TDB2: A Redesigning The Trivial DataBase

Rusty Russell, IBM Corporation

~~26-July~~1-September-2010

Abstract

The Trivial DataBase on-disk format is 32 bits; with usage cases heading towards the 4G limit, that must change. This required breakage provides an opportunity to revisit TDB's other design decisions and reassess them.

1 Introduction

The Trivial DataBase was originally written by Andrew Tridgell as a simple key/data pair storage system with the same API as dbm, but allowing multiple readers and writers while being small enough (< 1000 lines of C) to include in SAMBA. The simple design created in 1999 has proven surprisingly robust and performant, used in Samba versions 3 and 4 as well as numerous other projects. Its useful life was greatly increased by the (backwards-compatible!) addition of transaction support in 2005.

The wider variety and greater demands of TDB-using code has lead to some organic growth of the API, as well as some compromises on the implementation. None of these, by themselves, are seen as show-stoppers, but the cumulative effect is to a loss of elegance over the initial, simple TDB implementation. Here is a table of the approximate number of lines of implementation code and number of API functions at the end of each year:

Year End	API Functions	Lines of C Code Implementation
1999	13	1195
2000	24	1725
2001	32	2228
2002	35	2481
2003	35	2552
2004	40	2584
2005	38	2647
2006	52	3754
2007	66	4398
2008	71	4768
2009	73	5715

This review is an attempt to catalog and address all the known issues with TDB and create solutions which address the problems without significantly increasing complexity; all involved are far too aware of the dangers of second system syndrome in rewriting a successful project like this.

2 API Issues

2.1 tdb_open_ex Is Not Expandable

The tdb_open() call was expanded to tdb_open_ex(), which added an optional hashing function and an optional logging function argument. Additional arguments to open would require the introduction of a tdb_open_ex2 call etc.

2.1.1 Proposed Solution

tdb_open() will take a linked-list of attributes:

```
enum tdb_attribute {
    TDB_ATTRIBUTE_LOG = 0,
    TDB_ATTRIBUTE_HASH = 1
};
struct tdb_attribute_base {
    enum tdb_attribute attr;
    union tdb_attribute *next;
};
struct tdb_attribute_log {
    struct tdb_attribute_base base; /* .attr = TDB_ATTRIBUTE_LOG */
    tdb_log_func log_fn;
    void *log_private;
};
struct tdb_attribute_hash {
    struct tdb_attribute_base base; /* .attr = TDB_ATTRIBUTE_HASH */
    tdb_hash_func hash_fn;
    void *hash_private;
};
union tdb_attribute {
    struct tdb_attribute_base base;
    struct tdb_attribute_log log;
    struct tdb_attribute_hash hash;
};
```

This allows future attributes to be added, even if this expands the size of the union.

2.2 tdb_traverse Makes Impossible Guarantees

tdb_traverse (and tdb_firstkey/tdb_nextkey) predate transactions, and it was thought that it was important to guarantee that all records which exist at the start and end of the traversal would be included, and no record would be included twice.

This adds complexity (see3.12) and does not work anyway for records which are altered (in particular, those which are expanded may be effectively deleted and re-added behind the traversal).

2.2.1 Proposed Solution

Abandon the guarantee. You will see every record if no changes occur during your traversal, otherwise you will see some subset. You can prevent changes by using a transaction or the locking API.

2.3 Nesting of Transactions Is Fraught

TDB has alternated between allowing nested transactions and not allowing them. Various paths in the Samba codebase assume that transactions will nest, and in a sense they can: the operation is only committed to disk when the outer transaction is committed. There are two problems, however:

1. Canceling the inner transaction will cause the outer transaction commit to fail, and will not undo any operations since the inner transaction began. This problem is soluble with some additional internal code.

2. An inner transaction commit can be cancelled by the outer transaction. This is desirable in the way which Samba's database initialization code uses transactions, but could be a surprise to any users expecting a successful transaction commit to expose changes to others.

The current solution is to specify the behavior at tdb_open(), with the default currently that nested transactions are allowed. This flag can also be changed at runtime.

2.3.1 Proposed Solution

Given the usage patterns, it seems that the "least-surprise" behavior of disallowing nested transactions should become the default. Additionally, it seems the outer transaction is the only code which knows whether inner transactions should be allowed, so a flag to indicate this could be added to tdb_transaction_start. However, this behavior can be simulated with a wrapper which uses tdb_add_flags() and tdb_remove_flags(), so the API should not be expanded for this relatively-obscure case.

2.4 Incorrect Hash Function is Not Detected

tdb_open_ex() allows the calling code to specify a different hash function to use, but does not check that all other processes accessing this tdb are using the same hash function. The result is that records are missing from tdb_fetch().

2.4.1 Proposed Solution

The header should contain an example hash result (eg. the hash of 0xdeadbeef), and tdb_open_ex() should check that the given hash function produces the same answer, or fail the tdb_open call.

2.5 tdb_set_max_dead/TDB_VOLATILE Expose Implementation

In response to scalability issues with the free list (3.5) two API workarounds have been incorporated in TDB: tdb_set_max_dead() and the TDB_VOLATILE flag to tdb_open. The latter actually calls the former with an argument of "5".

This code allows deleted records to accumulate without putting them in the free list. On delete we iterate through each chain and free them in a batch if there are more than max_dead entries. These are never otherwise recycled except as a side-effect of a tdb_repack.

2.5.1 Proposed Solution

With the scalability problems of the freelist solved, this API can be removed. The TDB_VOLATILE flag may still be useful as a hint that store and delete of records will be at least as common as fetch in order to allow some internal tuning, but initially will become a no-op.

2.6 TDB Files Cannot Be Opened Multiple Times In The Same Process

No process can open the same TDB twice; we check and disallow it. This is an unfortunate side-effect of fcntl locks, which operate on a per-file rather than per-file-descriptor basis, and do not nest. Thus, closing any file descriptor on a file clears all the locks obtained by this process, even if they were placed using a different file descriptor!

Note that even if this were solved, deadlock could occur if operations were nested: this is a more manageable programming error in most cases.

2.6.1 Proposed Solution

We could lobby POSIX to fix the perverse rules, or at least lobby Linux to violate them so that the most common implementation does not have this restriction. This would be a generally good idea for other fcntl lock users.

Samba uses a wrapper which hands out the same tdb_context to multiple callers if this happens, and does simple reference counting. We should do this inside the tdb library, which already emulates lock nesting internally; it would need to recognize when deadlock occurs within a single process. This would create a new failure mode for tdb operations (while we currently handle locking failures, they are impossible in normal use and a process encountering them can do little but give up).

I do not see benefit in an additional tdb_open flag to indicate whether reopening is allowed, as though there may be some benefit to adding a call to detect when a tdb_context is shared, to allow other to create such an API.

2.7 TDB API Is Not POSIX Thread-safe

The TDB API uses an error code which can be queried after an operation to determine what went wrong. This programming model does not work with threads, unless specific additional guarantees are given by the implementation. In addition, even otherwise-independent threads cannot open the same TDB (as in 2.6).

2.7.1 Proposed Solution

Reachitecting the API to include a tdb_errcode pointer would be a great deal of churn; we are better to guarantee that the tdb_errcode is per-thread so the current programming model can be maintained.

This requires dynamic per-thread allocations, which is awkward with POSIX threads (pthread_key_create space is limited and we cannot simply allocate a key for every TDB).

Internal locking is required to make sure that fcntl locks do not overlap between threads, and also that the global list of tdbs is maintained.

The aim is that building tdb with -DTDB_PTHREAD will result in a pthread-safe version of the library, and otherwise no overhead will exist.

2.8 *_nonblock Functions And *_mark Functions Expose Implementation

CTDB[1] wishes to operate on TDB in a non-blocking manner. This is currently done as follows:

1. Call the _nonblock variant of an API function (eg. tdb_lockall_nonblock). If this fails:
2. Fork a child process, and wait for it to call the normal variant (eg. tdb_lockall).
3. If the child succeeds, call the _mark variant to indicate we already have the locks (eg. tdb_lockall_mark).

[1] Clustered TDB, see http://ctdb.samba.org

4. Upon completion, tell the child to release the locks (eg. tdb_unlockall).

5. Indicate to tdb that it should consider the locks removed (eg. tdb_unlockall_mark).

There are several issues with this approach. Firstly, adding two new variants of each function clutters the API for an obscure use, and so not all functions have three variants. Secondly, it assumes that all paths of the functions ask for the same locks, otherwise the parent process will have to get a lock which the child doesn't have under some circumstances. I don't believe this is currently the case, but it constrains the implementation.

2.8.1 Proposed Solution

Implement a hook for locking methods, so that the caller can control the calls to create and remove fcntl locks. In this scenario, ctdbd would operate as follows:

1. Call the normal API function, eg tdb_lockall().

2. When the lock callback comes in, check if the child has the lock. Initially, this is always false. If so, return 0. Otherwise, try to obtain it in non-blocking mode. If that fails, return EWOULDBLOCK.

3. Release locks in the unlock callback as normal.

4. If tdb_lockall() fails, see if we recorded a lock failure; if so, call the child to repeat the operation.

5. The child records what locks it obtains, and returns that information to the parent.

6. When the child has succeeded, goto 1.

This is flexible enough to handle any potential locking scenario, even when lock requirements change. It can be optimized so that the parent does not release locks, just tells the child which locks it doesn't need to obtain.

It also keeps the complexity out of the API, and in ctdbd where it is needed.

2.9 tdb_chainlock Functions Expose Implementation

tdb_chainlock locks some number of records, including the record indicated by the given key. This gave atomicity guarantees; no-one can start a transaction, alter, read or delete that key while the lock is held.

It also makes the same guarantee for any other key in the chain, which is an internal implementation detail and potentially a cause for deadlock.

2.9.1 Proposed Solution

None. It would be nice to have an explicit single entry lock which effected no other keys. Unfortunately, this won't work for an entry which doesn't exist. Thus while chainlock may be implemented more efficiently for the existing case, it will still have overlap issues with the non-existing case. So it is best to keep the current (lack of) guarantee about which records will be effected to avoid constraining our implementation.

2.10 Signal Handling is Not Race-Free

The tdb_setalarm_sigptr() call allows the caller's signal handler to indicate that the tdb locking code should return with a failure, rather than trying again when a signal is received (and errno == EAGAIN). This is usually used to implement timeouts.

Unfortunately, this does not work in the case where the signal is received before the tdb code enters the fcntl() call to place the lock: the code will sleep within the fcntl() code, unaware that the signal wants it to exit. In the case of long timeouts, this does not happen in practice.

2.10.1 Proposed Solution

The locking hooks proposed in2.8.1 would allow the user to decide on whether to fail the lock acquisition on a signal. This allows the caller to choose their own compromise: they could narrow the race by checking immediately before the fcntl call.[2]

2.11 The API Uses Gratuitous Typedefs, Capitals

typedefs are useful for providing source compatibility when types can differ across implementations, or arguably in the case of function pointer definitions which are hard for humans to parse. Otherwise it is simply obfuscation and pollutes the namespace.

Capitalization is usually reserved for compile-time constants and macros.

TDB_CONTEXT There is no reason to use this over 'struct tdb_context'; the definition isn't visible to the API user anyway.

TDB_DATA There is no reason to use this over struct TDB_DATA; the struct needs to be understood by the API user.

struct TDB_DATA This would normally be called 'struct tdb_data'.

enum TDB_ERROR Similarly, this would normally be enum tdb_error.

[2]It may be possible to make this race-free in some implementations by having the signal handler alter the struct flock to make it invalid. This will cause the fcntl() lock call to fail with EINVAL if the signal occurs before the kernel is entered, otherwise EAGAIN.

2.11.1 Proposed Solution

None. Introducing lower case variants would please pedants like myself, but if it were done the existing ones should be kept. There is little point forcing a purely cosmetic change upon tdb users.

2.12 tdb_log_func Doesn't Take The Private Pointer

For API compatibility reasons, the logging function needs to call tdb_get_logging_private() to retrieve the pointer registered by the tdb_open_ex for logging.

2.12.1 Proposed Solution

It should simply take an extra argument, since we are prepared to break the API/ABI.

2.13 Various Callback Functions Are Not Typesafe

The callback functions in tdb_set_logging_function (after 2.12 is resolved), tdb_parse_record, tdb_traverse, tdb_traverse_read and tdb_check all take void * and must internally convert it to the argument type they were expecting.

If this type changes, the compiler will not produce warnings on the callers, since it only sees void *.

2.13.1 Proposed Solution

With careful use of macros, we can create callback functions which give a warning when used on gcc and the types of the callback and its private argument differ. Unsupported compilers will not give a warning, which is no worse than now. In addition, the callbacks become clearer, as they need not use void * for their parameter.

See CCAN's typesafe_cb module at http://ccan.ozlabs.org/info/typesafe_cb.html

2.14 TDB_CLEAR_IF_FIRST Must Be Specified On All Opens, tdb_reopen_all Problematic

The TDB_CLEAR_IF_FIRST flag to tdb_open indicates that the TDB file should be cleared if the caller discovers it is the only process with the TDB open. However, if any caller does not specify TDB_CLEAR_IF_FIRST it will not be detected, so will have the TDB erased underneath them (usually resulting in a crash).

There is a similar issue on fork(); if the parent exits (or otherwise closes the tdb) before the child calls tdb_reopen_all() to establish the lock used to indicate the TDB is opened by someone, a TDB_CLEAR_IF_FIRST opener at that moment will believe it alone has opened the TDB and will erase it.

2.14.1 Proposed Solution

Remove TDB_CLEAR_IF_FIRST. Other workarounds are possible, but see 3.1.

3 Performance And Scalability Issues

3.1 TDB_CLEAR_IF_FIRST Imposes Performance Penalty

When TDB_CLEAR_IF_FIRST is specified, a 1-byte read lock is placed at offset 4 (aka. the ACTIVE_LOCK). While these locks never conflict in normal tdb usage, they do add substantial overhead for most fcntl lock implementations when the kernel scans to detect if a lock conflict exists. This is often a single linked list, making the time to acquire and release a fcntl lock O(N) where N is the number of processes with the TDB open, not the number actually doing work.

In a Samba server it is common to have huge numbers of clients sitting idle, and thus they have weaned themselves off the TDB_CLEAR_IF_FIRST flag.[3]

3.1.1 Proposed Solution

Remove the flag. It was a neat idea, but even trivial servers tend to know when they are initializing for the first time and can simply unlink the old tdb at that point.

3.2 TDB Files Have a 4G Limit

This seems to be becoming an issue (so much for "trivial"!), particularly for ldb.

3.2.1 Proposed Solution

A new, incompatible TDB format which uses 64 bit offsets internally rather than 32 bit as now. For simplicity of endian conversion (which TDB does on the fly if required), all values will be 64 bit on disk. In practice, some upper bits may be used for other purposes, but at least 56 bits will be available for file offsets.

tdb_open() will automatically detect the old version, and even create them if TDB_VERSION6 is specified to tdb_open.

32 bit processes will still be able to access TDBs larger than 4G (assuming that their off_t allows them to seek to 64 bits), they will gracefully fall back as they fail to mmap. This can happen already with large TDBs.

Old versions of tdb will fail to open the new TDB files (since 28 August 2009, commit 398d0c29290: prior to that any unrecognized file format would be erased and initialized as a fresh tdb!)

[3] There is a flag to tdb_reopen_all() which is used for this optimization: if the parent process will outlive the child, the child does not need the ACTIVE_LOCK. This is a workaround for this very performance issue.

3.3 TDB Records Have a 4G Limit

This has not been a reported problem, and the API uses size_t which can be 64 bit on 64 bit platforms. However, other limits may have made such an issue moot.

3.3.1 Proposed Solution

Record sizes will be 64 bit, with an error returned on 32 bit platforms which try to access such records (the current implementation would return TDB_ERR_OOM in a similar case). It seems unlikely that 32 bit keys will be a limitation, so the implementation may not support this (see 3.7).

3.4 Hash Size Is Determined At TDB Creation Time

TDB contains a number of hash chains in the header; the number is specified at creation time, and defaults to 131. This is such a bottleneck on large databases (as each hash chain gets quite long), that LDB uses 10,000 for this hash. In general it is impossible to know what the 'right' answer is at database creation time.

3.4.1 Proposed Solution

After comprehensive performance testing on various scalable hash variants[4], it became clear that it is hard to beat a straight linear hash table which doubles in size when it reaches saturation. ~~There are three details which become important:~~

1. ~~On encountering a full bucket, we use the next bucket.~~
2. ~~Extra hash bits are stored with the offset, to reduce comparisons.~~
3. ~~A marker entry is used on deleting an entry.~~

~~The doubling of the table must be done under a transaction; we will not reduce it on deletion, so it will be an unusual case. It will either be placed at the head (other entries will be moved out the way so we can expand). We could have a pointer in the header to the current hashtable location, but that pointer would have to be read frequently to check for hashtable moves.~~

~~The locking for this is slightly more complex than the chained case; we currently have one lock per bucket, and that means we would need to expand the lock if we overflow to the next bucket. The frequency of such collisions will effect our locking heuristics: we can always lock more buckets than we need.~~

~~One possible optimization is to only re-check the hash size on an insert or a lookup miss.~~ Unfortunately, altering the hash table introduces serious locking complications: the entire hash table needs to be locked to enlarge the hash

[4] http://rusty.ozlabs.org/?p=89 and http://rusty.ozlabs.org/?p=94 This was annoying because I was previously convinced that an expanding tree of hashes would be very close to optimal.

table, and others might be holding locks. Particularly insidious are insertions done under tdb_chainlock.

Thus an expanding layered hash will be used: an array of hash groups, with each hash group exploding into pointers to lower hash groups once it fills, turning into a hash tree. This has implications for locking: we must lock the entire group in case we need to expand it, yet we don't know how deep the tree is at that point.

Note that bits from the hash table entries should be stolen to hold more hash bits to reduce the penalty of collisions. We can use the otherwise-unused lower 3 bits. If we limit the size of the database to 64 exabytes, we can use the top 8 bits of the hash entry as well. These 11 bits would reduce false positives down to 1 in 2000 which is more than we need: we can use one of the bits to indicate that the extra hash bits are valid. This means we can choose not to re-hash all entries when we expand a hash group; simply use the next bits we need and mark them invalid.

3.5 TDB Freelist Is Highly Contended

TDB uses a single linked list for the free list. Allocation occurs as follows, using heuristics which have evolved over time:

1. Get the free list lock for this whole operation.
2. Multiply length by 1.25, so we always over-allocate by 25%.
3. Set the slack multiplier to 1.
4. Examine the current freelist entry: if it is $>$ length but $<$ the current best case, remember it as the best case.
5. Multiply the slack multiplier by 1.05.
6. If our best fit so far is less than length * slack multiplier, return it. The slack will be turned into a new free record if it's large enough.
7. Otherwise, go onto the next freelist entry.

Deleting a record occurs as follows:

1. Lock the hash chain for this whole operation.
2. Walk the chain to find the record, keeping the prev pointer offset.
3. If max_dead is non-zero:
   (a) Walk the hash chain again and count the dead records.
   (b) If it's more than max_dead, bulk free all the dead ones (similar to steps 4 and below, but the lock is only obtained once).
   (c) Simply mark this record as dead and return.

4. Get the free list lock for the remainder of this operation.

5. Examine the following block to see if it is free; if so, enlarge the current block and remove that block from the free list. This was disabled, as removal from the free list was O(entries-in-free-list).

6. Examine the preceeding block to see if it is free: for this reason, each block has a 32-bit tailer which indicates its length. If it is free, expand it to cover our new block and return.

7. Otherwise, prepend ourselves to the free list.

Disabling right-merging (step 5) causes fragmentation; the other heuristics proved insufficient to address this, so the final answer to this was that when we expand the TDB file inside a transaction commit, we repack the entire tdb.

The single list lock limits our allocation rate; due to the other issues this is not currently seen as a bottleneck.

3.5.1 Proposed Solution

The first step is to remove all the current heuristics, as they obviously interact, then examine them once the lock contention is addressed.

The free list must be split to reduce contention. Assuming perfect free merging, we can at most have 1 free list entry for each entry. This implies that the number of free lists is related to the size of the hash table, but as it is rare to walk a large number of free list entries we can use far fewer, say 1/32 of the number of hash buckets.

It seems tempting to try to reuse the hash implementation which we use for records here, but we have two ways of searching for free entries: for allocation we search by size (and possibly zone) which produces too many clashes for our hash table to handle well, and for coalescing we search by address. Thus an array of doubly-linked free lists seems preferable.

There are various benefits in using per-size free lists (see 3.6) but it's not clear this would reduce contention in the common case where all processes are allocating/freeing the same size. Thus we almost certainly need to divide in other ways: the most obvious is to divide the file into zones, and using a free list (or set of free lists) for each. This approximates address ordering.

Note that this means we need to split the free lists when we expand the file; this is probably acceptable when we double the hash table size, since that is such an expensive operation already. In the case of increasing the file size, there is an optimization we can use: if we use M in the formula above as the file size rounded up to the next power of 2, we only need reshuffle free lists when the file size crosses a power of 2 boundary, *and* reshuffling the free lists is trivial: we simply merge every consecutive pair of free lists.

The basic algorithm is as follows. Freeing is simple:

1. Identify the correct zone.

2. Lock the corresponding list.

3. Re-check the zone (we didn't have a lock, sizes could have changed): relock if necessary.

4. Place the freed entry in the list for that zone.

Allocation is a little more complicated, as we perform delayed coalescing at this point:

1. Pick a zone either the zone we last freed into, or based on a "random" number.

2. Lock the corresponding list.

3. Re-check the zone: relock if necessary.

4. If the top entry is -large enough, remove it from the list and return it.

5. Otherwise, coalesce entries in the list.If there was no entry large enough, unlock the list and try the next zone.

6. If no zone satisfies, expand the file.

This optimizes rapid insert/delete of free list entries by not coalescing them all the time.. First-fit address ordering ordering seems to be fairly good for keeping fragmentation low (see 3.6). Note that address ordering does not need a tailer to coalesce, though if we needed one we could have one cheaply: see 3.7.

I anticipate that the number of entries in each free zone would be small, but it might be worth using one free entry to hold pointers to the others for cache efficiency.

If we want to avoid locking complexity (enlarging the free lists when we enlarge the file) we could place the array of free lists at the beginning of each zone. This means existing array lists never move, but means that a record cannot be larger than a zone. That in turn implies that zones should be variable sized (say, power of 2), which makes the question "what zone is this record in?" much harder (and "pick a random zone", but that's less common). It could be done with as few as 4 bits from the record header.[5]

3.6 TDB Becomes Fragmented

Much of this is a result of allocation strategy[6] and deliberate hobbling of coalescing; internal fragmentation (aka overallocation) is deliberately set at 25%, and external fragmentation is only cured by the decision to repack the entire db when a transaction commit needs to enlarge the file.

[5] Using 2^{16+N*3} means 0 gives a minimal 65536-byte zone, 15 gives the maximal 2^{61} byte zone. Zones range in factor of 8 steps.

[6] The Memory Fragmentation Problem: Solved? Johnstone & Wilson 1995 ftp://ftp.cs.utexas.edu/pub/garbage/malloc/ismm98.ps

3.6.1 Proposed Solution

The 25% overhead on allocation works in practice for ldb because indexes tend to expand by one record at a time. This internal fragmentation can be resolved by having an "expanded" bit in the header to note entries that have previously expanded, and allocating more space for them.

There are is a spectrum of possible solutions for external fragmentation: one is to use a fragmentation-avoiding allocation strategy such as best-fit address-order allocator. The other end of the spectrum would be to use a bump allocator (very fast and simple) and simply repack the file when we reach the end.

There are three problems with efficient fragmentation-avoiding allocators: they are non-trivial, they tend to use a single free list for each size, and there's no evidence that tdb allocation patterns will match those recorded for general allocators (though it seems likely).

Thus we don't spend too much effort on external fragmentation; we will be no worse than the current code if we need to repack on occasion. More effort is spent on reducing freelist contention, and reducing overhead.

3.7 Records Incur A 28-Byte Overhead

Each TDB record has a header as follows:

```
struct tdb_record {
        tdb_off_t next; /* offset of the next record in the list */
        tdb_len_t rec_len; /* total byte length of record */
        tdb_len_t key_len; /* byte length of key */
        tdb_len_t data_len; /* byte length of data */
        uint32_t full_hash; /* the full 32 bit hash of the key */
        uint32_t magic;   /* try to catch errors */
        /* the following union is implied:
                union {
                        char record[rec_len];
                        struct {
                                char key[key_len];
                                char data[data_len];
                        }
                        uint32_t totalsize; (tailer)
                }
        */
};
```

Naively, this would double to a 56-byte overhead on a 64 bit implementation.

3.7.1 Proposed Solution

We can use various techniques to reduce this for an allocated block:

1. The 'next' pointer is not required, as we are using a flat hash table.

2. 'rec_len' can instead be expressed as an addition to key_len and data_len (it accounts for wasted or overallocated length in the record). Since the record length is always a multiple of 8, we can conveniently fit it in 32 bits (representing up to 35 bits).

3. 'key_len' and 'data_len' can be reduced. I'm unwilling to restrict 'data_len' to 32 bits, but instead we can combine the two into one 64-bit field and using a 5 bit value which indicates at what bit to divide the two. Keys are unlikely to scale as fast as data, so I'm assuming a maximum key size of 32 bits.

4. 'full_hash' is used to avoid a memcmp on the "miss" case, but this is diminishing returns after a handful of bits (at 10 bits, it reduces 99.9% of false memcmp). As an aside, as the lower bits are already incorporated in the hash table resolution, the upper bits should be used here. Note that it's not clear that these bits will be a win, given the extra bits in the hash table itself (see 3.4.1).

5. 'magic' does not need to be enlarged: it currently reflects one of 5 values (used, free, dead, recovery, and unused_recovery). It is useful for quick sanity checking however, and should not be eliminated.

6. 'tailer' is only used to coalesce free blocks (so a block to the right can find the header to check if this block is free). This can be replaced by a single 'free' bit in the header of the following block (and the tailer only exists in free blocks).[7] The current proposed coalescing algorithm doesn't need this, however.

This produces a 16 byte used header like this:

```
struct tdb_used_record {
        uint32_t magic : 16,
                 prev_is_free: 1,
                 key_data_divide: 5,
                 top_hash: 10;
        uint32_t extra_octets;
        uint64_t key_and_data_len;
};
```

And a free record like this:

```
struct tdb_free_record {
        uint32_t free_magic;
        uint64_t total_length;
        uint64_t prev, next;
        ...
```

[7] This technique from Thomas Standish. Data Structure Techniques. Addison-Wesley, Reading, Massachusetts, 1980.

```
        uint64_t tailer;
};
```

We might want to take some bits from the used record's top_hash (and the free record which has 32 bits of padding to spare anyway) if we use variable sized zones. See 3.5.1.

3.8 Transaction Commit Requires 4 fdatasync

The current transaction algorithm is:

1. write_recovery_data();
2. sync();
3. write_recovery_header();
4. sync();
5. overwrite_with_new_data();
6. sync();
7. remove_recovery_header();
8. sync();

On current ext3, each sync flushes all data to disk, so the next 3 syncs are relatively expensive. But this could become a performance bottleneck on other filesystems such as ext4.

3.8.1 Proposed Solution

Neil Brown points out that this is overzealous, and only one sync is needed:

1. Bundle the recovery data, a transaction counter and a strong checksum of the new data.
2. Strong checksum that whole bundle.
3. Store the bundle in the database.
4. Overwrite the oldest of the two recovery pointers in the header (identified using the transaction counter) with the offset of this bundle.
5. sync.
6. Write the new data to the file.

Checking for recovery means identifying the latest bundle with a valid checksum and using the new data checksum to ensure that it has been applied. This is more expensive than the current check, but need only be done at open. For running databases, a separate header field can be used to indicate a transaction in progress; we need only check for recovery if this is set.

3.9 TDB Does Not Have Snapshot Support

3.9.1 Proposed Solution

None. At some point you say "use a real database".

But as a thought experiment, if we implemented transactions to only overwrite free entries (this is tricky: there must not be a header in each entry which indicates whether it is free, but use of presence in metadata elsewhere), and a pointer to the hash table, we could create an entirely new commit without destroying existing data. Then it would be easy to implement snapshots in a similar way.

This would not allow arbitrary changes to the database, such as tdb_repack does, and would require more space (since we have to preserve the current and future entries at once). If we used hash trees rather than one big hash table, we might only have to rewrite some sections of the hash, too.

We could then implement snapshots using a similar method, using multiple different hash tables/free tables.

3.10 Transactions Cannot Operate in Parallel

This would be useless for ldb, as it hits the index records with just about every update. It would add significant complexity in resolving clashes, and cause the all transaction callers to write their code to loop in the case where the transactions spuriously failed.

3.10.1 Proposed Solution

We could solve a small part of the problem by providing read-only transactions. These would allow one write transaction to begin, but it could not commit until all r/o transactions are done. This would require a new RO_TRANSACTION_LOCK, which would be upgraded on commit.

3.11 Default Hash Function Is Suboptimal

The Knuth-inspired multiplicative hash used by tdb is fairly slow (especially if we expand it to 64 bits), and works best when the hash bucket size is a prime number (which also means a slow modulus). In addition, it is highly predictable which could potentially lead to a Denial of Service attack in some TDB uses.

3.11.1 Proposed Solution

The Jenkins lookup3 hash[8] is a fast and superbly-mixing hash. It's used by the Linux kernel and almost everything else. This has the particular properties that it takes an initial seed, and produces two 32 bit hash numbers, which we can combine into a 64-bit hash.

[8] http://burtleburtle.net/bob/c/lookup3.c

The seed should be created at tdb-creation time from some random source, and placed in the header. This is far from foolproof, but adds a little bit of protection against hash bombing.

3.12 Reliable Traversal Adds Complexity

We lock a record during traversal iteration, and try to grab that lock in the delete code. If that grab on delete fails, we simply mark it deleted and continue onwards; traversal checks for this condition and does the delete when it moves off the record.

If traversal terminates, the dead record may be left indefinitely.

3.12.1 Proposed Solution

Remove reliability guarantees; see 2.2.1.

3.13 Fcntl Locking Adds Overhead

Placing a fcntl lock means a system call, as does removing one. This is actually one reason why transactions can be faster (everything is locked once at transaction start). In the uncontended case, this overhead can theoretically be eliminated.

3.13.1 Proposed Solution

None.

We tried this before with spinlock support, in the early days of TDB, and it didn't make much difference except in manufactured benchmarks.

We could use spinlocks (with futex kernel support under Linux), but it means that we lose automatic cleanup when a process dies with a lock. There is a method of auto-cleanup under Linux, but it's not supported by other operating systems. We could reintroduce a clear-if-first-style lock and sweep for dead futexes on open, but that wouldn't help the normal case of one concurrent opener dying. Increasingly elaborate repair schemes could be considered, but they require an ABI change (everyone must use them) anyway, so there's no need to do this at the same time as everything else.

3.14 Some Transactions Don't Require Durability

Volker points out that gencache uses a CLEAR_IF_FIRST tdb for normal (fast) usage, and occasionally empties the results into a transactional TDB. This kind of usage prioritizes performance over durability: as long as we are consistent, data can be lost.

This would be more neatly implemented inside tdb: a "soft" transaction commit (ie. syncless) which meant that data may be reverted on a crash.

3.14.1 Proposed Solution

None.

Unfortunately any transaction scheme which overwrites old data requires a sync before that overwrite to avoid the possibility of corruption.

It seems possible to use a scheme similar to that described in 3.9,where transactions are committed without overwriting existing data, and an array of top-level pointers were available in the header. If the transaction is "soft" then we would not need a sync at all: existing processes would pick up the new hash table and free list and work with that.

At some later point, a sync would allow recovery of the old data into the free lists (perhaps when the array of top-level pointers filled). On crash, tdb_open() would examine the array of top levels, and apply the transactions until it encountered an invalid checksum.