Multimaps are used in such a manner that there is never a reason to pop a value. Instead of popping reorganized (but useful data) it is retained, relying on independent confirmation and validation state tables to determine whether it is within the strong chain, the transaction pool, a weak block, invalid, etc.⁹

Link Optimization

Hashmaps require one link per element. Multimaps are hashmaps that require an additional link per entry. Link size is consistent between the multimap and its hashmap. This is bit overkill as the multimap link must index only the count of its (fixed size) entries, not the entire table. This can be optimized by decoupling.

⁸ This requires a trivial modification to not reduce the end of file pointer on pop.

⁹ Given the hiding behavior of the multimap's hashmap, and the creation of a new element for each entry in the value stack, pop of a multimap entry can be achieved by simply popping hashmap. This is a small change to existing implementation, though multimap pop is not a current scenario.

Slabs are linked by byte position, while records are linked by record count. Any link size between 1 and 8 bytes is supported.¹⁰ Link sizes considered below are either 4 or 8 bytes. The presumption is that a count of outputs or inputs requires 8 bytes, that all other counts require 4 bytes, and that all slabs require 8 bytes. These should be resized using rational estimates based on blockchain inherent limits and growth.¹¹

Tables support element links of any byte size. Minimizing these reduces FK storage in related tables. Links are deserialized into uint16_t (1-2), uint32_t (3-4) or uint64_t (5-8) in computational use. Link sizes and types are table template arguments.

Element	Type	Link Bytes	Capacity	Current
• Point Index ¹²	value	2	65,536 ins or outs per tx	capped by block limit
• Header/Block	record	3	16,777,216 blocks	708,277
• Transaction	record	4	4,294,967,296 txs	683,832,665
• Input/Output	slab	5	1TB blockchain bytes	250GB total

Hashmap Optimization

Hashmap headers are created from a configurable bucket count. More buckets (sparse table) requires more header storage space, while fewer buckets increases hash collision. These should be configured for current and anticipated sizing based on a review of observed collision rates. Ideally an average of 1.5 collisions is reasonable. Multimap value stack search is always linear in the number of entries per element, though this is expected to always be very low (typically 1 or 2, maybe 3 entries).

State Table Compression

Two tables regularly orphan states. These can be removed by rewriting the tables using a compression function that loads the table and writes all logical entries to a new one using the table enumerator.¹³ This could be integrated into startup or the checkpointing phases of operation and/or provided as a command line option.

Data Consistency

All query through the store begins with natural keys. These are exposed by:

- confirmed table (block height)
- header table (block hash)
- transaction table (transaction hash)
- outputs table (output point)

¹⁰ The 8 byte limit allows classes to hold this in a uint64_t, while this could be expanded using uint<T> , this is a long time away for Bitcoin.

¹¹ Some minor work in the database primitives is required in order to generalize link sizing.

¹² Point index is size-constrained by the block size limit. It affects search key sizing but it is not used in linkage.

¹³ Chronological sort of most tables is also an option.

- short id table (compact tx hash)
- address table (output script hash)

Naturally-keyed entries are not stored until after all foreign keys navigable from that entry are stored. This ensures that all obtainable links remain navigable at all times.

Inputs and outputs are reachable by navigation through the transaction. Outputs are also independently accessible through a natural key, but only because of a significant parallel block download (under milestone) optimization that requires their natural key indexation. Short IDs are computed from transaction hash. Neutrino and Utreexo entries are reachable by first obtaining the block's foreign key from the header table.

There is no update to transaction or block confirmation state except by the blockchain validator, which operates on strands. So the validator will always see a consistent state. Independent queries across tables may see inconsistent results during a reorganization (e.g. a confirmed transaction in a subsequently unconfirmed block). This is perfectly rational behavior.

Repair

There is no repair mode. Unlike most databases, the store cannot become corrupted by process termination.

Checkpoints are atomically swapped by the operating system, and the operating system manages all file I/O through memory mapping. The only possible corruption is an application error, an operating system error, or a hardware failure. Operating systems and hardware go to great length to preclude the possibility of such errors. Detecting such an error would require more time than regenerating the store, which is after all just a cache of public data. And of course if there is a code error, the same is true.

The best check of the store is that the chain proceeds as expected. Scanning the last few blocks, for example, would be a pointless exercise, as there is no way to determine store validity without (at least) revalidating the entire chain. However resyncing the chain is faster. If an error was detected it would be the consequence of one of these above three classes of errors. In that case the platform is unreliable and repair remains pointless.

Store Tables

ARCHIVES

Archives are a cache of public data obtained from the Bitcoin P2P network.

Each **block/header requires 125 bytes**, consisting of 100 for structure, 12 for (empty) transactions association, 8 for height indexation (block flags storage is not required), and 5 for confirmation marking (if confirmed). Each **transaction requires 97 bytes**, 73 for structure, 8 for block association (if associated), and 16 for confirmation (if confirmed). Each **output requires 59 bytes** and each **input requires 62 bytes**, (including transaction associations). This assumes empty inputs and output scripts. Input and output scripts can be pruned to any “reorg safe” height, along with validation and spend tables, dramatically reducing the storage required for a pruned node. All other tables are optional, for either wallet services or mining performance.

“Header” Hashmap Archive [Block:96, LINK:4]

header NK [32] => context[flags [4], height [4], median-time-past [4]], parent FK [4], header [48]

Context provides the parameters used to validate the block, obtained from chain state during header validation. The block may later be declared invalid (by transaction content), so block validation state is stored independently.

“TXS” Association Multimap Archive [BLOCK:8+TX:4, LINK:4]

header FK [4] => transaction FK(s) [4]

This is header-indexed because the header cannot be updated as transactions are later populated. As the header does not contain transaction count, and the association table is an array, the first referenced element contains the transaction count.

“Transaction” Hashmap Archive [TX:69, LINK:4]

transaction NK [32] => out-count [2], outputs FK [8], in-count [2], inputs FK [8], locktime [4], version [4], size [4], weight [4], coinbase [1]

- **Size** zero indicates that size and weight are not set (not required under milestone).¹⁴
- **Weight** is witness-included size (zero is the sentinel for non-witness transaction).
- **Coinbase** is a binary state, derived from the block context (or tx with null inputs).

Transaction wire serialization is suboptimal in that it places outputs behind variable-length inputs, and because location of an input or output requires skipping through all prior by reading byte prefixes. Even with store serialization of outputs before inputs, outputs must be tediously walked across the preceding variably-sized output scripts. Storage of a redundant cache of unspent outputs (UTXOs) is wasteful for a full node, which already retains all outputs (within transactions). As such we factor inputs and outputs into independent slab hashmaps. This allows the transaction table to be a record hashmap.

“Output” Slab Hashmap Archive [OUTPUT:43+SCRIPT, LINK:8]

output NK [32, 2] => value [8], length [1-9], script [var]

¹⁴ Size can be obtained for free by storing it on the message object, from the buffer length.

This table is written by any valid output. As inputs under milestone are written as they arrive, they must reference outputs that do not yet exist. So to achieve this optimization the output must be indexed (and referenced by input and spend tables) using its natural key.¹⁵ This would not be necessary for blocks validated in order, all spends (and inputs) could use output foreign keys. See spent output table for optimized mining index.

Isolating prevouts to their own table significantly reduces the need for paging, and may allow this validation-critical table to be preloaded entirely into RAM.

“Input” Slab Hashmap Archive [OUTPUT:46+SCRIPT+WITNESS, LINK:8]

input FP [4, 2] => **output NK** [32, 2], **sequence** [4], **length** [1-9], **script** [var], **count** [1-9], **witness** [var]

A **foreign point (FP)** is a hybrid of foreign and natural key, used to reduce the size of input key storage. First write transaction outputs. Then open a writer for the transaction to obtain its FK but don't link it, then write all inputs, with input keys derived from the transaction FK and input index, then write all associations, and then the transaction. This ensures the inputs are associated when the transaction becomes searchable, while allowing inputs to use 6 vs 34 byte keys.

Use of the output natural key here is a significant performance-improving and space-increasing denormalization, as it allows transactions to be written in parallel without table updates.

“Puts” Association Array Archive [TX:0+PUT:8, LINK:0]

puts FK(s) [8]

Because this table is only useful to a transaction, and it can only be written at the time the associated values are available, it is not indexed and therefore an array, not a hash table. Because the transaction retains the input and output count values, no count or end sentinel is used here. These are differences from the transaction association hash table. The array is never popped, so remains append only.

¹⁵ This sync optimization requires extra storage of the natural output point, once for each input and output. Natively it is a transaction hash (no position value) stored once per input. If not for the sync optimization the database could use only foreign keys/points, not storing any natural points.

Indexes

Indexes provide reverse navigation through tables, for the purpose of accelerating query.

“Candidate Height” Array Dynamic Index (startup & validation) [BLOCK:4, LINK:0]

height NK [0] . header FK [4]

As a dynamic array this file cannot be append-only.

“Confirmed Height” Array Dynamic Index (startup & wallet) [BLOCK:4, LINK:0]

height NK [0] . header FK [4]

As a dynamic array this file cannot be append-only.

“Spent Output” Multimap Index (validation & wallet) [SPEND:42+DOUBLESP:42, LINK:8+8]

output NK [32, 2] ==> input FK [8]

This provides reverse navigation from outputs to their spending inputs, and thereby the spending transaction and confirmation state. This is written for every input in the checked state, often before the output is archived. As multiple inputs may spend an output (only one confirmed), the table is a multimap. Confirmation state is obtained by querying the transaction confirmed index with the input FP’s transaction FK.

“Input Tx” Hashmap Index (validation) [OUTPUT:20, LINK:4]

input FK [8] => parent FK [4], output FK [8]

This optimizes the determination of confirmed spentness by reducing it to a single search by input FK. Otherwise the spend must be searched by output NK, its spending input searched by FK, and the input parent confirmed height searched by FK. This table is populated by validation, not by download or confirmation. This contains only validated spends as it is written upon accept of the input’s transaction (first point where the prevout must exist).

“Output Tx” Hashmap Index (validation) [OUTPUT:12, LINK:4]

output FK [8] => parent FK [4]

This optimizes the determination of output confirmation by reducing it to a single search by output FK (already obtained from the “input transaction and output” table while validating spends). Otherwise the determination must use the spending input point’s transaction hash (transaction NK) to search for transaction FK and then its confirmation state searched by FK. This is written at the time the parent transaction and its outputs are archived.

“Compact” Hashmap Index (validation) [TX:10, LINK:4]

compact id NK [6] ==> transaction FK [4]

This is an index of pooled transactions, retrievable by short hash (compact blocks short transaction identifier). The table is populated from transactions that enter the transaction pool, either from network broadcast, deorganization, or startup population. This allows for navigation to any transaction available for compact block inclusion. If not found in this table the transaction will be requested by its short ID from the network, and upon retrieval enter into the pool for recovery by the compact blocks protocol.

The table can be configured to populate at startup and/or to delete after shutdown, and the two files may be deleted manually when the process is stopped.

“Address” Multimap Index (wallet) [OUTPUT-SCRIPT:40+SCRIPTREUSE:8, LINK:8+8]

address NK [32] ==> output FK(s) [8]

Addresses (output script hash, SHA256 - Electrum) of all valid transactions are indexed. The spending input point(s) of an output point can be obtained by querying the spender index with an output point. Confirmation state is obtained by querying the confirmed index with the hash of either the output point or input point(s), as required. As an address may be declared by multiple outputs, the table is a multimap.

This table uses 8 byte links despite not being a slab, which can be reduced based on expected limits. This will also reduce storage in other tables that hold FKs to it. Update orphaning costs 32 bytes plus an 8 byte internal link. So we waste 40 bytes for each address reuse :<.

“Confirmed TX” Hashmap Index (startup, Validation & wallet) [TX:12, LINK:4]

transaction FK [4] => header FK [4], height [4]

This provides reverse navigation from a transaction to its confirming block. A sentinel is pushed when a transaction becomes deconfirmed.

The block FK would be sufficient to obtain **height**, however caching it preempts navigation to the header table and confirmed chain index during spender validation. Height could be provided for wallet queries, but this presents a single query race in determining block inclusion. For these reasons both are included. Transactions are not validated for double spending, as this is valid for a transaction (blocks impose order). As such the validation of prevout spender height is only performed at block confirmation time, by navigating from an archived transaction (facilitating FK traversal).

As a single map hash table, a sentinel is pushed on deorganization, allowing append-only file storage.¹⁶

“Confirmed Block” Hashmap Index (startup & wallet) [BLOCK:8, LINK:4]

header FK [4] => height [4]

This table stores height so that a spent output check during validation can determine whether the confirmed spend is above the fork point (last common block). If this is the case the output is considered unspent.¹⁷

¹⁶ This would support navigation to reorganized blocks and even weak blocks if it was converted to a multimap. A reorg would create a weak block from a strong block, and a never-confirmed weak block could be stored with sentinel.

¹⁷ This is labeled an index because it can technically be used to navigate to the block (though with a race), but also for consistency with the “Tx Confirmed” index.

Caches

Caches provide information derived from tables, for the purpose of accelerating computation. Caches may be independently deleted without loss of store integrity.

“Validated TX” Multimap Cache (validation & wallet) [TX:32+REVALIDATION:28, LINK:4+4]

A validation state is required as the transaction may be found invalid after evaluation with its previous outputs.

transaction FK [4] ==> context[flags [4], height [4], median-time-past [4]], code [4], sigops [4], fee [8]

This associated metadata is used by block validation, to obtain previous validation flags and related state, so that the block can validate without loading the transaction. As a transaction may be validated in multiple contexts (fork flags, MTP, height), the table is a multimap. Block validation compares context to obtain the correct values. This largish table has zero (under milestone) to two (connected/invalid, and sometimes preconnected) entries per transaction in a given context.

These values are required for block inclusion:

- **Sigops** includes input and prevout sigops, the count is affected by flags.
- **Fee** requires prevouts and is cached for population of the transaction graph.
- **Final** is a binary summary of input finality based on time/height (code reflects error::not_final).
- Locktime and non-witness transaction hash are native to the transaction archive.
- Coinbase, weight and size are cached by the transaction archive.

Finality does not need to be evaluated, as it invalidates the transaction in the context, and therefore the block.

States (applicable only to the associated context) and their codes:

- | | | |
|------------------------|--|--------------------|
| • Preconnected: | error::preconnected | Non-terminal state |
| • Connected: | error::success (or no entry and under milestone) | Terminal State |
| • Invalid | any other error code | Terminal State |

No entry will be present when the transaction is obtained under milestone. Context under milestone is singular, so not required. The transaction is presumed to be checked and accepted/connected. Coinbase transactions follow the same state rules despite lack of connectivity (null inputs). Connected implies script validation, but not a lack of external double spend, which is only relevant to block confirmation (blocks impose order). No entry will be present following download above a milestone but before the accept/connect phase. The lack of an entry for the context, above a milestone, implies that the transaction is in the checked state.

Transactions are associated to their previous outputs out of order (via the input point, an output NK). This association is indexed for reverse navigation from the output in the “output spent” table. Any block-associated transaction may be found to be missing previous outputs after archival of all preceding transactions in the candidate chain. As the previous output may in fact exist (i.e. outside of candidate-included ancestors), and become archived in the future, the transaction remains in the checked state if one is missing at the time of transaction accept/connect evaluation. The transaction is at that point neither connected nor invalidated, and candidate chain PoW guards against DoS.

Cost of confirmability is the pool DoS guard. Pool acceptance requires connection to other transactions that are either currently confirmed or connected in pool context. Double spending a confirmed transaction is also criteria for pool exclusion. Any transaction accepted via the pool is therefore in the “connected” state.

It is also possible that a block-associated transaction may connect to an output in a circular manner or connect to the output of an invalid transaction (directly or transitively). The transaction pool graph precludes this (as the transaction is not confirmable), but an invalid block (DoS protected by PoW) may incorporate such a transaction. A **connected** transaction may be either in competition with another for an output or may form a circular reference. Both are precluded from block inclusion (consensus) and guarded against DoS. Server informational query of transactions can be restricted to confirmed (by reverse index) and confirmable (pool graph) as desired. This reflects expected behavior, while more specific query can also be exposed.

Only in the case where all previous outputs do exist, and the transaction fails to accept/connect, is the transaction marked as **invalid** (in the validation context). This state preempts validation when associated by another block (of a consistent context). Invalidation is also useful for server informational queries. Transactions are never declared valid as connectedness is sufficient for inclusion, with double-spending verified at the time of block confirmation. Double-spending is block-contextual and otherwise perfectly acceptable. Ordering transactions to resolve double spends is the reason that blocks exist. Double-spend is the property of a set of transactions, not any individual.

The **preconnected** state indicates that the transaction was successfully preconnected in the block check phase. No entry is written when a precheck evaluation is unsuccessful.

“Validated Block” Hashmap Cache (validation & wallet) [BLOCK:8, LINK:4]
header FK [4] => code [4], fees [8]

A validation state is required as the block may be found invalid after association of its transactions. Headers are associated to their transactions in the case where all transactions are in the checked state (exist in the store). The association itself is the “checked” state for the block, which is independent of that for headers and transactions. This small table has zero (under milestone) to two (connected and then valid/invalid) entries per confirmed block.

States (applicable only to the only context) and their codes:

- **Connected:** error::connected (or no entry and under milestone) Non-terminal State
- **Invalid** any other error code Terminal State
- **Valid:** error::success (or no entry and under milestone) Terminal State

The **connected** state applies to all inputs all of the block’s transactions only. It implies that all ancestor blocks are checked, and therefore all ancestor transactions as well. While it implies that all of the block’s transactions are connected, ancestor transactions may not be.

The **valid** state implies that the block and all of its transactions are valid, and may be confirmed if it is in the strongest branch. This state exists to preclude revalidation of spends for blocks that are confirmed, deconfirmed, and then reconfirmed. This may be otherwise costly in the midst of a 50% conflict.

The **invalid** state preempts revalidating a reverted invalidating candidate header chain. It is also useful for server informational queries. As such when a candidate block is invalidated, all subsequent blocks in the candidate branch are invalidated.

No entry will be present when the transaction is obtained under milestone. Given that block context is inherent and immutable outside of breaking confirmation changes, this table is not a multimap.

Fees are computed from cached transaction validations, as required during block validation, and preserved for fee historical estimation over different periods of time. Current fee estimation is based on block template computation.

“Neutrino” Slab Hashmap Cache (wallet) [BLOCK:37+FILTER, LINK:8+8]

header FK [4] => **filter-header** [32], **length** [1-9], **filter** [var]

“Utreexo” Slab Hashmap Cache (wallet) [BLOCK:5+PROOF, LINK:8+8]

header FK [4] => **length** [1-9], **proof** [var]

A utreexo proof is logarithmic in the size of the outstanding unspent outputs at a given block.

“Bootstrap” Array Cache (Startup) [BLOCK:32, LINK:0]

height [0] . **hash** [32]

This table requires either zero or 9,440,000 bytes (for the BTC chain).

Buffers

“Buffer” Slab Hashmap Buffer

transaction FK [4] => **length** [1-9], **transaction** [var]

This table accepts cache-overflowing transactions (excludes those that are successfully preconnected) streamed directly from their message buffer following transaction archival, for later retrieval during transaction accept/connect. The stream buffer is then purged and the transaction is submitted to the in-memory cache (when overflowing). A cache miss results in a stream deserialization from this table. Read-ahead population of the cache from this table utilizes I/O resources in the background while the CPU is busy validating, with the objective of ensuring that the validator is never starved for transactions. This also allows the transaction cache to span restarts. Transactions are not deleted, as no space would be recovered. The file can be deleted once the chain becomes current enough to no longer overflow the memory cache.

Validation

State Machines

Libbitcoin Blockchain manages the transitions of three finite state machines for managing the Bitcoin blockchain.

Optimization goals are, in order of priority:

1. Denial of service protection
2. Node restartability
3. Block template deduction speed
4. Node bootstrap speed
5. Server query data consistency
6. Server query response time
7. Storage efficiency

The state machines are:

- Archival FSM
- Validation FSM
- Confirmation FSM

The triggering events are:

- Network Events start the Archival FSM.
- Checked Events from the Archival FSM start the Validation FSM.
- Connected Events from the Validation FSM start the Confirmation FSM.

States are cumulative across machines:

- **Connected** or **invalid** state refines **checked** transaction.
- **Connected**, **valid** or **invalid** state refines **checked** block.
- **Confirmed** state refines **valid** block.
- **Candidate** state refines **checked** header and the header's block (if it exists).

States are dependent across object types as follows:

- All three objects (block, header transaction) have distinct archival state.
- Transaction and block have distinct validation state (with transaction contextual).
- Confirmation state applies to a header and its block (if it exists) as a single object.

Some transition rules:

- **Checked** terminal – the object never leaves the store.
- **Connected** implies locally valid only (an ancestor may be invalid).
- **Valid** block (and tx by association) is terminal and implies confirmable (not necessarily confirmed).
- **Invalid** is terminal and implies not confirmable in the context.
- **Candidate** (true) transitions only to **Confirmed** or false.

- **Confirmed** (true) transitions only to false.

Header Validation

Header Check (header or compact message)

Header **check** is called by network protocol that obtains header messages. This is a minimal check to preclude pushing work to the chaser that can be validated without context.

Given a downloaded header the following context free checks are performed:

- **Invalid proof of work** (context free)
- **Futuristic timestamp** (requires clock time)

This is the only point where clock time is used in validation. All other context necessary to confirm blocks is from chain history (i.e. context).

Upon successful completion, the header is passed to the headers chaser in the “header checked” signal, otherwise the header is discarded.

The header is not archived until it can be organized into a branch stronger than the candidate branch (DoS).

Header Connect

Header connect is based on existence of the header’s parent block, as determined by the header chaser. If the header does not connect it is discarded, otherwise the chaser then synchronously invokes header accept.

Header Accept

While accept checks could be performed by the network, pushing them to the header chaser isolates chain state the blockchain. This allows the network library protocols to check blockchain messages and hand them off . An invalid **block** acts as a negative checkpoint. Archived headers alone cannot be invalidated (required by DoS).

The following contextual (**accept**) checks are performed on the connected header:

- **Checkpoint conflict** (requires connected height and the set of checkpoints)
- **Insufficient proof of work** (requires flags and work required - accumulated by chain state)

If the header is above the last checkpoint, the following checks are also performed:

- **Version below minimum** (requires flags)
- **Timestamp not above median time past** (requires MTP)

Milestones are used against a candidate branch, to bypass block and transaction validation up to the top milestone contained within the branch. However if the header hash **matches** a configured milestone (or checkpoint), the individual header validation (both check and accept) can be bypassed as well. With a contiguous set of checkpoint (and/or milestone) hashes all accept checks would be bypassed. Otherwise PoW sufficiency must be verified as a DoS measure within the gaps. The connect check is always necessary to ensure that the parent is present.

Block Validation

Block Check (full or compact)

The network submits a deserialized **block object**. Header check/accept is already complete at this point. The block header archive is queried for the header FK for txs association enumeration. Each transaction's archival state is then queried. If the transaction exists, its validation state for the context is queried. If any transaction is invalid, the block is invalidated. If the transaction is connected or preconnected, preconnect is bypassed. Block check (below) is then executed. Each transaction that passes its **check** phase is archived and if all checked, the set of transactions is associated to the block's header. The result code is returned by completion handler.

If the preconnect method of any transaction object returns true, a "preconnected" value is written with the transaction validated entry. **If there is a block message we are optimizing for bootstrap, not top block(s) validation speed.** This implies full accept/connect execution during this check phase, with a simple output compare for later verification in the blocking accept/connect phase (where the actual prevouts are available). Preconnect jobs are posted asynchronously by transaction, see block accept/connect.

Compact blocks validation does not construct a block object or perform preconnect, it instead builds on pooled transactions. Given a (filled) compact block, these **consensus** checks¹⁸ are instead evaluated:

- Block empty (requires transaction count [this is just error reporting, coinbase checks catch this])
- **Block size limit** (requires transactions' byte sizes)
- **First transaction not coinbase** (requires transactions' coinbase state)]
- **Invalid coinbase script** (requires first coinbase input script)
- **Extra coinbases** (requires transaction coinbase states)
- **Invalid coinbase commitment** (requires coinbase first input script and last output commit script)
- **Internal forward reference**¹⁹ (requires all tx hashes and all inputs' points)
- **Internal double spend**²⁰ (requires all inputs' points)
- **Invalid merkle root** (requires all transaction hashes)

Weight and finality checks are contextual, so the required values cannot be cached in the transaction archive. But context is always available to the node, so they could be performed here. They are instead deferred to the accept phase so that transaction values can be computed and cached during validation, which uses a deserialized object.

Upon completion, the block's transactions are memory-cached and "block checked" is signaled.

These checks are observed within the `chain::block.check()` method.²¹ These checks are bypassed under milestone.

¹⁸ The duplicate transaction check is redundant with internal double spend, so we don't do it.

¹⁹ This is a side effect of Satoshi's ordered implementation, no other reason to do it.

²⁰ This short-circuits later spend invalidity and also guards against DoS (<https://bitcointalk.org/?topic=102395>).

²¹ This code will be factored into static methods and invoked both from blockchain validation and when validating from within a `chain::block` instance, in order to ensure there is a single implementation.

Block Accept/Connect

Block **accept** deserializes each connected transaction (unless found in the transaction cache) posts a transaction accept/connect validation job to the database threadpool for each transaction that is not in the connected state. As preconnected transactions must be postconnected, it remains ideal distribute the prevout query cost to the posted accept jobs, and simplifies implementation to implement this in the same handler. Any non-connected transaction invalidates the block, although other transactions can either be posted or not.²² The result is written to the block validated table. Nothing is returned to the validation chaser and it provides no completion handler.

After transaction accept/connect completes, block accept is invoked. This gives block accept the necessary transaction metadata values to full validate, populated while asynchronously accepting/connecting each transaction. By capturing a common shared pointer in the handler, completion for all transactions can be indicated by pointer destruct, which invokes these block accept checks.

Given all connected (or valid) transactions, these consensus checks are evaluated:²³

- **Weight limit** (requires all transactions' sizes and weights [**consensus**, static])
- **Coinbase overclaim** (requires all transactions' fees and the coinbase spend [**consensus**])
- **Signature operations limit** (requires transactions' inputs' and prevouts' sigops counts [**consensus**])

Context is obtained from the network header protocol, as derived from the candidate header chain. These checks also rely on metadata cached in the transaction validated table.²⁴ These checks are observed within the `chain::block.accept(context)` method.²⁵ This phase is bypassed under milestone.

Upon completion, a “connected” state is written to the block validated table, and “block connected” is signaled.²⁶

The **transaction cache** is the last ditch optimization for parallel block download, as we want to avoid ever reading a transaction that we just have archived. This is unavoidable if the node is stopped during bootstrap (or catch-up). In that case we are bootstrapping from the store. The archive tables are the primary download cache, this is just an option to fully scale by utilizing available resources. With sufficient RAM we can hold all transactions in the cache, though not an option for lighter nodes.

²² Accepting transactions here would allow the connected ones to become included in the pool.

²³ These checks may be performed concurrently.

²⁴ The first two checks could be executed in the block check phase. However it is CPU efficient to allow transaction validation to cache results from a deserialized transaction.

²⁵ This code will be factored into static methods and invoked both from blockchain validation and when validating from within a `chain::block` instance, in order to ensure there is a single implementation.

²⁶ Given that concurrent transaction validation spans blocks, it must be ensured that transactions that spend an output of an invalidated transaction are not prematurely considered valid. This is only a consideration for server queries, as an invalid ancestor inherently precludes descendent confirmation. Therefore, successful transaction validation writes an “connected” state to the transaction validated table.

Block Connect

Transaction accept completes its accept and connect phases, which is why there is no distinct block connect phase. The block connect phase logically consists of calling `transaction.connect(context)` for all of its transactions. These checks are observed within the chain:: `block.connect(context)` method.

Coinbase Considerations

Both phases require the coinbase, however this is always available given that all transactions are populated.²⁷ Given that the coinbase is essentially its own cache, and directly navigable from the already-necessary list of transaction foreign keys, it is not an optimization to cache any data from the coinbase. Furthermore the more complex coinbase checks are performed in the check phase, which is non-blocking (proceeds immediately upon population from block download or compact block protocol completion).

The coinbase claim check is blocking, however this requires only a summation of coinbase output values. We optimize this by caching the total spend value (8 bytes) of each transaction. This cache is more critical for graph initialization. Navigation to the output values of a coinbase requires only following links (no hash table searches).

Other Information

A block is never deserialized from the store for validation. Only transactions above a milestone, not cached, and not previously validated in the current context are deserialized. Pool transactions are validated in context of top block flags, median time past, and ++height. Compact block requested transactions enter the store as pool transactions, with compact block completion monitored and signaled by its network protocol. Milestone blocks and their transactions are not validated (just their headers), so don't require metadata computation.

²⁷ The coinbase is provided by the network in both full and compact block messages.

Transaction Validation

Transaction Check

A transaction message is obtained by the network and deserialized to a transaction object (if it does not already exist in the store). The object's **check** method is executed on the database threadpool, with context. Pool vs. compact status is provided by the caller and passed to accept. This status is obtained by the incoming transaction protocol, which synchronizes with the compact blocks protocol on a given channel.

Context free DoS checks:

- **Empty transaction** [redundant with more specific accept check, but trivial so short-circuit]
- **Null non-coinbase input** [redundant with missing prevout accept check]²⁸
- **Spend overflow** [redundant with overspend accept check]²⁹
- **Internal double spend** [redundant with necessary block check]
- **Coinbase transaction** [exclude coinbase from pool]³⁰

Upon successful completion, the accept phase is synchronously invoked, otherwise the transaction is discarded.

These checks are observed within the chain::transaction.check() method. These checks are bypassed for all full/ compact block transactions.³¹

²⁸ This is DoS protection given that the segwit coinbase commits to the merkle root, and given that another's coinbase would presumably never be used in a new block. So without it they would just accumulate at no cost.

²⁹ **TODO**: verify that this does not guard the overspend consensus check implementation against math overflow, so it can be safely skipped on block check.

³⁰ This is deceptive as a result of the segwit change to serialization. A transaction with no inputs (according to prior serialization) is converted to a transaction with inputs. With segwit active, the zero input transaction will have been deserialized into a transaction with inputs by this point. This is a change to the P2P protocol as a consequence of chain history (activation). We implement this in transaction deserialization, making this check context free. This is necessary as the transaction's consensus conformance is independent of its wire serialization. Necessarily, sighash serialization is also altered by segwit, as legacy hash computation would not include the witness (or be valid, given no inputs).

³¹ Given that we have a strong header PoW for full/compact block transactions, these DoS checks are unnecessary.

Transaction Accept/Connect

Transaction **accept** is invoked asynchronously from **block accept**, or synchronously by **transaction check**, which provide pool/compact/full status and context. A deserialized transaction object (smart pointer) is provided. This follows immediately from the point where all previous blocks are also in the checked state, which is the first state in which all relevant prevouts must exist.³² Previous outputs are first queried from the store and assigned to the transaction object.³³ **A missing previous output is not cause for transaction invalidation, though it terminates this phase and the transaction is either rejected by the pool or remains in the checked state.** The result is returned (pool) or back-propagated (block accept) by result code (e.g. error::success, error:: missing_previous_output, or first input failure code).

For independent transaction receipt (transaction pool), context is obtained from the transaction network protocol, as derived from the confirmed chain at the point of arrival (flags, ++height and median time past). For transactions obtained from block or compact block download, context is obtained from the respective network protocol, as derived from the candidate header chain, and passed through block validation.

Contextual consensus checks (pool and block):

- **Transaction empty** (flags [not checked by block])
- **Overspent (prevouts** [not checked by block])

Contextual DoS checks (pool only, block has PoW):

- **Block weight limit (prevouts, flags, [redundant block check, arbitrary here])**
- **Block signature operations limit (prevouts, flags [redundant block check])**
- **Missing previous outputs (prevouts** [prevout could appear, cannot become connected until they do])
- **Confirmed double spend (prevouts** [not relevant to transactions, verified on block confirm])

Relative checks (pool for DoS – but can allow in time/height window, consensus for block → non-relative context):

- **Transaction not final** (height, MTP [could become valid])³⁴
- **Immature coinbase spend (prevouts, height [could become valid])**
- **Sequence locked (prevouts, flags, height, MTP [could become valid])**

Finality, immaturity, and sequence are recoverable in another context of the same flags, without reconnection of outputs. We could optimize later context revalidation by setting these failure codes only if connect succeeds.

Following successful completion transaction connect is synchronously invoked.

³² With some presumably very expensive indexation, this could be determined even in the case of a preceding block check gap. Given that this could only accelerate bootstrap above the last milestone, this is not envisioned.

³³ This query may be performed concurrently (or batched as optimal).

³⁴ This doesn't need prevouts, but performed here because tx is deserialized and because it precludes the need to cache the value in a table for roll-up to block validation.

These checks are observed within the chain:: transaction.accept(context) method. This phase is bypassed for full/compact transactions under milestone.

Transaction Connect

Transaction **connect** is invoked directly from transaction accept, which passes pool/compact/full status and context.

Any (full block) transaction that was preconnected is tested against its prevout script to verify the preconnection assumption (postconnection). If the test succeeds for any input it is not posted.

All inputs of non-coinbase transactions³⁵ that are not postconnected are posted to the database threadpool for concurrent connect validation.³⁶ By capturing a common shared pointer in each input handler, the completion handler can be signaled by pointer destruct. The completion handler for a transaction completes transaction connect, which returns to block accept or the transaction pool caller. The script evaluation result code is back-propagated by result code.

Upon completion of all script evaluations, a “connected” state is written to the transaction validated table and “transaction” is signaled.

These checks are observed within the chain:: transaction.connect(context) method. This phase is bypassed for full/compact transactions under milestone, with no signal or write of a state change.

³⁵ There is no previous output for a coinbase transaction and the input script is never executed.

³⁶ For block-associated transactions, there is no reason to delay this for the transaction accept checks.

Blockchain Chasers

Block download is always parallel, not a distinct mode of operation for bootstrap. Downloads are initiated on the candidate chain when the top block time is close to current time.³⁷

The node manages efficient request and receipt of blocks across channels. It greedily manages peer performance by maintaining a measure of standard deviation in peer throughput, dropping uncompetitive peers that fall behind the configured deviation threshold.³⁸ To open any one channel it uses batch connection. Peers compete to be the first connected. Other attempts are dropped after one completes the handshake. It attempts to fill block population gaps in relation to the candidate header chain by prioritizing earlier blocks, while keeping all channels fully utilized.

Five blockchain chasers operate on independent strands of the database threadpool. These synchronize **header organization, download, validation, confirmation, and graph** operations to drive blockchain and template construction. This allows the minimal level of order to be imposed upon the related processes. These rationalize processes implemented in earlier v4 work.

Chasers are notified by **header, strong-block, block-check, block-accept, and transaction**. The notification is reposted the respective chaser strand, so a simple function call on a shared object, with candidate height (or transaction FK), is sufficient.

Header Chaser

The header chaser is notified by **header** message. Header population is non-competitive, allowing multiple and slower peers an opportunity to provide the strongest chain. The chaser obtains invokes header accept/connect. It maintains a pruned memory tree of accepted headers. These are retained as weak branches relative to the (archived) candidate chain.

When it discovers a stronger branch it notifies the other chasers of the branch point via the **strong block** message, and reorganizes the candidate chain. The branch point is a branch relative to the candidate chain, and may be at, above, or below the fork point (any point above the top checkpoint). By comparison, the fork point is the candidate chain relative to the confirmed chain.

DoS protection of the memory tree is based on valid PoW connected headers, though the chaser accepts any connected header above the top checkpoint (as no reorganization below this point is possible). Guarding against DoS beyond this requires the hard forking decision to insert checkpoints into config, but is easy.

Block Chaser

The block chaser is notified by the **strong block** message. Upon notification, if the top candidate header time is close to the current clock time (the message implies that the chain is both new and stronger than the confirmed chain), the block chaser scans for all unchecked blocks in the candidate chain above the branch point and posts them to the parallel block requester. This time interval is a configuration parameter. This is a DoS measure (and bootstrap optimization) to guard against prematurely downloading a weak chain and then having to reorganize it.

³⁷ This time interval is a configuration parameter.

³⁸ This also helps load balance the network by unloading slower peers and loading faster ones.

Each gapped block hash/height is collected in order. Download is managed by the network as in the previous implementation. If the branch point is above the fork point this must not preempt requesting the intervening downloads.

A reorganization prunes the existing download queue down to the new fork point, as all block above it are no longer required. Given that reorgs are presumed to be small, and that the blocks/txs are still presumed valid under candidate PoW, it is more efficient (and a lot simpler) to just blindly check all incoming blocks regardless of pruning. As the validation chaser follows the candidate chain's index, no conflict arises.

Validation Chaser

The validation chaser is notified of **block check**, and moves up sequentially until finding a candidate block that is not checked.³⁹ For each checked block it posts a block accept validation job to the database threadpool, with no completion handler.

When the chaser encounters a connected block it instead moves immediately to the next block. Upon encountering the top of the candidate chain, it stops and awaits the next block checked notification. The chaser is idle until the candidate chain passes the top milestone (or checkpoint if out of order). Milestone blocks and their transactions are not marked as valid. If an invalid block is found it pops and invalidates all blocks in the candidate chain and signals work-above termination (not required).

The fork point is initialized at startup by a linear scan of the candidate and confirmed height indexes for the last common block by FK comparison. The chaser is then initialized by a linear scan of the candidate height index for the last populated block above the fork point (or to the last checkpoint/milestone if higher). Population is determined if the txs association table has an entry (for the block FK). These are very fast searches (seconds).⁴⁰

Confirmation Chaser

The confirmation chaser is notified of **block connect**, and moves forward sequentially until finding a candidate block that is not connected.

Spentness must be determined sequentially by confirmed height, so done here. We don't do double spend checks until reorganization time, and at that point one block at a time, so all preceding blocks are confirmed. This eliminates the need for a candidate spend table (as in the previous design). If a connected candidate block is found here to double spend, it can be marked as invalid (or left in its state), along with all candidates above it.

If the validated portion of the candidate chain is not yet stronger than the confirmed chain, the chaser keeps moving until it is, there is nothing to do. When the chaser encounters a valid block it skips double spend checks (the block was previously confirmed and unconfirmed and is now being reconfirmed).

³⁹ All objects are pass check validation before archival. However with compact blocks the header and transactions are archived independently. The "block" is archived in the txs associations index. The header, transactions, and block each have static checks, so block checks are performed as a prerequisite of the association. The association existence is the indication of successful block check.

⁴⁰ At restart the chaser will not move until block checked is signaled, so it should be signaled once upon start.

Upon encountering the top of the candidate chain, the chaser stops and awaits the next block connected notification.

Otherwise it:

- Stacks the reorganized branch NKs in memory for notification, and a set of FKs for restoration.
- Unconfirms and pops the height index for all confirmed blocks/txs above the fork point in order.

Reorganizing the chain, one block at a time in order, it: ⁴¹

- Checks BIP30 (< BIP34 < BIP90) duplicate coinbase.
- Checks that all prevouts are not confirmed spent at or below the fork point. ⁴²
- Checks that all prevouts are confirmed if below the fork point (reusing prevout FK). ⁴³
- Pops a cache entry from the restoration stack.
- Marks the candidate block as valid (not double-spending), which is terminal.
- Confirms and pushes the candidate block (not terminal).
- Confirms non-coinbase transactions (not terminal).
- Advances the fork point indicator to the new (top) confirmed block.

If a double-spending block is found it pops and invalidates all blocks in the candidate chain, signals the other chasers, and restores the previously confirmed chain from the restoration stack. This includes pushing the restored blocks to both the candidate and confirmed height indexes, and resetting the chasers.

Restoration is straightforward given that the previous chain objects remain in the checked and valid states. Only block/txs confirmation state changes are required. If the node is halted during the process it resets to a checkpoint and continues normally. Shutdown is suspended until work completes, and the node can simply hold this thread until the end of a reorg and then honor the stop signal.

The only excess cost is in validating a stronger (by PoW) branch, above a buried (i.e. not top block) point of invalidity. The DoS protection of PoW is more than enough given the very low cost of restoration. Server queries would reflect the interim height drop, but can be suspended during a reorg if desired, but prob not worth it.

The chaser is initialized at startup to the last confirmed block above the fork point, piggy-backing on the validation chaser startup scan. Confirmation is determined by a true value in the block confirmed table, by block FK. At restart the chaser will not move until block connected is signaled, so it should be signaled once upon start.

bIP30 Duplicate Coinbase

A coinbase transaction is the only type that can be included in more than one block, as a consequence of not having any prevouts. Under the presumption of SHA256 hash non-collision, [BIP30](#) is applicable only

⁴¹ Spends are implicitly marked with height the by confirmation of their spending transaction (navigable).

⁴² These checks and transaction marking can be parallelized.

⁴³ If the prevout is in the same block, it is both valid and not double spent, as its transaction has been validated and the block has been statically checked for internal double spends.

to coinbase transactions. A duplicate tx with inputs is precluded as a double-spender. There is one pair of historically-duplicated coinbase transactions (91842 and 91880, both under checkpoint). BIP30 excludes all others, unless the preceding duplicate(s) are confirmed spent. This can be easily verified at confirmation.

A duplicated coinbase is a single record in the store. Validation states are contextual, so this does not produce any conflict. At confirmation time the output is checked for its current confirmation state and confirmed-spentness. However reorganization presents a problem. Given that confirmation state is not contextual, disorganization of the coinbase reverts its confirmation state. If there is a prior confirmation of the containing block, the confirmation state on the coinbase transaction will have been reverted. This will break subsequent BIP30 validation checks (and result in an inconsistent server query state on the individual coinbase). [BIP34](#) activation preempts this check, as a duplicate coinbase cannot exist given that height is embedded. Subsequent freezing of BIP34 by [BIP90](#) preempts the need to verify either.

So confirmation state is not associated with coinbase transactions. As transactions are archived with a coinbase flag, any coinbase transaction's confirmation state can easily be chased to the containing block's confirmation state. And given checkpoints and subsequent forks, it's extremely unlikely this code will ever execute.

Preconnection

Preconnection contends with the validation chaser. The block check phase may complete either after preconnect execution (synchronously) or before it may complete (asynchronously). Synchronous completion blocks the downloader, creating gaps that block the validator. Asynchronous completion creates a validation race for the preconfirmed transactions, resulting in unnecessary duplicate validation in the case where preconnection succeeds.

When there is a branch of candidate headers that is stronger than the confirmed chain, blocks are downloaded concurrently. The block network request is preempted by a compact blocks message, which issues a transactions request for missing transactions. The candidate chain manager subscribes to header and compact blocks messages and maintains an in-memory (pruned) tree of weak header branches. It organizes them into the (persisted) candidate chain as a stronger branch is identified.

Typically a single header enters the tree and is immediately reorganized into the candidate chain by archiving the header, incrementing the candidate height, and initiating a transaction subset download (or block). Whether based on a header->block or compact->transaction workflow, the outcome is the same. The validation chaser waits on a "block checked" notification, which indicates that a block is populated. If that block is the gap that the chaser was waiting on, the chaser invokes block accept/connect and proceeds to the next block in height. When the confirmation chaser received the "block accepted" notification, it confirms the block and increments the fork point (to the top in the single block scenario). At this point the confirmed and candidate chains are the same, the fork point is at the top, and both chasers are waiting at the top.

The objective of preconnection is to ensure that the CPU is not starved for work during I/O intensive blockchain bootstrap, when full blocks are arriving in parallel. While the sockets and disk are working overtime, the CPU may be sitting largely idle as the validation chaser waits on a gap to be filled. This was observed very clearly in early V4 work. The entire chain can be downloaded and stored as the validator sits on a single missing block, idling the CPU. This either overflows the in-memory transaction download

cache (held for the validator) or causes excessive transaction deserialization for the purpose of validation (depending on transaction cache configuration).

Preconnection can validate from 100% to around 20% in 2021 of transactions in a block. This decline progresses along with segwit adoption, as witness transactions cannot be preconnected (due to the requirement that the output amount is provided to script validation). Yet the period between top consensus checkpoint (block 295,000) and today contains a very significant number of non-segwit standard transactions. Preconnection allows these to be validated in full, during parallel block download, above gaps in the candidate chain, as no prevouts are required. This is later confirmed by the validation chaser, which compares the actual prevouts to those presumed by simply comparing the prevouts to those assumed.

This allows the CPU intensive script validation (and all other aspects) of the accept/connect phase to become unblocked by block download order. In fact, under the presumption of candidate chain validity (based on PoW) it does not matter at all when these checks are performed, as long as the CPU remains fully loaded. Though ideally to ensure this we want to complete ordered validations as a higher priority than unordered preconnections. This reduces the occurrence of revalidation for preconnectable transactions that fail to preconnect (though this should be very small given standardness). Otherwise, as long as the CPU stays full loaded, confirmation will complete as soon as it would if no gaps ever stalled the validator. It just changes varies the rate of confirmation at different parts of the chain, without varying total confirmation time.

Downloads are processed on the network thread pool, and stored on the database threadpool, both configured with normal thread priority. The validator posts jobs to a thread pool reserved for validation, configured with high priority. This configuration assumes that validating the entire chain requires more time than downloading and storing it. While this may not always be true, it presently seems to be the case by a wide margin. If this is not true, then thread pool configurations can of course be changed. Therefore the validator hogs CPU resources from the network/store. This slows downloading, which can have the adverse effect of creating gaps for the validator. However, the gap stalls the validator, freeing the CPU for the network/store. With prioritization on gap-filling, this is an optimal balance.

Validation Work Buffer Overflow

There is a material hazard to the validator posting too much work to the opaque ASIO work queue. This can be resolved by the validator and checker managing a shared work queue. All components that post jobs to a queue must have a way to manage the “size” of the queue. The estimate of work in transaction validation is based on its sigops count, which is required for validation (with the formula adjusted by segwit). This is already calculated during transaction accept, as required for subsequent block validation/inclusion. This calculation precedes the expensive script execution.

Therefore the size of the work queue can be closely estimated by incrementally incrementing and decrementing two atomic work accumulators owned by the validator and passed into each validation job (along with context), one for validator work and the other for preconnect. Preconnect jobs can be bypassed when the sum of the work queues reaches the configured limit. Isolating both values is trivial and allows for more advanced balancing as the need arises. The validator can also use this to throttle its progression.

Preconnect/Postconnect Synchronization

This does not resolve the race between asynchronous validations to do the same work. We don't ever want the same script executed twice. Given the goal of not starving the CPU while not gapping the validator, the solution is to only post preconnection jobs when the validator is starved. While it may seem straightforward to determine if this is the case, the validator posts jobs across blocks asynchronously. It will almost always be waiting on a gap as this work proceeds. The preconnector continues after block connect, with jobs all posted and executing asynchronously. So the validator can post jobs that end up doing the same work when the gap is filled, especially for the previously-gapping block.

The downloader might obtain all blocks in order. The validator would then sweep them once connected, posting validation jobs for all transactions, as they are all ungapped. By at this point preconnect has already been invoked on all of them (as check completion invokes preconnect and then signals the validator). Or there may be a gap. This is where preconnect is useful. It preconnects where the validator cannot connect. So the preconnector always wins the race, to start jobs (completion being the race, given synchronization on the validated table).

It's worth considering both the overhead of posting duplicate contending jobs, even if they complete in the desired order. With two jobs posted per transaction, overhead is doubled. Without preconnect, the block check operation completes synchronously. This is not a consideration for compact block connect, as all transactions are either in the connected state (via pool entry) or will be validated immediately following connect, as compact blocks are not utilized for bootstrapping (see "close to the tip of the best chain of the receiver" in [BIP152](#)). Block check merely associates them and executes block accept using their cached metadata. But this remains a consideration for full block download.

Synchronization is implemented through the transaction validated table. The connector checks for a preconnect entry of the given context. If the entry is found it then performs the postconnect operation, and otherwise accept/connect. The presumption is that if preconnect is possible the entry is likely there. It could test the transaction for preconnectability before querying for the table entry, but presumably the query is faster – and it's definitive. In order to limit overlapping connect/preconnect, the preconnector could also query the table, for the connected entry. Preconnector query for the transaction is necessary, as it may have been previously archived. This provides the FK for faster lookup in the validated table. In any case, this is a tradeoff against marginal costs.

Even if both actors query the table, a race exists. To mitigate this race requires creating some distance between work being posted by the validator by the preconnector, so that preconnector work completes before validator work for the same transaction is posted. There are several ways to achieve this. However, in order to resolve it entirely, the shared validator object could expose a synchronization object. This would be much cheaper to read than store table and provide a perfect guarantee (nevertheless, only the store should be relied upon for state changes).

Given that preconnect precedes postconnect, if the preconnector starts a job for a given transaction, it can block the validator from executing the posted connect job for that transaction until the preconnect job completes. Note that block check, accept and connect are presumably all implemented in the same class and it is from this code that all jobs are posted, so this very manageable from a code organization standpoint. Given that both actors are working on the same block, it would not significantly delay the block validation, and given that block validation spans ungapped blocks, there is just the overhead of the

unlikely failed preconnection and of posting two jobs for one operation, when the preconnection isn't really beneficial.

The preconnector “locks” the tx (it always wins the start race), and the postconnector provides its completion callback for deferred execution once the lock is released. The synchronization object ensures subscription atomicity using an internal mutex. Identity is established by the 32bit byte transaction FK. This creates a visible/manageable synchronization buffer, and allows execution to continue unblocked. By observing the buffer utilization (which should be low) a determination can be made if spacing by tx distance (height) would be beneficial. This is then just down to tuning for bootstrap.

But given overhead, and the fact that the validator will more quickly validate the entire ungapped section of blocks, it makes sense to never enable preconnection except above the first gap. The first gap is quickly discovered by the validator, as it waits there (it is also known to the download queue, managed by the network). The synchronization object can provide the current gap height, updated by the validator each time it stops. This height can be checked by the preconnector before posting work for its new block, avoiding posting any jobs for the currently-gapped block. This creates distance of one block, which will be closed more rapidly by connect than preconnect/postconnect. It also ensures all preconnection is at least started in the right place, where the subscription queue can then provide the final guarantee.

Non-blocking thread synchronization always comes down to message-passing with buffering, a lesson from the late Pieter Hintjens.

Transaction Chaser

The transaction chaser manages the memory-resident, metadata graph of transactions that are **independently spendable** at the top confirmed block.

There are three competing consumers of this critical resource.

- Transaction download
- Block confirmation
- Block template creation

Transaction download populates the graph between organizations, and uses the graph to correlate compact block short transaction identifiers. **Block confirmation** uses the graph to avoid hitting the store when evaluating previous output confirmation height and spentness, and reorganizes the graph.

Template creation uses the graph to generate preliminary and validated block mining templates.

Transaction Graph

Graph Structure

The graph is a DAG of transactions (nodes), containing a list of input objects with prevout references, and retains all necessary information to determine the validity of a block populated with its transactions. Cycles are inherently precluded under the presumption of SHA256 hash non-collision. Navigation is from transaction input to input's previous output transaction.

- **Transaction hash** [32]
- **Transaction FK** [4]
- **Compact ID** [6]
- **Markers** [3] (presumption of 8 bit bool)
- **Weight** [4] (as defined by context, block inclusion limit)
- **Sigops** [4] (full count, includes prevout as defined by context, block inclusion limit)
- **Fee** [8] (determines maximum coinbase value, used to maximize template value)
- **Reference** (this can be unioned into 18 bytes per reference)
 - **Input->Prevout-Transaction pointer** [8] (i.e. unconfirmed) navigation (FK cached)
 - **Input->Prevout-Transaction FK** [4] (i.e. confirmed) navigation (null pointer)
 - **Input->Prevout-Transaction height** [4] cached for confirmed (FK) references
 - **Output index** [2]

The graph holds double spends to the extent that they satisfy retention policy. In other words, a graphed transaction can be confirmed in the next block as long as it will “fit” and isn't joined with another that spends the same prevout.

Apart from flags, context is a minimum height/MTP constraint on block inclusion. Transactions are includable if they are “connected” for the validation context, and an including block is valid if they don't violate its sigop or size constraints, and there are no block-internal double spends.

Graph node references are retained in a bimap, indexed on transaction FK and compact ID. This provides navigation entry via transaction FK and in-memory association of an incoming compact block message with any graphed transaction.

Graph Construction

The chaser maintains three strands for synchronizing the three competing consumers. Insertion is triggered by both download and block deconfirmation. Removal is triggered by download (prioritization of double spends to save space, as configured) and block confirmation. Block deconfirmation shifts previously-confirmed transactions into the graph, converting any FK references to pointers. Block confirmation shifts out all confirmed graph transactions by converting pointers to FK references (the FK is always retained; the pointer just becomes null).

As neither prevout spentness nor prevout confirmation status are an aspect of transaction connectedness, these checks are performed as a prerequisite for graph entry. This is analogous to the confirmation chaser checks. By performing these upon pool entry they are cached in memory, allowing spentness to be assumed during block confirmation of graphed transactions, with the exception of double-spending among the set of confirmed transactions. The graph is always relative to the top block, so this remains true even in the case of reorganization.

As a DoS measure, confirmed spent prevout is criteria for rejection of a transaction by the transaction pool (and therefore the graph). For the same reason, double-spenders of a prevout are pruned when the prevout becomes confirmed spent. Finally, reorganization deconfirms outputs that may be graph-associated. While the spender is in this case retained in the graph, it's reference must be converted from FK to pointer. To avoid costly comparison against all deconfirmed prevout transactions, the height of confirmation is cached for all FK references. This allows reorganization to simply convert all references at the reorganized height.

When all transactions are graphed, there is no need to hit the store during block confirmation, with the exception of a trivial BIP30 check. Similarly, there is no need to hit the disk during graph generation. This work is front-run by graph construction. This is primarily a consequence of transaction broadcast, but also occurs due to compact block requests. These are processed in the same manner by the graph, though compact block transactions (i.e. under PoW) are not subject to policy. If a transaction is not graphed at the time of block confirmation, the confirmation chaser performs the store validation.

The graph is not populated until the confirmation chaser confirms the top candidate block. At this point header requests and therefore block downloads cease and compact blocks communication natural begins. Catch-up, after a restart, network outage, or bad peers automatically reverts to block download. Each peer connection begins with a header negotiation.

Validation Context Considerations

The graph can only contain transactions that are in the connected state for the top (confirmed) block. This state can change, but only as the result of confirmation. **Validation context is strictly a consequence of the set of preceding blocks.** There are three scenarios pertaining to context that arise from reorganization:

- Height and MTP increase
- Height and MTP decrease
- Change to flags

MTP is monotonically-increasing by height, as is the height integral value. **Finality, Maturity and Sequence** are the three transaction accept consensus checks affected by changes to MTP/height. Yet these are all minimum values. In other words, an increase to either value cannot invalidate a transaction. If this were not the case, a new block could invalidate its ancestors. Yet upon any deconfirmation, all transactions in the graph must be revalidated by these three checks in the accept phase. Importantly, **scripts are not affected**. This is then written with result code and satisfied (i.e. pool) context to the "transaction validated" table. The transaction is removed from the graph if it does not pass.

A simple approach to flags is to simply dump the entire transaction graph when there is any change to flags, either by confirmation or deconfirmation. However, depending upon the flag, transaction validation may not be affected. A context comparison function can be provided by the chain state class that provides the universal determination of context sufficiency. This can be refined by retaining transactions with another validation table entry that satisfies the context. Finally, a determination of whether script validation is affected may also be provided by the chain state class.

So there is no action for a confirmed block, a relatively simple/fast action for a deconfirmed block, and a potentially costly action in the event of a fork – though this can be minimized to the absolute minimum revalidation requirement (and it's exceedingly rare). Note that deconfirmation is always immediately preceded by confirmation, so this may not be the time to revalidate the scripts of the entire pool. But height/MTP checks are fast enough to warrant retention of the pool, if only to support the incoming block.

Along with the spender checks previously described, these are sufficient to ensure the independent confirmability of all transactions in the graph.

Synchronization

Given that independent threads contend to update the graph, these must be synchronized. It is an objective to avoid locking and even the need for atomic property guards. For this reason each consumer operates an independent strand, which provides order to each of them (but not across them). The pool strand precludes concurrent transaction pool updates. The transaction chaser maintains a strand for ordering template creation operations. And the confirmation chaser operates on its own strand, as its operations are strictly ordered.

Both pool and confirmation strands must perform insertion. The pool is stranded against its own inserts, and the deconfirmation workflow suspends and blocks all others. This block does not unreasonably affect the template workflow, as the underlying reorganization can only be triggered by preceding validation of all blocks in the new strong candidate chain, and template workflow is already working above the top candidate block as soon as that happens, and requires the graph insertions to formulate a populated template.

Object deletion would also require locks, but a garbage collection strategy is used instead. The pool workflow marks for removal (of double-spenders deprioritized by policy), as does confirmation (to accelerate its own operations by deferring the work). As we will see, template creation must perform logical removal, which it also performs by marking. By giving each stranded operation an independent flag on the transaction object, each may not only observe independent subgraphs, but may modify membership of that subset without the need for atomic locking on its flag. The garbage collector can be scheduled at any reasonable time over which it can lock the graph. This would generally follow immediately after the confirmation chaser confirms the top candidate block.

Optimization strategy allows four workflows to preempt each other in order of priority:

1. Block confirmation
2. Block template creation
3. Garbage collection
4. Transaction download

The most common new block scenario is concurrent read locks by the confirmation and template workflows, and these preempt the others. These critical workflows are unblocked unless there is deconfirmation.

Workflows

garbage Collection Workflow

Garbage collection is triggered as necessary after the top block is confirmed, and takes a **write lock**. It removes objects marked by the transaction (**double spent**) and confirmation (**confirmed**) workflows. This is not stranded as it is a singular operation.

Transaction Workflow

Connected transaction are posted to the transaction strand and are inserted as appropriate. As insertion necessarily modifies the graph, this takes a **write lock**. It also marks **double spent** for removal in accordance with configured policy.

Transactions accepted into the graph are cached with compact ID values and the values are written to the compact ID index. The cache provides for rapid creation of a compact block message for broadcast, once a valid block is mined. The index allows the compact blocks protocol to recover previously accepted transactions by compact ID.

Confirmation Workflow

This workflow has already been covered in parts. The confirmation workflow:

1. Reorganizes deconfirmed blocks into the graph (inserts transactions) in order.
2. Uses the graph as a validation query cache for the next block.
3. Signals **block confirmed**.
4. Organizes the next confirmed block into the graph (removes transactions), GOTO 2.

For the last block (usually the only block), step 4 does not need to block subsequent confirmation, up to the point where the graph is required as cache. However there are really no other steps performed by the confirmation chaser prior to graph navigation. Therefore this workflow is sequential in the validation process. **Step 1 takes a write lock** and **step 3 takes a read lock**, as it is only marking transactions (**confirmed**). These are independent atomic operations. Once deconfirmed transaction insertion is complete, the template and confirmation workflows may proceed concurrently (independently marking the entire candidate chain, vs. one block at a time respectively).

Template Workflow

The template workflow does not modify the graph, and as such takes a **read lock**. The objective is to maximize miner reward opportunity. This assumes a strategy of always following the candidate chain, based on the DoS protection of PoW. The template is built from the transaction graph alone, but always for the strongest visible block (**strong block**). The candidate chain may of course be invalid, but upon that determination the new strong block is again the top candidate.

The template workflow therefore transitions across three states in order:

1. **Empty** block template
2. **Preliminary** block template
3. **Valid** block template

These are based on the current confirmation status of the strong block. When the reorganization is initiated, an **empty block** is immediately issued. This limits time wasted mining a weak branch, while a preliminary template is constructed. Construction uses only coinbase metadata from configuration and

the strong block hash and height (height required in the coinbase by BIP34). Issuance is via the server mining endpoint.

Immediately the workflow proceeds to **preliminary** template generation. A preliminary is a valid template based on the presumption of strong block validity. The validity of any preliminary template is predicated on confirmation of the strong block. Once that is assured, all preceding preliminary templates are **valid** templates.

In the event that the candidate branch is found invalid, the candidate reorganization may be to a previously-mined block. In this case the workflow initially issues a template for the strong block from its **work heap**, or an empty one. The heap also provides for correlation of mined blocks, and retains the ordered set of short hashes and precomputed coinbase (with reward amount, height, script(s)) necessary to produce the compact block message upon successful discovery.

The transition from the preliminary to valid state is indicated by the **block confirm** notification for the parent of the preliminary block. At this point the workflow is optimizing by attempting to obtain a more valuable valid template until a reprioritization is indicated by the **strong block** notification. It optimizes on preliminary and valid templates by responding to **transaction** notifications, requering the graph for the best template as a consequence of successful transaction insertion.

The computer science magic behind this graph-navigation query is the **solver**. **Given that all transactions are confirmable, the sole responsibility of the solver is to find a connected and non-double-spending subset that maximizes fee and satisfies weight and sigop constraints.** Ultimately the entire objective of a node, for mining purposes, is to produce this graph as rapidly as possible so that the solver can work on this well-defined problem. Various solver strategies are possible, but the solution to a typical graph cannot actually be found in anything close to reasonable time, as this is an [NP-hard problem](#).

For example, one or more threads can be spun out to work on this problem between transaction notifications, and even ignoring them, with some jobs running longer than others. The initial approach is a simple knapsack solver, which appears to be the strategy of choice for Bitcoin mining. This well-defined and specialized problem is also ripe for hardware optimization for high end mining. The solver also provides a connected subset of the graph to satisfy the P2P “mempool” message.

Graph Pruning

The template workflow may require a template for the strong block when it is not the top block. This is the case in the simplest one block scenario. After issuing an empty template it wants a preliminary one, without waiting on strong block confirmation. However the graph is populated for the top block. It can either wait until confirmation, relying on the empty template, or navigate to obtain a preliminary template (i.e. one that is valid if all intervening blocks confirm, under the PoW presumption).

To do this requires ensuring that all presumed-confirmed transactions are logically removed from the graph. Knowledge of the set of presumed-confirmed transactions requires that the block chaser be positioned at the strong block (top of the candidate chain). So upon obtaining the **strong block** notification, the graph pruner navigates all candidate blocks from the fork point up to the block chaser, chasing it to the strong block. It enumerates all transaction FKs (one query per block), marking all that exist in the graph as **pruned**.

Pruning is the same operation that confirmation will perform on each block as they are confirmed, using its **confirmed** for removal marker, though this spans all blocks in the candidate branch and does not block on validation or confirmation. The pruned marker is ignored by all other workflows. Note that pruning and confirmation are both read locks, so the operations can execute concurrently. In other words, confirmation and template workflows are not in contention for the graph apart from deconfirmation.

Confirmation proceeds concurrently with workflow, though one block at a time, up to the highest block validated. The strong block may be more than one block stronger than the confirmed chain, though reorganization into the confirmed chain happens immediately when there is one stronger block. Other candidates may still be checked headers, checked blocks, or partially-connected at that point.

The template workflow does not want to wait on those processes. It wants the graph to be marked for the strong block, regardless of validation or confirmation of intervening transactions and blocks. So the template workflow marks (**pruned**) all transactions presumed-confirmed by all intervening blocks from the fork point (this will overlap with **confirmed** marking, which is expected and inconsequential – we don't want to couple them) through the strong block, regardless of the state of those blocks, chasing the download chaser as it fills gaps. Pruned state need be purged only following a **regression of the branch point**, as this implies a **reorganization of the candidate chain**. Otherwise pruned state remains valid within the graph. It is naturally purged by confirmations as candidate blocks are organized and their transactions removed from the graph by GC.

When contention does occasionally arise, only due to a **regression of the fork point**, the template workflow is preempted. This is a **reorganization of the confirmed chain**. However this does not materially delay the template workflow. In this case the graph should be moved to the new branch due to block deconfirmations anyway (for template optimization) – and confirmation is priority. All deconfirmed blocks are dumped from the graph by the preempting confirmation workflow, freeing the validation workflow to proceed against all blocks in the (transitioning) candidate chain.

Pruning requires one query on block FK per pruned block, in order to obtain the set of contained transaction FKs (stored as an array). The graph is indexed in a memory hashmap by transaction FK, making this process trivial (as with block confirmation marking and transaction graph double-spend marking). Pruning simply queries and marks. Finally, validation context must be verified in the case of a reorganization. It is worth considering that the pruned graph is sparse, as not all candidate transactions may be present. This reduces opportunity but not integrity.

Interface

The mining interface is a ZeroMQ endpoint, with an administrative endpoint for share accounting. Templates are issued and shares returned over this interface. The interface may be wrapped with another protocol for compatibility purposes. The server interacts with the template workflow by subscribing to templates and relaying to miners, performing the necessary accounting.

A mined block header is submitted by the server to the template workflow, which submits to the header chaser for tree integration. With a successful response the template workflow repackages the header as a compact block (using its work heap), posting it to the P2P network on all channels (via the “compact block out” protocol). If the block header is invalid or disconnected, the error is propagated from the header chaser back to the server call. If the block is invalid, the block will be reorganized out of the

strong chain. If it is weak it may never organize into the candidate chain, and the node is already working on the strong branch. If the template workflow cannot correlate it to the heap, the necessary information can be queried from the store to generate the compact block message.

The server may also relay through any other protocol (e.g. private relay network), either delaying until internal confirmation of the block (subscription) or posting on acceptance from the header chaser via template workflow.

Transaction Cache

It is important to avoid deserializing previously downloaded transactions for validation. This is unavoidable upon process restart in the case where blocks are already downloaded and are awaiting validation of certain transactions. Otherwise, transactions from blocks that are downloaded above a milestone are cached in memory. There is no caching below a milestone, as the block is already valid and spends are already written.

There is no need to cache the block itself. When block accept/connect encounters an unconnected transaction, it first attempts to load it from the download cache. When a transaction has been preconnected there is no need of the transaction in the accept/connect phase unless postconnection fails. So we can avoid caching all preconnected (and previously connected) transactions. This presently varies from 100% to 20% of all transactions based on segwit adoption. In the few cases where postconnection fails, it will cause in a deserialization from the store, but probably worth the savings on the running cache size.

The cache is implemented as an in-memory hash table of transactions, by transaction archive foreign key. If not located in the cache, the transaction is deserialized. Reading the transaction from the cache removes it, as it is only useful once. Preconnect has the original block object, so only accept/connect requires it from the cache.

The primary consideration with the cache is RAM consumption. The cache is therefore configurable in byte size. Transaction size is computed and retained when the transaction is initially archived (and cached), so this derivation is not an extra cost.

When the cache becomes full it rejects new submissions. Given parallel download this can be inefficient in that blocks are not ordered. There is an attempt to obtain older blocks before newer blocks, as this also reduces blocking the block chaser. With an unlimited cache configuration, there is never a transaction deserialization from the store. With a milestone at the top block, the cache would always be empty.

However, the cache hash table can easily be ordered by 32 bit block height, allowing it to operate efficiently as a circular buffer, discarding at the top, with search by height and transaction FK (essentially a sparse chain of indexed sparse blocks).

Given that compact blocks are expected, the cache should be empty after bootstrap. However if a full block is received, operation is the same as parallel block download (as there is no distinction, parallel block download is the normal behavior whenever there is a candidate header chain with missing blocks).

Slab Hashmap

There is plenty of opportunity for ongoing optimization and tuning. For example, by retaining the message buffer (read from the socket and used for initial hash generation) an overflowed object can be streamed directly to a slab hashmap by transaction FK, and deserialized upon a cache miss. The full cache can even be flushed at stop and used for transaction object repopulation to the memory cache upon restart.

Or just use the slab hashmap as the only object cache. This is probably where we should start, since it's easy and unbounded, and can be further optimized into an in-memory read-ahead buffer (populating while the CPU is validator bound). This would depopulate to empty once all transactions are connected. It could even be checkpointed along with the other table headers, allowing for process termination recovery. Though as space is never recovered, the file may contain a significant fraction of the entire chain (less headers). It could be closed and deleted once the connectable transaction count for a current chain falls below the transaction memory cache limit.

Bootstrap Cache

There is no reason to download the first 295,000 blocks from the peer-to-peer network, as these fall under consensus checkpoints (a sequence of soft forks, that to revert would constitute a hard fork). As with the genesis block, these are immutable history. However this is a non-trivial amount of data to store along with the genesis block within the implementation. Even obtaining these in an independent file is a costly download. There is no consensus risk to externally obtaining an invalid list of hashes, as the configured consensus checkpoints preclude validation of any non-conforming block. However the node will also generate the list itself.

The hashes of each of these blocks are an important denial of service protection during bootstrap. The checkpoints themselves do not fully serve this purpose, as any number of low work branches of any length may be created below or between them. Above the top checkpoint (height 295,000) there is no guard apart from work required. But with the ordered set of block hashes for all checkpointed blocks, all invalid blocks below the top checkpoint can be rejected.

There is also a speed advantage to having this list of block hashes. Header requests for this set of blocks can be bypassed and the list can be used to bootstrap the network immediately into parallel block download requests. If this file is present and conforms to the set of configured checkpoints, network header requests begin at height 295,001, following the successful confirmation of the first 295,000 blocks.

Storing all preaccepted hashes requires 9,440,000 bytes ($295,000 \times 32$), or ~9MB of data.⁴⁴ Preaccepted headers requires 23,600,000 bytes ($295,000 \times 80$), or ~22.5Mb of data. However given that blocks will be downloaded, which include their headers, this would be a double deoptimization (file size and network traffic). Of course blocks could also be stored, but it should be faster to obtain them from the p2p network in parallel then to download them from one site as a file, so this is also a deoptimization.

The set of preaccepted block hashes is serialized to the bootstrap array table in order when the top checkpointed header is reached. The file is then closed and never again read unless the node is started

⁴⁴ Hashes are not very compressible, given evenness of bit distribution. The genesis block is included to so that the intended coin can be easily identified.

up below the top configured checkpoint. The file is not store checkpointed and is deleted upon recovery from a checkpoint. It can be used to bootstrap another node instance by copying it to the store directory prior to startup. This simple feature accelerates node bootstrap and lightens peer loading by eliminating 295,000 header requests, eliminates a material DoS vector, and adds no centralization or consensus risk.

Dangling Inputs Under Milestone

Under a milestone, inputs can be written without having the output to link, as the output reference is a natural key. If the input is traversed to the missing output, we know that the output transaction/block has not been downloaded. However we prevent query above the confirmed height, so relational integrity is preserved. This allows gapped full blocks to be written with all spends marked as they arrive under a milestone. This is a space-costly but significant bootstrap optimization.

Miner Indexing

The archives and indexes are optimized for concurrency (bootstrap and catch-up) and append (store reliability). They are therefore fully relational and outputs are indexed on natural keys. This minimizes the number of updates required for validation and for confirmation, however the extra navigation adds more cost to spend verification during the confirmation process. This can be short-circuited by a pair of indexes, one for each prevout verification step, and has the side benefit of providing otherwise missing output-to-parent transaction navigation – making the graph fully traversable across all confirmed blocks and validated transactions. As these tables are otherwise not required for validation, they can be easily isolated as a mining optimization. This would trade a modest amount of storage for faster confirmation.

These are used to verify that prevouts are confirmed and not confirmed spent. It is important that the indexes are not costly to create and guarantee relational consistency with the primary tables. The tables do not track confirmation state, this is retained only by block and transaction reference – which accelerates confirmation marking. There are far more outputs than blocks and transactions so it is better to accept one FK navigation (cheap) than to mark each spend (not cheap) upon organization.

The “output transaction” table is a simple reverse navigation, and written at the time the transaction is archived. The “input transaction and output” is a table of validated spends, and is written after transaction validation of its inputs. This ensures that the table retains all validated spends (more specifically only validated spends, although that is not required for confirmation integrity). In both cases no additional search is required to obtain the data required to populate the table. Entries are never removed, as archival and validated are terminal states. Only confirmation is non-terminal, and reorganization is rare (and we store very little confirmation state).

These indexes grow along with the chain, adding 24 bytes to each valid spend and 16 bytes to each output (links included). At a maximum of 40 bytes per output this is a very reasonable tradeoff for disk space in the optimized mining scenario. In fact by using FKs this table is smaller per entry than just the natural key indexation cost of an unspent output cache. An unspent output index cannot be indexed by foreign key (because it doesn’t have an input association), making it slower to search due to the cost of hashing and collision comparisons. It requires 32 bytes for the parent transaction hash, 4 bytes for the input index, and 8 bytes for hash table links. And then of course there is the cost of redundantly storing each unspent output script and value.

Hashing Reserialization

When a block header or transaction is read from the network socket it is a byte buffer. This buffer has in the past been first deserialized and discarded. Later, in order to obtain the object hash, the object is reserialized and the buffer is hashed and discarded (the hash is cached in the object).

This has been the case even for objects for which we will ever only need the hash, in which case the waste is not just a serialization, but the deserialization. This is very common in the case of headers, as we build the chain competitively. It is also common in the receipt of transaction broadcasts. Presumably we generally only receive blocks that we have requested.

This is a CPU-wasteful process in time critical operations.

Change message classes to const, with a mutable containment pointer for chain objects, with deferred deserialization. Expose a method to initialize the object from its buffer as necessary following network deserialization, with an option to dispose the buffer. The buffer may be retained for later immediate serialization. If a deferred deserialization fails the protocol can drop the channel at that time (as opposed to immediately upon receipt).

This allows the contained object construction from the socket buffer to be deferred, holding a pointer to the buffer until that point, and generating the hash of the buffer from it directly upon construct of the message object.

Message Class Containment

Change `network::messages` classes to containment (vs. derivation) of `system::chain` objects on a mutable smart pointer. This eliminates the performance and complexity cost of move construction when building blocks from independently obtained transactions.

Chain Class Fields

Change `system::chain` classes to expose const public fields (vs. properties) for native values. Compute header, transaction and block identity hash from stream buffer upon construct, and provide as a construction parameter (already there), and when constructed without, compute the hash upon construct. This will make the hash const safe and unguarded, and improve other property read performance. This will also simplify source and test.

The identity hash is used as a native database key and is therefore now always present upon object deserialization, and it is directly obtainable (and necessary) for incoming wire traffic. It should only be computed from properties within a wallet and during signature hash generation.

Chain Class Containment

Change `system::chain` transaction inputs/outputs and block transactions to containment on a smart pointer vector. This avoids move construction in performance-critical script signature hash evaluation. It also simplifies and improves the performance of fan-out for concurrency by transaction and input/output, including database query.

Chain Class Store Serialization

Remove customized store serializations/deserializations in `system::chain` classes, as given the relational data model they are no longer required. This eliminates some interface complexity (and dead code).

Preconnect Method

Preconnect is a chain::transaction method that determines if the transaction is non-witness and with all standard input scripts. If not it returns false. Otherwise it generates and populates all prevouts and then invokes the check and accept methods. The presence of the populated prevouts may indicate that the object was preconnected.

Preconnect Verify Method

Preconnect verification accepts an ordered list of previous outputs and returns true if these correspond to the previous outputs presumed by the inputs (or those populated on the object if populated). If preconnect was true and this returns true against actual outputs, then the transaction is valid.

Blockchain Pruning

Each **block/header requires 125 bytes**, consisting of 100 for structure, 12 for (empty) transactions association, 8 for height indexation (block flags storage is not required), and 5 for confirmation marking (if confirmed). Each **transaction requires 97 bytes**, 73 for structure, 8 for block association (if associated), and 16 for confirmation (if confirmed). Each **output requires 59 bytes** and each **input requires 62 bytes**, (including transaction associations). This assumes empty inputs and output scripts.

Input and output scripts can be pruned to any “reorg safe” height, dramatically reducing the storage required for a pruned node. This requires a rewrite of the respective input and/or output table, which is very straightforward. The spend table may be pruned along with the sending inputs. Validation tables may be deleted in full at any time, as these are merely a performance-enhancing cache of previous block and contextual transaction validations. Missing validity results in recomputation as necessary. All other tables are optional - for either wallet services or mining performance. Confirmation tables are preserved in pruning.

Pruning requires the disk space necessary to hold the original table as well as the pruned table. Once complete, the tables can be atomically swapped and the original deleted. This can be done as a background process during operation, or as an independent tool similar to table compression. It can also be checkpointed and restarted using existing mechanisms.

Input validation Cache

Hashing the transaction in the unique ways required for signing is an expensive operation. In some cases this can be performed without the previous outputs, allowing it to be performed during parallel download. This is straightforward for standard scripts. However, with preconnection this is unnecessary as the entire transaction is instead validated. We could however cache individual input validation results for standard scripts, in the case where not all of the txs scripts are not standard non-segwit. This could be cached in a table indexed on input foreign point. It may not be worth the cost of maintaining another cache table. An analysis of the prevalence of such transactions should be done before this is attempted.

Gapped Transaction Validation

Any block-gapped transaction that can locate all of its unspent outputs can be validated, as opposed to waiting to become ungapped. We will not attempt to implement this optimization unless we can find an efficient way to signal to a transaction that all of its outputs are populated.

Reliable Shutdown

Problem Statement

The problem of reliable shutdown is significantly simplified by the use of append-only memory-mapped storage. All tables are implemented as memory stacks, logically pushing and popping data. In all but array storage (two identical simple height indexes), data is never actually popped. Instead the records are reverted by pushing.⁴⁵ This means that all changes are appends to the memory map.

Hash table headers are random access, and two array-based tables must be able to pop data. While array-popped records will always be rewritten within the same transactional commit, this implies that the file itself is not append-only. But isolating each hash table header to an independent file, these reduce to the same problem (random access memory maps). The data segment of hash tables becomes strictly appending.

Hash table headers and both height indexes are smallish files. They can be periodically copied in a checkpointing scheme. Data segments can be far too large for this approach. This presents two different but related problems. Following an uncontrolled shutdown it must be possible to definitively determine what data is intact. Based on this all other data is treated as corrupt. This determination must be made for random access files and for append-only files.

Furthermore there is a problem of logical transactionality across files. This problem exists in the case of each hash table (between a header file and its data file), and across tables (where logical relationships are persisted). For these purposes the two indexes can be treated as hash table headers with no data sections. This again presents two independent but related problems.

Checkpoint Creation

There is a dedicated checkpoint directory.

Checkpoint files, apart from the two arrays, do not change in size. The arrays have fairly predictable growth rates, which are very low (4 bytes each, every 10 minutes). The volume can be checked for sufficient checkpoint disk space at startup, and again prior to creation of the checkpoint.

At some point writes to the store are blocked, and all memory maps flushed, so that a consistent copy of each header file (a checkpoint) can be made, as well as a checkpoint sentinel file (with a well-known name) that indicates checkpoint completion by storing the file name prefix for the checkpoint files. These files serve as the restart point (checkpoint) for a subsequent startup, in the case where the active files were not closed.

Later this is done again, creating a second checkpoint, with the checkpoint sentinel name logically orphaning the preceding checkpoint. This advances the recoverable store position.

In order to safely swap a new checkpoint for its preceding checkpoint, the new checkpoint sentinel is created with a temporary name and the two files are atomically swapped by the operating system.

- Linux
<https://man7.org/linux/man-pages/man2/renameat.2.html>

⁴⁵ This is analogous to a GIT reverting commit, in contrast to a force commit to lower point in the history.

- macOS
https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/exchangedata.2.html
- Win32
<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntsetinformationfile>

If there is no store, the store is created. The checkpointing operation is then executed before the store is started, and again at each startup.

An “initial” checkpointing operation is one that detects no existing checkpoint, but will still rename from a temporary file. It is essential that the checkpoint sentinel be created atomically.

Following successful completion of the sentinel file rename, the older sentinel is deleted, followed by deletion of its checkpoint. This completes a checkpoint operation.

Checkpoint Recovery

If the initial checkpoint rename fails, a subsequent startup does not discover the sentinel.

If the sentinel is discovered, the checkpoint file names are deduced.

In either case all other files in the checkpoint directory may be cleared for good hygiene.

The checkpoint files are renamed⁴⁶ to replace the corresponding active header files.

Following successful update of the active header files, the store is recovered and the last checkpoint remains.

A process termination during the update results in a repeat of this operation on a subsequent startup.

Controlled Shutdown

A controlled shutdown completes by clearing the checkpoint directory following store close. This implies that it will be recreated at startup and will recover disk space when the process is not running.

The checkpoint sentinel must be successfully deleted before clearing the rest of the directory.

Triggering

Problem

It may be sufficient to checkpoint at long intervals and upon explicit command. With a reliable process termination (i.e. without the problem of hanging on shutdown) the problem is largely mitigated. For high performance servers an explicit checkpoint may be preferred to slowed operations. For low performance operations more frequent checkpointing may be desirable. However, checkpointing during database operations requires write cessation.

A call to stop database work would cause outstanding strands to complete, blocking on a join of the database threadpool. But this would cause stop failures in database calls, resulting in channel

⁴⁶ Can files be renamed across directories on all platforms? If not either use a common directory or copy the files.

termination and connection churn, which is not ideal. Another option is to stop the network, but this seems like costly overkill. Instead the network should “pause” on a signal that the database is pending a checkpoint operation.

Design

Given that the database supports concurrent write, it does not operate on strands. Blockchain sequential validation operates on a single strand to ensure order and non-concurrency (with fan-out and return on a priority threadpool for parallelized validation steps). The database provides the threadpool for the single blockchain validation strand. The network (and server) operates on an independent threadpool. The objective is to suspend database writes while not stopping the network, apart from delaying the completion of its requested writes. Given that the network cannot depend upon any specific completion time, this is sufficient.

In terms of denial of service, it is left to the network (or server) to manage the amount of work it is accumulating and either suspend network reads, drop messages, or drop the peer/client, as appropriate. The database threadpool has no guard on its work queue, as it has no knowledge of which peers/clients are submitting work, and should not.

Checkpointing also cannot stop the database threadpool, as otherwise reads would be unnecessarily blocked. So after being notified (with a completion handler) that a checkpoint is pending, the query interface wraps and accumulates commits on a subscriber queue and monitors for completion of all posted commits. When there are no outstanding commit handlers, the query interface invokes the completion handler for checkpoint start, providing its own completion handler for checkpoint stop.

Upon invocation of the checkpoint stop handler, the query interface notifies its subscriber queue to post all jobs to the database threadpool, with new jobs being posted directly. At this point all operations have resumed.

Monitoring the completion of all writes is done across all tables, as checkpointing is comprehensive. However individual table memory remap follows the same pattern. By monitoring for the completion of all reads or writes on a given table, this technique can be used to eliminate the database remap lock. This greatly facilitates the coding of operations across tables, as there is no need for care in terms of deadlock risk.

Monitoring store writes (for checkpointing) and individual table reads/writes (for remapping) is accomplished by injecting an object smart pointer into each query and handling the destruct of that object. For individual tables, the object is the instance of the table’s query class. The store write monitoring object is special purpose. The former is required for any query to execute for that table, and the latter is injected into database write closures received from the network.

By queuing all database requests, no continuity is lost. By monitoring them there is a clear indication of completion. And while the network may continue to operate as fully as possible, denial of service management remains within its control.

Header regeneration

It is possible to recover all headers (and height indexes) from a consistent set of logical table sizes. So the checkpoint files can be reduced to a single file that contains this logical size for each table. The size must be collected following a flush of all tables, with writes suspended. The recovery for headers can be

implemented in the header template for all tables, as it is entirely generic code. This should probably take about ten minutes for the full chain, as it's only walking and recovering pointers, and can proceed in parallel across tables.

Summary

Because headers are random access memory mapped files, their total corruption is assumed. These are replaced by a consistent set of headers (checkpoint) upon startup.

Because data files are append-only memory maps, all header references remain valid. Unindexed data is orphaned consistently and overwritten as normal operations proceed. Upon controlled shutdown, the store reduces data files to logical size (as is currently done).

This approach is based on one important assumption about memory mapped files. This assumption should be verified before implementing.

- It requires that memory maps cannot corrupt flushed file data that is never again written in memory.
- mmap may flush pages at a time, which may overlap previously-flushed (therefore unmodified) file data.
- **We assume that flushing the unmodified part of a modified page will not alter that part of the file.**

This also assumes that the file may be copied while open (assuming the map has been flushed with no subsequent writes), though this should be a safe assumption. Otherwise reads would have to be suspended to close the file, which would also dump all loaded pages, and significantly slow checkpointing.

The store is suspended in a consistent state during checkpointing, without adversely affecting network operations.

This fully resolves all four problems described above.