


# 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 provides a way to split Btrees into multiple parts, with some set of parts stored in cloud storage objects and another set of parts stored in local files. We often use the term *object* to refer one of these written Btree parts, whether it resides on the local disk or in the cloud. The mechanism to create new objects is a **WT\_SESSION::checkpoint** API call with the `flush_tier` configuration set. For brevity, we often call this a `flush_tier` call.

When in use, a tiered Btree, like a regular 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 regular Btree, it is only how data is stored on disk and in cloud objects that makes these Btrees different.

## 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 disk file, and the disk file name includes the name of the table and the *objectid*. This makes it 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 readonly, like all objects before it. At this point, the  $N$  object is queued to be copied to cloud storage. After that copy takes place, the local copy of  $N$  is queued for removal, see **Local File Removal**.

## Checkpoints

A normal, non-tiered table, although sometimes thought of as a "single" Btree, can also be thought of as an active Btree (a set of pages referencing each other, some in memory, some on disk), as well as zero or more checkpoints, that are fully represented in the single disk file. Each checkpoint has its own root page, and so is really its own Btree. A tiered table is the same, having an active Btree and a set of checkpoint Btrees. However, both the active Btree and the checkpoints may span multiple files and/or cloud objects.

For tiered storage, each object, other than the current file, is guaranteed to contain at least one checkpoint. When we switch to a new ( $N+1$ ) current object, a checkpoint in the previous ( $N$ ) file is guaranteed, because a

**WT\_SESSION::checkpoint** call with a `flush_tier` option is required to switch. A checkpoint in the  $N$  file refers to blocks in  $N$  as well as previous ( $N-1$ ,  $N-2$ , ...) objects.

The mechanics for coordinating the separate objects is handled in the block manager and the metadata.

## 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. The "packaging" uses the WiredTiger *pack* format described in `src/include/intpack_inline.h`. The caller to the block manager uses that returned cookie as a reference to the written page, it may be stored for example, in an internal page. Later, when the application wants to access that page and it is no longer cached in memory, the block manager is given the address cookie which it must decode to locate and read the block. Thus, the block manager "owns" the cookie and how it is packed and unpacked; code above the block manager view the cookie as an opaque set of bytes.

## 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.

Object numbering starts at 1, so an objectid of 0 references the same object as the cookie's location. Because of these properties, it's straightforward to see that a file from a non-tiered Btree could 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.

In addition to address cookies, there are also checkpoint cookies. Checkpoint cookies fully describe everything needed to access a checkpoint on an individual Btree. Like an address cookie, a checkpoint cookie is a set of bytes that can be unpacked into a set of values. A checkpoint cookie is logically composed of 4 address cookies and two additional values:

- an address cookie for the root page
- an address cookie for the extent list with allocated entries
- an address cookie for the extent list with available entries
- an address cookie for the extent list with discarded entries
- the file size for the checkpoint
- the checkpoint size

So for a non-tiered table, using triples for address cookies, there are 14 values in total. And for a tiered table, using four-tuples for address cookies, there are 18 values. The program in `tools/wt_ckpt_decode.py` can be used to unpack a set of bytes and show these values.

## Files in the Block Manager

The data structure that the block manager uses for accessing a Btree's files 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 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, we start with a new `WT_BM`. A `WT_BLOCK` for the current object is 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.

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 file, 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. The block manager participates in the checkpoint for the file, and when that is complete, the new active file is put into `WT_BM->block`.

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

## Local File Removal

When the tiered server determines that a local file should be removed, we know first and foremost that there are no writers to the file. This is because local file removal can only be done on files that have become readonly. However, there may be active readers in the file. At the time we remove the file, we'd like to remove the corresponding `WT_BLOCK` entry from the `handle_array`, but we cannot do that until all readers have finished using the block handle. When we want to remove a `WT_BLOCK`, we set a flag on it for our intent to remove. We also have an atomic reference counter for readers on every `WT_BLOCK`, and when the count goes to zero and the flag is set, we can remove it.

We must deal with a race with the reference count: when the count becomes zero and 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, we get a read lock on the block manager lock before we increment the reference count, and a write lock when we decide to free it.

## Tiered Server and Work Queue

When 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 file to the cloud
- **FLUSH\_FINISH:** notify any local cloud caches that a 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, or each type. That allows us to process items in a certain order, if we wish. 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

### Non-tiered Tables

When non-tiered table A is created (without named column groups), there are two entries in the metadata Btree, having these keys:

- "table:A"
- "file:A.wt"

For brevity, we are showing just the keys; the values in the metadata Btree are configuration strings that would show information about key/value format and other creation options, checkpoints associated with the table, etc.

The "table:A" represents the table interface that the API caller uses, and the associated data structure in memory for the table is WT\_TABLE. If there were column groups or indices, there may be multiple Btrees associated with the table, and listed in the metadata, but in this example there is just one. The "file:" entry represents the Btree stored on disk, as well as parts cached in memory. The data structure implied by this entry is a WT\_BTREE.

In summary, the relationship between prefixes and data structures is as follows:

URI prefix	struct type	has dhandle?	dhandle notes
table:	WT_TABLE	yes	the dhandle is cast to (WT_TABLE *)
file:	WT_BTREE	yes	dhandle->handle is a (WT_BTREE *)

### Tiered Tables

For a tiered table A that has gone through a few flush\_tiers, we might have the following entries in the metadata table:

- "table: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. Given this flag setting, WiredTiger knows to construct a matching "tiered:A" entry, which relates to a WT\_TIERED struct.

A WT\_TIERED 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 WT\_TIERED structure is also ready to support union tables, see future work in [Sharing of Tiered Objects](#). The local tier part of union tables would in WT\_TIERED->tiers[WT\_TIERED\_INDEX\_LOCAL], this is always NULL in the current system. Note that WT\_TIERED\_INDEX\_LOCAL is 0. The second part of the union table, and the part that we use is WT\_TIERED->tiers[WT\_TIERED\_INDEX\_SHARED]. (WT\_TIERED\_INDEX\_SHARED is 0). 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 we know that the current object id is 4, we can construct the needed name: "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 we need.

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?	dhandle notes
table:	WT_TABLE	yes	the dhandle is cast to (WT_TABLE *)
file:	(none)	yes, but...	does not relate to a Btree
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, we want the ability to have some tables stored in different places, so we need cloud location information 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 a several instances of WT\_STORAGE\_SOURCE, these include the drivers for the AWS, GCP, and Azure clouds. 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 readonly 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 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, there is a storage source called `dirstore` that emulates the behavior of a cloud provider, but stores objects in a directory. This is used for testing, avoiding the complication and expense of using cloud storage. Typically we use a subdirectory of the WiredTiger home directory as our `dirstore` repository. When a test breaks, we already have the "cloud" objects available for debugging.

We anticipate that buckets may be shared among multiple nodes, and possibly multiple clusters. To avoid name collisions, we provide a prefix that can be given on a **wiredtiger\_open** call, and assume the caller gives us 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\_tier

A **WT\_SESSION::checkpoint** call with `flush_tier` enabled (also known as `flush_tier`) puts everything we discussed on this page together. It identifies any tiered btree that have changes since the previous checkpoint, and for each such modified btree:

1. eviction is temporarily disabled on this btree, while waiting for any writes to the active file to drain
2. update the metadata to know about the next object
3. a new empty file is created named with the next object number, this will become the table's active file
4. switch the table's active file
5. add a FLUSH entry to the work queue for the previously active file to be copied to the cloud
6. enable eviction on this btree

When the FLUSH work entry is completed, we:

1. tell the chunk cache to ingest the object before it is removed
2. update the metadata to reference the new object in the cloud
3. queue a FLUSH\_FINISH operation
4. 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`, we call this 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 we've talked about thus far. 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 we have 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.

We expect future solutions to work either synchronously or asynchronously. A synchronous approach would probably have WiredTiger track references to all pages in either a checkpoint or a file (and persist that information as well), and notice when all references to a file 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), 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 us to identify and remove unused objects, and maybe also objects that are lightly filled. These objects could be made redundant by rewriting their useful content directly into the active Btree.