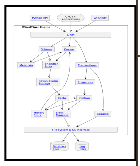
Configuration String Compilation

	Data Structures	Source Location

	`WT_CONF`	`src/include/conf.h`
		`src/conf/conf_bind.c`
		`src/conf/conf_compile.c`
		`src/conf/conf_get.c`

Caution: the Architecture Guide is not updated in lockstep with the code base and is not necessarily correct or complete for any specific release.

Introduction

WiredTiger's use of configuration strings in its API provides flexibility for callers, but when they are used in high performance code paths, they can introduce bottlenecks. This is typically because, in the worst case, internal WiredTiger code repeatedly checks for keys by scanning the entire configuration string. The longer the configuration string, the greater the scanning time. Compiled configuration strings provide a binary representation of the string, allowing for quick access to keys and their values.

The old configuration framework is known as *config*, where the new framework is *conf*. (We hope that a shorter name implies faster processing). The *conf* framework builds upon and uses elements of *config*.

Goals

As the configuration compilation subsystem (conf) was introduced after the WiredTiger code and API reached stability, there is a set of performance improvement goals and two compatibility goals.

On the performance side, the new conf processing should greatly reduce the impact of these actions, which take the bulk of the time in traditional configuration processing:

- checking the passed in configuration string for errors. That is, checking for unknown keys, value type errors (number expected but not provided), numbers out of specified range, values not one of a set of choices. These various constraints for each configuration key are listed in `api_data.py` .
- checking a configuration string for the existence of a key and retrieving its value. If the key is not in the configuration string, one or more backing configuration strings can be provided, these list all the possible keys with default values. Obviously, if a value is not provided, searching through a long default string is slow.
- checking the values of a configuration string against a set of choices. The check is currently done as a sequence of string comparisons. While this in itself is not a huge burden, it proportionally becomes a greater percentage of processing time as the rest of configuration processing performs better.

The first compatibility goal is on the calling side, to limit the API explosion that has otherwise happens when we provide a "fast" version of an API. One casualty of adding alternate APIs is that they are lightly tested in C, and often never tested in our Python tests. Our approach is to introduce a new API that can take a configuration string and returns another "compiled" string, that is actually a reference to the binary representation of the configuration string. That string can be used in any API that supports compiled configuration. This means that as APIs are adapted over time to allow a compiled configuration argument, no new changes or structures are needed for the

API. Callers can continue to use regular configuration strings, or can pre-compile any configuration string that would be used in a performance path.

A second compatibility goal is with respect to the internal implementation of the API within WiredTiger. These API(s) need to be changed to allow the compiled strings to be used, as well as allow non-compiled strings to be used. These changes should be easy to do and not prone to error.

The WT_CONF structure

The primary structure used by *conf* is `WT_CONF`. This provides information about both a single configuration string or a set of configuration strings. That is, in WiredTiger code, it replaces variables or arguments such is `const char *config` as well as arguments like `const char *cfg[]`.

To allow for fast checking of keys, we give an integer value to each key used in the entire set of configurations. This is done automatically by the `api_config.py` script in the `dist` directory. When WT code asks to get a configuration value for a key, it is now able to use an integer key rather than a string. Previously, code used "read_timestamp" to look up a key, now it uses the integer `WT_CONF_ID_checkpoint_read_timestamp`. Some of this is hidden from view via macros, so that code that previously did:

```
WT_ERR(__wt_config_gets_def(session, cfg, "read_timestamp", 0, &cval));
```

can now do:

```
WT_ERR(__wt_conf_gets_def(session, conf, read_timestamp, 0, &cval));
```

Note that both of these code snippets return `cval`, which is a **`WT_CONFIG_ITEM`**. So any code following this that uses the `cval` is unchanged. This has allowed us to make mostly mechanical changes on the WiredTiger side, reducing the possibility of errors.

Having one of these `WT_CONF_ID_*` (or "conf id") integer keys allows the conf functions to access needed information quickly. The `WT_CONF` struct (which corresponds to `conf` in the second code snippet above) contains a table that is indexed by this WT_CONF_ID_*. That is here:

```
struct __wt_conf {
    uint8_t key_map[WT_CONF_ID_COUNT]; /* For each key, a 1-based index into conf_key */
    ...
};
```

The value of `WT_CONF->key_map`[conf_id] for a given `conf_id` gives us another index, which is called `conf_key_index` in the code. This value is either `0`, meaning the key is not in this `WT_CONF`, or it is an offset into an array of `WT_CONF_KEY` entries. Typically, a WT_CONF_KEY has an embedded **WT_CONFIG_ITEM**, which is what we want. Thus, returning a value from a `WT_CONF` usually means two indirect references, and then we have a populated **`WT_CONFIG_ITEM`** to return.

If we were looking for greater performance, we could have modified this to use one table access by skipping having a `key_map`. Since there are currently over 300 possible keys in the system, this would have meant having a larger chunk of memory for each compiled string, 12K for starters (300 * ~40 bytes for each WT_CONF_KEY). But supporting composite keys means that most compiled strings would need some multiple of that value. For now, we use a `key_map`, this could be reassessed.

The WT_CONF_KEY structure

A `WT_CONF_KEY` is a simple struct that has a union along with a discriminant tag. The tag indicates how the union should be interpreted. The union is one of:

- a **`WT_CONFIG_ITEM`**. This is the typical case, it contains a value ready to return
- a sub-configuration index. This will include which `WT_CONF` in the `WT_CONF` array to use for subordinate keys.
- a `WT_CONF_BIND_DESC`. This struct contains an offset into WT_SESSION::conf_bindings.values table, that's where the bound value will be found. It also contains info to perform runtime checks on the bound value when we get it.

Position Independent Structures

The `WT_CONF` struct is meant to be position independent. A position independent struct can be copied or moved to another memory location without "fixing up" any internal pointers, or allocating new structs. The only pointers in `WT_CONF` are to items that are immutable and will never change locations, like strings. For example, when we do a compilation, we make a copy of the source string. That copy will always exist for as long as we have the connection open. The **`WT_CONFIG_ITEM`** entries that are are referenced by `WT_CONF` have pointers to parts of that immutable string. But the references in `WT_CONF` to **`WT_CONFIG_ITEM`** entries, though, are not pointers. The reference is achieved by using an offset to an adjacent piece of memory.

What if the the `WT_CONF_KEY` array was not adjacent to the WT_CONF? They could be allocated separately, but then the `WT_CONF` would need a pointer to where the array was allocated. Then, copying the `WT_CONF` would probably need a new `WT_CONF_KEY` array to be allocated and copied, and the pointer adjusted in the copy. And copies are needed to make changes to a base configuration. When we compile a configuration string, we want to take an (already compiled) WT_CONF for all the default values for that configuration and copy it, and then overwrite it with the values in the configuration string. To do that, we need a fast copy without extra allocations and fix-ups.

Why do we need compilation to be fast anyway? After all, an efficient program will do pre-compiling in advance. The reason is to also efficiently handle any uses of the API that don't use pre-compiled strings. For those uses, we go ahead and pre-compile the given configuration string. This has several advantages:

- after pre-compiling, we will always have a single way to get configuration values. Imagine an API that supports pre-compiled strings. Suppose, if we are called with a (non-pre-compiled) configuration string, we leave it as is. Then we would need two mechanisms to access any key in the input configuration. We could pull this off, but it's a bit clumsy.
- efficiency. By pre-compiling, we've walked through the config string exactly once, and we never need to walk through it again. This saves us a bunch of time - that is, we get speedups even if callers are not doing explicit pre-compilation.
- testing. All of our existing tests are testing the code paths to pre-compile and access the pre-compiled struct. We don't need two modes of testing, or lots of extra test cases.

The total size of the `WT_CONF` struct and its associated `WT_CONF_KEY` structs can be totally determined at compile time. This is also true for `WT_CONF` structs that have subordinate `WT_CONF` structs, which is used to support composite keys. Subordinate `WT_CONF` structs appear after the parent `WT_CONF` struct and before the `WT_CONF_KEY` array. Since they each have their own fixed size, but each size may be different, we have, within the `WT_CONF` struct, an offset of the `WT_CONF_KEY` array. The `WT_CONF_KEY_TABLE_ENTRY` macro is used to access the `WT_CONF_KEY` entries, using the offset.

Syntactic Sugar and Composite Keys

Suppose we have a fictional key `"blah"`, and we want a slick way to get its associated ID (WT_CONF_ID_blah) while using the syntax:

```
WT_ERR(__wt_conf_gets_def(session, blah, read_timestamp, 0, &cval));
```

Well, we can certainly use identifier pasting (the ## operator) via the C pre-processor to accomplish this. That's fine, but...

WiredTiger also makes heavy use of composite keys to reach down into sub-configurations (or "categories"). For example, suppose our fictional configuration string looks like `"blah=7,foo=(bar=123,hello=456)"` and we want to get the value for `foo.bar`. How do we identify these multiple parts to our access functions? And note that we have have depths of sub-configuration greater than two. Ideally, we want to take code that currently looks like this:

```
WT_ERR(__wt_config_gets_def(session, cfg, "foo.bar", 0, &cval));
```

and change it to:

```
WT_ERR(__wt_conf_gets_def(session, conf, foo.bar, 0, &cval));
```

Our solution is to create a const struct in `conf.h` that contains something like this:

```
static const struct {
    uint64_t blah;
    ...
    struct {
       uint64_t bar;
       ...
    } Foo;
    ...
} WT_CONF_ID_STRUCTURE = {
  {
    WT_CONF_ID_blah,
    ...
    {
       WT_CONF_ID_foo | (WT_CONF_ID_bar << 16),
       ...
    },
    ...
};
```

This is pretty close to our ideal, we need to use capitalization for walking through subordinate structures, like so:

```
WT_ERR(__wt_conf_gets_def(session, conf, Foo.bar, 0, &cval));
```

The capitalization was an unfortunate side effect of having an unregulated, but flexible namespace. For example, `"checkpoint"` is both a key and a category name.

As you might guess `__wt_conf_gets_def` is a macro that changes its third argument `Foo.bar` to `WT_CONF_ID_STRUCTURE.Foo.bar`. And that in turn is a `const` value, it takes no space and can be fully evaluated at compile time to be `WT_CONF_ID_foo | (WT_CONF_ID_bar << 16)`.

That allows the function that will be called (by the macro) to take a uint64_t arg, whose layers can be peeled off as we are evaluating composite queries. It also means that we have a depth limit of four for subcategories.

WT_CONF storage layout

A `WT_CONF` and all its sub-configurations fit contiguously in memory. The struct `WT_CONF` for the configuration in question appears first, followed by the `WT_CONF` for each sub-configurations, and additional `WT_CONF` structs for any other subordinate configuration. These effective form a `WT_CONF` array. Following this array, all the WT_CONF_KEY structs that are used appear in a single array. To avoid extra calculations during processing, each `WT_CONF` contains an offset to the slice of the `WT_CONF_KEY` array that it uses.

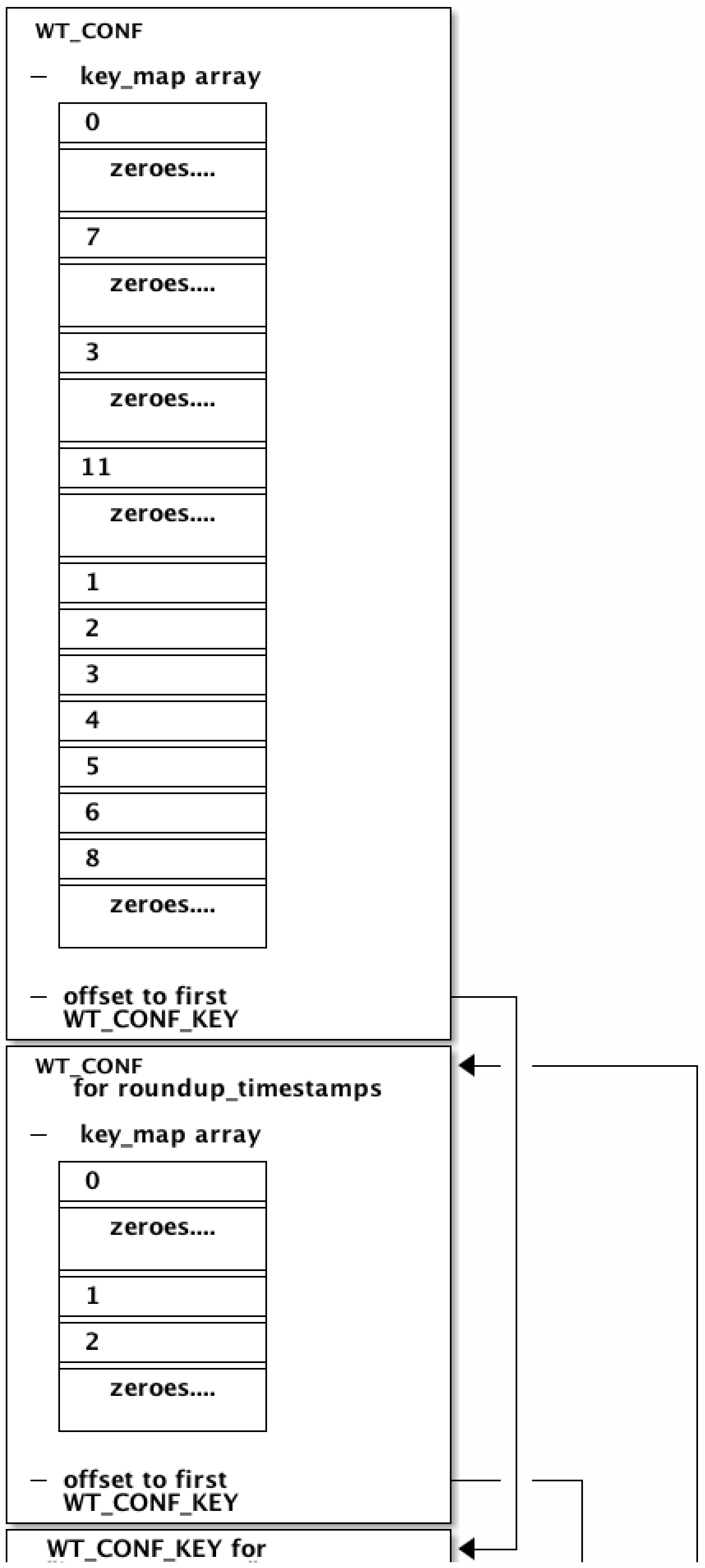
As an example, consider compiling the default configuration string for **`WT_SESSION::begin_transaction`**. Default configuration strings for every API (compiled or not) are listed in `config_def.c`. The string for `begin_transaction` lists all the legal keys and their default value:

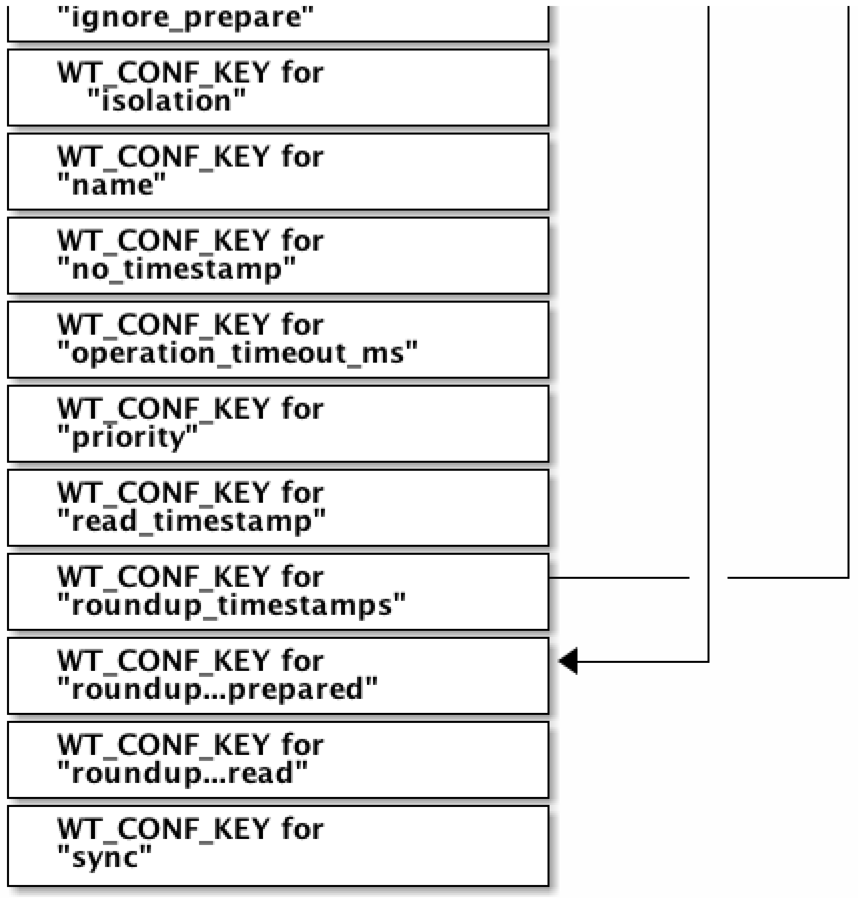
```
"ignore_prepare=false,isolation=,name=,no_timestamp=false,"
"operation_timeout_ms=0,priority=0,read_timestamp=,"
"roundup_timestamps=(prepared=false,read=false),sync=",
```

Notice that this has a sub-configuration:

```
"roundup_timestamps=(prepared=false,read=false)
```

that will have its own `WT_CONF` struct. In total, we have two WT_CONF structs. There are a total of eight top level keys, and two keys associated with `roundup_timestamps`. So we'll have an array of ten WT_CONF_KEY structs. This looks like:


WT_CONF
– key_map array
0
zeroes....
7
zeroes....
3
zeroes....
11
zeroes....
1
2
3
4
5
6
8
zeroes....
– offset to first WT_CONF_KEY
WT_CONF for roundup_timestamps
– key_map array
0
zeroes....
1
2
zeroes....
– offset to first WT_CONF_KEY
WT_CONF_KEY for


"ignore_prepare"
WT_CONF_KEY for "isolation"
WT_CONF_KEY for "name"
WT_CONF_KEY for "no_timestamp"
WT_CONF_KEY for "operation_timeout_ms"
WT_CONF_KEY for "priority"
WT_CONF_KEY for "read_timestamp"
WT_CONF_KEY for "roundup_timestamps"
WT_CONF_KEY for "roundup...prepared"
WT_CONF_KEY for "roundup...read"
WT_CONF_KEY for "sync"

You can see that `key_map` is a sparse array; only a small number of its 300 some entries are non-zero. Both key_map arrays are actually the same size, the second one is much more sparse, so the zeroed gaps are larger than they appear in this picture. Each entry in the map is a byte, which currently limits the number of WT_CONF_KEY array entries to 256.

Let's look at how we would look up `read_timestamp` in this `WT_CONF`. We'll always start at the top `WT_CONF`. `WT_CONF_ID_read_timestamp` is defined as 4, which means we'll look at `key_map`[4]. That is the entry that has 7. We then use 7 along with the offset to the associated `WT_CONF_KEY` array using a 1-based lookup. And that leads us to the `WT_CONF_KEY` for `read_timestamp` . A `WT_CONF_KEY` is a union, and its tag indicates that the **`WT_CONFIG_ITEM`** part of the union is used. That **`WT_CONFIG_ITEM`** is filled in according to the value indicated in the default string, that is, "`read_timestamp=`". That **`WT_CONFIG_ITEM`** is what is returned by the look up.

For a composite key, let's examine looking up `roundup_timestamps.read`. The definition of `WT_CONF_ID_roundup_timestamps` is 157, the top level `key_map`[157] has 8, that takes us to the eighth `WT_CONF_KEY`. Its union tag indicates it has sub-configuration at offset 1. That is, the second `WT_CONF` in the conf array. `WT_CONF_ID_read` is 159, so now we use the `key_map` in the second `WT_CONF`, that takes us to entry 2. We use the offset from the second `WT_CONF` to finally find the matching `WT_CONF_KEY` for `roundup_timestamps.read` . This is tagged as a **`WT_CONFIG_ITEM`**, so that's what we'll return.

When we compile any configuration string for a given API, we always know the total size of the `WT_CONF` and all of its parts, even without looking at the configuration string. That is because a configuration string constructed by a calling program is always used in conjunction with the set of defaults. So any `WT_CONF` for compiled string includes the values given by the input string, as well as the values given by the defaults. Using this approach means checking and getting configuration values involves looking at only one data structure. The cost is that any compiled

WT_CONF struct maybe large. However, we expect that most programs will only have a small number of pre-compiled strings.

Compiling Unbound Parameters

Up to now, we've been talking about fixed configuration strings. A more typical case is a configuration string where one or more parts of it may vary from call to call. For example, with begin_transaction, we might want to give each transaction a read_timestamp, and we only know that when we are ready to do the call. The compilation system allows unbound parameters that can later be bound. Currently, there are two kinds of bindings:

- strings, represented by "%s"
- integers and boolean, represented by "%d"

These special values indicate a deferred binding. For example, if we compile the string:

```
"ignore_prepare=%d,name=%s,read_timestamp=%d"
```

we'll need a way to mark these unbound parameters in the WT_CONF struct (discussed below). Before using the compiled string, the API user will call **WT_SESSION::bind_configuration** with the actual values. Using the above compiled string, the caller might say:

```
session->bind_configuration(session, compiled_config_string,
   1,                    /* ignore_prepare, boolean 'true' */
   "txn_name123",        /* name */
   0x1234);              /* read_timestamp */
```

The binding values are first checked for validity if needed, just as a configuration string checks its values. The values are then put into the session in an array of **WT_CONFIG_ITEM**. The ordering of values given in the bind_configuration must match the order of the '' entries given in the configuration string.

We said we need a way to mark the unbound parameters in the WT_CONF during compilation. That is done in the associated WT_CONF_KEY struct. It has a tag indicating a binding, and thus contains a WT_CONF_BIND_DESC struct. This struct has the offset into the session's array of bound values. Also, an ordered list of pointers to these WT_CONF_BIND_DESC structs makes it easy to find the right binding during the bind_configuration call.

This is one of the few times we allow pointers in WT_CONF. It is safe because unbound parameters are not used internally by WiredTiger. That is, a WT_CONF built for a default configuration has no unbound parameters, and thus has no pointers. So it can be freely copied when we create a WT_CONF as a basis for a user's request to compile a configuration string.

Binding Parameters

The **WT_SESSION::bind_configuration** call will be used to fill entries in an array encapsulated in the session with the WT_CONF_BINDINGS struct. Each element of this array has two entries, a pointer to the WT_CONF_BIND_DESC descriptor used and the **WT_CONFIG_ITEM**.

The **WT_SESSION::bind_configuration** does the following steps.

- Use the first argument to find the WT_CONF struct we are binding to. See **The Compiled String**.
- Within WT_CONF, get the list of unbound parameters WT_CONF->binding_descriptions.
- Walk through the variable arguments, matching each to a binding_description.

- For each binding_description, get a variable argument of the matching type and: – fill the next **WT_CONFIG_ITEM** in the session array, along with `WT_CONF_BIND_DESC` pointer from the WT_CONF->binding_descriptions array. – advance the position pointer in the session to the next array entry for parameter bindings.

At runtime, if we look up the key in question, in the WT_CONF_KEY union, we would find a `WT_CONF_BIND_DESC`, and know what offset in the session array to find the needed **`WT_CONFIG_ITEM`**. For sanity, we can check that the `WT_CONF_BIND_DESC` pointer in the array matches the union entry we are on.

Scripts Supporting Configuration Compilation

In `dist`, API methods that support compilation are marked in `api_data.py`. `api_config.py` contains functions that use the data in `api_data.py` to create or modify files in `src/include`, including `config.h` and `conf.h`. Knowledge of the configuration strings is used to set up initialized data structures that are used for compilation, argument checking, and default values.

Initial Pre-Compilation

For any API method that supports compilation, we do pre-compilation when a connection is opened. That is, we compile the configuration string that has all the default values listed, and store the result. The gives us a ready made "default" `WT_CONF`, that is stored in the connection. This is used for a common case when the API is called with no configuration string. It also is a basis for when we compile a configuration string.

Compiling a configuration string starts by knowing the size of the WT_CONF struct. We know the size of the default `WT_CONF`, derived from how many sub-configurations it has, and how many total keys it has. Our compiled configuration is the same size, so we allocate storage for it, and do a memory copy of the default `WT_CONF`. Then we can fill any parts that change.

The Compiled String

A call to WT_SESSION::compile_configuration compiles a configuration string and returns a `const char *` that can be used by the caller in place of a compilation string. When WiredTiger compiles the string a we've described, we end up with a `WT_CONF` struct, and we'd like to return that to the caller. For debugging purposes, we'd like what we return to look like a string.

Our current implementation returns a special string that is an offset into a dummy array associated with the **WT_CONNECTION**. That is, we create an string of size 1024 and put it into `WT_CONNECTION->conf_dummy`. When the application calls WT_SESSION::compile_configuration to compile a string the Nth time, we put a pointer to the compiled `WT_CONF` into entry *N* of the `WT_CONNECTION->conf_array` struct, and return `&WT_CONNECTION->conf_dummy[N]`. When the application uses that string later, we can easily recognize that it is an offset into the `conf_dummy` array, and find the needed `WT_CONF`.

No Scoped Identifiers

Above, we state that we have a separate identifier for every key used in the configuration system, that's more than 300 keys. Data structures could be smaller, and possibly made faster, if we had a smaller number of keys in play. Since each API might have only a small fraction of keys, perhaps a dozen or so, might it make sense to scope the identifiers. For example, instead of having:

```
#define WT_CONF_ID_sync 118
```

could we not have something like:

```
#define WT_CONF_ID_begin_transaction_sync 9
...
#define WT_CONF_ID_checkpoint_sync 13
...
#define WT_CONF_ID_flush_tier_sync 4
```

These scoped numbers would only need to be unique within their own API, so they could be a lot smaller.

This approach could work in some cases, but there are places in WiredTiger where certain sets of configuration keys are known to multiple APIs. There are then shared utility functions that manage the configuration parsing of these sets of keys. These utility functions currently take configuration string arguments, and would be converted over to using `WT_CONF` arguments. But such utility functions aren't necessarily scoped, for example, they may be used by multiple APIs. This is a case where we *would* want values between multiple APIs to be the same. There are workarounds for this, but it rapidly becomes messy. For simplicity, we went with identifiers that have no scope.