

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
				3		beer	32
		Value size	8	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)		

Notes

The array is the simplest table. The header is a position value, indicating the write point, which is also the logical body size. The table is resized to the corresponding byte offset before the file is closed.

The array is the only table that is not append only. 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.

In the current state of the table, the following result:

```
get[3].value = beer
```

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

Blob					
Header		Position	312	4	Header file offset (0)
Body		Elements	pizza candy eggplant beer milk veggies cake	16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304	Body file offset (0)
Position size	4	get[88]->			
Value size	1..n				
				320	Position (312)
				336	
				352	
				368	
				384	
Reservation (385 bytes)					Body file size (384 bytes)

Notes

A blob is simliar to an array, but must be externally indexed. The header is a position value, indicating the write point, which is also the logical body size.

The array is the only table that is not append only. 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.

In the current state of the table, the following result:

get[88].value = eggplant

A blob is simliar to an array, but must be externally indexed.
As value fields are not length prefixed, an empty value is not storable.

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.

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											
		15	256											
		16	272											
		17	288											
		18	304											
		...	...											
		96	1552											
		97	1568											
		98	1584											
		99	1600											
Reservation (100 records)							Body file size (1600 bytes)							

Stacks
maad
baad
abcd
abcd
dood
good
food

Notes

In the example we have a poor hash function. It ignores the first characted of the search key (key). The key size is a contant 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 chacters 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, the following result:

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

Notes

The multimap is a record hashmap with a stack of values. 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 a poor hash function. It ignores the first characted of the search key (key). Instead of the table iterating the deconfliction list, an iterator over the deconfliction list is returned to the caller. This 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.

In the current state of the table, the following result:

```
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								
Header		Position	312		4	Header file offset (0)		
		Buckets	0xffffffff	0xffffffff	0xffffffff	hash(maad)	20	
Buckets	20		0xffffffff	0xffffffff	0xffffffff	0xffffffff	36	
Link size	4		0xffffffff	hash(dood)	0xffffffff	0xffffffff	52	
Key size	4		0xffffffff	0xffffffff	0xffffffff	hash(abcd)	68	
Value size	1..n		0xffffffff	0xffffffff	0xffffffff	0xffffffff	84	
			key	link	value...			
Body	Elements		food	0xffffffff	pizza		16	
	get[28]->					abcd	32	
			0xffffffff	candy				48
								64
								80
	search(good)->				baad	0xffffffff	96	
			eggplant		good	0	112	
	search(dood)->		beer				128	
			dood	104	milk		144	
	search(abcd)->						160	
								176
					abcd	28	192	
			veggies				208	
			maad	88	cake	224		
						240		
						256		
						272		
						288		
						304		
						320		
						336		
						352		
						368		
						384		
						Position (312)		

Notes

In the example we have a poor hash function. It ignores the first characted of the search key (key). 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 preferable where possible. The only slabs in bitcoin are scripts/witnesses within inputs and outputs.

In the current state of the table, the following result:

```

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.