Write protocol

Protocol to ensure consistency : Write ahead logging (WAL)- The changes are first written to the log file using write system call and then a fdatasync on the log is done. The amount of data going to the log file is proportional to the data being written as part of the transaction. So this write can span many pages. It is possible that a crash can happen in the middle of the log writing. Berkeley DB handles this using checksums.

The transaction data going to the log file is protected by checksum in all cases. ie., when the system or the application crashes when writing the log file, the next open of the environment can detect that the last transaction had an incomplete write to the log file and can ignore to replay this transaction.

After writing to the log and syncing it, the datafile is written and a sync is done after writing. This write to the datafile can happen at various points. Below are some of them:

1. A close call on the database will flush all the datapages.
2. An explicit 'sync' call on the database.

Depending on a configuration flag, the log can be sync-ed to disk at various points (Please see last section for more details on this).

Assumptions and Vulnerabilities

1. Safe new file flush assumption on database file and the log files : Berkeley DB does not flush the directory after creating the datafile and the log files. In a scenario where the log files were written properly and a crash happens before writing the datafile, the next open of the database is supposed to replay the transaction to the data file. In this situation, Berkeley DB checks for the presence of all log files that were created. Because of the safe new file flush assumption, this check can fail and so the recovery will also fail. Similarly, in a non-transactional database, subsequent open calls may not show previously created databases.
2. This is a weird vulnerability : The write to the data file can span many pages. A crash in the middle of these writes can have different effects depending on when the crash happens. Berkeley DB documentation mentions about the atomic page writes requirement here. By this requirement, the application has to be configured to use the same write block size used by the operating system. For example, if the OS block size is 4K, then DB has to be also configured to write in 4K chunks. A crash that happens in middle of writing a datafile can cause corruption to the datafile. To catch this corruption, the database has to be configured with DB_CHKSUM. By

default, the datafile is *not* protected by checksums. If there were an interruption while writing to the datafile, next time open with DB_RECOVER flag can detect this and replay the transaction to the datafile from the log. In our experiment, the database was *not* enabled with checksums. The log is flushed to the disk on transaction commit. A crash happens when writing the pages to the data file. Also, this crash has to be page-non aligned -i.e., if the page size is 4K, the crash happens after writing 2K. The normal expectation would be that, the next open of the database should replay this transaction to the data file as it has been logged in the write-ahead log. But Berkeley does not do this and simply fails to open the database.

3. This is a documented bug : It is possible for the log file to get corrupted. The window of vulnerability is very small though.

4. Developer assumption: Following is a comment found in Berkeley DB code: "It must not be possible for the log file to be removed without previous file system level operations being flushed to stable storage. Berkeley DB applications write log records describing file system operations into the log, then perform the file system operation, then commit the enclosing transaction (which flushes the log file to stable storage). Subsequently, a database environment checkpoint may make it possible for the application to remove the log file containing the record of the file system operation. DB's transactional guarantees for file system operations require the log file removal not succeed until all previous filesystem operations have been flushed to stable storage. In other words, the flush of the log file, or the removal of the log file, must block until all previous filesystem operations have been flushed to stable storage. This semantic is not, as far as we know, required by any existing standards document, but we have never seen a filesystem where it does not apply." Note: We have not been able to reproduce a trace with unlink calls in checkpointing scenarios.

5. Environment open error because of missing log file : This is also an effect of safe new file assumption. In this case, if the directory entries for the log files were not persisted, subsequent environment open call fails with an error saying that the environment was not found.

Workloads and checkers

Berkeley DB supports transactional API and non-transactional API to interact with the store. Since the non-transactional API does not give any guarantee about consistency, our workloads were purely transactional.

We considered three different dimensions to write our workloads : STORAGE ENGINE, DURABILITY FLAGS, TRANSACTION SIZE.

There are four different storage engines internally used by Berkeley DB to organize the key value pairs: BTree, Hash, Queue & RecNo. There are three different durability flags: DB_TXN_SYNC, DB_TXN_NOSYNC, DB_TXN_WRITE_NOSYNC (Please see last section). Small transactions involve putting 10 k,v pairs as part of one transaction. Large transactions involve putting 1200 k,v pairs as part of one transaction - This internally involves log switching.

Apart from the above test matrix, few more experiments were run:

1. Multiple transactions - Berkeley DB uses same set of techniques to ensure integrity (Tested for all storage engines and transaction sizes).
2. Checkpointing - We tried to checkpoint the database at points. We did not notice significant changes to strace produced.

Results for small transactions experiment:

Engine	SYNC	NOSYNC	WRITENOSYNC
BTREE	pe	pew	pew
HASH	pe	pew	pew
QUEUE	p	pew	pew
RECNO	pe	pew	pew

Results for large transactions experiment:

Engine	SYNC	NOSYNC	WRITENOSYNC
BTREE	pe	pew	pew
HASH	pe	pew	pew
QUEUE	pe	pew	pew
RECNO	pe	pew	pew

LEGEND:

- p - Proper recovery
- w - Proper warning in documentation about durability
- e - Means a recovery after a page unaligned crash when writing datafile fails to recover though the log file is persistent. ie., the log file is persisted and a crash happens when the data file is being written. The crash

happens in such a way that it is not page aligned. Though the DB is not configured with checksum, they could have replayed this from the log. This could be prevented if checksums are enabled for the database and fatal recovery is done.

More notes about data persistence and integrity in Berkeley DB documentation

Following is a part of documentation that talks about system crashes and corruption:

"Note that as a normal part of closing a database, its cache is written to disk. However, in the event of an application or system failure, there is no guarantee that your databases will close cleanly. In this event, it is possible for you to lose data. Under extremely rare circumstances,it is also possible for you to experience database corruption.

Therefore, if you care if your data is durable across system failures, and to guard against the rare possibility of database corruption, you should use transactions to protect your database modifications. Every time you commit a transaction, DB ensures that the data will not be lost due to application or system failure.

If you do not want to use transactions, then the assumption is that your data is of a nature that it need not exist the next time your application starts. You may want this if, for example, you are using DB to cache data relevant only to the current application runtime. If, however, you are not using transactions for some reason and you still want some guarantee that your database modifications are persistent, then you should periodically call DB->sync(). Syncs cause any dirty entries in the in-memory cache and the operating system's file cache to be written to disk. As such, they are quite expensive and you should use them sparingly.

Remember that by default a sync is performed any time a non-transactional database is closed cleanly. (You can override this behavior by specifying DB_NOSYNC on the call to DB->close().) That said, you can manually run a sync by calling DB->sync().

NOTE: If your application or system crashes and you are not using transactions, then you should either discard and recreate your databases, or verify them. You can verify a database using DB->verify(). If your databases do not verify cleanly, use the db_dump command to salvage as much of the database as is possible. Use either the -R or -r command line options to control how aggressive db_dump should be when salvaging your databases."

Flags pertaining to durability and consistency

- DB_TXN_SYNC

- DB_TXN_NOSYNC

- DB_TXN_WRITE_NOSYNC

DB_TXN_SYNC - Synchronously flush the log when this transaction commits. This means the transaction will exhibit all of the ACID (atomicity, consistency, isolation, and durability) properties.

DB_TXN_NOSYNC - Do not synchronously flush the log when this transaction commits or prepares. This means the transaction will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained but it is possible that this transaction may be undone during recovery. This will not give durability in case of operating system and application crash.

DB_TXN_WRITE_NOSYNC - Berkeley uses 2 level caching. One that is internal to the application and the other is the operating system buffer cache. Write (from the application buffer to OS buffer cache), but do not synchronously flush, the log when this transaction commits. This means the transaction will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the system fails, it is possible some number of the most recently committed transactions may be undone during recovery. Note: This flag will still give durability in case of application crash.