ON THE IMPEDANCE MISMATCH OF VECTOR DATA

VECTOR DATABASES Vs. RELATIONAL DATABASES

Autors
Pablo Aparici Villalta
Yasser El Mansouri Abdoun

CBDE 2425-QT

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Índex

PostgreSQL

PostgreSQL és un sistema de gestió de bases de dades relacional, que es conforma amb els estàndards SQL. Permet gestionar grans volums de dades de manera eficient, depenent dels requeriments del projecte, i ofereix funcionalitats avançades com transaccions ACID i indexació. La seva principal motivació és proporcionar una solució robusta i flexible que s'adapti a diverses necessitats.

No obstant això, al ser un gestor de bases de dades relacionals, en el context de representar línies de text com vectors de floats, pot sorgir un *impedance mismatch* entre el model de dades relacional de PostgreSQL i les estructures de dades utilitzades en el processament de textos, requerint estratègies específiques per a una integració eficient.

P0

En el primer script, s'ha insertat en una taula anomenada *sentences* a la base de dades de PostgreSQL 10.000 frases d'un fitxer de text, anomenat *sentences.txt*.

En la creació de la taula de la base de dades, es va decidir utilitzar el tipus de dades *FLOAT[]* per a la columna *embedding, i el tipus de dades TEXT per a la* columna *sentence* en la taula *sentences*. Aquesta decisió permet emmagatzemar vectors de floats i strings directament a PostgreSQL, eliminant la necessitat de convertir dades entre l'aplicació i la base de dades. Això millora el rendiment en les operacions de lectura i escriptura.

S'ha implementat la funció *insert_sentences* per inserir les sentències en lots. Utilitzar *cursor.executemany*, oferida per PostgreSQL, amb una mida de lot (*BATCH_SIZE*) redueix significativament el nombre de crides a la base de dades, disminuint l'overhead associat a cada operació individual. Aquesta operació millora el rendiment en les operacions d'inserció massiva i redueix el nombre de línies de codi necessàries.

A més, la funció *load_sentences* utilitza un generador (*yield*) per carregar les sentències del fitxer de text de manera eficient, evitant carregar tot el fitxer a la memòria simultàniament.

P1

En el segon script, s'ha transformat en embeddings, utilitzant el model ***all-MiniLM-L6-v2,*** les frases de la taula *sentences* i actualitzat cadascuna de les instàncies de la taula amb el seu embedding corresponent.

La funció *extract_sentences* s'ha dissenyat per extreure les sentències de la base de dades de manera eficient mitjançant l'ús de *cursor.fetchmany(batch_size)*. Aquesta estratègia de processament per lots permet manejar grans volums de dades dividint-les en grups, cosa que redueix significativament el nombre de crides a la base de dades i minimitza l'overhead associat a operacions repetitives. S'ha tornat a fer l'ús d'un generador *yield* per la gestió eficient de la memoria.

La integració de la biblioteca *sentence_transformers* per generar els embeddings de les sentències redueix l'*impedance mismatch*. Utilitzant la funció *transform_sentences_embeddings*, els textos es transformen en vectors de floats directament a l'aplicació, eliminant la necessitat de conversions de dades entre l'aplicació i la base de dades. Aquesta integració permet emmagatzemar els embeddings com a vectors de floats (*FLOAT[])* en PostgreSQL, permeten una correspondència directa entre les estructures de dades de l'aplicació i la base de dades, i millorant el rendiment en les operacions de lectura i escriptura.

La funció *update_with_embeddings* optimitza la inserció dels embeddings a la base de dades mitjançant l'ús de *cursor.executemany*, que permet actualitzar múltiples instàncies en una sola operació SQL. Això redueix dràsticament el nombre de crides a la base de dades, millorant el rendiment global i minimitzant l'*impedance mismatch*.

P2

En el tercer script, s'ha calculat les top-2 frases més similars de les deu escollides en el fitxer *sentences.txt.* Amb la possibilitat de calcular la similaritat a partir de la similaritat *cosinus* i la *L2 quadrada*.

La funció *get_database_sentences_embeddings* implementa una estratègia de processament per lots mitjançant l'ús de *cursor.fetchmany(batch_size)*. Això permet extreure les sentències de la base de dades en grups, cosa que redueix el nombre de crides a la base de dades i minimitza l'overhead associat a operacions repetitives. L'ús de generadors amb *yield* assegura una gestió eficient de la memòria.

L'ús de la biblioteca *numpy* per al càlcul de similituds millora l'eficiència i el rendiment del programa. Les funcions *cosine_similarity* i *L2_squared_distance* utilitzen operacions vectorials de *numpy* per calcular ràpidament les similituds entre els vectors de floats. Això, permet realitzar càlculs complexos de manera eficient. Redueix l'*impedance mismatch* entre l'aplicació i la base de dades, ja que les operacions de càlcul es realitzen de forma optimitzada, millorant el rendiment general i reduint el nombre de línies de codi necessàries per gestionar aquests càlculs.

La funció *get_top_2_similar_sentences* selecciona les dues sentències més similars per a cada sentència de prova. Aquesta funció implementa una estratègia de processament per lots utilitzant generadors amb *yield* a la funció *get_database_sentences_embeddings*, que recupera les sentències de la base de dades en grups. Això redueix el nombre de crides a la base de dades i minimitza l'overhead associat a operacions repetitives, millorant així el rendiment. A més, l'ús de diccionaris per mantenir les dues sentències més similars per a cada sentència de prova permet una gestió eficient dels resultats.

PQ1

S'ha utilitzat la llibreria *time* de Python per mesurar el rendiment de l'aplicació segons els aspectes de les taules. A l'utilitzar l'estratègia de lots s'han fet els experiments segons el número de frases en un lot.

Lot	Emmagatzemar dades textuals	Emmagatzemar embeddings	Computar similitut amb cosinus	Computar similitut amb L2 quadrada
200	0,0005115	0,002216	1,04521	0,5684
400	0,00005333	0,002460	0,9846	0,5731
600	0,00005103	0,00238	0,9996	0,5621
800	0,0001785	0,000227	1,00125	0,5652
1000	0,00004885	0,002450	1,00771	0,5682

Taula 1. Taula de temps d'execució (s) normalitzat *per la inserció de frases i embeddings, a PostgreSQL per a diferents lots i distàncies*

Els dos primer aspectes s'han normalitzat dividint el temps entre el número de frases de cada lot. Els dos següents, dividint el temps entre el número de frases a trobar similitud.

	Mínim	Màxim	Desviació estàndard	Mitjana	Coeficient de variació
Emmagatzemar dades textuals	0,0000489	0,0005115	0,0001995	0,0001686	118,27%
Emmagatzemar embeddings	0,000227	0,002460	0,0009665	0,001947	49,65%
Computar similitut amb cosinus	0,9846	1,04521	0,02263	1,00766	2,25%
Computar similitut amb L2 quadrada	0,5621	0,5731	0,004098	0,5674	0,72%

Taula 2. Taula de les mètriques normalitzades (s) *per la inserció de frases i embeddings, a PostgreSQL per a diferents lots i distàncies*

L'anàlisi del rendiment en funció de la mida del lot en la Taula 2, ha demostrat que optar per mides de lot més grans pot millorar significativament l'eficiència en l'emmagatzematge de dades textuals i embeddings. En el cas d'emmagatzematge de dades textuals, el punt òptim es de 1000 frases per lot, i en l'emmagatzematge de embeddings, el punt òptim es de 800 frases per lot. El coeficient de variació indica com varien els resultats, en aquest cas el temps d'inserció de text i embeddings no son estables. Això, pot ser atribuït a factors com la càrrega de la base de dades, les limitacions de recursos de la màquina, la gestió de memoria quan hi han moltes frases carregades, el número de conexions a la base de dades i el temps per connectar-se a ella.

D'altra banda, les operacions de càlcul de similituds, tant amb cosinus com amb L2 quadrada, han demostrat una alta estabilitat, suggerint que aquestes funcions poden treballar eficientment amb diferents mides de lot.

La diferència en el temps de càlcul entre les mètriques de similaritat de cosinus i L2 quadrada és atribuïda per la complexitat computacional que requereix cadascuna. La similaritat de cosinus requereix de més operacions de càlcul que la L2 quadrada provocant, l'augment de temps de computació.

En relació a l'optimització del rendiment, ja s'han aplicat tècniques de millora com el processament de lots, l'ús de generadors i l'ús d'operacions oferides per la llibreria de PostgreSQL per fer moltes operacions d'escriptura i lectura en una sola connexió.

Chroma

ChromaDB és una base de dades dissenyada específicament per gestionar i cercar dades vectorials. Això vol dir que està optimitzada per treballar amb representacions matemàtiques de dades, com ara les que s'utilitzen en el món del *machine learning* i el processament del llenguatge natural. Permet emmagatzemar i recuperar informació de manera eficient, facilitant l'ús d'algoritmes d'aprenentatge automàtic.

La seva motivació rau en la necessitat creixent de gestionar grans quantitats de dades no estructurades i complexes, més específicament, vectors. En un món on les aplicacions d'intel·ligència artificial i d'anàlisi de dades són cada cop més comunes, ChromaDB ofereix una solució potent que ajuda a optimitzar el procés de cerca i a millorar la precisió dels models, els quals, es nodreixen d'aquest tipus de dades.

Per tant, experimentarem quin és el rendiment que es pot aconseguir amb aquest sistema, més específicament, amb les frases que hem decidit treballar.

C0

En el primer *script* d'aquest apartat *C0.py*, també s'ha mesurat quan es triga a carregar les mateixes deu mil frases que s'han fet servir en el bloc anterior. En aquest cas, hem treballat amb el client de Chroma, per tal de crear una base de dades, sigui volàtil per a l'experimentació o persistent per tal de fer servir les dades en els altres apartats. Per crear el sistema de persistència, hem de triar quin tipus de mètriques fem servir, afegint-ho com a metadades. Aquest fet ens ha forçat a generar dues col·leccions (el que seria una taula per a Chroma), una per cada mètrica, doncs, les dades es guarden segons aquestes, ja que fa d'indexació.

S'ha implementat la funció *insert_senteces_in_segments* per inserir les frases en lots. Chorma ens ofereix un mètode que permet inserir d'una fins a un nombre determinat de frases de cop, sempre que no s'excedeix un límit que té el mateix sistema. Això ajuda a minimitzar el overhead que es podria donar per inserir les files d'una en una. El que s'ha de tenir en compte és que per afegir una nova frase, hem d'assignar-hi un identificador en format string. A més, la col·lecció emmagatzema les frases en forma vectorial, gràcies als *embeddings*, siguin els per defecte o els que li passin en crear-la des d'un inici, cosa que permet reduir l'*impedance mismatch*, de les frases. Aquest fet, més l'afegit de la mètrica escollida, produeix que les frases s'acabin ordenant segons aquesta.

C1

Pel que fa al segon *script*, *C1.py*, s'han convertit les frases prèvies a *embeddings* i s'han afegit a la col·lecció, però com ja hem dit, ChromaDB ja guarda les frases d'aquesta forma un cop s'afegeixen, i, com de base fa servir el mateix model que hem utilitzat ***all-MiniLM-L6-v2***, és com si sobreescrivim la col·lecció amb el mateix contingut.

Per extraure les frases les hem demanat totes, però el mètode get, permet obtenir-les mitjançant un offset, fent que es vagin passant en blocs per si es vol tractar en lots. Per altra banda, per actualitzar-les, hem d'indicar, necessàriament, els seus identificadors. A més, hem anat processant les dades de forma agrupada, com hem fet en la inserció.

C2

Per últim, en el tercer *script* d'aquest bloc, *C2.py*, s'ha computat les dues frases més semblants per a deu frases concretes, prèviament escollides, tant amb la métrica de similaritat cosinus i la *L2 squared*.

Chroma ens permet fer una consulta a la col·lecció d'una frase concreta, però, el que fa, no és portar aquesta frase, sinó les n frases que més s'hi assemblen, i, com permet fer-ho per a un nombre determinat de frases, amb una sola consulta, es poden portar a l'aplicació les deu frases amb les dues frases que més se'ls hi assemblen. Com les frases a consultar es troben dins del conjunt de sentències sencer, sabem que la sentència que més s'assemblen seran les mateixes que consultem, per tant, hem de demanar les tres frases més semblants, però excloent la primera.

CQ1

S'ha utilitzat la llibreria *time* de Python per mesurar el rendiment de l'aplicació segons els aspectes de les taules. A l'utilitzar l'estratègia de lots s'han fet els experiments segons el número de frases en un lot.

	Emmagatzemar dades textuals		Emmagatzemar embeddings	
Lot	Similaritat Cosinus	*Squared L2*	Similaritat Cosinus	*Squared L2*
200	0,0263	0,0268	0,000450	0,00049
400	0,0264	0,0261	0,000453	0,0005
600	0,0246	0,0248	0,000496	0,00052
800	0,024	0,0266	0,000478	0,00053
1000	0,0265	0,0244	0,000477	0,00051

Taula 3*: Temps d'execució (s) normalitzat, per la inserció de frases i embeddings, a ChromaDB per a diferents lots i distàncies.*

	Mínim	Màxim	Desviació estàndard	Mitjana	Coeficient de variació
Emmagatzemar dades textuals	0,024	0,02683	0,00106	0,02566	4,12%
Emmagatzemar embeddings	0,00045	0,00053	0,00003	0,00049	5,32%
Computar similitut amb cosinus	0,19	0,3735	0,0742	0,24252	30,58%
Computar similitut amb L2 quadrada	0,1896	0,2245	0,0145	0,20658	7,04%

Taula 4*: Temps d'execució (s) normalitzat, per la inserció de frases i embeddings, a ChromaDB per a diferents lots i distàncies.*

Com es veu en la Taula 3, els temps d'execució per la inserció d'una sentència, és bastant estable, independentment de la mida del lot, a més, es triga de forma molt similar en ambdues distàncies.

Pel que fa a l'emmagatzematge del embeddings, succeeix el mateix, però amb dues diferències: la primera, els temps de la similaritat cosinus és menor en tots els casos, cosa que no passa amb les frases, on cada mètrica s'alternen el lloc de la més ràpida, però com la diferència no és molt gran, pot ser degut a l'indexació o als recursos que es fan servir. La segona diferencia, es que es triga molt menys en la inserció des embedding que no pas les dades textuals, doncs, quan afegim les frases, han de passar pel procés de conversió a embeddings i s'han d'indexar, mentres que en l'altre cas, s'afegeixen directament on toquen segons l'identificador de sentencia.

Pel que fa al càlcul de les similituds entre frases, veiem com per a cinc iteracions, el càlcul és estable, donat que a la Taula 4, s'observa com el coeficient de variació per aquest, es bastant raonable, com passa en els apartats anteriors. També es veu com per la similitud cosinus, es triga, en mitjana, més que amb la L2 quadrada, cosa que pot ser degut a que l'operació de similitud cosinus és més complexa.

A grans trets, la millor forma d'optimitzar les insercions és el que fem, agrupen els valors en lots, per tal de reduir l'*impedance mismatch*. Alhora, quan es busca les frases més similars, hem de procurar passar les que volem fer servir de cop, doncs, Chroma ja s'encarrega de trobar aquest tipus de consulta. Cal dir, que s'ha d'experimentar per trobar el nivell òptim d'agrupació. A més, si fem ús de metadades, podem facilitar la recuperació de dades rellevants, o models més entrenats.

Pgvector

Pgvector és una extensió per a PostgreSQL que permet emmagatzemar i gestionar vectors de flotants de manera eficient, ideal per a aplicacions d'aprenentatge automàtic i processament de dades. Aquesta extensió facilita la representació de línies de text com a vectors, cosa que és especialment útil per a tasques com la cerca semàntica i la classificació.

Pgvector integra perfectament les capacitats de PostgreSQL amb les necessitats de manipulació de vectors, oferint funcionalitats d'indexació que milloren el rendiment de les consultes. Això permet gestionar grans volums de dades i operar de manera eficient amb representacions vectorials en un entorn relacional.

G0

En el primer script, s'ha insertat en una taula anomenada *sentences* a la base de dades de PostgreSQL 10.000 frases d'un fitxer de text, anomenat *sentences.txt*.

En la creació de la taula, s'ha seguit els mateixos criteris pel tipus de dades, exceptuant la columna dels embeddings, la qual, s'ha fet servir el tipus vector, propi de l'extensió que fem servir, per tal d'aprofitar el màxim els seus beneficis.

Pel que fa als mètodes, es fan servir els mateixos que amb Postgres, donat que l'únic que canvia, és el nou tipus de dades.

G1

En el segon script, s'ha transformat en embeddings, utilitzant el model **all-MiniLM-L6-v2**, les frases de la taula sentences i actualitzat cadascuna de les instàncies de la taula amb el seu embedding corresponent.

Un altre cop, els mètodes utilitzats, són els propis que s'ha implementat a la primera secció.

G2

En el tercer script, s'ha calculat les top-2 frases més similars de les deu escollides en el fitxer sentences.txt. Amb la possibilitat de calcular la similaritat a partir de la similaritat cosinus i la L2 quadrada.

En aquest cas, no ha calgut la implementació del càlcul de les distàncies, donat que la mateixa extensió inclou una condició a la clàusula ORDER BY on pots especificar el tipus de distància pel qual vols que t'ordeni les frases.

GQ1

S'ha utilitzat la llibreria *time* de Python per mesurar el rendiment de l'aplicació segons els aspectes de les taules. A l'utilitzar l'estratègia de lots s'han fet els experiments segons el número de frases en un lot.

Lot	Emmagatzemar dades textuals	Emmagatzemar embeddings
200	0,0000460	0,001244
400	0,00004400	0,001269
600	0,00004400	0,001186
800	0,0000401	0,001169
1000	0,00004172	0,001182

Taula 5*: Temps d'execució (s) normalitzat, per la inserció de frases i embeddings, a Pgvector per a diferents lots i distàncies.*

	Mínim	Màxim	Desviació estàndard	Mitjana	Coeficient de variació
Emmagatzemar dades textuals	0,0000401	0,0012690	0,0000023	0,0000432	5,24%
Emmagatzemar embeddings	0,00004013	0,00126900	0,0000441	0,001210	3,64%
Computar similitut amb cosinus	0,053	0,057	0,00165	0,05558	2,97%
Computar similitut amb L2 quadrada	0,055	0,0584	0,001392	0,0569	2,45%

Taula 6*: Temps d'execució (s) normalitzat, per la inserció de frases i embeddings, a Pgvector per a diferents lots i distàncies.*

En aquest cas, també s'observa com afegir frases i els embeddings és bastant estable, igual que computar les dues frases més semblants a cada una de les sentències que hem tirat, fet que els coeficients de variació són bastant reduïts indicant, gran estabilitat. També observem com no hi ha una diferència significativa entre obtenir les frases amb una mètrica o un altre.

Al igual que PostgreSQL, ja s'han aplicat les millores d'optimització.

Discussió

Per diferenciar entre PostgreSQL i Chroma, hem de tenir en compte les bases de cadascun dels sistemes. PostgreSQL està dedicat a la gestió de bases de dades relacionals, mentre que Chroma està optimitzat per la gestió de vectors de manera eficient. Per tant, des del punt de vista de l'*impedance mismatch* en el context d'aquesta pràctica, la representació de frases com a vectors de dades fa que Chroma sigui més eficient en la gestió d'embeddings que PostgreSQL, ja que ho fa de manera nativa a partir d'indexs dedicats a aquesta tasca.

Utilitzar PostgreSQL és útil si es relacionen amb altres taules o s'apliquen altres funcionalitats com l'ús de transaccions complexes, etc. En canvi, Chroma gestiona de manera nativa la similitud entre vectors, permeten que sigui útil en contextos com

l'aprenentatge de dades, la creació de models d'IA, etc., on trobar la similitud entre dades és essencial. A més, al ser natiu, la complexitat del codi i el nombre de línies es redueixen.

PostgreSQL no és recomanable si només s'utilitza per fer el que ja fa de forma nativa Chroma i de manera més eficient. A més, en aquesta pràctica s'ha fet una configuració i aplicació d'estratègies que el programador/a hauria d'aprendre, provocant un augment de la complexitat del codi i del nombre de línies per tal que sigui eficient el seu ús. Encara que Chroma sigui molt eficient, està limitada a ser utilitzada per a la similitud entre vectors de dades i no es podria utilitzar en altres contextos.

Amb relació a les taules obtingudes de les mètriques de cada sistema. PostgreSQL presenta que pot anar més ràpidament que Chroma en l'emmagatzematge de dades textuals, això es deu al fet que les frases que s'afegeixen en aquest últim, passen per un procés de conversió i d'indexació, a més, el primer sistema ho fa amb una alta variabilitat indicant, una gran inestabilitat comparant-la amb Chroma. Pel que fa a l'emmagatzematge d'*embeddings*, Chroma és més ràpida indicant una major eficiència en l'enregistrament de vectors. Finalment, Chroma és més ràpida en el càlcul de les similaritats que PostgreSQL perquè fa ús dels seus índexs personalitzats per aquest tipus de tasques encara que tingui més variabilitat, però, té més impacte el temps d'inserció de dades que l'obtenció de similaritats.

Quan parlem de Pgvector, ens trobem una intersecció entre PostgreSQL i ChromaDB. Això es deu al fet que permet treballar de forma anàloga a Postgre, però amb els avantatges, sobretot de consulta, que te Chroma. A l'hora d'inserir frases triga de forma similar a Postgre sense l'extensió, cosa que fa que sigui més ràpid que Chroma, a més, que té una velocitat a l'hora de fer consultes similars a Chroma, aconseguint el millor d'ambdós mons. Encara que sigui el millor, hem de tenir en compte que té les mateixes desavantatges que PostgreSQL en la corba d'aprenentatge i saber aplicar les estratègies adients.

Bibliografia

1. *MissterP. (s. f.). GitHub https://github.com/MissterP/LAB1_CBDE*

2. *MissterP. (s. f.). GitHub https://github.com/MissterP/LAB1_CBDE/blob/main/BookCorpus/sentences.txt*