Backups

WiredTiger cursors provide access to data from a variety of sources. One of these sources is the list of files required to perform a backup of the database. The list may be the files required by all of the objects in the database, or a subset of the objects in the database.

WiredTiger backups are "on-line" or "hot" backups, and applications may continue to read and write the databases while a snapshot is taken.

Backup from an application

1. Open a cursor on the `"backup:"` data source, which begins the process of a backup.
2. Copy each file returned by the **WT_CURSOR::next** method to the backup location, for example, a different directory. Do not reuse backup locations unless all files have first been removed from them, in other words, remove any previous backup information before using a backup location.
3. Close the cursor; the cursor must not be closed until all of the files have been copied.

The directory into which the files are copied may subsequently be specified as an directory to the **wiredtiger_open** function and accessed as a WiredTiger database home.

Copying the database files for a backup does not require any special alignment or block size (specifically, Linux or Windows filesystems that do not support read/write isolation can be safely read for backups).

The database file may grow in size during the copy, and the file copy should not consider that an error. Blocks appended to the file after the copy starts can be safely ignored, that is, it is correct for the copy to determine an initial size of the file and then copy that many bytes, ignoring any bytes appended after the backup cursor was opened.

The cursor must not be closed until all of the files have been copied, however, there is no requirement the files be copied in any order or in any relationship to the **WT_CURSOR::next** calls, only that all files have been copied before the cursor is closed. For example, applications might aggregate the file names from the cursor and then list the file names as arguments to a file archiver such as the system tar utility.

During the period the backup cursor is open, database checkpoints can be created, but no checkpoints can be deleted. This may result in significant file growth. Additionally while the backup cursor is open automatic log file archiving, even if enabled, will not reclaim any log files.

Additionally, if a crash occurs during the period the backup cursor is open and logging is disabled (in other words, when depending on checkpoints for durability), then the system will be restored to the most recent checkpoint prior to the opening of the backup cursor, even if later database checkpoints were completed. **Note this exception to WiredTiger's checkpoint durability guarantees.**

The following is a programmatic example of creating a full backup:

```
    WT_CURSOR *cursor;
    const char *filename;
    int ret;

    /* Create the backup directory. */
    error_check(mkdir("/path/database.backup", 077));

    /* Open the backup data source. */
    error_check(session->open_cursor(session, "backup:", NULL, NULL, &cursor));

    /* Copy the list of files. */
    while ((ret = cursor->next(cursor)) == 0) {
        error_check(cursor->get_key(cursor, &filename));
        (void)snprintf(
          buf, sizeof(buf), "cp /path/database/%s /path/database.backup/%s", filename, filename);
        error_check(system(buf));
    }
    scan_end_check(ret == WT_NOTFOUND);

    error_check(cursor->close(cursor));
```

In cases where the backup is desired for a checkpoint other than the most recent, applications can discard all checkpoints subsequent to the checkpoint they want using the **WT_SESSION::checkpoint** method. For example:

```
    error_check(session->checkpoint(session, "drop=(from=June01),name=June01"));
```

Backup from the command line

The **wt backup** command may also be used to create backups:

```
rm -rf /path/database.backup &&
    mkdir /path/database.backup &&
    wt -h /path/database.source backup /path/database.backup
```

Backup and O_DIRECT

Many Linux systems do not support mixing `O_DIRECT` and memory mapping or normal I/O to the same file. If `O_DIRECT` is configured for data or log files on Linux systems (using the wiredtiger_open `direct_io` configuration), any program used to copy files during backup should also specify `O_DIRECT` when configuring its file access. Likewise, when `O_DIRECT` is not configured by the database application, programs copying files should not configure `O_DIRECT`.

Incremental backup using log files

Once a backup has been done, it can be rolled forward incrementally by adding log files to the backup copy. Adding log files to the copy decreases potential data loss from switching to the copy, but increases the recovery time necessary to switch to the copy. To reset the recovery time necessary to switch to the copy, perform a full backup of the database. For example, an application might do a full backup of the database once a week during a quiet period, and then incrementally copy new log files into the backup directory for the rest of the week. Incremental backups may also save time when the tables are very large.

Bulk-loads are not commit-level durable, that is, the creation and bulk-load of an object will not appear in the database log files. For this reason, applications doing incremental backups after a full backup should repeat the full backup step after doing a bulk-load to make the bulk-load durable. In addition, incremental backups after a bulk-load can cause recovery to report errors because there are log records that apply to data files which don't appear in the backup.

By default, WiredTiger automatically removes log files no longer required for recovery. Applications wanting to use log files for incremental backup must first disable automatic log file removal using the `log`=(archive=false) configuration to **wiredtiger_open**.

The following is the procedure for incrementally backing up a database and removing log files from the original database home:

1. Perform a full backup of the database (as described above).
2. Open a cursor on the `"backup:"` data source, configured with the `"target=(\"log:\")"` target specified, which begins the process of an incremental backup.
3. Copy each log file returned by the **WT_CURSOR::next** method to the backup directory. It is not an error to copy a log file which has been copied before, but care should be taken to ensure each log file is completely copied as the most recent log file may grow in size while being copied.
4. If all log files have been successfully copied, archive the log files by calling the **WT_SESSION::truncate** method with the URI `log:` and specifying the backup cursor as the start cursor to that method. (Note there is no requirement backups be coordinated with database checkpoints, however, an incremental backup will repeatedly copy the same files, and will not make additional log files available for archival, unless there was a checkpoint after the previous incremental backup.)
5. Close the backup cursor.

Steps 2-5 can be repeated any number of times before step 1 is repeated. Full and incremental backups may be repeated as long as the backup database directory has not been opened and recovery run. Once recovery has run in a backup directory, you can no longer back up to that database directory.

An example of opening the backup data source for an incremental backup using log files:

```
    WT_CURSOR *cursor;
    const char *filename;
    int ret;

    /* Open the backup data source for incremental backup using log files. */
    error_check(session->open_cursor(session, "backup:", NULL, "target=(\"log:\")", &cursor));

    /* Copy the list of files. */
    while ((ret = cursor->next(cursor)) == 0) {
        error_check(cursor->get_key(cursor, &filename));
        (void)snprintf(
          buf, sizeof(buf), "cp /path/database/%s /path/database.backup/%s", filename, filename);
        error_check(system(buf));
    }
    scan_end_check(ret == WT_NOTFOUND);
```

```
    error_check(cursor->close(cursor));
```

If incremental backup fails (for example, if the backup directory runs out of space), a full backup should be performed.

Incremental backup using log files cannot be used in conjunction with incremental backup using block transfer.

Incremental backup using block transfer

Once a backup has been done, it can be rolled forward incrementally by updating file blocks in the backup copy.

The following is the procedure for incrementally backing up a database using block transfer from the original database home:

1. Perform a full backup of the database (as described above), with the additional configuration `incremental_preserve`.
2. Save the name of the full backup's origin checkpoint for use in subsequent incremental backups (it's the **`WT_CURSOR::checkpoint`** attribute of the backup cursor used to perform the full backup).
3. Begin the incremental backup by opening a cursor on the `"backup:"` data source, configured with `"incremental_checkpoint=name"`, where the checkpoint name is the name of the starting checkpoint from which the backup is being rolled forward.
4. For each file returned by the **WT_CURSOR::next** method, open a duplicate backup cursor on the file, configured with `"incremental_file=name"`, where the file name is the key returned by the backup cursor.
5. The key format for the duplicate backup cursor is `qq`, representing a file offset and size pair. There is no associated value. For each offset/size returned by the duplicate backup cursor's **WT_CURSOR::next** method, read the block from the source database file and write the block to the backup database file, replacing the current backup file's contents. It is not an error for an offset/size pair to extend past the current end of the file, and any missing data should be ignored.
6. Close the duplicate backup cursor.
7. Save the name of the checkpoint to which the backup was rolled forward for use as an origin checkpoint in subsequent incremental backups (it's the **`WT_CURSOR::checkpoint`** attribute of the backup cursor used to perform the incremental backup).
8. Close the backup cursor.

Steps 3-8 can be repeated any number of times. Incremental backups may be repeated as long as the backup database directory has not been opened and recovery run. Once recovery has run in a backup directory, you can no longer back up to that database directory.

Each time the incremental backup is repeated, the origin checkpoint name should be updated to a new value. If failure occurs at any point, the incremental backup may be repeated from the same origin checkpoint name; once an incremental backup has been performed using an updated origin checkpoint name, it will no longer be possible to perform an incremental backup using a previous origin checkpoint name.

The following is a programmatic example of creating a full backup with the expectation of performing future incremental backups:

```
    WT_CURSOR *backup_cursor;
    const char *filename, *backup_checkpoint;
    int ret;

    /* Create the backup directory. */
    error_check(mkdir("/path/database.backup", 077));

    /* Open the backup data source. */
    error_check(
      session->open_cursor(session, "backup:", NULL, "incremental_preserve=true", &backup_cursor));

    /* Copy the list of files. */
    while ((ret = backup_cursor->next(backup_cursor)) == 0) {
        error_check(backup_cursor->get_key(backup_cursor, &filename));
        (void)snprintf(
          buf, sizeof(buf), "cp /path/database/%s /path/database.backup/%s", filename, filename);
        error_check(system(buf));
    }
    scan_end_check(ret == WT_NOTFOUND);

    /* Save the current checkpoint name for future incremental backup use. */
    backup_checkpoint = strdup(backup_cursor->checkpoint);

    error_check(backup_cursor->close(backup_cursor));
```

An example of opening the backup data source for an incremental backup using block transfer:

```
    WT_CURSOR *backup_cursor, *incremental_cursor;
    uint64_t offset, size;
```

```
    int ret, rfd, wfd;
    const char *filename;

    /* Open the backup data source for incremental backup using log files. */
    (void)snprintf(buf, sizeof(buf), "incremental_checkpoint=%s", backup_checkpoint);
    error_check(session->open_cursor(session, "backup:", NULL, buf, &backup_cursor));

    /* For each file listed, open a duplicate backup cursor and copy the blocks. */
    while ((ret = backup_cursor->next(backup_cursor)) == 0) {
        error_check(backup_cursor->get_key(backup_cursor, &filename));
        (void)snprintf(buf, sizeof(buf), "/path/database/%s", filename);
        error_check(rfd = open(buf, O_RDONLY, 0));
        (void)snprintf(buf, sizeof(buf), "/path/database.backup/%s", filename);
        error_check(wfd = open(buf, O_WRONLY, 0));

        (void)snprintf(buf, sizeof(buf), "incremental_file=%s", filename);
        error_check(session->open_cursor(session, NULL, backup_cursor, buf, &incremental_cursor));
        while ((ret = incremental_cursor->next(incremental_cursor)) == 0) {
            error_check(incremental_cursor->get_key(incremental_cursor, &offset, &size));
            error_check(lseek(rfd, (off_t)offset, SEEK_SET));
            error_check(read(rfd, buf, size));
            error_check(lseek(wfd, (off_t)offset, SEEK_SET));
            error_check(write(wfd, buf, size));
        }
        scan_end_check(ret == WT_NOTFOUND);
        error_check(incremental_cursor->close(incremental_cursor));

        error_check(close(rfd));
        error_check(close(wfd));
    }
    scan_end_check(ret == WT_NOTFOUND);

    /* Save the current checkpoint name for future incremental backup use. */
    backup_checkpoint = strdup(backup_cursor->checkpoint);

    error_check(backup_cursor->close(backup_cursor));
```

To support incremental backup using block transfer, WiredTiger must track block changes across checkpoints which can require significant resources. The granularity of this tracking can be configured using the `incremental_granularity` configuration to **WT_SESSION::open_cursor** when opening the first full-backup cursor. A larger configuration reduces the amount of information WiredTiger must retain by conflating multiple block changes.

If incremental backup fails (for example, if the backup directory runs out of space), a full backup should be performed.

Incremental backup using block transfer cannot be used in conjunction with incremental backup using log files.