Libbitcoin Version 4 Relational Database

Version 3 November 10, 2021

The libbitcoin-database database primitives are:

- **Array**
- **Blob**
- **Hashmap**
- **Multimap**

The primary implementation is over a set of memory mapped files. These produce very rapid read/write response given sufficient available RAM. This is ideal for high performance applications that require peak performance. However for wallet scenarios it is not always ideal. Memory map fault recovery requires append-only writes and a snapshotting system. This results in greater storage utilization than logically necessary. Furthermore paging costs can be high on low memory systems. To bridge these differences we design the store with an interface that can support a replaceable back end.

SQL Option

The mmap tables do not map perfectly to a set of standard relational tables, as the design is optimized for domain-specific constant time queries, compact storage, in-memory offset navigation, and append-only write operations. However, the design is strictly relational. With a little care this will give us the ability to drop in an in-process relational database (such as SQLite, a public domain C library) behind the query interface.

> *SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.*

The primary advantage of SQLite is the opportunity to perform dynamic compression and pruning, while also providing good fault tolerance and relational query support. This is ideal for a wallet scenario, especially on limited resource devices (in which case the engine is likely to be preinstalled). The mmap store is highly-optimized for mining and blockchain navigation servers, but requires explicit indexation for additional query support. General query is not at all required for mining, but can be very useful in wallet scenarios. The main disadvantage is the performance overhead. Concurrent read is supported, but concurrent writers are not. This only materially affects the bootstrap period(s). The snapshot operation requires write cessation, just as with the mmap design.

With this in mind the following sections relate the mmap table design to implementation in a relational database. No tradeoffs have propagated to the mmap design, it is fully optimized for its own requirements.

Navigation (mmap and SQL)

Keys

Primary Key (PK)

An arbitrary (database generated) unique row reference.

Foreign Key (FK)

A reference to the PK of another table. Foreign keys to record tables are exposed to public query iteration. Slab keys cannot be guarded, so parent FK or instance NK values are returned instead.

Natural Key (NK)

A value derived from the properties of a row, that unique identifies the row. The only natural keys exposed by the blockchain, and therefore the store, are block (header) hash and transaction hash, with input/output offset providing a public refinement of transaction hash to obtain the contained object by relative position. The coinbase transaction of a block is obtained from the first iterator element of a txs search.

Search Key (SK)

A value that is indexed for table search. A value is provided to the hashmap search function to retrieve elements with a matching SK. A hashmap returns 0..1 matching elements, while a multimap returns an iterator over 0..n matching elements.

Composite Key (CK)

A composite key is a search key that operates over more than one property of a row.

Natural Point (NP)

A natural point is a composite key (CK) consisting of a transaction hash (NK) and an input or output ordinal positional (index) within that transaction. This uniquely identifies an input or output.

Foreign Point (FP)

A foreign point is a composite key (CK) consisting of a transaction FK and an input or output ordinal positional (index) within that transaction. This is an efficient hybrid form of a NP.

Joins

Pointed arrows denote FK navigation. These do not pass through a search (hash) function and instead traverse links (converted to memory pointers) to the record or slab offset within a table. Unpointed arrows denote table navigation via SK, which transits a hash table header. Where these are of the form *SK_table_FK* both search of the key and link from it are available. A public search is entered by NK. Follow-on mmap navigation relies on either link navigation by FK or search by FK. SQL navigation relies on table joins between foreign and search keys, which requires storage of a PK value in associated tables. The mmap tables do not store their own PKs, relying instead on record and slab file offsets, though the hash tables pay a comparable cost in link storage.

Objects In

There are only two initial entry points to the store, Header and Transaction NK (hash) search, as these are the only natural keys exposed by the chain. Additionally, outputs may be reached through optional address indexing by script hash (the NK of any output), transactions may be reached by optional compact hash indexing, and following confirmation, a header may also be reached by confirmed height.

There are also only two points at which FK search is required to navigate the archive, Header->Txs and Output->Input. This is a necessary condition of headers first processing of blocks (headers are archived before their transactions arrive) and of concurrent download and validation (outputs may be archived before their spending inputs). The append-only nature of the store combined with out-of-order arrival requires that these two associations be deferred, which requires search.

Txs Table

The Txs set for a given block is read as a single slab. This reduces the size of an otherwise multimap by eliminating link traversal for every transaction_FK in a block. Slab storage of this association does not increase the link size for the table over use of a multimap, but eliminates 699,300,000 4 byte links (one per block vs. one per transaction).

Point Table

In the wire serialization, input references to outputs contain the output's transaction hash. This is collapsed into the Point table so that only one instance of the hash is retained for all spenders of the transaction's outputs. With an average of two outputs per transaction, this reduces the instance of transaction hash storage by 1/3, a savings of ~17GB. Collapsing to a single storage is possible, but due to parallelism this introduces complexity to avoid at this point. The Point table is prunable along with the inputs and outputs, when a transaction is fully spent.

If there is no entry in the table for a given transaction hash, no outputs of the transaction are yet spent. Otherwise the Input table is searched by FP for all spends of the output. There is an archival race between Point and Transaction/Input, so it's possible that no spender may yet be located despite a point, which is ok.

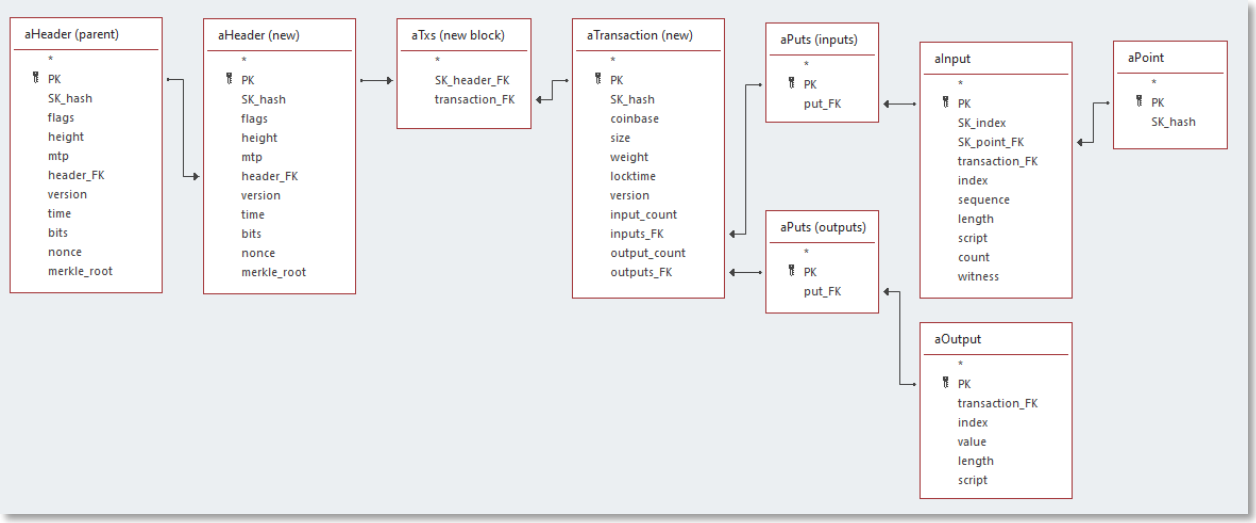
This optimization can be deferred, placing the point hash directly into the input and defining the input search key as a natural point (NP) as opposed to a foreign point (FP).

Archival

Header archival writes a single table row. Block and transaction archivals are performed independently, with header archival first.

Transactions may be archived before or after an associating header (and may never become associated). Transaction commitment is ordered such that none of its data is reachable until fully populated. The final commit is the Hash table, which exposes the transaction to search by its NK.

Block association of transactions always performed with the header and transactions already archived. Block association is a single row in the Txs table, atomically linked. Once it is linked the block is populated and fully reachable by its NK.



In this diagram (only) arrows denote the derivation of FKs for archival (i.e. the reverse of navigation).

Transaction Commit

Ordered archival ensures that no object is archived until it is fully composed, allowing it to be navigated upon an atomic pointer swap in an index exposed to NK search. Headers and Txs are single rows, with Txs only committed the block header and all of its transactions. Transactions include circular references, as inputs and outputs contain their composing transaction's FK for reverse navigation. This presents a complexity in their commit.

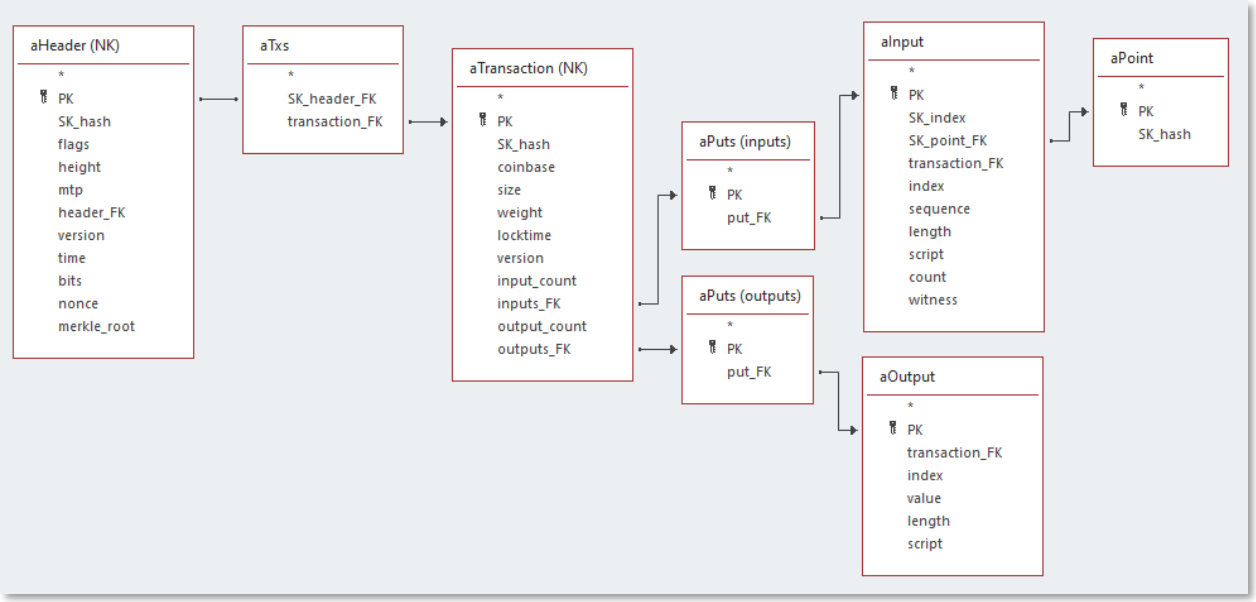
The transaction is allocated, which produces its FK, but commit is deferred. Then its components are committed in reverse order, propagating FKs back to the transaction. Subsequently the transaction is written and committed. However, Input is searchable by its previous output FP. So by committing in order this would expose an untraversable link (to the uncommitted transaction). As the memory is allocated the link is safe, however the transaction must be serialized by the time the input is committed. So the order is as follows:

- Allocate Transaction, obtain FK (non-navigable/searchable).
- Write and commit Output table, using transaction_FK, and obtain FKs (non-navigable).
- Write and commit Point (hash) as necessary, and obtain FK (can search Input, but not found).
- Write Input table, using transaction_FK and prevout_FK, and obtain FKs (non-navigable/searchable).
- Write and commit Puts table, using input and output FKs, obtain two FKs (non-navigable).
- Write Transaction, using Puts FKs and their count (navigable/non-searchable).
- Commit Inputs (searchable, with navigation to transaction).
- Commit Transaction (searchable).

By deferring commit of Transaction until after Input commit it is ensured that if a transaction exists its spends are navigable. Otherwise we could locate a spending transaction and not find its spends. This implies a simple two-phase commit capability be exposed by the hashmap implementation on a per element basis, which already exists. The write phase is internally isolated from the atomic swap (commit phase).

Objects Out

Block and transaction store deserialization consists of straightforward navigation of FKs, with the sole exception of the **Block=>Txs** search on header_FK. This search returns 0..1 results, and provides the full and ordered set of transaction_FK values (4 bytes each) for follow-on FK navigation. There is no other archival data apart from what makes up a block.



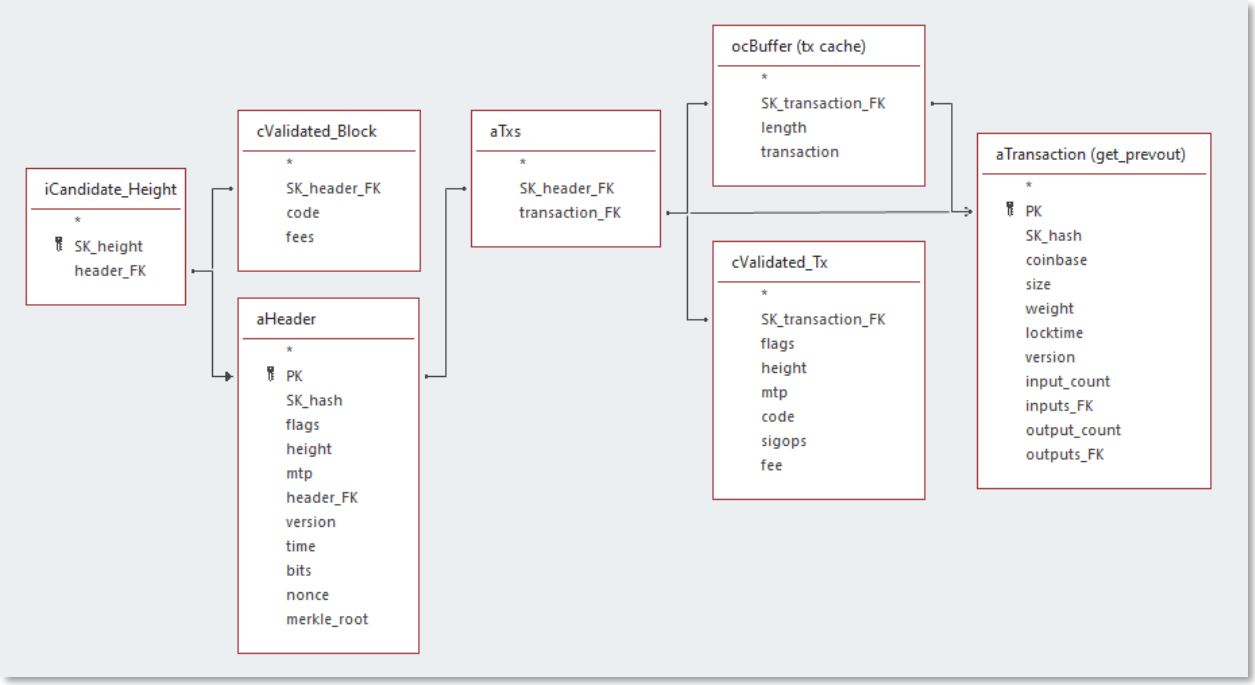
Deserialization of blockchain objects.

Validation

The validation chaser first queries for block invalidity at the given height in the candidate chain. Otherwise if the block is not accepted it checks all transactions for validated state. If an invalid transaction is found the block is invalidated. Otherwise unvalidated transactions are obtained from (1) memory cache, (2) disk cache (as shown), or deserialized. Each prevout that is not populated on the transaction (from cache) is then deserialized (see get_prevout subquery). Then a validation job is created for each transaction, with results collected by the validator.

UTXO Cache vs. Index

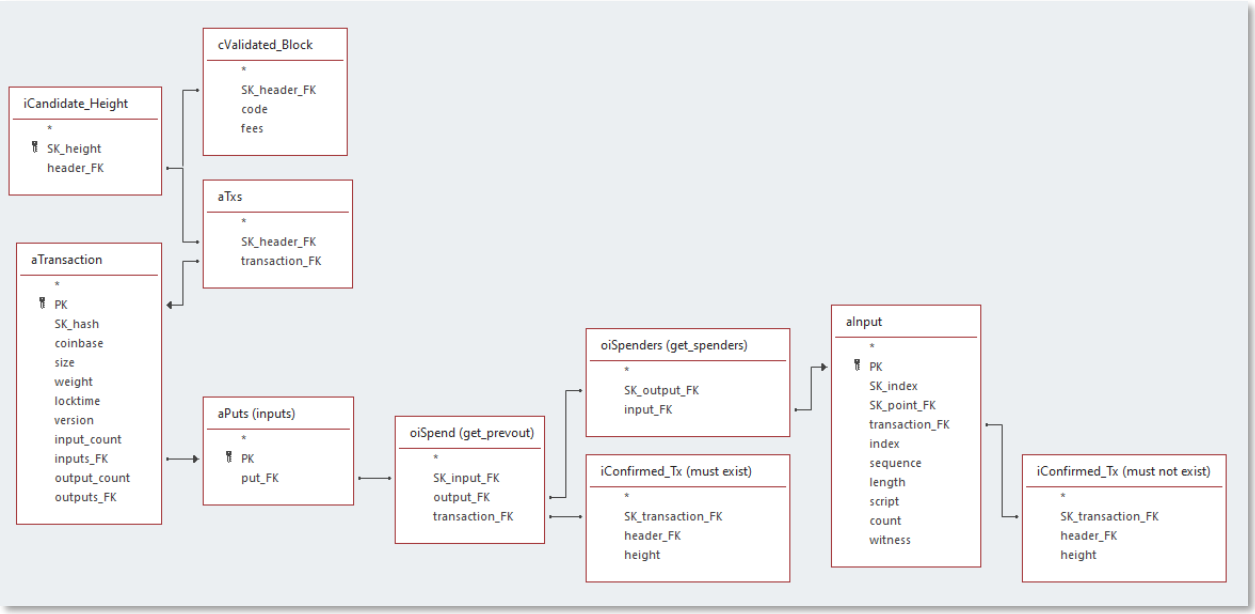
An unspent output cache is an index of unspent outputs, potentially reproducing their sizable payload. To recover an output requires a NP search (hash:index) identical in practice to searching for the output spent by an input, with the input indexed on its spend NP as opposed to the output indexed on its own NP.



Validation chaser queries.

Confirmation

The confirmation chaser follows deorganization of any pending fork. It first queries for block invalidity at the given height in the candidate chain. Otherwise if the block is accepted and not yet valid (from a previous reorganization and subsequent organization), it checks all outputs for confirmability.



Confirmation chaser queries.

The process is straightforward, as forked blocks have been deconfirmed at this point. All previous outputs are checked for confirmed state and all spenders of the outputs are checked for non-confirmed state. As the result of static double spend checks, there is no checking outputs that are within the block. These are all to-be-confirmed and are not double spent within the block. Height is preserved in the confirmation state, which provides for checking the confirmed-ness of coinbase prevouts (must be 100 blocks deep).

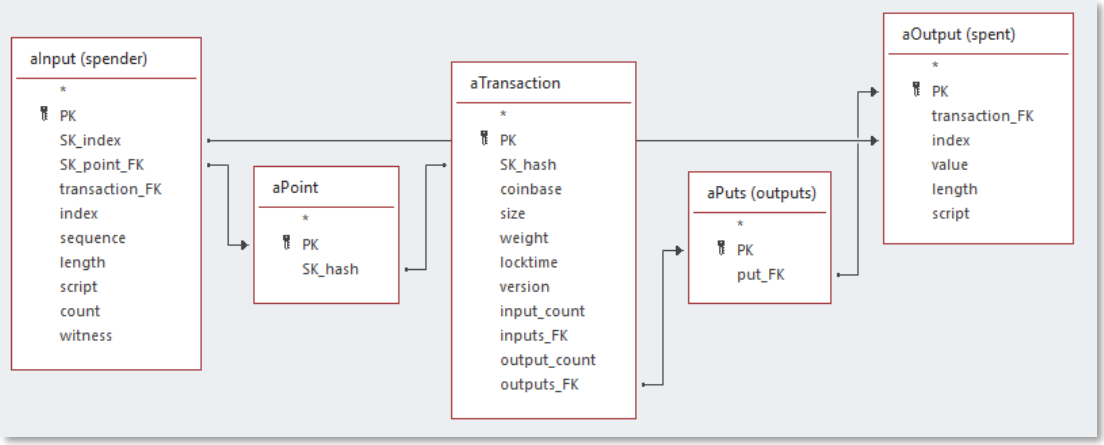
The diagram above shows abbreviated navigation through optional mining optimization tables. These index previously-queried **input->output** (get_prevout) and **output=>inputs** (get_spenders) relations. If these are not active (or not implemented) the two natural queries are used instead. These two indexes constitute ~50GB of storage at 700k blocks, but are optional.

Spend Index (vs. get_prevout)

During validation a transaction must at some point read the prevout for each of its inputs. These are populated upon transaction arrival, at which point the Spend index may be written. However in the case of block download (vs. pool) the prevout set may be incomplete. A partial set can be written, with others written as the prevouts arrive, until the final arrival results in transaction accept invocation. If the prevout never arrives this is trapped as invalidity by the validation chaser. So the Spend index is guaranteed by confirmation chaser arrival, for all transactions in each candidate block. However it's not obvious that the Spend index will actually be a net improvement in performance of the natural query, as it trades one NK search for a write and a FK search.

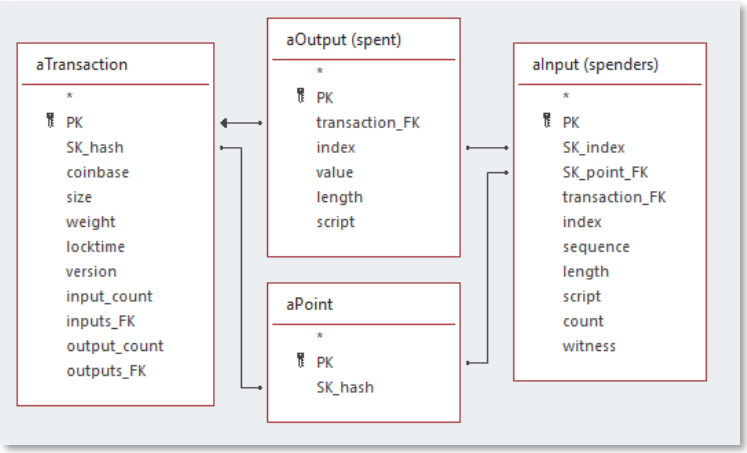
Spender Index (vs. get_spenders)

Double spenders are never obtained during validation, so there is no point in which the information necessary to populate the Spenders index would be at hand. If the information was available, it would be sufficiently accumulated by the arrival of the validation chaser, but it may be over-accumulated by higher candidate blocks, so spender height would need to be cached with each record. The trade would replace one NK and one FP search with an index write and FK search, so also not an obvious win.



The get_prevout subquery

This query is simpler in the mmap model than it appears above. There is no composite key join on the output object. This merely shows the data relations clearly. The spender uses its point_FK to obtain the prevout transaction by NK query, then navigates to the index of the intended previous output, and finally to the output. This is straightforward link navigation coupled with one NK search.

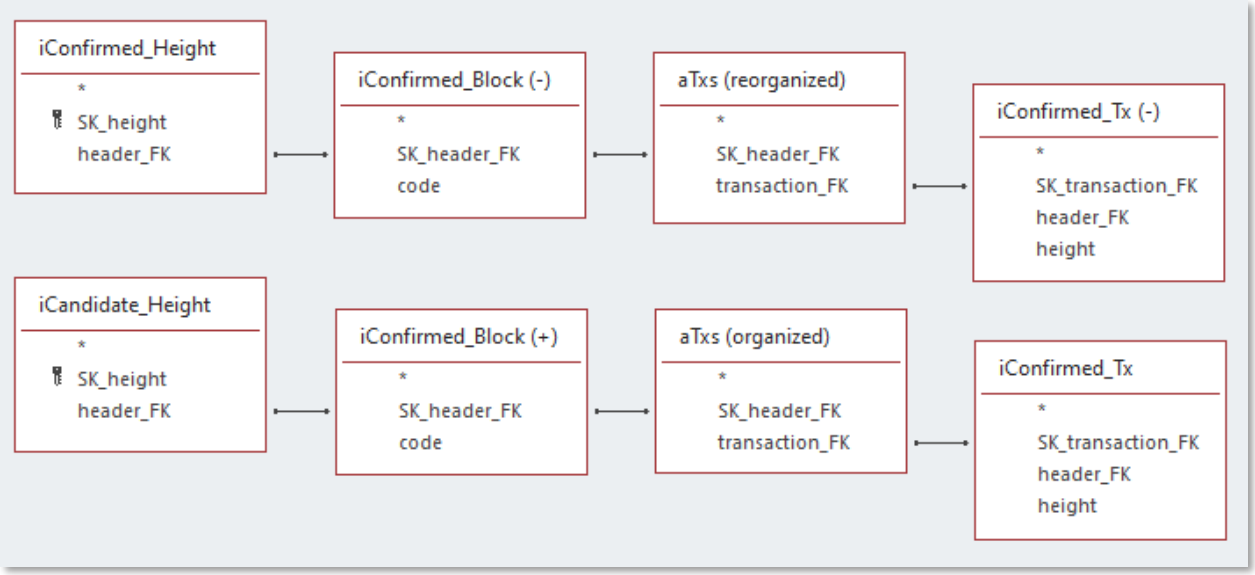


The get_spenders subquery

To obtain spenders of the output, the point table is queried with the transaction hash of the output. A FP is then constructed from the point_FK and the output's index, and Inputs is then searched for all spenders of the output represented by the FP.

Organization

Organization and reorganization of blocks is straightforward. States are merely set or unset (hidden) on the block and transaction objects.



Confirmation and deconfirmation of blocks and transactions.

The first and last associations in each path are never searched. Changing confirmation states requires only a write, no read, as the previous entry is just hidden. Though writing an entry does require the hash function and offset computation, it avoids navigation of the conflict list. The second search could be avoided in each by storing the Txs FK in the confirmed height array, and by rewriting the candidate height with Txs FK values once they are populated (height indexes are mutable).

Other Optionals

Other tables provide optional features (Address indexation, Utreexo proofs, Neutrino filters) and performance optimizations. The performance optimizations are bootstrap acceleration with DoS protection, compact block validation acceleration, and tx buffering to reduce RAM consumption during concurrent block validation. Input and output to transaction indexes are also optional optimizations (see Confirmation).

Table Data Structures (mmap)

The following data structures are applied to a memory map using a set of simple templates, called primitives. These consist of a hash table template, hash table header template, record, multimap, and slab manager templates, and list, list element and list iterator templates.

List templates implement the hash conflict stack (as a singly-linked list), managers implement link-to-pointer mapping, the header implements hash buckets, and the table combines the various templates with one or two file objects. The managers control atomic memory swaps for element insertion and provide reader/writer byte streams for callers.

All writes with the exception of the two height indexes and hash table headers are append only. By isolating all headers and these indexes to index (.idx) files, and retaining all bodies in independent files, we can provide fault recovery (with store writes acquiesced) by copying the rather small set of index files (3% to 4% of body storage depending on configuration). We perform an swap of previous snapshots with an atomic file system rename operation.

These are variations of code used in the previous two releases of libbitcoin, so we have quite a bit of experience with their operation over mmap. They are simple and well-tested, though some minor modifications are required to implement pure append-only as intended.

Array Table

The array is the simplest table. The header is a position value, indicating the write point, which is also the logical body size. The array is the only not inherently append only table, though it is also used in that manner.

Array						
Header			Position	7	4	Header file offset (0)
Body		Elements	0	pizza	8	Body file offset (4)
			1	candy	16	
Position size	4		2	eggplant	24	
Value size	8	get[3]->	3	beer	32	
			4	milk	40	
			5	veggies	48	
			6	cake	56	
		Position	7		64	
			8		72	
			9		80	
			10		88	
			11		96	
			12		104	
			13		112	
			14		120	
			15		128	
			16		136	
			17		144	
			18		152	
			...		...	
			96		776	
			97		784	
			98		792	
			99		800	
Reservation (100 records)						Body file size (800 bytes)

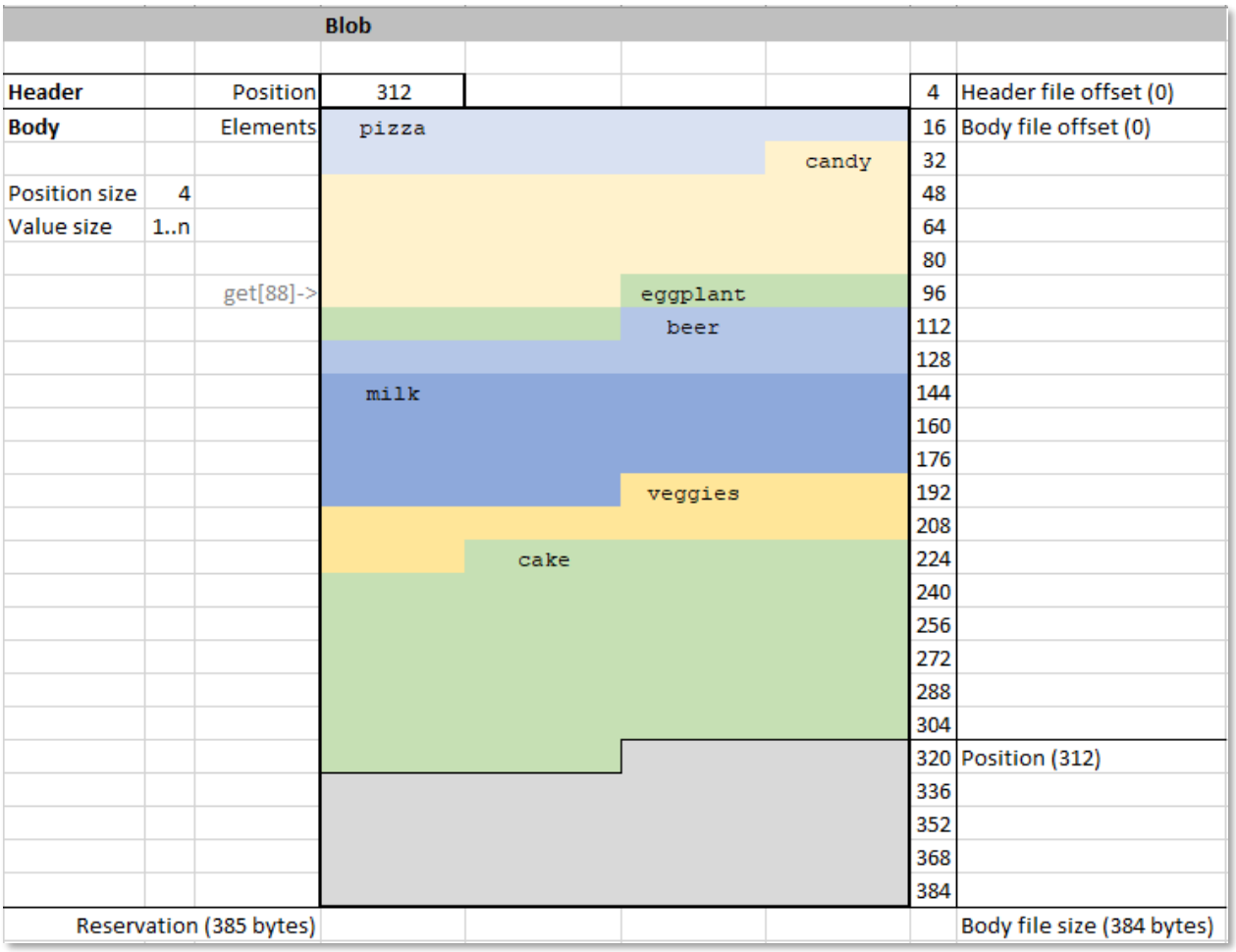
In the current state of the table:

- **get[3].value = beer**

These are used for height indexing, and the block at any given height may change through reorganization. These are very small bodies, smaller than other headers. So we simply snapshot the entire table for recovery, storing both header and body in the same file and memory map. Despite the presentation limits, value sizes are not multiples of links sizes and can be any byte-delimited size. The table is resized to the corresponding byte offset before the file is closed.

Blob Table

A blob is similar to an array, but must be externally indexed. As value fields are not length prefixed, an empty value is not storable. The header is a position value, indicating the write point, which is also the logical body size.



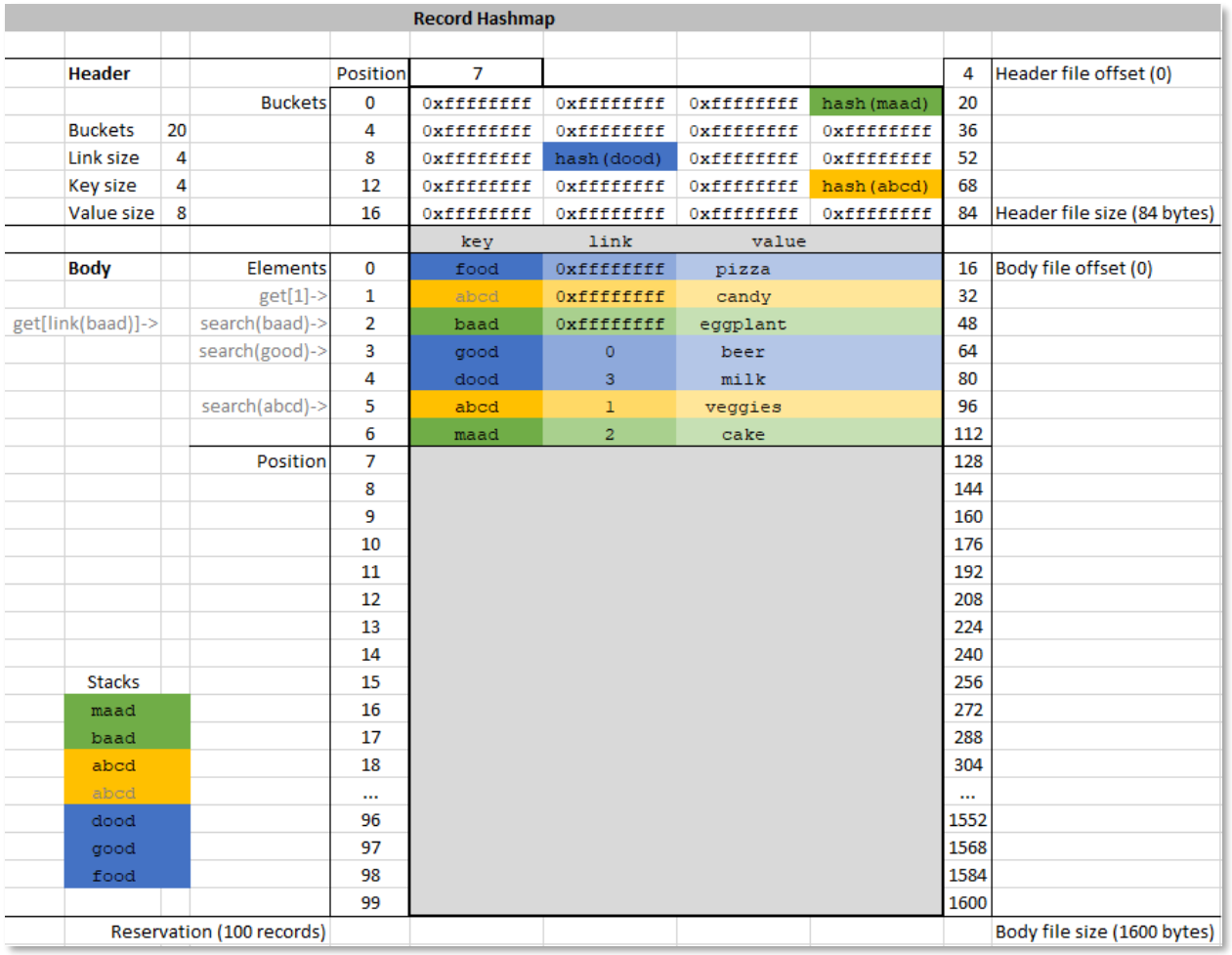
In the current state of the table:

- **get[88].value = eggplant**

As value fields are not length prefixed, an empty value is not storable, typically varint is used to determine size. Despite the presentation limits, value sizes are not multiples of links sizes and can be any byte-delimited size. The table is resized to the corresponding byte offset before the file is closed.

Record Hashmap

The hashmap is a basic hash table with fixed-size values. We use the name hashmap to avoid conflict with the term “table”, which is used to refer to all header/data pairings.



Record Hashmap

Header			Position	7				4	Header file offset (0)
		Buckets	0	0xffffffff	0xffffffff	0xffffffff	hash(maad)	20	
Buckets	20		4	0xffffffff	0xffffffff	0xffffffff	0xffffffff	36	
Link size	4		8	0xffffffff	hash(dood)	0xffffffff	0xffffffff	52	
Key size	4		12	0xffffffff	0xffffffff	0xffffffff	hash(abcd)	68	
Value size	8		16	0xffffffff	0xffffffff	0xffffffff	0xffffffff	84	Header file size (84 bytes)
				key	link	value			
Body		Elements	0	food	0xffffffff	pizza		16	Body file offset (0)
		get[1]->	1	abcd	0xffffffff	candy		32	
get[link(baad)]->		search(baad)->	2	baad	0xffffffff	eggplant		48	
		search(good)->	3	good	0	beer		64	
			4	dood	3	milk		80	
		search(abcd)->	5	abcd	1	veggies		96	
			6	maad	2	cake		112	
		Position	7					128	
			8					144	
			9					160	
			10					176	
			11					192	
			12					208	
			13					224	
			14					240	
Stacks			15					256	
maad			16					272	
baad			17					288	
abcd			18					304	
abcd			...					...	
dood			96					1552	
good			97					1568	
food			98					1584	
			99					1600	
Reservation (100 records)									Body file size (1600 bytes)

In the example we have a poor hash function. It ignores the first character of the search key (key). The key size is a constant byte length, in this case four bytes. Because of the hash function, any two keys with the same last three characters are hash collisions. However, if they have all same characters the treatment is the same as far as the header is concerned.

The previous “head” link (from the bucket) is assigned to the new element's “next” link. If the bucket is empty then this is the sentinel value. Then the position (link) of the new element is written to the bucket. This results in a linked list that forms a stack, with the top position retained in the bucket. A preallocated record is assigned to the writer and then the pair of links is swapped, making the record searchable.

In the current state of the table:

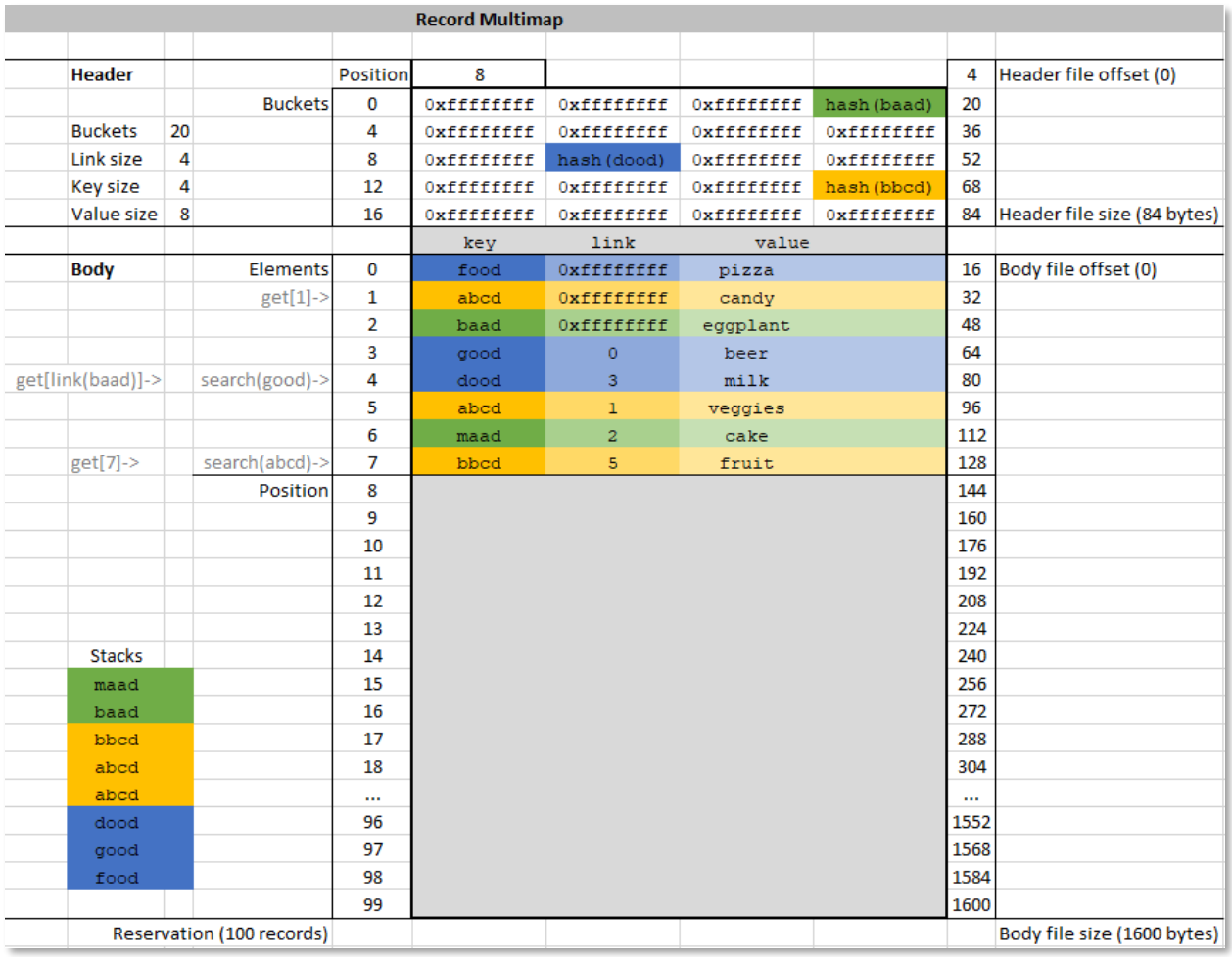
- **get[1].value = candy**
- **get[link(baad)].value = eggplant**
- **search(abcd).value = veggies**
- **search(good).value = beer**
- **key(5) = abcd**
- **link(abcd) = 5**
- **link(key(5)) = 5**
- **key(link(abcd)) = abcd**

Note that the key (SK) and link (FK) are exposed as properties of the record. A call to get[FK] bypasses the hash search and is mapped directly to the element. So the table operates as a key-link bimap, in addition to retaining values associated with either.

Despite the presentation limits, value sizes are not multiples of links sizes and cany be any byte-delimited size.

Record Multimap

The multimap is a record hashmap with a logical stack of values per key. The stack is implemented using the same list iterator as the deconfliction stack. A multimap doesn't support logical delete, as values cannot be removed from the middle of a value stack. They could be logically popped, although that is not an envisioned use of the multimap.



In the example we have the same poor hash function.

Instead of the table iterating the deconfliction list, an iterator over the deconfliction list is returned to the caller. This iterator simply skips over non-matching elements, returning the value for each matched key. Apart from conflicts the number of table entries does not affect search cost. The only distinction from a hashmap is return of multiple possible matches vs. only the first.

In the current state of the table:

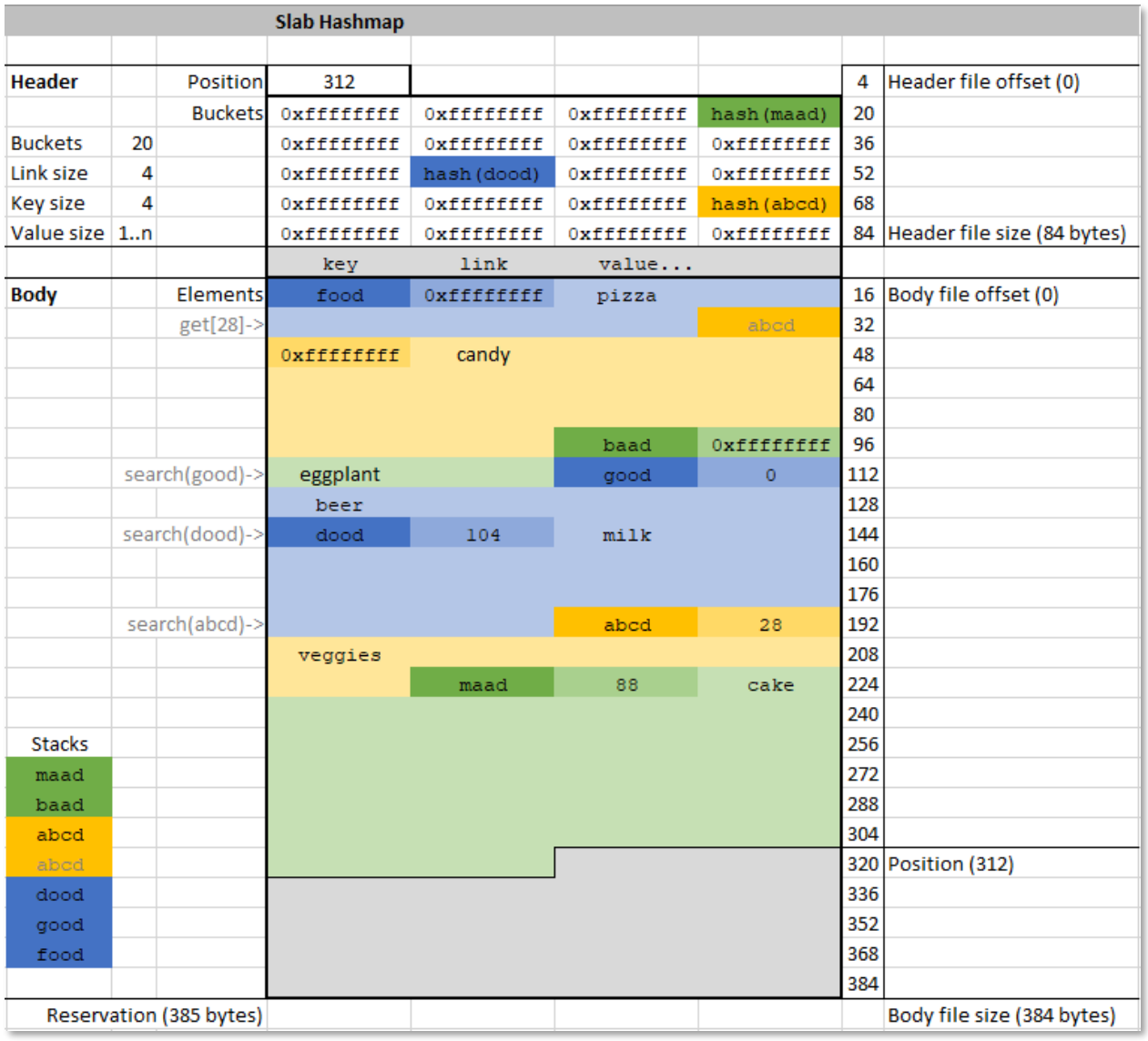
- **get[1].values = candy**
- **get[7] = fruit, veggies, candy (conflict stack)**
- **get[link(baad)].values = eggplant**
- **search(abcd).values = veggies, candy (value stack)**
- **search(good).values = beer**
- **key(5) = abcd**
- **link(abcd) = 5**
- **link(key(5)) = 5**
- **key(link(abcd)) = abcd**

Notice that element 1 is searchable, while in the hashmap it was hidden. Also notice that search(abcd) bypasses “bbcd” as a hash conflict. The search list can be also entered and iterated from any FK position, such as get[7], though all conflicts are iterated.

Despite the presentation limits, value sizes are not multiples of links sizes and cany be any byte-delimited size.

Slab Hashmap

The slab hashmap is nearly identical to the record hashmap, with the exception that values are variably-sized. As a consequence, all links are byte offsets, vs. record offsets. This causes the link domain to be consumed much faster, causing FKs to be larger. This is an impact on other tables that retain the keys. So records are preferrable where possible. The only slabs in archive data are scripts/witnesses and txs association.



In the example we have the same poor hash function.

In the current state of the table:

- **get[28].value = candy**
- **get[link(dood)].value = milk**
- **search(good).value = beer**
- **search(dood).value = milk**
- **search(abcd).value = veggies**
- **key(184) = abcd**
- **link(abcd) = 184**
- **link(key(184)) = 184**
- **key(link(abcd)) = abcd**

As value fields are not length prefixed, an empty value is not storable, typically varint is used to determine size. Despite the presentation limits, value sizes are not multiples of links sizes and cany be any byte-delimited size. The table is resized to the corresponding byte offset before the file is closed.

Files

Header Files (.idx)

- archive_header.idx
- archive_input.idx
- archive_output.idx
- archive_point.idx
- archive_puts.idx
- archive_transaction.idx
- archive_txs.idx
-
- candidate_height.idx
- confirmed_block.idx
- confirmed_height.idx
- confirmed_tx.idx
-
- option_address.idx
- option_bootstrap.idx
- option_buffer.idx
- option_compact.idx
- option_neutrino.idx
- option_spend.idx
- option_spenders.idx
- option_utreexo.idx
-
- validated_block.idx
- validated_tx.idx

Body Files (.dat)

- archive_header.dat
- archive_input.dat
- archive_output.dat
- archive_point.dat
- archive_puts.dat
- archive_transaction.dat
- archive_txs.dat
-
- confirmed_block.dat
- confirmed_tx.dat
-
- option_address.dat
- option_bootstrap.dat
- option_buffer.dat
- option_compact.dat
- option_neutrino.dat
- option_spend.dat
- option_spenders.dat
- option_utreexo.dat
-
- validated_block.dat
- validated_tx.dat

Notes

These MS Access table mock-ups can be populated with blockchain data, and defined queries against the tables can emit SQL statements for single table and joined queries. Following a detailed definition of each required query we can ensure that the design is complete and supported by SQLite. We may want to develop this support concurrently with the mmap design, and also investigate the requirements of an NVRAM store.