#include <SQLiteCpp/Database.h> #include #include #include #include #include #include #include #include #include #include #include #include “jtb/jtbstr.h” #include “jtb/jtbvec.h” #include <SQLiteCpp/SQLiteCpp.h>

enum { THREADLIMIT = 30000 };

namespace fs = std::filesystem; using st = std::vector::size_type;

struct Film { JTB::Str tconst = “N\a”; JTB::Str title = “N\a”; JTB::Str origtitle = “N\a”; JTB::Str year = “N\a”; JTB::Str length = “N\a”; JTB::Str genre = “N\a”; JTB::Str rating = “0”; JTB::Str numrates = “0”; JTB::Str lang = “N\a”; JTB::Str directors = ““; JTB::Str writers =””; JTB::Str actors =””; };

struct Ratepair { std::string rating; std::string numrates; };

template int icast(T thing) { return static_cast(thing); }

const int nofdsets = 5;

void loadBasics(SQLite::Database& db, std::ifstream& filestream) { /* throwing out the first line */ JTB::Str buf {}; buf.absorbLine(filestream).clear();

```
/* buffer variables */
enum Cols { TCONST, TYPE, PRIMARY, ORIGINAL, ISADULT, STARTYEAR, ENDYEAR, RUNTIME, GENRES };

SQLite::Statement select { db, "SELECT * FROM Films WHERE tconst = ?" };
SQLite::Statement film_insert { db, "INSERT INTO Films (tconst, title, originalTitle) VALUES
SQLite::Statement year_insert { db, "INSERT INTO Years (tconst, year) VALUES (?, ?)" };
SQLite::Statement runtime_insert { db, "INSERT INTO Runtimes (tconst, runtimeInMin) VALUES (
SQLite::Statement genre_insert { db, "INSERT INTO Genres (tconst, genre) VALUES (?, ?)" };

/* processing the data */
while ( filestream.good() && !buf.clear().absorbLine(filestream).isEmpty() ) {
JTB::Vec<JTB::Str> rowslicer = buf.split("\t");
try {
    if (rowslicer[TYPE] == "movie"
    && rowslicer[ISADULT] == "0"
    && rowslicer[STARTYEAR] != R"(\N)"
    && rowslicer[RUNTIME] != R"(\N)") {

    select.reset();
    select.bind(1, rowslicer.at(TCONST).c_str());
    select.executeStep();
    if (!select.hasRow()) {
        SQLite::Transaction tr {db};
        film_insert.bind(1, rowslicer.at(TCONST).c_str());
        film_insert.bind(2, rowslicer.at(PRIMARY).c_str());
        film_insert.bind(3, rowslicer.at(ORIGINAL).c_str());
        film_insert.exec();
        film_insert.reset();
```

```
        year_insert.bind(1, rowslicer.at(TCONST).c_str());
        year_insert.bind(2, rowslicer.at(STARTYEAR).c_str());
        year_insert.exec();
        year_insert.reset();

        runtime_insert.bind(1, rowslicer.at(TCONST).c_str());
        runtime_insert.bind(2, std::stoi(rowslicer.at(RUNTIME).c_str()));
        runtime_insert.exec();
        runtime_insert.reset();

        JTB::Vec<JTB::Str> genres = rowslicer.at(GENRES).split(",");
        genres.forEach([&](JTB::Str genre) {
        genre_insert.bind(1, rowslicer.at(TCONST).c_str());
        genre_insert.bind(2, genre.c_str());
        genre_insert.exec();
        genre_insert.reset();
        });
        tr.commit();
    }
    }
} catch (std::exception& e) {
    std::cerr << "Error reading basics: " << e.what() << '\n';
    std::cerr << "Rowslicer: " << rowslicer << '\n';
    exit(1);
}
}
std::cout << "Done reading the basics!" << '\n';
```

}

void loadRatings(SQLite::Database& db, std::ifstream& filestream) { /* buffers */ JTB::Vec rowslicer {}; JTB::Str buf {};

```
/* throwing out first line */
buf.absorbLine(filestream).clear();
enum Cols { TCONST, RATING, NUMRATES };

/* reading ratings into fdb */
while ( filestream.good() && !buf.clear().absorbLine(filestream).isEmpty() ) {
    rowslicer = buf.split("\t");
try {
} catch (std::exception& e) {
    std::cerr << "Problem inserting ratings" << '\n';
    std::cerr << "Error: " << e.what() << '\n';
    exit(1);
}
```

```
}

std::cout << "Done reading ratings!" << '\n';

}
```

void loadLanguage(SQLite::Database& db, std::ifstream& filestream) { /* buffers */ JTB::Vec rowslicer {}; JTB::Str buf {};

```
enum Cols { TCONST, LANG };

/* reading langs into buffer */
while ( filestream.good() && !buf.clear().absorbLine(filestream).isEmpty() ) {
    rowslicer = buf.split("\t");
if (rowslicer.size() < 2) continue;
try {
} catch (std::out_of_range& e) {
    std::cerr << "Problem inserting languages" << '\n';
    std::cerr << "Error: " << e.what() << '\n';
    std::cerr << "Rowslicer: " << rowslicer << '\n';
    exit(1);
}
}

std::cout << "Done reading languages!" << '\n';

};
```

void loadPrincipals(SQLite::Database& db, std::ifstream& principals_stream, std::ifstream& names_stream) { /* buffers */ JTB::Vec rowslicer {}; JTB::Str buf {};

```
/* buffer for names */
std::map<JTB::Str, JTB::Str> namebuf {};

enum class Names { NCONST, NAME };
enum class Principles { TCONST, ORDERING, NCONST, CATEGORY, JOB, CHARACTERS };


/* reading names into buffer */
while (names_stream.good() && !buf.clear().absorbLine(names_stream).isEmpty()) {
    rowslicer = buf.split("\t");
if (rowslicer.size() < 2) continue;
namebuf[rowslicer[static_cast<int>(Names::NCONST)]] = rowslicer[static_cast<int>(Names::NAME
}
std::cout << "Done reading names!" << '\n';

/* filling in principals */
buf.absorbLine(principals_stream);
```

```
while (principals_stream.good() && !buf.clear().absorbLine(principals_stream).isEmpty()) {
rowslicer = buf.split("\t");
if (rowslicer.size() < 4) continue;
/* if (!film_hashmap.contains(rowslicer[icast(Principles::TCONST)])) continue; */
/* if (rowslicer.at(icast(Principles::CATEGORY)).startsWith("a")) { */
/*     film_hashmap[rowslicer.at(icast(Principles::TCONST))].actors */
/*  = film_hashmap[rowslicer.at(icast(Principles::TCONST))].actors + namebuf[rowslicer.at(ic
/* } */
/* else if (rowslicer.at(icast(Principles::CATEGORY)).startsWith("d")) { */
/*     film_hashmap[rowslicer.at(icast(Principles::TCONST))].directors */
/*  = film_hashmap[rowslicer.at(icast(Principles::TCONST))].directors + namebuf[rowslicer.at
/* } */
/* else if (rowslicer.at(icast(Principles::CATEGORY)).startsWith("w")) { */
/*     film_hashmap[rowslicer.at(icast(Principles::TCONST))].writers */
/*  = film_hashmap[rowslicer.at(icast(Principles::TCONST))].writers + namebuf[rowslicer.at(i
/* } */
}
std::cout << "Done reading principals!" << '\n';
```

}

int main() {

```
/* reading the directory and opening the relevant files if they're found */
auto environ = std::getenv("__MOVIE_DATABASE_PATH");
std::stringstream movieDatabasePath { "" };
movieDatabasePath << (environ == nullptr ? "" : environ);

/* if the env var is not set we use current directory <= 02/24/24 12:22:49 */
if (movieDatabasePath.str().empty()) {
std::cout << "No __MOVIE_DATABASE_PATH env variable. Using current directory instead..." <<
movieDatabasePath << fs::current_path().string();
std::cout << movieDatabasePath.str() << '\n';
}

environ = std::getenv("MOVIES");
std::stringstream moviesWithPath { "" };
moviesWithPath << (environ == nullptr ? "" : environ);

/* if the env var is not set we use current directory<= 02/24/24 12:22:49 */
if (moviesWithPath.str().empty()) {
std::cout << "No MOVIES env variable. Creating file in current directory instead..." << '\n'
moviesWithPath << fs::current_path().string() << "movies.tsv";
std::cout << moviesWithPath.str() << '\n';
}

std::ifstream lang_stream {};
```

```
std::ifstream basics_stream {};
std::ifstream ratings_stream {};
std::ifstream principals_stream {};
std::ifstream name_basics_stream {};
try {
lang_stream.open( movieDatabasePath.str() + "/lang.tsv" );
basics_stream.open( movieDatabasePath.str() + "/title.basics.tsv" );
ratings_stream.open( movieDatabasePath.str() + "/title.ratings.tsv" );
principals_stream.open( movieDatabasePath.str() + "/title.principals.tsv" );
name_basics_stream.open( movieDatabasePath.str() + "/name.basics.tsv" );
} catch (std::exception& e) {
std::cerr << "Problem with opening filestreams" << '\n';
std::cerr << "Error: " << e.what() << '\n';
exit(1);
}

try {
SQLite::Database db { movieDatabasePath.str() + "/moviedatabase.db", SQLite::OPEN_READWRITE
db.exec(R"(CREATE TABLE IF NOT EXISTS "Films" (
    tconst TEXT NOT NULL PRIMARY KEY,
    title TEXT NOT NULL,
    originalTitle TEXT NOT NULL))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Genres" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    genre TEXT NOT NULL))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Runtimes" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    runtimeInMin INT NOT NULL,
    UNIQUE(tconst, runtimeInMin)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Years" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    year INT NOT NULL,
    UNIQUE(tconst, year)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Names" (
    nconst TEXT NOT NULL PRIMARY KEY,
    name TEXT NOT NULL,
    UNIQUE(nconst, name)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Directors" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    nconst TEXT NOT NULL REFERENCES Names(nconst),
    UNIQUE(tconst,nconst)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Actors" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    nconst TEXT NOT NULL REFERENCES Names(nconst),
    UNIQUE(tconst,nconst)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Writers" (
```

```
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    nconst TEXT NOT NULL REFERENCES Names(nconst),
    UNIQUE(tconst,nconst)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "KnownFor" (
    tconst TEXT NOT NULL REFERENCES Films(tconst),
    nconst TEXT NOT NULL REFERENCES Names(nconst),
    UNIQUE(tconst,nconst)))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Ratings" (
    tconst TEXT NOT NULL UNIQUE REFERENCES Films(tconst),
    rating FLOAT NOT NULL,
    numVotes INTEGER NOT NULL))");
db.exec(R"(CREATE TABLE IF NOT EXISTS "Languages" (
    tconst TEXT NOT NULL UNIQUE REFERENCES Films(tconst),
    lang TEXT NOT NULL))");

loadBasics(db, basics_stream);
/* loadRatings(db, ratings_stream); */
/* loadLanguage(db, lang_stream); */
/* loadPrincipals(db, principals_stream, name_basics_stream); */
} catch (std::exception& e) {
std::cerr << e.what() << '\n';
    exit(1);
}


/* std::map<JTB::Str, Film> fdata {}; */


/* std::ofstream os { moviesWithPath.str() }; */

/* char tab = '\t'; */

/* for (auto const& [k, film] : fdata) { */
/* if (film.numrates != "0") { */
    /* os << film.tconst << tab << film.title << ";" << film.origtitle << tab; */
    /* os << film.year << tab << film.length << tab << film.genre << tab; */
    /* os << film.rating << tab << film.numrates << tab << film.lang << tab << film.directo
    /* os << film.actors << tab << film.writers << '\n'; */
/* } */
/* } */
std::cout << "All done!" << '\n';
}
```