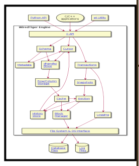
Tiered Storage

	Data Structures	Source Location
	`WT_TIERED` `WT_TIERED_TIERS`	`src/include/tiered.h` `src/tiered/`

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

Warning
Tiered storage is considered an experimental part of the WiredTiger library, and is not yet ready for production workloads.

Introduction and Definitions

Tiered storage allows B-Trees to be stored into multiple places, more recently updated blocks on local disk and less recently updated blocks in cloud storage. The term *object* refers to one of these written btree parts, whether the object resides on the local disk, as an *object file*, or in the cloud, as a *cloud object*. The mechanism to create new objects is a **WT_SESSION::checkpoint** API call with the `flush_tier` configuration set. For brevity, it is called a `flush_tier` call.

When in use, a tiered btree, like a non-tiered btree, may have some recently used or modified data that resides in memory pages. This in-memory representation is the same between a tiered btree and a non-tiered btree, it is only how data is stored on disk and in cloud objects that makes these btrees different.

Tiered storage is configured and enabled on the connection via **wiredtiger_open** configuration options (e.g. `tiered_storage=(name="s3_store",bucket="...")`). Once enabled on the connection, the configuration applies to all tables created. The configuration can be overridden on individual tables with configuration options for **WT_SESSION::create** . The related **WT_SESSION::create** configuration options allow the caller to specify a different storage provider or bucket name or to have the table opt out of tiered storage.

Object IDs

In response to a `flush_tier` call, WiredTiger creates a new object for each changed btree. Each new object created gets a new *objectid*, incremented from the previous objectid used with that btree. The new object lives initially as a regular local disk *object file*, and the disk file name includes the name of the table and the objectid. This makes it is easy to see from a directory listing which tables and which objects for that table are represented on disk. Once a new (*N+1*) object is created, all writes to the previous (*N*) object are completed and that *N* object becomes read-only, like all objects before it. At this point, the *N* object is queued to be copied to cloud storage. After that copy successfully completes, the local copy of *N* is queued for removal, see **Local File Removal**.

Checkpoints

A normal, non-tiered table, can be thought of as an active or *live* btree, with zero or more checkpoints that are fully represented in the single disk object file. Each checkpoint has its own root page, and can be considered its own btree. Each of these btrees is a set of pages referencing each other as a tree, some in memory, some on disk. A tiered table is the same, having a *live* btree and a set of checkpoint btrees. However, both the live btree and the checkpoints may have pages that span multiple object files and/or cloud objects.

For tiered storage, each object, other than the current file object, is guaranteed to contain at least one checkpoint. When there is a switch to a new (*N+1*) current object, a checkpoint in the previous (*N*) file object is guaranteed, because a **WT_SESSION::checkpoint** call with a `flush_tier` option is required to switch. A checkpoint in the *N* object refers to blocks in object *N* as well as previous (*N-1*, *N-2*, ...) objects.

Block Manager

A btree is organized logically as a set of pages. Each page is either a leaf page, containing keys and values, or an internal page, which has references to subordinate leaf pages. Writing a page goes through the block manager, which has the job of locating a free position in the file. That position, along with the block size and a checksum of the content, is packaged up as an *address cookie* (a sequence of bytes) that is returned to the caller. Address cookies are further described in **What is a block?**.

Cookies in Tiered Storage

For tiered storage, a reference to a written location needs to indicate an objectid. If the reference is from the same object, no explicit objectid is needed, and the `(position, size, checksum)` triple can be used. Otherwise, a four-tuple `(position, size, checksum, objectid)` is used. Because the byte size of the cookie is given, and the objectid appears last, the code that interprets cookies can deduce which kind it is given. If the decoder has seen three entries and it has reached the end of its data, then it has a triple. If the decoder still has more data, then it also contains an objectid.

Object numbering starts at 1, so an objectid of 0 references the same object as the cookie's location. Because of these properties, a file from a non-tiered btree can be upgraded to the first object of a tiered btree without changing its contents. Every reference in the non-tiered btree is a triple, meaning there is an implied a zero objectid. Thus each reference must be to the same object.

Files in the Block Manager

The data structure that the block manager uses for accessing a btree's files or objects is a `WT_BM`. For non-tiered tables, the `WT_BM->block` field references a single `WT_BLOCK` *handle*, which represents the single file. This block handle exists as long as the btree is open, so no locking or coordination is needed to access it.

For tiered tables, multiple objects are associated with a `WT_BM`, and each object is referenced by a `WT_BLOCK`. The current writable object is referred to by `WT_BM->block`, and a list of objects recently referenced appears in the `WT_BM->handle_array`. The `handle_array` always includes the current object. When an existing tiered table is opened from disk, a new `WT_BM` is created. A `WT_BLOCK` for the current writable object is also created, and `WT_BM->block` is set to that and also added to the (otherwise empty) `handle_array`. As other objects are referenced, new `WT_BLOCK` handles are created to represent them, and they are added to the `handle_array`.

Block handles (`WT_BLOCK`) in the handle array are cached from the complete list of open block handles kept in the connection (`WT_CONNECTION_IMPL->block_hash`). Blocks in the connection hash table are reference counted. When the reference count goes to zero, the handle is removed and freed, and the underlying file handle is closed.

When a `flush_tier` is done, each table that had changes written as part of the checkpoint will have its `WT_BM::switch_object` function called. This function creates a new `WT_BLOCK` for the new *N+1* file object, and adds it to the `handle_array`. It does not yet point the `WT_BM->block` to the new block handle, as there may still be writes in progress to the old block. When all writes from the checkpoint have completed, the block manager sets `WT_BM->block` to point to the block handle for the new file object (i.e., file object *N+1*).

Since there are multiple threads accessing the block manager, a read/write lock on the `WT_BM` is used to access, modify or grow the array of `WT_BLOCK` handles.

Local File Removal

When the tiered server determines that a local object file should be removed there are no writers to the file. This is because local object file removal can only be done on files that have become read-only. However, there may be active readers in the object file. At the time an object file is removed, the corresponding `WT_BLOCK` entry from the `handle_array` cannot be removed until all readers have finished using the block handle. The mechanism to remove a local object file has several parts.

When a cloud copy of an object is completed, the block manager is told, via `BM::switch_object_end` , that object id *N* (and therefore all ids less than *N*) can be removed. Then, when a block handle (`WT_BLOCK`) qualifies as removable and its reader reference count goes to zero, the block handle can be removed from the `handle_array`. This is triggered by calling a sweep function.

There is a potential for a race involving the read reference count: when the count becomes zero indicating the block can be removed, a new reader thread can enter the system and start using the block as it is being removed. To resolve the race, the reader thread takes a read lock on the block manager's lock while it increments the read reference count. When the sweep runs, it holds a write lock on the block manager's lock.

Tiered Server and Work Queue

Whether or not tiered storage is enabled, a *tiered server* thread is created. Its job is to process items on its work queue. Work items are actions that are generated by API calls that imply that some background action should take place. There is a single work queue, and items on it have a type, as follows:

- `FLUSH:` copy a local object file to the cloud.
- `FLUSH_FINISH:` notify any local cloud caches that an object file has been copied to the cloud, and a local copy is still available to ingest.
- `REMOVE_LOCAL:` remove a local file that is has already been copied to the cloud.
- `REMOVE_SHARED:` remove a cloud file, perhaps as part of a drop operation.

There are individual functions to add items of each type, and to get the next item, if any, of each type. That allows processing of items in a certain order. With only one server thread, all operations occur sequentially. It would make sense in the future to do `FLUSH` operations in parallel.

Metadata and Associated Data Structures

The WiredTiger metadata file persists important information about URIs known in the system. See **Example Metadata** for a discussion on the relationship between metadata entries and internal data structures for a simple non-tiered table `A`. That example is continued below for tiered tables.

For a tiered table `A` that has gone through a few flush_tiers, the following entries should appear in the metadata table:

- `"table:A"`
- `"colgroup:A"`
- `"file:A-0000000004.wtobj"`
- `"object:A-0000000003.wtobj"`
- `"object:A-0000000002.wtobj"`
- `"object:A-0000000001.wtobj"`
- `"tier:A"`
- `"tiered:A"`

The `"table:A"` entry, like before, is the top-level URI that is used for API calls. As before, it maps to a `WT_TABLE`, and the table has the `WT_TABLE->is_tiered_shared` flag set. The column group entry `"colgroup:A"` has in its metadata a reference to the URI for the btree. In the non-tiered case, this appeared as `source="file:A.wt"`. In this, the tiered case, the reference is `source="tiered:A"`, which relates to a `WT_TIERED` struct.

The `WT_TIERED` data structure has most of the useful information about a tiered table. In particular the btree is found in the dhandle. To be precise, `WT_TIERED->iface` is the dhandle, and `WT_TIERED->iface.handle` is a pointer to a WT_BTREE. The name of the current local object (`"file:A-0000000004.wtobj"` for this example) is stored in `WT_TIERED->tiers[WT_TIERED_INDEX_LOCAL]`. The URI of the btree (`"tiered:A"`) is kept in `WT_TIERED->tiers[WT_TIERED_INDEX_SHARED]`. Stored here is a pointer to a `WT_TIERED_TIERS` struct, this aligns with the `"tier:A"` entry in the metadata.

Also in the `WT_TIERED` struct is information about the current object id, and how many object ids are known. If the current writable object id is 4, the needed name can be constructed: `"file:A-0000000004.wtobj"`, likewise, previous cloud object names can be constructed, like `"object:A-0000000003.wtobj"`. Navigation like this, constructing names to look up in metadata or as data handles doesn't happen often, mostly during startup or flush_tier. During high performance paths, the `WT_TIERED` struct or the `WT_BM` (block manager struct) associated with the btree have everything needed.

A "tiered:" entry (associated with a tiered table) and a "file:" entry (associated with a non-tiered table) behave almost identically in the WiredTiger system. In fact, the `WT_BTREE_PREFIX` macro checks to see if a URI matches either one of these prefix strings. The macro basically means "does this thing walk and talk like a btree?". In both cases, the dhandle found with the given name has a `dhandle->handle` that points to the open `WT_BTREE`.

The `"object:"` entries do not have separate in-memory data structures (there is a `WT_TIERED_OBJECT` defined in `tiered.h`, but it is not used). And while `"table:"`, `"file:"` and `"tiered:"` all have data handles using those URIs, there are no separate data handles for each object. This is probably a good thing, as scaling the number of data handles system-wide has been challenging.

The following table summarizes the various relationships.

URI prefix	struct type	has dhandle?	notes
`table:`	WT_TABLE	yes	the dhandle is cast to (WT_TABLE *)
`file:`	(none)	yes, but...	... does not relate to a btree
`colgroup:`	(none)	no	stored in an array in the WT_TABLE, references the "tiered:" entry

`tiered:`	WT_TIERED, WT_BTREE	yes	the dhandle is cast to (WT_TIERED *), and dhandle->handle is a (WT_BTREE *)
`tier:`	WT_TIERED_TIERS	no	WT_TIERED->tiers[1] is a (WT_TIERED_TIERS *)
`object:`	(none)	no	no dhandle for objects

Storage Sources, Buckets, and Prefixes

The location of objects in the cloud requires some information to be stored: the cloud provider, the bucket used, and an id to reference any needed credentials in a key management system. When a connection is opened for tiered storage, these are specified and are used in the default case. However, to have the ability to store tables in different clouds or buckets, cloud location information needs to live in the `WT_TIERED` object as well. To make this happen, all this *location* information is abstracted into a `WT_BUCKET_STORAGE` . The connection has a pointer to a `WT_BUCKET_STORAGE`, and each `WT_TIERED` does as well.

Among the fields of a `WT_BUCKET_STORAGE` is a pointer to a `WT_STORAGE_SOURCE`. The storage source can be thought of as a driver, or an abstraction of a cloud provider with operations. WiredTiger has several instances of `WT_STORAGE_SOURCE`, these include the drivers for the AWS, GCP, and Azure clouds, as well as `dir_store`, used for testing and development. A storage source can be asked to create a custom file system (returning a **`WT_FILE_SYSTEM`**) from a bucket name and credentials. A file system created this way is stored in the `WT_BUCKET_STORAGE`.

The file system can be used for various read-only operations, like listing the bucket contents or opening a **`WT_FILE_HANDLE`** . That, in turn, can be used to get the contents of a cloud object. Note that **`WT_FILE_SYSTEM`** and **`WT_FILE_HANDLE`** are more generic WiredTiger concepts, and are used outside of tiered storage.

The file system obtained from a storage source cannot be used to write pieces of data to objects, rather there is a method on the `WT_STORAGE_SOURCE` (cloud driver) that is used to copy a source object file in its entirety to the cloud. This makes it clear that objects must be written in their entirety, but may be read, if desired, in pieces.

In addition to the cloud providers, the storage source called `dir_store` emulates the behavior of a cloud provider, but stores objects in a directory. This is used for testing and development, avoiding the complication and expense of using cloud storage. By default, tests use a subdirectory of the WiredTiger home directory as the `dir_store` repository. Then, when a test breaks, the "cloud" objects are readily available for debugging.

We anticipate that buckets may be shared among multiple nodes, and possibly multiple clusters. To avoid name collisions, there is a prefix that can be given on a **wiredtiger_open** call. It is expected that the caller gives WiredTiger a unique prefix. This prefix is stored as part of the `WT_BUCKET_STORAGE`, and so can also be specified per table upon creation. The prefix is prepended to every name stored in a bucket.

Flush Checkpoint Operations

A flush_tier checkpoint has a slightly different sequence of operations from a regular checkpoint without flush_tier. In particular, eviction is only disabled when we are checkpointing the tree.

The first part of the checkpoint is determining which btrees need to participate. Following that, those btrees are checkpointed to disk. Finally, when the files have been written, they can be asynchronously copied (*flushed*) to cloud storage.

Checkpoint Prepare Phase

- Assemble a list of all the dhandles that will be part of the checkpoint. For non-tiered btrees this will be btrees that have been modified since the last checkpoint. For tiered btrees this will be btrees modified since the last flush. See `checkpoint_flush_tier` called from `checkpoint_prepare`.
- For each tiered btree, create the new file, and queue a FLUSH work entry to flush the old one. The work entry includes checkpoint generation information that is used by the tiered server thread to know when it is okay to flush the file to object storage (i.e., after the current checkpoint has completed). See `tiered_switch`, which is called from `checkpoint_flush_tier`.
- This work is done now so that it doesn't add time to the actual checkpoint when eviction is disabled.

Data file checkpoint

Once the list of dhandles is known, each dhandle (this is, btree) is checkpointed:

- When doing the actual checkpoint for each btree, eviction to the btree is blocked. Hence checkpoint is the only thread updating the btree. This happens in `wt_sync_file`.
- At the end of the checkpoint, after writing the new checkpoint root but before re-enabling eviction, the active file switches to be the new file created during the prepare phase. This ensures that everything that is part of the checkpoint is in the old file, and anything evicted after the checkpoint is in the next file.
- If the btree is tiered, fsync is called on the old active file to ensure that all checkpoint updates are durable. Side note: the fsync doesn't have to be done here, it could happen after re-enabling eviction. But this describes the current actions of the code.
- See `bm_checkpoint` for the active file switch and fsync.
- After switching the file, eviction is allowed again as we finish `wt_sync_file`. Then the next dhandle is processed.

Flush actions

As indicated above, the FLUSH work entry is queued during the prepare phase and is processed asynchronously. The FLUSH entry indicates that WiredTiger should copy the completed file on disk to cloud storage.

When the FLUSH completes, these actions occur:

- tell the chunk cache to ingest the object before it is removed
- update the metadata to reference the new object in the cloud
- queue a FLUSH_FINISH operation
- queue a REMOVE_LOCAL operation

Future

There are some future features that are helpful to know about when studying the overall design.

Sharing of Tiered Objects

The current implementation of tiered storage supports btrees spanning objects that are stored locally and in cloud storage. Each cloud object is currently only useful and known to the system that created it. However, a larger design was in mind when the current implementation was made, and that informed a number of design decisions along the way.

The larger design allows all systems in a cluster to share knowledge about tiered objects. Generally, there is a single designated node in a cluster that calls flush_tier, this is the *flushing* node. Information about the objects stored as a result of the flush can be returned to the application, where it is transferred to other cooperating nodes in the cluster. These other nodes are known as "accepting" nodes, and accept this information, updating their metadata and incorporating references to the newly known objects. The mechanism for returning the flush information, and providing a way to incorporate the references, has not been implemented.

Another part of the sharing design is new kind of "union" table that is used to help incorporate new objects on the accepting nodes. The idea is that a tiered table has an additional layer. There is a local "tier", which is just a local btree, and a shared "tier" which is the tiered btree with multiple objects that has been described here. The cloud objects in the shared tier is shared among all the nodes of a replica set.

On both the flushing node and accepting nodes, any changes to a tiered table are inserted or updated into the local btree. Any lookups to the table consult the local btree first, then the shared btree. When a flush_tier is done on the flushing node, the set of changes up to a known timestamp *T* is moved from the local btree to the shared btree, a new object is created, and the previous object, which now contains the set changes, is pushed to the cloud. The accepting nodes receive the notification of the new cloud object, and the timestamp *T* associated with it. The accepting nodes trade out their own shared btree for a new shared btree rooted at the new cloud object. Any entries in an accepting node's local btree with a timestamp older than *T* are redundant as they must be included in the cloud object. There could be multiple strategies to remove the old entries.

The beauty of this design is that it is not difficult to understand (versus various alternatives), it does not require any downtime when new objects are accepted, nor any downtime if an accepting node is upgraded to a flushing node.

Garbage Collection

While there is a mechanism to create new objects, there is no removal, or *garbage collection* of objects that become redundant. That is, as new objects may completely cover sets of keys in the btree, pages having those keys in an older object are no longer needed. After all pages in an object are no longer needed, an object can be removed. The trick is in knowing when this can happen.

A future garbage collection solution could work well either synchronously or asynchronously. A synchronous approach would probably have WiredTiger track references to all pages in either a checkpoint or a object file (and persist that information as well), and notice when all references to an object have reached zero. This may require enhancements to extent lists in the block manager. An asynchronous approach could work mostly separately from WiredTiger (in another process possibly on a different node), and examine object files, visiting internal pages and tracing the references to all objects. As such, it can notice when an object file has no references. Either approach allows the identification and removal of unused objects, and maybe also identification of objects that are lightly filled. These objects could be made redundant by rewriting their useful content directly into the active btree.