NEUMANN JÁNOS

INFORMATIKAI
KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Németh Norbert**
2022	Hallgató törzskönyvi száma:	T/006332/FI12904/N

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Németh Norbert**
Törzskönyvi száma: T/006332/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Adatfeldolgozó ETL folyamat létrehozása Python nyelv használatával
Creating Data Processing ETL using the Python language

Intézményi konzulens: Rusznák Attila
Külső konzulens:

Beadási határidő: 2022. május 15.

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

A feladat

Az adatoknak jelentős szerepe van a vállalatok életében, ezért fontos a megfelelő Adatintegrálás a vállalatok életében, hogy el tudják érni az üzleti céljaikat. A Python az egyik legnépszerűbb programozási nyelv az IT-ban. A nyelvet nagyon sok típusú feladatra használják. A szakdolgozatom során szeretném megvizsgálni, hogy érdemes lehet-e a Python nyelv és csomagjai segítségével, adatintegrációs folyamatokat elvégezni. Ezáltal megvalósítva egy olyan adatfeldolgozási folyamatot, amellyel megfelelő módon lehet egy vállalat számára az Adatokat integrálni egy célrendszerbe, pl. egy adatbázisba.

A dolgozatnak tartalmaznia kell:

- a feladat ismertetését,
- a követelményeket,
- az adatfeldolgozás és adatminőség bemutatását,
- a lehetőségek vizsgálatát az adatfeldolgozó folyamat elkészítésére,
- egy relációs adatbázis megtervezését és létrehozását,
- az ETL folyamatok megtervezését és létrehozását a választott eszközök segítségével,
- a tesztelést,
- az eredményeket és azok vizsgálatát,
- végül a továbbfejlesztési lehetőségek vizsgálatát

Fleiner R

.....

Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

Ruscsák Ajk

.....
külső konzulens

.....
intézményi konzulens

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Németh Norbert ..... [redacted] ..... Nappali .....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Adatfeldolgozó ETL folyamat létrehozása Python nyelv használatával .....

Szakdolgozat / Diplomamunka[2] címe angolul:

Creating data processing ETL using the Python language .....

Intézményi konzulens: Külső konzulens:

Rusznák Attila ....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.09.20	Szakdolgozat témájának átbeszélése	Rusznák AAi
2.	2021.10.14	Feladatlap tartalmának megbeszélése	Rusznák AAi
3.	2021.11.30	Tartalmi elemek átbeszélése	Rusznák AAi
4.	2021.12.07	Formai követelemények pontosítása	Rusznák AAi

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 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 20 21. december 10.

Rusznák AAi

.....

intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Németh Norbert ..... [redacted] ..... Nappali.....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Adatfeldolgozó ETL folyamat létrehozása Python nyelv használatával.....

Szakdolgozat / Diplomamunka[2] címe angolul:

Creating data processing ETL using the Python language.....

Intézményi konzulens: Külső konzulens:

Rusznák Attila ....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.03.30	Tervezés átbeszélése	Rusznák AAik
2.	2022.04.15	Megvalósítás mód átbeszélése	Rusznák AAik
3.	2022.05.02	Megvalósítás bemutatása, Formai követelmények pontosítása	Rusznák AAik
4.	2022.05.12	Dolgozat bemutatása, formai követelmények ellenőrzése	Rusznák AAik

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, 20 22. 05. 13.

Rusznák AAik .....

intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

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, 2022.05.13

Németh Valeria

hallgató aláírása

TARTALOMJEGYZÉK

1. BEVEZETÉS

Manapság egy vállalat működése során rengeteg adat keletkezik az információ sokaságából adódóan. A vállalatok megfelelő működése érdekében a nagy mennyiségű adathalmazt olyan formában és rendezettségben kell tárolni, hogy az kielégítse a rendszer számára támasztott igényeket.

A piacon jelenlévő vállalatok a működésük során a megszerzett adatokra alapozzák a döntéseiket, ezért elengedhetetlen az, hogy az adatok megfelelő struktúrában kerüljenek eltárolásra a vállalat informatikai és döntéstámogatói rendszereiben. Természetesen egy vállalat működése során nagyon sokféle adat keletkezik, mindezek különböző forrásokból. Ezért érdemes egy vállalaton belül központilag megfontolni egy adatstruktúrát, amelynek segítségével átláthatóvá válnak az adatai a vállalatnak, természetesen ennek a struktúrán igazodnia kell ahhoz, hogy ezek az adatok milyen forrásból érkeznek és milyen szakterületet reprezentálnak.

A fent említett megfelelő struktúra mellet egy adatintegráció során figyelmet kell fordítani az Adatminőségre. Ugyanis az Adatokban hibák lehetnek, amiket kezelni kell. Az adatok ilyen szempontból történő feldolgozását szolgálják az ETL folyamatok. Az adatok feldolgozására, egységesítésére természetesen vannak különböző cégek által elkészített szoftverek, illetve szoftverrendszerek. Ezek között lehet válogatni ár, bonyolultság, használhatóság és technológia fejlettség alapján.

A dolgozatomban azt szeretném vizsgálni, hogy milyen hatékonysággal lehet létrehozni a fent említett mintára adatfeldolgozó folyamatot Python nyelv alapjain.

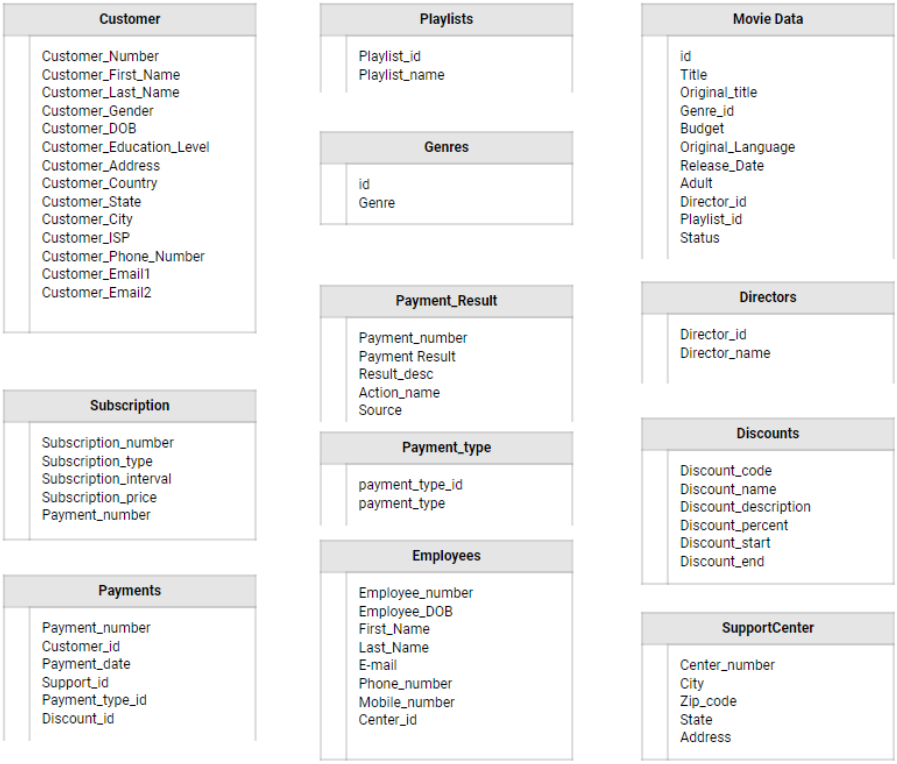
Azért választottam ezt a nyelvet ugyanis a felmérések alapján ez az egyik legnépszerűbb programozási nyelv jelenleg a piacon, valamint igen gyakran használják az adatelemzés és adattudománnyal kapcsolatos feladatokban. Valamint a nyelv rengeteg ingyenes és bárki számára elérhető kiegészítővel rendelkezik, ebből kifolyólag érdemes lehet ezt választani az igen drága kész eszközök helyett.

2. ADATFELDOLGOZÁS

2.1 Célok

Az adatfeldolgozás megvalósításának fő célja, a bevezetőben említett indok. Mégpedig a vállalatok számára az adatok üzleti értéket jelentenek. A vállalatok vezetői az adatok alapján hoznak döntéseket, irányítják a vállalatot és ezzel befolyásolják az üzleti eredményeiket. A vállalatnak viszont a fejlesztési folyamat elején tisztáznia kell a célokat az adatokkal kapcsolatban, vagyis, hogy azokat milyen módon szeretné hasznosítani az üzleti előrelépés szempontjából. Mivel a mai világban igen szerteágazó vállalatok és iparágak vannak ebből adódóan több lehetőség nyílik például az adatok felhasználására. [1]

Adott egy vállalat, amely film streaming szolgáltatással foglalkozik, ebből a szolgáltatásból adódóan megtalálható sok adat a vállaltnál. Ilyenek például ügyféladatok, előfizetés adatok. Mivel ezek az adatok már inkább személyes adatok ezeket generálni fogom a feladathoz egy séma alapján. A filmek adatai pedig internetes forrásból származnak, amihez tartoznak értékelések is. Ezt az adatforrást használtam részben a projektmunka 1 és projektmunka 2 során is. Ezek a fájlok csv és xlsx formátumban állnak rendelkezésre. A megoldás célja, hogy ezeket a fájlokat változtatás nélkül lehessen betölteni a rendszerbe, ami azt automatikusan feldolgozza. Az adatokat az 1. ábra tartalmazza.



1. ábra A jelenleg meglévő forrás adatok struktúrája

2.1.2 Adatelemzés

A vállalatokon belül az adatok felhasználásának egyik legelterjedtebb módszere az adatelemzés, ennek több fajtáját megkülönböztetjük.

Leíró elemzés: Az analitika az a típusa mely a múltbéli adatokon alapszik. Ebben az elemzési típusban arra a kérdésre keresnek választ, hogy mi történt. Természetesen az informatikai rendszertől függenek a lehetőségek. Ha a rendszer megengedi akkor a jelenlegi eseményeket is lehetséges elemezni.

Diagnosztikai elemzés: Az adatelemzés olyan formája amikor arra keres választ az elemzést végző személy, hogy mi miért történt a múltban. A folyamat lényege, hogy megpróbáljanak összefüggést keresni az elért eredményekben. Ez már jóval nagyobb eséllyel segíti a döntéshozást a cég életében.

Prediktív elemzés: Egy olyan forma amikor az adatokban talált minták alapján következtetést végeznek, különböző módszerek segítségével. Arra szeretnének választ kapni, hogy mi fog történni a jövőben. Ezen a területen már a gépi tanulást és a különböző statisztikai algoritmusokat is használnak megoldásként.

Preszkriptív elemzés: Szoros kapcsolatban van a leíró és prediktív elemzéssel, tulajdonképpen a kettőt kombinálja és a múltbéli eredmények, valamint az előre jósolható eredmények alapján próbálja meghatározni azt, hogy milyen döntéseket kell hoznunk a kívánt eredmény eléréséhez. [2]

2.1.3 Adatbányászat

A fent említett adatelemzési metodológiáktól eltérő technika, viszont valamilyen szinten átfedésben van a prediktív elemzéssel.

Az adatbányászat egy olyan folyamat, amely során nem konkrét kérdésre keresnek választ az adatokon belül, hanem rejtett mintákat keresnek benne. Ez egy jóval nehezebben behatárolható terület ugyanis a várható eredmény igen szerteágazó lehet.

Főként előre megírt algoritmusokat használnak nagymennyiségű adaton, amelyből mintát szeretnének kapni, például: Hogy mik a független és mik a függő változók egy adathalmazban. Az adatbányászatban egy meghatározott ipari folyamat által végzik a műveleteket, ami a CRISP-DM. [3]

Az Adatbányászat valamint az Adatelemzés közös tulajdonsága, hogy megköveteli azt, hogy az Adatok megfelelő módon legyenek integrálva egy rendszerben. A hatásos információk kinyeréséhez szükséges, hogy az Adatok jól strukturáltak legyenek, és az Adatok minősége biztosítva legyen.

2.2 Adattárház

Egyik lehetséges megoldás egy üzleti támogató rendszer alapjainak megalkotására az adattárház. Az adattárház egy adatfeldolgozó és adattároló rendszer egyvelege, amely számos forrásból táplálkozik és egy adott vállalat üzleti intelligencia rendszerét vagy elemző rendszerét támogatja. Egy adattárházba az adatok általában széleskörű forrásokból származnak, például logokból, tranzakciós rendszerekből, vállalati rendszerekből. A vállalatok ilyen rendszerek adatai alapján hoznak döntéseket, amelyek nagy mennyiségű adatot tudnak központilag, integráltan tárolni. Bill Inmon volt, aki először megfogalmazott egy általános definíciót az adattárház fogalmára, mégpedig:

„*A Data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decision.*" [4]

Bill Inmon a következőket említette:

- **Tárgyorientált**: Az adott rendszert az alapján tervezzük, hogy milyen feladatokat akarunk elvégezni és milyen funkcionalitással tudjuk kiszolgálni az adott témakört, amiről adattal rendelkezünk. Ez magát az adatbázisok felépítését is érinti. Például egy olyan vállalat, amely lakossági ügyfeleknek szolgáltat termékeket akkor üzleti szempontból az adattárházat érdemes az ügyfelek adatai köré felépíteni.
- **Integrált**: A lényege az integráltságnak, hogy az adott akár több forrásból származó adatainkat, a céloknak megfelelően egy adott szabvány által átalakítva egyhelyre gyűjtsük be.
- **Nem Felejtő**: Az adatok, amelyek bekerülnek az adattárházba tartósan meg is maradnak. A bekerülés során az adatok valamilyen formában jelölve vannak bekerülési idejük alapján. Ha pedig frissülnek az adatok akkor az új adat bekerülési idejét lerögzítjük ezáltal időben visszakereshető lesz több korábbi verzió is. Más néven historizálásnak hívják ezt a folyamatot.
- **Időfüggő**: Az adattáházzal szemben elvárás, hogy az adatok változása nyomon követhető legyen az egész adattárházban, ezáltal például elemezhetőek a múlt történései egy vállalattal kapcsolatban.

Az Inmon féle leírás alapján, az adattárház röviden összefoglalható egy olyan rendszernek, amelyben az adatokat egy adott téma alapján historikusan tároljuk, egységes formában, amely elemzésre bocsátható. Természetesen az elméleti megfogalmazás egyszerűnek hangozhat, de ez egy igen komplex rendszer. Számos megvalósítási módszer létezik az adattárházak témakörben, valamint fontos elvek, amelyeket szükséges betartani. [5]

2.2.1 Dimenzionális adatmodell

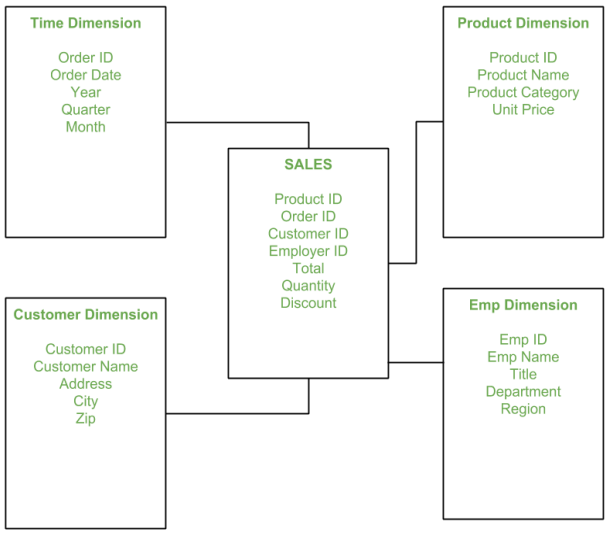
A dimenzionális adatmodellezés lényege, hogy az adattárház felépítését tényre és dimenziókra lehet bontani. A tények alatt a konkrét számszerű adatokat értjük a

dimenziók pedig amelyek hátteret adnak az adatainknak (úgymond dimenzióba helyezik azokat). Ez alapján meg kell határozni, hogy milyen tényadatokra van szükség és ezeket milyen dimenzióba helyezzük el. A dimenzióból mindig létezik egy fix tábla az üzleti rendszereknél, az idődimenzió. Ugyanis az alapkövetelmény minden esetben, hogy az adatainkat időben is el tudjuk helyezni. A ténytábla felel azért, hogy a dimenziók a kulcsokon keresztül össze legyenek kapcsolva, ennek köszönhetően az adatokat el lehet érni több dimenziótáblát is összekapcsolva.

2.2.2 Adattárház séma

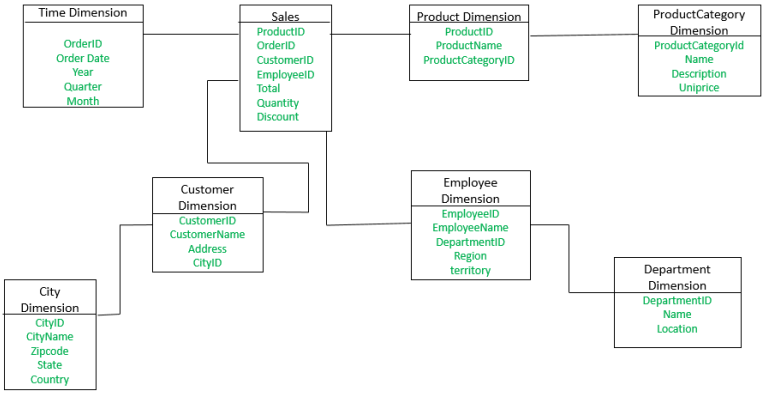
Az adattárház tervezési sémák között 2 alapmodellt különböztetnek meg, mindkettő az alakjáról kapta a nevét.

Csillagséma: Egy tény adatokat tartalmazó táblát tartalmaz a központban, amelyhez minden dimenziótábla hozzá van kapcsolódva az idegen kulcsaik által, felépítése a 2. ábrán látható.



2. ábra Csillagséma példa [6]

Hópehely séma: A központban itt is a ténytábla található, a különbség viszont a hozzájuk kapcsolódó dimenzió táblákban rejlik. Ebben az esetben a dimenzió táblák elrendezésében ugyanis hierarchia található, ezt szemlélteti 3. ábra. Ez akkor fordulhat elő, ha a dimenzió tábláink között kapcsolatok találhatóak. Például: A tény táblánk eladásokkal kapcsolatos számadatokat tartalmaz. Ehhez kapcsolódik egy vásárló dimenzió, amely a vásárló adatait tartalmazza, viszont a vásárlónak van lakhelye, amely megint egy külön dimenzió. [7]



3. ábra Hópehely séma ábrája [8]

2.2.3 Granularitás

Az analitikai rendszerekben az adatok granularitása fontos kérdés, viszont inkább a tervezési szempontok között szerepel, ez a fogalom írja le azt, hogy az adatok milyen részletességi szinten fognak szerepelni a rendszerben. Minél alacsonyabb egy szint annál részletesebb lehet az adatok elemzésének az eredménye a tervezett rendszerben. A tervezés során az adatok szemcsézettségét mind a ténytáblánál és a dimenzió tábláknál is azonos szinten kell kezelni és optimális esetben ez a szint minél alacsonyabb. Erre egy jó példa, ha eladási adatokat tárolunk és ezek az adatok napi bontásban vannak akkor a dátum dimenziónak is napi szinten kell tárolnia a dátum adatokat, ellenkező esetben probléma merülhet fel az adatok elemzésekor. Nem feltétlenül lehet minden adat között összefüggés a szemcsézettségi szempontból azért érdemes minden egyes táblára ezt külön meghatározni figyelembe véve a többi adatot.

2.3 Data Lake

Az adattárházak sokáig alapjai voltak az adatalapú üzleti döntéshozatalnak, ám az utóbbi pár évben az adatmennyiségek jelentős növekedésével és azzal, hogy ezek az adatok maguk is folyamatosan változnak megjelentek a data lake-ek. Főként az adattárház alternatívájaként tekintenek rá, ám annál sokkal nagyobb szabadságot enged a tervezés során a szakembereknek. A legnagyobb előnye az adattárházzal szemben, hogy nem egy előre definiált séma alapján működik és az adatok sem úgy helyezkednek el benne, hanem az összes nyers adat kerül betöltésre a fizikai adatbázisba. Itt természetesen kerülhet átalakításra valamilyen módon az adat, hogy az eltárolható legyen az adatbázisban viszont nem olyan mértékben, mint egy adattárházban. Mivel nincs előre definiált séma a tervezése sokkal egyszerűbb. Az adatkapcsolatok akkor alakulnak ki amikor az ügyfél vagy a rendszernek a használója eléri az adatokat, ezt nevezik schema-on-read-nek. A data lake a működéséből adódóan egyszerűbb, flexibilisebb gyorsabban alkalmazhatóbb rendszer, mint egy adattárház viszont az adatok megfelelő eléréséhez sokkal több szakértelem kell ezért általában informatikai cégek belső rendszerekben használják.

3. ADATMINŐSÉG

3.1 Adatminőség Definíciója

Az Adatminőség egy kulcsfontosságú tulajdonsága az Adatfeldolgozó rendszernek. Azt írja le, hogy az Adat milyen könnyen feldolgozható és elemezhető ezáltal pontos és valós információk lehet kinyerni az adatokból.

Adatminőség követelményeinek meghatározása erősen függ az üzleti céloktól, általában az üzleti célok eléréséhez szükséges követelmények felállításával együtt történik. [9]

3.2 Felmerülő problémák

Az adatminőséggel kapcsolatos problémák számtalan területről érkezhetnek, ugyanis a vállalatok általában több forrásból szedik az adataikat. Ha egy vállalat hatékony eredményeket szeretne elérni fontos, hogy ne legyenek problémák a különböző forrásból származó adatok integrálásával.

- Nem naprakész adatok

  Ezalatt az érthető, hogy az adataink nem naprakészek és nem azt az információt szolgálják, ami a jelenlegi helyzetet tükrözi üzleti szempontból.

- Kétértelműség

  Olyan Adatok érkeznek, amelyek nem feltétlenül tartoznak bele ebbe az adathalmazba, de ez nem állapítható meg teljes magabiztossággal. Általában ilyen esetben az adatintegrációs folyamatokban kell keresni a hibát. Ezekben az esetekben lehet, hogy rosszul lett egy nyersadat más típussá konvertálva, esetleg rossz karakterkódolást használt a forrásfájl, vagy rosszul lett elválasztva a folyamat során a többi adattól.

- Duplikáció

  Általában a nagy adatforgalomnak kitett rendszerek több forrásból nyerik az adatokat ezáltal előfordulhat olyan, hogy az adat azonosítói pl.: kulcsok többször szerepelnek, vagy egész adatsorok is duplikálva találhatók

- Inkonzisztencia

  Ennek a problémának ugyanaz az alapja, mint a duplikációknak, sokszor a többszörös adatforrásokból kiindulva az adatok nem ugyanazt az információt írják le, mint a körülötte levő többi adat.

- Adathiba

  Adathibával akkor találkozhatunk, ha olyan adat található egy adatmezőben ami nem a céladat típusnak megfelelő, vagy hiányzik belőle karakter. A problémának viszont lehet egy olyan forrása is, amikor olyan rendszerből érkezik az adat, ahol a feldolgozás előtt kézzel viszik be az információkat, ilyenkor emberi hiba okozza az inkonzisztenciát. [10]

3.3 Adatminőség javítása

A fentebb látható információkból levonható az a következtetés, hogy az adatminőség biztosítása igen kardinális kérdés egy informatikai rendszerben a vállalat számára. Az adatminőség biztosításához elengedhetetlen, hogy jól ismerjük az adatot, amivel dolgozunk és átlátható legyen számunkra, ezzel már az adathibák, megakadályozhatók vagy javíthatók lesznek, valamint az inkonzisztenciával kapcsolatos problémák nagy része is kiküszöbölhető, ha megtudjuk mondani, hogy a kérdéses adat bele illik-e az adott rendszerbe vagy sem. Az adatminőség javítására, biztosítására többféle eljárás létezik attól függően, hogy milyen rendszert szeretnénk kivitelezni. Általában 4 lépést határoznak meg amellyel önmagában az adatminőséget lehet menedzselni. Ezek pedig:

- Adatok felmérése
- Szabályok meghatározása
- Szabályok alkalmazása
- Adatminőség ellenőrzése

Van, ahol különválasztják az adatfeldolgozás folyamatától az adatminőség biztosítását. Például ahol Adatbányászattal foglalkoznak akkor az adatminőség biztosítását a CRISP-DM folyamat során biztosítják. Viszont lehetséges olyan eset is amikor az adatminőség biztosítása a konkrét feldolgozó folyamatok előtt végbemegy.

A gyakorlati esetek nagy részében ám nincs konkrétan meghatározva, hogy a folyamat melyik részében megy végbe az adatminőség biztosítása. Ezért érdemes megfontolni az ETL vagy ELT folyamatokat.

4 ETL FOLYAMAT

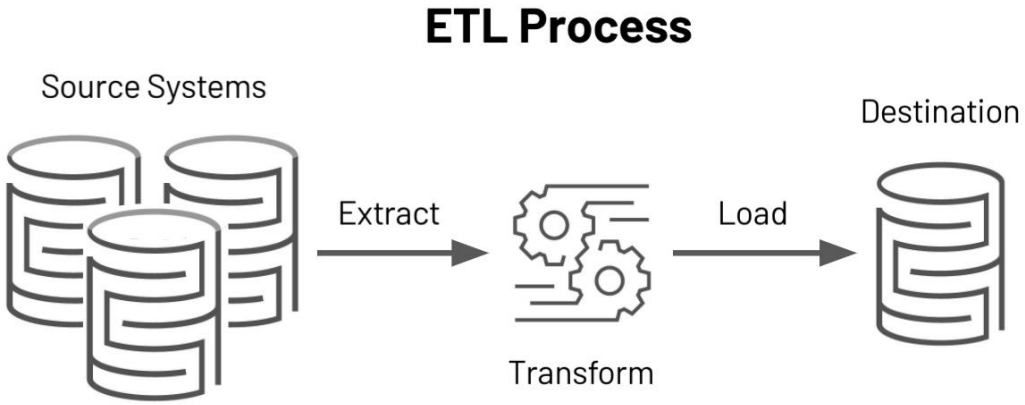
4.1 Bemutatása

Az ETL folyamat az adatintegrálás részfolyamata de tekinthető egy önálló folyamatnak is egy analitikai rendszeren belül. A folyamat maga az adatkinyerés átalakítás vagy transzformálás és adatbetöltés részfolyamatokra bontható. A lényege, hogy az adat megfelelő formában eljuttatható legyen a forrásrendszerből a célrendszerbe. Ezt röviden a Google felhő platformja is tökéletesen összefoglalja.

„ETL describes the end-to-end process by which a company takes its full breadth of data—structured and unstructured and managed by any number of teams from anywhere in the world—and gets it to a state where it's actually useful for business purposes." [11]

Az ETL folyamatok alkalmazása körülbelül a 90-es évektől kezdődött. A neve alapján lebontható három külön álló részre az elméletben, viszont a gyakorlati alkalmazásában egy bonyolultabb rendszernél lehetnek átfedések a részfolyamatai között. Valamint a rendszer kialakításától függően ezek akár teljes átfedésben is lehetnek egymással. Ez probléma lehet, ha már a rendszer nem teljesen átlátható így érdemes törekedni arra, hogy a részfolyamat el legyenek különítve egymástól.

Ezeket a folyamatokat a mai piacon legtöbbször üzleti intelligencia vagy analitikai rendszerekben használják viszont lehet más célokra is használni. Legújabban kutató rendszerekben és gépi tanulást alkalmazó rendszerekben ETL folyamatokat használnak. [12]



4. ábra ETL folyamat sorrendje [13]

4.2.1 Extract

Az Extract folyamat a legelső része az ETL folyamatnak. Ennek a lényege az adat kinyerése különböző forrás rendszerekből és ezeknek az adatoknak az átültetése egy úgynevezett staging rétegbe, ami lehet adatbázis vagy a kinyert adatokat tartalmazó fájl.

Az extract folyamat megtervezése során a legtöbb figyelmet az adatforrás igényli. Legtöbbször a forrás adatok több fájlban érkeznek a rendszerbe. Viszont a technológiák fejlődésével előálltak olyan rendszerek, amelyek saját API-val rendelkeznek és ezeken keresztül lehet elérni a szükséges adatokat. Ez önmagában nem jelent problémát ugyanis ezen keresztül elérhetővé válnak az adott forrás rendszer által elérhető adatok. Viszont ezek a rendszerek legtöbbször nem egységes alapokon működnek és mindegyik elérési módja különbözhet. Valamint probléma lehet az is, hogy ezek a rendszerek nem túl jól dokumentáltak vagy harmadik fél által vannak dokumentálva és így jóval bonyolultabb a használatuk. Természetesen, amikor az extract folyamatot tervezik akkor nem lehet megválasztani az adott forrást, hanem magát a folyamatot kell megfelelő módon megtervezni.

Másik szempont az Extract folyamat tervezése során az elindításának módja:

- **Valós idejű (Based on update)**: Tulajdonképpen valós idejű adatkinyerést lehet elérni ezzel a módszerrel. Azon alapszik, hogy amikor valamilyen rekord változik a forrásrendszerben, akkor a kinyerési folyamat elindul és az adat frissítésre kerül. Ez a rendszer viszont csak akkor alkalmazható, ha lehetőség van arra, hogy a forrásrendszerben vizsgálni lehessen, hogy az adat mikor frissül.
- **Ütemezett**: A kinyerési folyamat megadott időpontokban vagy megadott időközönként indul el
- **Esemény vezérelt**: Hasonlít a valós idejű végrehajtáshoz, viszont itt nem minden változásra indul el a kinyerés folyamat, hanem tetszőleges folyamatra. pl. megadott oszlopok frissítésére vagy ha az adatok mennyisége elér egy meghatározott korlátot.

Érdemes megjegyezni, hogy a megfelelő működési módszer kiválasztása elegendő a rendszer működéséhez, de a gyakorlatban ezeket a módszereket szokták kombinálni annak érdekében, hogy a rendszer minél optimálisabban működjön.

Harmadik szempont az extract folyamat végrehajtás módjának kiválasztása, itt alapvetően kétfajtát különböztetünk meg:

- Update: Az extract-ban csak a frissült adatokat kezeljük a változatlan adatokkal nem foglalkozunk.
- Full extract: Minden egyes extract folyamat futtatáskor minden adatot kinyerünk. Ez a folyamat igen időigényes és terhelheti a rendszert, ezért érdemes kis adatmennyiségeknél használni.

Az extract végrehajtása végén az adatok a staging area-ba kerülnek. Ez a részleg biztosítja az adatok hozzáférhetőségét a transform folyamat számára. Ebben a staging rétegben érdemes az adatokat olyan formában tárolni, ami egyszerűbbé teszi a transzformációk végrehajtását rajtuk. [14]

4.2.2 Transform

A transform részfolyamat az ETL második része és egyben a legfontosabb állomása. A nevéből is ered, hogy itt történik az adatok átalakítása a követelményekben meghatározott formátumra. Ami majd ezután a rész után tovább kerül a betöltésre. Ebben a folyamatban az átalakítás alatt több műveletet végez a rendszer. A cél, hogy egységes formátumban szerepeljenek az adatok függetlenül attól, hogy milyen rendszerből valók és milyen formátumban voltak, amikor kinyerésre kerültek.

A tradicionális ETL folyamatokat használó rendszernél, a részfolyamatok eredményeit tartalmazó részek elkülönítve tárolódnak egymástól (pl. különálló adatbázisokban).

Viszont a mai világban az elmosódott határoknak köszönhetően ez átalakulóban, erre a problémára két lehetséges megoldás létezik:

- **Elkülönített adatátalakítás (Multistage Data Transformation)**: A Transform réteg folyamatai elkülönülnek a többi részfolyamattól és jól átlátható határok mellett történik az adatok feldolgozása és átalakítása majd azok mentése. Ennek az előnye, hogy sokkal átláthatóbb és nyomon követhetőbb a rendszer. Valamint biztonsági szempontból is hatékonyabb.
- **Adatbázison belüli átalakítás (In stage data transformation)**: Az átalakítás egyből az adat végső helyén történik és onnan nem lesz tovább véve, csak exportálni lehet az adatokat további használatra. Ez a folyamat inkább, már az ELT folyamathoz közelít.

A transform folyamat tervezése során természetesen a legnagyobb hangsúly az eszközölni kívánt átalakításokon van. Ezeknek az átalakításoknak az eredményei erősen összefüggenek az adatminőséggel kapcsolatos elvárásoknak ezért fontos a gondos megtervezése. A tervezés során érdemes törekedni, hogy a műveletek minél egyszerűbbek legyenek és ezekkel a műveletekkel lehessen elérni egy üzleti szempontból standardizált végállapotot, amikor az adatok megfelelő formátumban helyezkednek és tényleges üzleti értéket képviselnek.

Általában a következő műveleteket kell elvégezni:

- Adatok tisztítása
- Duplikációk eltüntetése
- Szűrések (Validálás)
- Aggregálás
- Dátumhibák kezelése
- Kulcsok kialakítása

A transzformációk végrehajtása után amint az adatok megfelelőnek bizonyulnak betöltésre kerülhetnek az analitikai rendszerbe, ahol ezek az adatok tárolva lesznek és felhasználásra vagy exportálásra kerülnek. [15]

4.2.3 Load

A load folyamat végső része az ETL folyamatnak és általában a legegyszerűbb is ezek közül. Természetesen ahány rendszer annyi féle megvalósítás. A fő szempont, amit figyelembe kell venni, hogy milyen módon kerülnek áttöltésre az adatok a célrendszerbe. Ez attól is függ, hogy milyen alapokra épül a rendszerünk. pl., ha egy adattárházat fejlesztünk akkor figyelembe kell venni a ténytáblát és a dimenziótáblákat a feltöltés sorrendjét tekintve. Ezeket a szempontokat a konkrét rendszert ismerve lehet méretezni. Általánosságban a fő kérdés a betöltés módja:

- **Teljes betöltés**: A teljes betöltés az egész adathalmazt átmozgatja a célhelyre. Ezt a módot akkor alkalmazzák amikor egy friss rendszerbe először futtatják le a folyamatot vagy a rendszer olyan feldolgozásokat használ, hogy újra be kell tölteni az adatokat.
- **Inkrementális betöltés**: Ebben a betöltési módban csak a deltaadat vagyis a különbségek kerülnek áttöltésre. Azt megoldani, hogy ez lehetséges legyen valamilyen módon jelölni kell az adatoknál, hogy frissültek-e vagy sem.

A két betöltési mód közül érdemes úgy választani, hogy figyelembe kell venni magát a rendszert is teljesítmény szempontból és érdemesebb kisebb adagokban áttölteni az adatokat persze ha ez lehetséges.

4.3 ELT

Az eddig bemutatott tradicionális ETL folyamattól eltérő adatfeldolgozási folyamat az ELT. Az alapjai ugyanazok, mint a sima extract, transform, load csak a részfolyamatok sorrendjei vannak felcserélve. Korábban említettem, hogy a jelenlegi iparban a sokféle megoldásnak köszönhetően a határok elvannak mosódva a részfolyamatok között. Valamint vannak olyan helyzetek amikor jobb az adatátalakításokat elvégezni a folyamat végállomásán.

Az ELT folyamat lényege, hogy az transform és load rész helyet cserél az ETL-hez képest. Ennek megvannak a maga okai. A fő gondolatmenet, hogy az Adatok a kinyerés után betöltésre kerülnek az adatbázisba, ahol tárolva lesznek és csak az után kezdődnek el az átalakításuk, ami a kívánt formára alakítja az adatokat. A mai világban az adatmennyiségek növekedésével érdemes azzal számolni, hogy kevesebb a tárhely foglalása a rendszernek, ha egy helyen tároljuk az adatokat és végezzük az átalakításokat. Ez az egyik fő oka annak, hogy ez a folyamat létezik. Ennek köszönhető, hogy a felhő alapú rendszereknél az ELT folyamatot preferálják ugyanis kevesebb tárhely használ, ami felhő szolgáltatás esetén olcsóbbá teszi a működést.

A másik szempont a rugalmasság, ez abból következik, hogy először kerülnek betöltésre az adatok és utána átalakításra tehát nem kell megvárni amíg a szükséges adatokon végbe megy a transzformáció, hanem minden adat bekerül a rendszerbe és utána

csak a szükséges adatok lesznek átalakítva. Ezzel nagyobb sebességet is lehet elérni, és valamennyi százalékkal a rendelkezésre állást is lehet növelni.

A big data rendszereken, ahol rendszerint nagy adatmennyiség és strukturálatlan adattal dolgoznak érdemesebb ezt a módszert használni annak köszönhetően, hogy kinyerés után az adat betöltésre kerül a rendszerbe. Ez viszont sok hiba lehetőséget képez ezekben a folyamatokban, ezért sokkal kevésbé megbízható ez a módszer, mint egy tradicionális ETL. [16]

4.4 Historizálás

Az adatok nyomon követhetősége egy olyan rendszerben, ami a múlt történésein alapul, kulcsfontosságú az üzleti eredmények javítása szempontjából, és technikai szempontból sem előnyös, ha nem jegyezzük fel az adatokat valamilyen formában, hogy azok meddig voltak érvényesek és mikortól nem érvényesek. Alapelv az analitikai rendszereknél, hogy azokat az adatokat nem törlik amelyek bekerülnek a rendszerbe, adott idő után az adatok archiválásra kerülhetnek és nem lesznek benne a rendszerben, de így ezek is elérhetőek maradnak. Erre a problémára vannak különböző módok, hogy hogyan legyenek kezelve a változások. Például ilyen a CDC, SCD valamint a log triggerek. Ezek nem kész megoldások, hanem tervezési elvek, amelyeket az adatbázisfejlesztők használnak a tervezés során és leírják, hogy milyen módon őrizzék meg az adatok állapotát az adatbázison belül. Viszont előfordulnak olyan adatbázis típusok, amelyek rendelkeznek beépített funkciókkal és segítenek észlelni a változást a betöltött adatokban.

A CDC jelentése change data capture egy olyan módszerre utal, amellyel azonosítani lehet a változó adatokat különböző módokon. Négy fajta módot különböztet meg amellyel azonosítani lehet az eltéréseket.

Az SCD vagyis slowly changing dimension a CDC-vel átfedésben van, konkrétan egy továbbfejlesztett változata, amit inkább az adattárházakban szoktak használni. Akkor érdemes használni amikor nem tudjuk, hogy az adatok mikor változhatnak meg egy adott betöltéskör vagy frissítéskor. Az SCD módszereit típusokra osztják alkalmazási szempontoktól függően.

- SCD 0.: Ez az alap verzió, ilyenkor a tervezés alatt meghatározásra kerülnek azok az adatok, amelyek biztos nem változnak meg és azok is amelyek felülírásra kerülnek. Ez adja a kiindulási pontot a többi típus felé.
- SCD 1.: Az 1-es típusban minden friss rekorddal felülírják a régebbi rekordokat.
- SCD 2.: A 2-es típusban egy frissülő adat bekerülése során új sor kerül létrehozásra. Ilyenkor az azonosítás kiegészítő adatokon alapul például két dátum típusú mező, amelyek arra használunk, hogy mettől meddig volt az adat éles. Amikor új adat kerül be akkor pedig az adott bekerülési dátummal az

előző rekordot megjelöljük. Az új rekord pedig attól kezdődően érvényesek. Általában ezt a Valid_From és Valid_To oszlopokkal oldják meg.

- SCD 3.: A hármas típusban a táblákban szereplő fontos adatmezők mellé létrehoznak egy olyan mezőt egyenként, amely tartalmazza az előző értéket. A frissítés során az eddigi értékek átkerülnek az előző értéket jelölő adatmezőkbe és az aktuális értéket tartalmazó mezők pedig felülíródnak az új értékekkel. A gyakorlatban ezt a módszert használják legkevésbé ugyanis bonyolult a megvalósítása és igen nagy erőforrás igényekkel jár a végrehajtása.
- SCD 4.: A négyes típusban az adatok a használatban lévő táblákban simán felülírásra kerülnek. Viszont minden táblának van egy másolata, amelyben csak a régi adatokat tárolják. Ezeknek a tábláknak a felépítése megegyezik az aktuális értékeket tartalmazó táblákkal, viszont tartalmaznak extra oszlopokat az azonosításhoz. Általában időpontokat, hogy az adatok mettől meddig voltak érvényesek.
- SCD 6.: A hatos típus az 1-es 2-es és 3-as típus kombinációja, amelyben alkalmazzák a dátummal való jelölést. hogy az adat mettől meddig érvényes, valamint egy külön oszlopot használnak logikai jelölésre, hogy több adat közül melyik érvényes jelenleg ezáltal bebiztosítva az egyszerű azonosítást. Valamint a régebbi adatokat itt is egy másik oszlopban tárolják, és a friss adat bekerül az felülírja az eddigi aktuális adatokat. A historizálás szempontjából, ez a legjobb megoldás viszont, nagy terhelést jelent az adatbázis számára. [17]

A SCD típusok azokra az adatokra fókuszálnak, amelyek megváltoznak a múlt során. Viszont előfordulhat olyan, hogy egy adat kikerül a forrás rendszerből, ahogy ezt említettem nem kerül törlésre, viszont ezt is érdemes külön jelzéssel kezelni, egyrészt egy dátummal, ami megmutatja, hogy meddig volt a rendszerben, valamint egy flaggel ami megmutatja, hogy törölve van-e az adat vagy sem.

5. ADATTÁROLÁS ÉS FELDOLGOZÁS ALAPJA

5.1 Adatforrás

Az analitikai rendszerekben, amelyeket a vállalatok használnak egy alap kérdés az adatok tárolásának módja. Míg maga a folyamat fejlesztésekor szinte korlátlan lehetőség áll rendelkezésre addig itt viszonylag korlátolt a választási lehetőség a tárolási módok közül. Az adatokból megkülönböztetünk strukturált, részben strukturált és nem strukturált adatokat. Ennek alapján szükséges kiválasztani a kívánt tárolás módot. Az is előfordulhat, hogy a több forrásban különböző forrás típusok is vannak. Ezért szükséges felmérni, hogy milyen átalakításokat lehet eszközölni. Ugyanis ezek között az adatforrás típusok között van átalakítási lehetőség.

- Strukturált adat: Olyan adatforma amely előre átláthatóan definiált adatmodellt képez. Oszlopokból és sorokból áll ezáltal táblázatban ábrázolható.
- Részben strukturált: Olyan adatforma ami formai szempontból elhagyja a strukturáltságat. Viszont logikailag tartalmaz struktúrát, amit elválasztásokkal tagol. pl.: JSON fájlban a tag-ek és egyéb jelzők. Ezeket másnéven leíró fájloknak is nevezik. Ezekkel a jelzésekkel kialakítható a kapcsolat az adatok között.
- Nem strukturált: A nem strukturált adat általában szöveg alapú és nincsen előre meghatározott konkrét adatmodellje és semmilyen formai alapjai. A való életben a legtöbbször ezek szöveges fájlok, számsorok.

Ha az adatainkat egységes formában szeretnénk tárolni az analitikai rendszerben akkor a nem strukturált adatokat szükséges strukturált formába átalakítani amennyire ez lehetséges. [18]

Az analitikai rendszerek általában relációs adatbázisokat használnak ennek az az oka, hogy a relációs adatbázisokban az adatelérés általában gyorsabban történik nagyobb számú adaton, mint a nem strukturált adatokat tartalmazó adatbázisokban (pl. MongoDB). Azonban minden a körülményektől függ, hogy mit érdemes használni. A másik oldalon például a növekvő adatmennyiség áll, ami kedvez a NoSQL adatbázisoknak ugyanis sokkal flexibilisebbek, mint az relációs adatbázis ugyanis ezek nem tartalmaznak konkrét sémát az adatokhoz. [19]

5.2 Python

A Python egy sokcélú, magas szintű programozási nyelv. Egyes információk szerint a legnépszerűbb programozási nyelv. A legelterjedtebb alkalmazása a webprogramozás valamint az adatelemzés és a gépi tanulással kapcsolatos területeken figyelhető meg. Ennek alapján döntöttem úgy, hogy ezzel a programozási nyelvvel szeretnék megvalósítani ETL folyamatokat, amelyekkel az adatokat akár különböző forrásból egy olyan rendszerbe lehet integrálni, amely kiszolgálhat egy analitikai rendszert vagy adattárházat.

Az okai amiért szerintem alkalmas lehet az adatfeldolgozást végezni ezzel a nyelvvel:

- Könnyű fejlesztés, gyorsan tesztelhető kódok.
- Alapból sok fájl formátum támogatottsága (JSON, XML, CSV)
- Rengeteg harmadik féltől származó kiegészítő könyvtár, amely hasznos funkciókkal bővíti a nyelv lehetőségeit.
- Alapból támogat adatbázisok, mint pl.: SQLite vagy PostgreSQL de ez kiterjeszthető több SQL és NoSQL adatbázisra.

Az említett könyvtárakkal a Python tovább bővíthető és sokkal több funkciót képes ellátni, emiatt van nagy lehetőség ebben a nyelvben. Fontos megjegyezni, hogy érdemes választani valamilyen kiegészítő könyvtárat ugyanis ezek segíthetik az adatfeldolgozás és transzformálás műveletét a folyamat során. A legfőbb segítséget ezek a csomagok az átalakítások során nyújthatják így érdemes arra fókuszálni.

5.3 Pandas

A pandas egy 2008-tól kezdődően kiadásra kerülő Python csomag, amelynek fő célja, hogy hasznos funkciókkal lássa el azokat, akik adatok feldolgozásával átalakításával és elemzésével akarnak foglalkozni. Egy magas szintű adatmanipulációt lehet vele megvalósítani. Előnye, hogy igen gyors a memóriában történő műveletvégzésnek köszönhetően. Az egyik alap csomaggá vált az iparban, ahol valamilyen analitikai rendszert használnak Python alapokon. Igen flexibilis ugyanis sok formátumot támogat, és az azok közötti átalakításokat is lehet kezelni vele pl.: SQL elérése, CSV fájlok, Excel fájlok, HDFS formátum. [20]

Az egyik előnye, amivel igazán kiegészíti a Python környezetet azok a dataframe-ek. Ezek olyan objektumok, amelyek 2 dimenziós adattárolásra adnak lehetőséget, úgy érdemes elképzelni, mint egy adatbázis táblát. Egy dataframe-et elő lehet állítani nyers fájlokból, amik megfelelnek egy kétdimenziós táblázatnak és rendelkeznek oszlopnevekkel. Valamint a Pythonban használt egyéb adattípusokból is elő lehet állítani ilyen dataframe-et.

A népszerűség alapján egy tökéletes csomagnak tűnik a Python bővítésére, pár kisebb hátránya is van. Az egyik, hogy nagy méretű adatoknál lassú lehet a típuskonverzió a dataframe-en belül. A másik pedig, hogy adott műveletekhez eltérő szintaktika szükséges, mint az alap Python nyelvben használt. Ez utóbbi inkább személyes preferencia kérdése. [21]

A pandas csomag sok olyan beépített adat transzformációs eljárást tartalmaz, ami segítséget nyújthat az ETL folyamat létrehozásához, amik igen hatékonyan működnek. Pl. de duplikálás. dataframe-ek összeillesztése, felbontása, szétvágása, dátumformátumok átalakítása. Ezekkel már alap transzformációs műveletek megvalósíthatóak amik a transform folyamathoz szükségesek.

Érdemes összeszedni szempont előtt tartott tulajdonságokat:

- Egyszerű a működése, és a fejlesztése
- sok dokumentáció található hozzá
- gyors az adatmanipulációs sebessége mindaddig amíg nem túl sok az adat
- bonyolult szintaxis
- futása alatt csak egy szálat képes használni
- mivel a lokális számítógépen fut ezáltal az összes adatot a RAM-on belül tárolja

5.3 Apache Spark

Az apache spark egy egyesített analitikai eszköz, amelyet nagy adatmennyiség feldolgozására használnak. Scala alapokon íródott viszont készült hozzá több API így lehetséges például R-ben és Java-ban is használni. A fő célja, hogy az adatfeldolgozást és az adatelemzési műveleteket egységes rendszer segítségével lehessen elvégezni. Ennek okaként gyors működés szükséges. Az egyik nagy előnye, hogy a memóriában tárolja az adatokat ezért sokkal gyorsabb adatfeldolgozási sebességgel rendelkezik, mint például a Hadoop. [22]

A Spark az RDD használatára épül, ami a resilient distributed dataset rövidítése. Ez az alap adatstruktúrája a spark-nak. Az RDD egy csak olvasható elosztott tárolása az adatoknak, amely lusta kiértékeléssel dolgozik. Ez azt jelenti, például, ha a műveleteket hívjuk meg az adatokra, csak akkor fognak lefutni amikor ezeknek az eredményeit vissza akarjuk kérni. Ezáltal hatékonyabb működése lesz a spark-nak. Mivel az RDD-k nem megváltoztatható struktúrák ezért minden átalakításnál új RDD jön létre, ennek köszönhetően visszakövethető az adatátalakítási folyamatunk. Ennek köszönhetően a Spark egy hibatűrő rendszer. [23]

A hibatűrés képességét tovább növeli amikor több klaszteren fut az alkalmazás. Ilyenkor Master-Slave kapcsolatban vannak egymással a gépek. Ebben a helyzetben a cluster manager osztja ki a jobokat a node-ok közt így valósul meg az elosztott adattárolás.

Spark architektúra részei:

- Spark core: A Spark alap részegysége, a core szolgáltatja az elosztott adattárolás funkcióit, I/O műveleteket, job ütemezőket, RDD kezelését. Ennek alapjaira épül fel az egész rendszer, ami elérhető több nyelven is az API-on keresztül.
- Spark SQL: Az egyik leghasznosabb eleme a rendszernek. A core-ra épül és a dataframe-ek kezeléséért felelős, valamint SQL támogatást biztosít, aminek használatával SQL segítségével is lehet adatátalakítást végezni. A Strukturált adatok feldolgozására érdemes használni ugyanis SQL támogatással lehet optimalizálni a műveleteket.

- Spark Streaming: A Spark core segítségével egy adatfolyamot kis adagokra felbontva juttatja a rendszerbe, amelyet maga a Spark core dolgoz fel és a feldolgozott adat ugyanúgy kis adagokban folyamatosan kerül ki a rendszerből.
- Spark MLIB: Magasszintű gépi tanulás algoritmusokat tartalmazó egység, amelyeket gyakorlatban lehet alkalmazni éles adatelemző rendszereken. Osztályozó, regressziós, klaszterező algoritmusok találhatóak meg benne. Fontos, hogy a machine learning algoritmusokat nem RDD-re hanem dataframe-re alkalmaz alapból a Spark. [24]

Az látható, hogy a Spark tartalmaz minden lehetőséget, amellyel létre lehet hozni egy adatintegrációs eszközt. Kifejezetten erre a célra is tervezték és ami előny, hogy lehet standalone üzemmódban futtatni, viszont egy éles vállalati környezetben minden bizonnyal nem így fog működni. standalone módban viszont gyengébb lesz a teljesítménye ugyanis csak annyi adatot bír feldolgozni amennyi a memóriában elfér az adott számítógépen. A Spark a források alapján kifejezetten alkalmas lehet ETL vagy ELT folyamatok létrehozására is, viszont szem előtt érdemes tartani, hogy ez a rendszer akkor hatékony amikor minél több adat feldolgozására használják.

Egyes információk szerint igazából nincsen felső adatmennyiségi korlátja a rendszernek, hanem a feldolgozási idő és a szerver kapacitások szabnak határt az adatfeldolgozásnak.

Apache Spark fontosabb tulajdonságai:

- Több célú eszköz, mint a Pandas.
- Nagyobb adatmennyiségen gyorsabb feldolgozásra képes, mint a Pandas.
- Alkalmas adatfeldolgozó folyamatok létrehozására és sokkal hatékonyabban használható ilyen célokra, mint a Pandas.
- Az Apache Spark-ban megtalálható beépített gépi tanulás alapú alkalmazások viszont ezek igen kezdetleges, ezért inkább érdemes kombinálni egyéb library-val a python rendszerből pl.: scikit learn.

Mind a Pandas és a Spark is egy hasznos eszköz, még a Spark egy komplettebb területet fed le addig a Pandas inkább egy kiegészítő eszköz, amely leginkább az adatelemzés és a gépi tanulás területén hasznos. Az Apache Spark összekapcsolható egyéb Python csomagokkal ezáltal sokrétű rendszert lehet kivitelezni a segítségével, ezt a Spark Python API segítségével, vagyis a pyspark-al lehet megtenni.

6. KÖVETELEMÉNYEK MEGFOGALMAZÁSA

A fejlesztés célja, hogy a program alkalmas legyen különböző forrásokból az adatok integrálására egy célrendszerbe a megfelelő követelmények teljesítésével. A célrendszer alatt jelen esetben egy adatbázis rendszer értendő. Az fejlesztés során szükséges a feldolgozás folyamatának a megtervezése és a kialakítása. Várhatóan a feldolgozás folyamatának a megtervezése és kivitelezése fogja a legtöbb időt igénybe venni. Az adatok, amelyek feldolgozva lesznek a folyamatok során csupán példa adatok, ettől függetlenül a rendszernek alkalmasnak kell lenni az egyszerű átalakíthatóságra, aminek köszönhetően könnyen használható más adatokon. A rendszer megtervezésében figyelembe kell venni, hogy több futtatási lehetőséggel is számolni kell, pl.: lehessen időzítve futtatni, automatikusan vagy teljesen manuálisan futtatni a feldolgozást. A rendszer futása során szolgáltasson információkat a feldolgozás jelenlegi állapotáról, valamint arról, ha elkészült az adatok feldolgozásával.

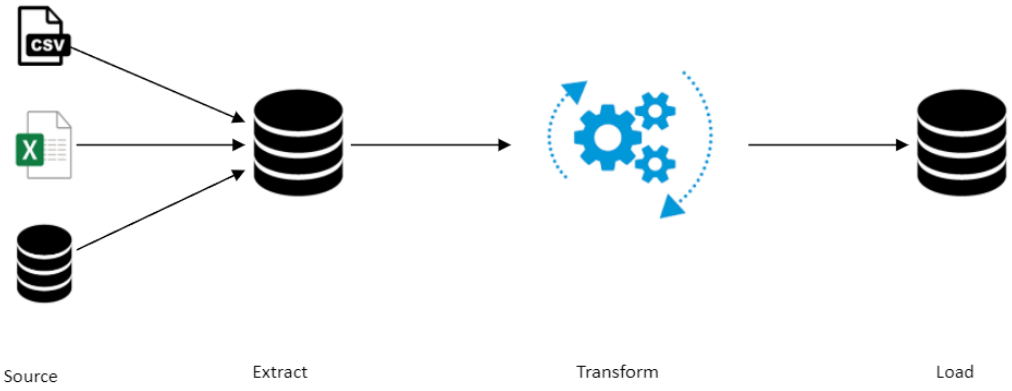
- A program funkciót tekintve képes végrehajtani egy teljes ETL adatintegrációs folyamatot.
- A folyamat során az adatokat a megadott forrásokból kinyeri majd az átalakítás után betölti az erre kialakított cél adatbázisba.
- A feldolgozás pyspark és SQL segítségével történik.
- A futtatás indításánál legyen lehetőség paramétereket megadni a program végrehajtási módjának kiválasztásához
- Az extract folyamat, full extract jellegű végrehajtást használ.
- A transform folyamat adatbázisok közti átalakítást használ.
- A transform folyamat során a szükséges adatok legyenek megjelölve a nyomon követhetőség érdekében SCD 2-es típusú módszert alkalmazva
- A load folyamat teljes betöltést fog futtatni az adattárházban.
- A feldolgozás folyamatáról adjon információt a felhasználónak.
- A program konzolos alkalmazásként fog futni.
- Egy esetleges feldolgozási hiba esetén kapjon a felhasználó információt a hibáról.
- Az extract folyamat után mentse el a forrás fájlokat az esetleges adatvesztés megelőzése érdekében.

7. TERVEZÉS

A rendelkezésre álló információk alapján amellett döntöttötem, hogy a rendszer a pyspark alapjaira fog épülni, amelyet a saját gépemre fogok telepíteni így standalone módban fogom használni. A fejlesztéshez a Visual Studio Code editort fogom használni, ami egy univerzális fejlesztő környezetet biztosít. A kódok egyszerű tesztelése érdekében ez ki lesz egészítve jupyter notebook-al ezáltal kódrészletek is lefuttathatók és tesztelhetőek.

Az adatbázisok kialakításához Microsoft SQL Server adatbázist fogom alkalmazni. Ennek egyik fő oka, hogy jdbc connection string segítségével könnyen összekapcsolható az Apache Spark rendszerével. Valamint könnyen kezelhető rendszerrel rendelkezik és hiba esetén egyszerűen visszaállítható az adatbázis.

Az ETL folyamat tervezése következik az adatbázisok megtervezése után. A folyamatok tervezése során a három különböző réteget külön hozom létre. Az extract folyamat során az adatok kinyerésre kerülnek egy tetszőleges forrásból, viszont a rendszert úgy szeretném elkészíteni, hogy különböző forrásokból is lehessen adatot kinyerni, ez egy fontos szempont az extract folyamat tervezése során.



5. ábra A tervezett ETL folyamat

A transform folyamat során az adatok átalakításra kerülnek, a tervezés közben ezeken az átalakításokon valószínűleg többször módosítani kell majd a tervezés során. Az ETL folyamatok tervezése során általában a transform réteg igényli a legtöbb időt. Ebben a folyamatban kerülnek majd kialakításra a kiegészítő adatok, amelyek bekerülnek majd a végső adatbázisba és az adatokról majd információval szolgálnak és az adatok nyomon követését is elősegítik.

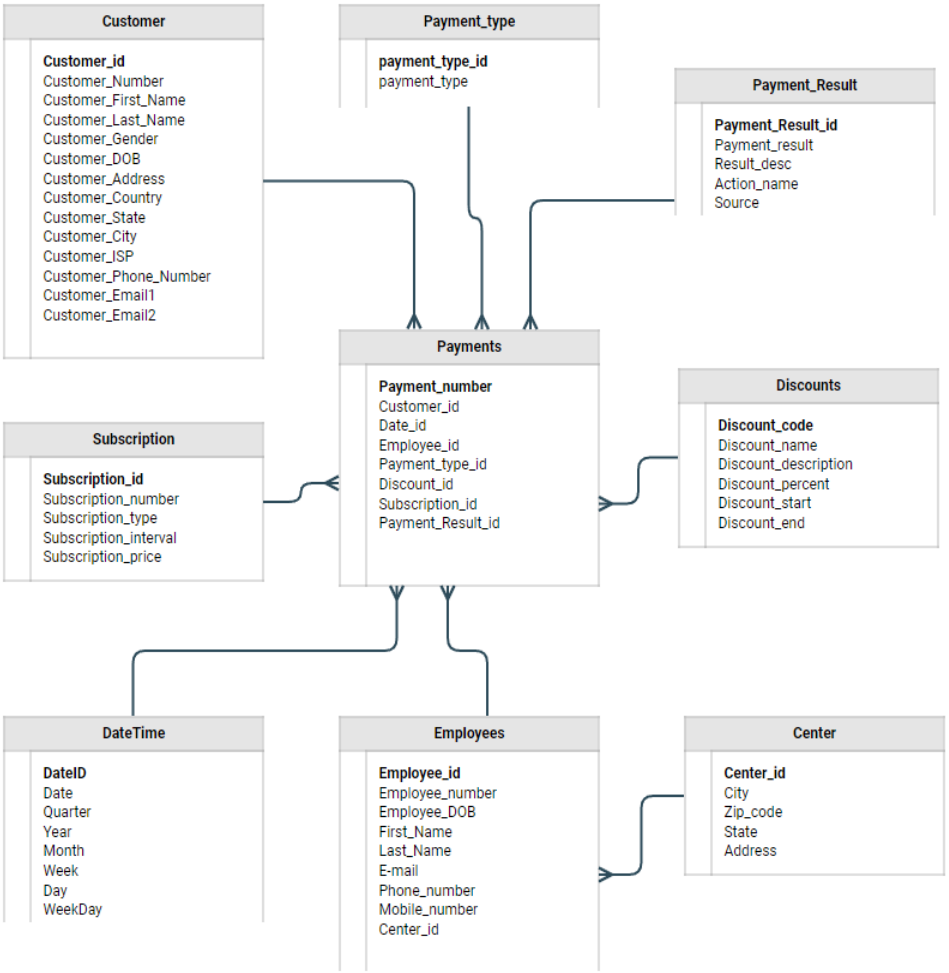
A load folyamat során az adatok betöltésre kerülnek a céladatbázisba amely azt a sémát fogja követni, amelyet kialakítunk a transform folyamat során. Ehhez szükséges megtervezni először az adatbázist, amely tárolni fogja az átalakított adatokat. Maga a

folyamat tervezése során pedig a legfontosabb, hogy az adatok áttöltési sorrendje és módja meglegyen határozva

7.1 Adatbázis tervezése

A meglévő adatok alapján szükséges kialakítani a cél adatbázis alap elrendezését. Az eddig megszerzett ismeretek alapján a legjobb megoldás egy egyszerűbb adattárház architektúra alapjának a kialakítása, amely egy adatpiaccal rendelkezik és egy adott területre fókuszál az adatok alapján. Egy olyan architektúra kialakítása, amely hibrid sémát követ és olyan tény táblával rendelkezik, amely kulcsokat tárol csak, ezt hívják tény nélküli tény táblának vagy factless fact table-nek. A meglévő adatok közül nem minden lesz felhasználva. Az adattárház az előfizetésekre és a felhasználók adataira fog fókuszálni ugyanis üzleti szempontból leginkább a cég vezetése ezekre az adatokra lehet kíváncsi. Így tehát az eredeti adatokból a filmekhez kapcsolódó adatok nem kerülnek felhasználásra. Mivel ezek példa adatok, a valóságban ennél sokkal több adatot is lehet tárolni az előfizetőkről. Hasonló helyzetekben sok haszontalannak tűnő adatot is eltárolnak az adattárházak ugyanis nem mindig tudni előre, hogy mire lehet szükség a meghatározott adatok mellet. Természetesen érdemes egy adott határon belül maradni, hogy túl sok felesleges adat se kerüljön be a rendszerbe, ami irreleváns az üzleti információk szempontjából.

Az adatok rendelkeznek azonosító kódokkal van olyan adattábla amely alapból rendelkezik elsődleges kulcsnak alkalmas adattal, viszont érdemes kijelölni ezek a számok mellet saját elsődleges kulcsnak való oszlopot, amelyeket a transform folyamat során kell majd létrehozni. Ha van egy alap azonosítónk akkor érdemes ezt abból előállítani vagy létrehozni például sorban növekvő számot.



6. ábra A kialakított hibrid séma

7.1.2 Adatok bemutatása

Adatmező neve:	Típus:	Tartalom:
Customer_id	int	Az előfizető id-ja
Customer_Number	int	Az előfizető azonosító kódja
Customer_First_Name	varchar	Az előfizető keresztneve
Customer_Last_Name	varchar	Az előfizető vezeték neve
Customer_Gender	varchar	Az előfizető neme
Customer_DOB	date	Az előfizető születési dátuma
Customer_Address	varchar	Az előfizető címe
Customer_Country	varchar	Az előfizető országa
Customer_State	varchar	Az előfizető állama/megyéje
Customer_City	varchar	Az előfizető városa
Customer_ISP	varchar	Az előfizető internet szolgáltatója
Customer_Phone_Number	varchar	Az előfizető telefonszáma
Customer_Email1	varchar	Az előfizető elsődleges e-mail címe
Customer_Email2	varchar	Az előfizető másodlagos e-mail címe

7. ábra A vásárló/felhasználó adatai

Adatmező neve:	Típus:	Tartalom:
Subscription_id	int	Az előfizetési mód elsődleges kódja
Subscription_number	int	Az előfizetési mód azonosító kódja
Subscription_type	varchar	Az előfizetés módja
Subscription_interval	varchar	Az előfizetés időtartama
Subscription_price	bigint	Az előfizetés ára

8. ábra Az előfizetés adatok

Adatmező neve:	Típus:	Tartalom:
Payment_Number	int	A fizetési adatok azonosító száma
Customer_id	int	A felhasználó adatok idegenkulcsa
Date_id	int	A dátumdimenzió idegenkulcsa
Employee_id	int	Az alkalmazottak tábla idegen kulcsa
Payment_type_id	int	A fizetés típus tábla idegen kulcsa
Discount_id	int	A Kedvezmények tábla idegen kulcsa
Subscription_id	int	Az előfizetés tábla idegen kulcsa
Payment_Result_id	int	A fizetés eredmény táblának az idegen kulcsa

9. ábra Ténytábla adatai

Adatmező neve:	Típus:	Tartalom:
Payment Result_id	int	A fizetés eredményadatainak kulcsa
Payment Result	int	Fizetési eredményadatok azonosítója
Result_desc	varchar	Az eredmény leírása
Action_name	varchar	A teendő neve
Source	varchar	Fizetési hiba forrása

10. ábra Fizetés eredményeiről az adatok

Adatmező neve:	Típus:	Tartalom:
payment_type_id	int	Fizetési mód elsődleges kulcsa
payment_type	varchar	Fizetési mód

11. ábra Fizetés típusának az adatai

Adatmező neve:	Típus:	Tartalom:
Center_id	int	A központ kulcsa
City	varchar	A város neve
Zip_code	varchar	A város irányítószáma
State	varchar	Állam/Megye
Address	varchar	Cím

12. ábra Központ adatai

Adatmező neve:	Típus:	Tartalom:
DateID	int	A dátumdimenzió elsődleges kulcsa
Date	date	A teljes dátum
Quarter	int	A negyedév száma
Year	int	Az évszám
Month	int	Hónap száma
Week	int	A hét száma
Day	int	A nap száma
WeekDay	int	A hét hányadik napja.

13. ábra Dátum dimenzió adatai

Adatmező neve:	Típus:	Tartalom:
Employee_id	int	Alkalmazott tábla elsődleges kulcsa
Employee_number	int	Alkalmazott azonosítója
First_Name	varchar	Keresztnév
Last_Name	varchar	Vezetéknév
Employee_DOB	date	Születési dátum
E-mail	varchar	E-mail cím
Phone_number	varchar	Telefonszám
Mobile_number	varchar	Mobilszám
Center_id	varchar	Idegen kulcs a központra hivatkozva

14. ábra Az alkalmazottak adatai

Adatmező neve:	Típus:	Tartalom:
Discount_code	bigint	Akció azonosítója
Discount_name	varchar	Akció neve
Discount_description	varchar	Akció leírása
Discount_percent	float	Akció értéke
Discount_start	date	Akció kezdete
Discount_end	date	Akció vége

15. ábra Leértékelés adatok

7.2 Alkalmazás tervezése

A program futása során az ETL folyamatok tervezése és megvalósítása veszi várhatóan a legtöbb időt igénybe. Ezek közül is a legtöbb időt veszi igénybe a Transform része az ETL folyamatnak. A program fő célja, hogy adatbetöltést valósítson meg egy céladatbázisba.

A futtatható programnak szükséges egy keretet készíteni, amely kezeli a konkrét folyamatot. A fő cél, hogy a program egyszerűen futtatható legyen és konfigurálni is egyszerű legyen a futtatás előtt. A program konzolos alkalmazás formájában fog futni és parancssorból lehet indítani. A konfigurációt egy fájlon keresztül lehessen állítani. Erre egyszerű használni egy Excel fájlt, vagy egy config fájlt. Ebben a fájlban lesznek eltárolva a fájlok adatai, amelyek alapján betöltésre kerülnek az adatok. A fájl tartalmazni fogja a forrásfájlok nevét az elérési útját, az elválasztó karaktereket fájlon belül és a cél adatbázis tábla nevét, amelybe betöltésre kerülnek az adatok.

Az előzetes terveim alapján, a program futása során ellenőriz egy kijelölt mappát, amelybe a beérkező fájlok kerülnek. Amint ez az esemény megtörténik, az adott fájlok betöltésre kerülnek. A fájlok betöltésének folyamata függ attól, hogy melyik adatbázis táblába tartoznak, ennek oka, hogy az adatbázis táblái hibrid sémába rendeződnek. A dimenzió táblák betöltésére kerül sor elsődlegesen és ha ezek végbementek akkor kerül sor a ténytábla betöltésére.

A program felépítését, amely kezeli a folyamatokat, érdemes jól széttagolt osztályokra bontani és megfelelően felépíteni az objektumokat, amelyek az egyes részekért felelősek. A python nyelv sajátossága, hogy igen nagy szabadságot enged a fejlesztőnek és objektum orientált használata is igen hatékony. Ennek következtében az egyes részegségeit a programnak érdemes a célnak megfelelő módon elkészíteni, vagyis a részegységet osztályokba rendezni.

Maga a program, amely vezérli a folyamatokat objektum orientált elvet fog követni, így átláthatóbb kódot lehet készíteni. A program alapja a fájlok kezelését és a pyspark kódok vezérlését fogja végrehajtani. A konkrét feldolgozó folyamatokat is lehetne script

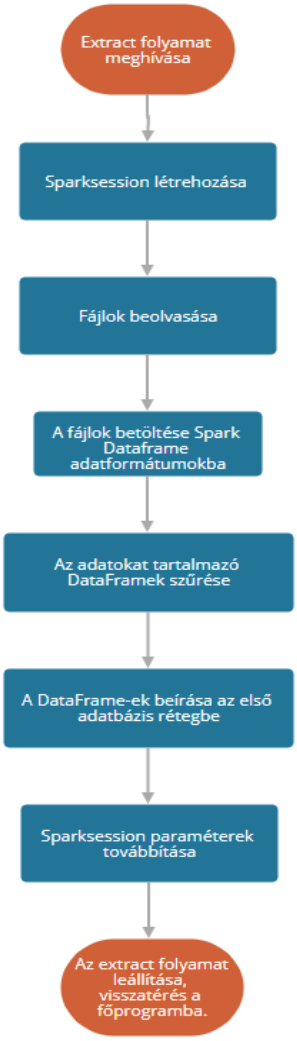
formájában futtatni viszont szerintem elegánsabb ezeket is osztályokba rendezni. A pyspark használata körülményes olyan szempontból, hogy figyelni kell a sparksession helyes meghívására, ami tulajdonképpen a belépési pontja az apache spark-nak és amint a sparksession létrejön onnantól kezdve érjük el a funkcionalitásokat. Az átláthatóság érdekében a spark belépési pontjának létrehozásához is külön osztályt érdemes létrehozni, amely akkor kerül meghívásra, ha az adott bejövő fájl beazonosításra kerül a program által. A bejövő fájl beazonosítása után érdemes elvégezni egy ellenőrzést, amely során a program megerősítést kap, hogy a kritériumoknak megfelelő fájlt talált az ellenőrzött mappában. Ezt elérési út ellenőrzéssel, valamint fájlnév ellenőrzéssel és fájltípus ellenőrzéssel lehet elvégezni. Miután bebizonyosodott, hogy a kritériumoknak megfelelő fájl került a mappába ezután elindul a feldolgozás és a program fő részéből az extract folyamat meghívásával fog folytatódni a program.

Egy adott fájl feldolgozása utána a visszakövethetőség érdekében az adott fájlt érdemes lehet archiválni és időbélyeggel ellátni. Ezt meglehet oldani amikor a fájl beolvasása megtörténik az extract folyamatban. Ekkor a fájlt áthelyezhetjük egy másik könyvtárba.

7.3 Extract folyamat

Az ETL folyamat első része, amely az adatok beolvasásáért felelős az egyik legegyszerűbb folyamat. A jelenlegi esetben leghatékonyabb módja, hogyha minden adatot ki olvasunk a forrás fájlból. A legnagyobb bejövő fájl kb. 250 000 rekordot tartalmaz ezt a pyspark könnyedén kezelni tudja egy dataframe-ben. Az Extract folyamat végén az adatok adatbázisba kerülnek betöltésre, amelyből a transform folyamat során kerülnek kiolvasásra és átalakításra. A 16. ábra mutatja az extract folyamat ábráját.

A forrás fájl beolvasása már a pyspark segítségével történik, ehhez már szükséges az extract folyamatnak paraméterként átadni egy spark objektumot, amin keresztül létre lehet hozni egy sparksession-t és elérhető válnak a funkcionalitások. A fájlok beolvasása függetlenül történik egymástól így lehet garantálni, hogy mindegy legyen milyen sorrendben és mikor kerülnek be abba a mappába, ahonnan beolvasásra kerülnek csak az legyen a lényeg, hogy a frissítéshez szükséges fájlok bekerüljenek.



16. ábra Extract folyamat ábrája

7.4 Transform folyamat

A transform folyamat rendelkezik a legnagyobb erőforrás igénnyel az ETL folyamatok közül. Itt történik a lényegi adatfeldolgozás és átalakítás. Jelen esetben az adatok megfelelő struktúrába rendezése az adatpiachoz. Valamint itt fog történni a historizálás kialakítása, amely szükséges az adatok nyomon követéséhez. Mivel generált adatokkal dolgozok így nem biztos, hogy maguk az adatok hasonlítanak egy valós helyzethez. Viszont eddigi munkatapasztalataim alapján az adatok átalakítása során a legtöbbször az elsődleges kulcsokkal és üres adatmezőkkel szokott probléma lenni, valamint az adatkonverziós hibák fordulnak elő a legtöbbször. Adott táblákban lesz generálva egy helyettesítő kulcs (surrogate key) amely magának a táblának az elsődleges kulcsa lesz. Valamint a folyamat során a Transform részei külön lesznek szervezve és adott metódusokból fog összeállni a folyamat, amely bemenetként egy dataframe-et kap és kimenetként a megváltozott dataframe-et kell visszaadnia eredményként. Ez csak egy lehetséges megoldása az adatok átalakításának. Azért gondolom, hogy ez alkalmas

módszer mert a pyspark rengeteg beépített, előre implementált művelettel rendelkezik. Meglehetne csinálni azt is, hogy maguk a műveletek vannak megírva és adott táblára az arra szükséges műveleteket hívja meg a program, ez egy elegánsabb megoldás, de ha szükség van egy új műveletre akkor azt el kell készíteni és beépíteni a feldolgozásba. A transform folyamat elején a megadott adatbázisból kiolvasásra kerülnek az adatok. Mindegyik kiolvasott adatot a forrástáblájuk alapján lehet azonosítani ezáltal egyből el is kezdődhet a maga a feldolgozás. A transform folyamat végén, az adatok ideiglenesen adatbázisba írásra kerülnek, ugyanis az adatokat köztes módon valahol tárolni kell a cél adatbázisba írás előtt és addig még a rendszer dolgozza fel az aktuális adatokon kívüli adatokat. A transform folyamat ábrája a 17. ábrán látható.

Customer tábla:

A legelső tábla egyben a legnagyobb ebben a bejövő fájlban az ügyfelek adatai lesznek eltárolva, ezt a fájlt beolvasva nem kerülnek maguk az adatok átalakításra csak alap szűrések lesznek elvégezve a terveim szerint. Ez a folyamat először is a duplikációk törlésével kezdődik, ahol a többször szereplő sorok törlésén van a lényeg. Ez a folyamat az elsődleges azonosító alapján fog keresni, amely az ügyfél száma lesz. Majd olyan sorok törlése következik, ahol a releváns adatok értéke null, pl.: Az ügyfél nevei vagy születési dátuma.

Subscription tábla:

A subscription tábla bejövő fájljaiban a feliratkozásokra való adatok vannak elhelyezve. Ebben a táblában az adatok minden fizetési tranzakcióhoz vannak kapcsolva, amelyeket a Payment number jelez. A Payment number alapján át kell vinni a feliratkozás típusait majd a tény táblába. Természetesen ezek előtt végezni kell egy ismétlődés ellenőrzést és a NULL értékek ellenőrzését. Ezek után a táblában csak az adott előfizetési lehetőségek lesznek viszont hozzá lesznek kapcsolva a ténytáblához. A subscription táblánál is meggondolandó a csere kulcs használata, ez a forrásfájlokban a payment azonosítóval együtt található ezáltal azonosítható lesz mindig a helyes érték a táblában a normalizálás után is viszont én érdemesnek láttam a csere kulcs használatát így nyomon követhető az árváltozás az táblában.

Payment type tábla:

Ez a tábla tartalmazza a fizetési lehetőségeket. Ebből a táblából vannak hozzárendelve a fizetési lehetőségek a tényadatokhoz. Ez a tábla fix, nem fog változni előreláthatólag csak akkor, ha létrejön egy új fizetési mód.

Payment result tábla:

Az adott tranzakciókhoz tartozó fizetési eredmények csoportosítása, viszont a nyers fájl nem normalizált adatokat tartalmaz ezért, ezeket is át kell rendezni. Ezt az átrendezést a payment_number mező alapján lehet megtenni. A tény tábla adatainak betöltésénél

lesznek ezek az adatok normalizálva. A csere kulcs használata érdemes, ugyanis ez az adat a valós helyzetben változhat és érdemes erre felkészülni, hogy ne legyenek kulcsütközések a változás esetén.

Discounts tábla:

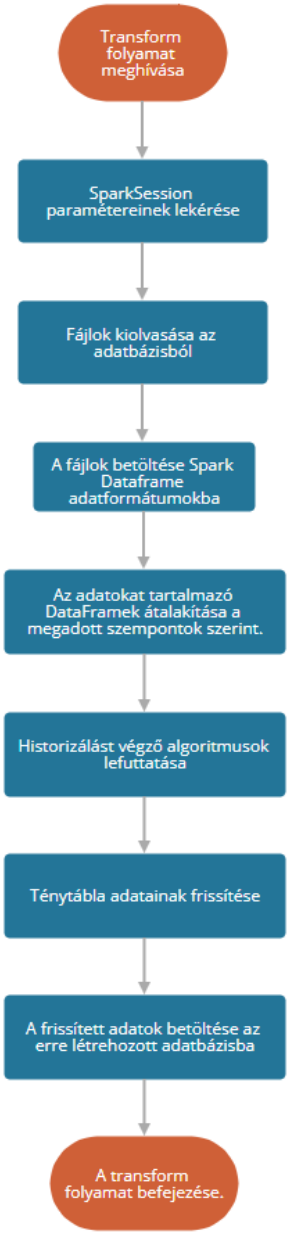
A kedvezményeket tartalmazó tábla. Ebben a táblában is kevés rekord van és hosszú időnként frissül. Az elsődleges kulcsát tartalmazza a tény adatokat tartalmazó nyers fájl, amely alapján összeköthető, hogy az adott tranzakcióhoz egy felhasználónak volt-e kedvezménye vagy sem. Ezt a táblát nem szükséges érvényességet jelző adatokkal ellátni ugyanis a leértékelések rekordjai közt szerepel az esemény vége és eleje ebből alapból lehet következtetni arra, hogy meddig volt aktuális egy árengedmény. Valamint időszakosak az engedmények és utólag nem lenne értelme, ha változnának.

Employees tábla:

A vállalaton belül azokat a dolgozókat tartalmazza, amelyek kapcsolatba kerülnek egy adott tranzakcióval. Alapesetben mindegyik tranzakcióhoz van rendelve egy dolgozó, mint technical support. Az employees tábla ritkán változik ezáltal nem lesz állandóan frissítve, de minden frissítés esetén null értékek ellenőrzésre kerülnek. Helyettesítő kulcsot ebben a táblában is létrehozunk Employee_id néven, mert a dolgozók adatai változhatnak.

Center tábla:

Az adott központok adatait tartalmazza, ebből viszonylag kevés található. Ezek az adatok az Employees táblához lesznek hozzákapcsolva, ez alapján lehet csoportosítani a dolgozókat munkahelyük szerint.



17. ábra A transform folyamat ábrája

7.4.1 Ténytábla adatainak frissítése

Egy kulcsfontosságú pont az adattárház működésében a ténytábla, amit a táblák összekapcsolására használnak az adatbázisban. A frissítésére különböző módok lehetségesek, jelenlegi terveim alapján a pyspark-on belül hasonló adat átalakításokkal kell betölteni az adatokat, mint a többi tábláknál. A ténytábla adatokat átalakítását meglehetne oldani az adatbázison belül is union utasítások segítségével, mert a ténytáblához szükséges adatok már szerepelni fognak a többi táblában, viszont a spark használata egyszerűbb ugyanis így nem terheljük külön az adatbázist a kiszolgáló gépen,

valamint a sparksession is rendelkezésre áll majd. Természetesen, ha megfelelő erőforrás áll rendelkezésre akkor ezeket a feladatokat nyugodtan rá lehet bízni az adatbázis szerverre és SQL utasítások keretein belül végrehajtani.

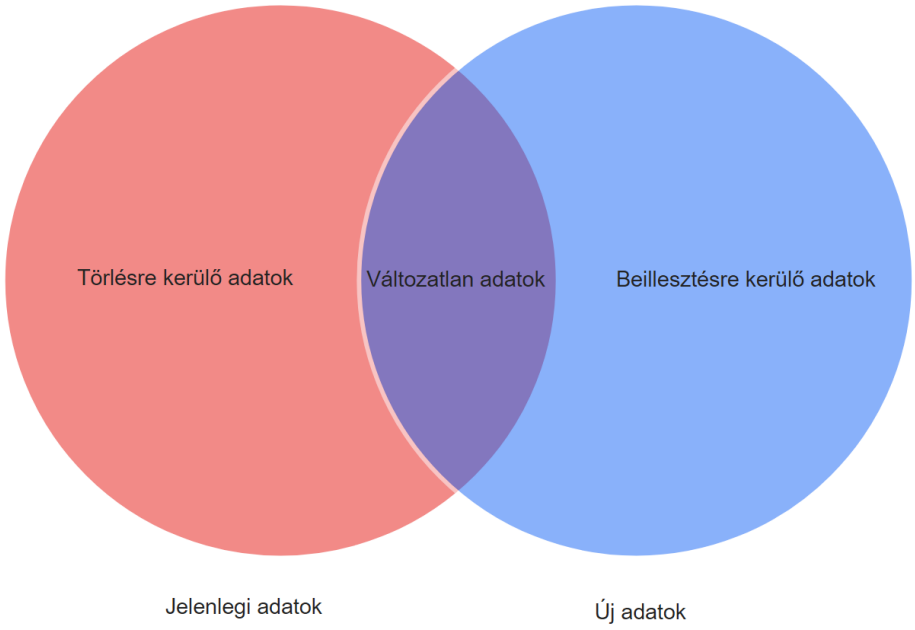
A ténytábla adatait legutoljára kell átalakítani a Payment_result valamint a Subscriptions tábla tartalmaival együtt, valamint a Customer tábla átalakított kulcsaival, ugyanis ilyenkor már meglévő dimenzió táblák adataira támaszkodik az átalakítás. A ténytáblába át kell helyezni azokat az adatokat, amelyek majd idegenkulcsként fognak szerepelni a táblában, de még a dimenzió táblákban szerepelnek. A táblák, amelyeket át kell rendezni, a rekordokhoz rendelt Payment Number érték van és ezek alapján át lehet csoportosítani a rekordokat. tény tábla adatait nem érdemes historizálni ugyanis egyszeri fizetéshez kapcsolódik az adatuk ez alapján vannak kialakítva a kapcsolatok. A historizálás helyett csak a már bent lévő tényadatokhoz történik a hozzákapcsolás union utasítás segítségével.

7.5 Slowly Changing Dimension

A historizálás kezelésére a slowly changing dimension 2-es verzióját választottam, szakirodalom válogatja, hogy hogy nevezik van, ahol a jelenlegi megoldást az SCD 7-es verziójának hívják. Ez nem a legegyszerűbb módja a korábbi rekordok nyilvántartásának. A program ez a része egy külön osztályt fog kapni. Ez az SCD osztály hozzáfér a sparkcontext-hez amelyet paraméteren keresztül kap meg. Ennek segítségével hozzáférhet a program az aktuális dataframe-hez és módosíthatja azt ezen az osztályon belül. A historizálás elvégzéséhez 3 adat lesz használva, CreatedTS amely a létrehozás időpontját jelöli, innentől érvényes az adott sor. A Valid_to jelöli azt az időpontot, hogy meddig érvényes az adott sor, alap esetben ez az adatmező null értéket kap. Valamint van egy isValid oszlop, amely bit típusú adatot fog tárolni, ez 1 értéket fog felvenni, ha érvényes az adat és 0 értéket, ha nem érvényes. A 18. ábrán látható ahogy a megváltozott rekord is benne marad a táblában jelölve az érvényességet. Ezt a módszert lehet alkalmazni simán a rekord törlésénél is.

Subscription_id	Subscription_number	Subscription Interval	Subscription type	Subscription price	CreatedTS	Valid_to	isValid
1	1	1	1 Month	10	2019-01-01	NULL	1
2	2	6	6 Month	40	2019-01-01	2021-12-31	0
3	2	6	6 Month	45	2022-01-01	NULL	1
4	4	12	12 Month	90	2019-01-01	NULL	1

18. ábra Az SCD ábrázolása

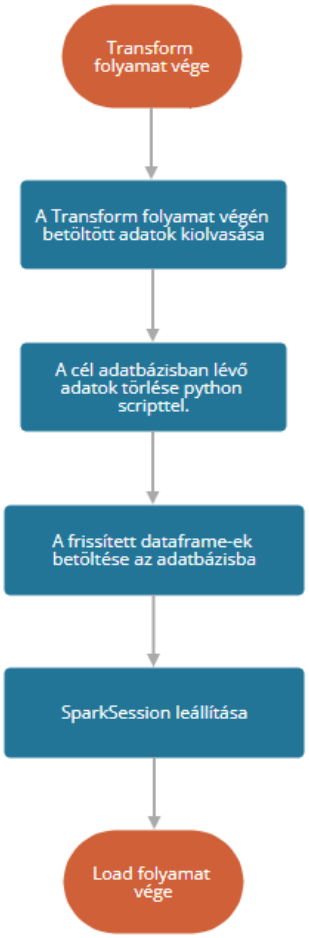


19. ábra Venn-diagram az adatok állapotáról

A program ezen részében érdemes szétválasztani az eseteket, hogy külön lehessen őket kezelni. Megkülönböztetésre kerül 3 eset, ha új rekord kerül a rendszerbe, ha egy sor törlésre kerül, valamint, ha frissítés történik egy sorban. Az esetek ellenőrzése olyan módon történik, hogy az osztály konstruktorában megkapjuk a friss adatot, valamint a jelenleg adatbázisban szereplő adatot kiolvassuk, mindkettő adathalmaz egy-egy dataframe-be kerül és összehasonlításra kerülnek, hogy történt-e valami változás. Ezt az azonosítást többféle módszerrel lehet elvégezni. A két dataframe összes oszlopát és sorát költséges folyamat lenne összehasonlítani egymással és a tábláknak különböző oszlopai vannak így nem lehetne újra felhasználható kódot írni az ellenőrzésükre. A kulcsok össze hasonlítása is lehetséges lenne viszont, ha már egy meglévő adat változik és csak a lényegi adat változik a sorban a kulcs nem akkor ezek változások nem kerülnek be az adatbázisba majd. Ennek megoldására létre lehet hozni egy új oszlopot mindkét dataframe-ben ideiglenesen, amely az adott sor hash értékeit tartalmazza és ezek alapján összehasonlítani az adott sor hash értékeit. Ugyanis, ha egy oszlopban akár egy rekord tartalma is részben változik akkor maga a hash értéke is változni fog. Ez alapján a hash alapján egyesíteni lehet a két dataframe-et. Az adatok egyesítése a pyspark union metódusával fog történni. Ezt a módszert az SQL-ből is union néven lehet ismerni és a működése is hasonló. Maguk az esetek megkeresése a táblákban viszont joinokkal fog megvalósulni. A 19. ábra szemlélteti Venn diagramm segítségével. A joinon kívül lehetne különböző select utasításokkal is ezt megoldani, viszont a join pont alkalmas erre a feladatra hiszen alapból is különböző táblák összekapcsolására használható mind a spark-ban mind az SQL-ben.

7.6 Load folyamat

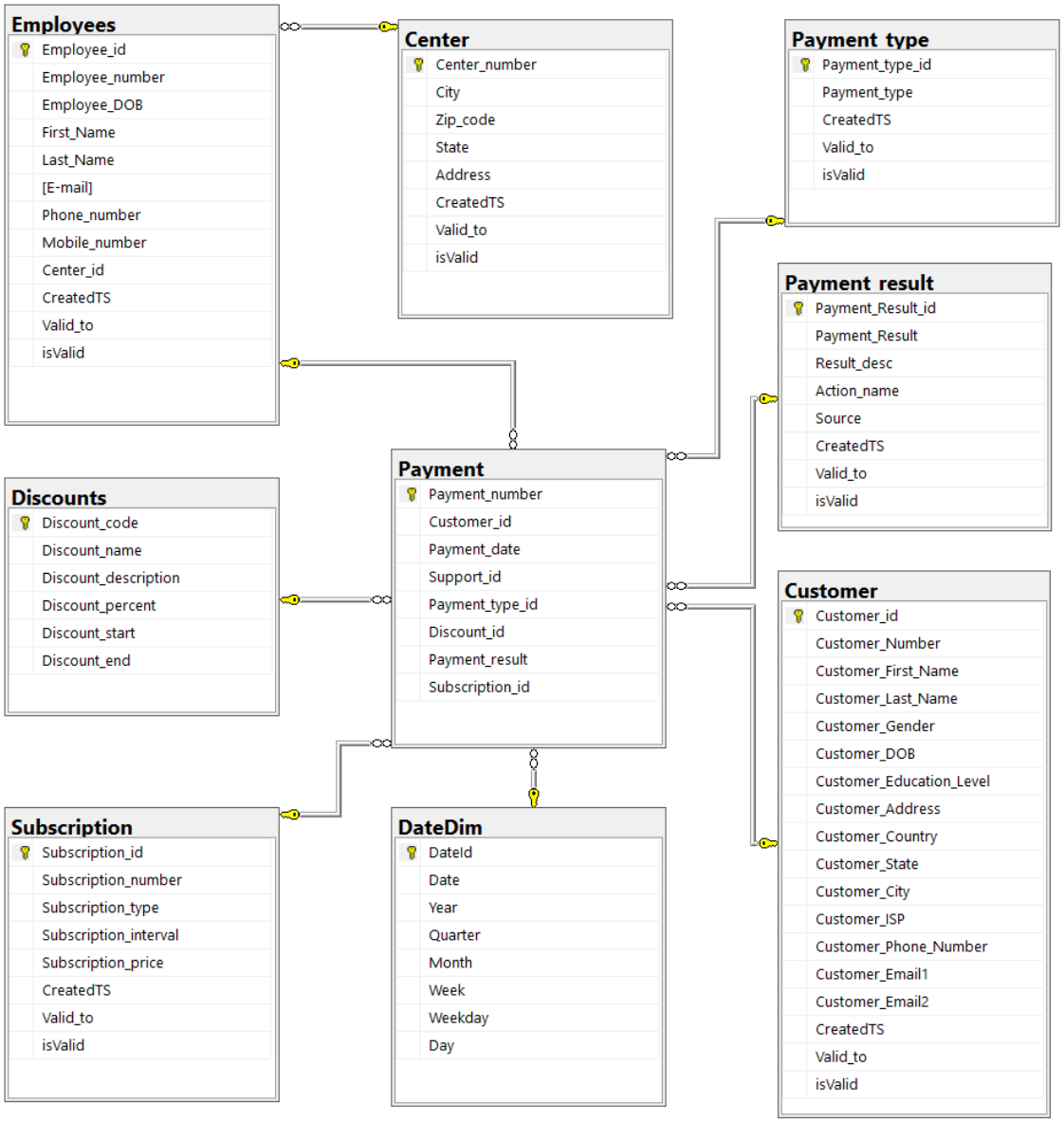
A Transform folyamat után az adatbázisba töltésre kerülnek az adatok. Az MSSQL adatbázis eléréséhez egy külön jdbc driverre van szükség, ennek segítségével érhető el az adatbázis az adott számítógépen. Személyes tapasztalat, hogy a spark-on keresztül az MSSQL elérése igen körülményes és új felhasználót kell létrehozni az SQL Serverben, amely adminisztrátori joggal rendelkezik az adatbázis felett. Erre azért van szükség mert alapértelmezett esetben az SQL Server Windows felhasználó alapján azonosít viszont ez a spark-on keresztül néha kiszámíthatatlanul működik. A transform folyamat végén beírt adatokat ki kell olvasni a köztes adatbázisból és betölteni az adattárházba. Ez azért szükséges mert az idegen kulcs megszorítások miatt probléma lenne a módosításokkal egyből a transform folyamat végén betölteni az adatokat a végleges adatbázisba. A load folyamat során a historizálás következtében törölni kell majd a rekordokat és újra betölteni azokat az adattárházba ahogy a 20. ábrán látható. A Spark több módot is megenged az adatbázisba írásnál, a jelenleg használt módszer az hozzáfűzés lesz. Ez az adatbázisba való írásnál hozzácsatolja az adatokat a jelenleg meglévő adatokhoz.



20. ábra Load folyamat ábrája

8. MEGVALÓSÍTÁS

A fejlesztés során a keretprogram elkészítésével érdemes kezdeni, ugyanis ez előreláthatólag nem fog változni érdemi lényegben megvalósítás során. Valamint érdemes létrehozni az adatbázisokat, amelyek az adatokat fogják tárolni. A folyamat 3 adatbázist fog használni az első kettőt ideiglenes adattárolásra. A harmadikat pedig a végleges adattárolásra, ez a 21. ábrán látható.



21. ábra Az SQL serverben elkészített adatbázis

8.1 Alap program megvalósítása

A programnak elsősorban szükséges létrehozni egy fő osztályt, amely kezeli a az egész folyamatot. Ebben a főosztályban helyezkedik el 4 metódus, amelyek a vezérlést és az azokhoz szükséges feladatokat látják el.

A program elindítását .py fájl futtatásával lehet megtenni. Van lehetőség .exe fájl készítésére viszont ahhoz a python fájlt fordítani kell egy plugin segítségével és úgy kell menteni. Ez a fordítási folyamat a csomagok betöltésével és elmentésével jár, ez jelen esetben az Apache Spark miatt több Gb-os fájlt eredményezne.

A main metódus ellenőrzi a kijelölt mappát, amelybe az input fájlok kerülnek és amennyiben változás történik akkor elindítja a feldolgozási folyamatot. A változás figyelése és azonosítása érdekében van szükség a másik 3 metódusra, amely ebben az osztályban található. A program elindításakor lehet paraméterezni magát a .py fájlt elsősorban a mappa ellenőrzési idejét lehet állítani ez alapesetben 30 másodperc, ha nem ad meg a felhasználó paramétert. Valamint lehet választani, hogy milyen módon ellenőrizze a bejövő fájlokat. Ebből a választási lehetőségből 2 van, az összes fájlt figyeli a rendszer, vagy csak a tranzakciós adatokat tartalmazó fájlokat. A kód alapvetően úgy épül fel, hogy egy osztályt csak egy dologra használunk kivétel ez alól a program fő osztálya, ahonnan a többi részegységek kerülnek meghívásra.

A program működéséhez szükséges beállításokat egyszerűen lehet változtatni, pl., hogy melyik mappát figyelje vagy ennek mi a kritériuma. Alap esetben egy olyan mappát keres, amelynek szerepel a nevében ’Drop’ és a programot tartalmazó könyvtárban helyezkedik el. Természetesen ezt meg lehet változtatni, de a fejlesztés és tesztelés során így egyszerűbb volt dolgozni vele.

A 6 metódus a fájlok vizsgálatáért felel:

- checkTable: Az osztály létrehozásakor kerül meghívásra konstruktorból és egy osztályszintű változóba menti el a fájlok ellenőrzéséhez szükséges táblát, amely a fájlokhoz tartozó adatokat tartalmazza. Ebben a fájlban található, hogy az adott fájl adatai majd melyik folyamatban lesznek feldolgozva.
- checkFiles: Amennyiben egy fájl került hozzáadásra abban az esetben a programnak szükséges lekérnie a fájlokat. A metódus összehasonlítja a táblában található fájlnevekkel és hasonlóságot keres. A kritérium, hogy a táblában található karakterláncot tartalmaznia kell a fájlnévnek azon kívül tartalmazhat pl. dátumot az adott fájl neve.
- getPath: Amikor azonosításra került egy fájl és az szerepel a táblában akkor továbbításra kerülhet feldolgozásra. Ehhez a spark-nak szüksége van egy elérési útra, ez ennek a kimentésére szolgál és ez az elérési út tovább adásra kerül majd string formájában.

- checkFileList: Annak az ellenőrzésére szolgál, hogy a szükséges fájlok feldolgozásra kerülnek, ami után elindulhat a Transform folyamat.
- moveprocessedFiles: Azért felel, hogy a fájlok a beolvasások után átkerüljenek egy olyan könyvtárba, ahol időbélyeggel lesznek ellátva, így megvizsgálható lesz, hogy mikor lettek feldolgozva.
- callTransform: Meghívja a transform folyamatot az extract folyamat befejezése után.

A program indításakor paramétert lehet megadni a parancssorban ami a fájlok ellenőrzésére van kihatással. Amennyiben „full” - paramétert kap a program akkor az összes fájlt elvárja az extract folyamat során, amennyiben „semi” paramétert kap a program akkor csak a tranzakciós fájlokat várja el a program, természetesen ez adatfüggő, hogy milyen táblákat várunk el a betöltés során. Ez azért fontos mert, ha adott fájlok nem érkeznek meg, akkor értelmetlen elindítani az adatok betöltését.

8.2 ETL folyamat megvalósítása

8.2.1 Extract

Az Extract folyamat megvalósítása a legkönnyebb egy szimpla osztályban létre kell hozni a megfelelő beolvasásért felelős python kódot a megfelelő pyspark modulok használatával. Az adatok egy köztes adatbázisba kerülnek innen, ez jdbc driver segítségével kerül beírásra az SQL Server adatbázisba.

Ennek a folyamatnak a megvalósításakor el kellet dönteni, hogy milyen típusban kerüljenek az adatok az adatbázisba. Ahol az extract folyamat végén az adatok egy köztes adatbázisba kerülnek sokszor a sima szöveges formátumot használják (jelen esetben varchar) ennek megvannak az előnyei és a hátrányai is. Ebben az esetben azt a döntést hoztam, hogy az adatok típusai automatikusan legyenek beállítva az extract folyamat során és ami szükséges az az SQL serverrel kompatibilis formátumra legyen állítva. Ilyenek például a dátumformátumok, ezt a nyers fájl beolvasásakor tudjuk megtenni a inferSchema lehetőséggel. Ez annyit tesz, hogy a pyspark automatikusan detektálja az adattípusokat és ennek megfelelően alakítja ki a dataframe típusait. A megoldás a 22. ábrán látható.

```
def Extract(self, path):
    #Loads the data from the source file
    df = self.sparksession.read.option("delimiter", ";")\
        .option("header", True) \
        .option("inferSchema", True) \
        .csv(path)
    self.Convert(df)
    pass
```

22. ábra CSV fájlból kiolvasás

Alapvető szűréseket is alkalmaztam, például a bejövő adatoknál mindennek van egy azonosítója, itt még nem elsődleges kulcsként szerepelnek az adatbázisban, viszont azok a sorok, amikből ez a szám hiányzik azok törlésre kerülnek, így megelőzhető a null értékek elsődleges kulcsként való beillesztése, természetesen azt a transform során is meg lehet tenni, viszont így ezzel a folyamattal nem terheli a program a későbbiekben a rendszert. Részben lehet validációt végezni, például a dátum egy adott értékek között szerepel, valamint adott értékeket vesz fel vagy null értékek ellenőrzése. Amit szükséges megtenni, még hogy azokat az adatforrásokat amelyekben van dátum egységes formában kell beilleszteni. Ezt eredetileg nem akartam majd csak a transform részben elvégezni, viszont 2 dátummezőnek az adatbázisba töltése a tesztek során igen kiszámíthatatlan volt, ezért szükséges egységes módon átalakítani. Ezt a 23. ábra szemlélteti.

```
elif self.tablename == 'dbo.Employees':
    df = df.dropna('all')
    df = df.where(col("Employee_number").isNotNull())
    df = df.withColumn("Employee_DOB",date_format(to_date(col("Employee_DOB")\
        ,"yyyy.MM.dd"),"yyyy-MM-dd"))
```

23. ábra A Employees adatok dátum normalizálása, valamint null érték kiszűrése

Az adatok köztes adatbázisba töltésénél figyelni kell arra, hogy felülírjuk az adatbázisban levő adatokat, így biztosítani lehet, hogy ebben az adatbázisban mindig a friss adatok legyenek. Ennek a betöltésnek módjáról a pyspark gondoskodik, viszont többféle lehetősége van, alapból 4 módot támogat a rendszer:

- **append**: Az adatbázisban meglévő adatokhoz hozzácsatolja az új adatokat.
- **overwrite**: Az adatbázisban lévő adatokat felülírja.
- **ignore**: Az adatokat beilleszti az adatbázisba viszont, ha létezik már egy rekord akkor azt ignorálja.
- **error**: Az adatokat beilleszti, viszont, ha már létezik egy rekord az adatbázisban, akkor hibával megszakad a betöltés (Amennyiben nem állítunk be opciót akkor ez az alapértelmezett.).

A fentebb található lehetőségek közül az overwrite mód lesz használva, itt viszont szükséges beállítani egy másik beállítást, amikor megtiltjuk a pyspark-nak hogy törölje a táblát ugyanis az overwrite működési módban alapból mindig kitörli a táblát és újra létrehozza ezt, így garantálva mindig az aktuális adatoknak megfelelő adattípust. Ez viszont költséges művelet ezért ezt szükséges megtiltani és így csak az adatokat fogja törölni, ez látható a 24. ábrán.

```
def LoadStage(self, df):
    df.write.format("jdbc") \
        .option("url", f"jdbc:sqlserver://localhost:1433;databaseName={self.database}") \
        .option("dbtable", self.tablename) \
        .option("user", self.user) \
        .option("password", self.password) \
        .option("driver", self.driverString) \
        .mode("overwrite") \
        .option("truncate","true") \
        .save()
```

24. ábra Az Extract folyamat végén az adatbázisba töltés kódja

8.2.2 Transform

A transform folyamat megvalósítását nagy részben az adatokat átalakító scriptek megírásai tették ki. Ennek a fázisnak a nehézsége, hogy a megvalósítás közben kell tesztelni a programot. A táblákat feldolgozó utasítások ütemezése megvalósítható olyan módon, hogy újra felhasználható legyen, viszont maguk a konkrét adatokat reprezentáló táblákra meghívott utasítások különböznek ezért ezek helyes működését a fejlesztés során folyamatosan ellenőriztem. A legtöbb hiba az adatoknál a típus különbségekből adódott. Ezek mind az átalakítás folyamán keletkeztek, és később a load folyamat során az végleges adatbázisba való töltéskor merült fel probléma. Ez abból adódott, hogy a pyspark táblák adattípusai nem egyeznek meg minden esetben az alap python és SQL adattípusokkal. Ezért különös figyelmet kell fordítani a program futása során a típuskonverziókra.

Az átalakítások során, először minden dataframe-re az adattisztítás folyamata kerül meghívásra, ebben az esetben a sorok törlése, ha rendelkezik egy adott oszlop null értékkel, ez tá változhat. Ezután a dátum formátumok konverziója. Egységesen minden dátum az ÉÉÉÉ-HH-NN sablont követi. A dátumok átalakítása után következik az egyedi átalakítás pl. Oszlopok összevonása vagy új oszlop létrehozása és feltöltése értékkel. Ezek után a historizálás meghívása következik. A helyettesítő kulcs (surrogate key) is ilyen módon készül el, viszont historizálás után, ezt mutatja a 25. ábra. Az adatok visszakereshetősége érdekében használt SCD-ről a 8.3-as pontban.

```
def TrCustomer(self, df , tablename):
    df1 = df.dropDuplicates(["Customer_number"])
    df2 = df1.withColumn("Customer_number",df["Customer_number"].substr(6,11).cast("int"))
    historized=scd.SCD('dbo.Customer',self.sparksession).SCDHandler(df2)
    historized = historized.dropDuplicates(["Customer_number"])
    historized = historized.withColumn("Customer_id",row_number().over(W.Window.orderBy("CreatedTS")))
    self.Load(historized, tablename)
    pass
```

25. ábra A Customer adatok átalakítása és a historizálás meghívása

A táblák feldolgozásának sorrendje függ az adattárház felépítésétől, jelen esetben ez azt jelenti, hogy a ténytábla kialakításához szükség van az előfizetés adataihoz, valamint a fizetési eredmények adatához ugyanis ezek nincsenek benne a tényadatokba a

forrásfájlokban. Ahhoz, hogy ezek az adatok bekerüljenek, átalakításokat kell végezni. Először is ezeket ki kell olvasni az adott dataframe-ből és át kell helyezni a tényadatokhoz, ezt mindkét esetben a Payment number alapján lehet megtenni. Ebben az esetben is join műveletet használtam. A join művelet sokkal egyszerűbb ugyanis csak a payment_number mezők alapján a feliratkozások és a fizetési adatok közül kiemelésre kerül az elsődleges kulcs, hogy majd lehessen hivatkozni rá a tény táblából. Ez a művelet teljesen hatékony a spark-on belül. Ugyanerre a problémára lehetne használni withColumn() műveletet is, ilyenkor egy új oszlopot hoznánk létre a tényadatok táblájában és ki kellene olvasni az adatokat a forrásból, ehhez a withColumn() utasításon kívül select utasítást is kellene alkalmazni és emiatt hatékonyabbnak láttam join műveletekkel megoldani a problémát.

Hasonló probléma szintén fennál az Employees tábla adatainál, itt viszont más okból. Az adatbázis táblák kapcsolata egymással csillagsémát formáz kivétel az Employees és Support_center táblákat. Emiatt ez már egy hibrid sémát alkot, természetesen egy adatbázis tervezése rugalmas lehet egy adott kereten belül és át lehetne szervezni ezeket az adatokat a ténytáblába az átalakítások során, viszont az én véleményem szerint ebben az esetben a táblák ilyen módon való elrendezése logikusabb, valamint jobb megoldás, ha a feltöltés módjára figyelünk. Mivel olyan típusú adatokról van szó, amely ritkán változik ezért is egyszerűbb ezt a megoldást választani.

8.2.3 Load

A load folyamat megvalósításához is hoztam létre külön osztályt ugyanis itt a táblák betöltésére kerül sor és maga a betöltési mód megegyezik az extract folyamat során használt köztes adatbázisba való betöltéssel, viszont az adatokat frissíteni kell az adattárházban. A load folyamat a legegyszerűbb a 3 közül a jelenlegi esetben ugyanis a Transform után az adatok feldolgozása egyszerre történik és ezáltal a load folyamatot is egyszerűen úgy kellet elkészíteni, hogy minden táblát kiolvasson a köztes adatbázisból és áttöltse az adattárházba. Amire figyelni kell, hogy a tényadatok betöltése maradjon a folyamat legvégére. A tervezés során említésre került, hogy az adatokat ki kell törölni a cél adatbázisból. A pyspark adatbázisba töltésének kezelése miatt, ugyanis az adatokat a historizálás miatt mindig frissíteni kell az adatbázisokban, ezért újra írásra kerülnek a jelenlegi esetben. Ez a betöltési mód hibát okoz az idegen kulcsok miatt, amelyek a ténytáblában szerepelnek. Ez a hiba azért keletkezik mert pyspark adatbázisba írása felülírás módban truncate utasítással történik a háttérben. Az idegen kulcs megszorítások miatt viszont ez nem lehetséges, csak a tényleges törlés, delete paranccsal lehet. Ebben az esetben minden betöltés előtt az adatbázis tábláiból törölni kell a rekordokat. Ezt a törlést azért lehet megtenni mert a megszorításhoz hozzá van adva az a kiegészítés, hogy ha a ’szülő’ táblából törlünk egy elemet akkor a ’gyerek’ táblából is törlődik az adott elem.

Így a load végrehajtási folyamatában az adatokat törölni kell a végleges adatbázisból és beírni a friss rekordokat. Ezt a python nyelvben meglehet oldani egy library segítségével, amellyel csatlakozni lehet az adatbázishoz és SQL parancsokat küldeni string formájában. A folyamat során arra kell figyelni, hogy a parancsok lefuttatása után egyből meg kell szüntetni a kapcsolatot.

8.3 SCD

A historizálás(Slowly Changing Dimension) implementálása egy teljesen külön osztályba történt. A program futása során jön létre az objektumpéldány, amikor egy adott táblára meghívásra kerül a historizálást végző program rész. Ez a módszer azért előnyös megoldás mert nem kell foglalkozni külön a fájlokkal ezen programrészen belül, csak arra figyelni, hogy helyes paraméterekkel legyen meghívva a végrehajtást vezérlő script.

A dataframe-ek kezelésekor figyelmet kell fordítani arra, hogy az adott tábla rendelkezik-e helyettesítő kulccsal, ha igen akkor ezt ki kell törölni, mivel ez az adat csak az adatbázisban tárolás miatti indexelést szolgálja ezért újra lehet generálni. Fontos, hogy az eredeti adat azonosítói többször is szerepelhetnek a táblában mivel historizálva vannak ezért az helyettesítő kulcsot minden alkalommal újra kell generálni.

A tervezés során megemlítettem, hogy a feldolgozott rekordokat union művelettel lehet összekapcsolni, viszont ezen belül az esetek kezelése, a pyspark join műveletével történik, ez teljesen megegyezik az SQL-ből ismert joinokkal. Három esetet különböztet meg a program, az új rekordokat, a törlésre kerül rekordokat és a változatlan rekordokat. Ezeket az első két esetben leftanti joinnal lehet megvalósítani.

```
new = newdf.alias("left").join(currentdf.filter(col("Valid_to").isNull()).alias("right"),
    col("left.rowhash") == col("right.rowhash"),"leftanti").select("left.*") \
    .withColumn("CreatedTS", to_timestamp(lit(now),"yyyy-MM-dd HH:mm:ss")) \
    .withColumn("Valid_to", to_timestamp(lit(None),"yyyy-MM-dd HH:mm:ss")) \
    .withColumn("isValid", lit(1))
```

26. ábra Az új rekordok megkeresése

Az 26. ábrán az új eseteket tartalmazó dataframe létrehozása látható, az összekapcsolás a hash értékek alapján történik, amellyel biztosítani lehet, hogy csak megváltozott rekordok kerüljenek az eredménybe. Az összekapcsolás módja left anti, ez azért szükséges, hogy csak a baloldali táblából adja vissza a rekordokat, amelyek eleget tesznek a feltételeknek. A withColumn() utasítással pedig új oszlopokat ad a dataframe-hez és értékekkel tölti fel azt.

```
deleted = currentdf.alias("left").join(newdf.alias("right"),
    col("left.rowhash") == col("right.rowhash"),"leftanti").select("left.*")\
    .withColumn("Valid_to",when(col("Valid_to").isNull(), \
    to_timestamp(lit(now),"yyyy-MM-dd HH:mm:ss")).otherwise(col("Valid_to")))\
    .withColumn("isValid", lit(0))
```

27. ábra A törölt rekordok megkeresése

A 27. ábrán a törlésre került esetek megkeresése történik, itt a join esetében a bal és jobb oldalt felcseréltem, de ezt tetszőlegesen lehet változtatni. Itt kimondottan azokra az értékekre irányul a keresés, ahol a Valid_to értéke null, az azt jelenti, hogy érvényes adat. Ezek az egyezés esetén átállításra kerülnek a jelenlegi időpontra, valamint az is_Valid értéke 0 lesz. Ebben a join műveletben is left anti módszert kell használni ugyanis itt csak a korábbi értékek kellenek.

```
unchanged = currentdf.alias("left").join(newdf.alias("right"),
    col("left.rowhash") == col("right.rowhash"),"inner").select("left.*") \
    .withColumn("isValid",col("isValid").cast("int"))
```

28. ábra A változatlan rekordok megkeresése

Az 28. ábrán a változatlan adatok megkeresésének módja látható. A két tábla metszettéből inner join módszerével választjuk ki az egyező hash értékeket. Amennyiben a hash értékek egyeznek mindkét dataframe-ben abban az esetben arra enged következtetni, hogy az adott érték nem változott.

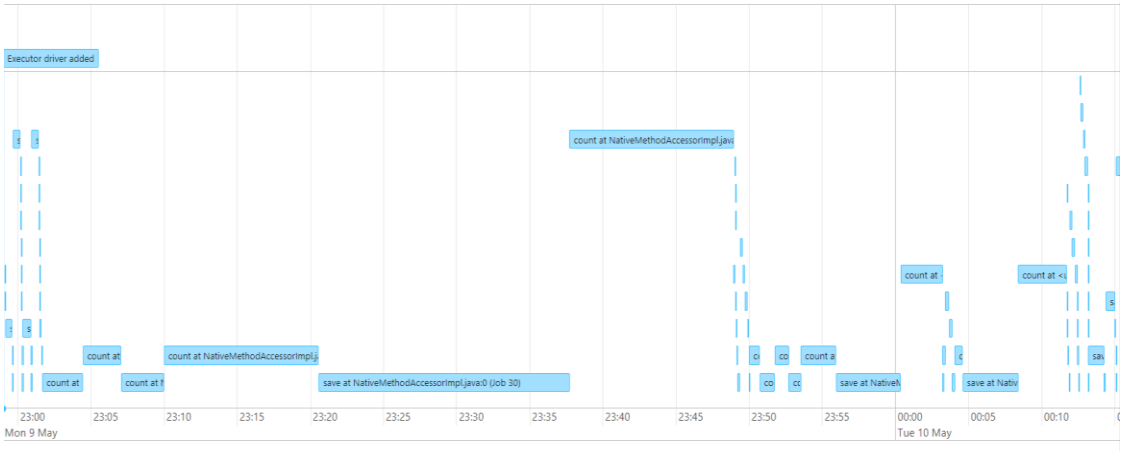
A program működése során a legnagyobb erőforrás igénye az SCD alkalmazásának van, én a tapasztalataim miatt használtam ezt a megoldást, valamint mert az adattárházak esetében ez az egyik népszerű megoldás a nyomon követhetőség implementálásra. A konkrét megoldási lehetőségeknek számtalan formája van, például léteznek olyan esetek, ahol csak a historizáláshoz egy külön adatbázist hoznak létre, viszont ennek kezelése is teljesítmény igényes.

8.4 Dátum dimenzió

A dátum dimenzió kulcsfontosságú a dimenzionális adatmodellekben. Ennek segítségével szemléltetni tudjuk az adatokat többfajta idő hierarchiában. A dátum dimenzió ellenőrzését és betöltését a Load folyamaton belül metódus végzi.

9. TESZTELÉS ÉS EREDMÉNYEK

A fejlesztés során az ETL folyamatokat végző pyspark szkripteket teszteltem futtatások során. A fejlesztés során kevesebb adattal teszteltem a programot. A tranzakciós adatok forrás fájljai 10000 sorral rendelkeztek. Ezt azért kellet a fejlesztés során redukálnom mert a pyspark igen erőforrás igényes, ha saját gépen kerül futtatásra. Ezekből a fájl csomagokból többet generáltam, így próbáltam ki magának az ETL folyamatnak a működését. A 29. ábrán látható a spark beépített web felülete, amin keresztül nyomon követhetőek azok a spark job-nak nevezett egységek, amiket kioszt a vezérlő, hogy végezze el a végrehajtást. A konkrét folyamat itt a 250 000 soros fájlok teljes integrálása az adattárházba.



29. ábra A Spark webes felületén található vizualizáció

Eszköz neve:	Acer Aspire A715-71G
Processzor:	Intel Core i5-7300HQ
Memória:	2x4GB DDR4 2400 MHz
Adatbázis háttértára:	WDC WD10SPZX - 1TB HDD

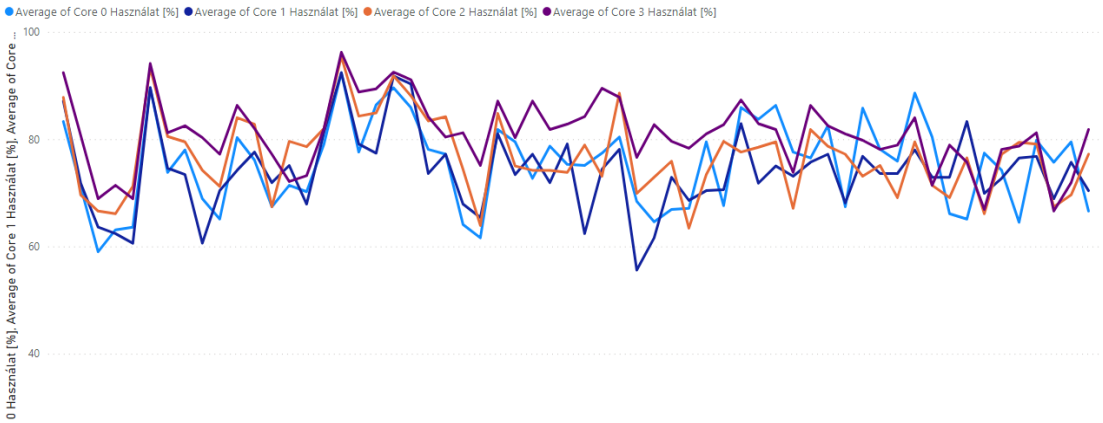
30. ábra A számítógép hardverkomponensei

Az adatok feldolgozásának ideje lineárisan arányos volt a betölteni kívánt rekordok számával. A hardver adatai, amin a feldolgozás és a fejlesztés történt a 30. táblázatban látható. A feldolgozás esetén azok a táblák vették igénybe a legtöbb időt, amelyek a legtöbb oszloppal rendelkeznek. Jelen esetben a Customer tábla adatainak a feldolgozása. A 10 000 sort tartalmazó fájlok tesztelésén kívül végeztem futtatást 100 000 valamint 250 000 sort tartalmazó fájlokkal. A generált adatok között előfordult ugyanolyan rekord is amit elsődleges kulcsnak lett használva a későbbiekben, ezek törölve lettek ilyenkor a duplikációk elkerülése érdekben.

A 31. táblázatban található futási idők másodpercben értendőek, amelyeket a programba található számlálóval mértem, ennek a számlálónak az adata mutatja meg a felhasználónak, hogy mennyi idő alatt futott le a feldolgozás.

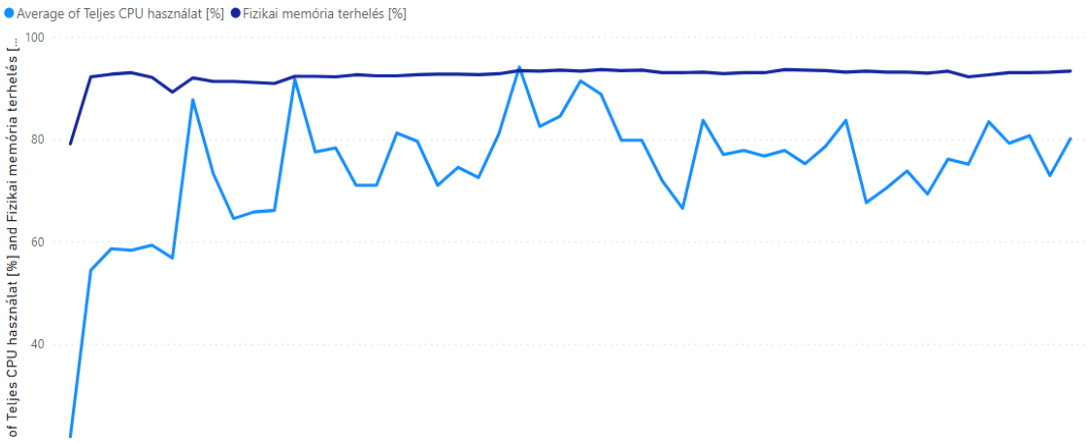
10 000 sor	100 000 sor	250 000 sor
809 s	2071 s	4441 s
906 s	2152 s	4871 s
823 s		
817 s		

31. ábra A futási idők a tényadatok sorszáma szerint rendezve.



32. ábra A processzor magok átlagolt felhasználása az ETL folyamat során.

A 32. ábrán látható az ETL folyamat alatt a processzorok átlagolt kihasználtsága. Ebből látható, hogy a processzort annyira nem terheli a feldolgozás, viszont érdemes figyelembe venni azt, hogy más program is használta a processzort amikor ezeket az adatokat mértem. Amin látható, hogy a spark memóriában dolgozza fel az adatokat, az a 33. ábra, amelyen látható, hogy a program indulásakor megtörténik a beolvasás és feldolgozás kezdetén már több mint 90%-át a fizikai memóriának felhasználja. Az adatok mérését a HWiNFO64 nevű alkalmazással végeztem.



33. ábra Fizikai memória és teljes processzor terhelés a program futása folyamán

```
+--------------+---------------+------------+----------+---------------+-----------+
|Payment_number|Customer_number|Payment_date|Support_id|Payment_type_id|Discount_id|
+--------------+---------------+------------+----------+---------------+-----------+
|       6331197|       84838795|    20200428|        47|              2|          4|
|       9495994|       62878799|    20200111|         5|              1|          7|
|      10732991|       11904957|    20201030|        40|              1|         12|
|      16488889|       14531382|    20200522|        10|              5|          6|
|      16930585|       31969190|    20200904|        25|              2|          8|
|      18696981|       53071865|    20210407|         2|              6|         13|
|      22872737|       52346840|    20211229|        23|              5|         15|
|      23856763|       15041224|    20200612|        69|              2|          3|
|      26368737|       21123681|    20211007|        12|              5|          4|
|      31513590|       13682405|    20201230|        23|              2|          1|
+--------------+---------------+------------+----------+---------------+-----------+
only showing top 10 rows

+--------------+---------------+------------+----------+---------------+-----------+--------------+
|Payment_number|Customer_number|Payment_date|Support_id|Payment_type_id|Discount_id|Payment_result|
+--------------+---------------+------------+----------+---------------+-----------+--------------+
|       6331197|       84838795|    20200428|        47|              2|          4|             3|
|       9495994|       62878799|    20200111|         5|              1|          7|             5|
|      10732991|       11904957|    20201030|        40|              1|         12|             5|
|      16488889|       14531382|    20200522|        10|              5|          6|             4|
|      16930585|       31969190|    20200904|        25|              2|          8|             2|
|      18696981|       53071865|    20210407|         2|              6|         13|             2|
|      22872737|       52346840|    20211229|        23|              5|         15|             3|
|      23856763|       15041224|    20200612|        69|              2|          3|             1|
|      26368737|       21123681|    20211007|        12|              5|          4|             2|
|      31513590|       13682405|    20201230|        23|              2|          1|             2|
+--------------+---------------+------------+----------+---------------+-----------+--------------+
only showing top 10 rows

+--------------+---------------+------------+----------+---------------+-----------+--------------+-------------------+
|Payment_number|Customer_number|Payment_date|Support_id|Payment_type_id|Discount_id|Payment_result|Subscription_number|
+--------------+---------------+------------+----------+---------------+-----------+--------------+-------------------+
|       6331197|       84838795|    20200428|        47|              2|          4|             3|                  2|
|       9495994|       62878799|    20200111|         5|              1|          7|             5|                  2|
|      10732991|       11904957|    20201030|        40|              1|         12|             5|                  2|
|      16488889|       14531382|    20200522|        10|              5|          6|             4|                  3|
|      16930585|       31969190|    20200904|        25|              2|          8|             2|                  2|
|      18696981|       53071865|    20210407|         2|              6|         13|             2|                  3|
|      22872737|       52346840|    20211229|        23|              5|         15|             3|                  2|
|      23856763|       15041224|    20200612|        69|              2|          3|             1|                  1|
|      26368737|       21123681|    20211007|        12|              5|          4|             2|                  2|
|      31513590|       13682405|    20201230|        23|              2|          1|             2|                  1|
+--------------+---------------+------------+----------+---------------+-----------+--------------+-------------------+
```

34. ábra A ténytábla az átalakítások folyamán

A 34. ábra mutatja meg a futtatás során a ténytábla adatainak változását az oszlopok hozzáfűzése után. A tesztek futtatása során kiírattam az adatokat, ellenőrzés céljából, viszont ez a spark futása szempontjából, nagy erőforrást igényel, ezért ezeket a dataframe-ket csak teszt célból érdemes kiírni.

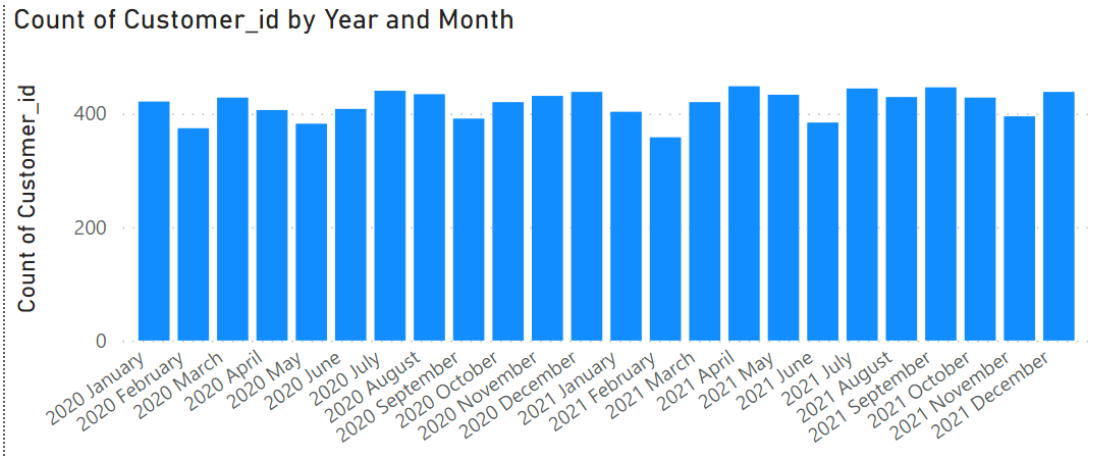
```
+-------------------+-----------------+---------------------+------------------+--------------+
|Subscription_number|Subscription_type|Subscription_interval|Subscription_price|Payment_number|
+-------------------+-----------------+---------------------+------------------+--------------+
|                  2|         6 month|                    6|                25|      66606221|
|                  1|         1 month|                    1|                 5|      30367308|
|                  1|         1 month|                    1|                 5|      27383504|
|                  3|        12 month|                   12|                50|      27219459|
|                  2|         6 month|                    6|                25|      85428420|
|                  2|         6 month|                    6|                25|      83426768|
|                  3|        12 month|                   12|                50|      33455156|
|                  3|        12 month|                   12|                50|      85206324|
|                  3|        12 month|                   12|                50|       3344981|
|                  1|         1 month|                    1|                 5|      24653407|
+-------------------+-----------------+---------------------+------------------+--------------+
only showing top 10 rows


##############################################################################################
SCD calling for dbo.Subscription table and removing first column.
The table contains 0 new rows, 0 deleted rows and 9999 unchanged rows.
+-------------------+-----------------+---------------------+------------------+-------------------+--------+-------+---------------+
|Subscription_number|Subscription_type|Subscription_interval|Subscription_price|          CreatedTS|Valid_to|isValid|Subscription_id|
+-------------------+-----------------+---------------------+------------------+-------------------+--------+-------+---------------+
|                  3|        12 month|                   12|                50|2022-05-11 20:56:07|    null|      1|              1|
|                  1|         1 month|                    1|                 5|2022-05-11 20:56:07|    null|      1|              2|
|                  2|         6 month|                    6|                25|2022-05-11 20:56:07|    null|      1|              3|
+-------------------+-----------------+---------------------+------------------+-------------------+--------+-------+---------------+
```

35. ábra A Subscription tábla adatai feldolgozás előtt és után

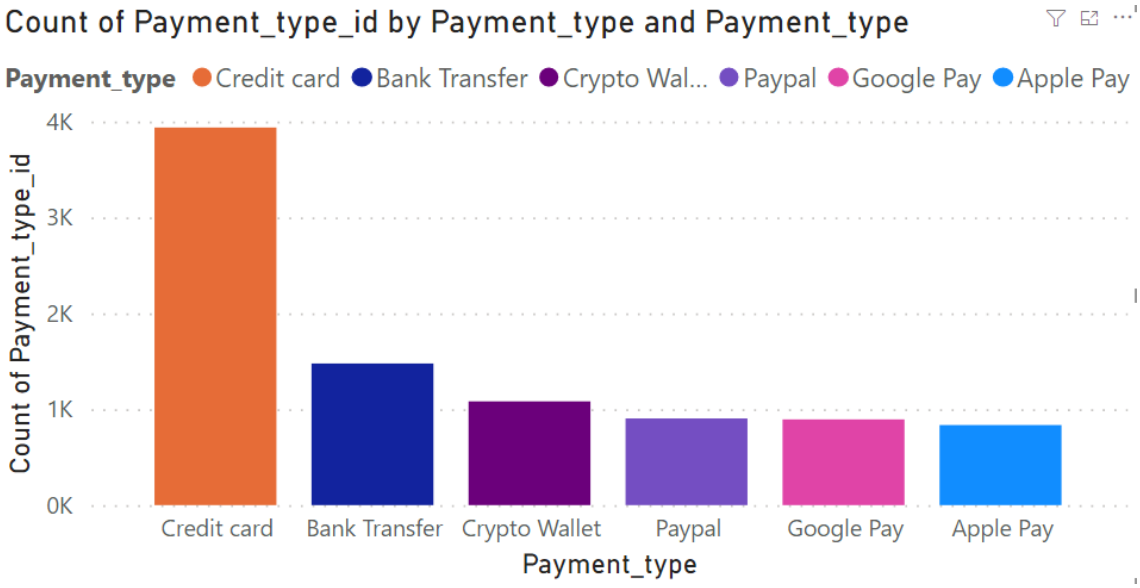
A 35. ábra mutat egy példát az historizálás alkalmazására. A feliratkozás típusai találhatóak benne. A fentebb szereplő tábla a nyers adatból kiolvasott, az alsó pedig a normalizált és feldolgozott adatokat mutatja, valamint a hozzákapcsolt CreatedTS, Valid_to és isValid oszlopokat. Valamint a generált helyettesítő kulcsot, a Subscription_id-t.

Az adattárház tesztelése érdekében, az adatbázishoz csatlakoztam a Power BI Desktop alkalmazásból és pár vizualizációt készítettem az adatokból. A 36. ábrán látható az előfizető személyek száma havi bontásban. Ez megmutatja az eloszlást az időszakok között.



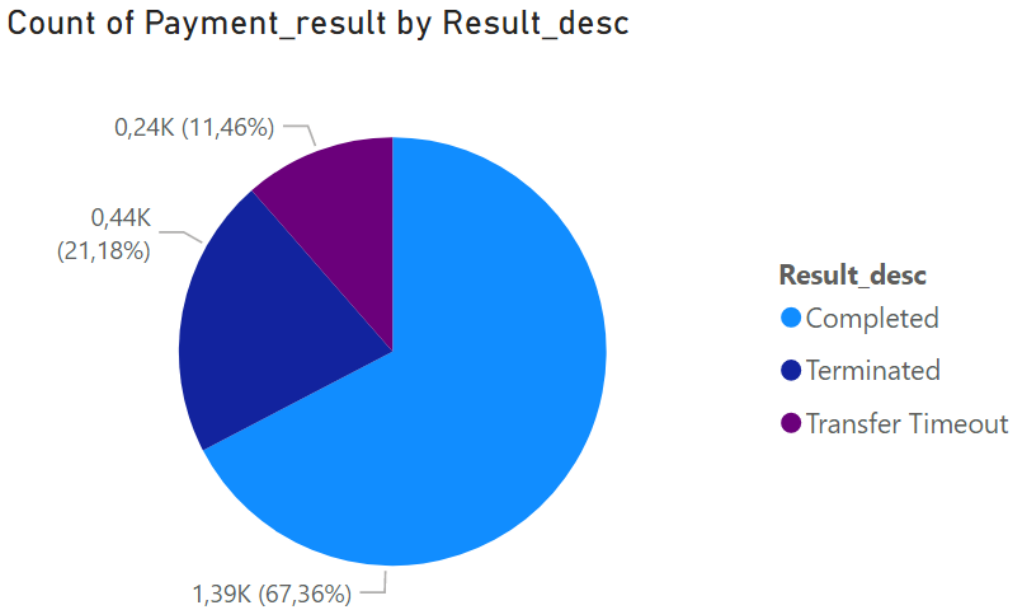
36. ábra Az előfizetések száma évenként, azon belül havonta

A 37. ábrán látható a fizetési módok arányai egymáshoz képest. Ebben az esetben a számokat maguk a fizetések megszámlálása adja. Az ábrából látható, hogy a legnépszerűbb fizetési eszköz a bankkártyás utalás.



37. ábra A fizetési módok eloszlása az előfizetések között

A 38. ábrán a fizetési eredményekről szelet diagram, ez azt mutatja meg, hogy teljesült-e a fizetés, vagy meg lett szakítva, vagy nem valósult időtúllépés miatt.



38. ábra A fizetési rendszer eredményei, amik a tényadatokhoz lettek rendelve

10 TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A fejlesztés során szembesültem pár konkrét problémával és ezek figyelembevételével, kiküszöbölésével, valamint továbbfejlesztéssel lehet javítani.

- A megvalósítás során a fő cél az volt, hogy a megvizsgáljam az Apache Spark mennyire alkalmas egy ETL folyamat létrehozásához. Jelen esetben az adatok simán szöveges fájlként érkeznek, viszont a Spark támogat többféle adatbetöltési lehetőséget (pl. Kafka, SQL adatbázisok, WebSocket). Ahhoz, hogy több eset legyen kezelhető érdemes lehet ezeket a lehetőségeket is implementálni. Például Kafka adatforráshoz a pyspark-ba beépített beolvasás is található.
- Az adat validációt korlátozott mértékben lehet alkalmazni, érdemes megfontolni gépi tanulás alapú módszer alkalmazását, ami sokkal pontosabb eredményeket hozhat.
- A program konzolos alkalmazásként működik amennyiben érdemes lehet használni szélesebb körben akkor egy Web interfész fejlesztése a programhoz, bár a gyakorlatban az adatfeldolgozást végző spark-on alapuló programokhoz nem szoktak GUI-t készíteni, de megfontolandó lehet.
- A Spark működése saját gépen igen nagy erőforrás igénnyel rendelkezik. Ezért szokták a gyakorlatban a felhőbe integrálni sok esetben. Erre a legnépszerűbb megoldás a DataBricks. Azért is érdemes megfontolni mert saját tárolási módszerrel rendelkezik, valamint SQL, Python, Scala nyelven is használható. Valamint az Azure, AWS és Google Cloud rendszerében érhető el, más funkcionalitásokkal együtt.
- Mivel a historizálás igényli az egyik legtöbb erőforrást a feldolgozás során érdemes lehet megfontolni, hogy más módszer alapján kezeljük a régi rekordokat. Például külön táblák létrehozása kimondottan erre a célra.

ÖSSZEFOGLALÁS

A szakdolgozatomban meg akartam vizsgálni milyen lehetőség van egy olyan adatfeldolgozó alkalmazás készítésére, amely python nyelven alapszik. Az indíttatás személyes tapasztalat volt, mert adatfeldolgozó szkriptekkel dolgoztam. Az adatfeldolgozás a napjainkban igen fontos és egy kardinális kérdés lett az iparban a növekvő adatmennyiség miatt. Különböző módszerek léteznek arra, hogy valamilyen módon biztosítva legyenek az adatminőség tényezői. Egyik ezek közül az iparban is elterjedt ETL folyamat. Ez a folyamat általában az adattárházakhoz kapcsolódik, de lehet más alkalmazási célja is. Az ETL folyamatok jól definiált és előre megtervezett feldolgozó eljárások. Általában egy jó megoldás az adatok feldolgozására, attól függően, hogy milyen adatokkal dolgozik az ember. A Python nyelv hasznos a szoftveriparban, és számos könyvtárral és csomaggal rendelkezik a különböző feladatok kezelésére és problémák megoldására. Az egyik a pyspark, amely az apache spark python API-val ellátott verziója. Az apache spark egy nagy teljesítményű elemző és feldolgozó szoftver. Kíváncsi voltam, hogy használhatom-e a Spark-ot az ETL folyamatok létrehozására, és hogy mennyire hatékony ebben a környezetben. Erre a célra egy hibrid sémával rendelkező alap adattárházat használtam. Az sql szerverrel 3 adatbázist hoztam létre, amelyeket különböző rétegként használtam a feldolgozáshoz. Az első az extract folyamathoz, a második az transform folyamathoz, a harmadik pedig a load folyamathoz, ahol az adatok az adattárházba kerülnek. Az adattárházak egyik kulcsfontosságú része, hogy a múltbeli adatok hozzáférhetők, annak ellenére, hogy a forrásból a legfrissebb adatokat is tartalmazzák. Ennek biztosítására ún. historizálást kellett készítenem, amely megmutatja, hogy az adott adat aktuális-e vagy sem. Ezek a historizált rekordok azt tartalmazzák, hogy az adat mikor került be az adattárházba, és mikor került törlésre az adattárházból. Ehhez a historizáláshoz különböző pyspark join-okat használtam az új adatok, a változatlan adatok és a törölt adatok különböző feltételeinek kezelésére. A historizálás mellett minden adatkészletet más-más módon kell kezelni, hogy biztosítsuk az adatok egyértelműségét az adatbázisban, ezért eltérő transzformációkat alkalmaztunk az adatkészletekre. A fejlesztés során, tapasztalataim szerint, a spark egy alkalmas eszköz a python-nal, amellyel adatfeldolgozásra is használható, nem csak ETL-hez hanem más típusú integrációs folyamatokhoz is. Számos harmadik féltől származó alkalmazás is található a piacon, de a Spark ingyenesen használható, és különböző célokra is lehet használni.

SUMMARY

In my thesis I wanted to investigate and develop a data processing application with Python and its packages. The motivation was my personal and work experience, because I worked with data processing scripts and appplications. The data processing in today's industry is a key factor because there is more and more data. There are different methods to ensure the requirements for good quality of data for the companies. One of them is ETL processes. ETL processes usually loads up data warehouses with data, but it can be used with other type of data storage architectures. The ETL processes are rigid and require a lot of development and tuning to work well, but in the end, ETL usually provide a good solution for data processing. The Python language is useful in the software industry and it has many librarys and packages to handle different tasks and solve problems. One of them is PySpark which is the python version of the apache spark. Apache spark is a powerful analytics engine for large scale data processing. I was curious to be able to use spark to create etl processes and how effective spark is in this environment. For this purpose I used sample data for a basic data warehouse with hybrid schema. With sql server I created 3 databases which has been used as different layers for the processing. The first one is for the extract process the second one is for the transform process and the third one is for the load process where the data gets into the data warehouse. One key part of the data warehouses is the past data can be accessed despite it contains the most recent data from the sources. To ensure this, i had to create so called historization which shows, how actual the specific data is. These historized records contains when was the data inserted into the data warehouse and when got deleted from the data warehouse. For this historization, I used pyspark joins to handle different conditions for the new data, the unchanged data and deleted data. Besides of the historization, every dataset need to be handled different way to ensure data quality and performance in the database, therefore had apply different transformations for the datasets. During the development, in my experience, I found out the spark is a great tool with python to use it for data processing nut just for ETL but other types of integration processes. There are also many third party applications on the market, but spark is free to use and can be used for many things. One drawback was spark's performance during the develeopment.

IRODALOMJEGYZÉK

[1] „towardsdatascience.com,” [Online]. Available: https://towardsdatascience.com/how-important-is-data-for-your-business-c15a35c6935e. [Hozzáférés dátuma: 28. 10. 2021.].

[2] „Investopedia,” [Online]. Available: https://www.investopedia.com/terms/d/data-analytics.asp. [Hozzáférés dátuma: 24. 10. 2021.].

[3] [Online]. Available: https://www.datascience-pm.com/crisp-dm-2/. [Hozzáférés dátuma: 28 10 2021].

[4] W. H. Inmon, Building the Data Warehouse, Wiley.

[5] [Online]. Available: http://scs.web.elte.hu/Work/DW/adattarhazak.htm#3. [Hozzáférés dátuma: 23. 11. 2021.].

[6] „Geeksforgeeks,” [Online]. Available: https://media.geeksforgeeks.org/wp-content/uploads/Untitled-drawing-3-2.png. [Hozzáférés dátuma: 13. 11. 2021.].

[7] [Online]. Available: https://www.geeksforgeeks.org/snowflake-schema-in-data-warehouse-model/. [Hozzáférés dátuma: 23. 11. 2021.].

[8] „Geeksforgeeks,” [Online]. Available: https://www.geeksforgeeks.org/snowflake-schema-in-data-warehouse-model/. [Hozzáférés dátuma: 23. 11. 2021.].

[9] „informatica.com,” [Online]. Available: https://www.informatica.com/resources/articles/what-is-data-quality.html. [Hozzáférés dátuma: 23. 11. 2021.].

[10] „omnisci.com,” [Online]. Available: https://www.omnisci.com/technical-glossary/data-quality. [Hozzáférés dátuma: 23. 11. 2021.].

[11] „Google,” [Online]. Available: https://cloud.google.com/learn/what-is-etl. [Hozzáférés dátuma: 02. 11. 2021.].

[12] [Online]. Available: https://www.stitchdata.com/etldatabase/etl-process/. [Hozzáférés dátuma: 15. 11. 2021.].

[13] „Databricks,” [Online]. Available: https://databricks.com/glossary/extract-transform-load. [Hozzáférés dátuma: 14 11. 2021.].

[14] [Online]. Available: https://www.stitchdata.com/etldatabase/etl-extract/. [Hozzáférés dátuma: 5. 11. 2021.].

[15] [Online]. Available: https://www.stitchdata.com/etldatabase/etl-transform/. [Hozzáférés dátuma: 05. 11 2021.].

[16] [Online]. Available: https://www.xplenty.com/blog/etl-vs-elt/#elt. [Hozzáférés dátuma: 06. 11. 2021.].

[17] „datawarehouse4u,” [Online]. Available: https://www.datawarehouse4u.info/SCD-Slowly-Changing-Dimensions.html. [Hozzáférés dátuma: 23. 11. 2021.].

[18] [Online]. Available: https://www.bigdataframework.org/data-types-structured-vs-unstructured-data/. [Hozzáférés dátuma: 08. 11. 2021.].

[19] [Online]. Available: https://www.geeksforgeeks.org/sql-vs-nosql-which-one-is-better-to-use/. [Hozzáférés dátuma: 08. 11. 2021.].

[20] [Online]. Available: https://pandas.pydata.org/about/index.html. [Hozzáférés dátuma: 11. 11. 2021.].

[21] [Online]. Available: https://pandas.pydata.org/docs/user_guide/dsintro.html. [Hozzáférés dátuma: 11. 11. 2021.].

[22] [Online]. Available: https://spark.apache.org/docs/latest/index.html. [Hozzáférés dátuma: 15. 11. 2021.].

[23] „Tutorialspoint,” [Online]. Available: https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm. [Hozzáférés dátuma: 25. 11. 2021.].

[24] „Tutorialspoint,” [Online]. Available: https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm. [Hozzáférés dátuma: 23. 11. 2021.].

[25] [Online]. Available: https://www.geeksforgeeks.org/snowflake-schema-in-data-warehouse-model/.

[26] [Online]. Available: https://media.geeksforgeeks.org/wp-content/uploads/Untitled-drawing-3-2.png.

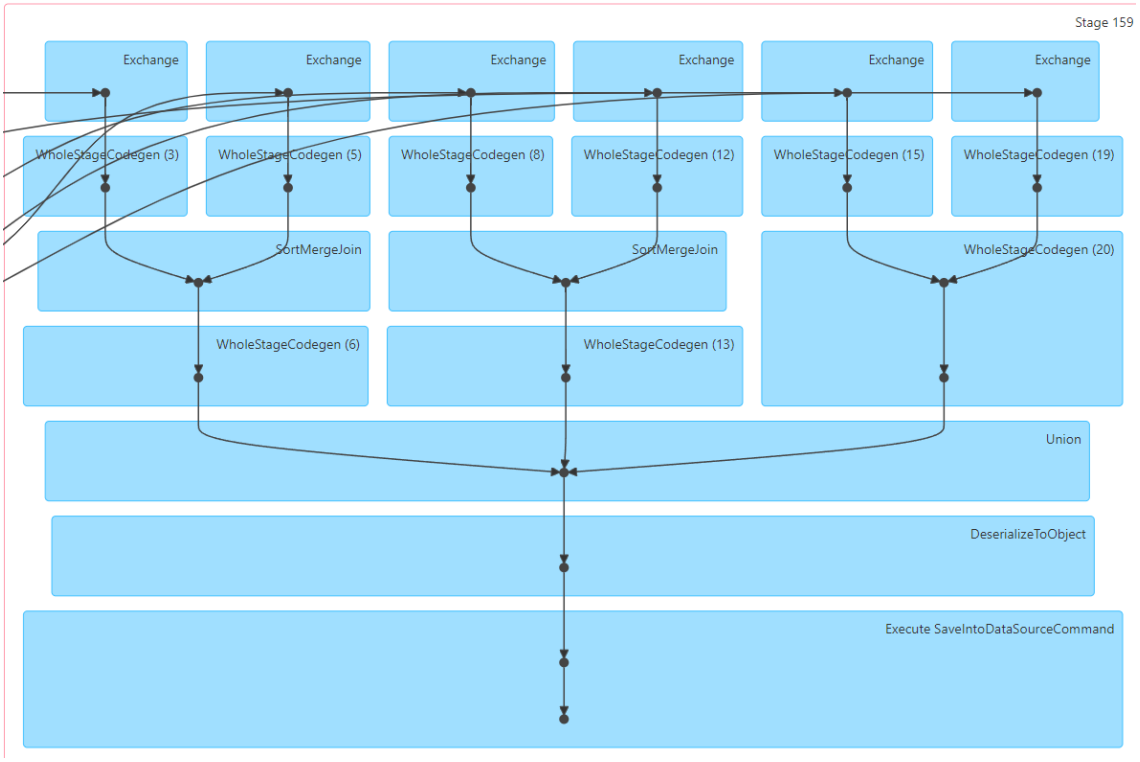
[27] „Talend,” [Online]. Available: https://www.talend.com/resources/what-is-data-lake/.

[28] [Online]. Available: https://pandas.pydata.org/docs/user_guide/dsintro.html. [Hozzáférés dátuma: 11 11 2021].

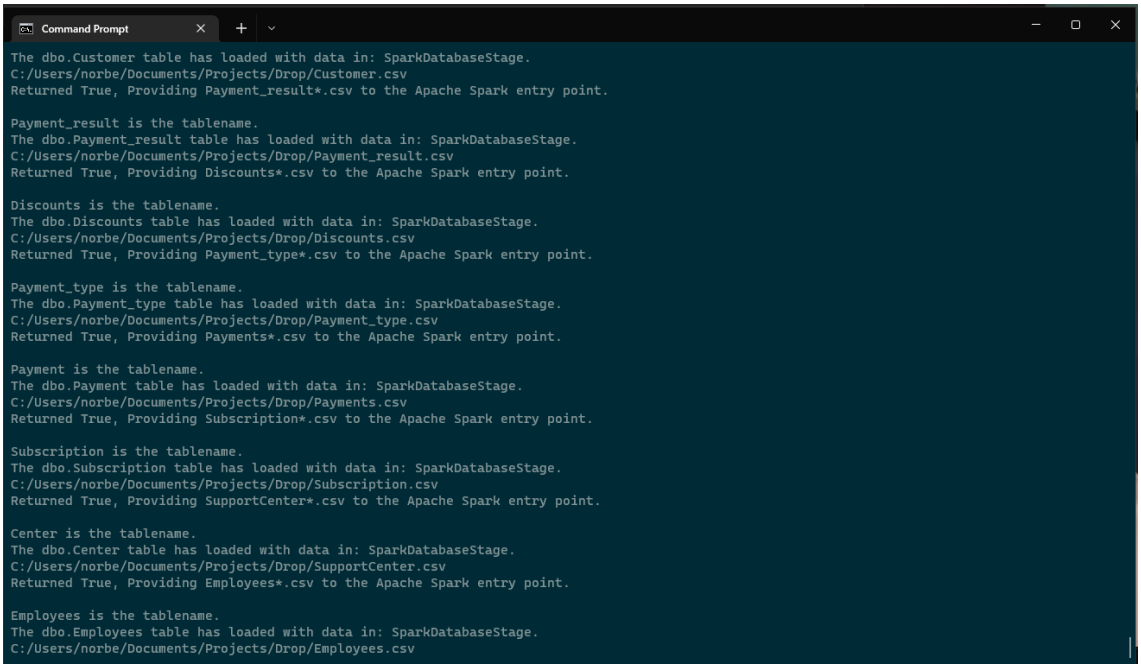
[29] [Online]. Available: https://databricks.com/glossary/extract-transform-load. [Hozzáférés dátuma: 14. 11. 2021.].

ÁBRAJEGYZÉK

MELLÉKLETEK



39. ábra A Spark végrehajtási módjának ütemezése az SCD műveletek során



40. ábra A program működése az extract folyamat során

```
##################################################################################################
SCD calling for dbo.Payment_result table and removing first column.
The table contains 0 new rows, 0 deleted rows and 5 unchanged rows.
The dbo.Payment_result table has loaded with data. The loaded data contains 5 rows
##################################################################################################

##################################################################################################
SCD calling for dbo.Subscription table and removing first column.
The table contains 0 new rows, 0 deleted rows and 9999 unchanged rows.
The dbo.Subscription table has loaded with data. The loaded data contains 3 rows
##################################################################################################
The dbo.Payment table has loaded with data. The loaded data contains 9999 rows
##################################################################################################

##################################################################################################
SCD calling for dbo.Center table.
The table contains 0 new rows, 0 deleted rows and 15 unchanged rows.
The dbo.Center table has loaded with data. The loaded data contains 15 rows
##################################################################################################
```

41. ábra A program működése a Transform folyamat során

```
def IntegrateSubscription(self,targetdf,sourcedf):

    res = targetdf.join(sourcedf,targetdf["Payment_number"] == sourcedf["Payment_number"],'inner') \
        .select(targetdf["*"],sourcedf["Subscription_number"])

    res.show(n=10)
    return res

def IntegrateCustomerid(self,targetdf,sourcedf):


    res = targetdf.join(sourcedf,targetdf["Customer_number"] == sourcedf["Customer_Number"],'inner') \
        .select(targetdf["*"],sourcedf["Customer_id"])

    res = res.drop("Customer_number")

    res.show(n=10)
    return res
```

42. ábra Subscription és Customer adatok ténytáblába integrálása

```
df1 = df.withColumn("rowhash", sha1(concat_ws("||", *df.columns)))
new = self.newRow(df1,self.dbdfhashed)
deleted = self.deletedRow(df1,self.dbdfhashed)
unchanged = self.UnchangedRow(df1,self.dbdfhashed)

count3 = new.count()
count4 = deleted.count()
count5 = unchanged.count()
new = new.drop("rowhash")
deleted = deleted.drop("rowhash")
unchanged = unchanged.drop("rowhash")
print('The table contains {0} new rows, {1} deleted rows and {2} unchanged rows.'.format(count3,count4,count5))
result = new.union(deleted).union(unchanged)
result.rdd.count
```

43. ábra A historizálást és egyesítést meghívó metódus