VIŠJA STROKOVNA ŠOLA ACADEMIA

MARIBOR

Historizacija tabel iz CDC zapisov v Azure SQL bazi

Kandidat: Tine Ogrinc

Vrsta študija: študent izrednega študija

Študijski program: Informatika

Mentor predavatelj: mag. Ervin Schaff

Mentor v podjetju: Žiga Prajs, dipl. org. inf. (UN)

Lektor: mag. Ervin Schaff

Maribor, 2024

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani Tine Ogrinc, sem avtor diplomskega dela z naslovom Historizacija tabel iz CDC zapisov v Azure SQL bazi, ki sem ga napisal pod mentorstvom mag. Ervina Schaffa.

S svojim podpisom zagotavljam, da:

- je predloženo delo izključno rezultat mojega dela,
- sem poskrbel, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia Maribor,
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli, kot moje lastne kaznivo po Zakonu o avtorski in sorodnih pravicah (Uradni list RS, št. 16/07 – uradno prečiščeno besedilo, 68/08, 110/13, 56/15 in 63/16 – ZKUASP); prekršek pa podleže tudi ukrepom Višje strokovne šole Academia Maribor skladno z njenimi pravili,
- skladno z 32.a členom ZASP dovoljujem Višji strokovni šoli Academia Maribor objavo diplomskega dela na spletnem portalu šole.

Logatec, junij 2024 Podpis študenta:

ZAHVALA

Zahvaljujem se mentorju na šoli mag. Ervinu Schaffu, mentorju v podjetju Žigi Prajsu in svojim staršem za številna leta potrpežljivosti pri mojem šolanju.

POVZETEK

V diplomskem delu je predstavljen projekt, pri katerem sem postavil podatkovni tok v oblačnem okolju Azure. Podatkovni tok se je začel na virni bazi, ki je bila Azure SQL Database. Na tej bazi sem postavil orodje Debezium, ki je zajemalo spremembe na konfiguriranih tabelah na tej virni bazi, ter te spremembe posredoval v streaming orodje, v tem primeru je bilo to Azure Event Hubs. Prihod sporočila v instanco Azure Event Hubs je prožilo instanco Azure Functions, ki omogoča izvajanje kode brez strežnika. Tam se je sprožil program, spisan v programskem jeziku Python, ki je podatke razbral in shranil v ponorno bazo Synapse Dedicated SQL Pool. Na ponorni bazi so se podatki shranili v več različnih načinih hrambe historičnih podatkov, potem so se sestavile SQL-poizvedbe, s katerimi se je evalvirala performanca teh načinov.

Povezano s tem projektom sem raziskal še štiri hipoteze. Prva hipoteza, H1, je bila, da so orodja za streaming podatkov v različnih oblačnih storitvah zelo podobna. Ta hipoteza je bila potrjena, saj sem ugotovil, da imajo orodja za streaming pri treh največjih oblačnih ponudnikih zelo podobne funkcije in obnašanje.

Zatem sem preiskoval podatkovne baze v oblačnem okolju Azure, pri čemer je bila hipoteza H2, da je za podatkovno skladiščenje najustreznejša Synapse Serverless SQL Pool. Ta hipoteza je bila ovržena, saj sem ugotovil, da Synapse Dedicated SQL Pool vsebuje vse funkcionalnosti Serverless SQL Poola, vendar poleg tega omogoča tudi uporabo standardne podatkovne baze, kar je bilo v našem primeru obvezno, in da je tudi v drugih primerih bolje imeti standardno podatkovno bazo, če ne drugega, za administrativne tabele.

Kot omenjeno v opisu projekta sem preiskoval tudi oblike hrambe historičnih podatkov, pri čemer je bila hipoteza H3, da je SCD 6 najustreznejši. Ta hipoteza je bila ovržena, saj sem ugotovil, da je SCD 2 ekvivalenten pri vseh testnih primerih, ki sem si jih zamislil, a hkrati dosti boljši pri vstavljanju podatkov.

Zadnja hipoteza, H4, je bila, da bodo relacijske podatkovne baze obdržale svojo dominanco in da novi tipi baz, kot so vektorske in NoSQL-baze, ne bodo nadomestile relacijskih baz. To hipotezo sem potrdil.

Ključne besede: historizacija, podatki, skladišče, oblak, CDC, baze

ABSTRACT

Historising tables from CDC records in Azure SQL Database

This thesis presents a project where a cloud dataflow is setup in Azure Cloud. The data flow begins in a source database, in our case, Azure SQL Database. Debezium was setup on this database, which is a tool that captures changes on configured tables and forwards those changes to a data streaming service, in our case that was Azure Event Hubs. The arrival of the message on the Event Hubs instance triggered an Azure Functions program, which allows for serverless execution of a program. The program in question was code written in Python, which parsed the message and entered the data in question into a final sink database, in our case that was Synapse Dedicated SQL Pool. The data was saved in multiple ways, so as to compare different methods of historical data storage, and those tables were also queried to evaluate the performance of those types of historical data storage.

As part of this thesis, 4 hypothesis were evaluated. The first, H1, was that streaming tools are functionally equivalent between the main cloud providers. This hypothesis was confirmed, as I found that streaming services amongst the three main cloud providers had very similar functionalities and behaviour.

The next hypothesis, H2, dealt with SQL databases in Azure. Here I believed that Synapse Serverless SQL Pool was the database most suitable to data warehousing. This hypothesis was mistaken, as Serverless SQL Pool does not allow for any traditional database tables, which were needed in our project. So Synapse Dedicated SQL Pool was chosen as the most suitable database for our purposes.

Hypothesis H3 dealt with storage of historical data. Here I believed SCD 6 type of historical data was most suitable. But this was proven to be wrong in our examples, and SCD 2 provides the same performance in querying, but is significantly better for inserting new data.

The last hypothesis, H4, was that relational databases will maintain their dominance amongst databases, and that NoSQL and Vector databases will not replace them. This hypothesis was confirmed.

Keywords: historisation, data, warehouse, cloud, CDC, database

KAZALO VSEBINE

KAZALO SLIK

KAZALO TABEL

1 UVOD

1.1 Opis področja in opredelitev problema

Za temo diplomskega dela sem izbral podatkovni inženiring, specifično pot iz streaming podatkov v podatkovno skladišče v oblaku. V oblačnem okolju bom sestavil celo pot od virne baze do ponorne baze.

To temo sem izbral, ker sem se z njo srečal v okviru zaposlitve in se mi je zdela zanimiva. Vse več podjetij migrira svojo infrastrukturo v oblak in prehaja iz velikih performantnih baz v majhne baze, ki so dostopne samo svojemu microservisu in torej podatkovno skladišče ne more neposredno dostopati do njih. Rešitev za ta problem je program, kot je Debezium, ki brez obremenjevanja baze kopira evidenco sprememb (angl. Change Data Capture – CDC) in jo posreduje v storitev za streaming. Nato od tam podatke poberemo in vstavimo v ponorno bazo, ki je namenjena analizi.

V okviru tega diplomskega dela bom ugotavljal, kakšna je prihodnost podatkovnega skladiščenja, kako se različne oblačne storitve primerjajo med seboj in kateri je najučinkovitejši način shranjevanja spremenljivih podatkov.

1.2 Namen, cilji in osnovne trditve

Namen diplomskega dela je ugotoviti, kakšna je prihodnost podatkovnega skladiščenja, kako se različne oblačne storitve primerjajo med seboj in kateri je najučinkovitejši način shranjevanja spremenljivih podatkov.

Postavil sem si štiri glavna vprašanja:

Kako se dela data streaming v oblaku?

Pri tem je moja hipoteza, H1, da so si glavni trije oblačni ponudniki enakovredni.

Kaj je ekvivalent SQL-baze na oblaku Azure?

Pri tem je moja hipoteza, H2, da je za podatkovno skladiščenje najustreznejši Synapse serverless SQL pool.

Kako strukturirati historizirano tabelo?

Pri tem je moja hipoteza, H3, da je za namene, kjer je treba večinoma brati trenutno stanje in občasno pogledati tudi kakšen historični podatek, najustreznejši SCD6.

Ali bodo NoSQL ali vektorske baze nadomestile klasične relacijske podatkovne baze?

Pri tem je moja hipoteza, H4, da bodo še vedno najpomembnejše relacijske podatkovne baze.

Moj cilj je odgovoriti na ta štiri vprašanja in potrditi ali ovreči zastavljene hipoteze.

1.3 Predpostavke in omejitve

Za varnost poslovnih podatkov se delo na projektnem delu zanaša na generirane testne podatke.

Omejitev je tudi finančna, saj se praktični del diplomskega dela opravlja na oblačnem okolju Azure, ki ni brezplačen. Ob vpisu sicer dobimo neki znesek brezplačnih kreditov za eksperimentiranje, vendar nekatere storitve porabijo veliko kreditov. Tako smo se omejili na nekaj tisoč testnih podatkov namesto na milijone, kar bi bilo potrebno za pridobitev res temeljitih rezultatov.

1.4 Uporabljene raziskovalne metode

Diplomsko delo je sestavljeno iz teoretičnega in praktičnega dela.

V teoretičnem delu sem uporabil deskriptivno oziroma opisno metodo zbiranja podatkov iz različnih virov, ki so tudi navedeni na koncu v poglavju Viri. Viri so bili predvsem razna dokumentacija različnih programov, skupaj z nekaj strokovne literature in člankov.

To sem komplementiral z metodo dedukcije, pri čemer sem na podlagi pridobljenih informacij prišel do določenih spoznanj.

Pri praktičnem delu sem na oblačni storitvi izdelal pot za podatke, kar je pomenilo postavitev in konfiguracijo številnih različnih storitev in pisanje programske kode v več jezikih.

2 BIG DATA STREAMING V OBLAKU

2.1 Big Data

Big Data ali po slovensko velepodatki je pojem, ki se nanaša na hrambo in obdelavo velike količine podatkov.

Enotna jasna opredelitev sicer ne obstaja. Izvorno je šlo za količino podatkov, ki je ne zmorejo obdelovati trenutni glavni sistemi. Vendar se z izboljšavami sistemov ta opredelitev premika, in kar so bili pred desetimi leti velepodatki, je dandanes majhna tabela v podatkovnem skladišču. Kar je skupno večini opredelitev, je, da gre za veliko količino podatkov, manj strukturiranih kot v sistemih za podatkovno analitiko, podatki so tudi manj zanesljivi, pogosto se pri obdelavi uporablja vzporedno računanje (angl. Parallel Computing) in se podatki povečini uporabljajo za prediktivno analitiko in ne za analizo stanja tako kot pri poslovni analitiki.

Večina modernih implementacij Big Data je osnovanih okoli tako imenovanih Data Lakeov, ki so bolj kot ne samo velik disk za podatke, brez jasno opredeljene strukture. Podatki tudi pogosto pridejo s pomočjo streaming virov, kot je Apache Kafka, saj je lažje vse zapise poslati na enotno točko, iz katere lahko potem različni sistemi vlečejo zapise, ki jih potrebujejo. (Big data, 2024)

Uporaba Big Data je izjemno obširna, povsod, kjer obstajajo meritve, se najde nekdo, ki želi iz teh podatkov izvleči vrednost. Ni nenavadno tudi, da se meritve hranijo, tudi če se ne ve, ali obstaja v njih vrednost, za primer, da se bo v prihodnje odkrila kakšna vrednost v njih. Številna moderna podjetja še celo svoj obstoj delno ali v celoti financirajo skozi prodajo podatkov. Brezplačne storitve na spletu niso zares brezplačne, zanje plačujemo skozi oglase ali skozi prodajo naših podatkov. Ta prodaja je lahko specifična, pri čemer podjetja, kot sta Google ali Meta, na podlagi naših podatkov generirajo marketing profil za nas, pri čemer ugotovijo, kaj nam naj oglašujejo. Lahko so tudi nespecifična, pri čemer lahko podjetje prodaja recimo statistiko, koliko oseb išče določene stvari s pomočjo njihove platforme.

Pri tem se že srečamo s težavami z zasebnostjo, namreč Google recimo ve dosti več o nas, kot si mislimo, ravno zaradi vseh teh podatkov, ki jih zbirajo. Vsakdo se je že znašel v položaju, ko se je z nekom pogovarjal o nekem izdelku in kmalu za tem prejel oglas za ta izdelek. Pogosto se zato sklepa, da nam telefon prisluškuje, vendar se mi resnica zdi bolj strašna, prediktivni modeli so tako močni, da pogosto na podlagi tega, kar že vedo o nas, uganejo, kaj nas bo zanimalo in o čem bi se pogovarjali s prijatelji. In s čedalje več podatki, z bolj performantnimi računalniki in boljšimi modeli bodo ti prediktivni modeli čedalje boljši. To izpostavlja

pomembnost varovanja osebnih podatkov. Na srečo je Evropska unija nekaj že naredila glede tega z GDPR-regulativami (angl. General Data Protection Regulation, zakoni, ki veljajo po celotni Evropski uniji ter regulirajo hrambo in uporabo osebnih podatkov), vendar bo potrebno še več.

Big Data je prihodnost in v neki meri tudi že sedanjost. Podatki so novo zlato in smo sredi zlate mrzlice. Obstajajo dobre strani, kot so recimo vsi napredki v zdravstvu, in slabe strani, kot je recimo predobro oglaševanje. S tehnološke strani predstavlja zanimive izzive s čedalje večjimi podatkovnimi zbirkami.

2.2 Streaming

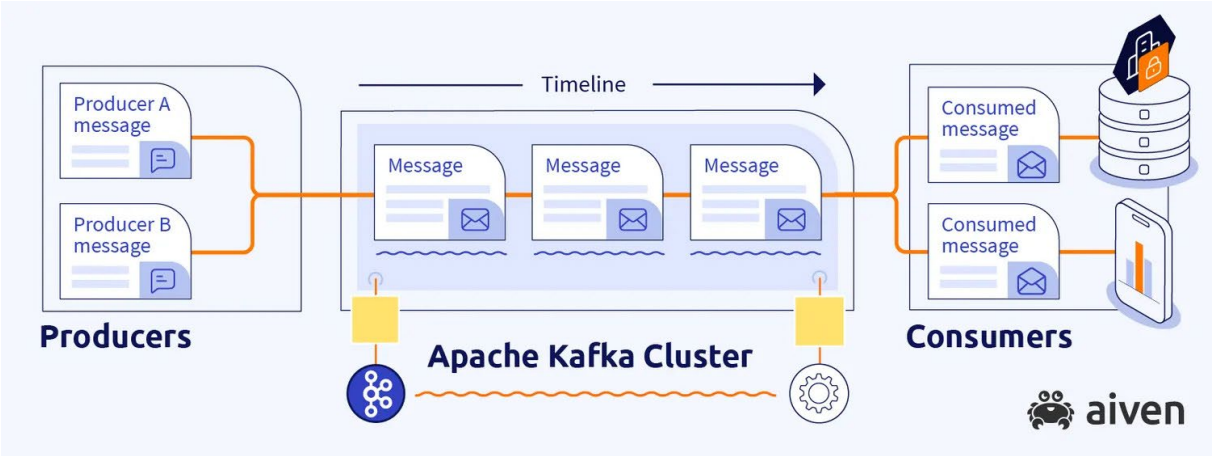
Podatki streaming postajajo čedalje pomembnejši v modernih podatkovnih sistemih.

Stari standardi so bili, da se podatki procesirajo občasno, recimo enkrat dnevno v enem velikem procesu. Vendar to pomeni, da so podatki vedno stari, in strežniki so izjemno obremenjeni enkrat na dan, medtem ko so večino časa popolnoma neobremenjeni. To je neučinkovito in neustrezno za številne moderne podatkovne analize, pri čemer je želja imeti podatke v real-time oziroma čim prej po tem, ko se podatek generira.

Rešitev za to je streaming.

Glavno ime podatkov streaming je Apache Kafka. Apache Foundation je nepridobitna organizacija, ki podpira številne izjemno priljubljene odprtokodne programske rešitve. Apache Foundation se je začel z enim programom, strežnikom Apache HTTP, to je program, ki se še po 29 letih uporablja pri skoraj tretjini spletnih strani.

Apache Kafka je program, ki je bil razvit leta 2011 na LinkedInu, leto kasneje ga je pa prevzel Apache Foundation. Program omogoča distribuirano in performantno hrambo velike količine manjših sporočil. Glavna struktura je Kafka Cluster, ki vključuje enega ali več strežnikov, ki dejansko hranijo in posredujejo podatke, ti se imenujejo brokerji. Znotraj clustra se lahko postavi več topicov oziroma tem, ki delujejo kot kakšni poštni nabiralniki. Pri tem je lahko poljubno veliko producerjev podatkov, ki polnijo te nabiralnike, in potem poljubno število consumerjev teh podatkov. Pri tem Kafka skrbi, da vsak consumer dobi svojo kopijo teh podatkov, da vedno dobi samo sporočila, ki jih še ni prebral. (Apache Foundation, 2024)



Slika 1: Prikaz strukture podatkovnega toka Kafka

Vir: (https://cdn.sanity.io/images/sczeoy4w/production/c0db44fb8695a8b3d3d0c32abcdeda887d121a57-1600x602.png?w=1000&q=80&fit=min&auto=format&dpr=1.5)

2.3 Primerjava orodij streaming v različnih oblačnih okoljih

Kafka je najbolj znano ime v event streaming, vendar ni edino. Glavni trije oblačni ponudniki so: Amazon Web Services, Microsoft Azure in Google Cloud (Statista, 2024), ki so vsi vpeljali tudi svoje rešitve. In moje vprašanje na to temo je bilo: Katero od teh storitev naj izberem?

Storitve, o katerih govorimo, so:

Amazon Web Services:

- Amazon MSK (Managed streaming for Apache Kafka),
- Amazon Kinesis;

Microsoft Azure:

- Azure Event Hubs;

Google Cloud:

- Google Pub/Sub.

Amazon se je odločil kar za dve rešitvi, Kinesis je njihova lastna rešitev, medtem ko je MSK samo Apache Kafka, integrirana v njihov oblak, in omogoča poenostavljeno upravljanje.

Te storitve sem primerjal na podlagi nekaj dejavnikov:

- Ali ima storitev topic ali neki ekvivalent?
- Ali storitev omogoča push, torej da se consumerju sporoči, da je sporočilo, in da ni treba njemu vprašati, ali je kakšno sporočilo.
- Ali lahko producer polni več topicov in več producerjev polni isti topic?
- Ali lahko en consumer spremlja več topicov in več consumerjev isti topic?
- Kakšne so omejitve sporočil?
- Ali obstajata particioniranje in distribucija med strežniki?
- Kako dolgo se hranijo neprebrana sporočila?
- Povezljivost z drugimi storitvami?

Tabela 1: Primerjava rešitev streaming v različnih oblakih

	Amazon MSK	Amazon Kinesis	Google Pub/Sub	Azure Event Hubs
Obstaja topic?	TOPIC	STREAM	TOPIC	TOPIC
Omogoča PUSH?	NE	DA	DA	DA
Producer več topicov in več producerjev?	DA	DA	DA	DA
Consumer več topicov in več consumerjev?	DA	DA	DA	DA
Omejitve sporočil?	1 MB	1 MB	10 MB	1 MB
Particioniranje in replikacija?	DA	DA	DA	DA

Hramba?	Default neomejeno, konfigurabilno	Default 1 dan, maks 1 leto	Default 7 dni, maks. 31 dni	Default 7 dni, maks. 90. Ponujajo storitev, kjer se samodejno v Cloud storage piše za neomejeno shranjevanje.

(Amazon Web services, 2024) (Amazon Web Services, 2024) (Azure, 2024) (Google Cloud, 2024)

Kot lahko vidimo iz zgornje primerjave, razlik med streaming orodji ni preveč. Vse delujejo zelo podobno in tudi za kompatibilnost z zunanjimi orodji skrbijo. Storitve, ki so ustvarjene za oblak, imajo še možnost PUSH v consumerja, kar Apache ne podpira, saj je samostojna aplikacija. Prav tako so še razlike v velikosti sporočil, čeprav naj bi bila sporočila v streamingu manjša, tako da ta razlika ni občutna. Tu so še razlike pri hrambi podatkov, vendar je spet namen streaminga, da se podatki takoj procesirajo in dolgoročno hranijo drugje, če je potrebno.

Torej smo ugotovili, da so ta orodja predvsem ekvivalentna, in če moramo izbirati med njimi, je bolje, da se oziramo na zunanje dejavnike, kot so izkušnje s tem oblakom, kje so locirane povezane storitve, kakšno razmerje ima podjetje z oblačnim ponudnikom. Hkrati nam ni treba skrbeti, da ta storitev ne bi bila ustrezna.

Hipoteza, ki sem jo postavil pri tej temi, je bila:

> Orodja za Big Data streaming pri največjih treh oblačnih ponudnikih so funkcionalno enakovredna.

Ta hipoteza je potrjena.

3 RAZNOLIKOST SQL-BAZ V AZURE

Oblak Azure nam za hrambo podatkov v bazah ponuja veliko različnih možnosti. Tako imamo dejansko težavo izbrati bazo.

Glavne možnosti, s katerimi sem se srečal, so:

- Azure Cosmos DB,
- Azure SQL Database,
- Azure Database for PostgresSQL,
- Azure Database for MySQL,
- Azure Database for MariaDB,
- Azure Cache for Redis,
- Azure Synapse dedicated SQL pool,
- Azure Synapse serverless SQL pool.

Od teh osmih baz jih je polovica odprtokodnih rešitev: PostgreSQL, MySQL, MariaDB, Redis.

Redis, kot že polno ime Azure Cache for Redis nakazuje, pravzaprav ni baza, ampak Cache za začasne podatke. Tako vsekakor ni ustrezen za naš namen historizirane baze.

MariaDB, MySQL in PostgresSQL so vse odlične baze, vendar so vse te baze zasnovane za lokalne instance, ne oblak. Čeprav jih je verjetno Azure malo prilagodil, da se lahko lažje integrirajo v oblak, niso od začetka zasnovane z oblakom v mislih.

Te baze bi bile izvrstne v primeru, da bi morali nekaj postaviti zelo hitro, pri čemer naši razvijalci že poznajo ta okolja, ali samo migriramo okolje iz lokalnih strežnikov na oblak, vendar za namene tega diplomskega dela niso prava izbira.

Potem nam ostanejo samo še štiri Azurjeve oblačne rešitve: Cosmos, SQL Database, Synapse dedicated SQL pool in Synapse serverless SQL pool.

Pri tem lahko takoj izločimo Azure Cosmos DB, saj je to NoSQL-baza, medtem ko bodo naši podatki dobro strukturirani in relacijski. (Azure, 2024)

Azure SQL Database je primarna podatkovna baza na Azure, kar je razvidno tudi iz imena. Vendar je bolj usmerjena v transakcijske podatke, torej podpira veliko število uporabnikov, veliko število sej. Medtem je bil Synapse včasih celo poimenovan SQL Data Warehouse in je namenjen za analitične namene, kar ustreza našemu projektu. (All about Data Engineering, 2024)

Nazadnje se moramo še odločiti znotraj Synapse, ali bomo imeli dedicated SQL pool ali serverless SQL pool. Oba nam omogočata hrambo podatkov v datotekah na Data Lake, kar je precej ceneje kot v bazi, a še vedno lahko dostopamo s pomočjo SQL-vmesnika. Vendar serverless SQL pool omogoča samo to, ne vključuje namreč nobene baze, je samo okvir okoli datotek. To pomeni, da s pomočjo SQL ne moremo insertirati v tabele.

Ta ovira, da s pomočjo SQL ni mogoče urejati podatkov, pomeni, da je treba vse obdelave delati z neposrednim urejanjem zalednih datotek v programskem jeziku Python ali drugem programskem jeziku. Tak proces je možen, vendar po mojem mnenju neoptimalen, saj je priročno vsaj kakšne administracijske tabele urejati v tabelah.

Medtem Synapse dedicated SQL pool omogoča vse, kar ima serverless pool, poleg tega omogoča še DML-operacije in neeksterne tabele. (R, 2024)

Hipoteza, ki sem jo postavil pri tem vprašanju, je:

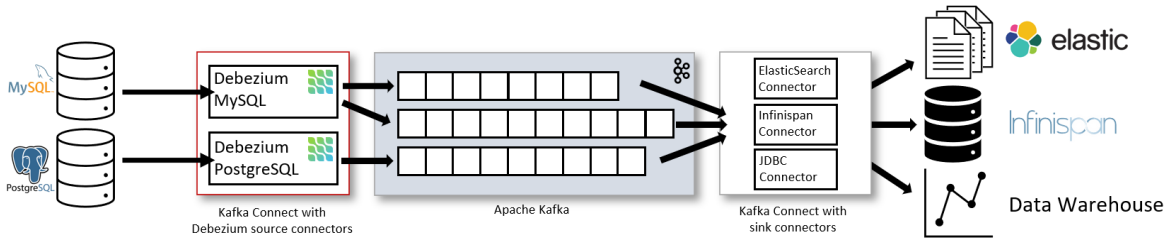
> Synapse serverless SQL pool je za naš namen (change data capture v historizirano tabelo) najprimernejši SQL storage na Azure.

To hipotezo sem ovrgel, saj je Synapse dedicated SQL pool ustreznejši, saj omogoča tako eksterne tabele kot tudi interne tabele z manipulacijo podatkov v SQL.

4 PROGRAM ZA HISTORIZACIJA CDC-ZAPISOV V SQL-BAZI

Projekt, ki je del te diplomskega dela, je Historizacija iz CDC-zapisov v Azure SQL-bazi.

Torej želimo postaviti celo pot podatkov od virne baze do ponorne. Pot je sestavljena iz virne baze, na njo je povezan program Debezium, ki na podlagi sprememb ustvari CDC (angl. change data capture) in te spremembe pošilja v event streaming, v našem primeru je to Azure Event Hubs.



Slika 2: Umestitev orodja Debezium v podatkovni tok

Vir: (https://debezium.io/documentation/reference/stable/_images/debezium-architecture.png)

Nato sledi skripta, napisana v programskem jeziku Python in pognana v Azure Functions, ki je sprožena, ko prispe kakšen zapis v Event

Hub in prebere tisti CDC-zapis ter ga shrani v ponorno bazo. Nazadnje smo še to

shranjevanje v ponorno bazo razbili na več različnih možnosti, da smo analizirali, kateri način shranjevanja historiziranih podatkov je najboljši.

Projekt se je začel z registracijo v Microsoft Azure. Na srečo večina oblačnih ponudnikov novim uporabnikom ponudi nekaj brezplačnih kreditov, da lahko preizkusijo storitve. V primeru Microsoft Azure je to 200 USD, vendar je veljavno samo 30 dni od začetka. Vendar se tudi teh 200 USD lahko kar hitro porabi, če nismo pozorni glede tega, kaj smo pustili prižgano.

Prva faza projekta je bila postavitev virne baze, postavitev in konfiguracija Debeziuma in postavitev Event Hubs. Cilj pri tem je bil, da ko se v določeni virni tabeli naredi Insert, Update ali Delete, v Event Hubu dobimo evidenco spremembe.

Pri tej postavitvi sem si pomagal s projektom, ki ga je pripravila ekipa Azure, ki je naredila prav to: https://github.com/azure-samples/azure-sql-db-change-stream-debezium/tree/master/.

(Azure, 2024)

Dodal bi opomnik, da po navadi ni priporočljivo kjerkoli javno objavljati gesla ali ključe oblačnih storitev. Vendar jih v tem diplomskem delu nisem odstranjeval iz vključenih skript, saj so to gesla samo za te storitve, medtem ko je naročnina Azure že ukinjena, tako da ključi ne morejo biti več uporabljeni.

4.1 Postavitev virne baze

Postavitev virne baze je bila presenetljivo preprosta, saj sem samo na spletni strani Azure poklikal, da želim SQL-bazo, default je pri tem Azure SQL Database, kar mi je povsem ustrezalo. Tam sem vpisal administratorja, geslo, ime baze, in v nekaj sekundah je bila baza postavljena.

Za vpogled v bazo sem povezal program VS Code, s pomočjo katerega sem programiral cel ta projekt. To sem dosegel tako, da sem se na Azure pod SQL-strežnik postavil na Networking, tam pod Public network access kliknil na Selected Networks in vpisal svoj IP-naslov. Del časa sem delal od doma, kjer imam statičen IP-naslov, in del iz tujine, kjer sem imel dinamičen IP-naslov, tako da sem moral ta korak večkrat ponoviti, da sem lahko dodajal nove IP-je.

Nato sem samo v VS Code potreboval SQL Server extension in v njem vpisal naslov strežnika, ki je bil viden v Azure, in sicer na tej SQL-bazi pod Server Name.

Ko sem imel dostop do baze, sem postavil tabele. Odločil sem se, da bom delal replikacijo tabele CUSTOMERS:

```
CREATE TABLE CUSTOMERS ( ID INT PRIMARY KEY,
 CUSTOMER_NAME VARCHAR(100),
 CUSTOMER_LOCATION VARCHAR(100),
 STATUS VARCHAR(10)
 );
```

(4.1.1) Opredelitev tabele CUSTOMERS

Vendar, da lahko naredim kakšno analizo, je potrebna malo večja količina podatkov, več, kot sem bil pripravljen ročno vpisovati. Tako sem ustvaril še tabelo z ukazi za Insert, Update, Delete:

```
CREATE TABLE TEST_COMMANDS (
ID INT PRIMARY KEY,
COMMAND VARCHAR(500)
);
```

(4.1.2) Opredelitev tabele TEST_COMMANDS

Nato sem to tabelo napolnil z ukazi za vnos generiranih testnih podatkov. Generirane podatke in ukaze za polnjenje tabele sem uredil v zelo preprosti skripti, spisani v programskem jeziku Python:

Programska koda v priloženi datoteki Prepare_data.py

(4.1.3) Programska koda za generiranje testnih podatkov

Nato sem imel pripravljene vse te komande, vendar sem moral poiskati še način, kako jih pognati, ne da bi jih moral ročno vnašati, tako da sem pripravil tri skripte še za to:

```
CREATE PROCEDURE ExecuteInsertCommands
AS
BEGIN
DECLARE @ID INT;
DECLARE @Command NVARCHAR(MAX);

DECLARE CommandCursor CURSOR FOR
SELECT ID, COMMAND
FROM TEST_COMMANDS
WHERE COMMAND LIKE 'INSERT%';

OPEN CommandCursor;
….....
```

Celotna koda na voljo v priloženi datoteki DML_procedures.sql.

(4.1.4) Programska koda za izvajanje pripravljenih testnih izjav

Ko je prišel čas za izvedbo dejanskega eksperimenta, sem moral samo pognati tri skripte in počakati nekaj minutk:

```
--EXECUTE TEST
ExecuteInsertCommands;

ExecuteUpdateCommands;

ExecuteDeleteCommands;
```

(4.1.5) Koda za pogon pripravljenih postopkov za vstavitev, posodobitev in brisanje podatkov

Seveda sem med postavitvijo celega sistema nekajkrat testiral z enim ali dvema ročnima vnosoma. Stvar, ki me je presenetila, je bila, da ko sem želel tabelo CUSTOMERS spet izprazniti, sem moral izvesti DELETE-izjavo in ne optimalnejše TRUNCATE-izjave, saj če je na tabeli omogočen CDC, ta ne dovoljuje TRUNCATE-izjave. Ko bolje pomislim, je to razumljivo, saj je DELETE-izjave treba tudi beležiti.

4.2 Postavitev Event Hubs

Naslednji korak je bil postavitev streaminga, torej v našem primeru Azure Event Hubs.

Ta postavitev je bila zelo preprosta, saj sem jo izvedel kar čez Azure CLI, saj sta samo dva ukaza:

```
# create group
az group create -n debezium -l eastus
# create eventhuvbs with kafka enabled
az eventhubs namespace create -n debezium -g debezium -l eastus –enable-kafka
```

(4.2.1) Programska koda za postavitev okolja Event Hubs

Nato sem moral pridobiti še connection string, s katerim se bo lahko Debezium povezal na Event Hub. Ukaz, ki je bil naveden v navodilih, ni deloval, saj se je vmes verjetno nekaj spremenilo. Tako sem samo s pomočjo GUI navigiral na Event Hub in pod Shared access policies odprl kar defaulten polic in pri njemu pogledal Connection string.

Nato sem želel še vpogled v ta novi event hub, kar je možno kar na VS Code, potreben je bil samo plugin Azure Event Hub Explorer, nakar sem se po njegovih navodilih povezal.

4.3 Postavitev in konfiguracija Debezium

Ko sta bila postavljena virna baza in Event Hub, na katerem želimo sporočila, sem moral povezati še ta dva, kar sem dosegel z odprtokodnim programom Debezium, ki brez obremenjevanja virne baze spremlja CDC-zapise in jih pošlje v stream.

Za ta namen sem moral na virni bazi ustvariti uporabnika za Debezium in ga omogočiti na tabeli CUSTOMERS.

Za ta namen sta bili v vodiču pripravljeni skripti:

```
-- Create user used in the sample
CREATE USER [debezium-wwi] WITH PASSWORD = 'Abcd1234!'
GO

-- Make sure user has db_owner permissions
ALTER ROLE [db_owner] ADD MEMBER [debezium-wwi]
GO

-- Enable CDC on database
EXEC sys.sp_cdc_enable_db
GO

-- Enable CDC on selected tables
EXEC sys.sp_cdc_enable_table N'Sales', N'Orders', @role_name=null, @supports_net_changes=0
EXEC sys.sp_cdc_enable_table N'Warehouse', N'StockItems', @role_name=null, @supports_net_changes=0
GO

-- Verify the CDC has been enabled for the selected tables
EXEC sys.sp_cdc_help_change_data_capture
GO
```

(4.3.1) Programska koda za ureditev dostopa na virno bazo za Debezium uporabnika in konfiguracija CDC na tabeli

Potem sem moral postaviti Debezium, ki se nahaja v svojem containerju na Azure.

Zaradi preprostosti je bila pri tem uporabljena javno dostopna Docker slika Debeziuma. Skripta za postavitev je bila vzeta iz vodiča in potem modificirana, da je delala z mojo postavitvijo:

```
az container create -g debezium -n dm-debezium \
        --image debezium/connect:1.8 \
        --ports 8083 --ip-address Public \
        --os-type Linux --cpu 2 --memory 4 \
…
```

Celotna koda je na voljo v priloženi datoteki debezium_setup.txt.

(4.3.2) Komanda za postavitev Debezium instance

Ta koda je poskrbela za kreacijo containerja in povezavo tega na Event Hub. Tam je ustvaril tudi tri Debezium interne topice.

Vendar na tej točki Debezium še ni bil povezan na noben vir, torej dejansko ni delal še nič.

Torej je bilo treba v Debezium registrirati konekcijo.

V ta namen sem modificiral konfiguracijsko datoteko register-sqlserver-eh.json in jo poleg skripte shranil v Azure CLI in pognal od tam.

```
{
 "name": "wwi",
 "config": {
        "snapshot.mode": "schema_only",
        "connector.class": "io.debezium.connector.sqlserver.SqlServerConnector",
        "database.hostname": "diplomska-db-source.database.windows.net",
        "database.port": "1433",
        "database.user": "debezium-wwi",
        "database.password": "Abcd1234!",
        "database.dbname": "diplomska-db-source",
        "database.server.name": "SQLAzure",
        "tasks.max": "1",
        "decimal.handling.mode": "string",
        "table.include.list": "dbo.CUSTOMERS",
        "transforms": "Reroute",
        "transforms.Reroute.type": "io.debezium.transforms.ByLogicalTableRouter",
        "transforms.Reroute.topic.regex": "(.*)",
```

```
        "transforms.Reroute.topic.replacement": "wwi",
        "tombstones.on.delete": false,
        "database.history": "io.debezium.relational.history.MemoryDatabaseHistory"
  }
}
```

(4.3.3) Konfiguracijska datoteka s podatki za povezavo Debezium instance in virne baze. Datoteka poimenovana instanceregister-sqlserver-eh.json

```
#!/bin/bash
# Strict mode, fail on any error
set -euo pipefail

export RESOURCE_GROUP="debezium"
export CONTAINER_NAME="dm-debezium"

echo "finding debezium ip"
export DEBEZIUM_IP=`az container show -g $RESOURCE_GROUP -n $CONTAINER_NAME -o tsv --query "ipAddress.ip"`

echo "registering connector"
curl -i -X POST \
-H "Accept:application/json" -H "Content-Type:application/json" \
http://${DEBEZIUM_IP}:8083/connectors/ \
-d @register-sqlserver-eh.json
```

(4.3.4) Programska skripta za konfiguracijo povezave med Debezium in virno bazo. Datoteka poimenovana 01-register-connector.sh

Ko sem pognal to skripto, se je v Event Hubu ustvaril nov topic, poimenovan wwi (poimenovanje izhaja iz vodiča, kjer se je delalo na bazi World Wide Importers, in nisem spreminjal, saj ni pomembno). Ko sem se v VS Code povezal na ta topic, da sem lahko poslušal in potem izvedel DML-operacijo na CUSTOMERS-tabeli na virni bazi, sem videl CDC-sporočilo, kakršnega generira Debezium.

4.4 Postavitev ponorne baze

V teoretični primerjavi sem ugotovili, da bo kot ponorna baza za analitične namene najboljša Synapse Dedicated SQL Pool. Vendar sem pri tem hitro opazil zaplet v tem, da najcenejša

različice te baze stane 1,5 evra na uro. Zato sem bazo, ko je nisem potreboval za testiranje, imel ugasnjeno, kar je pametna ideja z vsemi storitvami v oblaku.

Postavitev je bila izjemno preprosta, spet sem jo lahko poklikal s pomočjo Azure GUI. Najprej je treba omogočiti Synapse, potem tam ustvariti Dedicated SQL pool, vpisati konfiguracijo, omogočiti dostop s svojega IP-ja in se povezati na bazo s pomočjo VS Code. A to sem opisal že v točki 4.1.

Za namene preizkusa sem na bazi ustvaril tabele, v katere bomo polnili podatke. Tabele so bile različnih oblik, za različne načine shranjevanja historičnih podatkov:

```
CREATE TABLE CUSTOMERS_CDC (
ID INT,
CUSTOMER_NAME VARCHAR(100),
CUSTOMER_LOCATION VARCHAR(100),
STATUS VARCHAR(10),
EVENT_DATE INT,
EVENT_TYPE VARCHAR(1)
);

CREATE TABLE CUSTOMERS_SCD1 (
ID INT,
CUSTOMER_NAME VARCHAR(100),
CUSTOMER_LOCATION VARCHAR(100),
STATUS VARCHAR(10)
);

CREATE TABLE CUSTOMERS_SCD2 (
ID INT,
CUSTOMER_NAME VARCHAR(100),
CUSTOMER_LOCATION VARCHAR(100),
STATUS VARCHAR(10),
VALID_FROM INT,
VALID_UNTIL INT
);

CREATE TABLE CUSTOMER_NAME_SCD5 (
ID INT,
```

```
CUSTOMER_NAME VARCHAR(100),
VALID_FROM INT
);
CREATE TABLE CUSTOMER_LOCATION_SCD5 (
ID INT,
CUSTOMER_LOCATION VARCHAR(100),
VALID_FROM INT
);
CREATE TABLE STATUS_SCD5 (
ID INT,
STATUS VARCHAR(10),
VALID_FROM INT
);

CREATE TABLE CUSTOMERS_SCD6 (
ID INT,
CUSTOMER_NAME VARCHAR(100),
CUSTOMER_LOCATION VARCHAR(100),
STATUS VARCHAR(10),
VALID_FROM INT,
VALID_UNTIL INT,
CUR_CUSTOMER_NAME VARCHAR(100),
CUR_CUSTOMER_LOCATION VARCHAR(100),
CUR_STATUS VARCHAR(10)
);
```

(4.4.1) Programska skripta za ustvarjanje ponornih tabel

Nato sem dodal še tabelo, v katero sem ob insertu v ponorno bazo shranjeval, kako dolgo je trajal insert v posamično tabelo:

```
CREATE TABLE QUERY_TIMES (
EVENT_TYPE VARCHAR(1),
CDC_TIME FLOAT,
SCD1_TIME FLOAT,
SCD2_TIME FLOAT,
SCD5_TIME FLOAT,
SCD6_TIME FLOAT
);
```

(4.4.2) Programska skripta za ustvarjanje tabele za evalvacijo performanc operacij

Za konec sem moral še ugotoviti, kako evaluirati, koliko časa vsak SELECT na teh tabelah traja, nakar sem na koncu v dokumentaciji Synapse našel tabelo sys.dm_pdw_exec_requests:

SELECT request_id, status, submit_time, command, total_elapsed_time FROM sys.dm_pdw_exec_requests where session_id = 'SID3927' AND COMMAND LIKE 'SELECT%' order by submit_time desc;

(4.4.3) SQL-poizvedba za pridobitev podatkov o zahtevnosti izvedenih poizvedb

4.5 Postavitev funkcije za vnos v ponorno bazo

Za prenos podatkov iz Event Hub v ponorno bazo sem uporabljal Azure Functions, kjer lahko spišemo program, ki se izvaja v oblaku brez dediciranega strežnika.

Za ta namen sem najprej s pomočjo Azure GUI poklikal, da sem ustvaril nov Azure Functions App, ki je dejansko okolje, znotraj katerega se ustvari Azure Function.

To funkcijo sem pisal znotraj VS Code, tako da sem potreboval še dodatni plugin, tokrat kar cel Azure plugin, ki je vključeval Azure Functions. (Azure, 2024)

S pomočjo tega plugina sem se povezal na svojo naročnino in lahko upravljal številne storitve. Vendar je mene je samo Function App.

S klicem funkcije tega plugina sem ustvaril novo funkcijo znotraj Function Appa, kjer sem tudi vpisal, da želim, da se funkcija sproža, ko pride novo sporočilo v Event Hub, in podal connection string od Event Hub topica. Poleg funkcije se je v lokalnem okolju ustvarila tudi datoteka local.settings.json, kjer se je ta connection string shranil, poleg drugih spremenljivk.

Ta datoteka je namenjena shranjevanju spremenljivk, ki so sicer shranjene v Azure Functions, kot so gesla in connection stringi, torej skrivnosti, ki jih ne želimo shraniti neposredno v kodo. In s pomočjo te datoteke lahko funkcijo poženemo lokalno, za namene debugiranja. Čeprav je bilo treba za ta namen z npm instalirati še knjižnico azure-functions-event-hubs.

Ko sem imel urejeno povezljivost z Event Hubom in se je funkcija sprožila, ko je prišlo novo sporočilo, sem uredil še povezljivost na ponorno bazo, kar je bilo zelo preprosto s knjižnico pyodbc.

Nazadnje sem moral razbrati še sporočila, ki jih pošilja Debezium. Primer sporočila je:

{

```
  "schema":{
    "type":"struct",
    "fields":[
      {
        "type":"struct",
        "fields":[
          {
            "type":"int32",
            "optional":false,
            "field":"ID"
          },
```

…

Celotna koda je na voljo v priloženi datoteki debezium_message.json.

(4.5.1) JSON-formatirano sporočilo, ki prihaja iz sistema Debezium

To je JSON-sporočilo, kjer se najprej opredeli tabela, potem stara vrstica v tabeli in kakšna je vrstica sedaj. Nazadnje so tu še podatki, kdaj je bilo sporočilo poslano in kakšna je operacija. C je za insert, kjer so podatki za vrstico before: null, U za update, kjer imamo vrstico prej in potem, ter D za delete, kjer je after:null. (Debezium, 2024)

Tega sporočila ni bilo težko brati s knjižnico JSON.

Končna različica programa je:

```
import azure.functions as func
import logging
import pyodbc
import json
import time

# Define your connection parameters
server = 'diplomskasynapse.sql.azuresynapse.net'
database = 'diplomskadedicatedpool'
username = 'diplomska-admin'
```

…

Celotna koda je na voljo v priloženi datoteki function_app.py.

(4.5.2) Program, spisan v programskem jeziku Python, namenjen branju Azure Event Streama, dekodiranju JSON-sporočila in shranjevanju podatkov v končnem formatu v ponorno bazo

Poleg te funkcije je bilo seveda ustvarjenih še nekaj konfiguracijskih datotek:

```
{
 "bindings": [
   {
    "type": "eventHubTrigger",
    "name": "azeventhub",
    "direction": "in",
    "path": "debeziumdiplomska",
    "connection": "DebEventHubConnString"
   }
 ],
 "disabled": false
}
```

(4.5.3) Konfiguracijska datoteka, povezana z zgornjo function_app.py, za opredelitev proženja programa. Datoteka: function.json

```
{
 "version": "2.0",
 "logging": {
   "applicationInsights": {
     "samplingSettings": {
       "isEnabled": true,
       "excludedTypes": "Request"
     }
   }
 },
 "extensionBundle": {
   "id": "Microsoft.Azure.Functions.ExtensionBundle",
   "version": "[3.*, 4.0.0)"
 }
}
```

(4.5.4) Konfiguracijska datoteka, povezana z zgornjo function_app.py, za opredelitev okolja. Datoteka: host.json

Pri tem bi izpostavil pomembnost, da je konfiguracija v function.json enaka kot na začetku dejanske funkcije:

```
@app.event_hub_message_trigger(arg_name="azeventhub", event_hub_name="debeziumDiplomska",
connection="DebEventHubConnString")
```

(4.5.5) Izsek kode iz programa function_app.py pod točko 4.5.2

Ko je funkcija delovala v lokalnem okolju, sem jo s pomočjo VS Code Azure plugina lahko prenesel v Azure, tam je bila možnost deploy, ki prepiše okolje na Azure s tem, kar imam lokalno.

Sicer se spremenljivke v local.settings.json ne prenesejo na oblak, saj v dejanskem projektu ne bi razvijali na produkciji, torej bi želeli razvoj usmeriti na eno bazo in produkcijo na drugo. Tako sem te spremenljivke moral potem še s pomočjo Azure GUI vnesti v Azure Function App.

4.6 Preizkus in meritve

Ko so vse komponente delovale, je sledil še zadnji integration test, kjer sem preizkusil celo pot od začetka do konca. Ročno sem spremenil eno vrstico na viru in nekaj sekund kasneje je bila spremenjena tudi na ponorni bazi.

Pri tem sem šele odkril, da je funkcija počasnejša, kot sem pričakoval, da je dejansko trajalo nekaj sekund za vsako spremembo.

Nisem prepričan, od kod izvira ta počasnost, verjetno bi lahko učinkoviteje naredil skripto, vendar za namene preizkusa to ne predstavlja težave, saj so vse operacije enako upočasnjene.

Tako sem počistil virno tabelo CUSTOMERS, potem počistil še vse ponorne tabele in pognal postopke na virni bazi, ki so vnašale podatke v tabelo CUSTOMERS.

Pogon teh postopkov je trajal nekaj minut, in sicer zaradi zamude, ki sem jo sam dodal v postopke. Vendar se je zaradi slabše performance Azure Function naredila precej velika čakalna vrsta v Event Hubu, pri čemer je trajalo več ur, da so se obdelale vse operacije.

Nato je sledil samo še zadnji korak, na ponornih tabelah izvesti poizvedbe in ugotoviti, katere poizvedbe so najhitrejše.

Najprej sem s pomočjo dogodkov v CUSTOMERS_CDC ugotovil, kdaj je bil konec dogodkov, da sem za trenutni čas določil kar koli več kot tisto ter našel čas, ko so se končali inserti in začeli updatei, da sem dobil datum pred spremembami. Nazadnje sem našel še eno stranko, ki se ji je lokacija posodobila.

Za reprezentativni vzorec uporabe sem se namreč odločil za sedem vprašanj, na katera sem želel odgovore:

- Koliko strank je trenutno?

- Koliko je strank v trenutku v preteklosti?
- Kako izgleda določena stranka trenutno?
- Kako je izgledala določena stranka v določenem trenutku v preteklosti?
- Katere stranke so v določenem mestu?
- Katere stranke so kadar koli bile v določenem mestu?
- Katere stranke so bile v določenem trenutku v preteklosti v določenem mestu?

Poizvedbe in rezultate te analize bom predstavil v naslednjem poglavju.

5 NAJBOLJŠI SCD

SDC ozriroma Slowly Changing Dimension je način shranjevanja historičnih podatkov v tabelah. Po navadi pri principih podatkovnega skladiščenja govorimo o factih oziroma dejstvih, ki so dogodki. Prav tako tudi o dimenzijah, ki se vežejo na dejstva, da jih opremijo s podrobnostmi. Tako imamo lahko recimo dejstvo nakup in v dejstvu je stolpec z ID-stranke, medtem ko so podatki o stranki v povezani dimenzijski tabeli. Če ta stranka redno nakupuje, vendar je vmes nekajkrat recimo spremenila naslov, se pojavi potreba po historizaciji dimenzijske tabele. Cena poštnine je recimo lahko odvisna od lokacije stranke, in če se stranka v vmesnem času preseli, se lahko cena poštnine spremeni, a še vedno moramo za stare nakupe vedeti, zakaj je bila poštnina tako velika.

Tipi historiziranih dimenzij so opisani v eni izmed najpomembnejših knjig podatkovnega skladiščenja, »The Data Warehouse Toolkit«, avtorja Ralph Kimball.

V svoji knjigi Kimball opiše osem različnih oblik SCD:

- SCD 0: V dimenziji se vedno ohranijo samo prve vnesene vrednosti.
- SCD 1: V dimenziji se vrednosti vedno posodobijo na najnovejše.
- SCD 2: Ob vsaki spremembi se doda nova vrstica, vrstice imajo začetek in konec, tako da se stara vrstica zapre in nova pa odpre.
- SCD 3: Ob spremembi se stara vrednost shrani v drug stolpec. Torej ima tabela podatek recimo o naslovu, a hkrati ima tudi podatek o starem naslovu, lahko tudi podatek o tem, kdaj je bil spremenjen naslov. Ko bi prišlo do še ene spremembe, bi se tista prva sprememba prepisala in izgubila.
- SCD 4: Če se en stolpec v dimenziji zelo pogosto spreminja, se lahko naredi iz tega stolpca lastna dimenzija, ki se neposredno poveže na dejstvo, in samo s tem dejstvom poveže na glavno dimenzijo. To bi bilo lahko recimo, če imamo dejstvo nakup, dimenzijo stranka in lastnost stranke je tudi bančno stanje, potem bančno stanje ločimo od stranke in povežemo neposredno na dejstvo.
- SCD 5: To je modificiran SCD 4, kjer se poleg bančnega stanja, povezanega na nakup, trenutno stanje v stilu SCD 1 še vsakič posodobi na trenutno aktivni vrstici glavne dimenzije

stranke. Tako da če gledamo stranko, vedno vidimo samo trenutno stanje, medtem ko je na nakup povezano tudi stanje v tistem trenutku.

- SCD 6: To je modificiran SCD 2, kjer ima dimenzijska tabela dodatne stolpce za trenutno stanje. Torej, ko se stanje spremeni, se odpre nova vrstica v dimenziji, hkrati se tudi posodobijo vse prejšnje vrstice v historični verigi te stranke, da imajo v dediciranih stolpcih zapisane trenutno aktivne podatke.
- SCD 7: To je modificiran SCD 2, kjer na dejstvu vidimo tako stranko v trenutku nakupa kot tudi stranko, kot je v trenutnem trenutku. Torej se tabela dejstev povezuje na SCD 2-tabelo dvakrat, enkrat na zapis v tistem trenutku in enkrat na historično verigo sprememb stranke na trenutno aktiven zapis.

(Ross & Kimball, 2013)

To so tipi spremenljivih dimenzij po Kimballu in SCD 0–3 so zelo splošno prepoznani.

Medtem opredelitve SCD 4-7 niso splošno prepoznane.

In po mojem vedenju in spletnih virih sta SCD 4 in SCD 5 precej drugačna.

- SCD 4: To je sestavljena dimenzija iz glavne dimenzije tipa SCD 1, torej kjer se vedno beleži trenutno stanje, in iz historične tabele, kjer beležimo vse spremembe na dimenziji in datum, ko je do spremembe prišlo. (Shenoy, 2024) (Schott, 2024)
- SCD 5: To je spet sestavljena dimenzija, sestavljena iz glavne tabele tipa SCD 1, ki namesto vrednosti na spremenljivih stolpcih vključuje ključ poddimenzije za ta stolpec, medtem ko poddimenzija vključuje ta glavni ključ, datum spremembe in vrednost takrat veljavno. (Whiteley, 2024) (Slowly changing dimension, 2024)

Ti dve opredelitvi SCD 4 in SCD 5 sta mi bolj poznani, tako da ju bom v nadaljevanju opredelil tako.

Potem nam preostane le še en tip historizacije, ki sicer ni najprimernejši za povezavo na dejstva, vendar je zelo preprost za grajenje iz CDC-zapisov, saj je samo CDC-tabela.

Torej je to tabela, v kateri shranjujemo neposredno zapise CDC, v primeru insert, update se zapišejo nove vrednosti, tip operacije in datum operacije. Medtem ko se v primeru delete zapiše

tako imenovani tombstone oziroma nagrobnik, kjer so v vrstici stari podatki, tip operacije in datum operacije.

V praksi se tak tip tabele ne uporablja kot dimenzija za povezovanje na tabelo dejstev, vendar sem ga že večkrat uporabljal, ko sem potreboval evidenco sprememb, če se je iz virne tabele gradila kakšna naprednejša sestavljena tabela in bi moral kdaj dokazovati, kakšni so bili virni podatki v nekem trenutku.

5.1 Naša primerjava

V svojem primeru imamo samo dimenzijo, brez tabele dejstev, in želimo obdržati historične podatke. Tako sem za primerjavo izbral SCD 2, SCD 4, SCD 5, SCD 6, in CDC ter za primerjavo še SCD 1, čeprav ne shranjuje historičnih podatkov.

Težavnost operacij ob spremembi sem dobil iz v prejšnjem poglavju opredeljenega programa. Ob izvedbi vsakega stavka Insert, Update in Delete sem spremljal, kako dolgo je trajalo, in to shranil v tabelo. Povprečni rezultati, skupaj z mojo oceno zapletenosti poizvedbe, so podani spodaj:

Tabela 2: Primerjava kompleksnosti in performance SQL-kode za vnos podatkov v ponorno bazo

	Virna operacija	Povprečen čas poizvedbe (ms)	Ocena težavnosti poizvedbe (1–10)
SCD1	INSERT	2184.42	1
	UPDATE	4524.55	1
	DELETE	4180.02	1
SCD2	INSERT	2186.21	2
	UPDATE	4796.03	5
	DELETE	4325.27	4
SCD4(CDC+SCD1)	INSERT	4385.86	3
	UPDATE	6707.87	3
	DELETE	6350.08	3
SCD5(SCD5+SCD1)	INSERT	7258.89	5

	UPDATE	11878.69	5
	DELETE	16903.79	5
SCD6	INSERT	2214.35	2
	UPDATE	8368.59	7
	DELETE	7984.37	5
CDC	INSERT	2201.44	1
	UPDATE	2183.32	1
	DELETE	2170.06	1

Iz teh podatkov je razvidno, da je CDC seveda najbolj performanten, pri čemer je razlika med SCD 1 in SCD 2 skoraj zanemarljiva, medtem ko imata SCD 4 in SCD 5 znatno slabše performance. SCD 6 je na Insertu skoraj identična SCD 2, kar je pričakovano glede na to, da je praktično izjava Insert enaka, medtem ko je na Update in Delete skoraj dvakrat slabša, kar je tudi pričakovano, saj se povsod še dodaja Update, ki vse stare zapise posodobi.

Tipe SCD bi po DML operacijah rangiral CDC, SCD1, SCD 2, SCD 4, SCD 6, SCD 5.

Vendar se podatek v podatkovnem skladišču vstavi le enkrat, pri čemer se uporablja dolgo časa, tako da je pomembnejše, kako se tabele obnašajo pri poizvedbah.

V ta namen sem za vsako od tabel (oziroma skupek tabel) in za vsako od vprašanj, ki sem si jih zadal, pripravil poizvedbo, s katero bi ugotovil, kakšen je odgovor na vprašanje iz tabele. Te poizvedbe sem potem tudi pognal in s pomočjo tabele sys.dm_pdw_exec_requests ugotovil trajanje poizvedbe. Poleg tega sem spet ročno ocenil kompleksnost poizvedbe.

Tabela 3: Primerjava poizvedb na različnih oblikah SCD

	Vprašanje	Ocena težavnosti poizvedbe (1–10)	Povprečen čas poizvedbe (ms)
SCD1	Koliko je strank trenutno?	1	296.5
	Koliko je strank v enem trenutku v preteklosti?		0
	Kako izgleda stranka z imenom X trenutno?	1	163.5
	Kako je izgledala stranka X v preteklosti?		0
	Najdi vse stranke z lokacijo, ki vključuje x.	1	179.5
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.		0
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.		0
SCD2	Koliko je strank trenutno?	3	257.5
	Koliko je strank v enem trenutku v preteklosti?	3	265
	Kako izgleda stranka z imenom X trenutno?	3	195
	Kako je izgledala stranka X v preteklosti?	3	187
	Najdi vse stranke z lokacijo, ki vključuje x.	3	171
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.	4	960.5
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.	3	203
SCD4 (CDC+SCD1)	Koliko je strank trenutno?	1	265

	Vprašanje	Ocena težavnosti poizvedbe (1–10)	Povprečen čas poizvedbe (ms)
	Koliko je strank v enem trenutku v preteklosti?	8	2702.5
	Kako izgleda stranka z imenom X trenutno?	1	171
	Kako je izgledala stranka X v preteklosti?	8	1515
	Najdi vse stranke z lokacijo, ki vključuje x.	1	171
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.	6	960.5
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.	8	1788
SCD5	Koliko je strank trenutno?	1	265
	Koliko je strank v enem trenutku v preteklosti?		0
	Kako izgleda stranka z imenom X trenutno?	1	171
	Kako je izgledala stranka X v preteklosti?	7	2912
	Najdi vse stranke z lokacijo, ki vključuje x.	1	173
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.		0
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.		0
SCD6	Koliko je strank trenutno?	3	263.5
	Koliko je strank v enem trenutku v preteklosti?	3	262.5

	Vprašanje	Ocena težavnosti poizvedbe (1–10)	Povprečen čas poizvedbe (ms)
	Kako izgleda stranka z imenom X trenutno?	3	197
	Kako je izgledala stranka X v preteklosti?	3	185
	Najdi vse stranke z lokacijo, ki vključuje x.	3	171.5
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.	4	974.5
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.	3	203
CDC	Koliko je strank trenutno?	7	1913.5
	Koliko je strank v enem trenutku v preteklosti?	7	1945
	Kako izgleda stranka z imenom X trenutno?	7	1578
	Kako je izgledala stranka X v preteklosti?	7	1507
	Najdi vse stranke z lokacijo, ki vključuje x.	7	1750
	Najdi vse stranke, ki so kadar koli imele lokacijo, ki vključuje x.	4	1007.5
	Najdi vse stranke, ki so v enem trenutku v preteklosti imele lokacijo, ki vključuje x.	7	1695

Celotna tabela, vključno s poizvedbami, je na voljo v priloženi datoteki Primerjava_poizvedb.xlsx.

Na podlagi zgornjih podatkov vidimo, da je seveda najpreprostejši in najbolj performanten SCD 1, vendar ta ne more odgovoriti na nobeno historično vprašanje.

SCD 2 je zelo performanten pri vseh poizvedbah in vse so zelo podobne, torej ko razumeš eno poizvedbo, je zelo preprosto razumeti tudi vse ostale.

Skoraj identičen SCD 2 je tudi SCD 6, saj je enak, le nekaj dodatnih stolpcev ima. Vendar ti stolpci pri nobenem od vprašanj niso prišli v poštev, tako da so bile poizvedbe poponoma enake. Prednosti SCD 6 se lahko pokažejo v primeru, da bi imeli dejansko tabelo dejstev in bi tam zraven pridružili vrednosti iz dimenzije. Razlike v performanci so zanemarljive.

SCD 4 je zelo raznolik, zelo preprost in performanten za trenutna stanja, saj tam gleda SCD 1 tabelo, vendar takoj, ko potrebujemo kakšno zgodovinsko informacijo, se poizvedbe znatno zapletejo in tudi čas za izvedbo se močno podaljša.

SCD 5 ima omejitev, ker je namreč osnovna tabela SCD 1, v primeru, da se vrstica na viru zbriše, ne ostane noben nagrobnik. Torej na vprašanja o splošni zgodovini ne moremo odgovoriti. Vemo sicer, kako je izgledala določena stranka v preteklosti, vendar ne tudi, koliko je bilo v preteklosti strank, ker so katere bile lahko izbrisane. Obstaja precej realnih scenarijev, v katerih se stranke nikoli ne brišejo in kjer bi bil lahko SCD 5 uporabljen, ampak tudi za tiste scenarije ni performanten in dodaja veliko kompleksnosti.

CDC je zelo konstanten, ampak konstantno zapleten in počasen. Kot sem omenil, se CDC v praksi tudi ne uporablja za sisteme, kjer je veliko poizvedb, uporablja se bolj za beleženje zgodovine za namene revizije, kar se redko gleda.

Moja hipoteza pri tem vprašanju je bila:

> SCD 6 je najboljši model historizacije tabele za namene, kjer je veliko branja trenutnega stanja, potrebno pa je tudi stanje v kakršnem koli trenutku v preteklosti.

Na podlagi pridobljenih podatkov bi to hipotezo ovrgel.

SCD 6 je sicer zelo primeren, vendar je bolj tipičen SCD 2 enako performanten pri poizvedbah, vendar je bolj performanten pri DML-operacijah. SCD 6 je modificiran SCD 2 in je primernejši samo v robnih primerih, v katerih želimo hkrati vedeti stanje v zgodovini in trenutno stanje.

6 PRIHODNOST PODATKOVNIH BAZ

V hrambi velike količine podatkov se je že dolgo nazaj razvila potreba po sistemu za učinkovito hrambo in vpogled v te podatke. Leta 1970 je E. F. Codd v svojem članku » A Relational Model of Data for Large Shared Data Banks« postavil prvo zasnovo za relacijsko podatkovno bazo. To zasnovo je dodelal še do leta 1974 in leta 1979 je temu sledil še SQL-programski jezik, s katerim vpogledamo v podatke v relacijskih bazah.

Od takrat so se razvile številne različne relacijske baze, vsaka s svojimi posebnostmi, vendar so se vse še vedno držale osnovnega modela relacijskih baz. Prav tako so skoraj vse uporabljale SQL-jezik, ki se je čez leta tudi nadgrajeval. Številne baze imajo sicer svoje posebnosti pri SQL-u, tako da, čeprav je SQL med različnimi bazami dovolj podoben, da ga verjetno lahko razumemo, niso vsi enaki in med seboj kompatibilni.

V zadnjih desetletjih se pojavlja in raste tudi čedalje več alternativnih podatkovnih baz. Najbolj znani tipi so vektorske baze in NoSQL-baze.

Vektorske baze so narejene za hrambo podatkov v večdimenzionalno vektorski obliki. Torej kjer bi relacijska baza shranila podatek v vrstico tabele, ga vektorska baza shrani kot nestrukturiran objekt, skupaj z vektorjem, ustvarjenim iz njegovih lastnosti. Torej, na primeru stranke bi recimo shranili en objekt na primer s sliko stranke in podatki o strani, hkrati bi imeli s to stranko še koordinate v N-dimenzionalnem prostoru sestavljene iz na primer starosti, velikosti mesta, kjer živi, premoženja ... In stranko bi potem iskali po teh koordinatah.

Čeprav namen vektorskih baz ni, da bi se stranko iskali, namen je, da se izkoristijo lastnosti N-dimenzionalnega prostora, da se ugotovi, kako podobna je ena stranka drugi, saj lahko iz koordinat preprosto poračunamo relacijo med njima. To nam omogoča tudi iskanje skupin podobnih objektov.

Primer stranke, ki sem ga pri tem uporabil, je bil za razumevanje, v realnosti se vektorske baze največ uporabljajo v umetni inteligenci, napredni podatkovni analitiki in na podobnih področjih. Tam se namreč lahko na primer iz slike mačke naredi N-dimenzionalni vektor, ki opiše to sliko, in druge slike mačk bodo imele podobne N-dimenzionalne vektorje, tako da bomo lahko ugotavljali, kaj so mačke in kaj delfini.

Vektorske baze so zelo koristne, vendar samo za nekatere posebne namene, pri čemer bi bile relacijske baze zelo neustrezne.

NoSQL-baze oziroma Not Only SQL-baze so bližje relacijskim bazam. Njihova posebnost je prilagodljivost. Relacijske baze so namenjene za močno strukturirane, med seboj povezane podatke. Medtem ko NoSQL-baze dovoljujejo manj strukturirane podatke, pri čemer je struktura podatkov prepuščena uporabniku, in lahko se vnesejo popolnoma nestrukturirani podatki in na njihovo strukturo gledamo šele ob branju. Še ena lastnost NoSQL-baz je, da po navadi žrtvujejo malo konsistence v zameno za hitrost. Torej se lahko zgodi, da baza vrne star podatek, ki se je vmes že spremenil, ker je lahko razporejena po številnih različnih strežnikih, s čimer imajo relacijske baze težave, saj tam napačen podatek nikoli ni dovoljen.

(Finkel, 2024)

Ne obstaja sicer samo ena opredelitev NoSQL-baze, saj lahko hranijo podatke na različne načine.

Najpogostejši sta key-value pair in document database, to pomeni, da so podatki hranjeni predvsem kot JSON, torej kombinacije imena spremenljivke in vsebine, pri čemer je lahko vsebina tudi seznam več imen in priloženih spremenljivk.

Sicer pod NoSQL spadajo tudi graf baze, kolumnarne baze, time-series baze in object storage.

NoSQL-baze so bolj konkurenti relacijskim bazam in v nekaterih funkcionalnostih, kjer je hitrost pomembnejša od učinkovitosti in ni kritično, da so rezultati nujno popolni, so NoSQL-baze boljše. Trenutno gre pri tem za spletne vsebine, brskalnike ...

V teh vsebinah, kjer so se do sedaj pogosto uporabljale relacijske baze, preprosto zato, ker ni bilo druge možnosti, pričakujem, da bodo relacijske baze izgubile pomembnost.

Vendar je v bolj klasičnih vsebinah, kjer so podatki strukturirani, dvomljivo, da bodo NoSQL-baze pridobile več uporabe.

(Abolo, 2024)

Moja hipoteza na tem področju je bila:

Navkljub priljubljenosti NoSQL- in vektorskih baz bodo tudi še v prihodnje najpomembnejše relacijske baze.

Rekel bi, da je ta hipoteza deloma potrjena, saj sem ugotovil, da v strukturiranih podatkih NoSQL-baze in vektorske baze ne konkurirajo. Vendar prisotnost NoSQL- in vektorskih baz vsekakor pomeni, da bo v primerih, kjer relacijske baze niso optimalne, izbrana baza, ki je. Tako da bodo verjetno relacijske baze izgubile nekaj pri tržnem deležu.

7 NAPOTKI ZA NADALJNJO RAZISKAVO

Med raziskavo smo odgovorili na številna vprašanja, vendar jih vedno obstaja še več. Če bi želeli raziskavo poglobiti, bi lahko pri primerjavi streaming aplikacij primerjali še glede na ceno. Tega v tem diplomskem delu nisem naredil, saj so ceniki za oblačne storitve neberljivi, razbiti na veliko količino slabo opredeljenih postavk. In čeprav primerjamo streaming storitve, bi morali za dejansko uporaben podatek primerjati celoten projekt, saj nam nič ne koristi, da je streaming storitev poceni, če je potem hramba podatkov precej dražja ali je večji strošek za prenos podatkov ven iz oblaka. To bi lahko temeljito raziskali s postavitvijo reprezentativnega projekta na vseh oblakih in storitvah hkrati.

Pri primerjavi baz bi za nadaljnje raziskovanje lahko razširili primerjavo Synapse serverless SQL poola. Ta bi namreč bil verjetno cenovno ugoden za poizvedbe, medtem ko bi se za vstavitev podatkov lahko s programskim jezikom Python neposredno modificirale zaledne datoteke.

Za ugotavljanje najboljše oblike SCD bi za zanesljivejše meritve potrebovali znatno večjo količino podatkov in bolj reprezentativne podatke. Pri tem smo imeli namreč samo naključno generiran tekst. Za boljšo primerjavo in hkrati tudi poenostavitev eksperimenta bi lahko tudi ukaze za modifikacijo podatkov prožili neposredno, brez celega toka od virne baze, čez debezium, streaming, functions. To bi nam prihranilo veliko časa pri postavitvi eksperimenta in ponudilo boljše podatke, saj se ne bi dodajala napaka iz vseh vmesnih sistemov.

8 SKLEP

V diplomskem delu sem želel raziskati prihodnost podatkovnega inženiringa in se naučiti uporabljati nekaj modernih orodij za podatkovni inženiring. Raziskoval sem, kako se dela Big Data Streaming v različnih oblačnih okoljih in katero je najboljše. Moja hipoteza H1 je bila, da so si orodja v največjih treh oblačnih okoljih ekvivalentna. Pri tem sem prišel do odgovora, da so si oblačna okolja med seboj dokaj podobna in je izbira bolj odvisna od tega, kaj razvijalci poznajo in kje se dobi najboljša pogodba. Ta rezultat je bil tudi pričakovan, saj če bi bile storitve znatno različne, bi trg poskrbel, da najboljša zmaga in se ostale prilagodijo.

Zanimalo me je tudi, katere so SQL-baze na oblačnem okolju Azure in katera bi bila najprimernejša za namene podatkovnega skladiščenja. Pri hipotezi H2 sem pričakoval, da bo najustreznejša baza Synapse Serverless SQL Pool, saj omogoča hrambo podatkov v datotekah na poceni diskovju, namesto v bazi, medtem ko se poizvedbe izvedejo brez dediciranega strežnika. Vendar se je izkazalo, da je to večja omejitev, kot sem pričakoval, saj Serverless SQL Pool ne omogoča nobenih standardnih baznih tabel, ki so bile za naš namen potrebne. Prav tako tudi iz izkušenj vem, da so pomembne že za namene administracije procesov. Tako sem ugotovil, da je najustreznejša baza dejansko Synapse Dedicated SQL Pool, saj vključuje opcijo za hrambo podatkov na poceni diskovju, hkrati tudi vključuje osnovno relacijsko bazo.

Za tem sem se poglobil v hrambo historičnih podatkov. Historični podatki predstavljajo zanimiv izziv, kako najbolje strukturirati podatke, da bodo razumljivo, hitro in brez preveč podvojitve dosegljivi trenutni in historični podatki. Pri tem si je že precej pametnih ljudi zamislilo pametne rešitve za učinkovito hrambo podatkov, in sicer različne oblike SCD (angl. Slowly Changing Dimension) oziroma po slovensko, počasi se spreminjajoče dimenzije. Pri tem vprašanju sem na prvi pogled pričakoval, da bo najidealnejši odgovor SCD 6, hipoteza H3, kjer se ob vsaki spremembi ustvari nova vrstica z veljavnostjo, hkrati tudi vse historične vrstice za ta zapis hranijo glavne trenutno aktualne podatke. Vendar sem ob temeljitem pregledu podatkov ugotovil, da je ta način manj idealen za vnos podatkov, saj je ob vsaki spremembi treba vse stare vrstice za neki podatek spremeniti, medtem ko je za branje podatkov v veliki večini primerov ekvivalenten SCD 2, ki mu je podoben, le da v vrstici ne vključuje podatkov o trenutnem stanju.

Ker SCD 6 predstavlja dodatno kompleksnost in je boljši pri vpogledih v podatke le v redkih primerih, sem se na podlagi primerov, ki sem jih spisal, odločil, da je SCD 2 boljši.

Za zaključek sem želel preiskati še, kakšna je prihodnost podatkovnih baz. To mi je bilo v posebnem interesu, saj sem si izoblikoval kariero v relacijskih podatkovnih bazah, tako da sem želel izvedeti, ali bo to znanje v prihodnosti še relevantno, ali bo kakšna drugačna baza pomembnejša in se bom moral naučiti drugih veščin. Na to temo sem imel pričakovanje, da relacijske baze ne bodo šle nikamor, kar je bila hipoteza H4. Sicer se trg širi z novimi bazami, kot so No-SQL, vektorske, Time-series, vse te baze služijo svojim namenom in niso nadomestek za relacijske podatkovne baze. Sicer se bo pametno naučiti še te nove podatkovne baze in kako jih najbolje uporabiti, vendar bodo relacijske baze obdržale svojo uporabnost.

9 SEZNAM UPORABLJENIH VIROV

Abolo, S. (19. Januar 2024). *Will NoSQL Databases Replace SQL Databases in the future*. Pridobljeno iz DEV: https://dev.to/tecnosam/will-nosql-databases-replace-sql-databases-in-the-future-3hoa

All about Data Engineering. (23. Februar 2024). *Azure SQL database Vs Azure Synapse SQL Dedicated Pool*. Pridobljeno iz https://medium.com/@Lijitha/azure-sql-database-vs-azure-synapse-sql-dedicated-pool-4531c68a1722

Amazon Web Services. (17. Februar 2024). *Amazon Kinesis Documentation*. Pridobljeno iz https://docs.aws.amazon.com/kinesis/

Amazon Web services. (17. Februar 2024). *Amazon Managed Streaming for Apache Kafka Documentation*. Pridobljeno iz https://docs.aws.amazon.com/msk/

Apache Foundation. (22. April 2024). *Apache Foundation*. Pridobljeno iz https://www.apache.org/

Azure. (23. Februar 2024). *Azure Cache for Redis*. Pridobljeno iz https://azure.microsoft.com/en-us/products/cache/

Azure. (23. Februar 2024). *Azure Cosmos DB*. Pridobljeno iz https://azure.microsoft.com/en-us/products/cosmos-db/

Azure. (23. Februar 2024). *Azure Database for MariaDB*. Pridobljeno iz https://azure.microsoft.com/en-us/products/mariadb/

Azure. (23. Februar 2024). *Azure Database for MySQL*. Pridobljeno iz https://azure.microsoft.com/en-us/products/mysql/

Azure. (23. Februar 2024). *Azure Database for PostgreSQL*. Pridobljeno iz https://azure.microsoft.com/en-us/products/postgresql/

Azure. (17. Februar 2024). *Azure Event Hubs – A real-time data streaming platform with native Apache Kafka support*. Pridobljeno iz https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about#event-hubs-for-apache-kafka

Azure. (24. Februar 2024). *Azure SQL / SQL Server Change Stream with Debezium*. Pridobljeno iz Github: https://github.com/azure-samples/azure-sql-db-change-stream-debezium/tree/master/

Azure. (25. Februar 2024). *Develop Azure Functions by using Visual Studio Code*. Pridobljeno iz https://learn.microsoft.com/en-us/azure/azure-functions/functions-develop-vs-code?tabs=node-v3%2Cpython-v2%2Cisolated-process&pivots=programming-language-csharp

Azure. (24. Februar 2024). *Use Visual Studio Code to connect and query Azure SQL Database or Azure SQL Managed Instance*. Pridobljeno iz https://learn.microsoft.com/en-us/azure/azure-sql/database/connect-query-vscode?view=azuresql

Big data. (4. Maj 2024). Pridobljeno iz https://en.wikipedia.org/wiki/Big_data

Debezium. (25. Februar 2024). *Debezium documentation*. Pridobljeno iz https://debezium.io/documentation/reference/stable/index.html

Finkel, B. (19. Januar 2024). *How nosql databases work*. Pridobljeno iz Youtube: https://youtu.be/hYWDivcffQ0?si=VTI5vfw6YJBTqCxp

Google Cloud. (17. Februar 2024). *Pub/Sub documentation*. Pridobljeno iz https://cloud.google.com/pubsub/docs

R, M. M. (23. Februar 2024). *Choosing your Data Warehouse on Azure*. Pridobljeno iz https://www.clearpeaks.com/choosing-your-data-warehouse-on-azure-synapse-dedicated-sql-pool-vs-synapse-serverless-sql-pool-vs-azure-sql-database-part-2/

Ross, M., & Kimball, R. (2013). *The Data Warehouse toolkit the Definitive Guide to Dimensional Modeling.* Wiley.

Schott, M. (15. Marec 2024). *Slowly Changing Dimensions: What they are and why they matter*. Pridobljeno iz ThoughtSpot: https://www.thoughtspot.com/data-trends/data-modeling/slowly-changing-dimensions-in-data-warehouse

Shenoy, S. (15. Marec 2024). *Slowly Changing Dimensions: 5 Key Types and Examples*. Pridobljeno iz Hevodata: https://hevodata.com/learn/slowly-changing-dimensions/

Slowly changing dimension. (15. Marec 2024). Pridobljeno iz Wikipedia: https://en.wikipedia.org/wiki/Slowly_changing_dimension

Statista. (16. Marec 2024). *Market share of leading cloud infrastructure service providers*. Pridobljeno iz Statista: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers

Which are the most popular web servers? (23. April 2024). Pridobljeno iz https://www.stackscale.com/blog/top-web-servers/

Whiteley, S. (15. Marec 2024). *Introduction to Slowly Changing Dimensions (SCD) Types*. Pridobljeno iz Adatis: https://adatis.co.uk/introduction-to-slowly-changing-dimensions-scd-types/

10 PRILOGE

Priloga 1: Prepare_data.py

Priloga 2: DML_procedures.sql

Priloga 3: debezium_setup.txt

Priloga 4: debezium_message.json

Priloga 5: function_app.py

Priloga 6: Primerjava_poizvedb.xlsx