#include "SQLQuerier.h"
#include "TableNames.h"

SQLQuerier::SQLQuerier(std::string a, std::string r, std::string i, bool ram) {
ram_tablespace = ram;
announcements_table = a;
results_table = r;
inverse_results_table = i;
//default host and port numbers
//strings for connection arg
host = "127.0.0.1";
port = "5432";

```
read_config();
open_connection();
```

}

SQLQuerier::~SQLQuerier(){
C->disconnect();
delete C;
}

void SQLQuerier::open_connection(){
std::ostringstream stream;
stream << "dbname = " << db_name;
stream << " user = " << user;
stream << " password = " << pass;
stream << " hostaddr = " << host;
stream << " port = " << port;

```
try {
   pqxx::connection *conn = new pqxx::connection(stream.str());
   if (conn->is_open()) {
      std::cout << "Connected to database : " << db_name <<std::endl;
      C = conn;
   }
   else{
      std::cout << "Failed to connect to database : " << db_name <<std::endl;
      return;
   }
} catch (const std::exception &e){
   std::cerr << e.what() << std::endl;
```

```
    }
    return;
```

}

void SQLQuerier::close_connection(){
C->disconnect();
}

pqxx::result SQLQuerier::execute(std::string sql, bool insert){
pqxx::result R;
if(insert){
//TODO maybe make one work object once on construction
//work object may be same as nontransaction with more functionality

```
    try{
        pqxx::work txn(*C);
        R = txn.exec(sql);
        txn.commit();
        return R;
    }
    catch(const std::exception &e){
        std::cerr << e.what() <<std::endl;
    }
}
else{
    try{
        pqxx::nontransaction N(*C);
        pqxx::result R( N.exec(sql));

        return R;

    }
    catch(const std::exception &e){
        std::cerr << e.what() <<std::endl;
    }
}
return R;
```

}
//TODO maybe rename/overload this for selection options
pqxx::result SQLQuerier::select_from_table(std::string table_name, int limit){
std::string sql = "SELECT * FROM " + table_name;

```
if(limit){
```

```
        sql += " LIMIT " + std::to_string(limit);
    }
    return execute(sql);
```

}

/**
*/
pqxx::result SQLQuerier::select_ann_records(std::string table_name, std::string prefix, int limit,
uint64_t offset){
std::string sql = "SELECT host(prefix), netmask(prefix), as_path, origin FROM ";
if (!table_name.empty()) {
sql += table_name;
} else {
sql += announcements_table;
}
sql += " ORDER BY prefix DESC";
if (!prefix.empty()){
sql += (" WHERE prefix = "+ std::string("'") + prefix + std::string("'"));
}
if (limit){
sql += " LIMIT " + std::to_string(limit);
}
if (offset) {
sql += " OFFSET " + std::to_string(offset);
}
sql += ";";
return execute(sql);
}

/**
*/
pqxx::result SQLQuerier::select_ann_records(std::string table_name, std::vector prefixes, int limit)
{
// std::cerr << "Selecting announcement records..."<< std::endl;
std::string sql = "SELECT host(prefix), netmask(prefix), as_path, origin FROM " + table_name;
sql += " WHERE prefix IN (";
int comma_limit = prefixes.size();
int i = 0;
for (const std::string prefix : prefixes){
i++;
sql+= "'" + prefix + "'";

```
if(i < comma_limit)
sql+= ",";
}
//add when using old data format
// sql += ") AND element_type = 'A'";
sql += ")";
if(limit){
sql += " LIMIT " + std::to_string(limit);
}

// std::cerr << sql << std::endl;
return execute(sql);
}

pqxx::result SQLQuerier::select_distinct_prefixes_from_table(std::string table_name){
std::string sql = "SELECT DISTINCT prefix, family(prefix) FROM " + table_name;
return execute(sql);
}

pqxx::result SQLQuerier::select_roa_prefixes(std::string table_name, int ip_family){
std::string sql = "SELECT DISTINCT prefix, family(prefix) FROM " + table_name;
if(ip_family == IPV4)
sql += " WHERE family(prefix) = 4";
else if(ip_family == IPV6)
sql += " WHERE family(prefix) = 4";
return execute(sql);
}

pqxx::result SQLQuerier::select_as_types(std::string table_name) {
std::string sql = "SELECT asn, as_type FROM " + table_name;
return execute(sql);
}

//TODO add return type
void SQLQuerier::check_for_relationship_changes(std::string peers_table_1,
std::string customer_provider_pairs_table_1,
std::string peers_table_2,
std::string customer_provider_paris_table_2){
```

// std::vector<std::vector<uint

```
    std::string sql = "SELECT * FROM " + peers_table_1;
    //peers_1_result = execute(sql);

    sql = "SELECT * FROM " + peers_table_2;
    //peers_2_results = execute(sql);
```

}

// this should use the STUBS_TABLE macro
void SQLQuerier::clear_stubs_from_db(){
std::string sql = "DELETE FROM stubs";
execute(sql);
}

/*

- Generate an index on the results table.
  */
  void SQLQuerier::create_results_index() {
  // postgres version must support this
  std::string sql = std::string("CREATE INDEX ON " + results_table + " USING GIST(prefix inet_ops, origin)");
  std::cout << "Generating index on results..." << std::endl;
  execute(sql, false);
  }

/*

- Takes a .csv filename and bulk copies all elements to the stubs table.
  */
  void SQLQuerier::clear_non_stubs_from_db(){
  std::string sql = "DELETE FROM non_stubs";
  execute(sql);
  }

void SQLQuerier::copy_stubs_to_db(std::string file_name){
std::string sql = "COPY " STUBS_TABLE "(stub_asn,parent_asn) FROM '" +
file_name + "' WITH (FORMAT csv)";
execute(sql);
}

/*

- Takes a .csv filename and bulk copies all elements to the supernodes table.
  */
  void SQLQuerier::copy_supernodes_to_db(std::string file_name){
  std::string sql = "COPY " SUPERNODES_TABLE "(supernode_asn,supernode_lowest_asn)
  FROM '" +
  file_name + "' WITH (FORMAT csv)";
  execute(sql);
  }

```
void SQLQuerier::copy_non_stubs_to_db(std::string file_name){
std::string sql = "COPY " NON_STUBS_TABLE "(non_stub_asn) FROM '" +
file_name + "' WITH (FORMAT csv)";
execute(sql);
}

/*
```

- Takes a .csv filename and bulk copies all elements to the results table.
  */
  void SQLQuerier::copy_results_to_db(std::string file_name){
  std::string sql = std::string("COPY " + results_table + "(asn, prefix, origin,
  received_from_asn)") +
  "FROM '" + file_name + "' WITH (FORMAT csv)";
  execute(sql);
  }

```
/*
```

- Instantiates a new, empty supernodes table in the database, if it doesn't exist.
  */
  void SQLQuerier::create_supernodes_tbl(){
  std::string sql = std::string("CREATE TABLE IF NOT EXISTS " SUPERNODES_TABLE "
  (supernode_asn BIGSERIAL PRIMARY KEY, supernode_lowest_asn bigint)");
  std::cout << "Creating supernodes table..." << std::endl;
  execute(sql, false);
  }

```
void SQLQuerier::create_rovpp_blacklist_tbl() {
// Drop the results table
std::string sql = std::string("DROP TABLE IF EXISTS " ROVPP_BLACKLIST_TABLE " ;");
std::cout << "Dropping rovpp_blacklist table..." << std::endl;
execute(sql, false);
// Create it again
```

sql = std::string("CREATE TABLE IF NOT EXISTS " ROVPP_BLACKLIST_TABLE "(rovpp_asn BIGINT, prefix CIDR, hijacked_ann_received_from_asn BIGINT)");
std::cout << "Creating rovpp_blacklist table..." << std::endl;
execute(sql, false);
}

/*

- Instantiates a new, empty stubs table in the database, if it doesn't exist.
  */
  void SQLQuerier::create_stubs_tbl(){
  std::string sql = std::string("CREATE TABLE IF NOT EXISTS " STUBS_TABLE " (stub_asn BIGSERIAL PRIMARY KEY,parent_asn bigint);");
  std::cout << "Creating stubs table..." << std::endl;
  execute(sql, false);
  }

/*

- Instantiates a new, empty non_stubs table in the database, if it doesn't exist.
  */
  void SQLQuerier::create_non_stubs_tbl(){
  std::string sql = std::string("CREATE TABLE IF NOT EXISTS " NON_STUBS_TABLE " (non_stub_asn BIGSERIAL PRIMARY KEY);");
  std::cout << "Creating non_stubs table..." << std::endl;
  execute(sql, false);
  }

/*

- Instantiates a new, empty results table in the database, dropping the old table.
  */
  void SQLQuerier::create_results_tbl(){
  // Drop the results table
  std::string sql = std::string("DROP TABLE IF EXISTS " + results_table + " ;");
  std::cout << "Dropping results table..." << std::endl;
  execute(sql, false);
  // And create it again
  sql = std::string("CREATE UNLOGGED TABLE " + results_table + " (ann_id serial PRIMARY KEY,
 asn bigint,prefix cidr, origin bigint, received_from_asn
 bigint); GRANT ALL ON TABLE " + results_table + " TO bgp_user;");
  std::cout << "Creating results table..." << std::endl;
  execute(sql, false);

```
}

/*
```

- Instantiates a new, empty inverse results table in the database, dropping the old table.
  / void SQLQuerier::create_inverse_results_tbl(){ // Drop the results table std::string sql; //std::string sql = std::string("DROP TABLE IF EXISTS " + inverse_results_table + " ;"); //std::cout << "Dropping inverse results table…" << std::endl; std::cout << "Not dropping inverse results table..." << std::endl;
  //execute(sql, false);
  // And create it again
  sql = std::string("CREATE UNLOGGED TABLE IF NOT EXISTS ") + inverse_results_table + "(asn bigint,prefix cidr, origin bigint) ";
  sql += (ram_tablespace ? "TABLESPACE ram;" : ";");
  sql += "GRANT ALL ON TABLE " + inverse_results_table + " TO bgp_user;";
  std::cout << "Creating inverse results table..." << std::endl;
  execute(sql, false);
  }
  void SQLQuerier::copy_inverse_results_to_db(std::string file_name){
  std::string sql = std::string("COPY " + inverse_results_table + "(asn, prefix, origin)") + "FROM '" + file_name + "' WITH (FORMAT csv)";
  execute(sql);
  }

```
//Reads credentials/connection info from .conf file
void SQLQuerier::read_config(){
using namespace std;
string file_location = "./bgp.conf";
ifstream cFile(file_location);
if (cFile.is_open()){
//map config variables to settings in file
map config;
string line;
while(getline(cFile, line)){
//remove whitespace and check to ignore line
line.erase(remove_if(line.begin(),line.end(),::isspace), line.end());
if(line.empty() || line[0] == '#' || line[0] == '['){
continue;
}
auto delim_index = line.find("=");
std::string var_name = line.substr(0,delim_index);
std::string value = line.substr(delim_index+1);
config.insert(std::pair(var_name, value));
```

}

```
    //Add additional config options to this
    for (auto const& setting : config){
        if(setting.first == "user")
            user = setting.second;
        else if(setting.first == "password")
            pass = setting.second;
        else if(setting.first == "database")
            db_name = setting.second;
        else if(setting.first == "host"){
            if(setting.second == "localhost")
                host = "127.0.0.1";
            else
                host = setting.second;
        }
        else if(setting.first == "port")
            port = setting.second;
        else{
            std::cerr << "Setting \"" <<
                    setting.first << "\" undefined." << std::endl;
        }
    }
}
else{
    std::cerr << "Error loading config file \"" << file_location << "\"" << std::endl;
}
```

}