توضیحات تکمیلی خط لوله

آرمان رضایی – ۹۷۲۳۰۳۴

این فایل در واقع حکم documentation برای کد نوشته شده را دارد.

مقدمه

اگر پیش از مطالعه و بررسی محتویات این پوشه به دو پوشه‌ی دیگر (Database و Warehouse) سر نزده‌اید، توصیه می‌کنم که حتما ابتدا آنان را بررسی نمایید؛ چراکه درک ساختار دیتابیس‌های مبدا و مقصد کمک شایانی به درک این خط لوله، که حکم رابط بین آن دو را دارد، خواهد نمود.

همانطور که خودتان بهتر از بنده آگاه هستید، همواره به برنامه‌نویسان توصیه می‌شود که در نوشتن برنامه‌های خود از قاعده DRY پیروی کنند. طی گشت و گذار برای یافتن یک framework مناسب برای کمک در پیاده‌سازی خط لوله، با کتابخانه‌های مختلفی برخورد کردم. برخی مانند Apache Airflow زیادی پیچیده، عده‌ای دیگر مانند Mara ناسازگار با برخی سیستم‌عامل‌ها و برخی دیگر همانند Odo و ETLAlchemy را غبار زمان پوشانده بودند. از میان این فریمورک‌ها، دو کتابخانه با نام‌های petl و pygrametl به نظرم مناسب آمدند. لذا تصمیم بر آن گرفتم که از کمک آنان استفاده کنم. در فایل requirements.txt می‌توانید کتابخانه‌های مورد نیاز برای ساختن یک Virtual Environment در پایتون را بیابید و در صورت لزوم، با دستورات زیر محیط پروژه‌ی ETL را در سیستم خود شبیه‌سازی نمایید:

```
python -m venv 'venv'
python -m pip install -r ./requirements.txt
```

کمک گرفتن از کتابخانه‌های third-party می‌تواند خوب باشد، اما نه به خوبی خود PostgreSQL. تلاش بر این است که بیشتر پردازش‌های مورد نیاز در داخل خود پایگاه داده‌ها انجام گیرند و تا حد امکان از استفاده از کدهای پایتون خودداری شود، چراکه دهه‌ها بهینه‌سازی پشت این DBMS قرار دارد و از طرف دیگر به زبان C نوشته شده است: زبانی که گاه تا صدها برابر از پایتون سریعتر عمل می‌کند. از سوی دیگر فریمورک‌های یاد شده برای ETL تنها دو یا سه سال عمر دارند و بعضا حتی وظایف خود را به درستی انجام نمی‌دهند. به همین خاطر است که triggerهای متعددی در

داخل انبار داده نوشته شده‌اند تا حالت‌های مختلف را تحت پوشش قرار دهند و بخش‌هایی از پردازش مورد نیاز در ETL را حذف می‌کنند (برای مشاهده تریگرهای نوشته شده به فایل‌های schema.sql در دو پوشه‌ی دیگر مراجعه نمایید).

شروع کار: init.sql

این فایل جهت ساخت تعدادی TEMPORARY VIEW در دیتابیس مبدا نوشته است. این viewها هم‌شکل با ساختار جداول انبار داده می‌باشند و نیاز به نوشتن JOINهای مختلف در داخل کد پایتون را حذف می‌کنند. از آنجایی که temporary هستند نیز پس از پایان یافتن فرآیند ETL به طور خودکار نابود می‌شوند. توجه کنید که نیازی به اجرای جداگانه‌ی این فایل نیست و صرفا جهت استفاده در داخل pipeline.py نوشته شده است.

فرآیند اصلی: pipeline.py

با اضافه کردن کتابخانه‌های مورد نیاز برنامه کارمان را آغاز می‌کنیم:

- `psycopg2`: جهت متصل شدن به پایگاه‌های داده از داخل کد پایتون استفاده می‌گردد
- `pygrametl`: فریمورک اصلی مورد استفاده جهت ساده‌سازی بسیاری از فرآیندهای ETL
  - `SQLSource`: کلاسیست که جهت استخراج داده‌ها از مبدا مورد استفاده قرار میگیرد
  - `FactTable`: جهت شبیه‌سازی جدول حقیقت (یا همان `borrows`) در داخل انبار داده به کار خواهد رفت
  - `TypeOneSlowlyChangingDimension`: جهت شبیه‌سازی جداول بُعد (یا همان `users` و `books`) به کار خواهد رفت

در ادامه به توضیح مفصل هر بخش خواهیم پرداخت.

تابع `psycopg2.connect`

پارامترهای ورودی این تابع شامل اطلاعات لازم برای اتصال به دیتابیس می‌باشند. در صورتی که پارامتری ارائه نشود، مقادیر پیش‌فرض سیستم جایگزین آن خواهند شد. پارامترهای قابل دریافت شامل `dbname`، `user`، `password`، `host` و `port` می‌باشند.

دو بار این تابع را صدا خواهیم زد: بار اول برای اتصال به پایگاه داده‌ی مبدا (`library`) و بار دوم جهت اتصال به انبار داده (`warehouse`).

کلاس `pygrametl.ConnectionWrapper`

مقدار بازگردانده شده توسط تابع `psycopg2.connection` را دریافت کرده و از آن برای ذخیره‌ی اطلاعات دیتابیس مقصد استفاده خواهد کرد تا نیاز به ارائه‌ی اطلاعات اتصال به دیتابیس به هنگام استفاده از کلاس‌های دیگر ارائه شده توسط این کتابخانه را حذف کند.

کلاس `SQLSource`

یک کانکشن و یک کوئری دریافت می‌کند. کوئری را در دیتابیس مورد نظر اجرا کرده و رکوردهای حاصل از اجرای کوئری را در قالب یک Python Dictionary باز می‌گرداند. در این بخش ما کوئری‌های لازم برای استخراج داده‌ها از viewهایی که توسط init.sql ساخته شده‌اند را نوشته‌ایم و نتایج آنان را در قالب `books_source`، `users_source` و `borrows_source` ذخیره کرده‌ایم که منتاظر با جداولی هستند که در انبار داده `INSERT` خواهند شد.

کلاس `FactTable`

با دریافت اطلاعات جدول حقیقت (که همان جدول `borrows` می‌باشد) آبجکتی معادل با آن در داخل برنامه می‌سازد. از متد `ensure` این آبجکت جهت `INSERT` کردن داده‌ها به داخل جدول `borrows` در انبار داده استفاده خواهد شد. مزیت استفاده از این کلاس این است که اگر رکوردی از قبل موجود باشد، به آن دست نخواهد زد.

کلاس `TypeOneSlowlyChangingDimension`

با دریافت اطلاعات یکی از `Dimension`های انبار داده، آبجکتی معادل با آن در داخل برنامه می‌سازد. در ادامه برنامه می‌توانیم از این آبجکت‌ها جهت `INSERT` کردن اطلاعات (آن هم بدون دردسر بررسی وجود یا عدم وجود یک رکورد در انبار داده از قبل) استفاده کنیم. توجه کنید که نوع جدول حاصل از این کلاس Type One می‌باشد که به معنی Overwrite کردن داده‌های قدیمی می‌باشد. این در حالیست که در توضیحات انبار داده اشاره شده بود که از ساختار Type 4 استفاده خواهیم کرد: نگهداری اطلاعات تاریخی در جدولی جداگانه. این دو در تناقض با یکدیگر **نیستند**! چراکه triggerهای نوشته شده در داخل انبار داده پس از حذف یا آپدیت شدن یک رکورد در جداول، به صورت خودکار اطلاعات

قدیمی آن رکورد را در جدول `history` متناظر وارد خواهند کرد. در نتیجه ترکیب این دو، کلاس‌های داخل برنامه و triggerهای داخل انبار، خط لوله‌ای امن جهت انتقال اطلاعات بدون از بین رفتن هیچ داده‌ای در طی فرآیند ایجاد می‌کنند.

مزیت استفاده از این کلاس، ارائه‌ی متد `scdensure` می‌باشد که به عنوان ورودی یک رکورد [از دیتابیس مبدا] دریافت کرده و به صورت خودکار بررسی می‌کند که آیا این رکورد از پیش در انبار داده موجود است یا خیر: در صورت موجود بودن، اگر اطلاعات آن با اطلاعات رکورد معادل در انبار داده هماهنگ نباشد آن را آپدیت می‌کند (که در این صورت اطلاعات قدیمی آن به صورت خودکار وارد جدول `history` متناظر می‌شوند) و در غیر این صورت هیچ‌کاری نمیکند. در صورت موجود نبودن نیز آن را به جدول اضافه می‌کند. توجه کنید که انبار داده ما به صورت خودکار زمان `INSERT` شدن رکوردها به داخل دیتابیس را ثبت می‌کند.

اما همه چیز به این سادگی‌ها نیست! کتابخانه‌ی pygrametl هیچ تابع یا کلاسی برای تشخیص رکوردهای حذف شده در دیتابیس مبدا در اختیارمان قرار نمی‌دهد. در اینجاست که مجبور می‌شویم از توابع خودنوشته استفاده کنیم!

توابع `check_borrows`، `check_books` و `check_users`

همانطور که اشاره شد، کتابخانه‌ی pygrametl کلاس یا تابعی جهت بررسی حذف شدن داده‌ای از جدول مبدا به ما ارائه نمی‌دهد. لذا بنده این سه تابع را پیاده‌سازی کردم که در آنان رکوردهای جداول انبار داده با معادل‌های آنان در دیتابیس مبدا مقایسه می‌شوند و در صورت عدم حضور رکوردی معادل در دیتابیس مبدا، این رکورد در انبار داده حذف [و به جدول `history` متناظر انتقال داده] خواهد شد.

سخن نهایی

امیدوارم که این پروژه‌ی [نه چندان!] مختصر رضایتتان را جلب نموده باشد. زمان زیادی برای طراحی و پیاده‌سازی باکیفیت و بهینه‌ی هر جز از این پروژه صرف شده است: مطالعه‌ی بخش‌هایی از کتاب معروف Ralph Kimball در زمینه‌ی Data Warehousing، مطالعه‌ی Documentation فریمورک‌های مختلف، طراحی، پیاده‌سازی و دیباگ پایگاه داده‌ها و کد ETL، نوشتن توضیحات مرتبط به هر بخش و ... . تمامی بخش‌های پروژه روی سیستم خود بنده تست شده‌اند و تا آنجا که اطلاع دارم به درستی عمل می‌کنند. یکی از ایراداتی که از حضور آن در پروژه آگاهم، پیاده‌سازی غیربهینه‌ی توابع `check_`*`tablename`* می‌باشد. متاسفانه به دلیل نزدیک بودن به ضرب‌العجل تحویل پروژه امکان بهبودبخشی آن موجود نمی‌باشد. سپاس‌گزار خواهم بود که پس از بررسی پروژه، در صورت امکان، ایرادات آن را به اطلاع بنده برسانید تا قادر شوم دانش خود را هرچه بیشتر در این زمینه افزایش دهم.

با سپاس و احترام،

آرمان رضایی – ۹۷۲۳۰۳۴