Rogue Wave SOFTWARE

Accelerating Great Code

USING REDIS WITH SOURCEPRO DB

INTRODUCTION TO REDIS

Redis is a NoSQL database specialized as a key-value, in-memory data structure store. Redis is configured to persist by dumping data to disk at specified intervals, or disable persistence for an in-memory only database. Each database in the Redis server stores a collection of key-value pairs. Redis supports strings, hashes, lists, sets, and sorted sets, among other data types[1]. The stored keys and values are binary-safe, enabling the storage of large binary objects[2].

Redis cluster enables Redis as a distributed database system by automatically sharding data across multiple Redis nodes[3], reducing contention.

Redis also provides data replication and high availability via Redis Sentinel. Sentinel checks to ensure that all master and slave instances are working, sends related notifications, and initiates a failover process should the master instance fail, in which case a slave is promoted as master[4][5].

USING REDIS WITH SOURCEPRO DB

Unlike an RDBMS in which a database has multiple tables containing data in rows and columns, each database in Redis stores only key-value pairs. Also, the type of stored data can differ, making Redis quite a different data structure than that contained in an RDBMS. Because of these fundamental design differences, Redis complements an RDBMS by storing large amounts of data that do not require transactional ACID properties, leaving concise but transaction-sensitive data to an RDBMS. Using these in combination leverages the fast data storage and access of Redis while incorporating the transaction control of an RDBMS.

Installing and configuring Redis server and client

Redis is officially supported only on Unix platforms; however, Windows binaries are available from the Microsoft Open Tech Group.

- For Unix: Download the Redis server and command line tool at http://redis.io/download. Follow the instructions in the archive's README file to build it.
- For Windows: Download the installer from https://github.com/MSOpenTech/redis/releases. The installation starts the server as a service and opens the port, by default 6379, in the firewall.

Redis official C client, hiredis, can be downloaded from https://github.com/redis/hiredis and can be built using the `Makefile` included.

Use case with SourcePro DB

This example splits the data stored by a job application system between Redis and a relational database. You can use any SourcePro DB compatible RDBMS. The system stores the following applicant data, grouped as a `struct`.

[1] "Introduction to Redis", redis.io, accessed May 2016.
[2] "An introduction to Redis data types and abstractions", redis.io, asscessed May 2016.
[3] "Redis cluster tutorial", redis.io, asscessed May 2016.
[4] "Replication", redis.io, asscessed May 2016.
[5] "Redis Sentinel Documentation", redis.io, asscessed May 2016.

```
struct JobAppl
{
    int applId_;                    // Application ID
    int jobId_;                     // ID of job applied for
    RWCString name_;                // Applicant name
    RWCString email_;               // Applicant email
    RWDateTime applDate_;           // Application date
    RWCString status_;              // Application status
    RWDBBlob resume_;               // Applicant resume
    RWTValSlist<RWDBBlob> notes_; // List of notes
};
```

Data requiring transactional integrity, such as application id, job id, applicant name, email, and application date, are stored in the relational database, while large data that does not require ACID transactions but would benefit from the fast data storage and retrieval —for example, resume and list of notes —are stored in Redis.

Class `JobApplManager` defines the database operations to be performed on the applicant data. The constructor accepts the database parameters to initialize the database-specific objects for both databases. The class also declares functions `addApplication`, `addNote`, `updateStatus`, and `retrieveAppl` to perform various database operations.

```
class JobApplManager
{
public:
    // Constructor taking connection parameters for both databases
    JobApplManager(const RWCString& rdbServer, const RWCString& rdbUser,
                   const RWCString& rdbPassword, const RWCString& rdbTabName,
                   const RWCString& redisHost, int redisPort, int redisDB);

    ~JobApplManager();

    // Add a new application.
    void addApplication(const JobAppl& appl);

    // Add new note to an existing application.
    void addNote(int applId, const RWDBBlob& note);

    // Update status of an existing application.
    void updateStatus(int applId, const RWCString& status);

    // Retrieve an entire application.
    JobAppl retrieveAppl(int applId);

private:
    // Relational database objects
    RWDBDatabase rdbDB_;
    RWDBConnection rdbConn_;
    RWDBTable rdbTab_;

    // Redis database objects
    redisContext* redisContext_;
};
```

Function `addApplication` adds data from a new `JobAppl` to the databases. As it inserts data in both databases, it uses transactions to revert data insertion in the relational database if data insertion in Redis fails. Please note that error handling is not included.

```
void
JobApplManager::addApplication(const JobAppl& appl)
{
    // Insert data in relational database
    RWDBInserter rdbIns = rdbTab_.inserter();
    rdbIns << appl.applId_ << appl.jobId_ << appl.name_ << appl.email_
           << appl.applDate_ << appl.status_;

    // Use transaction to ensure that either both RDBMS and Redis data is
    // inserted or none.
    rdbConn_.beginTransaction();
    if (!rdbIns.execute(rdbConn_).isValid()) {
        rdbConn_.rollbackTransaction();
        return;
    }roll

    // Insert resume in Redis as a document to key "<applID>:resume"
    char buff[32];
    sprintf(buff, "SET %d:resume %%b", appl.applId_);
    freeReplyObject(
        redisCommand(redisContext_, buff,
                     appl.resume_.data(), appl.resume_.length()));
    // In case of errors in Redis, rollback RDBMS transaction.

    // Insert notes in Redis as a list of documents to key "<applID>:notes"
    sprintf(buff, "RPUSH %d:notes %%b", appl.applId_);
    RWTValSlist<RWDBBlob>::const_iterator iter;
    for (iter = appl.notes_.cbegin(); iter != appl.notes_.cend(); ++iter) {
        freeReplyObject(
            redisCommand(redisContext_, buff,
                         (*iter).data(), (*iter).length()));
        // In case of errors in Redis, rollback RDBMS transaction.
    }

    rdbConn_.commitTransaction();
}
```

Function `addNote` adds a note to the list of notes for an application. Function `updateStatus` updates the status of the application. Each of these functions interfaces with a single database and hence does not need to interact with the other database.

```
void
JobApplManager::addNote(int applId, const RWDBBlob& note)
{
    // Add note in Redis by appending it to the list of notes to key
    // "<applID>:notes"
    char buff[32];
```

```
    sprintf(buff, "RPUSH %d:notes %%b", applId);
    freeReplyObject(redisCommand(redisContext_, buff,
                                 note.data(), note.length()));
}

void
JobApplManager::updateStatus(int applId, const RWCString& status)
{
    // Update status in relational database.
    RWDBUpdater rdbUpd = rdbTab_.updater();
    rdbUpd << rdbTab_["status"].assign(status);
    rdbUpd.where(rdbTab_["appl_id"] == applId);

    rdbUpd.execute(rdbConn_);
}
```

Function `retrieveAppl` retrieves application data from both the databases in a `JobAppl`.

```
JobAppl
JobApplManager::retrieveAppl(int applId)
{
    JobAppl appl;

    // Fetch data from RDBMS.
    RWDBSelector rdbSel = rdbDB_.selector();
    rdbSel << rdbTab_;
    rdbSel.where(rdbTab_["appl_id"] == applId);

    RWDBReader rdbRdr = rdbSel.execute(rdbConn_).table().reader(rdbConn_);
    if (rdbRdr()) {
        rdbRdr >> appl.applId_ >> appl.jobId_ >> appl.name_ >> appl.email_
               >> appl.applDate_ >> appl.status_;

        // Fetch resume from Redis
        char buff[32];
        sprintf(buff, "GET %d:resume", applId);
        RWCString str;
        redisReply* reply = (redisReply*)redisCommand(redisContext_, buff);
        appl.resume_.putBytes(reply->str, reply->len);
        freeReplyObject(reply);

        // Fetch notes from Redis
        sprintf(buff, "LRANGE %d:notes 0 -1", applId);
        RWTValSlist<RWDBBlob> list;
        reply = (redisReply*)redisCommand(redisContext_, buff);
        for (size_t i = 0; i < reply->elements; ++i) {
            RWDBBlob blob;
            blob.putBytes(reply->element[i]->str, reply->element[i]->len);
            list.append(blob);
        }
        freeReplyObject(reply);
```

```
        appl.notes_.swap(list);
    }
    return appl;
}
```

With simple routines using the hiredis C client, data from Redis can easily be integrated with data fetched from relational databases supported by SourcePro DB. Using RDBMS transactional controls, operations in both databases can be synchronized.

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times.