UUID Version 6

Goal: Create a UUID version that works well as a database key.

Why is this important?

- Many systems already use UUIDs as database keys and just deal with the non-optimum issues that arise. MySQL, PostgreSQL, Cassandra, MongoDB (and others) each have at least some support for UUIDs as keys.
- UUIDs come close to working well as a database key, only a few changes are needed.
- It’s easier and more productive to adjust UUIDs for this purpose than to try to figure out a completely new standard.
- Databases are increasingly distributed. Simple solutions like “auto-increment” are becoming less and less workable. IDs need to be unique across multiple physical machines.

What Doesn’t Work About Existing UUID Versions

- For UUID versions 2-5, poor database index locality. Random values inserted into a B-tree may require reading and modifying many database pages. Having new values inserted at the end of the index can provide significant performance improvement.
- Version 1 UUIDs are time-ordered but still have a sorting inconvenience in that they require special logic to determine the sort order (not a “deal breaker”, but worth fixing in a new version).

What Doesn’t Work About Existing UUID Versions

- Version 1 UUIDs, per the existing standard, contain a MAC address in the “node” field (last 48 bits). This can be a security issue as it leaks a network address and potentially information about the machine itself. (Furthermore it is arguable if this is a good source of “uniqueness”, more on this in a moment.)
- The text format (hex, written as AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA) is unnecessarily long. While we should still support it for backward compatibility, base32 or base64 are more compact and just as useful for most applications. (Think about cases like URLs or human entry.)

An Observation About Uniqueness

- The current specification goes to great lengths to attempt a guarantee of uniqueness. Fields like *node* and *clock sequence* are specified in detail in RFC 4122.
- However, seeing as there is no known way to actually guarantee uniqueness without some prior knowledge between systems, UUIDs are only as unique as their inputs.
- Any predetermined value being relied on for uniqueness (which is the purpose of using the MAC address in Version 1), along with the collision probability for any other random or hashed value, are what determine uniqueness.
- Uniqueness requirements per-application are also different. In a distributed database, an already-known database node ID could be used to guarantee values are unique across a database cluster.
- So if UUIDs are only as unique as their inputs, and different applications have different uniqueness requirements, it would probably be beneficial to have a less complicated specification and allow variations that are suitable for specific applications. I.e. if an implementation wants to use a database node instead of a Mac address, I believe this should be valid for that application.

An Observation About Creating vs Storing and Reading UUIDs

- The various fields in a UUID (timestamp, version, clock sequence, node, and some of the individual bits therein) have different significance when creating a UUID as compared to reading and storing one.
- The version field is useful to determine the format of the rest of UUID.
- The timestamp is useful to provide date-ordering and can be used as a creation time and for debugging purposes, etc.
- The rest of the UUID is essentially devoted to uniqueness during creation, but then when reading has no semantic meaning other than an opaque chunk of bytes. (i.e. nobody is supposed to be reading the MAC address or the clock sequence from a UUID, it just needs to be unique.)
- Specifying in great detail how the remainder of a UUID is laid out, in the opinion of the author of this slide, is unnecessary and only tends to obscure the factors that go into what makes a UUID unique. I think the existing detail from the spec is fine, but we should essentially be saying “but if you want to put something else in this field that you are going to guarantee is unique for a specific purpose/application, that’s fine, just do that and document it – that’s not an incorrect implementation”

UUID Version 6 Aims to...

- Maintain as much compatibility as possible (e.g. variable lengths are out as they are too different from existing implementations).
- Time-ordered for good index locality.
- Allow applications to provide their own uniqueness guarantees where applicable.
- Sortable as a raw block of bytes.
- More compact text format(s). (Also sortable as binary bytes even in text format. Why complicate sorting if we can avoid it with almost zero additional effort.)

UUID v6 Format in a Nutshell

- **We want to maintain binary compatibility** (128 bit values). Lots of existing code treats UUIDs as opaque values and we don't want to break this.
- **Version field has to stay where it is for compatibility** (most significant 4 bits of the 7th byte). Would contain the value 6.
- **We use the same time format as version 1** (100-nanosecond intervals since midnight 15 October 1582), but we store them in big endian, skipping the 4 bits for the version number. This makes it so v6 UUIDs sort correctly when treated as an opaque bunch of bytes.
- **We relax the requirements on the latter 64-bits (node and clock sequence fields).** The details from RFC 4122 become an implementation suggestion, but we also explicitly state that if a database wants to use some other node identifier to guarantee uniqueness within the context of it's own system, that is okay and they should just document the uniqueness guarantees, e.g. values are guaranteed to be unique within one database cluster but not across separate clusters. (Again, the author of the application and UUID implementation will understand the implications of these sorts of decisions case-by-case much better than we can do in a general specification. The definition of "unique" is application-specific.)
- **We introduce two new text formats:** "URL-safe base64" (but with a sequence change for sorting) and "Crockford's base32". Base64 is compact and mind as well use the url-safe version, but we need proper sorting. Base32 is case insensitive and the Crockford variation has some improvements to help avoid generation of curse words and also sorts correctly as binary. These text formats are useful in modern applications such as in identifiers in URLs, or by being shorter for easier human entry, etc.