RocksDB Labels Store

- Implementation of `LabelsStore` using RocksDB `(key,value)` -store
- focus on simplest components of `Labels` API
  - `add(s,p,o,value)` - store
    some wildcard combniations are valid
  - `labelsForTriples(s,p,o)` - retrieve

Why another Labels Store ?

- Present one is in-memory
  - Limited by available memory
  - reload time
- Using RocksDB (On-Disk)
  - Removes size constraints
  - Automatically persisted, just re-open the previously created on restart

What is RocksDB ?

- Open-source *Key,value Store* implemented using an LSM-tree (log structured merge tree)
  - Implemented by, and used extensively by, facebook
  - github repo
- LSM-tree is a *locking light* write-optimized `(key,value)` -store
  - Fast to write to
  - Reads take a little more effort

RocksDB and LSM Trees

- Writes (`put()` or `merge()`) are simple and fast
  - Immediately written to the log
  - buffered in the current `memtable` until buffer is full
    - typically 64M
  - sorted and written as an `L0` SST file.
- Deletion requires tombstones

Compaction in LSM Trees

- RocksDB background process moving data to lower levels
  - `Ln+1` SST is created from multiple `Ln` SST files.
  - Multiple values for the same key are merged
  - Each level fans out to more SST files
  - Each SST file stores a restricted range of values

Reading LSM Trees

- Values can exist in all levels of the LSM tree
  - Top level values 'win'
- If values are written using `merge()` values existing at all levels must be merged.
- Finding and potentially merging values is a necessarily complex process
- Faster `write`, slower `read` than a BTree

Mapping Labels to RocksDB

- `SPO` , `SP*` , `S**` , `*P*` , `*` patterns are supported
  - use separate *column families* (think namespaces) in *RocksDB*
  - `SPO` , `SP*` share a column family
  - other key patterns have their column families
- Reads can be `get()` or `multiGet()`
  - `multiGet()` finds multiple keys
    - e.g. `SPO` and `SP*` together
    - uses a special value for `*` in the key

LabelsStoreRocksDB (Writing)

- `LabelsStoreRocksDB` class implements the above mapping
  - Uses the supplied `StoreFmt` to write `SPO` and `value` to `ByteBuffer`s
  - These are the `(key,value)`-pair written (`put()` or `merge()`) to RocksDB
- Writes can be simple `put()` calls or `merge()` calls
  - `merge()` simplifies read/modify/write

Transactions

- Naive `TransactionalRocksDB`
  - Writes in same transaction share a `WriteBatch`
  - The write batch is written to RocksDB as a unit on commit.
- Our test loader uses `LabelsLoadingConsumer` which maps the load of a test message (derived from a test file) to a single transaction.

Reading

- Implementation of `labelsForTriple()`
- Reads can be `get()` or `multiGet()`
    - `multiGet()` finds multiple keys
        - e.g. `SPO` and `SP*` together
        - uses a special value for `*` in the key
- We do not scan/iterate, so don't implement a custom comparator on any of the column families.

Store Format

- RocksDB Labels store formatter object
- Abstract mapping from `SPO` to data in Rocks
  - `StoreFmtbyString`
    - Store nodes directly
    - as `(type,string content)`
  - `StoreFmtById`
    - Store nodes indirectly
    - Look node up by value in `NodeTable`
      - store the resultant 64-bit id in RocksDB

ID Format

- If the key is a triple (`SPO` and `SP*` column family)
  - 3 nodes formatted one after the other
- If the key is a single node, see node formatting
  - `S**` , `*P*` , `***` column families
- Enforce little-endian formatting of numbers (using `nio` buffers)
  - it is an on-disk-structure
  - most machines are little-endian
  - so generally most efficient

ID Format a node

```
byte 0 : 7 .. 4
```

- ordinal of `NodeType`

```
byte 0 : 3 .. 0
```

- ordinal of `IntSize` of the node id
- that is all, if the node being encoded is a `Node.Any`
- otherwise, byte 1..n as many bytes of integer as are encoded in the IntSize

```
bytes 1..1,2,4,8
```

for `IntSize.OneByte`, `IntSize.TwoBytes`, `IntSize.FourBytes`, `IntSize.EightBytes` respectively

- a URI node with a `long` node-id takes up 9 bytes, with an `int` node-id takes up 5 bytes.

String Format (SPO,SP* triples)

- Simple **reference implementation**

```
byte 0 : NodeType ordinal of S
byte 1..4 : size of S string encoding
```

```
byte 5 : NodeType ordinal of P
byte 6..9 : size of P string encoding
```

```
byte 10 : NodeType ordinal of P
byte 11..14 : size of P string encoding
```

```
S,P,O string encodings as indicated by sizes above
```

String Format (single node)

- `S**` , `*P*` , `***` column families

```
byte 0 : NodeType ordinal
byte 1..4 : size of node string encoding
```

```
node string encoding as indicated by size
```

Testing / Simulated Workloads

- `BulkDirectory` test class
  - subclasses for
    - current labels store
    - RocksDB labels store with strings
    - RocksDB labels store with a naive test node table
    - RocksDB labels store with a trie-based test node table
  - `starWars()` runs by default as part of the test suite, and loads the eponymous test files as part of UT
    - subclassed

Bigger/Biggest Workloads

- `biggerFiles()` and `biggestFiles()` are marked as `@Disabled`
  - enable and run manually, -D where to find the files

```
mvn test -Dtest=BulkDirectoryRocksDBTestsByIdTrie#biggestFilesReadLoad
-Dabac.labelstore.biggestfiles=/Users/alan/Downloads/biggest_data_files
```

- these take minutes or hours respectively

Read Workloads

- `starWarsReadLoad()`, `biggerFilesReadLoad()`, `biggestFilesReadLoad()`
  - variants on the load tests which
    a. load the labels store
    b. Repeatedly query a subset (10%, 1%, per the test) of the labels

Code

- In the `rdf-abac-core` repo, currently the `rocksdb-store-m2` branch
- In `src/main/java` package `io.telicent.jena.abac.labels`
  - `LabelsStoreRocksDB` class
  - `StoreFmt` interface and its subclasses
    - `StoreFmtById` and
    - `StoreFmtByString`

Test Code

- in `src/test/java` package `io.telicent.jena.abac`
  - `BulkDirectory` bulk loading tests
- in package `io.telicent.jena.abac.test`
  - `NaiveNodeTable`
  - `TrieNodeTable`
  - other helpers
  - `TestStoreFmt` and subclasses