ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

**NEUMANN JÁNOS
INFORMATIKAI KAR**

5. sz. melléklet

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Pollák Péter**

2021 Hallgató törzskönyvi száma: **T/006350/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Pollák Péter**
Törzskönyvi száma: T/006350/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Hadoop data lake fejlesztése kódgeneráláson alapulva
Hadoop data lake development based on code generation

Intézményi konzulens: Dr. Fleiner Rita
Külső konzulens: Kakuk Tamás

Beadási határidő: 2021. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Big Data és üzleti intelligencia specializáció

A feladat

A hallgató feladata egy olyan alkalmazás fejlesztése, amely képes létrehozni azokat a töltési folyamatokat, amelyek betöltik a fájlokat a Hadoop adattárolóba, illetve képes onnan az Oracle adatbázis irányába tetszőleges adatállományt átadni. Az alkalmazás a kódokat metaadat vezérelten hozza létre. Mutassa be a metaadat adatbázist, az állítható paramétereket és a töltés folyamatát.

A dolgozatnak tartalmaznia kell:

- a feladat részletes ismertetését,
- a követelmény specifikációt,
- a hardver specifikációt,
- a rendszertervet, mely tartalmazza a funkciókat, az adatmodellt,
- a választott fejlesztő eszközök ismertetését és a választás indoklását,
- a tesztelés terveit és az eredményeket,
- az elért eredményeket,
- a továbbfejlesztési lehetőségeket.

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

Ph.

.....
Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2023. december 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

Fleiner Rita
.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
7. sz. melléklet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2021.12.13.

Bölök Péter
.....
hallgató aláírása

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun kód:	Tagozat:
Pollák Péter	[redacted]	Mérnök informatikus BSc 2018

Telefon:	Levelezési cím (pl: lakcím):
[redacted]	[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Hadoop data lake fejlesztése kódgeneráláson alapulva

Szakdolgozat / Diplomamunka[2] címe angolul:

Hadoop data lake development based on code generation

Intézményi konzulens:	Külső konzulens:
Fleiner Rita	Kakuk Tamás

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.02.26.	Szakdolgozat témájának megbeszélése	Fleiner Rita
2.	2021.03.05.	Feladatlap megbeszélése	Fleiner Rita
3.	2021.04.29.	Irodalomkutatás megbeszélése	Fleiner Rita
4.	2021.05.06.	Dokumentumok beadásának megbeszélése	Fleiner Rita

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021.05.10.

Fleiner Rita
intézményi konzulens

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun kód:	Tagozat:
Pollák Péter	[redacted]	Mérnök informatikus BSc 2018

Telefon: [redacted]

Levelezési cím (pl: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Hadoop data lake fejlesztése kódgeneráláson alapulva

Szakdolgozat / Diplomamunka[2] címe angolul:

Hadoop data lake development based on code generation

Intézményi konzulens:	Külső konzulens:
Fleiner Rita	Kakuk Tamás

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	**2021.09.26.**	**Irodalomkutatás véglegesítése**	Fleiner Rita
2.	**2021.10.05.**	**Követelmény specifikáció, rendszerterv**	Fleiner Rita
3.	**2021.11.29.**	**Kódolás, eredmények összegzése**	Fleiner Rita
4.	**2021.12.06.**	**Dokumentumok beadásának megbeszélése**	Fleiner Rita

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021.12.10.

Fleiner Rita
intézményi konzulens

Tartalomjegyzék

1. Bevezetés

Szakdolgozatom témája egy olyan alkalmazás készítése, amelyen gyakornokként dolgoztam, és amelynek segítségével nagy mennyiségű napi adatot lehet eltárolni és archiválni a Hadoop adattároló rendszerében, illetve onnan adatot betölteni Oracle adatbázisba.

A beérkező adatok első lépésben egy data lake-be kerülnek be. A data lake abban különbözik a data warehouse-tól, azaz adattárháztól, hogy az adattárházakat riportozási és analitikus célokra használják, az adatszerkezet ennek megfelelően van létrehozva, "schema on write" elv alapján, relációs adatbázisban. Az adatok már a betöltéskor úgy át vannak transzformálva, hogy azok ezekre a célokra felhasználhatóak legyenek.

Ezzel szemben a data lake-ben az adatokat nyers formában tároljuk, úgy, ahogy megérkeznek, "schema on read" elv alapján, tehát adatszerkezetet csak akkor adunk hozzá, amikor ezeket az adatokat feldolgozzuk. A data lake előnye a rugalmasság, illetve olyan problémák megoldása, amelyek relációs adatbázisokkal nehezen, vagy csak drágán valósíthatóak meg. Egy új vonal a kettő keveréke, az úgy nevezett lakehouse, ahol egy data lake-re épül a data warehouse. [1][2][3]

Az alkalmazás alapja egy metaadat adatbázis, amelyben a betöltendő fájlok metaadatai, a táblák szerkezetei, az alkalmazás paraméterei és konfigurációi, illetve a töltés lépései, és ezeknek a lépéseknek az egymástól való függőségeik vannak eltárolva. A betöltendő fájlok egy megadott mappába érkeznek, ahol az alkalmazás automatikusan elkezdi a betöltésüket. A betöltéshez szükséges kódokat az alkalmazás generálja le, a metaadat adatbázisban található metaadatok és paraméterek alapján.

Szakdolgozatomban bemutatom ennek az alkalmazásnak a megvalósítását, a fejlesztés közben felmerült problémákat és azoknak a megoldásait, illetve a tesztelés eredményeit és a továbbfejlesztési lehetőségeket.

2. Lehetséges eszközök áttekintése

2.1. Adattárolás Hadoop-on

A rendszer alapja a Hadoop, ami egy nyílt forráskódú keretrendszer, aminek a célja az adatok több számítógép között elosztott feldolgozása, tárolása. Ebből eredően a Hadoop-ra épülő rendszerek könnyen skálázhatóak. A Hadoop-on több alkalmazás is van, amivel adatokat tudunk eltárolni, feldolgozni és elemezni, mind a maguk előnyeivel és hátrányaival. [4]

2.1.1. Hadoop Distributed File System

Mivel a rendszerbe nagy mennyiségű napi adat fog beérkezni, ezért ennek az eltárolása egy számítógépen sem sebesség, sem tárhely szempontjából nem ideális. A Hadoop Distributed File System, röviden HDFS, egy elosztott fájlrendszer, ami több, tipikusan Linux-t futtató számítógépre lett tervezve, így egy szuperszámítógép helyett használhatunk több olcsóbbat. A HDFS erősen hibatűrő. Az adatok több node-ra vannak másolva és minden adat minimum három példányban van eltárolva, így ha ezek a számítógépek esetlegesen elromlanak, akkor sem történik adatvesztés. A HDFS master/slave architektúrát használ, namenode-okból, és datanode-okból épül fel. A namenode a master, ez tárolja el a fájlok metaadatát, osztja ki a feladatokat, és vezérli a folyamatokat. Több namenode is lehet, ha egy kiesik, akkor egy másik standby namenode lép a helyébe, így növelve a rendszer hibatűrő képességét. A datanode-ok a slave-k, ezek tárolják a blokkokra feldarabolt fájlokat. [5]

A HDFS-nek vannak alternatívái, mint például az S3, ami egy felhős megoldás, ebből eredően az S3 könnyebben skálázható. Ennél az alkalmazásnál viszont fontos az ügyfél saját adatbiztonsági szabályai miatt, hogy egy saját rendszerben legyenek eltárolva az adatok, nem a felhőben, ezért a felhő alapú alternatívák nem ideálisak. [6]

2.1.2. Hive

A Hive egy adattárház kezelő rendszer, ami nagy mennyiségű adat olvasására és írására készült. A Hive a HiveQL nyelvet használja, ami az SQL-hez hasonló nyelv. Az SQL széles körben ismert és használt, így a HiveQL nagyban megkönnyíti a Hive megtanulását és használatát. [7][8]

Fontos megemlíteni, hogy míg a Hive első ránézésre erősen hasonlít a relációsadatbázis-kezelő rendszerekhez, mégsem az, máshogy működik, és másra lett kitalálva. A Hive arra az elképzelésre épül, hogy „write once, read many times”, tehát egyszeri írás és sok olvasás, míg más rendszerek a „read and write many times”, tehát sok írásban és sok olvasásban erősek. Ebből eredően Hive-ot nem érdemes akkor használni, ha például sok dinamikusan változó adatunk van, hanem inkább akkor, ha nagy mennyiségű statikus adatot szeretnénk tárolni, elemezni. Továbbá a Hive az RDBMS-ekkel szemben schema on read, tehát az adatok nem betöltésnél, hanem lekérdezésnél validálódnak. Ez a rendszert rugalmasabbá, az adatbetöltéseket gyorsabbá teszi, viszont a lekérdezéseket lassabbá. A Tez-en vagy Spark-on futó újabb Hive LLAP verziók a nagy mennyiségű adatok feldolgozása mellett az ad-hoc lekérdezéseket is elég gyorsan végre tudják hajtani. [9][10][11][12][13][14]

A Hive táblákhoz tartozó metaadatok a Hive metastore-ban tárolódnak. A metastore egy relációs adatbázissal működik, a Hive jelenleg a MySQL-t, a PostgreSQL-t, Oracle-t és az MS SQL-t támogatja. [15][16][17]

2.1.3. HBase és Impala

A HBase ellentétben a Hive-val egy NoSQL oszlop orientált adatbázis. A HBase erőssége a real-time feldolgozás és az ad-hoc lekérdezések, az Impala forrásként használja. [18]

A Hive mellett a Hadoop-on használhatjuk még az Impalát, ha az RDBMS-ekhez hasonló adatbázist szeretnénk használni. Az Impala is egy SQL-hez hasonló nyelvet használ, mint a Hive, gyors interaktív és ad-hoc lekérdezésekhez találták ki. Az Impala a Hive metastore-ját használja alapul, és a HDFS-t vagy a HBase-t használja forrásként.

Általánosságban elmondható, hogy a Hive-hoz hasonlítva az Impala gyorsabb, ha ad-hoc lekérdezésekről van szó, mivel párhuzamosan képes azokat feldolgozni, de hosszabb lekérdezéseknél vagy bonyolultabb ETL folyamatoknál nagyon memóriaigényes, ilyen esetekben a Hive jobb választás. Továbbá a Hive jobban hibatűrő, mint az Impala, de mivel az Impalát általában inkább gyors, akár többször lefuttatott lekérdezésekhez szokták használni, ezért ez az Impalánál elfogadható kompromisszum. [19][20][21][22]

2.2. Programozási nyelvek a kódgeneráláshoz

2.2.1. Python

A Python nagyon népszerű az adatelemzők körében, sok hasznos könyvtár létezik hozzá, többek között a pandas, ami sokféle forrásból tud adatokat betölteni, majd átalakításokat, elemzéseket végezni rajta. [23]

A pandas-nak forrásként meg lehet adni egy SQL kapcsolatot és egy lekérdezést, eredményként pedig egy dataframe-t fog visszaadni, ami a lekérdezés eredményét tárolja el. A dataframe egy sorokból és oszlopokból álló struktúra, amin a pandas funkcióival komplex átalakításokat lehet végezni. [24]

Az SQL kapcsolat megadásához is több opciónk van. A legtöbb népszerű adatbázishoz létezik Python könyvtár.

Mivel az adatelemzés és a big data nem áll távol egymástól, ezért a Python a big datában is eléggé elterjedt, mindkettő területen lehet használni. Egyik legnépszerűbb Python-os big data eszköz a PySpark, amellyel nagy mennyiségű adatot tudunk transzformálni és adatbázisokba betölteni.

2.2.2. Java

Egy alternatíva a Python-hoz a Java, egy szintén népszerű nyelv, nem csak a big data-ban. A Hadoop is Java-ban íródott. A Java-ban fordításidejű típusellenőrzés van, bonyolultabb, mint a Python, de ha a performancia a legfontosabb szempont, akkor mindenképpen a Java a jobb választás.

2.2.3. Scala

A Scala egy viszonylag új programozási nyelv, ami szintén nagyon elterjedt a big data világában, főleg annak köszönhetően, hogy a Spark is Scala-ban íródott. A Scala ugyanarra a technológiára alapszik, mint a Java, így a kettő kompatibilitása elég jó, a Java-oz hasonlóan fordításidejű típusellenőrzés van benne. A Scala komplexebb nyelv a Python-nál, de gyorsabb. [25]

2.3. Hadoop összekötése Oracle-val

2.3.1. OSCH

Oracle-t HDFS-sel összekapcsolni az Oracle SQL Connector for Hadoop Distributed File System-mel, röviden OSCH-val lehet. Az OSCH lehetővé teszi, hogy a HDFS-ben eltárolt fájlokból external táblákat hozzunk létre Oracle oldalon. Ebből fakadóan például Hive táblákat is át tudunk vinni Oracle-ba. Hátránya, hogy a használatának elég magas licenc díja van, és csak egy pillanatképet ad a HDFS-ben tárolt fájlokról, ha azok módosulnak, az Oracle-ban nem látszik. [26]

2.3.2. PySpark

Másik módja, hogy HDFS-ből adatokat vigyünk át Oracle-ba, az a PySpark használata. Az OSCH-hoz képest ennek az előnye az, hogy jobban elterjedt, illetve, hogy ingyenes. Hátránya, hogy használata lassabb és bonyolultabb, mint az OSCH-nak, mivel nekünk kell megírni és karbantartani az adatbetöltéshez szükséges kódokat.

2.4. Adatbázis-kezelő rendszerek a metaadatbázishoz

A kódgeneráláshoz szükségünk van egy adatbázisra, ami a bejövő fájlok és az eltárolt táblák metaadatait, illetve a rendszer paramétereit tárolja. Első gondolatra a Hive-ot is lehetne használni a metadatok tárolására, de mivel az ad-hoc lekérdezések nem az erőssége, és a kódgenerálás közben elképzelhető, hogy az alkalmazás többször is lekérdez majd különböző metaadatokat, ezért érdemesebb egy tradicionális adatbázis kezelőt használni.

A fizetett opciók között a két talán legerősebb adatbázis kezelő rendszer az SQL Server és az Oracle, míg az open-source adatbázis kezelők között a PostgreSQL és a MySQL. Az adatbázisok közti választást az befolyásolja, hogy mekkora rendszerhez és milyen feladatra kell, hogy milyen mértékű támogatás szükséges, és természetesen az adatbázis ára. [27]

3. Követelmény specifikáció

3.1. Áttekintés

Az alkalmazás egy banknak készült Hadoop alapú on-prem megoldás. Az oka, hogy on-prem és nem felhős megoldás az, hogy a bank az adatbiztonsági szabályai miatt helyi szervereken szeretné eltárolni az adatokat. Az alkalmazás az optimális működésének érdekében több számítógépen fut, így kihasználva a Hadoop által nyújtott előnyöket, ezzel biztosítva, hogy könnyen skálázható legyen. Az alkalmazás működése a beállított paraméterektől és a fájlok metaadataitól függ, ezért ezeknek a helyes megadásuk elengedhetetlen az alkalmazás megfelelő működéséhez. Ezeknek a paramétereknek és metaadatoknak a beállítása közvetlenül a metaadat adatbázisban és egyéb konfigurációs fájlokban történik.

A legenerált kódok ezeknek a metaadatoknak és paramétereknek a segítségével jönnek létre. A kódgenerálás azért szükséges, mivel a bank rengeteg fájlt tárol el a szerverein, és ezeknek a betöltése és a hozzájuk tartozó adatstruktúrák létrehozása nagyon monoton munka lenne és nagyon sok időbe telne. Kódgenerálás segítségével csak fel kell vennünk a szükséges metaadatokat, és az alkalmazás ez alapján létrehoz minden szükséges kódot, ezzel lényegesen csökkentve azt az időt és munkát, ami az adatok betöltéséhez szükséges. Ez azt is eredményezi, hogy a jövőbeni új fájlok felvétele nem igényel hosszú fejlesztést, paraméterezés segítségével gyors átfutási idővel élesbe tudnak kerülni.

A bank jelenleg adattárházként Oracle-t használ, de nagy mennyiségű adat hosszan tartó tárolása az Oracle licenc díja miatt drága lenne, így nem Oracle-ba, hanem egy másik, olcsóbb rendszerbe szeretné betölteni az adatokat. Az alkalmazásnak viszont kompatibilisnek kell lennie a bank jelenlegi megoldásával, tehát képesnek kell lennie Oracle irányába tetszőleges adatot betöltenie Hadoop-ból.

Fontos, hogy az alkalmazás logolja a töltési folyamatokat és a kódgenerálást, így könnyítve a hibakezelést és az üzemeltetést.

3.2. Funkcionális követelmények

3.2.1. Töltési folyamatokhoz szükséges kódok legenerálása

Az alkalmazás alapja a metaadatbázis. Az alkalmazásnak ebben az adatbázisban megtalált adatok alapján létre kell hoznia azokat a kódokat, amik majd a fájlok betöltésénél szükségesek lesznek. Ahhoz, hogy ezek a kódok létre tudjanak jönni, a metaadatbázisba fel kell venni a betöltendő fájloknak a struktúráját, tehát azt, hogy milyen oszlopokat tartalmaznak. Továbbá szükséges megadni a fájl típusát. Az alkalmazás képes fixed width TXT és CSV fájlok betöltésére, a fixed width TXT fájlokat az alkalmazás először átkonvertálja CSV formátumba. Erre azért van szükség, mert Hive-

on belül a CSV fájlokkal ellentétben a fixed width TXT fájlok feldolgozására nincs szabványmegoldás, minden fájlhoz külön regex-et kellene írni.

Szükséges még felvenni a metaadatbázisba azokat a táblákat, ahova az adatok be lesznek töltve. Ezekhez a táblákhoz az alkalmazásnak attól függően kell létrehoznia a szükséges kódokat, hogy új tábláról, vagy már létező tábla módosításáról van szó.

3.2.2. Fájlok betöltése Hadoop-ba

Amikor az extract fájlok megérkezenek a landing zone-ba, az alkalmazásnak ezeket észre kell vennie, és automatikusan el kell indítania a HDFS-be és a Hadoop adattárolójába való betöltést a legenerált kódokkal. Továbbá a beérkezett fájllal történő folyamatokat logolnia kell a metaadatbázisba, hogy azok nyomon követhetőek legyenek.

3.2.3. Adatok betöltése Oracle-ba Hadoop-ból

Az alkalmazásnak képesnek kell lennie a Hadoop adattárolójából Oracle-ba tetszőleges adatokat betölteni. Az ehhez szükséges paramétereket szintén fel kell venni a metaadatbázisba.

3.2.4. Betöltött adatok karbantartása

A metaadatbázisban megadott paraméterek alapján az adatokat archiválnia, esetleg bizonyos idő után automatikusan törölnie kell az alkalmazásnak.

3.2.5. Kódgenerálás és töltési folyamatok logolása

Az alkalmazásnak a kódgenerálást és a töltési folyamatokat logolnia kell, hogy a folyamatok visszakövethetőek legyenek, ezzel lehetővé téve a monitorozást, így könnyítve az üzemeltetést és a hibajavítást.

4. Az eszközök kiválasztása

A beérkezett fájlokat HDFS-en tároljuk el, majd onnan Hive-ba töltjük be. A Hive előnye az Impalával és a HBase-el szemben az, hogy sokoldalúbb, batch futtatásban erősebb, illetve hibatűrőbb és több eszközzel kompatibilis. Az is előny a Hive-nál, hogy az adatokat hasonlóan az Oracle-hoz, relációs formában tudjuk eltárolni, ezért a Hive-ból való áttöltés megvalósítása egyszerűbb, mint például ha HBase-t használnánk, ami egy NoSQL adatbázis és az adatokat nem relációs formában tárolja. Továbbá a Hive Tez-en fog futni, ez a lekérdezéseket és az adatfeldolgozást gyorsítja fel. A Tez kiváltja a MapReduce-t, és előnye, hogy amíg a MapReduce az átmeneti eredményeket kiírja HDFS-re, addig a Tez képes teljesen a memóriából dolgozni. A Tez irányított körmentes

gráfokkal dolgozik, egy gráf egy job-nak, a gráfok csúcsai a job-on belüli feladatoknak, az élek pedig az adatmozgásoknak felelnek meg. A Spark is hasonlóan működik, gráfokat használ az adatfeldolgozáshoz és memóriából dolgozik, de végül azért a Tez-re esett a választás, mert jobban ki tudja használni a YARN-t mint erőforráskezelőt, és kevesebb erőforrást használ, mint a Spark. [28][29][30]

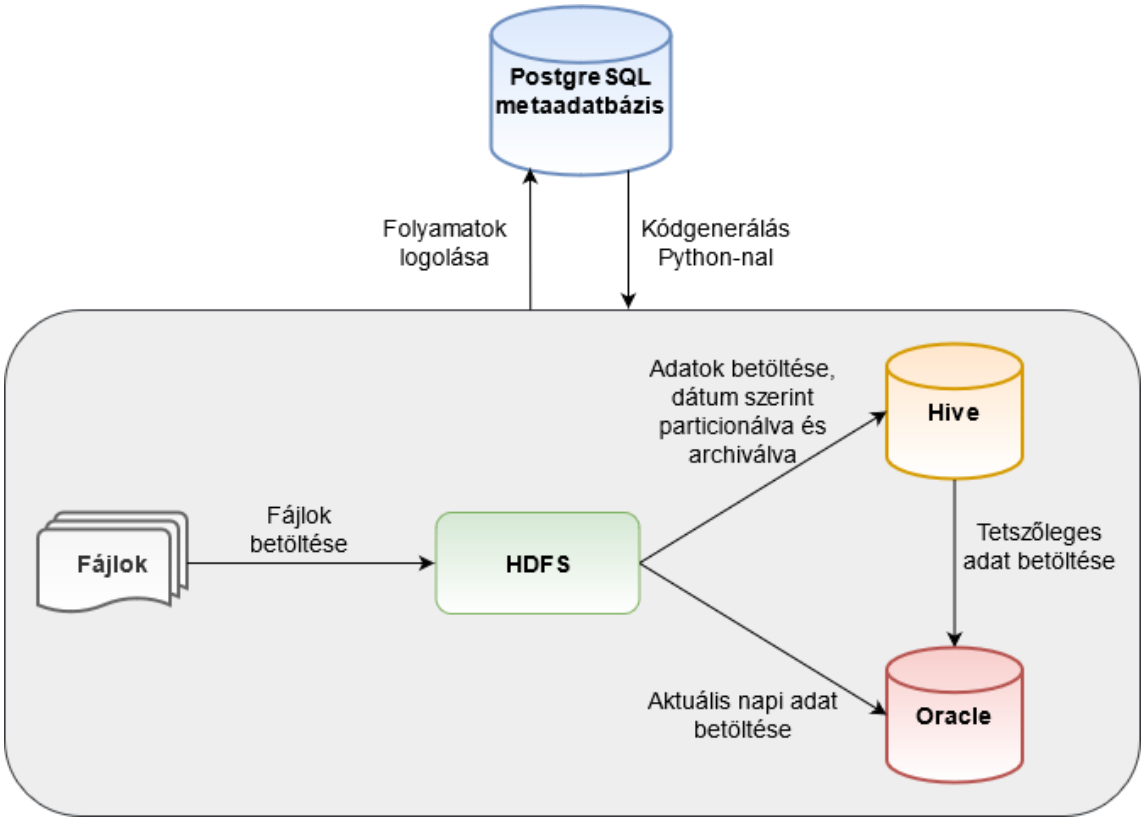
Ahogy megérkeznek a fájlok, első lépésben HDFS-be lesznek betöltve. Az egyetlen átalakítás, ami történni fog velük, hogy a fixed width TXT fájlok át lesznek konvertálva CSV fájlokká.

Ezután a fájlok PySpark segítségével betöltődnek Oracle-ba, illetve egy egyszerű HiveQL kód segítségével létrejönnek azok a Hive external táblák, amik a HDFS-en eltárolt adatokra mutatnak. Az OSCH előnye, hogy external táblákat is létre tud hozni a HDFS-en lévő fájlokból, és a használata egyszerűbb, de a PySpark ingyenes, és az external táblák hiánya nem jelent olyan nagy problémát. Végül a Hive external táblákban lévő adatokat betöltjük dátum szerint particionált Hive táblákba, HiveQL kódok segítségével.

A szükséges PySpark és HiveQL kódokat Python scriptek hozzák létre a metaadatok és template-ek segítségével. A Python egyszerűsége és a big data-ban való elterjedtsége miatt jobb választásnak bizonyul a Scala-nál és a Java-nál, és mivel komolyabb számításokra nincs szükség, ezért a performancia sem olyan fontos szempont.

A metaadatbázis egy PostgreSQL adatbázis, mivel ingyenes, open-source, széles körűen elterjedt, és képes ellátni azokat a feladatokat, amelyekre egy ilyen alkalmazásnak szüksége lehet. Továbbá a Hive-nak is szüksége van egy metastore adatbázisra, és a PostgreSQL ezt a szerepet is be tudja tölteni. Így ugyanazzal a Python kóddal el lehet érni a saját metaadatbázisunkat és a Hive metastore-t is.

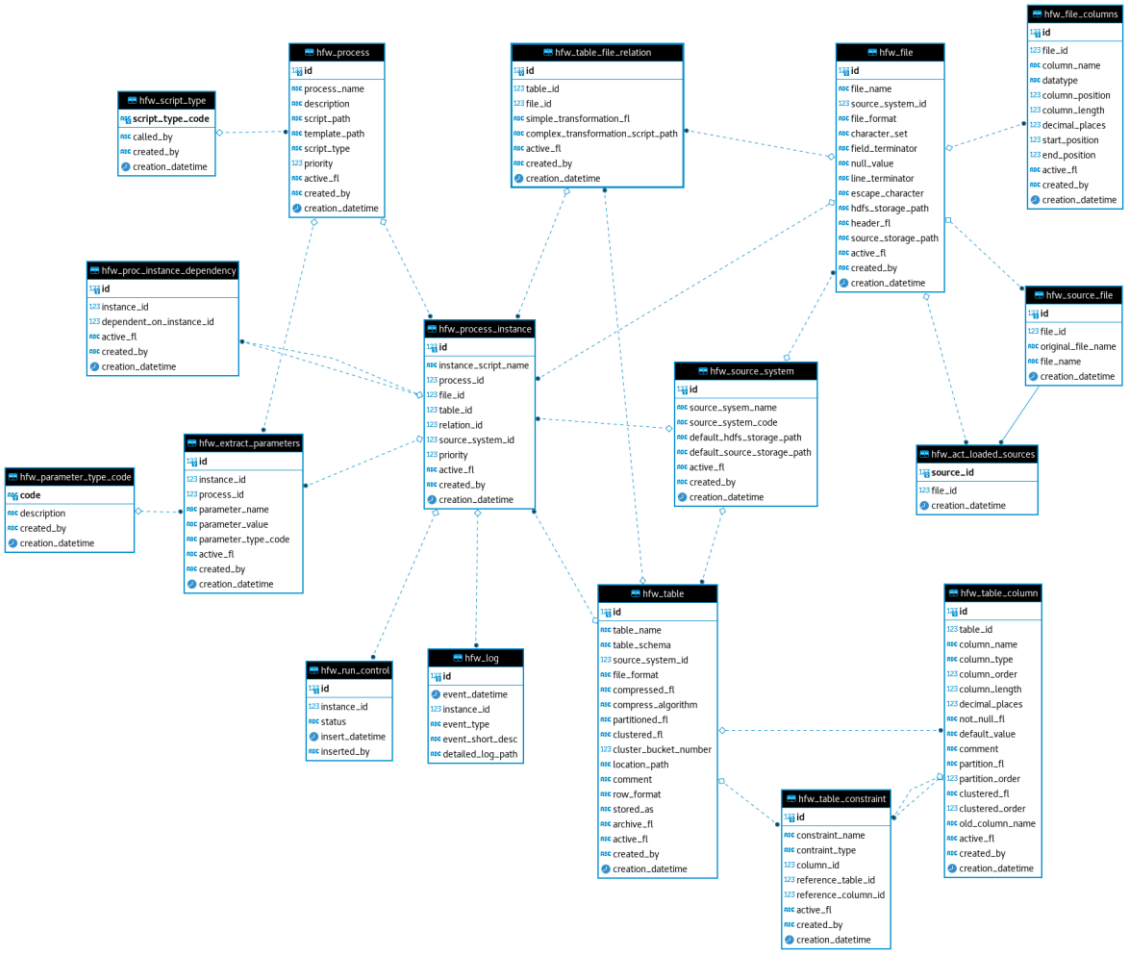
5. Részletes rendszerterv



1. ábra: A töltési folyamatok

5.1. Metaadatbázis bemutatása

A fájlok betöltéséhez és az ehhez tartozó kódok legenerálásához több táblára is szükség van, amelyek eltárolják a fájlok, illetve azon Hive és Oracle táblák metaadatait, ahova azokat betöltjük. Az éppen futó töltési folyamat, a logok és az alkalmazás paraméterei is a metaadatbázisban találhatók meg. A HFW a Hadoop Framework rövidítése.



2. ábra: A metaadatbázis EK diagramja

HFW_TABLE

A táblalétrehozó kódok legeneráláshoz szükséges adatokokat tartalmazza, mint például a tábla neve és sémája. Olyan információkat is tartalmaz, ami kifejezetten a Hive-hoz szükséges, mint például, hogy alkalmazunk-e particionálást vagy klaszterezést a táblán.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott tábla egyedi azonosítója.	serial

table_name	A tábla neve, ezen a néven lesz létrehozva a Hive és az Oracle tábla is.	varchar(100)
table_schema	A tábla sémája.	varchar(100)
source_system_id (idegen kulcs a HFW_SOURCE_SYSTEM táblára)	Idegen kulcs, a forrásrendszer egyedi azonosítója.	integer
file_format	A forrásfájl formátuma. TXT vagy CSV.	varchar(100)
compressed_fl	Flag, hogy tömörítve legyen-e a tábla vagy sem.	char(1)
compress_algorithm	Amennyiben a tömörítés be van kapcsolva, abban az esetben milyen algoritmust használjon.	varchar(100)
partitioned_fl	Flag, hogy particionálva van-e a tábla vagy sem.	char(1)
clustered_fl	Flag, hogy a klaszterezés be van-e kapcsolva, vagy sem.	char(1)
cluster_bucket_number	Amennyiben a klaszterezés be van kapcsolva, a bucketek száma.	integer
location_path	A tábla elérési útja HDFS-en.	varchar(200)
comment	Komment a táblához.	varchar(500)
row_format	A Hive tábla sor formátuma.	varchar(100)
stored_as	Miként tároljuk el HDFS-en a Hive táblát.	varchar(100)

archive_fl	Flag, hogy az archiválás be van-e kapcsolva.	char(1)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

1. táblázat: HFW_TABLE tábla attribútumai

HFW_TABLE_COLUMN

A HFW_TABLE táblában megtalálható táblák oszlopait tartalmazza. Ezekkel az oszlopokkal fog létrejönni a legenerált kód. Itt lehet beállítani, hogy mely oszlopok szerint legyen particionálva, illetve klaszterezve a tábla.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott oszlop egyedi azonosítója.	serial
table_id (idegen kulcs a HFW_TABLE táblára)	Idegen kulcs, azon tábla egyedi azonosítója, amelyhez az oszlop tartozik.	integer
column_name	Az oszlop neve.	varchar(100)
column_type	Az oszlop típusa.	varchar(100)
column_order	Az oszlop helye az oszlopok sorrendjében.	integer
column_length	Amennyiben az oszlop típusának van hossza (például varchar), abban az esetben szükséges megadni.	integer
decimal_places	Amennyiben az oszlop decimal típusú, abban az esetben szükséges	integer

	megadni, hogy hány tizedesjegy pontosságú.	
not_null_fl	Flag, hogy lehete-e null az oszlop vagy sem.	char(1)
default_value	Az oszlop alapértelmezett értéke.	varchar(500)
comment	Komment az oszlophoz.	varchar(500)
partition_fl	Flag, hogy e szerint az oszlop szerint van-e particionálva a tábla.	char(1)
partition_order	Amennyiben az oszlop szerint particionálunk, abban az esetben az oszlop helye a particionáló oszlopok sorrendjében.	integer
clustered_fl	Flag, hogy e szerint az oszlop szerint van-e klaszterezve a tábla.	char(1)
clustered_order	Amennyiben az oszlop szerint klaszterezünk, abban az esetben az oszlop helye a klaszterező oszlopok sorrendjében.	integer
old_column_name	Az oszlop előző neve, az alter kódok legenerálásához szükséges. Ezt egy oszlop átnevezésekor kézzel kell beállítani.	varchar(100)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

2. táblázat: HFW_TABLE_COLUMN tábla attribútumai

HFW_TABLE_CONSTRAINT

A HFW_TABLE_COLUMN táblában megtalálható oszlopokhoz tartozó megszorításokat tartalmazza. Elsődleges és idegen kulcs létrehozása lehetséges.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott tábla megszorítás egyedi azonosítója.	serial
constraint_name	A megszorítás neve.	varchar(100)
constraint_type	A megszorítás típusa, PRIMARY KEY vagy FOREIGN KEY.	varchar(100)
column_id (idegen kulcs a HFW_TABLE_COLUMN táblára)	Idegen kulcs, azon oszlop egyedi azonosítója, amelyhez a megszorítás tartozik.	integer
reference_table_id (idegen kulcs a HFW_TABLE táblára)	Idegen kulcs, amennyiben a megszorítás FOREIGN KEY típusú, az összekapcsolt tábla egyedi azonosítója.	integer
reference_column_id (idegen kulcs a HFW_TABLE_COLUMN táblára)	Idegen kulcs, amennyiben a megszorítás FOREIGN KEY típusú, az összekapcsolt oszlop egyedi azonosítója.	integer
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

3. táblázat: HFW_TABLE_CONSTRAINT tábla attribútumai

HFW_SOURCE_SYSTEM
A forrásrendszerekhez tartozó adatokat tartalmazza, például, hogy hol találhatóak meg a hozzá tartozó fájlok HDFS-en.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott forrásrendszer egyedi azonosítója.	serial
source_system_name	A forrásrendszer neve.	varchar(500)
source_system_code	A forrásrendszerhez tartozó rövid azonosító kód.	varchar(100)
default_hdfs_storage_path	Hol tároljuk el a forrásrendszerhez tartozó fájlokat HDFS-en.	varchar(100)
default_source_storage_path	A forrásrendszerhez tartozó fájlok helye, mielőtt azokat HDFS-be betöltjük.	varchar(100)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

4. táblázat: HFW_SOURCE_SYSTEM tábla attribútumai

HFW_SOURCE_FILE
A forrásfájlok metaadatait tartalmazó tábla.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott forrásfájl egyedi azonosítója.	serial
file_id (idegen kulcs a HFW_FILE táblára)	Idegen kulcs, ami összeköti a forrásfájlt a HFW_FILE táblában tárolt metaadataival.	integer

original_file_name	A forrásfájl eredeti neve. Abban az esetben szükséges, ha a fájl át lett nevezve a betöltés során.	varchar(200)
file_name	A forrásfájl neve.	varchar(200)
creation_datetime	Mikor jött létre a rekord.	timestamp

5. táblázat: HFW_SOURCE_FILE tábla attribútumai

HFW_ACT_LOADED_SOURCES

Azokat a forrásfájlokat tartalmazó tábla, amelyek a legutoljára voltak betöltve.

Név	Magyarázat	Típus
source_id (elsődleges kulcs) (idegen kulcs a HFW_SOURCE_FILE táblára)	Az adott forrásfájl egyedi azonosítója. Összeköti a táblát a HFW_SOURCE_FILE táblával.	serial
file_id (idegen kulcs a HFW_FILE táblára)	Idegen kulcs, ami összeköti a betöltött forrásfájlt a HFW_FILE táblában tárolt metaadataival.	integer
creation_datetime	Mikor jött létre a rekord.	timestamp

6. táblázat: HFW_ACT_LOADED_SOURCES tábla attribútumai

HFW_FILE

A forrásfájlok metaadatait tartalmazza. Ennek a táblának a segítségével töltődnek be a fájlok HDFS-be.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott fájl egyedi azonosítója.	serial
file_name	A fájl neve	varchar(100)

source_system_id (idegen kulcs a HFW_SOURCE_SYSTEM táblára)	Idegen kulcs, a forrásrendszer, amihez a fájl tartozik.	integer
file_format	A fájl formátuma.	varchar(100)
character_set	A fájl karakterkódolása.	varchar(100)
field_terminator	Az oszlopokat elválasztó karakter.	varchar(100)
null_value	A null értéket jelző karakter vagy kifejezés.	varchar(100)
line_terminator	A sor végét jelző karakter.	varchar(100)
escape_character	A fájlban használt feloldójel.	varchar(100)
hdfs_storage_path	Hol tároljuk el HDFS-en a fájlt.	varchar(100)
header_fl	Flag, hogy van-e a fájlnak fejléce.	char(1)
source_storage_path	A fájl helye, mielőtt HDFS-be betöltjük.	varchar(100)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

7. táblázat: HFW_FILE tábla attribútumai

HFW_FILE_COLUMNS

A HFW_FILE táblában eltárolt fájlok oszlopait tartalamzza.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott oszlop egyedi azonosítója.	serial

file_id (idegen kulcs a HFW_FILE táblára)	Idegen kulcs, ami összeköti az oszlopot azzal a fájllal, amihez tartozik.	integer
column_name	Az oszlop neve.	varchar(100)
datatype	Az oszlop típusa.	varchar(100)
column_position	Az oszlop helye az oszlopok sorrendjében.	integer
column_length	Az oszlop hossza.	integer
decimal_places	Amennyiben az oszlop decimal típusú, abban az esetben szükséges megadni, hogy hány tizedesjegy pontosságú.	integer
start_position	Fixed width TXT fájloknál az oszlop kezdő pozíciója a fájlban.	integer
end_position	Fixed width TXT fájloknál az oszlop végének a pozíciója a fájlban.	integer
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

8. táblázat: HFW_FILE_COLUMNS tábla attribútumai

HFW_TABLE_FILE_RELATION

A táblák és a fájlok közötti kapcsolatokat, illetve a transzformációkhoz szükséges adatokat tartalmazza.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott rekord egyedi azonosítója.	serial

table_id (idegen kulcs a HFW_TABLE táblára)	A tábla id-ja.	integer
file_id (idegen kulcs a HFW_FILE táblára)	A fájl id-ja.	integer
simple_transformation_fl	Amennyiben ez igaz, a fájlhoz az alkalmazás fogja legenerálni a szükséges Hive insert kódokat.	char(1)
complex_transformation_script_path	Ha a simple_tranformation_fl nem igaz, akkor az alkalmazás az itt megadott scriptet fogja lefuttatni Hive insert-kor.	varchar(500)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

9. táblázat: HFW_TABLE_FILE_RELATION tábla attribútumai

HFW_EXTRACT_PARAMETERS

A fájlok betöltéséhez szükséges paramétereket tartalmazza. A paraméterek lehetnek globális, process és process instance szintűek.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott paraméter egyedi azonosítója.	serial
instance_id (idegen kulcs a HFW_PROCESS_INSTANCE táblára)	Annak a process instance-nek az id-ja, amihez a paraméter tartozik.	integer

process_id (idegen kulcs a HFW_PROCESS táblára)	Annak a process-nek az id-ja, amihez a paraméter tartozik.	integer
parameter_name	A paraméter neve.	varchar(100)
parameter_value	A paraméter értéke.	varchar(100)
parameter_type_code (idegen kulcs a HFW_PARAMETER_TYPE_CODE táblára)	Egy rövid kód, a paraméter típusa.	varchar(10)
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

10. táblázat: HFW_EXTRACT_PARAMETERS tábla attribútumai

HFW_PROCESS

A töltési folyamatok általános, logikai reprezentációit tartalmazza. Ezeknek a folyamatoknak a fizikai reprezentációi a HFW_PROCESS_INSTANCE táblában lesznek eltárolva, ott a folyamatok konkrét fájlokhoz vagy táblákhoz lesznek kötve.

Név	Magyarázat	Típus
id	Az adott process egyedi azonosítója.	serial
process_name	A process neve.	varchar(100)
description	A process rövid leírása.	varchar(100)
script_path	A script elérési útja, amiben a process-hez tartozó kód található.	varchar(100)
template_path	A template fájl elérési útja, amennyiben a process egy template alapú kódgenerálás.	varchar(100)

script_type (idegen kulcs a HFW_SCRIPT_TYPE táblára)	A script típusa, például Python.	varchar(50)
priority	A process prioritása.	integer
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

11. táblázat: HFW_PROCESS tábla attribútumai

HFW_PROCESS_INSTANCE

A konkrét fájlokhoz és táblákhoz tartozó legenerált scripteket tartalmazza, illetve a generáló scripteket, amik ezeket létrehozzák. Ebben a táblában megtalált scriptek fognak majd a töltési folyamatok során lefutni.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott process instance egyedi azonosítója.	serial
instance_script_name	Az adott instance-hoz tartozó script neve, a process nevéből, a source system nevéből és a fájl nevéből épül fel.	varchar(500)
process_id (idegen kulcsa a HFW_PROCESS táblára)	Idegen kulcs, ami összeköti az instance-t a process-el, amit megvalósít.	integer
file_id (idegen kulcsa a HFW_FILE táblára)	Idegen kulcs, abban az esetben kell kitölteni, ha a process valamilyen fájllal dolgozik (például fixed width TXT fájl	integer

	átkonvertálása CSV fájllá).	
table_id (idegen kulcsa a HFW_TABLE táblára)	Idegen kulcs, abban az esetben kell kitölteni, ha a process valamilyen táblával dolgozik (például táblalétrehozó kód legenerálása).	integer
relation_id (idegen kulcsa a HFW_TABLE_FILE_RELATION táblára)	Idegen kulcs, abban az esetben kell kitölteni, ha a process egy fájllal és egy táblával dolgozik (például fájl betöltése táblába).	integer
source_system_id (idegen kulcsa a HFW_SOURCE_SYSTEM táblára)	Idegen kulcs, hogy melyik forrásrendszerhez tartozik az adott process instance.	integer
priority	A process instance prioritása.	integer
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

12. táblázat: HFW_PROCESS_INSTANCE tábla attribútumai

HFW_PROC_INSTANCE_DEPENDENCY

Ez a tábla tartalmazza, hogy melyik process instance melyik másik process instance-től függ. Például, ha sikeresen lefut egy HDFS-be fájl betöltés process, akkor, ha ebben a táblában szerepel egy ettől függő process instance, például betöltés Oracle-ba, akkor ezt követően az is lefut.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott rekord egyedi azonosítója.	serial
instance_id (idegen kulcs a HFW_PROCESS_INSTANCE táblára)	A függő process instance id-ja.	integer
dependent_on_instance_id (idegen kulcs a HFW_PROCESS_INSTANCE táblára)	A process instance id-ja, amitől függ.	integer
active_fl	Flag, hogy az adott rekord aktív-e vagy sem.	char(1)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

13. táblázat: HFW_PROC_INSTANCE_DEPENDENCY tábla attribútumai

HFW_RUN_CONTROL

Ebbe a táblába kerülnek be azok a process instance-ok, amik le fognak futni. Ezt a táblát egy script periodikusan ellenőrzi, és lefuttatja a benne lévő process-eket.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott rekord egyedi azonosítója.	serial
instance_id (idegen kulcs a HFW_PROCESS_INSTANCE táblára)	A process instance id-ja, ami le fog futni.	integer
status	A process instance státusza.	varchar(50)

insert_datetime	Mikor lett beillesztve az adott process instance a táblába.	timestamp
inserted_by	User, aki beillesztette az adott process instance-t.	varchar(50)

14. táblázat: HFW_RUN_CONTROL tábla attribútumai

HFW_LOG

Ez a tábla tartalmazza a lefutott process instance-ok logjait. Ezek a logok csak rövid összefoglalók, hogy mi futott le, mikor, és milyen eredménnyel. A futás részletei különböző log fájlokban vannak eltárolva.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott log egyedi azonosítója.	serial
event_datetime	Mikor futott le az adott process.	timestamp
instance_id (idegen kulcs a HFW_PROCESS_INSTANCE táblára)	A loghoz tartozó process instance id-ja.	integer
event_type	A log típusa, például START vagy ERROR.	varchar(100)
event_short_desc	A logolt process rövid leírása.	varchar(1000)
detailed_log_path	A részletes logfájl elérési útja.	varchar(500)

15. táblázat: HFW_ LOG tábla attribútumai

HFW_SCRIPT_TYPE

Az alkalmazásban megtalálható scriptek típusait tartalmazza, és azt, hogy milyen paranccsal hívhatók meg. Például a Python scripteket python3 paranccsal lehet meghívni.

Név	Magyarázat	Típus
script_type_code (elsődleges kulcs)	A script típus neve.	varchar(30)

called_by	Milyen paranccsal lehet meghívni az adott típusú scripteket.	varchar(100)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

16. táblázat: HFW_ SCRIPT_TYPE tábla attribútumai

HFW_PARAMETER_TYPE_CODE

A HFW_EXTRACT_PARAMETERS táblában lévő paraméterek típusait tartalmazó tábla.

Név	Magyarázat	Típus
code	Paraméter típushoz tartozó rövid kód.	varchar(10)
description	Paraméter típus rövid leírása.	varchar(100)
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

17. táblázat: HFW_PARAMETER_TYPE_CODE tábla attribútumai

5.2. Interface táblák bemutatása

Az interface táblák arra szolgálnak, hogy az Oracle és a metaadatbázis között lehessen kommunikálni. Az interface táblák szinkronizálásáért egy bash script felel, amely szükség esetén az Oracle oldalon lévő interface táblák tartalmát átmásolja a metaadatbázis oldalon lévő interface táblákba.

INT_LOAD_PARTITION

Az INT_LOAD_PARTITION tábla megtalálható metaadatbázis és Oracle oldalon is. Itt adhatjuk meg azoknak a tábláknak a neveit és Hive partícióit, amelyeket be szeretnénk tölteni Oracle-ba.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott rekord egyedi azonosítója.	serial

table_name	A tábla neve, aminek az egyik Hive partícióját be szeretnénk tölteni Oracle-ba.	varchar(100)
calendar_date	Minden Hive tábla fel van osztva calendar_date alapján partíciókra. Ez az oszlop azt adja meg, hogy melyik partíciót, tehát melyik napi adatot szeretnénk betölteni Oracle-ba.	date
created_by	User, aki létrehozta az adott rekordot.	varchar(50)
creation_datetime	Mikor jött létre a rekord.	timestamp

18. táblázat: INT_LOAD_PARTITION tábla attribútumai

INT_TABLE_COPY

Az INT_TABLE_COPY tábla csak Oracle oldalon található meg. Ebbe a táblába íródnak be azoknak az interface tábláknak a nevei, amelyekben valamilyen változás történt. Az alkalmazás ennek a táblának az alapján tudja eldönteni, hogy mely interface tábláknak a tartalmát szükséges átmásolnia Oracle-ból a metaadatbázisba. A tábla nem kézzel töltődik, a többi interface tábla triggerek segítségével ír bele, amikor azokban új rekord keletkezik, vagy létező rekord változik meg vagy törlődik ki.

Név	Magyarázat	Típus
id (elsődleges kulcs)	Az adott rekord egyedi azonosítója.	number(22)
table_name	Annak az interface táblának a neve, aminek a tartalmát át kell másolni Oracle-ból a metaadatbázisba.	varchar2(60)
creation_datetime	Mikor jött létre a rekord.	timestamp

19. táblázat: INT_TABLE_COPY tábla attribútumai

6. Implementáció

6.1. Postgres metaadatbázis

Mivel az alkalmazás alapja a metaadatbázis, ezért először ezt hoztam létre. Postgres-en belül létrehoztam egy új adatbázis-t HFW_TEST néven, majd ezen belül létrehoztam a HFW_METADATA sémát. Ezután felvettem a táblákat és a köztük lévő relációkat.

Továbbá létrehoztam az alábbi triggereket, amelyek a metaadatbázis használatát segítik majd elő, ezek a következőek:

- ADD_DEFAULT_COLUMN_TRIGGER: Azután aktiválódik, hogy a HFW_TABLE táblába felveszünk egy új rekordot, és létrehozza a következő három oszlopot a HFW_TABLE_COLUMN táblába az új táblához. CALENDAR_DATE, amely az adat effektív dátumát adja meg, a SOURCE_ID, amely a HFW_SOURCE_FILE tábla egyik id-ját tárolja, és azt mutatja meg, hogy az adat melyik forrásból jött, illetve a RECORD_INSERT_DATETIME, amely az adat betöltésének a pontos idejét adja meg. Ezek az oszlopok csak Hive oldalon fognak megjelenni, a tábla többi oszlopa mellett.

- FILE_SOURCE_SYSTEM_ID_UPDATED,
  TABLE_SOURCE_SYSTEM_ID_UPDATED,
  FILE_WITH_SOURCE_SYSTEM_ID_INSERTED,
  TABLE_WITH_SOURCE_SYSTEM_ID_INSERTED,
  SOURCE_SYSTEM_UPDATED: Ezek a triggerek mind arra szolgálnak, hogy amennyiben egy fájlnak a HFW_FILE-ban, vagy egy táblának a HFW_TABLE-ben meg van adva egy SOURCE_SYSTEM_ID, akkor az ezzel kapcsolatos adatok automatikusan frissüljenek. Az adatok akkor frissülnek, ha egy olyan új fájlt vagy táblát veszünk fel, ami rendelkezik SOURCE_SYSTEM_ID-val, vagy ha egy fájlnak vagy táblának megváltozik a SOURCE_SYSTEM_ID-ja, illetve ha a HFW_SOURCE_SYSTEM táblában egy forrásrendszernek megváltoznak az adatai. Ezek az adatok a HFW_FILE-ban a HDFS_STORAGE_PATH, ami az a HDFS elérési út, ahova a fájl be lesz töltve, és a SOURCE_STORAGE_PATH, ami a fájlnak a forrásrendszerbeli elérési útja. HFW_TABLE-ben ez az adat a LOCATION_PATH, ami a Hive tábla HDFS elérési útja lesz. Természetesen lehetőség van ezeket az adatokat felülírni, ha például valamelyik fájl egy adott forrásrendszerhez tartozik, de máshol szeretnénk HDFS-en eltárolni, mivel megszorítás nincs ezekre.

Végül létrehoztam nézeteket, amelyek arra szolgálnak, hogy könnyebb legyen bizonyos táblákat monitorozni, vagy, hogy az alkalmazás könnyedén le tudja kérdezni bizonyos táblák tartalmát.

- HFW_EXTRACT_PARAMS_BY_INSTANCE_ID_V: A HFW_EXTRACT_PARAMETERS egy globális paraméter tábla, minden paraméter ebben található meg. A táblában lehetőség van a paramétereknek megadni egy PROCESS_ID-t vagy egy PROCESS_INSTANCE_ID-t, így a paramétereknek három szintje van. A globális szint, ahol egyik sincs megadva, az ebbe a szintbe felvett paraméterek alapértelmezett értékként szolgálnak, hogy ne kelljen minden process-hez és process instance-hoz külön felvenni a paramétereket. A további két szint a process szint és process instance szint. Ez a nézet hozzárendeli az összes process instance-hoz az összes paramétert úgy, hogy a paraméterek értéke a megfelelő szint szerint legyenek megadva. Ez olyan process instance-oknál lesz fontos, mint például a Hive táblalétrehozó kód generáló, amely képes archív táblákat létrehozni adott tömörítési codec-kal. Egy ARCHIVE_COMPRESS_CODEC paraméter alapból fel van véve globális szinten, de ha egy adott táblát máshogy szeretnénk tömöríteni, akkor felvehetjük ugyanezt a paramétert az adott process instance id-val, és a nézet ahhoz a process instance-hoz azt az értéket fogja majd visszaadni.

	id	instance_id	process_id	parameter_name	parameter_value
1	3	[NULL]	[NULL]	PARTITION_RETENTION_DAYS	30
2	12	[NULL]	6	PARTITION_RETENTION_DAYS	14
3	11	10	[NULL]	PARTITION_RETENTION_DAYS	60

3. ábra: Globális, process és process instance szintű paraméterek a HFW_EXTRACT_PARAMETERS táblában

	instance_id	parameter_name	parameter_value	parameter_type_code	creation_datetime
1	34	PARTITION_RETENTION_DAYS	14	FW	2021-11-29 13:33:17
2	41	PARTITION_RETENTION_DAYS	14	FW	2021-11-29 13:33:17
3	50	PARTITION_RETENTION_DAYS	14	FW	2021-11-29 13:33:17
4	101	PARTITION_RETENTION_DAYS	30	FW	2021-11-29 13:33:17
5	102	PARTITION_RETENTION_DAYS	30	FW	2021-11-29 13:33:17
6	103	PARTITION_RETENTION_DAYS	30	FW	2021-11-29 13:33:17
7	21	PARTITION_RETENTION_DAYS	30	FW	2021-11-29 13:33:17
8	100	PARTITION_RETENTION_DAYS	30	FW	2021-11-29 13:33:17
9	10	PARTITION_RETENTION_DAYS	60	FW	2021-11-29 13:33:17

4. ábra: A HFW_EXTRACT_PARAMS_BY_INSTANCE_ID_V nézet globális, process és process instance szintű paraméterekkel

- HFW_TABLE_RETENTION_DAY_PARAMETER: Ez a nézet megadja az összes táblához a HFW_EXTRACT_PARAMETERS táblában található PARTITION_RETENTION_DAYS, ARCHIVE_AFTER_DAYS és EXTRACT_TABLE_RETENTION_DAYS paraméter értékeket, illetve az ARCHIVE_FL flag-et, amely azt adja meg, hogy archiválni kell-e az adott táblát vagy sem. Hasonlóan működik, mint a

HFW_EXTRACT_PARAMS_BY_INSTANCE_ID_V, tehát a paraméterek értékét a megfelelő szint szerint fogja visszaadni. Ez a nézet a karbantartó kód esetében lesz fontos, ezek a paraméterek adják meg, hogy a forrásfájlra épülő Hive external táblát, illetve a Hive tábla partícióit hány napig kell megőrizni, esetleg hány nap múlva kell azokat archiválni.

	table_name	table_schema	source_sysem_name	retention_days	archive_after_days	extract_retention_days	archive_fl
1	airbnb	airbnb_schema	test_src_system	30	30	30	N
2	BIRTH	BIRTH_schema	test_src_system	30	30	30	N
3	BUDGETFOOD	BUDGETFOOD_schema	test_src_system	30	30	30	N
4	chess	chess_schema	test_src_system	30	30	30	N
5	earthquakes	earthquakes_schema	test_src_system	30	30	30	N
6	online_retail	online_retail_schema	test_src_system	30	30	30	N
7	people	people_schema	test_src_system	30	30	30	N
8	retail	retail_schema	test_src_system	30	30	30	N

5. ábra: A HFW_TABLE_RETENTION_DAY_PARAMETER nézet

- HFW_ACTUAL_RUN_STATUS_V: Segéd nézet a monitorozáshoz, a HFW_RUN_CONTROL-ban található process instance-okról ad részletes információkat.

	instance_id	instance_script_name	process_name	effected_object	status
1	126	pyspark_script_people.py	pyspark_script.py	people.csv	RUNNING
2	124	pyspark_script_BUDGETFOOD.py	pyspark_script.py	BUDGETFOOD.txt	RUNNING

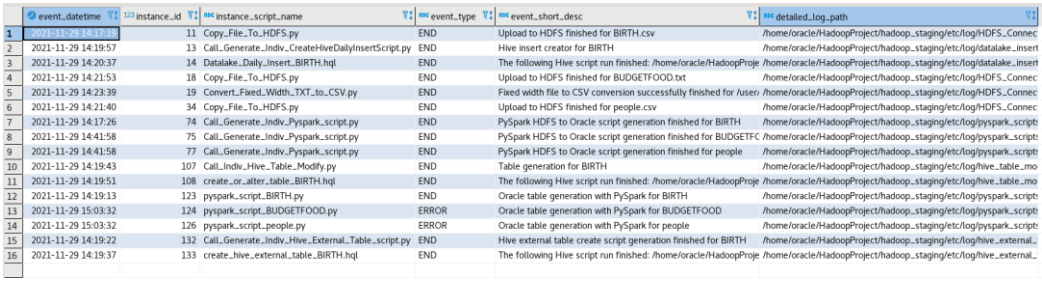
6. ábra: A HFW_ACTUAL_RUN_STATUS_V nézet

- HFW_PROC_INSTANCE_DEPENDENCY_V: Segéd nézet, a HFW_PROC_INSTANCE_DEPENDENCY táblában megtalálható függőségeket adja meg olvasható formában. Ezek a függőségek adják meg, hogy az egyes process instance-ok mely más process instance-okat hívnak meg, miután lefutottak.

	instance_script_name	dependent_instance_script_name	effected_object
42	Process_Landing_Zone_File.py	Copy_File_To_HDFS.py	airbnb.txt
43	Copy_File_To_HDFS.py	Convert_Fixed_Width_TXT_to_CSV.py	airbnb.txt
44	Convert_Fixed_Width_TXT_to_CSV.py	Call_Generate_Indiv_Pyspark_script.py	airbnb.txt
45	Call_Generate_Indiv_Pyspark_script.py	pyspark_script_airbnb.py	airbnb.txt
46	pyspark_script_airbnb.py	Call_Generate_Indiv_Hive_External_Table_script.py	airbnb.txt
47	Call_Generate_Indiv_Hive_External_Table_script.py	create_hive_external_table_airbnb.hql	airbnb.txt
48	create_hive_external_table_airbnb.hql	Call_Indiv_Hive_Table_Modify.py	airbnb.txt
49	Call_Indiv_Hive_Table_Modify.py	create_or_alter_table_airbnb.hql	airbnb_schema.airbnb
50	create_or_alter_table_airbnb.hql	Call_Generate_Indiv_CreateHiveDailyInsertScript.py	airbnb_schema.airbnb
51	Call_Generate_Indiv_CreateHiveDailyInsertScript.py	Datalake_Daily_Insert_airbnb.hql	airbnb_schema.airbnb
52	Call_Generate_Indiv_Pyspark_hive_script.py	pyspark_to_oracle_hive_airbnb.py	airbnb.txt

7. ábra: A HFW_ACTUAL_RUN_STATUS_V nézet

- HFW_LATEST_PROCESS_STATUS_V: Segéd nézet, amely a HFW_LOG táblából az egyes process instance-okhoz tartozó legfrissebb logokat adja vissza.



	event_datetime	instance_id	instance_script_name	event_type	event_short_desc	detailed_log_path
1	2021-11-29 14:17:18	11	Copy_File_To_HDFS.py	END	Upload to HDFS finished for BIRTH.csv	/home/oracle/HadoopProject/hadoop_staging/etc/log/HDFS_Connec
2	2021-11-29 14:19:57	13	Call_Generate_Indiv_CreateHiveDailyInsertScript.py	END	Hive insert creator for BIRTH	/home/oracle/HadoopProject/hadoop_staging/etc/log/datalake_insert
3	2021-11-29 14:20:37	14	Datalake_Daily_Insert_BIRTH.hql	END	The following Hive script run finished: /home/oracle/HadoopProje	/home/oracle/HadoopProject/hadoop_staging/etc/log/datalake_insert
4	2021-11-29 14:21:53	18	Copy_File_To_HDFS.py	END	Upload to HDFS finished for BUDGETFOOD.txt	/home/oracle/HadoopProject/hadoop_staging/etc/log/HDFS_Connec
5	2021-11-29 14:23:39	19	Convert_Fixed_Width_TXT_to_CSV.py	END	Fixed width file to CSV conversion successfully finished for /user/	/home/oracle/HadoopProject/hadoop_staging/etc/log/HDFS_Connec
6	2021-11-29 14:21:40	34	Copy_File_To_HDFS.py	END	Upload to HDFS finished for people.csv	/home/oracle/HadoopProject/hadoop_staging/etc/log/HDFS_Connec
7	2021-11-29 14:17:26	74	Call_Generate_Indiv_Pyspark_script.py	END	PySpark HDFS to Oracle script generation finished for BIRTH	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
8	2021-11-29 14:41:58	75	Call_Generate_Indiv_Pyspark_script.py	END	PySpark HDFS to Oracle script generation finished for BUDGETFC	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
9	2021-11-29 14:41:58	77	Call_Generate_Indiv_Pyspark_script.py	END	PySpark HDFS to Oracle script generation finished for people	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
10	2021-11-29 14:19:43	107	Call_Indiv_Hive_Table_Modify.py	END	Table generation for BIRTH	/home/oracle/HadoopProject/hadoop_staging/etc/log/hive_table_mo
11	2021-11-29 14:19:51	108	create_or_alter_table_BIRTH.hql	END	The following Hive script run finished: /home/oracle/HadoopProje	/home/oracle/HadoopProject/hadoop_staging/etc/log/hive_table_mo
12	2021-11-29 14:19:13	123	pyspark_script_BIRTH.py	END	Oracle table generation with PySpark for BIRTH	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
13	2021-11-29 15:03:32	124	pyspark_script_BUDGETFOOD.py	ERROR	Oracle table generation with PySpark for BUDGETFOOD	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
14	2021-11-29 15:03:32	126	pyspark_script_people.py	ERROR	Oracle table generation with PySpark for people	/home/oracle/HadoopProject/hadoop_staging/etc/log/pyspark_script
15	2021-11-29 14:19:22	132	Call_Generate_Indiv_Hive_External_Table_script.py	END	Hive external table create script generation finished for BIRTH	/home/oracle/HadoopProject/hadoop_staging/etc/log/hive_external_
16	2021-11-29 14:19:37	133	create_hive_external_table_BIRTH.hql	END	The following Hive script run finished: /home/oracle/HadoopProje	/home/oracle/HadoopProject/hadoop_staging/etc/log/hive_external_

8. ábra: A HFW_LATEST_PROCESS_STATUS_V nézet

A HFW_EXTRACT_PARAMETERS táblába a következő paramétereket vettem fel, mindegyik globális szintű:

- CALENDAR_DATE: Az effektív dátum, az adatok eszerint lesznek particionálva.
- ARCHIVE_COMPRESS_CODEC: Azt adja meg, hogy amennyiben egy táblánál be van kapcsolva az archiválás, akkor az milyen tömörítési algoritmust használjon.
- PARTITION_RETENTION_DAYS: Azt adja meg, hogy hány napig legyenek eltárolva a Hive táblák CALENDAR_DATE szerint felosztott partíciói.
- ARCHIVE_AFTER_DAYS: Azt adja meg, hogy hány nap után legyenek archiválva a Hive táblák CALENDAR_DATE szerint felosztott partíciói.
- EXTRACT_TABLE_RETENTION_DAYS: Azt adja meg, hogy hány napig legyenek eltárolva a forrásfájlra épülő Hive external táblák.
- url: PySpark kód generálásához használt paraméter, az Oracle szerver elérési url-je.
- driver: PySpark kód generálásához használt paraméter, az Oracle-hez használt driver.
- user: PySpark kód generálásához használt paraméter, az Oracle szerverbe való belépéshez szükséges user neve.
- password: PySpark kód generálásához használt paraméter, az Oracle szerverbe való belépéshez szükséges jelszó.
- batchsize: PySpark kód generálásához használt paraméter, a batch-ek mérete, amit a Spark használni fog Oracle-ba való adatbetöltéskor.

6.2. A töltési folyamat

6.2.1. A metaadatbázis elérése

A metaadatbázist a legtöbb kódgeneráló illetve töltést megvalósító scriptnek el kell érnie. Ehhez létrehoztam a DatabaseFunctions.py fájlt és azon belül a Connection osztályt, ami képes Postgres adatbázisokhoz kapcsolódni. Bemenő paraméternek egy séma nevet vár el, ami alapján felolvassa az adatbázis kapcsolódási beállításokat a megfelelő konfigurációs fájlból, majd létrehozza a kapcsolatot. Ezzel az osztállyal lehet elérni a Hive metastore-ját is, ami szintén egy Postgres adatbázis. A Postgres eléréséhez a psycopg2 nevű Python modult használtam. A psycopg2 egy adatbázis driver, ami a Python DB API implementációja, és egy wrapper a libpq köré. A libpq egy C könyvtár, amivel lekérdezéseket lehet küldeni Postgres adatbázisok felé, a Python DB API pedig annak a specifikációja, hogy az adatbázisokat elérő Python modulokat hogyan érdemes implementálni. [31] [32] [33]

A metaadatbázis esetén a konfigurációs fájl neve hfw_metadata.conf. Ez a fájl tartalmazza az adatbázis nevét, elérési címét, illetve a belépéshez szükséges felhasználónevet és jelszót. Ez a fájl tartalmaz még további konfigurációkat is, amelyek a Hive adatbázishoz való kapcsolódáshoz szükségesek, illetve bizonyos, az alkalmazás működéséhez szükséges konfigurációkat is, mint például a HDFS elérési címét.

A Connection osztály sok metódussal rendelkezik, ezek közül a következőek a legtöbbet használtak és a legfontosabbak:

- select_from_database: Ezzel a metódussal lekérdezéseket küldhetünk az adatbázis felé, a lekérdezések eredményét a metódus egy pandas dataframe-ben adja vissza. A metódus a lekérdezéseket úgy írja meg, hogy a metaadatbázisban inaktívnak jelölt rekordokat alapból kiszűri.

- get_instance_by_process: Ez a metódus egy vagy több, a megadott process-hez tartozó process instance-ok id-jait adja vissza. Lehetőség van fájl név alapján szűrni, ha csak egy adott fájlhoz tartozó process instance id-ra van szükségünk. A process instance id azonosítja be a futtatott kódot.

- execute_dml_statement: Ez a metódus bemenő paraméternek egy DML utasítást vár, majd azt futtatja meg az adatbázisban. Visszatérési értéke azt jelzi, hogy sikeresen le futott-e az utasítás vagy sem.

- cleanup_process_instance: Ezt a metódust minden process instance meghívja miután lefutott. Paraméternek a process instance id-ját várja el. Az adott process instance-ot törli a HFW_RUN_CONTROL-ból, majd amennyiben létezik ettől a process instance-tól függő másik process instance a HFW_PROC_INSTANCE_DEPENDENCY táblában, azt beírja a HFW_RUN_CONTROL-ba.

- get_executable_processes: Ez a metódus lekérdezi a HFW_RUN_CONTROL tartalmát prioritás és beillesztés dátuma szerint sorba rendezve.

6.2.2. Segéd- és ősosztályok

Létrehoztam egy segédfájlt Utils.py néven. Ebben található a directory_check segédmetódus, ami létrehozza a bemenő paraméterként megadott mappát, amennyiben az még nem létezik, illetve a Logger osztály, amely a logolást valósítja meg. A Logger osztály bemenő paraméternek egy mappa nevet, egy process nevet és a logolás módját várja el. A logfájlok a megadott mappában fognak létrejönni, és a process nevéről lesznek elnevezve. A logolás módja lehet „NORMAL”, „HOURLY” vagy „DAILY”, ez azt adja meg, hogy az alkalmazás milyen intervallummal hozzon létre új logfájlt.

A process-eknek létrehoztam egy ősosztályt, a BaseProcess-t, minden process ennek a leszármazottja. Ez az ősosztály bemenő paraméterként a process almappájának a nevét, a process nevét és a logolás módját várja el. Az ősosztály felveszi változókba a HFW_PATH környezeti változó alapján az alkalmazás elérési útját, a megadott almappa elérési útját és a template fájlok elérési útját. Továbbá példányosítja a Logger osztályt és elindítja a logolást, illetve példányosítja a DatabaseFunctions.py-ban található Connection osztályt a metaadatbázis eléréséhez.

Szinte minden process-hez létezik egy “call” script. Ezek a call scriptek felelősek azért, hogy a szükséges process osztályokat példányosítsák, illetve, hogy meghívják a megfelelő metódusokat. Továbbá a call scriptek felelnek azért is, hogy lekérdezzék a metaadatbázisból a megfelelő process instance id-t, ez azonosítja be, hogy melyik fájlra fut a script. A process instance id-hoz tartozó process neve általában megegyezik a call script nevével.

6.2.3. Process-ek automatikus futtatása

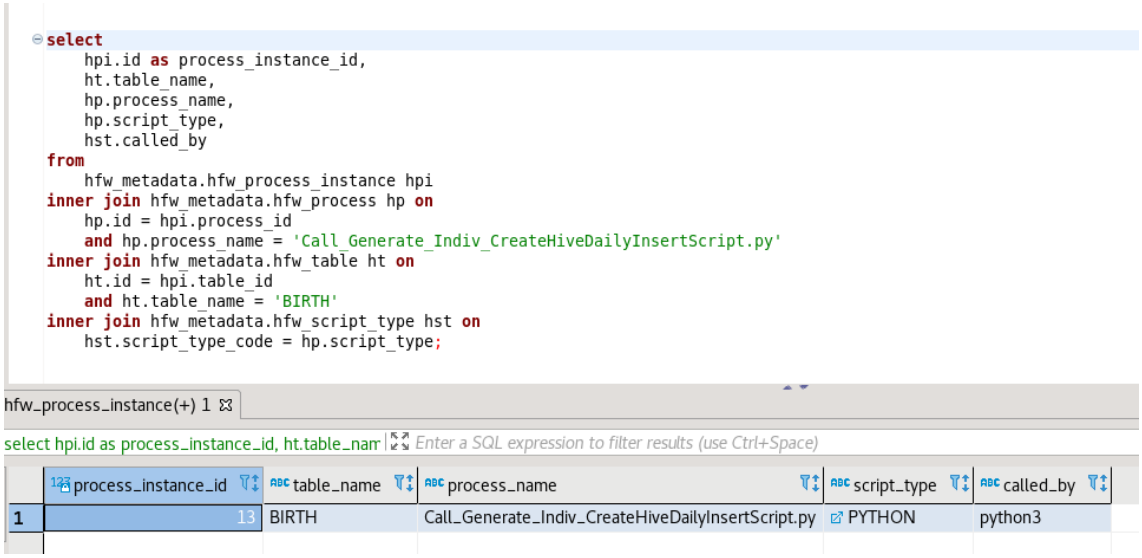
Minden futtatandó process a HFW_RUN_CONTROL táblába kerül be, a process instance id-jával együtt. Létrehoztam egy bash scriptet, a Monitor_Run_Control.sh-t, amely a hfw_metadata.conf-ban található run_control_check_interval szerinti időközönként meghívja a Call_Process_Run_Control.py-t. A Call_Process_Run_Control.py példányosítja a Process_Run_Control.py-ban megtalálható Process_Run_Control osztályt, és meghívja a RunProcesses metódusát.

A RunProcesses metódus lekérdezi a HFW_RUN_CONTROL tábla tartalmát a Connection osztályban található get_executable_processes metódus segítségével, és elindít megadott számú futtatható process-t. Azok a process-ek számítanak futtathatónak, amelyek nem „RUNNING” státuszban vannak. Az, hogy a metódus egyszerre hány process-t indíthat el, a hfw_metadata.conf-ban található parallel_process_number

határozza meg. A konfigurációs fájl a Process_Run_Control osztály példányosításakor kerül felolvasásra, tehát ha a parallel_process_number-t megváltoztatjuk, az a run_control_check_interval szerinti következő futáskor érvényesülni fog. A HFW_RUN_CONTROL-ban található process-ek futási sorrendjét a process-ek prioritása, illetve a run control-ba való beillesztésüknek az ideje határozza meg. A process-ek prioritása process szinten a HFW_PROCESS tábla PRIORITY oszlopával, process instance szinten pedig a HFW_PROCESS_INSTANCE tábla PRIORITY oszlopával változtatható.

A process-ek futtatására a Python subprocess nevű modulját használtam. Azt, hogy egy process milyen módon hívódik meg, a HFW_PROCESS tábla SCRIPT_TYPE oszlop értéke, és a HFW_SCRIPT_TYPE tábla ahhoz tartozó CALLED_BY oszlop értéke határozza meg. Argumentumnak mindig meg van adva a process instance id, a meghívott process-ek ez alapján tudják megállapítani, hogy melyik fájlra vagy táblára kell lefutniuk, vagy azt, hogy melyik scriptet kell lefuttatniuk.

Például a „BIRTH” táblához a Hive INSERT kódokat generáló process a következőképpen hívódik meg: A HFW_PROCESS táblában a process neve Call_Generate_Indiv_CreateHiveDailyInsertScript.py, a SCRIPT_TYPE oszlop értéke pedig „PYTHON”. A HFW_SCRIPT_TPYE táblában a „PYTHON” SCRIPT_TYPE_CODE oszlophoz tartozó CALLED_BY oszlop értéke „python3”. A HFW_PROCESS_INSTANCE táblában a Call_Generate_Indiv_CreateHiveDailyInsertScript.py process „BIRTH” táblára vonatkozó process instance id-ja 13. A RunProcess metódus a következő parancsot fogja meghívni subprocess segítségével: python3 Call_Generate_Indiv_CreateHiveDailyInsertScript.py 13. Végül a Call_Generate_Indiv_CreateHiveDailyInsertScript.py az argumentumban megkapott process instance id alapján legenerálja a Hive INSERT kódokat a „BIRTH” táblához.



9. ábra: A „BIRTH” táblához a Hive INSERT kódokat generáló process meghívásához szükséges adatok

6.2.4. Fájlok betöltése HDFS-be

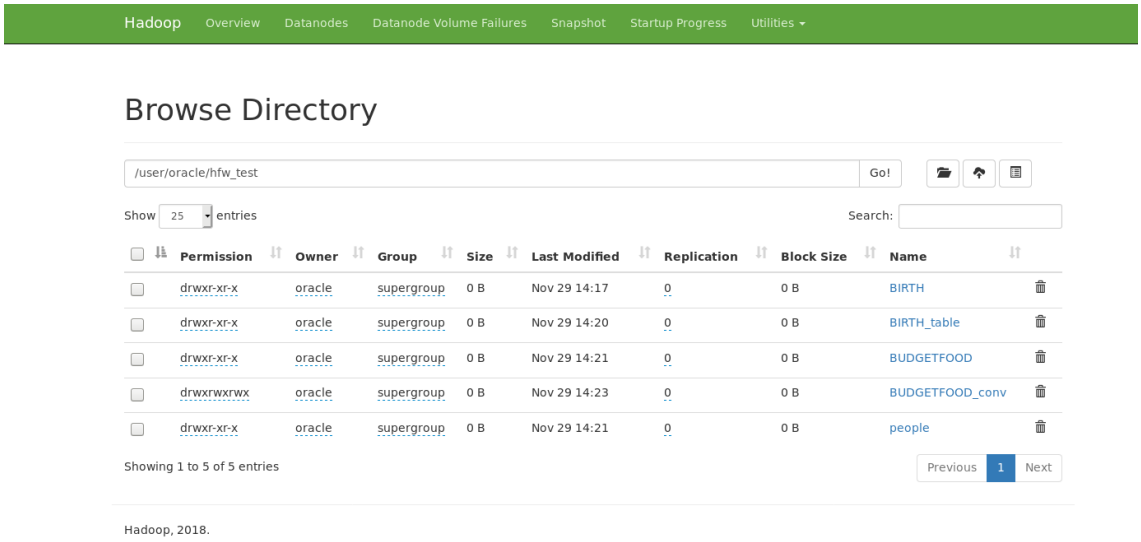
A fájlok egy megadott mappába érkeznek, ennek a mappának a monitorozásához létrehoztam egy bash scriptet Monitor_Landing_Zone.sh néven. Ez a script az inotifywait-et használja, hogy figyelje a megadott mappát, és amennyiben létrejön ott egy fájl, azt átmásolja egy átmeneti mappába, majd meghívja a Process_Landing_Zone_File.py kódot, amelynek argumentumként átadja a létrejött fájl nevét. Az inotifywait egy Unix segédprogram, amely képes monitorozni a fájlrendszert és jelezni a változásokat. [34]

A Process_Landing_Zone_File.py a megérkezett fájlhoz a HFW_SOURCE_FILE táblába felvesz egy új rekordot a fájl eredeti és aktuális nevével, illetve a HFW_ACT_LOADED_SOURCES táblában az adott fájlhoz beállítja a HFW_SOURCE_FILE-ba felvett új rekord id-ját, hogy meg lehessen nézni, hogy melyik forrásfájl van éppen betöltve Oracle-ba. A fájl eredeti neve abban az esetben különbözik az aktuális nevétől, ha az eredeti neve egy dátummal végződik. Ebben az esetben az alkalmazás a fájlt átnevezi úgy, hogy a nevéből levágja a dátumot. Ezek után beíródik a következő függő process instance a HFW_RUN_CONTROL-ba, ami a Copy_File_To_HDFS.py. Jelenleg az alkalmazás csak olyan fájlokat tud betölteni, amelyeknek a formátuma például “fájlnév.txt” vagy “fájlnév_yyyymmdd.txt”, más dátum formátumokat nem képes feldolgozni.

A Copy_File_To_HDFS.py példányosítja a HDFS_Functions.py HDFS_Connection osztályát, majd meghívja a copy_file_to_hdfs metódust és paraméternek átadja a process instance id-ját. A HDFS_Connection osztály felelős azért, hogy elérje a HDFS-t, a HDFS elérési címe a hfw_metadata.conf-ban található. A copy_file_to_hdfs metódus a process

instance id alapján lekérdezi a HFW_FILE táblából a fájl nevét és a HDFS_STORAGE_PATH oszlopot, ami a fájl elérési útja lesz HDFS-en, majd ezek alapján betölti a fájlt HDFS-be. A HDFS eléréséhez a hdfs nevű Python modul InsecureClient osztályát használtam.

A HDFS_Connection osztályban létrehoztam még egy convert_fwf_to_csv nevű metódust. Amennyiben a fájl fixed width TXT fájl, abban az esetben a Convert_Fixed_Width_TXT_to_CSV.py fut le a következő lépésben, amely ezt a metódust hívja meg, egyébként pedig a Call_Generate_Indiv_Pyspark_script.py. Az alkalmazás a fájl formátumát a HFW_FILE tábla FILE_FORMAT oszlopából tudja megállapítani. A convert_fwf_to_csv metódus átkonvertálja a TXT fájlokat CSV-be, a konvertálás már HDFS-en történik. Az átkonvertált fájlok új mappákba kerülnek, ezeket a mappákat és az átkonvertált fájlokat a „_conv" végződés jelzi, az eredeti fájl megmarad a helyén. Ahhoz, hogy a konverzió sikeres legyen, a TXT fájlokhoz tartozó oszlopoknak fel kell venni a kezdő- és végpozíciójukat a HFW_FILE_COLUMNS táblában a START_POSITION és END_POSITION mezőkbe, az alkalmazás ez alapján tudja kiszámolni az oszlopok szélességét. Továbbá fontos az oszlopok sorrendjének is a megadása a COLUMN_POSITION mezőben. Az átkonvertált fájlokban az oszlopok mindig pontosvesszővel lesznek elválasztva. A fájlok HDFS-be való betöltéséért és TXT-ből CSV-be való átkonvertálásáért felelős kódok megtalálhatóak a mellékletben.



10. ábra: Fájlok a HDFS-en betöltést követően

6.2.5. Adatok betöltése HDFS-ből Oracle-ba

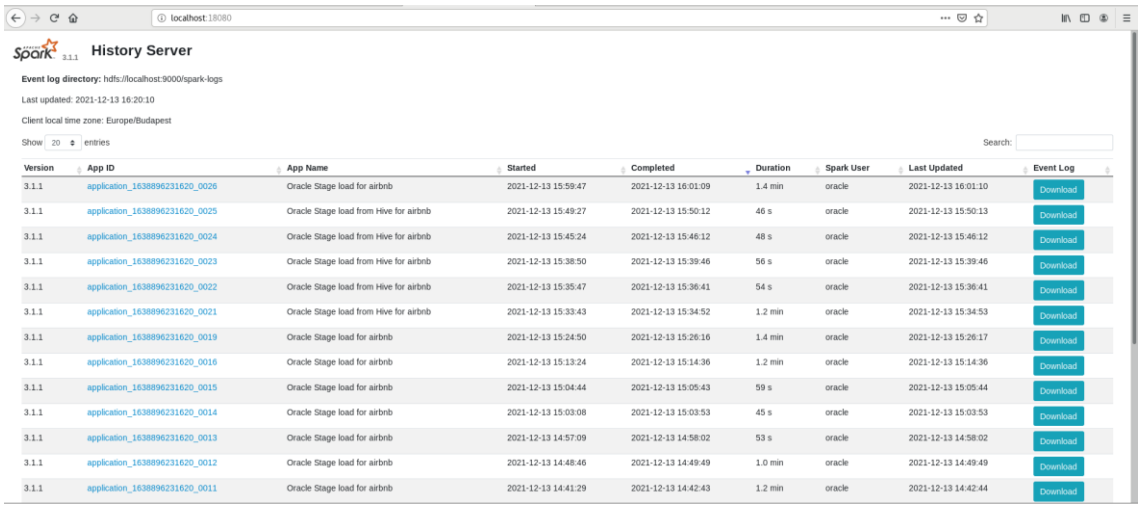
A PySpark kódok generálásához létrehoztam kettő template fájlt, pyspark_to_oracle.py, és pyspark_to_oracle_hive.py néven. A pyspark_to_oracle.py HDFS-ből Oracle-ba

töltésért, a pyspark_to_oracle_hive.py pedig Hive-ból Oracle-ba egy megadott nap betöltéséért felelős.

A PySpark kódok generálásához létrehoztam a Generate_Pyspark_script.py-t és azon belül a PySparkScriptGenerator osztályt. A Call_Generate_Indiv_Pyspark_script.py ezt az osztályt példányosítja, majd meghívja a CreateHdfsToOracleScript metódusát, aminek egy fájlnevet, fájl id-t és egy process instance id-t ad át.

A CreateHdfsToOracleScript metódus létrehozza a pyspark_to_oracle_fájlnév.py scriptet a pyspark_to_oracle.py template alapján, ez az, ami a generálás befejezését követően le fog futni. A template fájlokban behelyettesítendő kódrészletek a következőképpen vannak jelezve: <<BEHELYETTESÍTENDŐ>>. Az alkalmazás az ehhez hasonló formátumú string-ek helyére fogja behelyettesíteni a megfelelő értékeket a Python replace metódusának segítségével. Ez a metódus első bemenő paraméternek a keresett string-et várja el, második bemenő paraméternek pedig azt, amivel helyettesíteni szeretnénk. A kódgeneráló script beolvassa az egész template fájlt, majd egymás után meghívja a replace metódust az összes behelyettesítendő értékre. A pyspark_to_oracle.py template kódja megtalálható a mellékletben.

A generált script létrehoz egy SparkSession-t és egy új Spark applikációt a betöltendő fájl nevével, kiolvassa a fájl tartalmát HDSF-ből, majd nyit egy Oracle kapcsolatot és a fájl tartalmát beírja a fájl nevével megegyező Oracle táblába. Az Oracle táblából az adatok a beillesztést megelőzően törlődnek, így mindig csak egy napi adat lesz eltárolva benne. A generált PySpark scriptek a spark-submit paranccsal kerülnek futtatásra.



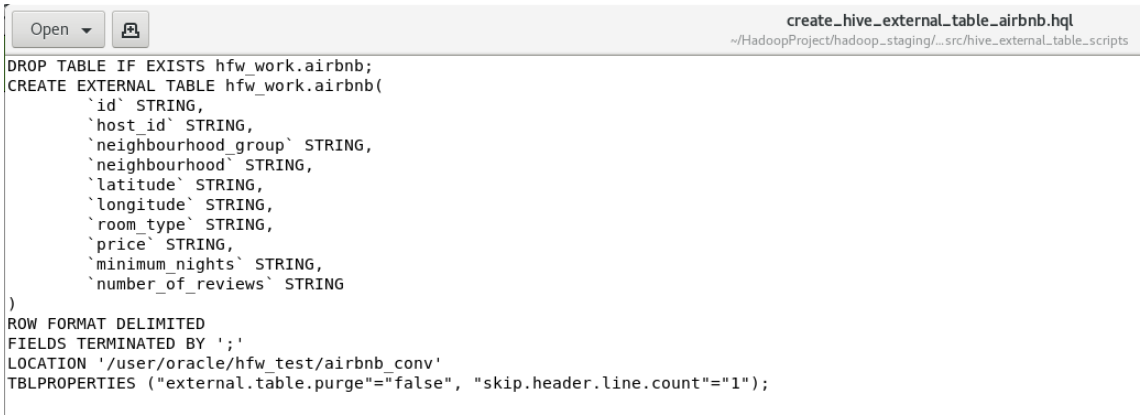
11. ábra: A Spark webes felülete

Az Oracle-ba az adatokat az alkalmazás csak a stage, vagy más néven extract rétegig tölti, onnantól a további ETL folyamatok az Oracle rendszer felelőssége. Miután az adatok

betöltődtek, bekerül a run control-ba a következő lépés, ami a Hive external táblalétrehozó kódok generálása.

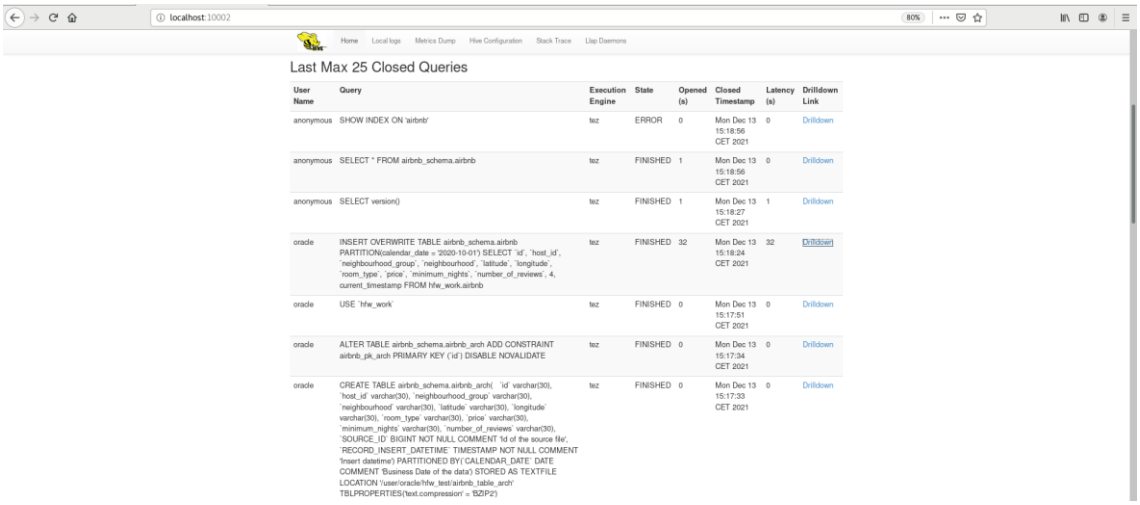
6.2.6. Adatok betöltése Hive-ba

Az adatok Hive-ba való betöltése több lépésből áll. Először létre kell hozni egy Hive external táblát, ami HDFS-en a betöltendő fájlra mutat. Ez azért szükséges, hogy a managed Hive táblákba való töltést megvalósító INSERT kódok generálása egyszerűbb legyen. Az external táblák az adatokat már egységes formában fogják tárolni, elrejtve a CSV fájlok sajátosságait, mint például, hogy milyen karakter tagolja fel az oszlopokat. Ehhez létrehoztam a Generate_Hive_External_Table_script.py-t, és azon belül a HiveExternalTableScriptGenerator osztályt. Ennek az osztálynak a GenerateHiveExternalTable metódusa felel a Hive external táblalétrehozó HiveQL kódok generálásáért, bemenő paraméternek a fájl nevét, id-ját, és a process instance id-ját várja el. A metódus a HiveQL scriptet a HFW_FILE táblában található paraméterek és a HFW_FILE_COLUMNS táblában lévő rekordok alapján hozza létre, create_hive_external_table_fájlnév.hql néven. A fájlokhoz a következő opcionális paramétereket adhatjuk meg a HFW_FILE táblában. A LINE_TERMINATOR oszlop meghatározza, hogy a fájl sorainak a végét milyen karakter jelzi, az ESCAPE_CHARACTER oszlopban a szöveges mezők feloldójelét adhatjuk meg, a NULL_VALUE oszlopban pedig azt adhatjuk meg, hogy a fájlban milyen érték felel meg NULL-nak. A fájlokhoz tartoznak kötelező paraméterek is, ezek a következőek: A HEADER_FL flag, ami azt adja meg, hogy a fájlnak van-e fejléce, és a FIELD_TERMINATOR, ami a CSV fájl oszlopait elválasztó karaktert adja meg, ez TXT-ből CSV-be átkonvertált fájl esetén mindig pontosvessző. A fájl elérési útját a metódus a HDFS_STORAGE_PATH oszlopból szedi ki, átkonvertált fájl esetén pedig automatikusan hozzáteszi a „_conv” végződést. Végül a létrejött HiveQL script beíródik a run control-ba



12. ábra: Egy generált Hive external táblalétrehozó kód

A HiveQL kódok futtatásához írtam egy HiveWrapper osztályt, ez bemenő paraméterként a futtatandó script nevét és process instance id-ját várja el. A HiveWrapper nyit egy kapcsolatot Hive felé a Python pyhive nevű moduljával, majd a kapott scriptet feldarabolja külön utasításokra. A wrapper a feldarabolt utasításokat egyesével küldi el Hive felé és mindegyiknek logolja a futását. A lefuttatott kódokat a Hive webes felületén is tudjuk monitorozni. Amennyiben mindegyik utasítás sikeresen lefutott, a wrapper a paraméterként megkapott process instance id alapján beírja a run control-ba a következő lépést, ami a managed Hive táblalétrehozó kódok generálása.



13. ábra: A Hive webes felülete

A managed Hive táblalétrehozó kódok generálása talán a legbonyolultabb része az alkalmazásnak, hiszen sok dologra kell odafigyelni. A kódgenerálást a Hive_Table_Modify.py-ban található Hive_Table_Modify osztály végzi. Az osztálynak a metaadatbázis mellett nyitnia kell egy kapcsolatot a Hive metastore-ja felé, innen tudja lekérdezni, hogy milyen Hive táblák léteznek, és, hogy azok milyen oszlopokkal és megszorításokkal rendelkeznek. Ezek az adatok szükségesek ahhoz, hogy a generált CREATE TABLE és ALTER TABLE kódok pontosak legyenek.

A táblalétrehozó kód generálását a GenerateTable metódus végzi, bemenő paraméternek a tábla nevét, sémáját és egy process instance id-t vár el. A metódus első lépésben megnézi a Hive metastore-ban, hogy létezik-e már az adott tábla. Ha nem, akkor a HFW_TABLE táblában található adatok alapján legenerálja a táblalétrehozó kódot. A táblához tartozó oszlopokat a HFW_TABLE_COLUMN, a megszorításokat a HFW_TABLE_CONSTRAINT táblákban megtalálható metaadatok alapján hozza létre az alkalmazás. Minden tábla legalább a CALENDAR_DATE oszlop által particionálva van, ez az adatok effektív dátuma. Jelenleg csak elsődleges és idegen kulcs megszorítások felvételére van lehetőség. A generált kódot a metódus a create_or_alter_table_táblanév.hql fájlba írja bele.

Ha az adott tábla már létezik, akkor meghívódik a GenerateAlterScript metódus. Ez a metódus összehasonlítja a Hive metastore adatait a metaadatbázis adataival, és amennyiben eltérést talál az oszlopok vagy a megszorítások között, létrehozza a szükséges ALTER TABLE kódokat. Végül a generált ALTER kód beíródik a create_or_alter_table_táblanév.hql fájlba.

Amennyiben egy táblára be van kapcsolva az archiválás, abban az esetben a kódgeneráló a metastore-ban az archív tábla létezését is megnézi, és szükség esetén legenerálja az azt létrehozó kódot is a create_or_alter_table_táblanév.hql fájlba, a többi generált kód alá. A tömörítés algoritmusát a HFW_EXTRACT_PARAMETERS táblában található ARCHIVE_COMPRESS_CODEC paraméter értéke adja meg. Alapból ennek a paraméternek az értékét BZIP2-re állítottam, mivel ez jobban betömöríti az adatokat, mint a GZIP, Snappy vagy LZO. A kitömörítési sebessége lassabb, de mivel az archivált táblákba csak viszonylag régebbi adatok kerülnek be, ezért ezeknek a kitömörítésére kevesebb eséllyel lesz szükség.

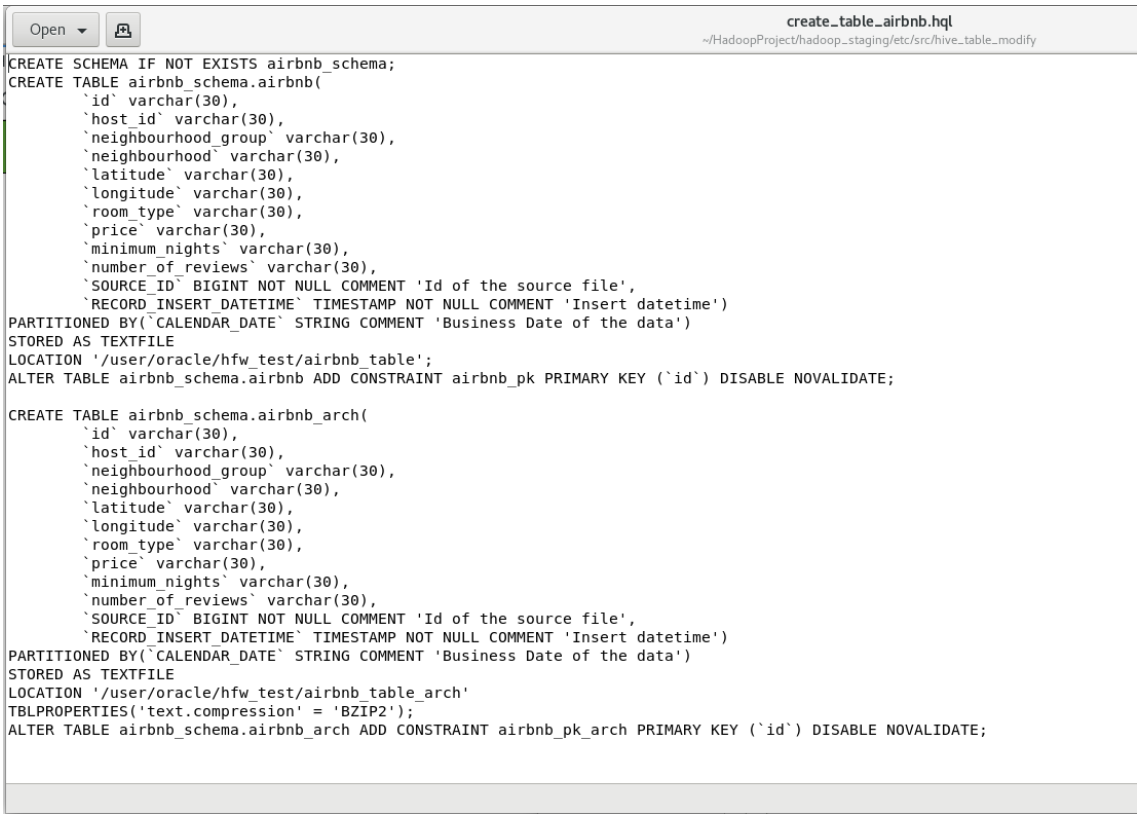
Table 5-1 Hadoop Codecs

Codec	*File Extension*	*Splittable?*	*Degree of Compression*	*Compression Speed*
Gzip	`.gz`	No	Medium	Medium
Bzip2	`.bz2`	Yes	High	Slow
Snappy	`.snappy`	No	Medium	Fast
LZO	`.lzo`	No, unless indexed	Medium	Fast

14. ábra: Tömörítési formátumokat összehasonlító táblázat [35]

A create_or_alter_table_táblanév.hql fájlba mindig csak azok a kódok kerülnek, amiknek le kell futniuk, tehát ha a táblában nem történt változtatás, akkor a fájl üres lesz. Emiatt az alkalmazás külön eltárolja a generált táblalétrehozó kódot a create_table_táblanév.hql nevű fájlba. Ez nem kerül automatikusan futtatásra, csak arra szolgál, hogy a generált kódok el legyenek tárolva és azokat utólag is vissza lehessen nézni.

Végül a generált HiveQL kódokat a Hive wrapper megfuttatja, majd beírja a run control-ba a következő lépést, ami a Call_Generate_Indiv_CreateHiveDailyInsertScript.py.



15. ábra: Egy generált managed Hive táblalétrehozó kód, archív táblával együtt

A Call_Generate_Indiv_CreateHiveDailyInsertScript.py példányosítja a CreateHiveDailyInsertScripts.py-ban található InsertCreator osztályt, és az osztály CreateInsertScript metódusa létrehozza a Hive táblákba való betöltéshez szükséges INSERT kódokat. A generált kódok a korábban létrehozott Hive external táblákból olvassák ki az adatokat, majd írják azokat be a managed Hive táblákba. A managed Hive táblák az effektív dátum alapján particionálva vannak, a beillesztendő adatok effektív dátumát az alkalmazás a HFW_EXTRACT_PARAMS_BY_INSTANCE_ID_V nézet CALENDAR_DATE paraméteréből olvassa ki, ami mindig az aktuális nap, kivéve, ha egy régebbi napot újratöltünk. A HFW_TABLE_FILE_RELATION táblában a tábláknak opcionálisan megadhatunk egy COMPLEX_TRANSFORMATION_SCRIPT_PATH értéket. Amennyiben a táblához a SIMPLE_TRANSFORMATION_FL flag értéke hamisra van állítva, abban az esetben az INSERT kód generálása helyett a generált fájlba a COMPLEX_TRANSFORMATION_SCRIPT_PATH-ban megadott fájl tartalma kerül beírásra. Erre például akkor van szükség, ha az adatot nem nyersen akarjuk tárolni, hanem valamilyen transzformációt akarunk rajta végezni. A generált kód neve Datalake_Daily_Insert_táblanév.hql, és a Hive wrapper futtatja meg. Ez a napi töltési folyamat utolsó lépése.

16. ábra: Egy generált Hive INSERT kód

6.2.7. Adatok betöltése Hive-ból Oracle-ba

Mivel az Oracle a felhasználói oldal, ezért szükséges volt azt megoldani, hogy lehessen onnan kezdeményezni adatbetöltést Hive-ból tetszőleges napra. Ez teszt környezetbe való betöltéshez fontos, illetve hibajavításnál, hogy meg lehessen nézni, hogy egy adott napra milyen adatok érkeztek a forrásfájlból Oracle-ba. Mivel az adatbetöltést a metaadatbázis irányítja, ezért az Oracle adatbázisnak valahogyan kommunikálnia kell a metaadatbázissal. Erre a célra létrehoztam egy új sémát a metaadatbázisban INTERFACES néven, illetve Oracle-ban egy új táblát INT_TABLE_COPY néven. Írtam egy bash scriptet Monitor_Oracle_Interfaces.sh néven, ami a hfw_metadata.conf-ban megadott időközönként meghívja a Call_Check_oracle_interfaces.py-t. A Call_Check_oracle_interfaces.py példányosítja a Check_oracle_interfaces osztályt, ami bejelentkezik Oracle-ba, majd meghívja a CheckOracleInterfaces metódusát, ami kiolvassa az INT_TABLE_COPY tábla tartalmát. Az INT_TABLE_COPY-ban azoknak az interface tábláknak a neve szerepel, amelyekben valamilyen változás történt, a változásokat az interface táblákra írt triggerek figyelik. Végül a CheckOracleInterfaces metódus átmásolja azoknak az interface tábláknak a tartalmát a metaadatbázisba, amelyek be vannak írva az INT_TABLE_COPY-ba, majd törli az INT_TABLE_COPY tartalmát. Így szinkronizálja az alkalmazás az interface táblákat Oracle és a metaadatbázis között.

Létrehoztam egy interface táblát INT_LOAD_PARTITION néven Oracle és metaadatbázis oldalon. Ebbe az interface táblába fel lehet venni azoknak a tábláknak a nevét, amiket Hive-ból be akarunk tölteni Oracle-ba. A táblák neve mellé azt is meg kell adni, hogy melyik napot szeretnénk betölteni. Az INT_LOAD_PARTITION táblára Oracle oldalon létrehoztam egy triggert ON_CHANGE_IN_INT_LOAD_PARTITION_TRIGGER néven, ami új rekord beillesztése után, vagy meglévő rekord módosítása vagy törlése után beírja az INT_LOAD_PARTITION tábla nevét az INT_TABLE_COPY-ba, ezzel jelezve, hogy ennek a táblának a tartalmát át kell másolni a metaadatbázisba. Továbbá metaadatbázis oldalon is létrehoztam egy triggert az INT_LOAD_PARTITION táblára, ami beillesztés után aktiválódik, és beírja a run control-ba a Call_Generate_Indiv_Pyspark_hive_script.py process-t a megfelelő táblára.

A Call_Generate_Indiv_Pyspark_hive_script.py példányosítja a PySparkScriptGenerator osztályt, és a CreateHiveToOracleScript metódus segítségével hasonlóan, mint a HDFS-ből való adat betöltéskor, létrehozza a Hive-ból Oracle-ba töltő PySpark scriptet a pyspark_to_oracle_hive.py template alapján. A CreateHiveToOracleScript egy fájlnevet, fájl id-t, és egy process instance id-t vár el bemenő paraméternek. A generált script neve

pyspark_to_oracle_hive_fájlnév.py lesz, ez fog lefutni a következő lépésben. A scriptben a SparkSession az enableHiveSupport konfiguráció segítségével olvassa ki az adatokat Hive-ból. Az enableHiveSupport kapcsolatot nyit a Hive metastore felé, elérhetővé teszi a Hive funkciókat a Spark számára, és lehetővé teszi, hogy a Spark lekérdezéseket küldjön Hive felé. [36]

6.3. Karbantartás

A Hive táblák karbantartásához létrehoztam a Maintain_Datalake_Partitions.py-t és azon belül a Datalake_Maintainer osztályt. A karbantartás nincs beütemezve, kézzel kell lefuttatni.

A karbantartó kód első lépésben a régi partíciókat dobja el. Lekérdezi a Hive metastore-ból egy adott táblához tartozó összes partíciót, majd a metaadatbázisból a HFW_TABLE_RETENTION_DAY_PARAMETER nézetből lekérdezi a táblához tartozó karbantartási paramétereket. Amennyiben egy partíció régebbi, mint RETENTION_DAYS paraméterben megadott érték, abban az esetben a partíció eldobásra kerül. Az alkalmazás az archív táblák partícióit is megnézi, és a RETENTION_DAYS alapján azoknak a partícióit is eldobja, amennyiben az szükséges. A partíciók a Hive DROP PARTITION parancsával kerülnek eldobásra, a partíciókat eldobó kód megtalálható a mellékletben.

Következő lépésben az alkalmazás archiválja azokat a táblákat, ahol az be van kapcsolva. Az archiválást a HFW_TABLE-ben az ARCHIVE_FL flag-nek az igazra állításával lehet bekapcsolni. Amennyiben az archiválás be van kapcsolva, és egy partíció régebbi, mint az ARCHIVE_AFTER_DAYS paraméterben megadott érték, abban az esetben az adott partíció a Hive-nak az EXCHANGE PARTITION parancsával áthelyezésre kerül az archív táblába. Az archív táblák nevét az „_arch” kifejezés jelzi a tábla neve után. Ezek a táblák az ARCHIVE_COMPRESS_CODEC paraméterben megadott tömörítési algoritmus alapján vannak tömörítve, tehát kevesebb helyet foglalnak, de lassabb lekérdezni őket. A Hive partíciók archiválást megvalósító kód szintén megtalálható a mellékletben.

Utolsó lépésként az alkalmazás azokat a Hive external táblákat dobja el, amelyek régebbiek az EXTRACT_RETENTION_DAYS paraméter értékénél. Azt, hogy egy Hive external tábla milyen régi, a metastore-ból olvassa ki az alkalmazás, ott el van tárolva a táblák létrehozási ideje.

6.4. Verziókezelés

A verziókezelés elengedhetetlen része a szoftverfejlesztésnek, ezt a GitHub-on keresztül végeztem, mivel ez a legnépszerűbb verziókezelő szolgáltatás. Az alkalmazás funkcióit

külön branch-eken fejlesztettem, majd miután elkészültek és leteszteltem őket, pull request-ekkel merge-eltem vissza azokat a master branch-re. A pull request-ek több commit-ot foglalnak magukba, és előnyük, hogy egy merge előtt át lehet nézni az összes változtatást, ami a cél branch-hez képest történt. [37]

7. Tesztelés

Az alkalmazást a fejlesztői számítógépemen teszteltem, egy Centos virtual machine-en. A virtual machine-t VirtualBox-on futtattam, és a következő szolgáltatások vannak feltelepítve rá:

- Hadoop 3.1.1
- Hive 3.1.2
- Oracle 12c
- Tez 0.9.2
- Spark 2.4.7
- Python 2.7.5
- PostgreSQL 12.6

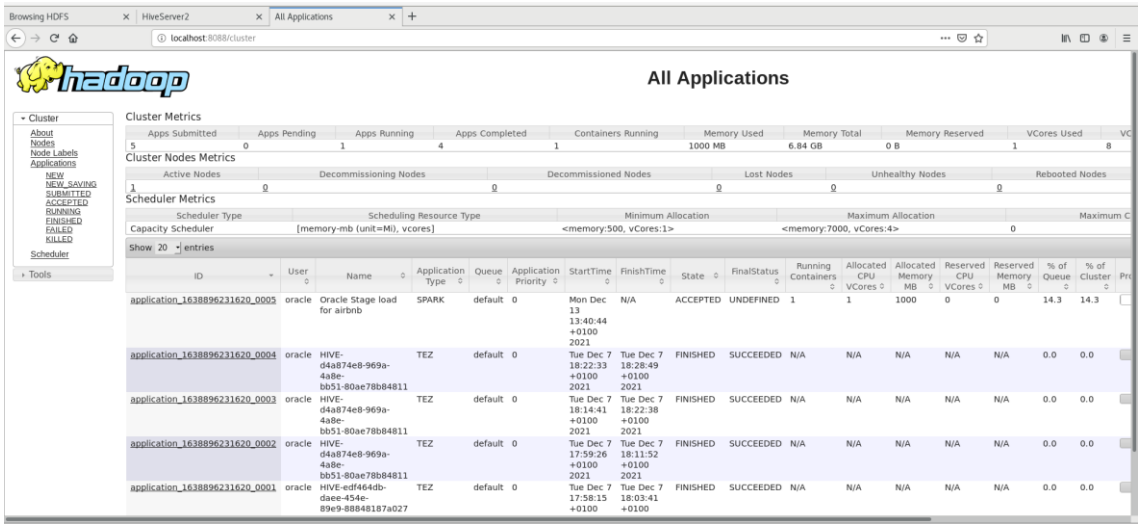
Fontosabb környezeti változók:

- HFW_PATH: A projekt elérési útja. A kódokban többször is hivatkozunk rá.
- PYTHONPATH: A HFW_PATH-on belül az a mappa, ahol a Python kódok vannak tárolva. Ez azért fontos, mert a Python itt fogja keresni a modulokat, így ha ez nincs rendesen beállítva, akkor a kódok nem tudnak hivatkozni egymásra.

Processzor	Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz 1.90GHz
Memória	8,00 GB
Virtual machine-nek szánt memória	5131 MB
Virtual machine-nek szánt CPU magok száma	4
Virtual machine operációs rendszere	Linux Red Hat (64-bit)

20. táblázat: A tesztelői számítógép és a virtual machine adatai

A teszteléshez szükséges teszt fájlokat, amiken a töltési folyamatokat teszteltem, a Kaggle-ről töltöttem le. 4-4 fixed width TXT és CSV fájllal teszteltem, az egyik TXT és CSV fájl dátum szerint több darabra volt felosztva. Adatbáziskezelőnek a DBeaver-t használtam, fejlesztőikörnyezetnek pedig a Visual Studio Code-ot. A töltési folyamatok monitorozásához a logokon és a metaadatbázison kívül a Hadoop webes felületeit használtam.



17. ábra: A Hadoop erőforráskezelőjének, a YARN-nak a webes felülete

A tesztelés előtt fel kellett vennem az összes szükséges metaadatot a fájlokhoz, mint például a fájlok és táblák oszlopai, illetve az összes process-t és a hozzájuk tartozó process instance-okat és az azok közötti függőségeket.

A metaadatok felvétele után a tesztelést a következő módon végeztem. Elindítottam a Monitor_Landing_Zone.sh bash scriptet, majd bemásoltam a forrásfájlokat a megadott landing zone mappába. A bash script automatikusan beírta a HFW_RUN_CONTROL táblába az első process-t. A Monitor_Run_Control.sh bash scriptet nem indítottam el, helyette a Call_Process_Run_Control.py-t kézzel futtatam le minden lépésben. Ez abban segített, hogy amikor valamelyik lépés hibára futott, akkor az alkalmazás nem próbálta meg magától újra lefuttatni a kódot feleslegesen, hanem meg tudtam keresni a hibát és ki tudtam javítani. Az előjövő hibákat a logfájlokból szedtem ki, a kódokban van kivételkezelés, de a Logger osztály képes logolni a nem elkapott kivételeket is.

Miután végigmentem egy fájlnak az összes töltési lépésén, leteszteltem más fájlokkal, illetve más paraméterekkel is az alkalmazást. Végül leteszteltem azokat a kódokat is, amelyek nem részei a napi töltési folyamatnak, például a Hive-ból Oracle-ba való töltést, vagy a karbantartó kódot. Összességében az alábbi követelményeknek a megvalósításait teszteltem le.

Fájlok betöltése HDFS-be	sikeres
TXT fájlok CSV-be konvertálása HDFS-en	sikeres
Hive external táblalétrehozó kódok generálása	sikeres
Managed Hive táblalétrehozó és módosító kódok generálása	sikeres
Hive INSERT kódok generálása	sikeres
HDFS-ből Oracle-ba töltő PySpark kódok generálása	sikeres
Hive-ból Oracle-ba töltő PySpark kódok generálása	sikeres
Tetszőleges napi adat betöltése Hive-ból Oracle-ba interface-k segítségével	sikeres
Hive táblák karbantartása	sikeres
Kódgenerálás és töltési folyamatok logolása	sikeres
Töltési folyamat lépéseinek automatikus futtatása	sikeres
Landing zone mappa monitorozása, töltési folyamat elindítása a beérkező fájlokkal	sikeres

21. táblázat: A letesztelt funkciók és azoknak eredményei

A tesztelés alatt előjöttek olyan problémák is, amelyek a tesztelésre használt számítógép miatt adódtak. A Spark kódok például sokszor elbuktak memóriahiány miatt, erre a megoldás több memória használata lenne. Ideális esetben a Hadoop rendszerek nem egy node-on futnak, és nem ilyen kevés memóriával. A minimális hardver elvárások egy Hadoop alkalmazásnak a következőek: [38]

Processzor	2 - 2.5 GHz
Memória	16,00 GB
Node-ok száma	3

22. táblázat: Hadoop alkalmazások minimális hardver elvárásai

8. Elért eredmények

Az elkészült alkalmazás képes a bejövő fájlok feldolgozására, tárolására és karbantartására. A metaadatbázison keresztül paraméterek segítségével egyszerűen testreszabható az alkalmazás, globális és folyamat szinten is, a töltési folyamatok, illetve a fájlok és táblák tárolási helye könnyen módosíthatóak.

A fájlok betöltéséhez szükséges nagy mennyiségű PySpark illetve HiveQL kódok generálása a metaadatok alapján történik, ezzel megspórolva azt a sok munkaórát, ami ezeknek a kódoknak a kézzel való megírásához kell. A generált kódok eltárolódnak, így az alkalmazás átlátható, transzparens módon működik, az üzemeltetők könnyen vissza tudják nézni milyen kódok futottak le. A töltési és kódgenerálási lépések részletesen logolva vannak, ami lényegesen megkönnyíti a hibakezelést.

A töltési folyamatok lépései automatikusan lefutnak, amennyiben nem történik hiba, abban az esetben a napi töltés mindenféle beavatkozás nélkül működik.

Oracle oldalon a felhasználóknak interface-k segítségével lehetőségük van tetszőleges napi adat betöltésére Hive-ból, így ezeknek a felhasználóknak nem kell biztosítani hozzáférést a metaadatbázishoz.

Az alkalmazás szükség esetén viszonylag egyszerűen felskálázható több node hozzáadásával. A karbantartó kódok biztosítják, hogy a régi és már irrelevánssá vált adatok ne használjanak fel feleslegesen erőforrásokat.

Az alkalmazás kódját átlátható módon írtam meg, úgy, hogy az könnyen bővíthető legyen, ha új folyamatok fejlesztésére van szükség.

9. Továbbfejlesztési lehetőségek

Bár az alkalmazás képes ellátni a követelményekben megfogalmazott feladatokat, sok lehetőség van optimalizálni, javítani a működését.

Az egyik fejleszthető része az alkalmazásnak, az a folyamatok függőségének a rugalmasabbá tétele. A jelenlegi megvalósításban, ha egy folyamat függ egy másiktól, akkor az mindenképpen beírásra kerül a run control táblába, ahogy a másik befejeződik. Itt be lehetne vezetni elágazásokat, hiszen vannak olyan esetek, amikor egy folyamat

átugorható, vagy esetleg olyan, hogy egy folyamatra több különböző függőségi láncot szeretnénk felépíteni. Ez javítaná a kód újrafelhasználhatóságát.

Egy másik fejleszthető terület, az a fájlok és táblák elnevezésének a rugalmasabbá tétele. A mostani működés szerint az alkalmazás sokszor kénytelen egy fájl vagy tábla nevét használni, amikor adatokat kérdez le az adatbázisból, ez főleg a Hive metastore-nál jellemző. Itt nyilván akkor merülnek fel problémák, ha egy fájlnak és a hozzátartozó táblának eltérő neve van, vagy akkor, ha két fájlnak vagy táblának azonos a neve, bár valószínűleg ez az eset elég ritka ugyanazon a forrásrendszeren vagy sémán belül.

A kód olvashatóságát is lehetne javítani például type-ok vagy konstansok használatával. A Python alapvetően egy dinamikus nyelv, sokkal megengedőbb, mint például a C#, viszont ez néha azt eredményezi, hogy a kód nehezebben olvasható. Konstansokat olyan helyeken lehetne például használni, mint a logolás, ahol jelenleg a log típusát string-ként adjuk meg, konstansokkal itt ki lehetne küszöbölni az esetleges elírásokból eredő problémákat.

A hibakezelésen és a logoláson is van lehetőség fejleszteni, hogy ne fordulhasson az elő, hogy az alkalmazás hibára fut, de nem logolja le sehova, illetve az se, hogy a folyamatok a run control-ban beragadnak futó státuszba nem várt hibák esetén.

A monitorozás segítése érdekében hasznos lehet valamilyen felhasználói felület fejlesztése, illetve egy töltés elbukása esetén érdemes lehetne az üzemeltetőknek emailt küldeni a hibáról.

Az alkalmazáshoz unit, integrációs és rendszertesztek írása jelentősen felgyorsítaná a tesztelés folyamatát, illetve alaposabban lefedné a use case-eket, mint a manuális tesztelés.

10. Összefoglalás

A szakdolgozatomban elkészült alkalmazást a gyakornoki időszakom alatt írtam, és egy demónak készült. Célja az volt, hogy egy banknak egy modern, big data technológiákon alapuló megoldást nyújtson, ami a stage táblák napi töltését vezérli. Az alkalmazás a nagy mennyiségű és különböző forrásfájloknak a feldolgozását segíti elő azzal, hogy a szükséges kódokat automatikusan létrehozza.

Az alkalmazás elkészítése közben nagy gyakorlatot szereztem szoftverfejlesztésben, sokat tanultam a fejlesztési folyamatokról, mint például a verziókezelésről, illetve olyan területeken is sok hasznos tapasztalatot szereztem, amik egy sikeres alkalmazásnál elengedhetetlenek, mint például az alapos kivételkezelés és logolás.

Megismertem továbbá a Hadoop ökoszisztémáját, kiemeltképpen a Hive-ot és a HDFS-t. A Hive sajátosságairól, illetve arról, hogy miben tér el a relációs adatbáziskezelőktől, miben erősebb, és miben gyengébb, sokat tudtam tanulni.

Megismertem a Postgres-t és a PL/pgSQL nyelvét, illetve a Python-t és annak sok hasznos könyvtárát, mint például a pandas-t, amit data science-ben széleskörűen használnak. A Spark-kal is megismerkedtem, bár csak adatkiolvasásra és betöltésre használtam, komolyabb transzformációkat nem végeztem vele.

A Hadoop használatához elengedhetetlen volt a Linux operációs rendszer mélyebb megismerése, illetve a bash scriptelés alapjainak az elsajátítása. Alapszinten szükséges volt megtanulnom, hogy hogyan lehet bekonfigurálni a Hadoop-ot és annak komponenseit, és azt, hogy Linuxon a különböző környezeti változókat hogyan kell beállítani.

11. Summary

The software for my thesis was developed during my internship and was made as a demo. Its purpose was to provide a modern, big data based solution for a bank that can handle the daily stage load. The software helps with the processing of large quantities of different files by generating the necessary codes automatically.

During the development of the software, I gained a lot of experience in software engineering. I learned a lot about the process of software development, like version control. I also improved in areas that are essential to good software, like good exception handling and logging.

Furthermore, I have become more familiar with the Hadoop ecosystem, especially with Hive and HDFS. I learned a lot about the features of Hive, and how it is different from relational databases, in what ways it is better and worse.

I have become familiar with Postgres and the language of PL/pgSQL, and with Python and its many useful libraries, like pandas, which is widely used in data science. I have also become familiar with Spark, although I only used it to read and write data and have not used it for more complex transformations.

To use Hadoop, it was essential to familiarize myself more with the Linux operating system and with the basics of bash scripting. I had to learn the basics of configuring Hadoop and its components and the basics of setting up environment variables in Linux.

12. Irodalomjegyzék

[1] Data Lake vs Data Warehouse, (https://www.talend.com/resources/data-lake-vs-data-warehouse/), utoljára megtekintve: 2021. 05. 01.

[2] Data Lake Architecture, (https://www.guru99.com/data-lake-architecture.html), utoljára megtekintve: 2021. 05. 01.

[3] What is a Lakehouse?, (https://databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html), utoljára megtekintve: 2021. 05. 11.

[4] Apache Hadoop bemutató, (https://hadoop.apache.org/), utoljára megtekintve: 2021. 05. 01.

[5] HDFS Architecture, (https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#NameNode_and_DataNodes), utoljára megtekintve: 2021. 05. 01.

[6] S3 vs HDFS, (https://resources.useready.com/blog/s3-vs-hdfs-comparing-technologies-in-the-big-data-ecosystem/), utoljára megtekintve: 2021. 05. 11.

[7] Mi a Apache Hive és a HiveQL az Azure HDInsight? (https://docs.microsoft.com/hu-hu/azure/hdinsight/hadoop/hdinsight-use-hive), utoljára megtekintve: 2021. 05. 01.

[8] Hive bemutató, (https://hive.apache.org/), utoljára megtekintve: 2021. 05. 01.

[9] Hive vs Postgres, (https://db-engines.com/en/system/Hive%3BPostgreSQL), utoljára megtekintve: 2021. 05. 01.

[10] Difference between RDBMS and Hive, (https://www.geeksforgeeks.org/difference-between-rdbms-and-hive/), utoljára megtekintve: 2021. 05. 01.

[11] Difference between Hive and RDBMS, (https://beyondcorner.com/learn-apache-hive/difference-hive-rdbms/), utoljára megtekintve: 2021. 05. 01.

[12] Hive vs RDBMS, (http://hadooptutorial.info/hive-vs-rdbms/), utoljára megtekintve: 2021. 05. 01.

[13] Schema-on-Read vs Schema-on-Write, (https://luminousmen.com/post/schema-on-read-vs-schema-on-write), utoljára megtekintve: 2021. 05. 01.

[14] Hive LLAP, (https://cwiki.apache.org/confluence/display/Hive/LLAP), utoljára megtekintve: 2021. 05. 11.

[15] Hive-Metastore: A Basic Introduction, (https://dzone.com/articles/hive-metastore-a-basic-introduction), utoljára megtekintve: 2021. 05. 01.

[16] Configuring the Hive Metastore, (https://docs.cloudera.com/documentation/enterprise/5-8-x/topics/cdh_ig_hive_metastore_configure.html), utoljára megtekintve: 2021. 05. 01.

[17] Different Ways to Configure Hive Metastore, (https://data-flair.training/blogs/apache-hive-metastore/), utoljára megtekintve: 2021. 05. 05.

[18] HBase wiki, (https://hu.wikipedia.org/wiki/Apache_HBase), utoljára megtekintve: 2021. 05. 06.

[19] Apache Impala wiki, (https://en.wikipedia.org/wiki/Apache_Impala), utoljára megtekintve: 2021. 05. 05.

[20] Impala vs Hive, (https://blog.cloudera.com/choosing-the-right-data-warehouse-sql-engine-apache-hive-llap-vs-apache-impala/), utoljára megtekintve: 2021. 05. 05.

[21] Hive vs Impala, (https://www.educba.com/hive-vs-impala/), utoljára megtekintve: 2021. 05. 05.

[22] Hive vs HBase (https://www.dezyre.com/article/hive-vs-hbase-different-technologies-that-work-better-together/322), utoljára megtekintve: 2021. 05. 06.

[23] Introduction to Data Science: How to “Big Data” with Python, (https://dataconomy.com/2016/10/big-data-python/), utoljára megtekintve: 2021. 05. 01.

[24] About pandas, (https://pandas.pydata.org/about/index.html), utoljára megtekintve: 2021. 05. 01.

[25] Scala vs Python vs R vs Java (https://www.knowledgehut.com/blog/programming/scala-vs-python-vs-r-vs-java), utoljára megtekintve: 2021. 05. 06.

[26] Oracle SQL Connector for Hadoop Distributed File System, (https://docs.oracle.com/cd/E37231_01/doc.20/e36961/sqlch.htm#BDCUG125), utoljára megtekintve: 2021. 05. 01.

[27] PostgreSQL vs SQL Server (https://www.enterprisedb.com/blog/microsoft-sql-server-mssql-vs-postgresql-comparison-details-what-differences), utoljára megtekintve: 2021. 05. 06.

[28] What is Apache Tez? (https://www.infoq.com/articles/apache-tez-saha-murthy/), utoljára megtekintve: 2021. 12. 10.

[29] Apache Tez (https://www.cloudera.com/products/open-source/apache-hadoop/apache-tez.html), utoljára megtekintve: 2021. 12. 10.

[30] Spark vs. Tez: What's the Difference? (https://www.xplenty.com/blog/apache-spark-vs-tez-comparison/), utoljára megtekintve: 2021. 12. 10.

[31] psycopg2 - Python-PostgreSQL Database Adapter (https://pypi.org/project/psycopg2/), utoljára megtekintve: 2021. 12. 07.

[32] libpq - C Library (https://www.postgresql.org/docs/current/libpq.html), utoljára megtekintve: 2021. 12. 07.

[33] Python Database API Specification v2.0 (https://www.python.org/dev/peps/pep-0249/), utoljára megtekintve: 2021. 12. 07.

[34] inotify wiki (https://en.wikipedia.org/wiki/Inotify), utoljára megtekintve: 2021. 12. 07.

[35] Dirk deRoos, Paul C. Zikopoulos, Bruce Brown, Rafael Coss, Roman B. Melnyk: Hadoop For Dummies. *John Wiley & Sons, Inc.*, 2014

[36] Builder - Building SparkSession using Fluent API (https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-SparkSession-Builder.html#enableHiveSupport), utoljára megtekintve: 2021. 12. 07.

[37] GitHub wiki (https://en.wikipedia.org/wiki/GitHub), utoljára megtekintve: 2021. 12. 10.

[38] Hadoop Cluster Hardware Recommendations (https://docs.informatica.com/data-engineering/data-engineering-integration/h2l/1415-tuning-and-sizing-guidelines-for-data-engineering-integrati/tuning-and-sizing-guidelines-for-data-engineering-integration--1/sizing-recommendations/hadoop-cluster-hardware-recommendations.html), utoljára megtekintve: 2021. 11. 30.

13. Mellékletek

A fájlok HDFS-be való betöltését megvalósító kód

```
def copy_file_to_hdfs(self, instance_id):
        file_data = self.metadata_conn.select_from_database(
            select="f.file_name ||'.'|| lower(f.file_format) as fname,
f.hdfs_storage_path as target",
            from_table='hfw_file f',
            join1='hfw_process_instance pi',
            on1='f.id = pi.file_id and pi.id = ' + str(instance_id)
        ).iloc[0]

        landing_folder = os.getenv('HFW_PATH') + '/data/temp/'
        target_hdfs_path = file_data['target']
        source_file_name = file_data['fname']

        self.metadata_conn.change_run_control_status("RUNNING", instance_id)
        self.metadata_conn.insert_record_into_logtable(instance_id, 'START', 'Upload
to HDFS started for ' + source_file_name, self.logger_instance.log_file)

        try:
            hdfs_client = InsecureClient(self.hdfs_client_conn)
            hdfs_client.delete(target_hdfs_path + '/' + source_file_name)
            hdfs_client.upload(target_hdfs_path + '/' + source_file_name,
landing_folder + source_file_name)
            logging.info('HDFS upload completed!')
            self.metadata_conn.cleanup_process_instance(instance_id)
            self.metadata_conn.insert_record_into_logtable(instance_id, 'END',
'Upload to HDFS finished for ' + source_file_name, self.logger_instance.log_file)
        except Exception as e:
            logging.error(e)
            logging.error('HDFS upload failed!')
            self.metadata_conn.change_run_control_status("FAILED", instance_id)
            self.metadata_conn.insert_record_into_logtable(instance_id, 'ERROR',
'Upload to HDFS failed for ' + source_file_name, self.logger_instance.log_file)
```

A TXT fájlok CSV-be való átkonvertálását megvalósító kód

```
def convert_fwf_to_csv(self, instance_id):
        file_data = self.metadata_conn.select_from_database(
            select="f.hdfs_storage_path, f.file_name, lower(f.file_format) as
file_format, f.header_fl, string_agg(cast(fc.end_position-fc.start_position + 1 as
varchar), ',' order by fc.column_position) as column_width_list",
            from_table='hfw_file f',
            join1='hfw_file_columns fc',
            on1='f.id = fc.file_id',
            join2='hfw_process_instance pi',
            on2='f.id = pi.file_id and pi.id = ' + str(instance_id),
            groupby="f.hdfs_storage_path, f.file_name, f.file_format, f.header_fl"
        ).iloc[0]

        hdfs_path = file_data['hdfs_storage_path']
        file_name = file_data['file_name']
```

```
        file_format = file_data['file_format']
        column_width_list = [int(width) for width in
file_data['column_width_list'].split(",")]
        if file_data['header_fl'] == 'Y':
            header_value = 1
        else:
            header_value = None

        self.metadata_conn.change_run_control_status("RUNNING", instance_id)
        self.metadata_conn.insert_record_into_logtable(instance_id, 'START', 'Started
fixed width file to CSV conversion with ' + hdfs_path + '/' + file_name + '.' +
file_format, self.logger_instance.log_file)

        try:
            hdfs_client = InsecureClient(self.hdfs_client_conn)
            conversion_postfix = '_conv'
            conversion_dir = hdfs_path + conversion_postfix
            converted_file = file_name + conversion_postfix + '.csv'

            with hdfs_client.read(hdfs_path + '/' + file_name + '.' + file_format,
encoding = 'utf-8') as file_reader:
                df = pd.read_fwf(file_reader, header=header_value,
widths=column_width_list)

            hdfs_client.delete(conversion_dir + '/' + converted_file)
            hdfs_client.delete(conversion_dir)
            hdfs_client.makedirs(conversion_dir, permission='777')

            with hdfs_client.write(conversion_dir + '/' + converted_file, encoding =
'utf-8') as writer:
                df.to_csv(writer, sep=';', index=False)

            self.metadata_conn.cleanup_process_instance(instance_id)
            self.metadata_conn.insert_record_into_logtable(instance_id, 'END', 'Fixed
width file to CSV conversion successfully finished for ' + hdfs_path + '/' +
file_name + '.' + file_format, self.logger_instance.log_file)
        except Exception as e:
            logging.error(e)
            self.metadata_conn.change_run_control_status("FAILED", instance_id)
            self.metadata_conn.insert_record_into_logtable(instance_id, 'ERROR',
'Fixed width file to CSV conversion failed for ' + hdfs_path + '/' + file_name + '.'
+ file_format, self.logger_instance.log_file)
```

A pyspark_to_oracle.py template kódja

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from BaseProcess import BaseProcess

class Pyspark_to_oracle(BaseProcess):
    def __init__(self):
        super().__init__('pyspark_scripts', 'pyspark_script_<<FILE>>', "Normal",
False)
```

```
        self.instance_id = <<INSTANCE_ID>>
        self.file_name = '<<FILE>>'

    def Load(self):
        self.metadata_conn.change_run_control_status("RUNNING", self.instance_id)
        self.metadata_conn.insert_record_into_logtable(self.instance_id, 'START',
'Oracle table generation with PySpark for ' + self.file_name,
self.logger_instance.log_file)

        try:
            spark = SparkSession.builder.appName("Oracle Stage load for " +
self.file_name).getOrCreate()

            schema = StructType() \
<<COLUMNS>>

            df = spark.read.option("header",<<HEADER_BOOL>>) \
                .option("delimiter","<<FIELD_TERMINATOR>>") \
                .option("nullValue","<<NULL_VALUE>>") \
                .option("escape","<<ESCAPE_CHARACTER>>") \
                .schema(schema) \
                .csv("<<SERVER_PATH>><<HDFS_STORAGE_PATH>>")

            df.write.format('jdbc') \
                .options(url='<<URL>>') \
                .options(driver='<<DRIVER>>') \
                .options(user='<<USER>>') \
                .options(password='<<PASSWORD>>') \
                .options(batchsize=<<BATCHSIZE>>) \
                .options(dbtable=self.file_name) \
                .options(truncate='True') \
                .option("sessionInitStatement", """BEGIN execute immediate 'ALTER
SESSION ENABLE PARALLEL DML'; END;""") \
                .mode('overwrite').save()

        except Exception as e:
            print(e)
            self.metadata_conn.change_run_control_status("FAILED", self.instance_id)
            self.metadata_conn.insert_record_into_logtable(self.instance_id, 'ERROR',
'Oracle table generation with PySpark for ' + self.file_name,
self.logger_instance.log_file)
            exit()

        self.metadata_conn.cleanup_process_instance(self.instance_id)
        self.metadata_conn.insert_record_into_logtable(self.instance_id, 'END',
'Oracle table generation with PySpark for ' + self.file_name,
self.logger_instance.log_file)


Pyspark_to_oracle = Pyspark_to_oracle()
Pyspark_to_oracle.Load()
```

A Hive partíciók eldobását megvalósító kód

```
def DropPartitions(self, table_name, drop_date):
        logging.debug('Dropping ' + table_name + ' partitions before ' +
str(drop_date))
        statement = 'ALTER TABLE ' + table_name + " DROP PARTITION (calendar_date <
'" + str(drop_date) + "')"
        drop_status = True

        with self.__hive_conn.cursor() as cursor:
            try:
                logging.info('The following Hive statement will run: ' + statement)
                cursor.execute(statement)
                logging.info('Statement succeeded')
            except Exception as e:
                logging.error(e)
                drop_status = False

        if drop_status:
            logging.debug('Dropped ' + table_name + ' partitions before ' +
str(drop_date))
        else:
            logging.error('Dropping ' + table_name + ' partitions before ' +
str(drop_date) + ' failed!')
```

A Hive partíciók archiválását megvalósító kód

```
def ArchivePartitions(self, table_name, archive_partitions, archive_date):
        logging.debug('Archiving ' + table_name + ' partitions before ' +
str(archive_date))
        archive_status = True

        for p in archive_partitions:
            statement = 'ALTER TABLE ' + table_name + '_arch' + " EXCHANGE PARTITION
(calendar_date = '" + p + "') WITH TABLE " + table_name
            with self.__hive_conn.cursor() as cursor:
                try:
                    logging.info('The following Hive statement will run: ' +
statement)
                    cursor.execute(statement)
                    logging.info('Statement succeeded')
                except Exception as e:
                    logging.error(e)
                    archive_status = False
                    break

        if archive_status:
            logging.debug('Archived ' + table_name + ' partitions before ' +
str(archive_date))
        else:
            logging.error('Archiving ' + table_name + ' partitions before ' +
str(archive_date) + ' failed!')
```