

---

## Storm



Storm is a simple and powerful toolkit for BoltDB. Basically, Storm provides indexes, a wide range of methods to store and fetch data, an advanced query system, and much more.

In addition to the examples below, see also the examples in the GoDoc.

*For extended queries and support for Badger, see also Genji*

### Table of Contents

- Getting Started
- Import Storm
- Open a database
- Simple CRUD system
  - Declare your structures
  - Save your object
    - \* Auto Increment
  - Simple queries
    - \* Fetch one object
    - \* Fetch multiple objects
    - \* Fetch all objects
    - \* Fetch all objects sorted by index
    - \* Fetch a range of objects
    - \* Fetch objects by prefix
    - \* Skip, Limit and Reverse
    - \* Delete an object
    - \* Update an object
    - \* Initialize buckets and indexes before saving an object
    - \* Drop a bucket
    - \* Re-index a bucket
  - Advanced queries
  - Transactions
  - Options
    - \* BoltOptions
    - \* MarshalUnmarshaler

- 
- Provided Codecs
    - \* Use existing Bolt connection
    - \* Batch mode
  - Nodes and nested buckets
    - Node options
  - Simple Key/Value store
  - BoltDB
  - License
  - Credits

## Getting Started

```
1 GO111MODULE=on go get -u github.com/asdine/storm/v3
```

## Import Storm

```
1 import "github.com/asdine/storm/v3"
```

## Open a database

Quick way of opening a database

```
1 db, err := storm.Open("my.db")
2
3 defer db.Close()
```

`Open` can receive multiple options to customize the way it behaves. See Options below

## Simple CRUD system

### Declare your structures

```
1 type User struct {
2     ID int // primary key
3     Group string `storm:"index"` // this field will be indexed
4     Email string `storm:"unique"` // this field will be indexed with a
        unique constraint
```

---

```
5 Name string // this field will not be indexed
6 Age int `storm:"index"`
7 }
```

The primary key can be of any type as long as it is not a zero value. Storm will search for the tag `id`, if not present Storm will search for a field named `ID`.

```
1 type User struct {
2     ThePrimaryKey string `storm:"id"` // primary key
3     Group string `storm:"index"` // this field will be indexed
4     Email string `storm:"unique"` // this field will be indexed with a
        unique constraint
5     Name string // this field will not be indexed
6 }
```

Storm handles tags in nested structures with the `inline` tag

```
1 type Base struct {
2     Ident bson.ObjectId `storm:"id"`
3 }
4
5 type User struct {
6     Base `storm:"inline"`
7     Group string `storm:"index"`
8     Email string `storm:"unique"`
9     Name string
10    CreatedAt time.Time `storm:"index"`
11 }
```

## Save your object

```
1 user := User{
2     ID: 10,
3     Group: "staff",
4     Email: "john@provider.com",
5     Name: "John",
6     Age: 21,
7     CreatedAt: time.Now(),
8 }
9
10 err := db.Save(&user)
11 // err == nil
12
13 user.ID++
14 err = db.Save(&user)
15 // err == storm.ErrAlreadyExists
```

That's it.

---

**Save** creates or updates all the required indexes and buckets, checks the unique constraints and saves the object to the store.

**Auto Increment** Storm can auto increment integer values so you don't have to worry about that when saving your objects. Also, the new value is automatically inserted in your field.

```
1
2 type Product struct {
3     Pk                int `storm:"id,increment"` // primary key with
                        auto increment
4     Name              string
5     IntegerField      uint64 `storm:"increment"`
6     IndexedIntegerField uint32 `storm:"index,increment"`
7     UniqueIntegerField int16 `storm:"unique,increment=100"` // the
                        starting value can be set
8 }
9
10 p := Product{Name: "Vaccum Cleaner"}
11
12 fmt.Println(p.Pk)
13 fmt.Println(p.IntegerField)
14 fmt.Println(p.IndexedIntegerField)
15 fmt.Println(p.UniqueIntegerField)
16 // 0
17 // 0
18 // 0
19 // 0
20
21 _ = db.Save(&p)
22
23 fmt.Println(p.Pk)
24 fmt.Println(p.IntegerField)
25 fmt.Println(p.IndexedIntegerField)
26 fmt.Println(p.UniqueIntegerField)
27 // 1
28 // 1
29 // 1
30 // 100
```

## Simple queries

Any object can be fetched, indexed or not. Storm uses indexes when available, otherwise it uses the query system.

### Fetch one object

```
1 var user User
```

---

```
2 err := db.One("Email", "john@provider.com", &user)
3 // err == nil
4
5 err = db.One("Name", "John", &user)
6 // err == nil
7
8 err = db.One("Name", "Jack", &user)
9 // err == storm.ErrNotFound
```

#### Fetch multiple objects

```
1 var users []User
2 err := db.Find("Group", "staff", &users)
```

#### Fetch all objects

```
1 var users []User
2 err := db.All(&users)
```

#### Fetch all objects sorted by index

```
1 var users []User
2 err := db.AllByIndex("CreatedAt", &users)
```

#### Fetch a range of objects

```
1 var users []User
2 err := db.Range("Age", 10, 21, &users)
```

#### Fetch objects by prefix

```
1 var users []User
2 err := db.Prefix("Name", "Jo", &users)
```

#### Skip, Limit and Reverse

```
1 var users []User
2 err := db.Find("Group", "staff", &users, storm.Skip(10))
3 err = db.Find("Group", "staff", &users, storm.Limit(10))
4 err = db.Find("Group", "staff", &users, storm.Reverse())
5 err = db.Find("Group", "staff", &users, storm.Limit(10), storm.Skip(10),
6     , storm.Reverse())
7 err = db.All(&users, storm.Limit(10), storm.Skip(10), storm.Reverse())
8 err = db.AllByIndex("CreatedAt", &users, storm.Limit(10), storm.Skip(10),
9     storm.Reverse())
9 err = db.Range("Age", 10, 21, &users, storm.Limit(10), storm.Skip(10),
    storm.Reverse())
```

#### Delete an object

```
1 err := db.DeleteStruct(&user)
```

---

### Update an object

```
1 // Update multiple fields
2 // Only works for non zero-value fields (e.g. Name can not be "", Age
  can not be 0)
3 err := db.Update(&User{ID: 10, Name: "Jack", Age: 45})
4
5 // Update a single field
6 // Also works for zero-value fields (0, false, "", ...)
7 err := db.UpdateField(&User{ID: 10}, "Age", 0)
```

### Initialize buckets and indexes before saving an object

```
1 err := db.Init(&User{})
```

Useful when starting your application

### Drop a bucket Using the struct

```
1 err := db.Drop(&User)
```

Using the bucket name

```
1 err := db.Drop("User")
```

### Re-index a bucket

```
1 err := db.ReIndex(&User{})
```

Useful when the structure has changed

## Advanced queries

For more complex queries, you can use the `Select` method. `Select` takes any number of `Matcher` from the `q` package.

Here are some common Matchers:

```
1 // Equality
2 q.Eq("Name", John)
3
4 // Strictly greater than
5 q.Gt("Age", 7)
6
7 // Lesser than or equal to
8 q.Lte("Age", 77)
9
10 // Regex with name that starts with the letter D
11 q.Re("Name", "^D")
```

---

```

12
13 // In the given slice of values
14 q.In("Group", []string{"Staff", "Admin"})
15
16 // Comparing fields
17 q.EqF("FieldName", "SecondFieldName")
18 q.LtF("FieldName", "SecondFieldName")
19 q.GtF("FieldName", "SecondFieldName")
20 q.LteF("FieldName", "SecondFieldName")
21 q.GteF("FieldName", "SecondFieldName")

```

Matchers can also be combined with **And**, **Or** and **Not**:

```

1
2 // Match if all match
3 q.And(
4     q.Gt("Age", 7),
5     q.Re("Name", "^D")
6 )
7
8 // Match if one matches
9 q.Or(
10    q.Re("Name", "^A"),
11    q.Not(
12        q.Re("Name", "^B")
13    ),
14    q.Re("Name", "^C"),
15    q.In("Group", []string{"Staff", "Admin"}),
16    q.And(
17        q.StrictEq("Password", []byte(password)),
18        q.Eq("Registered", true)
19    )
20 )

```

You can find the complete list in the documentation.

**Select** takes any number of matchers and wraps them into a `q.And()` so it's not necessary to specify it. It returns a **Query** type.

```

1 query := db.Select(q.Gte("Age", 7), q.Lte("Age", 77))

```

The **Query** type contains methods to filter and order the records.

```

1 // Limit
2 query = query.Limit(10)
3
4 // Skip
5 query = query.Skip(20)
6
7 // Calls can also be chained
8 query = query.Limit(10).Skip(20).OrderBy("Age").Reverse()

```

---

But also to specify how to fetch them.

```
1 var users []User
2 err = query.Find(&users)
3
4 var user User
5 err = query.First(&user)
```

Examples with `Select`:

```
1 // Find all users with an ID between 10 and 100
2 err = db.Select(q.Gte("ID", 10), q.Lte("ID", 100)).Find(&users)
3
4 // Nested matchers
5 err = db.Select(q.Or(
6     q.Gt("ID", 50),
7     q.Lt("Age", 21),
8     q.And(
9         q.Eq("Group", "admin"),
10        q.Gte("Age", 21),
11    ),
12)).Find(&users)
13
14 query := db.Select(q.Gte("ID", 10), q.Lte("ID", 100)).Limit(10).Skip(5)
15     .Reverse().OrderBy("Age", "Name")
16
17 // Find multiple records
18 err = query.Find(&users)
19 // or
20 err = db.Select(q.Gte("ID", 10), q.Lte("ID", 100)).Limit(10).Skip(5).
21     Reverse().OrderBy("Age", "Name").Find(&users)
22
23 // Find first record
24 err = query.First(&user)
25 // or
26 err = db.Select(q.Gte("ID", 10), q.Lte("ID", 100)).Limit(10).Skip(5).
27     Reverse().OrderBy("Age", "Name").First(&user)
28
29 // Delete all matching records
30 err = query.Delete(new(User))
31
32 // Fetching records one by one (useful when the bucket contains a lot
33 // of records)
34 query = db.Select(q.Gte("ID", 10), q.Lte("ID", 100)).OrderBy("Age", "
35     Name")
36
37 err = query.Each(new(User), func(record interface{}) error) {
38     u := record.(*User)
39     ...
40 }
```



---

```
35     return nil
36 }
```

See the documentation for a complete list of methods.

## Transactions

```
1 tx, err := db.Begin(true)
2 if err != nil {
3     return err
4 }
5 defer tx.Rollback()
6
7 accountA.Amount -= 100
8 accountB.Amount += 100
9
10 err = tx.Save(accountA)
11 if err != nil {
12     return err
13 }
14
15 err = tx.Save(accountB)
16 if err != nil {
17     return err
18 }
19
20 return tx.Commit()
```

## Options

Storm options are functions that can be passed when constructing you Storm instance. You can pass it any number of options.

**BoltOptions** By default, Storm opens a database with the mode 0600 and a timeout of one second. You can change this behavior by using [BoltOptions](#)

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

**MarshalUnmarshaler** To store the data in BoltDB, Storm marshals it in JSON by default. If you wish to change this behavior you can pass a codec that implements [codec.MarshalUnmarshaler](#) via the [storm.Codec](#) option:

---

```
1 db := storm.Open("my.db", storm.Codec(myCodec))
```

**Provided Codecs** You can easily implement your own `MarshalUnmarshaler`, but Storm comes with built-in support for JSON (default), GOB, Sereal, Protocol Buffers and MessagePack.

These can be used by importing the relevant package and use that codec to configure Storm. The example below shows all variants (without proper error handling):

```
1 import (
2     "github.com/asdine/storm/v3"
3     "github.com/asdine/storm/v3/codec/gob"
4     "github.com/asdine/storm/v3/codec/json"
5     "github.com/asdine/storm/v3/codec/sereal"
6     "github.com/asdine/storm/v3/codec/protobuf"
7     "github.com/asdine/storm/v3/codec/msgpack"
8 )
9
10 var gobDb, _ = storm.Open("gob.db", storm.Codec(gob.Codec))
11 var jsonDb, _ = storm.Open("json.db", storm.Codec(json.Codec))
12 var serealDb, _ = storm.Open("sereal.db", storm.Codec(sereal.Codec))
13 var protobufDb, _ = storm.Open("protobuf.db", storm.Codec(protobuf.Codec))
14 var msgpackDb, _ = storm.Open("msgpack.db", storm.Codec(msgpack.Codec))
```

**Tip:** Adding Storm tags to generated Protobuf files can be tricky. A good solution is to use this tool to inject the tags during the compilation.

**Use existing Bolt connection** You can use an existing connection and pass it to Storm

```
1 bDB, _ := bolt.Open(filepath.Join(dir, "bolt.db"), 0600, &bolt.Options{
2     Timeout: 10 * time.Second})
3 db := storm.Open("my.db", storm.UseDB(bDB))
```

**Batch mode** Batch mode can be enabled to speed up concurrent writes (see Batch read-write transactions)

```
1 db := storm.Open("my.db", storm.Batch())
```

## Nodes and nested buckets

Storm takes advantage of BoltDB nested buckets feature by using `storm.Node`. A `storm.Node` is the underlying object used by `storm.DB` to manipulate a bucket. To create a nested bucket and use

---

the same API as `storm.DB`, you can use the `DB.From` method.

```
1 repo := db.From("repo")
2
3 err := repo.Save(&Issue{
4     Title: "I want more features",
5     Author: user.ID,
6 })
7
8 err = repo.Save(newRelease("0.10"))
9
10 var issues []Issue
11 err = repo.Find("Author", user.ID, &issues)
12
13 var release Release
14 err = repo.One("Tag", "0.10", &release)
```

You can also chain the nodes to create a hierarchy

```
1 chars := db.From("characters")
2 heroes := chars.From("heroes")
3 enemies := chars.From("enemies")
4
5 items := db.From("items")
6 potions := items.From("consumables").From("medicine").From("potions")
```

You can even pass the entire hierarchy as arguments to `From`:

```
1 privateNotes := db.From("notes", "private")
2 workNotes := db.From("notes", "work")
```

## Node options

A Node can also be configured. Activating an option on a Node creates a copy, so a Node is always thread-safe.

```
1 n := db.From("my-node")
```

Give a bolt.Tx transaction to the Node

```
1 n = n.WithTransaction(tx)
```

Enable batch mode

```
1 n = n.WithBatch(true)
```

Use a Codec

---

---

```
1 n = n.WithCodec(gob.Codec)
```

## Simple Key/Value store

Storm can be used as a simple, robust, key/value store that can store anything. The key and the value can be of any type as long as the key is not a zero value.

Saving data :

```
1 db.Set("logs", time.Now(), "I'm eating my breakfast man")
2 db.Set("sessions", bson.NewObjectId(), &someUser)
3 db.Set("weird storage", "754-3010", map[string]interface{}{
4     "hair": "blonde",
5     "likes": []string{"cheese", "star wars"},
6 })
```

Fetching data :

```
1 user := User{}
2 db.Get("sessions", someObjectId, &user)
3
4 var details map[string]interface{}
5 db.Get("weird storage", "754-3010", &details)
6
7 db.Get("sessions", someObjectId, &details)
```

Deleting data :

```
1 db.Delete("sessions", someObjectId)
2 db.Delete("weird storage", "754-3010")
```

You can find other useful methods in the documentation.

## BoltDB

BoltDB is still easily accessible and can be used as usual

```
1 db.Bolt.View(func(tx *bolt.Tx) error {
2     bucket := tx.Bucket([]byte("my bucket"))
3     val := bucket.Get([]byte("any id"))
4     fmt.Println(string(val))
5     return nil
6 })
```

A transaction can be also be passed to Storm

---

```
1 db.Bolt.Update(func(tx *bolt.Tx) error {  
2     ...  
3     dbx := db.WithTransaction(tx)  
4     err = dbx.Save(&user)  
5     ...  
6     return nil  
7 })
```

## License

MIT

## Credits

- Asdine El Hrychy
- Bjørn Erik Pedersen