2021-11-21

Leveraging RocksDB with F#

Read Time: 14 minutes

Sometimes an application has a need for a local key/value store. In these scenarios, there are several options, including RocksDB. Today's exploration will be to dig into using RocksDB with F#.

If you're unfamilar with RocksDB, it is a local key/value store that you can embed in your application. I've found it to be a valuable addition to the application toolbox. For anyone following along, The below examples use .NET version 5 and the RocksDB wrapper library RocksDbSharp version 6.2.2. I've also included some simple setup and helper functions that are used in the later examples.

```
$ dotnet add package RocksDbNative --version 6.2.2
$ dotnet add package RocksDbSharp --version 6.2.2
```

```
open RocksDbSharp

/// Convert a string to a byte array
let inline toBytes (s: string) = Encoding.UTF8.GetBytes s

/// Convert a byte array to a string
let inline fromBytes (b: byte[]) = Encoding.UTF8.GetString b
```

The obvious place to get started is with some simple saving and retrieval of key/value pairs. Before jumping right in, it is useful to know that RocksDB stores keys and values as byte arrays. This provides a good deal of flexibility, but it puts the responsibility on the developer to determine the best serialization method for object storage. Depending on the data being stored, this can be an extra step to worry about, but I like the power it provides with a raw interface. To this end, RocksDBSharp supports direct interactions using byte array keys and values. For convenience is also supports the common case of accepting strings as keys and values, converting them to byte arrays under the covers. For the following example, the scenario is storing multiple worker states in the key/value store. The first thing to do is open the database. In this particular case, I’ll also create the database if it doesn’t exist. The library supports many of the standard RocksDB database configuration options. Once the database is open, I can start to do something useful. Data is added using `Put`, retrieved using `Get`, and deleted using `Remove`. It also provides a handy `MultiGet` for retrieving multiple values into a collection.

```fsharp
let dbPath = "/var/data/worker-data"
let options = DbOptions().SetCreateIfMissing(true)
use db = RocksDb.Open(options, dbPath)

// Save worker states to database
db.Put("worker1", "34,18,72")
db.Put("worker2", "1,42.2,15.4")
db.Put("worker3", "9.8,13.5,3.8")

// Get worker states from database
printfn "worker 1: %s" (db.Get("worker1"))
printfn "worker 2: %s" (db.Get("worker2"))
printfn "worker 3: %s" (db.Get("worker3"))

// Delete a worker state
db.Remove("worker2")

// Byte array example
let worker4Name = toBytes "worker4"
let worker4State = [| byte 26; byte 27; byte 28 |]
db.Put(worker4Name, worker4State)

printfn "worker 4: %A" (db.Get(worker4Name))

// MultiGet
let workerStates = db.MultiGet([| "worker1"; "worker2"; "worker3" |])

workerStates
```

```
|> Seq.iter (fun worker -> printfn $"{worker.Key} {worker.Value}")

for worker in workerStates do
  printfn $"{worker.Key} {worker.Value}
```

Anyone who has worked with a key/value store recognizes an inherent challenge of key organization. A lot can be done using naming conventions, but sometimes there is a need for better segmentation. Although separate databases are an option, RocksDB has a nicer option. It supports column families. Column families are a way to group together related data into its own structure within the same database. By specifying the Column Family when doing gets/puts, the data is segmented appropriately. In the previous example I was storing just worker states. Assuming I need to support different types of data, it potentially makes sense to segment worker states from user session data. Obviously proper naming conventions for keys can provide simple groupings, but column families bring a more proper segmentation of data. It should be noted, this isn't a security boundary, but a structural one to assist with data interactions.

Looking at the example below, there are a couple key parts. First is that the database must be opened with the available column family definitions. More specifically, this must include the definitions of all column families in the database. In this case, I'm defining two column families: one for worker states, and one for user session data. The second is that `Get` and `Put` must specify the column family where the data is located. Beyond that, the interactions are similar to the previous example.

```
let dbPath = "/var/data/state-data"
let options =
  DbOptions()
    .SetCreateIfMissing(true)
    .SetCreateMissingColumnFamilies(true)

// Define column families
let columnFamilies = ColumnFamilies()
columnFamilies.Add(ColumnFamilies.Descriptor("workers", ColumnFamilyOptions()))
columnFamilies.Add(ColumnFamilies.Descriptor("sessions", ColumnFamilyOptions()))

use db = RocksDb.Open(options, dbPath, columnFamilies)

// Define column family references
let workersFamily = db.GetColumnFamily("workers")
let sessionsFamily = db.GetColumnFamily("sessions")

```

```
db.Put("worker5", "12,34,56", cf = workersFamily)
db.Put("session100", "3000,4021", cf = sessionsFamily)

printfn "worker5: %s" (db.Get("worker5", cf = workersFamily))
printfn "session100: %s" (db.Get("session100", cf = sessionsFamily))
```

RocksDB isn't limited to just single key lookups. It also supports iterators. Say, for example, that I want to grab a set of session data. Below is an example of how to do that. To start out, I create a fake set of sessions and store them in the database. This way I have something to query against. The iterator can limit by a range of keys. The Lower bound is defined by the initial `Seek()` method. The upper bound is defined by the `SetIterateUpperBound()` option defined when opening the iterator. An upper bound isn't strictly required, if not defined it reads to the end of all keys. The example below will return all key/value pairs where the key is >= `session_40` and < `session_50`.

```
// Setup db options
let options = DbOptions().SetCreateIfMissing(true)
use db = RocksDb.Open(options, dbPath)

// Add session data
let rand = Random()
for _ in [1..1000] do
  let sessionId = $"session_%02d{rand.Next(100)}"
  let sessionData = Array.init 5 (fun _ -> rand.Next(100).ToString()) |> String.concat ","
  db.Put(sessionId, sessionData)
  printfn $"added: {sessionId} {db.Get(sessionId)}"

// Iterate over a segment of the sessions
use iterator = db.NewIterator(readOptions = ReadOptions().SetIterateUpperBound("session_50"))
iterator.Seek("session_40") |> ignore
while iterator.Valid() do
  let key = iterator.Key()
  let value = iterator.Value()
  printfn $"session: {fromBytes key} = {fromBytes value}"
  iterator.Next() |> ignore
```

Sometimes it is useful to save a set of changes to the database in a single batch or to work with a set of data prior to actually writing it to the database. RocksDB provides a WriteBatch interface that permits just that. It supports the common actions like Get/Put/Remove. This allows for the ability to keep data in memory to do data manipulation while leveraging the familiar database interface. Once the new data is in the desired state, then it can be saved to the database by calling `Write`. This call is an atomic transaction for saving the data to key/value store.

```
let options = DbOptions().SetCreateIfMissing(true)
use db = RocksDb.Open(options, dbPath)

// Save worker states to batch
use writeBatch =
  (new WriteBatchWithIndex())
    .Put("worker1", "34,18,72")
    .Put("worker2", "1,42.2,15.4")
    .Put("worker3", "9.8,13.5,3.8")

// Get worker states from write batch
printfn "worker 1: %s" (writeBatch.Get("worker1"))
printfn "worker 2: %s" (writeBatch.Get("worker2"))
printfn "worker 3: %s" (writeBatch.Get("worker3"))

// Write the batch to the persistent store
db.Write(writeBatch)

// Get worker states from persistent store
printfn "worker 1: %s" (db.Get("worker1"))
printfn "worker 2: %s" (db.Get("worker2"))
printfn "worker 3: %s" (db.Get("worker3"))
```

Next I want to discuss transaction support. This is also the perfect time to address some deficiencies and features in this particular library. The RocksDbSharp library does not have a nice wrapper for every bit of RocksDB functionality, including transaction support. Nicely enough, it does provide a reasonable middle ground to fill in those gaps. It includes “Native” wrappers around the raw apis. This means even if RocksDbSharp doesn’t have a nice wrapper, it at least provides a lower level mechanism to access the underlying RocksDB APIs. This is great, and exactly what I need to get transactions working. For example purposes I’m going to keep all the `Native` calls together. In a real project, I’d put this into it’s own module/class to properly abstract the underlying apis for the rest of the application.

Since this is all lower level, it won't be as clean as the previous code, but it gets me where I need to go. The most obvious thing about the code below is all the calls using the `Native.Instance.<method>` syntax. This is the RocksDbSharp interface to the lower-level apis. Although I try to avoid them when possible, I need to use some mutable variables in order to manually cleanup objects with some `.Dispose()` calls.

Now to walk through the process. First, RocksDB uses a specific `transaction database` object. It also requires its own options object. For the particular example I needed to increase the transaction expiration timeout, the default was just too short. Your mileage may vary. The transaction object needs a write options object, so I set that up as well. I then setup a couple mutable variables so I can properly dispose of them in the `finally` block.

There are a couple things to call out for the general flow.

- Open the transaction-supporting database
  `dbTrans <- Native.Instance.rocksdb_transactiondb_open(options.Handle, transactionOptions, dbPath)`
- Begin the transaction
  `txn <- Native.Instance.rocksdb_transaction_begin(dbTrans, writeOptions, transactionOptions, nullptr)`
- Add a key/value pair to the transaction
  `Native.Instance.rocksdb_transaction_put(txn, key, unativeint key.Length, value, unativeint value.Length, ref err)`
- Rollback the transaction if necessary
  `Native.Instance.rocksdb_transaction_rollback(txn)`
- Commit the transaction
  `Native.Instance.rocksdb_transaction_commit(txn)`
- In the finally block, perform all the cleanup necessary

```
// Define worker states
let workerData =
  [
    ("worker1", [ 1; 2; 3 ])
    ("worker2", [ 10; 20; 30 ])
    ("worker3", [ 100; 200; 300 ])
  ]

// Setup db options
let options = DbOptions().SetCreateIfMissing(true)
let transactionOptions = Native.Instance.rocksdb_transactiondb_options_create()
// Increase the expiration time for the transaction to complete
```

```fsharp
Native.Instance.rocksdb_transaction_options_set_expiration(transactionOptions, int64 10000)
let writeOptions = Native.Instance.rocksdb_writeoptions_create()

// Keep everything in a try block, and cleanup in the finally block
let mutable dbTrans = IntPtr.Zero
let mutable txn = IntPtr.Zero
let nullptr = IntPtr.Zero
try
  // Open database supporting transactions
  dbTrans <- Native.Instance.rocksdb_transactiondb_open(options.Handle, transactionOptions, dbPath)

  // Begin transaction
  txn <- Native.Instance.rocksdb_transaction_begin(dbTrans, writeOptions, transactionOptions, nullptr)

  // Save new workers' states to db
  for worker in workerData do
    let workerId =
        fst worker
        |> toBytes
    let workerState =
        snd worker
        |> List.map (fun x -> x.ToString())
        |> String.concat ","
        |> toBytes

    let mutable err = IntPtr.Zero
    Native.Instance.rocksdb_transaction_put(txn, workerId, unativeint workerId.Length, workerState, unativeint workerState.Le
    if err <> IntPtr.Zero then
      // There was an error, rollback
      printfn $"Error: {err}"
      Native.Instance.rocksdb_transaction_rollback(txn)

  if someRandomError then
    // There was an error, rollback
    Native.Instance.rocksdb_transaction_rollback(txn)
```

```
      printfn "Transaction rolled back"
    else
      // Commit transaction
      Native.Instance.rocksdb_transaction_commit(txn)
      printfn "Transaction complete"
  finally
    // Cleanup everything created using native interfaces
    Native.Instance.rocksdb_transaction_destroy(txn)
    Native.Instance.rocksdb_transactiondb_close(dbTrans)
    Native.Instance.rocksdb_transactiondb_options_destroy(transactionOptions)
    Native.Instance.rocksdb_writeoptions_destroy(writeOptions)

  // Read data after transaction is complete
  use db2 = RocksDb.Open(options, dbPath)
  for worker in [ "worker1"; "worker2"; "worker3"] do
    let workerState = db2.Get(worker)
    printfn $"{worker}: {workerState}"
```

Using a database is more than just saving and retrieving data. Backups and snapshots are often something that need to be handled. RocksDB and the RocksDbSharp library provide a simple way to address these issues using checkpoints. This is a way to easily snapshot the database state for either a point-in-time reference as a full data store backup.

```
let options = DbOptions().SetCreateIfMissing(true)
use db = RocksDb.Open(options, dbPath)

// Save a checkpoint
let checkpointPath = $"{dbPath}.chk"
if IO.Directory.Exists(checkpointPath) then IO.Directory.Delete(checkpointPath, recursive = true)
db.Checkpoint().Save(checkpointPath)
```

There is a lot more of RocksDB I could cover, but the goal is to give a taste of how the RocksDBSharp library can be leveraged. Hopefully this gives you enough of a start to take your F# project further using RocksDB. Until next time, rock on.

Related Posts:

Discriminated Unions and Dapper

F# and Dapper

F# and SQLite

#Database #F# #FSharp Share

NEWER

Data in Motion - Precipitation Map

OLDER

Examining Boyer-Moore String Search with F#

TAG CLOUD

.NET Core Accord.NET Algorithms Analytics Audio Benchmarking Chiron Classification Cognitive Services Compiler Computer Vision Cordova DTW Data Database Decision Trees Deedle Delegate Detection Docker Dynamic Time Warping EEG Edges Education Elmish EmguCV Entropy F# FParsec FSAdvent FSharp Fable Faces Filters Http Images Ionide Kaggle Keyboard Legos Logging MLNet MSIL Machine Learning MathNet Messaging Mobile Mono Morse Code Mqtt Nix OpenCV Optimization Parsing Performance Presentations Propagator Racket Regression Rekognition Robotics Rust STEM Scoring Search Serialization SignalR Signals Similarity Sound Statistics Text Timeseries Tips Tools Translation Types VS Code Web Webapi Zig admin

RECENT POSTS

Dynamic Time Warping with Zig

Estimated Shortest Path with F#

Estimating Distinct Element Counts with F#

Harnessing the Anthropic API with F#

Using F# and Flurl