odbc_adapter

JUNE 09, 2017

Connecting to Snowflake with Ruby on Rails

By Kevin Deisz

odbc_adapter

At Localytics, one of the tools we use for data processing is the Snowflake data warehouse. We connect to Snowflake in a couple different ways, but our main data retrieval application is a Ruby on Rails API. To accomplish this we use a combination of unixODBC (an open-source implementation of the ODBC standard), Snowflake's ODBC driver, and our own ODBC ActiveRecord adapter for Ruby on Rails. This sequence of tools allows us to take full advantage of ActiveRecord's query generation and general ease-of-use while still enjoying all the benefits of a fully cloud-enabled data warehouse such as Snowflake.

ODBC

First, a bit of background on the ODBC standard. ODBC is a common interface through which you can connect to multiple backend databases in the same manner. In this way it enables users to write code now and maintain the ability to migrate later, while also mitigating the pain of learning each DBMS's idiosyncrasies. You connect to a data store through an ODBC adapter, which implements the ODBC interface for that specific DBMS.

For example, the following code will execute the query `SELECT id, name FROM users` on any database without you needing to make changes to the code, just by passing in a data store name (DSN) as the first command-line argument to this script.

```
run: odbc
  ./odbc $(DSN)

odbc:
  gcc -lodbc odbc.c -o odbc
```

Makefile hosted with ❤ by GitHub view raw

```
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>
```

odbc_adapter

```
int main (int argc, char **argv) {
  SQLHENV henv   = SQL_NULL_HENV;  // Environment
  SQLHDBC hdbc   = SQL_NULL_HDBC;  // Connection handle
  SQLHSTMT hstmt = SQL_NULL_HSTMT; // Statement handle

  SQLRETURN retcode;      // Return status of a query
  SQLBIGINT userId;       // holds the ID of the user
  SQLTCHAR userName[256]; // buffer to hold name of the user

  // Establish the connection string
  char connStr[strlen(argv[1]) + 5];
  sprintf(connStr, "DSN=%s;", argv[1]);

  // Allocate and initialize the environment and connection handles
  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
  SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3,
  SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

  // Connect to data source and allocate the statement handle
  SQLDriverConnect(hdbc, NULL, (SQLCHAR *)connStr, SQL_NTS, NULL, 0, NU
  SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

  // Fetch the results of a query and bind the columns
  SQLExecDirect(hstmt, (SQLCHAR *)"SELECT id, name FROM users", SQL_NTS
  SQLBindCol(hstmt, 1, SQL_C_SBIGINT, (SQLPOINTER)&userId, sizeof(userI
  SQLBindCol(hstmt, 2, SQL_C_TCHAR, (SQLPOINTER)userName, sizeof(userNa

  // Fetch and print each row of data until SQL_NO_DATA returned.
  while (SQL_SUCCEEDED(retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT,
    printf("User %ld: %s\n", userId, userName);
  }

  // Free the allocated handles
  if (hstmt != SQL_NULL_HSTMT)
    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);

  if (hdbc != SQL_NULL_HDBC) {
    SQLDisconnect(hdbc);
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
  }

```

odbc_adapter

```
50
51    return 0;
52  }
```

odbc.c hosted with ❤ by GitHub view raw

DSNs are a string of key-value pairs representing the connection configuration. They correspond to an entry in an `odbc.ini` file that you configure. You can then reference the configured DSN using an implementation of ODBC (e.g., unixODBC) to connect to an ODBC DBMS like Snowflake. For example, in your `odbc.ini` file you might have:

```
[LocalyticsProductionSnowflake]
Driver         = SnowflakeDSIIDriver;
Locale         = en-US;
Server         = yoursnowflakeaccount.snowflakecomputing.com;
Port           = 443;
Account        = yoursnowflakeaccount;
Database       = PRODUCTION;
Schema         = PRODUCTION;
Warehouse      = QUERY_WH;
Role           = QUERY;
SSL            = on;
Query_Timeout  = 270;
uid            = ...;
pwd            = ...;
```

The configuration above operates under the assumption that you've previously installed the adapter for each type of DBMS to which you're attempting to connect.

Installation

Installing `unixODBC` is relatively straightforward on *NIX-based machines (on Windows ODBC actually ships with the OS by default). Run whichever package manager your machine uses (e.g., `brew`, `apt-get`, `yum`, etc.) to install

odbc_adapter

Fortunately Snowflake provides great documentation on how to handle the Snowflake-specific steps of getting ODBC set up, so follow those instructions as well.

Once you do, make sure to take full advantage of the `isql` utility that comes with `unixODBC`, as it can be invaluable for debugging. `isql` will drop you into an SQL terminal connected to any given DSN; for example:

```
[17:38:30] ~ $ isql LocalyticsProductionSnowflake
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
SQL> SELECT COUNT(*) FROM fact_events WHERE app_name = 'Localytics
Test';
+----------+
| COUNT(*) |
+----------+
| 226975   |
+----------+
SQLRowCount returns 1
1 rows fetched
SQL>
```

odbc_adapter

Once you're comfortably set up with `unixODBC` and Snowflake's adapter, you can configure your Ruby on Rails app to connect to Snowflake like you would any other data store. First, add the `odbc_adapter` gem to your `Gemfile` like so:

```
gem 'odbc_adapter', '~> 5.0.3'
```

odbc_adapter

major and minor version of the gem are linked to the dependent Rails version, so if your app is not yet running Rails `5.0.x`, you'll need to specify `4.2.3` or `3.2.0`). Then, edit your `config/database.yml` to specify the Snowflake connection for a given environment, like so:

```
snowflake:
  adapter: odbc
  dsn: LocalyticsProductionSnowflake
```

This tells Rails to use those connection settings when running in production mode. The final step is to register Snowflake as a valid connection option within the `odbc_adapter` gem. By default, `odbc_adapter` ships with support for `MySQL` and `PostgreSQL`. Fortunately, it also ships with the ability to register you own adapters as well. To accomplish this, add the following code to an initializer, e.g. `config/initializers/odbc.rb`:

```
1  require 'active_record/connection_adapters/odbc_adapter'
2  require 'odbc_adapter/adapters/postgresql_odbc_adapter'
3
4  ODBCAdapter.register(/snowflake/, ODBCAdapter::Adapters::PostgreSQLODBC
5    # Explicitly turning off prepared statements as they are not yet work
6    # snowflake + the ODBC ActiveRecord adapter
7    def prepared_statements
8      false
9    end
10
11   # Quoting needs to be changed for snowflake
12   def quote_column_name(name)
13     name.to_s
14   end
15
16   private
17
18   # Override dbms_type_cast to get the values encoded in UTF-8
19   def dbms_type_cast(columns, values)
20     values.each do |row|
21       row.each_index do |idx|
```

odbc_adapter

```
      end
    end
  end
end
```

odbc.rb hosted with ❤️ by GitHub view raw

This code does a couple of things. It tells the `odbc_adapter` gem that if when ODBC reports back the connected DBMS's type it matches the `/snowflake/` regex, to use the subsequent block to create a class to act as the adapter. We're then using the PostgreSQL adapter as the superclass, because the syntax is close enough so as it work. Finally, it handles the Snowflake-specific setup of turning off prepared statements, quoting column names correctly, and forcing strings to come back in UTF-8 encoding.

ActiveRecord

Once you've configured the `odbc_adapter` gem, you can take advantage of it by connecting your models to that connection. First, create a model that corresponds to a table in your Snowflake schema. For instance, in our production schema we have a table called `fact_events`. Second, call `establish_connection` to tell `ActiveRecord` to connect to the correct database configuration from `database.yml`. For example:

```
class FactEvent < ApplicationRecord
  establish_connection(:snowflake)
end
```

Note that if all of your models are going to be reading and writing from Snowflake for a given environment (development, production, etc.) then you can name the connection after the environment and the `establish_connection` call becomes unnecessary. With these models in place, you can perform any of the normal `ActiveRecord` queries.

odbc_adapter

This configuration works for us, and we've been happily running this code in production since January of 2017. That being said, there are still a couple of things that we'd like to build into our adapter to make it even better.

OUT-OF-THE-BOX SNOWFLAKE SUPPORT

Currently, every project that uses Snowflake needs the initializer mentioned above because the `odbc_adapter` gem doesn't come with Snowflake support baked in. At the moment subclassing the PostgreSQL adapter works for us, but we'd like to fully support Snowflake's driver so that we can take advantage of some of the more advanced UDF capabilities that Snowflake has to offer.

RAILS 5.1

The latest version of Rails was recently released, so in order to upgrade our applications we need to go through and ensure that our adapter works with all of the new capabilities of the latest version of ActiveRecord.

PREPARED STATEMENTS

Our adapter supports prepared statements for the PostgreSQL adapter, but it's explicitly turned off for MySQL and Snowflake. We'd like to take advantage of caching prepared statements to cut down on memory allocations and generally improve performance by enabling it for these two adapters.

Wrapping up

Snowflake is a great option for a cloud-based data warehouse, and solved a lot of problems that we've had with previous solutions to the problem of storing massive amounts of data. By being ODBC compliant, it enables us to connect using all of our favorite tools with minimal setup. If you also would like to use Snowflake with Ruby on Rails, feel free to install our `odbc_adapter` gem and give it a shot. When you do please share your experience, approach, and any feedback in a gist, on a blog, or in the comments.

odbc_adapter

Serverless Slackbots Powered by AWS

4 years ago • 4 comments

Today Localytics is open sourcing two tools that help you quickly scaffold …

Performance in Big Data Land: Efficiency

4 years ago • 1 comment

Let's continue to explore the idea that "every CPU cycle matters" when dealing …

Meet Our Engine Brian Zeligson

5 years ago • 1 comm

![](/content/images/20 BrianZ-localytics-1.j

4 Comments LocalyticsEng Privacy Policy Erianz

Recommend 2 Tweet Share Sort by Best

Join the discussion…

Steve P • 3 years ago

Great write up, thanks for the details. Are you guys using Snowflake's variant type at all? How have you integrated its more open structure with ActiveRecord?

Reply • Share ›

Kevin Deisz ➚ **Steve P** • 3 years ago

Like Michal said, we don't integrate it too tightly with ActiveRecord. That being said, it would be relatively trivial to create a custom ActiveRecord type that would handle this logic in a centralized place. Here's a good blog post to get you started: https://ideamotive.co/blog/...

1 Reply • Share ›

Steve P ➚ **Kevin Deisz** • 3 years ago

Thanks for the info!

Reply • Share ›

Michal Klos ➚ **Steve P** • 3 years ago

Yes we do use variant, but we don't try to handle in a fancy manner in ActiveRecord but just return it as JSON:

```
module Dimensions
class JsonDimension < Abstract::Dimension
def value_to_db(v)
v.nil? ? nil : "PARSE_JSON(#{v})"
```

odbc_adapter

Product

Features

Showcase

Demo

Documentation

Getting Started

Using Localytics

Developer Docs

FAQ

Resources

Overview

Case Studies

Webinars

eBooks

Videos

Blog

Company

About Us

Careers

Partners

Press

Awards

Contact Us