

bbolt

go report **A+**  codecov **85%** build unknown go **documentation** release **v1.3.5**

license **MIT**

bbolt is a fork of [Ben Johnson's Bolt](#) key/value store. The purpose of this fork is to provide the Go community with an active maintenance and development target for Bolt; the goal is improved reliability and stability. bbolt includes bug fixes, performance enhancements, and features not found in Bolt while preserving backwards compatibility with the Bolt API.

Bolt is a pure Go key/value store inspired by [Howard Chu's LMDB project](#). The goal of the project is to provide a simple, fast, and reliable database for projects that don't require a full database server such as Postgres or MySQL.

Since Bolt is meant to be used as such a low-level piece of functionality, simplicity is key. The API will be small and only focus on getting values and setting values. That's it.

Project Status

Bolt is stable, the API is fixed, and the file format is fixed. Full unit test coverage and randomized black box testing are used to ensure database consistency and thread safety. Bolt is currently used in high-load production environments serving databases as large as 1TB. Many companies such as Shopify and Heroku use Bolt-backed services every day.

Project versioning

bbolt uses [semantic versioning](#). API should not change between patch and minor releases. New minor versions may add additional features to the API.

Table of Contents

- [Getting Started](#)
 - [Installing](#)
 - [Opening a database](#)
 - [Transactions](#)
 - [Read-write transactions](#)
 - [Read-only transactions](#)
 - [Batch read-write transactions](#)
 - [Managing transactions manually](#)
 - [Using buckets](#)
 - [Using key/value pairs](#)
 - [Autoincrementing integer for the bucket](#)
 - [Iterating over keys](#)
 - [Prefix scans](#)
 - [Range scans](#)
 - [ForEach\(\)](#)
 - [Nested buckets](#)
 - [Database backups](#)

- [Statistics](#)
- [Read-Only Mode](#)
- [Mobile Use \(iOS/Android\)](#)
- [Resources](#)
- [Comparison with other databases](#)
 - [Postgres, MySQL, & other relational databases](#)
 - [LevelDB, RocksDB](#)
 - [LMDB](#)
- [Caveats & Limitations](#)
- [Reading the Source](#)
- [Other Projects Using Bolt](#)

Getting Started

Installing

To start using Bolt, install Go and run `go get` :

```
$ go get go.etcd.io/bbolt/...
```

This will retrieve the library and install the `bolt` command line utility into your `$GOBIN` path.

Importing bbolt

To use bbolt as an embedded key-value store, import as:

```
import bolt "go.etcd.io/bbolt"

db, err := bolt.Open(path, 0666, nil)
if err != nil {
    return err
}
defer db.Close()
```

Opening a database

The top-level object in Bolt is a `DB`. It is represented as a single file on your disk and represents a consistent snapshot of your data.

To open your database, simply use the `bolt.Open()` function:

```
package main

import (
    "log"

    bolt "go.etcd.io/bbolt"
)

func main() {
```

```
// Open the my.db data file in your current directory.
// It will be created if it doesn't exist.
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
    log.Fatal(err)
}
defer db.Close()

...
}
```

Please note that Bolt obtains a file lock on the data file so multiple processes cannot open the same database at the same time. Opening an already open Bolt database will cause it to hang until the other process closes it. To prevent an indefinite wait you can pass a timeout option to the `Open()` function:

```
db, err := bolt.Open("my.db", 0600, &bolt.Options{Timeout: 1 * time.Second})
```

Transactions

Bolt allows only one read-write transaction at a time but allows as many read-only transactions as you want at a time. Each transaction has a consistent view of the data as it existed when the transaction started.

Individual transactions and all objects created from them (e.g. buckets, keys) are not thread safe. To work with data in multiple goroutines you must start a transaction for each one or use locking to ensure only one goroutine accesses a transaction at a time. Creating transaction from the `DB` is thread safe.

Transactions should not depend on one another and generally shouldn't be opened simultaneously in the same goroutine. This can cause a deadlock as the read-write transaction needs to periodically re-map the data file but it cannot do so while any read-only transaction is open. Even a nested read-only transaction can cause a deadlock, as the child transaction can block the parent transaction from releasing its resources.

Read-write transactions

To start a read-write transaction, you can use the `DB.Update()` function:

```
err := db.Update(func(tx *bolt.Tx) error {
    ...
    return nil
})
```

Inside the closure, you have a consistent view of the database. You commit the transaction by returning `nil` at the end. You can also rollback the transaction at any point by returning an error. All database operations are allowed inside a read-write transaction.

Always check the return error as it will report any disk failures that can cause your transaction to not complete. If you return an error within your closure it will be passed through.

Read-only transactions

To start a read-only transaction, you can use the `DB.View()` function:

```
err := db.View(func(tx *bolt.Tx) error {
    ...
    return nil
})
```

You also get a consistent view of the database within this closure, however, no mutating operations are allowed within a read-only transaction. You can only retrieve buckets, retrieve values, and copy the database within a read-only transaction.

Batch read-write transactions

Each `DB.Update()` waits for disk to commit the writes. This overhead can be minimized by combining multiple updates with the `DB.Batch()` function:

```
err := db.Batch(func(tx *bolt.Tx) error {
    ...
    return nil
})
```

Concurrent Batch calls are opportunistically combined into larger transactions. Batch is only useful when there are multiple goroutines calling it.

The trade-off is that `Batch` can call the given function multiple times, if parts of the transaction fail. The function must be idempotent and side effects must take effect only after a successful return from `DB.Batch()`.

For example: don't display messages from inside the function, instead set variables in the enclosing scope:

```
var id uint64
err := db.Batch(func(tx *bolt.Tx) error {
    // Find last key in bucket, decode as bigendian uint64, increment
    // by one, encode back to []byte, and add new key.
    ...
    id = newValue
    return nil
})
if err != nil {
    return ...
}
fmt.Println("Allocated ID %d", id)
```

Managing transactions manually

The `DB.View()` and `DB.Update()` functions are wrappers around the `DB.Begin()` function. These helper functions will start the transaction, execute a function, and then safely close your transaction if an error is returned. This is the recommended way to use Bolt transactions.

However, sometimes you may want to manually start and end your transactions. You can use the `DB.Begin()` function directly but **please** be sure to close the transaction.

```
// Start a writable transaction.
tx, err := db.Begin(true)
```

```

if err != nil {
    return err
}
defer tx.Rollback()

// Use the transaction...
_, err := tx.CreateBucket([]byte("MyBucket"))
if err != nil {
    return err
}

// Commit the transaction and check for error.
if err := tx.Commit(); err != nil {
    return err
}

```

The first argument to `DB.Begin()` is a boolean stating if the transaction should be writable.

Using buckets

Buckets are collections of key/value pairs within the database. All keys in a bucket must be unique. You can create a bucket using the `Tx.CreateBucket()` function:

```

db.Update(func(tx *bolt.Tx) error {
    b, err := tx.CreateBucket([]byte("MyBucket"))
    if err != nil {
        return fmt.Errorf("create bucket: %s", err)
    }
    return nil
})

```

You can also create a bucket only if it doesn't exist by using the `Tx.CreateBucketIfNotExists()` function. It's a common pattern to call this function for all your top-level buckets after you open your database so you can guarantee that they exist for future transactions.

To delete a bucket, simply call the `Tx.DeleteBucket()` function.

Using key/value pairs

To save a key/value pair to a bucket, use the `Bucket.Put()` function:

```

db.Update(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    err := b.Put([]byte("answer"), []byte("42"))
    return err
})

```

This will set the value of the `"answer"` key to `"42"` in the `MyBucket` bucket. To retrieve this value, we can use the `Bucket.Get()` function:

```

db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    v := b.Get([]byte("answer"))
    fmt.Printf("The answer is: %s\n", v)
    return nil
})

```

The `Get()` function does not return an error because its operation is guaranteed to work (unless there is some kind of system failure). If the key exists then it will return its byte slice value. If it doesn't exist then it will return `nil`. It's important to note that you can have a zero-length value set to a key which is different than the key not existing.

Use the `Bucket.Delete()` function to delete a key from the bucket.

Please note that values returned from `Get()` are only valid while the transaction is open. If you need to use a value outside of the transaction then you must use `copy()` to copy it to another byte slice.

Autoincrementing integer for the bucket

By using the `NextSequence()` function, you can let Bolt determine a sequence which can be used as the unique identifier for your key/value pairs. See the example below.

```

// CreateUser saves u to the store. The new user ID is set on u once the data is
// persisted.
func (s *Store) CreateUser(u *User) error {
    return s.db.Update(func(tx *bolt.Tx) error {
        // Retrieve the users bucket.
        // This should be created when the DB is first opened.
        b := tx.Bucket([]byte("users"))

        // Generate ID for the user.
        // This returns an error only if the Tx is closed or not writeable.
        // That can't happen in an Update() call so I ignore the error check.
        id, _ := b.NextSequence()
        u.ID = int(id)

        // Marshal user data into bytes.
        buf, err := json.Marshal(u)
        if err != nil {
            return err
        }

        // Persist bytes to users bucket.
        return b.Put(itob(u.ID), buf)
    })
}

// itob returns an 8-byte big endian representation of v.
func itob(v int) []byte {
    b := make([]byte, 8)
    binary.BigEndian.PutUint64(b, uint64(v))
}

```

```

    return b
}

type User struct {
    ID int
    ...
}

```

Iterating over keys

Bolt stores its keys in byte-sorted order within a bucket. This makes sequential iteration over these keys extremely fast. To iterate over keys we'll use a `Cursor` :

```

db.View(func(tx *bolt.Tx) error {
    // Assume bucket exists and has keys
    b := tx.Bucket([]byte("MyBucket"))

    c := b.Cursor()

    for k, v := c.First(); k != nil; k, v = c.Next() {
        fmt.Printf("key=%s, value=%s\n", k, v)
    }

    return nil
})

```

The cursor allows you to move to a specific point in the list of keys and move forward or backward through the keys one at a time.

The following functions are available on the cursor:

```

First()  Move to the first key.
Last()   Move to the last key.
Seek()   Move to a specific key.
Next()   Move to the next key.
Prev()   Move to the previous key.

```

Each of those functions has a return signature of `(key []byte, value []byte)` . When you have iterated to the end of the cursor then `Next()` will return a `nil` key. You must seek to a position using `First()` , `Last()` , or `Seek()` before calling `Next()` or `Prev()` . If you do not seek to a position then these functions will return a `nil` key.

During iteration, if the key is non- `nil` but the value is `nil` , that means the key refers to a bucket rather than a value. Use `Bucket.Bucket()` to access the sub-bucket.

Prefix scans

To iterate over a key prefix, you can combine `Seek()` and `bytes.HasPrefix()` :

```

db.View(func(tx *bolt.Tx) error {
    // Assume bucket exists and has keys

```

```

c := tx.Bucket([]byte("MyBucket")).Cursor()

prefix := []byte("1234")
for k, v := c.Seek(prefix); k != nil && bytes.HasPrefix(k, prefix); k, v =
c.Next() {
    fmt.Printf("key=%s, value=%s\n", k, v)
}

return nil
})

```

Range scans

Another common use case is scanning over a range such as a time range. If you use a sortable time encoding such as RFC3339 then you can query a specific date range like this:

```

db.View(func(tx *bolt.Tx) error {
    // Assume our events bucket exists and has RFC3339 encoded time keys.
    c := tx.Bucket([]byte("Events")).Cursor()

    // Our time range spans the 90's decade.
    min := []byte("1990-01-01T00:00:00Z")
    max := []byte("2000-01-01T00:00:00Z")

    // Iterate over the 90's.
    for k, v := c.Seek(min); k != nil && bytes.Compare(k, max) <= 0; k, v = c.Next()
    {
        fmt.Printf("%s: %s\n", k, v)
    }

    return nil
})

```

Note that, while RFC3339 is sortable, the Golang implementation of RFC3339Nano does not use a fixed number of digits after the decimal point and is therefore not sortable.

ForEach()

You can also use the function `ForEach()` if you know you'll be iterating over all the keys in a bucket:

```

db.View(func(tx *bolt.Tx) error {
    // Assume bucket exists and has keys
    b := tx.Bucket([]byte("MyBucket"))

    b.ForEach(func(k, v []byte) error {
        fmt.Printf("key=%s, value=%s\n", k, v)
        return nil
    })

    return nil
})

```


Please note that keys and values in `ForEach()` are only valid while the transaction is open. If you need to use a key or value outside of the transaction, you must use `copy()` to copy it to another byte slice.

Nested buckets

You can also store a bucket in a key to create nested buckets. The API is the same as the bucket management API on the `DB` object:

```
func (*Bucket) CreateBucket(key []byte) (*Bucket, error)
func (*Bucket) CreateBucketIfNotExists(key []byte) (*Bucket, error)
func (*Bucket) DeleteBucket(key []byte) error
```

Say you had a multi-tenant application where the root level bucket was the account bucket. Inside of this bucket was a sequence of accounts which themselves are buckets. And inside the sequence bucket you could have many buckets pertaining to the Account itself (Users, Notes, etc) isolating the information into logical groupings.

```
// createUser creates a new user in the given account.
func createUser(accountID int, u *User) error {
    // Start the transaction.
    tx, err := db.Begin(true)
    if err != nil {
        return err
    }
    defer tx.Rollback()

    // Retrieve the root bucket for the account.
    // Assume this has already been created when the account was set up.
    root := tx.Bucket([]byte(strconv.FormatUint(accountID, 10)))

    // Setup the users bucket.
    bkt, err := root.CreateBucketIfNotExists([]byte("USERS"))
    if err != nil {
        return err
    }

    // Generate an ID for the new user.
    userID, err := bkt.NextSequence()
    if err != nil {
        return err
    }
    u.ID = userID

    // Marshal and save the encoded user.
    if buf, err := json.Marshal(u); err != nil {
        return err
    } else if err := bkt.Put([]byte(strconv.FormatUint(u.ID, 10)), buf); err != nil {
    }

    return err
}
```

```

    // Commit the transaction.
    if err := tx.Commit(); err != nil {
        return err
    }

    return nil
}

```

Database backups

Bolt is a single file so it's easy to backup. You can use the `Tx.WriteTo()` function to write a consistent view of the database to a writer. If you call this from a read-only transaction, it will perform a hot backup and not block your other database reads and writes.

By default, it will use a regular file handle which will utilize the operating system's page cache. See the [Tx](#) documentation for information about optimizing for larger-than-RAM datasets.

One common use case is to backup over HTTP so you can use tools like `cURL` to do database backups:

```

func BackupHandleFunc(w http.ResponseWriter, req *http.Request) {
    err := db.View(func(tx *bolt.Tx) error {
        w.Header().Set("Content-Type", "application/octet-stream")
        w.Header().Set("Content-Disposition", `attachment; filename="my.db"`)
        w.Header().Set("Content-Length", strconv.Itoa(int(tx.Size())))
        _, err := tx.WriteTo(w)
        return err
    })
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}

```

Then you can backup using this command:

```
$ curl http://localhost/backup > my.db
```

Or you can open your browser to `http://localhost/backup` and it will download automatically.

If you want to backup to another file you can use the `Tx.CopyFile()` helper function.

Statistics

The database keeps a running count of many of the internal operations it performs so you can better understand what's going on. By grabbing a snapshot of these stats at two points in time we can see what operations were performed in that time range.

For example, we could start a goroutine to log stats every 10 seconds:

```

go func() {
    // Grab the initial stats.
    prev := db.Stats()

```

```

for {
    // Wait for 10s.
    time.Sleep(10 * time.Second)

    // Grab the current stats and diff them.
    stats := db.Stats()
    diff := stats.Sub(&prev)

    // Encode stats to JSON and print to STDERR.
    json.NewEncoder(os.Stderr).Encode(diff)

    // Save stats for the next loop.
    prev = stats
}
}()

```

It's also useful to pipe these stats to a service such as `statsd` for monitoring or to provide an HTTP endpoint that will perform a fixed-length sample.

Read-Only Mode

Sometimes it is useful to create a shared, read-only Bolt database. To this, set the `Options.ReadOnly` flag when opening your database. Read-only mode uses a shared lock to allow multiple processes to read from the database but it will block any processes from opening the database in read-write mode.

```

db, err := bolt.Open("my.db", 0666, &bolt.Options{ReadOnly: true})
if err != nil {
    log.Fatal(err)
}

```

Mobile Use (iOS/Android)

Bolt is able to run on mobile devices by leveraging the binding feature of the [gomobile](#) tool. Create a struct that will contain your database logic and a reference to a `*bolt.DB` with a initializing constructor that takes in a filepath where the database file will be stored. Neither Android nor iOS require extra permissions or cleanup from using this method.

```

func NewBoltDB(filepath string) *BoltDB {
    db, err := bolt.Open(filepath+"/demo.db", 0600, nil)
    if err != nil {
        log.Fatal(err)
    }

    return &BoltDB{db}
}

type BoltDB struct {
    db *bolt.DB
    ...
}

```

```

func (b *BoltDB) Path() string {
    return b.db.Path()
}

func (b *BoltDB) Close() {
    b.db.Close()
}

```

Database logic should be defined as methods on this wrapper struct.

To initialize this struct from the native language (both platforms now sync their local storage to the cloud. These snippets disable that functionality for the database file):

Android

```

String path;
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
    path = getNoBackupFilesDir().getAbsolutePath();
} else {
    path = getFilesDir().getAbsolutePath();
}
Boltmobiledemo.BoltDB boltDB = Boltmobiledemo.NewBoltDB(path)

```

iOS

```

- (void)demo {
    NSString* path = [NSSearchPathForDirectoriesInDomains(NSLibraryDirectory,
                                                            NSUserDomainMask,
                                                            YES) objectAtIndex:0];

    GoBoltmobiledemoBoltDB * demo = GoBoltmobiledemoNewBoltDB(path);
    [self addSkipBackupAttributeToItemAtPath:demo.path];
    //Some DB Logic would go here
    [demo close];
}

- (BOOL)addSkipBackupAttributeToItemAtPath:(NSString *) filePathString
{
    NSURL* URL= [NSURL fileURLWithPath: filePathString];
    assert([[NSFileManager defaultManager] fileExistsAtPath: [URL path]]);

    NSError *error = nil;
    BOOL success = [URL setResourceValue: [NSNumber numberWithBool: YES]
                                forKey: NSURLIsExcludedFromBackupKey error:
&error];
    if(!success){
        NSLog(@"Error excluding %@ from backup %@", [URL lastPathComponent], error);
    }
    return success;
}

```

Resources

For more information on getting started with Bolt, check out the following articles:

- [Intro to BoltDB: Painless Performant Persistence](#) by [Nate Finch](#).
- [Bolt -- an embedded key/value database for Go](#) by Progvile

Comparison with other databases

Postgres, MySQL, & other relational databases

Relational databases structure data into rows and are only accessible through the use of SQL. This approach provides flexibility in how you store and query your data but also incurs overhead in parsing and planning SQL statements.

Bolt accesses all data by a byte slice key. This makes Bolt fast to read and write data by key but provides no built-in support for joining values together.

Most relational databases (with the exception of SQLite) are standalone servers that run separately from your application. This gives your systems flexibility to connect multiple application servers to a single database server but also adds overhead in serializing and transporting data over the network. Bolt runs as a library included in your application so all data access has to go through your application's process. This brings data closer to your application but limits multi-process access to the data.

LevelDB, RocksDB

LevelDB and its derivatives (RocksDB, HyperLevelDB) are similar to Bolt in that they are libraries bundled into the application, however, their underlying structure is a log-structured merge-tree (LSM tree). An LSM tree optimizes random writes by using a write ahead log and multi-tiered, sorted files called SSTables. Bolt uses a B+tree internally and only a single file. Both approaches have trade-offs.

If you require a high random write throughput (> 10,000 w/sec) or you need to use spinning disks then LevelDB could be a good choice. If your application is read-heavy or does a lot of range scans then Bolt could be a good choice.

One other important consideration is that LevelDB does not have transactions. It supports batch writing of key/values pairs and it supports read snapshots but it will not give you the ability to do a compare-and-swap operation safely. Bolt supports fully serializable ACID transactions.

LMDB

Bolt was originally a port of LMDB so it is architecturally similar. Both use a B+tree, have ACID semantics with fully serializable transactions, and support lock-free MVCC using a single writer and multiple readers.

The two projects have somewhat diverged. LMDB heavily focuses on raw performance while Bolt has focused on simplicity and ease of use. For example, LMDB allows several unsafe actions such as direct writes for the sake of performance. Bolt opts to disallow actions which can leave the database in a corrupted state. The only exception to this in Bolt is `DB.NoSync`.

There are also a few differences in API. LMDB requires a maximum mmap size when opening an `mdb_env` whereas Bolt will handle incremental mmap resizing automatically. LMDB overloads the getter and setter functions with multiple flags whereas Bolt splits these specialized cases into their own functions.

Caveats & Limitations

It's important to pick the right tool for the job and Bolt is no exception. Here are a few things to note when evaluating and using Bolt:

- Bolt is good for read intensive workloads. Sequential write performance is also fast but random writes can be slow. You can use `DB.Batch()` or add a write-ahead log to help mitigate this issue.
- Bolt uses a B+tree internally so there can be a lot of random page access. SSDs provide a significant performance boost over spinning disks.
- Try to avoid long running read transactions. Bolt uses copy-on-write so old pages cannot be reclaimed while an old transaction is using them.
- Byte slices returned from Bolt are only valid during a transaction. Once the transaction has been committed or rolled back then the memory they point to can be reused by a new page or can be unmapped from virtual memory and you'll see an `unexpected fault address` panic when accessing it.
- Bolt uses an exclusive write lock on the database file so it cannot be shared by multiple processes.
- Be careful when using `Bucket.FillPercent`. Setting a high fill percent for buckets that have random inserts will cause your database to have very poor page utilization.
- Use larger buckets in general. Smaller buckets causes poor page utilization once they become larger than the page size (typically 4KB).
- Bulk loading a lot of random writes into a new bucket can be slow as the page will not split until the transaction is committed. Randomly inserting more than 100,000 key/value pairs into a single new bucket in a single transaction is not advised.
- Bolt uses a memory-mapped file so the underlying operating system handles the caching of the data. Typically, the OS will cache as much of the file as it can in memory and will release memory as needed to other processes. This means that Bolt can show very high memory usage when working with large databases. However, this is expected and the OS will release memory as needed. Bolt can handle databases much larger than the available physical RAM, provided its memory-map fits in the process virtual address space. It may be problematic on 32-bits systems.
- The data structures in the Bolt database are memory mapped so the data file will be endian specific. This means that you cannot copy a Bolt file from a little endian machine to a big endian machine and have it work. For most users this is not a concern since most modern CPUs are little endian.
- Because of the way pages are laid out on disk, Bolt cannot truncate data files and return free pages back to the disk. Instead, Bolt maintains a free list of unused pages within its data file. These free pages can be reused by later transactions. This works well for many use cases as databases generally tend to grow. However, it's important to note that deleting large chunks of data will not allow you to reclaim that space on disk.

For more information on page allocation, [see this comment](#).

Reading the Source

Bolt is a relatively small code base (<5KLOC) for an embedded, serializable, transactional key/value database so it can be a good starting point for people interested in how databases work.

The best places to start are the main entry points into Bolt:


- `Open()` - Initializes the reference to the database. It's responsible for creating the database if it doesn't exist, obtaining an exclusive lock on the file, reading the meta pages, & memory-mapping the file.

- `DB.Begin()` - Starts a read-only or read-write transaction depending on the value of the `writable` argument. This requires briefly obtaining the "meta" lock to keep track of open transactions. Only one read-write transaction can exist at a time so the "rwlock" is acquired during the life of a read-write transaction.
- `Bucket.Put()` - Writes a key/value pair into a bucket. After validating the arguments, a cursor is used to traverse the B+tree to the page and position where they key & value will be written. Once the position is found, the bucket materializes the underlying page and the page's parent pages into memory as "nodes". These nodes are where mutations occur during read-write transactions. These changes get flushed to disk during commit.
- `Bucket.Get()` - Retrieves a key/value pair from a bucket. This uses a cursor to move to the page & position of a key/value pair. During a read-only transaction, the key and value data is returned as a direct reference to the underlying mmap file so there's no allocation overhead. For read-write transactions, this data may reference the mmap file or one of the in-memory node values.
- `Cursor` - This object is simply for traversing the B+tree of on-disk pages or in-memory nodes. It can seek to a specific key, move to the first or last value, or it can move forward or backward. The cursor handles the movement up and down the B+tree transparently to the end user.
- `Tx.Commit()` - Converts the in-memory dirty nodes and the list of free pages into pages to be written to disk. Writing to disk then occurs in two phases. First, the dirty pages are written to disk and an `fsync()` occurs. Second, a new meta page with an incremented transaction ID is written and another `fsync()` occurs. This two phase write ensures that partially written data pages are ignored in the event of a crash since the meta page pointing to them is never written. Partially written meta pages are invalidated because they are written with a checksum.

If you have additional notes that could be helpful for others, please submit them via pull request.

Other Projects Using Bolt

Below is a list of public, open source projects that use Bolt:

- [Algernon](#) - A HTTP/2 web server with built-in support for Lua. Uses BoltDB as the default database backend.
- [Bazil](#) - A file system that lets your data reside where it is most convenient for it to reside.
- [bolter](#) - Command-line app for viewing BoltDB file in your terminal.
- [boltcli](#) - the redis-cli for boltdb with Lua script support.
- [BoltHold](#) - An embeddable NoSQL store for Go types built on BoltDB
- [BoltStore](#) - Session store using Bolt.
- [Boltdb Boilerplate](#) - Boilerplate wrapper around bolt aiming to make simple calls one-liners.
- [BoltDbWeb](#) - A web based GUI for BoltDB files.
- [BoltDB Viewer](#) - A BoltDB Viewer Can run on Windows, Linux, Android system.
- [bleve](#) - A pure Go search engine similar to Elasticsearch that uses Bolt as the default storage backend.
- [btcwallet](#) - A bitcoin wallet.
- [buckets](#) - a bolt wrapper streamlining simple tx and key scans.
- [cayley](#) - Cayley is an open-source graph database using Bolt as optional backend.
- [ChainStore](#) - Simple key-value interface to a variety of storage engines organized as a chain of operations.
-  [Chestnut](#) - Chestnut is encrypted storage for Go.
- [Consul](#) - Consul is service discovery and configuration made easy. Distributed, highly available, and datacenter-aware.
- [DVID](#) - Added Bolt as optional storage engine and testing it against Basho-tuned leveldb.
- [dcrwallet](#) - A wallet for the Decred cryptocurrency.
- [drive](#) - drive is an unofficial Google Drive command line client for *NIX operating systems.

- [event-shuttle](#) - A Unix system service to collect and reliably deliver messages to Kafka.
- [Freehold](#) - An open, secure, and lightweight platform for your files and data.
- [Go Report Card](#) - Go code quality report cards as a (free and open source) service.
- [GoWebApp](#) - A basic MVC web application in Go using BoltDB.
- [GoShort](#) - GoShort is a URL shortener written in Golang and BoltDB for persistent key/value storage and for routing it's using high performant HTTPRouter.
- [gopherpit](#) - A web service to manage Go remote import paths with custom domains
- [gokv](#) - Simple key-value store abstraction and implementations for Go (Redis, Consul, etcd, bbolt, BadgerDB, LevelDB, Memcached, DynamoDB, S3, PostgreSQL, MongoDB, CockroachDB and many more)
- [Gitchain](#) - Decentralized, peer-to-peer Git repositories aka "Git meets Bitcoin".
- [InfluxDB](#) - Scalable datastore for metrics, events, and real-time analytics.
- [ipLocator](#) - A fast ip-geo-location-server using bolt with bloom filters.
- [ipxed](#) - Web interface and api for ipxed.
- [Ironsmitth](#) - A simple, script-driven continuous integration (build -> test -> release) tool, with no external dependencies
- [Kala](#) - Kala is a modern job scheduler optimized to run on a single node. It is persistent, JSON over HTTP API, ISO 8601 duration notation, and dependent jobs.
- [Key Value Access Language \(KVAL\)](#) - A proposed grammar for key-value datastores offering a bbolt binding.
- [LedisDB](#) - A high performance NoSQL, using Bolt as optional storage.
- [lru](#) - Easy to use Bolt-backed Least-Recently-Used (LRU) read-through cache with chainable remote stores.
- [mbuckets](#) - A Bolt wrapper that allows easy operations on multi level (nested) buckets.
- [MetricBase](#) - Single-binary version of Graphite.
- [MuLiFS](#) - Music Library Filesystem creates a filesystem to organise your music files.
- [NATS](#) - NATS Streaming uses bbolt for message and metadata storage.
- [Prometheus Annotation Server](#) - Annotation server for PromDash & Prometheus service monitoring system.
- [Rain](#) - BitTorrent client and library.
- [reef-pi](#) - reef-pi is an award winning, modular, DIY reef tank controller using easy to learn electronics based on a Raspberry Pi.
- [Request Baskets](#) - A web service to collect arbitrary HTTP requests and inspect them via REST API or simple web UI, similar to [RequestBin](#) service
- [Seaweed File System](#) - Highly scalable distributed key~file system with O(1) disk read.
- [stow](#) - a persistence manager for objects backed by boltddb.
- [Storm](#) - Simple and powerful ORM for BoltDB.
- [SimpleBolt](#) - A simple way to use BoltDB. Deals mainly with strings.
- [Skybox Analytics](#) - A standalone funnel analysis tool for web analytics.
- [Scuttlebutt](#) - Uses Bolt to store and process all Twitter mentions of GitHub projects.
- [tentacool](#) - REST api server to manage system stuff (IP, DNS, Gateway...) on a linux server.
- [torrent](#) - Full-featured BitTorrent client package and utilities in Go. BoltDB is a storage backend in development.
- [Wiki](#) - A tiny wiki using Goji, BoltDB and Blackfriday.

If you are using Bolt in a project please send a pull request to add it to the list.