Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

Informatikai Intézet

DIPLOMAMUNKA

Révész Gergő László

2024

Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

Informatikai Intézet

Elasticsearch alapú szabadszöveges keresőszolgáltatás Forrás rendszerekhez

Diplomamunka

Készítette:

Révész Gergő László

Programtervező informatikus
MSc hallgató

Témavezető:

Kaposvári Dániel

Csoportvezető
GriffSoft Informatikai Zrt.

Szeged

2024

Feladatkiírás

Egy szabadszöveges keresési szolgáltatás implementálása a Forrás rendszerbe. A Forrás egy modulokból – alrendszerekből – álló, SQL adatbázisra épülő integrált ügyviteli és bonyolítási rendszer. A kereső szolgáltatás általános – modulokon és alrendszereken átívelő – keresési lehetőséget biztosít tetszőlegesen meghatározott adatkörökre az ügyviteli egyedek metaadataiban. A különböző modulok, különböző ügyviteli területeket (pl. pénzügy, beszerzés) fednek le. A keresés célja a modulok szorosabb integrálása.

A diplomamunka célja a keresést, az adattranszformációt, valamint a kereső adatbázisba való áttöltést biztosító eszközök megismerése és megértése után a kereső szolgáltatás C# nyelven történő implementálása. A tervezés során fontos szempont a Forrás rendszerhez történő integrálhatóság. Továbbá kritérium, hogy a találati listából az adattranszformációt követően is visszaazonosíthatók maradjanak az adatforrásként szolgáló ügyviteli egyedek. Feladat még az elkészült program tesztelése, és összehasonlítása SQL Server Full-Text Search alapú megoldásával.

Tartalmi összefoglaló

- **Téma:**

  Elasticsearch alapú szabadszöveges keresőszolgáltatás Forrás rendszerekhez

- **Feladat:**

  Egy szabadszöveges keresőszolgáltatás megvalósítása Elasticsearch felhasználásával, illetve a keresőadatbázis naprakészen tartása az üzleti SQL adatbázisok szinkronizálásának implementálásával.

  Feladat továbbá egy felhasználói felület elkészítése, amely lehetővé teszi a különböző keresési algoritmusok használatát. Biztosítja a feltételeket kielégítő találatok megjelenítését és az azokon végzett további műveletek végrehajtását.

- **Használt technológiák:**

  Az Elasticsearch-csel való kommunikációt végző keresőszolgáltatás logikája NET Standard 2.0-t használva készült.

  A Microsoft SQL Server valamennyi verziójával kompatibilis adatszinkronizáló .NET 8-ban került implementálásra.

  A keresési feltételek beviteléért és a megjelenítésért felelős webes felhasználói felület Blazor keretrendszer segítségével valósult meg.

- **Megvalósítás:**

  Keresés végrehajtása az Elasticsearch-ben tárolt adatokon, majd az eredményként kapott találatok megjelenítése a felhasználói felületen. Szinkronizáció a különböző struktúrájú adatok egységesített alakjának Elasticsearch-ben történő aktualizálása céljából.

- **Eredmény:**

  Az elkészült alkalmazás rugalmasabbnak, skálázhatóbbnak és bővíthetőbbnek bizonyult, mint az SQL Server Full-Text Search funkciója.

- **Kulcsszavak:**

  Elasticsearch, keresés, adattranszformáció, adatszinkronizáció

Tartalomjegyzék

Bevezetés

A Forrás egy modulokból álló, SQL adatbázisra épülő integrált ügyviteli és bonyolítási rendszer. A Forrás használata közben számos alkalommal felmerül az igény a különböző területeket lefedő modulok szorosabb integrálhatóságára. A probléma áthidalására egy szöveges keresőszolgáltatás ötlete merült fel, amely modulokon átívelő keresést biztosít tetszőleges struktúrájú adatbázis táblák szöveges mezőiben.

Egy felhasználási eset lehet egy bizonylat rögzítése a Forrás beszerzési moduljába egy adott ügyintéző által. A beszerzéshez kapcsolódó pénzügyi bonyolítást a könyvelő végzi a pénzügyi modulban, egymás moduljainak struktúráját viszont nem ismerik. A modulokat szorosan integráló keresés a partner neve vagy a megjegyzés rovat alapján gondoskodik a bizonylat megtalálásáról.

Léteznek részintézményekből álló intézmények, amik megyénként vagy telephelyenként eltérő Forrás adatbázist használnak. Az adatbázisokon átívelő keresés segíthet az intézmény főkönyvelőjének egy bizonylat kikeresésében.

A Forrást használó intézmények többsége a pénzügyi év váltásánál adatbázist is vált, emiatt ezekben csak egy év adatai vannak jelen. Az adatbázisokon átívelő keresés hozzájárul az olyan bizonylatokra való rátalálásban is, amelyeknek rögzítési éve ismeretlen.

Az alkalmazás a .NET Standard 2.0-ás verziójának köszönhetően magasfokú kompatibilitást biztosít. .NET Framework és .NET (Core) rendszerekben egyaránt felhasználható, így könnyedén illeszkedik a Forrás régebbi és újabb komponenseihez is.

Az adatszinkronizációnak köszönhetően a keresés az Elasticsearch-ben tárolt naprakész állapotú adatokon végezhető. Kereséseinek sebessége megközelíti az SQL Server által biztosított Full-Text Search funkcióét.

Az alkalmazás rendelkezik egy felhasználói felülettel, amely biztosítja a különböző keresési algoritmusok használatát, a feltételeket kielégítő találatok megjelenítését, illetve az azokon végzett további szűrési és rendezési műveletek végrehajtását.

1. Alapfogalmak, használt technológiák

1.1. Szabadszöveges keresés

Szöveges keresőknek hívnak minden olyan szolgáltatást, amely lehetővé teszi a felhasználók számára, hogy megadott szavakból választva szűrést végezzenek egy vagy több adatforráson. A keresés eredményéről egy megjeleníthető lista készül a benne levő adatok valamely tulajdonsága alapján rendezve.

Például egy ruhákat kínáló web-áruházban egy ruhadarab mérete "kicsi", "közepes" vagy "nagy" lehet, így ezek a szavak használhatók a méret alapján történő szűrésére. Ebből akár többet is kiválasztva egy pontos egyezést vizsgáló algoritmus elvégzi a szűrést.

A szabadszöveges keresők esetében a felhasználók bármilyen szöveges kifejezést vagy mondatot megadhatnak, a szűrés nincs lekorlátozva előre megadott szavakra. Ebből kifolyólag ezek a keresők inkább hosszabb szövegben való keresésre alkalmasak, a találatokat pedig bonyolultabb algoritmusok határozzák meg.

Az előző web-áruházas példával élve a szabadszöveges keresők a szűrést a ruhák mérete helyett inkább olyan tulajdonságok alapján végzik, mint a ruhák leírása.

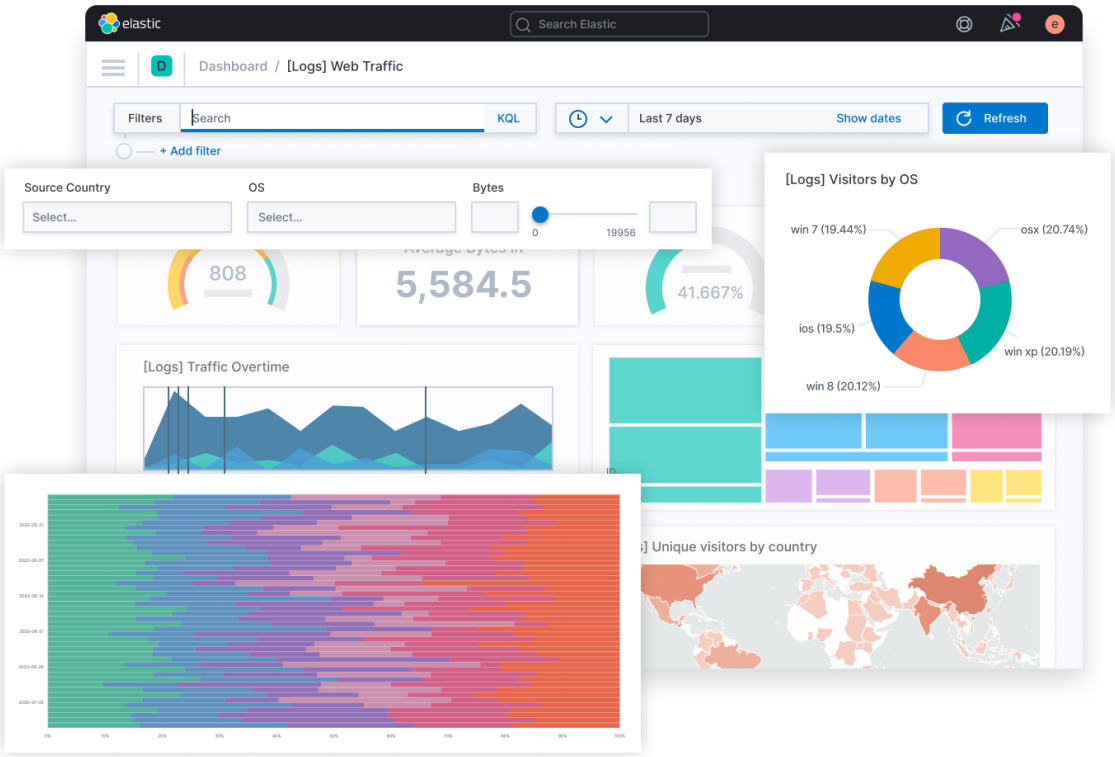
1.1.1. Elasticsearch

Az Elasticsearch [1] egy szoftver, amely többféle keresési módszert kínál. Nagy népszerűségnek örvend az olyan rendszerek körében, ahol a kereső fontos szerepet játszik. Ilyen rendszer például a Stack Overflow [2] vagy a Netflix [3], amik hatalmas mennyiségű kérést dolgoznak fel másodpercenként.

A siker titka az olyan gyors és hatékony keresési módszerekben rejlik, mint például Match boolean prefix [4]. A fordított index [5] mechanizmusnak köszönhetően közel valós időben képes futtatni a lekérdezéseket, amelyek a keresendő kifejezés mellett tartalmazhatnak aggregációt és rendezést is. Remekül skálázódik, mivel többszörös adatnövekedés mellett sem feltétlen várható különösebb lassulás. A szoftver az adatok biztonsági mentését automatikusan kezeli, így, ha a rendszer egy része fizikálisan meghibásodna, a jól működő rész képes pótolni a meghibásodottak hiányát.

A szoftver egy platformfüggetlen REST API-n [6] keresztül érhető el, emiatt beépíthető bármely olyan alkalmazásba, ami képes HTTP [7] kérések küldésére és válaszaik fogadására. A népszerűbb keretrendszerekhez és programozási nyelvekhez készült csomag is, ami osztályokon, típusokon és függvényeken keresztül megkönnyíti az Elasticsearch API-val való kommunikációt. Ilyen z Elasticsearch .NET Client [8], ami a hivatalos, Elastic által támogatott csomag .NET-hez [9].

A segítő csomagok mellett az Elastic fejleszt kiegészítő szoftvereket is. A Kibana [10] például az Elasticsearch-csel együtt használható felhasználói felület. A szoftver minden Elasticsearch API által nyújtott funkciót meghívására alkalmas, emellett biztosít számos olyan eszközt is, mint az adatvizualizáció vagy a dashboard-kezelés.



1.1. ábra: Kibana Dashboard, forrás: [11]

1.2. Adatszinkronizáció

Adatszinkronizációnak hívnak minden olyan folyamatot, amelyek során két vagy több adatforrásban tárolt adat összehangolódik. Ez lehetővé teszi, hogy különböző rendszerek ugyanazokkal az adatokkal tudjanak számolni még akkor is, ha azok különböző adatforrásból származnak.

Adatszinkronizáció segítségével, ha egy adatbázisban változás történik, a módosítás bizonyos idő után átkerül az érintett rendszerekbe is, fenntartva közöttük a konzisztenciát. Egy szinkronizáció sebessége számos tényezőtől függhet, többek között a folyamat futtatásának gyakoriságától, a rendelkezésre álló erőforrásoktól vagy attól, hogy egy vagy többirányú szinkronizációról van szó.

1.2. ábra: Adatszinkronizáció illusztráció, forrás: [12]

Egyirányú adatszinkronizáció esetén az adatok csak az egyik irányba mozognak, így elég csak a forrás rendszert monitorozni és szinkronizálni a változásokat a cél rendszerek felé. Például adatbázis biztonsági másolat készítésének egyik módja, ha időnként a forrás adatbázisban történt módosítások áttöltésre kerülnek a biztonsági mentések tárolására használt cél rendszerekbe is annak érdekében, hogy azok naprakész állapotba kerülhessenek.

Többirányú adatszinkronizáció esetén az adatok oda-vissza mozognak, emiatt nem elég csak az egyik rendszert monitorozni. Nyomon kell követni valamennyi adatbázis

módosulásait azért, hogy a változásokat szinkronizálni lehessen a megfelelő rendszerbe. Három rendszer esetén például bármely változásainak szinkronizálására szükség lehet a másik kettőben. A többirányú szinkronizáció megvalósítása egy bonyolultabb feladat, hiszen a helyes szinkronizálási sorrend mellett ügyelni kell az adatvesztés és a redundancia elkerülésére is.

1.2.1. Temporal Table

Az Temporal Table [13] egy beépített Microsoft SQL Server [14] adatbázis funkció. Segítségével lekérdezhetők az adatbázisban tárolt múltbeli és legfrissebb adatok egyaránt. Az adatmanipuláló műveletek végrehajtása során az adatbázis motor a háttérben automatikusan gondoskodik a történetiség aktualizálásáról és visszanézhetőségéről.

Működése az eredeti táblákon, a hozzájuk tartozó history táblákon és bennük két datatime2 típusú mezőn alapszik, amik a rekordok verzióit szolgálnak jelölni bármely időpillanatban. Az egyik mező az érvényesség kezdetét, a másik a végét jelöli. Legyen szó létrehozás, módosítás vagy törlés műveletről, az adott rekord bekerül, illetve frissül az eredeti táblában, a history táblában pedig létrejön egy új rekord az előző verzió lejárati dátumával aktualizálva.

A funkció a Microsoft SQL Server 2016-os verziójától érhető el. Használatához szükség van a system_versioning bekapcsolására a megfelelő táblákon. A táblákhoz lehetőség van hozzárendelni egyéni history táblát is, aminek hiányában a rendszer automatikusan létrehoz egy alapértelmezettet. Engedélyezés után a for_system_time kulcsszó és egy dátum intervallum megadásával lekérdezhetők a táblák intervallum alatt érvényes állapotai. A lekérdezés során a háttérben egy összefűzés és egy szűrés megy végbe, aminek eredmény halmaza az SQL lekérdezés műveletek segítségével tovább szűkíthető, illetve alakítható.

1.3. Felhasználói felület

Egy interaktív alkalmazásnak érdemes tartalmaznia grafikus felületet annak érdekében, hogy azt a különösebb technikai ismeretekkel nem rendelkező felhasználók is használni tudják. A felület amellett, hogy lehetővé teszi a felhasználók és az alkalmazás közti interakciót és kommunikációt, segít az adatok kezelésében, illetve megjelenítésében.

Egy jól megtervezett felhasználói felület hozzájárul a felhasználói élmény javításához. Használata egyszerű és intuitív, a félreértések és hibák lehetősége minimális, a vezérlése pedig legtöbbször egy egéren, egy billentyűzeten vagy egy érintőképernyőn keresztül történik.

1.3. ábra: Felhasználói felület illusztráció, forrás: [12]

1.3.1. Blazor

Számos keretrendszer és technológia létezik annak érdekében, hogy megkönnyítsék az interaktív, korszerű webalkalmazások elkészítését. Segítségükkel a webes böngészőkben futó alkalmazásokat már nem csak JavaScript nyelven van lehetőség fejleszteni. Gondoskodnak arról, hogy a webalkalmazások a JavaScript akár teljes mértékben való elkerülése mellett is gyorsan és biztonságosan tudjanak funkcionálni.

A Blazor [15] egy olyan webes keretrendszer, amivel a jól ismert HTML és CSS mellett C# és .NET technológiák segítségével teljes mértékben felépíthető egy webalkal-

mazás. A Blazor WebAssembly-nek köszönhetően a böngésző képes natív .NET kód futtatására is, növelve ezzel a teljesítményt és az interaktivitást.

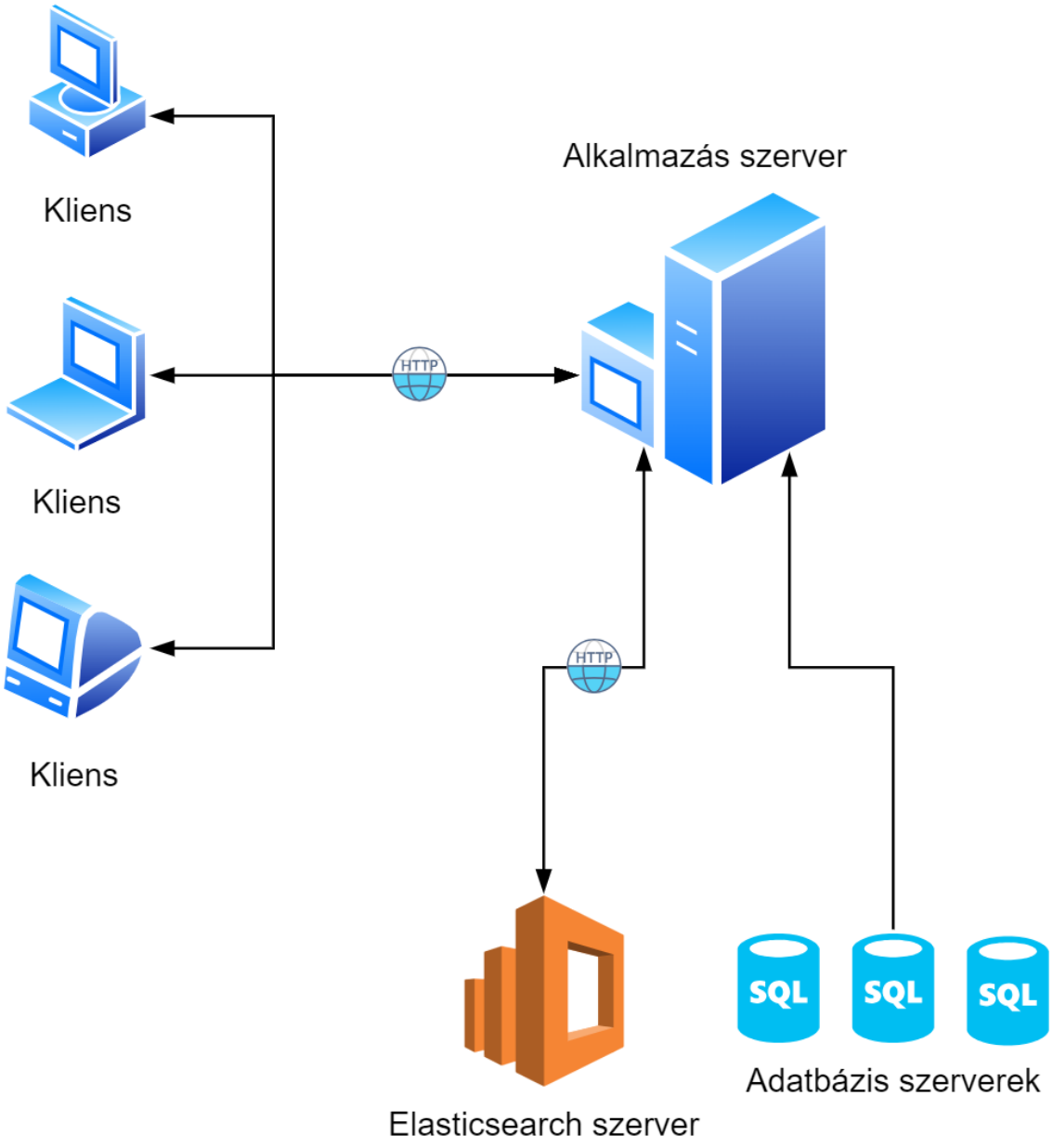
A Blazor két fő modellje közül az egyik a Blazor Server, ami kliens oldali kódot szerver oldalon futtatja, így nincs szükség annak böngészőbe való letöltésére. A DOM renderelést is a szerver végzi, a felhasználói interakciók és a DOM változtatások kommunikálása pedig SignalR kapcsolaton keresztül valósul meg.

A Blazor WebAssembly modell letölti a .NET runtime-ot a böngészőbe, ami WebAssembly segítségével kliensoldalon futtatja a C# kódot. Használatával a JavaScript teljesmértékben elkerülhető, a weboldal betöltése viszont lassabb és a böngésző képességei miatt az elérhető .NET funkciók listája szűkebb a Blazor Server-hez képest.

2. Tervezés

2.1. Áttekintés

Az alkalmazás egységes alakra hozza a különböző struktúrájú adatforrások rekordjait. Az Elasticsearch-ben való tárolás lehetővé teszi az adatokon való különböző algoritmusokkal végzett nagy teljesítményű kereséseket, szűréseket és rendezéseket. A módosult adatok ütemezetten szinkronizálásra kerülnek a rendszerbe az adatforrások legfrissebb állapotainak tükrözése érdekében.



2.1. ábra: Kommunikációs diagram

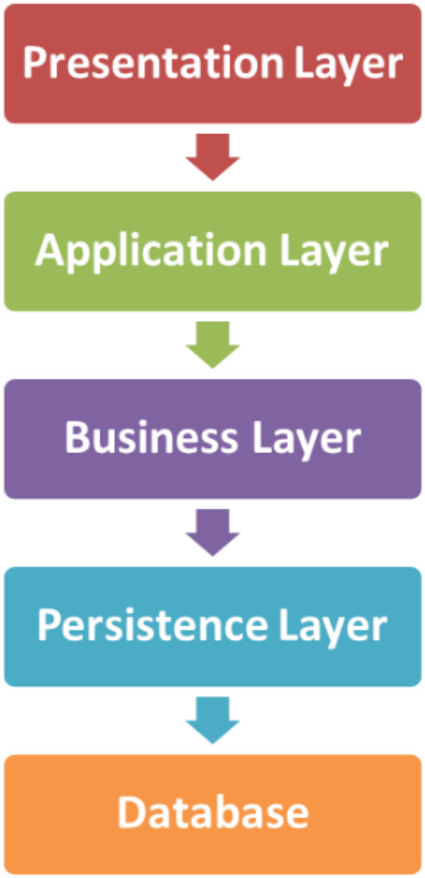
A rendszer használata az adatbázisok összekapcsolásával biztosítja a köztük levő integráltságot. Segít megkönnyíteni az adatok közti átláthatóságot, csökkentve ezzel a keresésekre fordítandó időt. Redukálja az információ megszerzéséhez szükséges lépések számát, ami hozzájárul a felhasználói élmény növeléséhez is.

A skálázható, rugalmas architektúrája lehetővé teszi a karbantarthatóságot és a további bővíthetőségeket. Lehetőséget nyújt az adatok hatékony elemzésére is ábrák, illetve diagramok készítésén keresztül.

2.2. Rétegelt architektúra

A rétegelt architektúra [16] egy olyan szoftverfejlesztési minta, amely az alkalmazás hasonló funkcionalitású komponenseinek rétegekre osztására helyezi a hangsúlyt. A hierarchikusan rendezett rétegek mindegyike specifikus feladatot lát el a rendszerben.

A minta használata lehetővé teszi az egyes rétegek lecserélhetőségét, független fejlesztését és tesztelését. A modularitás, a karbantarthatóság és a bővíthetőség növelése miatt használata elterjedt kisebb, illetve nagyobb projektek körében egyaránt.



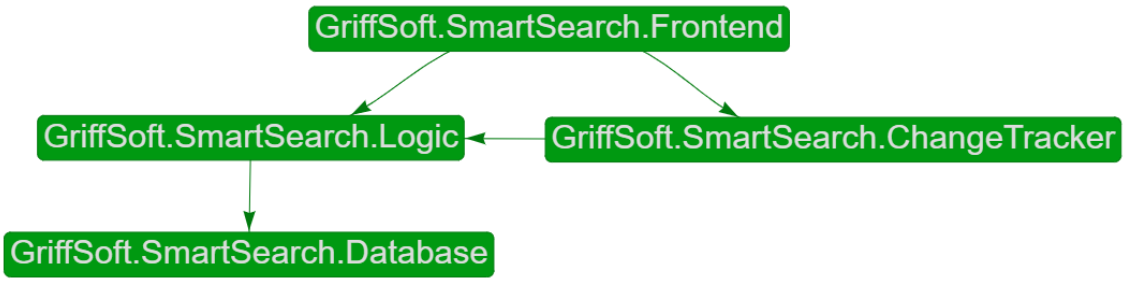
2.2. ábra: Rétegelt architektúra

Az alkalmazás a rétegelt architektúra mintának megfelelően négy rétegre lett bontva. A legalsó az adatbázis kezelő réteg, aminek már további függősége nincsen. Feladata az adatbázisokhoz való hozzáférés és az adataikon végezhető műveletek biztosítása, szükség esetén azok lecserélése anélkül, hogy ez bármilyen módon befolyásolná a tőle függő rétegek működését.

A hierarchiában szereplő második réteg az üzleti logika rétege. Itt helyezkednek el az üzleti folyamatok és szabályok megvalósításai, amik kifelé publikus osztályokon és interface-eken keresztül érhetők el. Működéséhez csak az adatbázis műveletekre van szüksége, emiatt egyetlen függősége az adatbázis kezelő réteg.

Legfelső réteg a felhasználói felület, amely az adatmegjelenítéssel és a felhasználói interakciókkal kapcsolatos funkciók kezelését végzi. Egyedüli függősége az üzleti logika, ezáltal könnyedén tud alkalmazkodni az alapfunkciókhoz.

A kvázi másik legfelsőnek számító réteg egy háttérszolgáltatás. Feladata az adatszinkronizálás végrehajtása megadott időközönként. Az akár önállóan is működni képes háttérszolgáltatást a könnyebb telepíthetőség és üzemeltethetőség érdekében a webszolgáltatás Dependency Injection [17] segítségével futtatja, ebből kifolyólag a réteg a felhasználói felület második függőségét képezi. A szinkronizálási folyamatot az üzleti logika tartalmazza, emiatt az függősége az ütemező rétegnek is.



2.3. ábra: Dependensee által generált függőség gráf

2.3. .NET verziók és kompatibilitás

Egy rendszer tervezése során jelentős szerepet játszik a modern, korszerű technológiák használata mellett a visszafelé kompatibilitás biztosítása is. Fontos mindkét tervezési szempont, azonban mindkettő általában egyszerre nem érhető el, hiszen az egyikre való fokozott törekvés a másik háttérbe szorítását eredményezi.

Az ilyen technológiai döntéseket fontos még a rendszer fejlesztésének megkezdése előtt meghozni, mert később egy ehhez hasonló alappillér megváltoztatása hatalmas költségekkel járhat. Amikor csak tehető, ajánlott a korszerűség felé venni az irányt, mert olykor egy verzióemelés is gyorsabb és könnyebben bővíthető alkalmazáshoz vezet.

Igény lehet egy alkalmazás más rendszerekbe való beintegrálására is. Ilyen esetben érdemes megvizsgálni azokat a verziókat, amikkel kompatibilisnek kell maradni és úgy megválasztani a keretrendszer verziót, hogy az mindegyiknek megfelelő legyen.

Az alkalmazás üzleti logika részének használatára .NET Framework és .NET Core projektekben egyaránt szükség lehet. A .NET Standard 2.0 verzió a legtöbb általános célú könyvtárat már tartalmazza és minden modern verzió támogatja, így az mindkét szükséges verzióval kompatibilis.

A visszafele kompatibilitás miatt az üzleti logika és az adatbázis kezelést végző projektek is .NET Standard 2.0 keretrendszert használnak. A felhasználói felületet biztosító webszolgáltatás és a szinkronizálást ütemező háttérszolgáltatás a rájuk mutató függőségek hiánya és a korszerűségre való törekvés miatt a jelenleg legújabb LTS (Long Term Support) .NET 8 keretrendszerben kerültek megvalósításra.

2.4. Konfiguráció

Egy alkalmazás fejlesztése során fontos szerepet játszik annak konfigurálhatósága. Minél konfigurálhatóbb egy rendszer, annál jobban lehet a forráskód módosítása nélkül az adott környezetre vagy ügyféligényre szabni.

Egy rendszer konfigurációját szokás külön helyen, tipikusan egy vagy több szöveges leírófájlban tárolni annak érdekében, hogy a program működése megváltoztatható legyen a forráskód átalakítása nélkül is. Ilyen leírásra szolgáló szabvány például a JSON [18], ami programozói tudás nélkül is könnyedén értelmezhető és szerkeszthető.

Az alkalmazás egy konfigurálható funkciója az adatforrások rekordjainak lekérdezése, azok Elasticsearch-be való áttöltésének, majd kereshetőségének céljából. Az összekötendő adatforrások az appsettings.json fájlban egy jól definiált JSON struktúrát követve állíthatók be.

Adatforrásnak számítanak az adatbázisok, a tábláik, mezőik és egyéb információinak együttes konfigurációja. Beállításuk biztosítja az adataikon átívelő keresés működését.

Az alkalmazás induláskor kiolvassa, majd betölti a bekonfigurált adatforrásokat és az egyéb, adatszinkronizálás megkezdéséhez szükséges beállításokat.

```
"ElasticsearchData": {
  "BatchSize": "<int>",
  "ElasticTargets": [
    {
      "ConnectionString": "<string>",
      "Server": "<string>",
      "Database": "<string>",
      "Tables": [
        {
          "Table": "<string>",
          "Type": "<int>",
          "Keys": [ "<string>" ],
          "Columns": [ "<string>" ]
        }
      ]
    }
  ]
}
```

3. Megvalósítás

3.1. Alapötlet

Az adatbázis táblák struktúrái általában nem egyeznek meg, ezért ezeket a keresés megvalósításához – a beazonosíthatóság megtartása mellet – egységes alakra kell hozni. Az ElasticDocument olyan adatstruktúra, amely bármely dokumentum esetén biztosítja forrásának egyértelmű visszakereshetőségét. Lehetővé tesz számos olyan továbbfejlesztési funkciót, mint a találatokra való kattintás, amely hatására a megfelelő, találathoz tartozó adatbázis rekord megjelenítésre kerülhet a felhasználó számára.

Az azonosító, a tábla, az adatbázisának és a szerver neve bármely rekord esetén egyedi dokumentumot alkot. E redundáns adattárolás jól skálázhatóságot és gyors keresést tesz lehetővé, hiszen az Elasticsearch kereséseinek sebessége nem az adatbázisában tárolt dokumentumok mennyiségétől, hanem azok szövegeiben szereplő szavak szórásától függ.

```
public class ElasticDocument
{
    public required string Server { get; init; }

    public required string Database { get; init; }

    public required string Table { get; init; }

    public required TableType Type { get; init; }

    public required Dictionary<string, object> Keys { get; init; }

    public required string Column { get; init; }

    public required string Value { get; init; }
}
```

A kereshető tartalmat a dokumentum oszlopának neve és a szöveges Value mezőjének együttese biztosítja. A több kulcsú táblákat is támogató Keys mezője az adott rekord azonosítóit tartalmazza egy szótár segítségével kulcs-érték párok formájában. A táblákhoz tartozik egy bekonfigurálható, magyarázatul szolgáló típus is. A dokumentum Type mezője a TableType enum egy értéke segítségével keresés során extra szűrhető információt nyújt.

Az ElasticDocument adatstruktúra az Elasticsearch adatbázisán kívül a megvalósításban szereplő legtöbb generikus művelet paraméterét és a megjelenítési felület táblázatának sorait is egyaránt képezi.

3.2. Előkészítés

Az Elasticsearch API-hoz való csatlakozás egy költséges művelet, amire a keresésnek és a szinkronizálásnak is szükség van. Annak érdekében, hogy csatlakozások száma minimalizálva legyen, az Elasticsearch Client biztosít egy Thread Safe megoldást arra, hogy ezt elég legyen csupán egyszer megtenni.

Az inicializáló lépéseket egy Provider [19] osztály végzi. Ez a Singleton ElasticsearchClientProvider biztosítja azt az egy klienst, amely bárhonnan elérhető és amin keresztül az Elasticsearch API funkciói meghívhatók.

A Singleton tervezési minta lehetővé teszi, hogy az alkalmazás élettartama alatt egy adott osztálynak mindössze egy objektum példánya létezhessen. A Provider-rel való együttes használata biztosítja a kliens egyszeri inicializálhatóságát.

A kliens az osztály egy Client nevű publikus mezőjén keresztül érhető el, amely az objektum létrejöttekor inicializálódik.

```
_initializationTask = InitializeAsync();

public Task<ElasticsearchClient> Client => GetClientAsync();

private async Task<ElasticsearchClient> GetClientAsync()
{
    await _initializationTask;
    return _elasticsearchClient;
}
```

Az inicializációs folyamat létrehozza a klienst, megpróbál kapcsolódni az Elasticsearch API-hoz, végül előkészíti az Indexet [20].

```
private async Task InitializeAsync()
{
    _elasticsearchClient = CreateClient();
    await EnsureAvailableAsync();
    await ReCreateIndexAsync();
}
```

Ha inicializáció közben bármelyik művelet sikertelenül érne véget, eldobásra kerül egy specifikus, hibát magyarázó kivétel, amely az Elasticsearch hibaüzenetével együtt naplózásra kerül.

```
if (!pingResponse.IsValidResponse)
{
    string reason = pingResponse.GetExceptionMessage();
    var serverUnavailableException = new ServerUnavailableException(reason);
    _logger.LogError(serverUnavailableException, "Elastic client is not working.");

    throw serverUnavailableException;
}
```

A folyamat befejezését követően a Task típusú kliens mező egy hagyományos szinkron módon működő getter-ként fog viselkedni.

```
var client = await _elasticsearchClientProvider.Client;
```

3.3. Keresés

3.3.1. Keresés művelet

A keresési kérelem az ISearchService interface SearchAsync függvénye fogadja és dolgozza fel. Segítségével a keresés megvalósítása teszteléskor könnyedén lecserélhető bármely másikra, így az SQL Full-Text Search-re [21] is.

```
public interface ISearchService<T> where T : class
{
    Task<SearchResult<T>> SearchAsync(SearchRequest searchRequest);
}
```

Az interface feladata a SearchRequest DTO (Data Transport Object) -ban található feltételeket tartalmazó kérelem kiszolgálása.

```
public class SearchRequest
{
    public List<SearchFilter> Filters { get; set; } = new List<SearchFilter>();

    public List<SearchOr> Ors { get; set; } = new List<SearchOr>();

    public List<SearchAnd> Ands { get; set; } = new List<SearchAnd>();

    public List<SearchSort> Sorts { get; set; } = new List<SearchSort>();

    public int Size { get; set; }

    public int Offset { get; set; }
}
public class SearchFilter
{
    public required FilterMatchType FilterMatchType { get; init; }

    public required string FieldName { get; init; }

    public required string FieldValue { get; init; }
}
```

A kereső eljárás a találatokat egy ElasticDocument-eket tartalmazó generikus SearchResult DTO-ba csomagolva juttatja vissza.

```
public class SearchResult<T>
{
    public required long TotalCount { get; init; }

    public required IReadOnlyCollection<T> Hits { get; init; }
}
```

Az ISearchService interface-t alapértelmezett megvalósítása az ElasticsearchService. A szervíz a találatok megkeresésének feladatát tovább delegálja az Elasticsearch-nek.

3.3.2. Transzformálás

Az Elasticsearch API egy megadott struktúrájú HTTP kérésben definiálva várja a keresési feltételeket. A SearchRequest-et a RequestApplicator osztály ApplyRequestOn eljárása transzformálja át ideális alakra.

```
var requestApplicator = new RequestApplicator(searchRequest);
var searchResponse = await elasticSearchClient
    .SearchAsync<ElasticDocument>(requestApplicator.ApplyRequestOn);
```

A SearchRequestDescriptor egy a számos kliens által létrehozott, keresés leírására szolgáló objektumból. Függvényeik segítségével felépíthető bármely olyan keresési kérelem, amely eleget tesz az Elasticsearch API szintaxisának. Az ApplyRequestOn ráilleszti a keresési feltételeket, a rendezést és a paginációhoz szükséges oldalméretet, illetve oldalszámot a paraméterül megkapott SearchRequestDescriptor-ra.

```
public void ApplyRequestOn(
    SearchRequestDescriptor<ElasticDocument> searchRequestDescriptor)
{
    searchRequestDescriptor
        .Query(ApplyQuery)
        .Sort(Sorts)
        .Size(_requestSize)
        .From(_requestOffset);
}
```

A keresési feltételeket a QueryApplicator osztály alakítja át. A szűrések, az és illetve a vagy feltételek QueryDescriptor-ra való felillesztését az ApplyQueryOn végzi.

```
_queryApplicator.ApplyQueryOn(queryDescriptor);
```

```
public void ApplyQueryOn(QueryDescriptor<ElasticDocument> queryDescriptor)
{
    queryDescriptor.Bool(ApplyBoolQuery);
}
```

Mindhárom feltétel típus transzformálása azonos sémára épül. Szűrések esetén a MultiFilterApplicator, ések esetén a MultiAndApplicator, vagyok esetén pedig a MultiOrApplicator osztály feladata az átalakítás.

```
_multiFilterApplicator.ApplyFiltersOn(boolQueryDescriptor);
_multiAndsApplicator.ApplyAndsOn(boolQueryDescriptor);
_multiOrApplicator.ApplyOrsOn(boolQueryDescriptor);
```

Az ApplyFiltersOn például ráilleszti az érvényes szűrések leíróit a paraméterül kapott BoolQueryDescriptor-ra.

```
public void ApplyFiltersOn(BoolQueryDescriptor<ElasticDocument> boolQueryDescriptor)
{
    var filters = Filters;
    if (!filters.Any())
    {
        return;
    }
```

```
    boolQueryDescriptor.Filter(filters);
}
```

A leírókat a MultiFilterDescriptor osztály készíti el.

```
private Action<QueryDescriptor<ElasticDocument>>[] Filters =>
    _multiFilterDescriptor.CreateFilterDescriptors();
```

A CreateFilterDescriptors létrehozza a transzformált szűréseket tartalmazó listát. Minden egyes szűrő leírójának létrehozásáért egy FilterApplicator felel.

```
public Action<QueryDescriptor<ElasticDocument>>[] CreateFilterDescriptors()
{
    var filterDescriptions = new List<Action<QueryDescriptor<ElasticDocument>>>();

    foreach (var searchFilter in validSearchFilters)
    {
        var filterApplicator = new FilterApplicator(searchFilter);
        filterDescriptions.Add(filterApplicator.ApplyFilterOn);
    }

    return filterDescriptions.ToArray();
}
```

Az ApplyFilterOn feladata a QueryDescriptor MatchApplicator-nak való átadása a szűrő tartalmának átalakítása érdekében. A SearchFilter FilterMatchType enum-ja alapján a ToFilterMatchApplicator átkonvertálja a szűrési feltételt egy absztrakt MatchApplicator-ra. A létrejött objektum ApplyMatchOn metódusa végzi a szűrő tartalmának átalakítását.

```
public void ApplyFilterOn(QueryDescriptor<ElasticDocument> queryDescriptor)
{
    var filterMatchApplicator = _filterMatchTypeConverter.ToFilterMatchApplicator();
    filterMatchApplicator.ApplyMatchOn(queryDescriptor);
}

public MatchApplicator ToFilterMatchApplicator() =>
    _searchFilter.FilterMatchType switch
    {
        FilterMatchType.BoolPrefix => new BoolPrefixMatchApplicator(
            _searchFilter.FieldName, _searchFilter.FieldValue),
        FilterMatchType.PhrasePrefix => new PhrasePrefixMatchApplicator(
            _searchFilter.FieldName, _searchFilter.FieldValue),
        FilterMatchType.Wildcard => new WildcardMatchApplicator(
            _searchFilter.FieldName + DefaultKeywordFieldPrefix,
                _searchFilter.FieldValue),
        FilterMatchType.Prefix => new PrefixMatchApplicator(
            _searchFilter.FieldName + DefaultKeywordFieldPrefix,
                _searchFilter.FieldValue),
        FilterMatchType.Term => new TermMatchApplicator(
            _searchFilter.FieldName + DefaultKeywordFieldPrefix,
                _searchFilter.FieldValue),
        _ => throw new Exception(
            $"No query mapping found for {_searchFilter.FilterMatchType}."),
    };
```

Az ApplyMatchOn felel a szűrést végző algoritmusnak, illetve a szűrendő mező nevének és értékének a paraméterül kapott QueryDescriptor-ra való felillesztéséért. BoolPrefixMatchApplicator esetében például az átkonvertált MatchApplicator felparaméterezi a MatchBoolPrefix keresést a megfelelő értékekkel. Az algoritmus Elasticsearch oldalon Match boolean prefix lekérdezést futtatva teljes, illetve szó eleji egyezéseket keres.

```
public override void ApplyMatchOn(QueryDescriptor<ElasticDocument> queryDescriptor)
{
    queryDescriptor
        .MatchBoolPrefix(p => p
            .Field(_fieldName)
            .Query(_fieldValue));
}
```

Az Elasticsearch elvégzi a keresést, végül a találatokat átalakításra kerülnek az ISearchService interface visszatérési értékének típusára.

```
    var searchResponse = await elasticSearchClient
        .SearchAsync<ElasticDocument>(requestApplicator.ApplyRequestOn);

    return searchResponse.ToSearchResult();
}

public static SearchResult<T> ToSearchResult<T>(
    this SearchResponse<T> searchResponse) where T : class
{
    return new SearchResult<T>
    {
        TotalCount = searchResponse.Total,
        Hits = searchResponse.Documents,
    };
}
```

3.4. Adatszinkronizálás

3.4.1. Ütemezés

Az adatszinkronizálás a CronSchedulerService ütemezi. A szolgáltatás az ICronSchedulerService interface megvalósítása, illetve a BackgroundService leszármazottja egyben.

Az előbbi tartalmaz egy NextOccurance mezőt, ami megmutatja következő futtatás idejét, továbbá rendelkezik egy RunJobsAsync függvénnyel, ami a szükséges feladatokat hívatott lefuttatni.

```
public interface ICronSchedulerService
{
    TimeSpan NextOccurance { get; }

    Task RunJobsAsync();
}
```

Az utóbbi egy hosszú ideig futó háttérszolgáltatás megvalósítása, amire az ütemező Dependency Injection Container-be HostedService-ként [22] való beregisztrálhatósága miatt van szükség. Deklarál egy felülírható ExecuteAsync metódust, ami az alkalmazás indítása során meghívásra kerül és akár annak leállásáig is futhat.

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
```

```
    while (!stoppingToken.IsCancellationRequested)
    {
        await RunJobsAsync();
        await Task.Delay(NextOccurance, stoppingToken);
    }
}
```

A következő futtatás időpontját egy bekonfigurálható Cron [23] kifejezés alapján a GetNextOccuranceAsync számolja ki. A futtatásra való várakozás aszinkron, a szálakat nem blokkoló módon történik.

```
"CronSchedulerOptions": {
  "Cron": "* * * * *"
}
```

```
public TimeSpan NextOccurance => GetNextOccurance();

private TimeSpan GetNextOccurance()
{
    var nextOccurance = _cronSchedulerOptions.CronExpression
        .GetNextOccurrence(DateTimeOffset.Now, TimeZoneInfo.Local);
    var differenceTimeSpan = nextOccurance?.Subtract(DateTimeOffset.Now)
        ?? TimeSpan.FromDays(1);

    return differenceTimeSpan;
}
```

3.4.2. Bejárás

A szinkronizálásért az ISynchronizerService felelős. Tartalmazza az utolsó szinkronizálás időpontját és az azt végző eljárást.

```
public interface ISynchronizerService
{
    DateTime LastSynchonizationDate { get; }

    Task SynchronizeAsync();
}
```

```
await _synchronizerService.SynchronizeAsync();
```

Az interface implementációja egyesével végig iterál a bekonfigurált adatforrásokon, csatlakozik hozzájuk és elvégzi az előző áttöltés óta módosult adatok szinkronizálását.

```
DateTime synchronizationDate = DateTime.UtcNow;

foreach (var elasticTarget in _elasticsearchData.ElasticTargets)
{
    using var sqlConnector = new SqlConnector(elasticTarget.ConnectionString);

    foreach (var elasticTable in elasticTarget.Tables)
    {
        await RunSynchronizationAsync(elasticSyncrhonizationDto);
    }
}

LastSynchonizationDate = synchronizationDate;

protected abstract Task RunSynchronizationAsync(
    ElasticSynchronizationDto elasticSynchronizationDto);
```

3.4.3. Csatlakozás

A csatlakozás az SqlConnector segítségével történik, ami a bekonfigurált ConnectionString-et [24] használva megpróbál kapcsolódni az adatbázishoz. Az Elasticsearch API-hoz hasonlóan a kapcsolódás a konstruktorban aszinkron módon indul, az eredmény pedig egy Task típusú mezőn keresztül érhető el.

```
_connectionOpenerTask = _sqlConnection.OpenAsync();

public Task<SqlConnection> Connection => GetConnectionAsync();

private async Task<SqlConnection> GetConnectionAsync()
{
    await _connectionOpenerTask;
    return _sqlConnection;
}
```

Sikeres csatlakozást követően először a létrehozandó, illetve módosítandó, majd pedig a törlendő adatok szinkronizálódnak.

```
protected override async Task RunSynchronizationAsync(
    ElasticSynchronizationDto elasticSynchronizationDto)
{
    var sqlBatchUpsertDataReader = new SqlBatchDataReader<ElasticDocument>(
        sqlBatchUpsertQueryFactory, elasticDocumentMapper);
    await sqlBatchUpsertDataReader.ProcessDataAsync(
        postProcessCallback: _elasticBulkOperationService.BulkUpsertAsync);

    var sqlBatchDeleteDataReader = new SqlBatchDataReader<ElasticDocument>(
        sqlBatchDeleteQueryFactory, elasticDocumentMapper);
    await sqlBatchDeleteDataReader.ProcessDataAsync(
        postProcessCallback: _elasticBulkOperationService.BulkDeleteAsync);
}
```

3.4.4. Feldolgozás

Az adatok az IDataReader dolgozza fel. A nagy szinkronizálandó adatmennyiség miatt kötegelt feldolgozásra van szükség, amit a ProcessDataAsync eljárás postProcessCallback paramétere tesz lehetővé. A BatchSize, vagyis a kötegek mérete konfigurálható, így azt érdemes a rendelkezésre álló erőforrásokhoz mérten meghatározni.

```
internal interface IDataReader<T> where T : class
{
    Task ProcessDataAsync(Func<List<T>, Task> postProcessCallback);
}

public async Task ProcessDataAsync(Func<List<T>, Task> postProcessCallback)
{
    bool hasUnprocessedData = true;

    while (hasUnprocessedData)
    {
        using var query = await _sqlBatchQueryFactory.CreateNextAsync();
        var data = await QueryDataAsync(query);
```

```
            await postProcessCallback(data);

            hasUnprocessedData = data.Any();
        }
    }
```

3.4.5. Kötegek létrehozása

A kötegek tartalmának előállításához szükséges adatbázis lekérdezéseket az IBatchQueryFactory készíti el. A CreateNextAsync az adatbázis táblák bekonfigurált kulcsai alapján rendezett rekordoknak csak a sorban következő kötegének lekérdezését hozza létre.

```
public interface IBatchQueryFactory<T> where T : DbCommand
{
    Task<T> CreateNextAsync();
}
```

```
public async Task<SqlCommand> CreateNextAsync()
{
    string paramaterisedSqlQuery = GetParamaterisedSqlQuery();
    var sqlConnection = await _sqlQueryDto.SqlConnector.Connection;
    var sqlCommand = new SqlCommand
    {
        Connection = sqlConnection,
        CommandText = paramaterisedSqlQuery,
    };

    _batchCount++;
    return sqlCommand;
}
```

A GetParamaterisedSqlQuery az eredeti és a history tábla segítségével olyan lekérdezéseket állít elő, amik kizárólag az előző szinkronizáció óta módosult rekordokat tartalmazzák.

```
private const string SqlQuery =
    "SELECT {0} " +
    "FROM ( " +
    "   SELECT {1}, ValidFrom, ValidTo, ROW_NUMBER() OVER ( " +
    "       PARTITION BY {2} " +
    "       ORDER BY ValidTo DESC " +
    "   ) RowNumber " +
    "   FROM {3} " +
    "   FOR SYSTEM_TIME " +
    "   BETWEEN {4} AND {5} " +
    "   WITH (NOLOCK)" +
    ") a " +
    "WHERE RowNumber = 1 " +
    "AND ValidFrom > {6} " +
    "AND ValidFrom <= {7} " +
    "AND ValidTo > {8} " +
    "ORDER BY {9} " +
    "OFFSET {10} ROWS " +
    "FETCH NEXT {11} ROWS ONLY;";

protected abstract string GetParamaterisedSqlQuery();
```

3.4.6. Áttöltés

A létrehozott lekérdezés végrehajtását követően a szinkronizáláshoz az egy kötegbe tartozó rekordok ElasticDocument-ekké mappelése szükséges.

```
private async Task<List<T>> QueryDataAsync(DbCommand query)
{
    using var dataReader = await query.ExecuteReaderAsync();
    var documents = await _dataMapper.MapAsync(dataReader);
    return documents;
}

public async Task<List<ElasticDocument>> MapAsync(DbDataReader dataReader)
{
    var documentCollection = new List<ElasticDocument>();

    while (await dataReader.ReadAsync())
    {
        var elasticDocuments = MapToDocument(dataReader);
        documentCollection.AddRange(elasticDocuments);
    }

    return documentCollection;
}
```

Mappelés során a beazonosíthatóság céljából minden dokumentum kap egy egyedi ID-t. Az azonosítót az MD5 algoritmus [25] számítja ki a dokumentum tartalmának JSON-né alakított formátuma alapján.

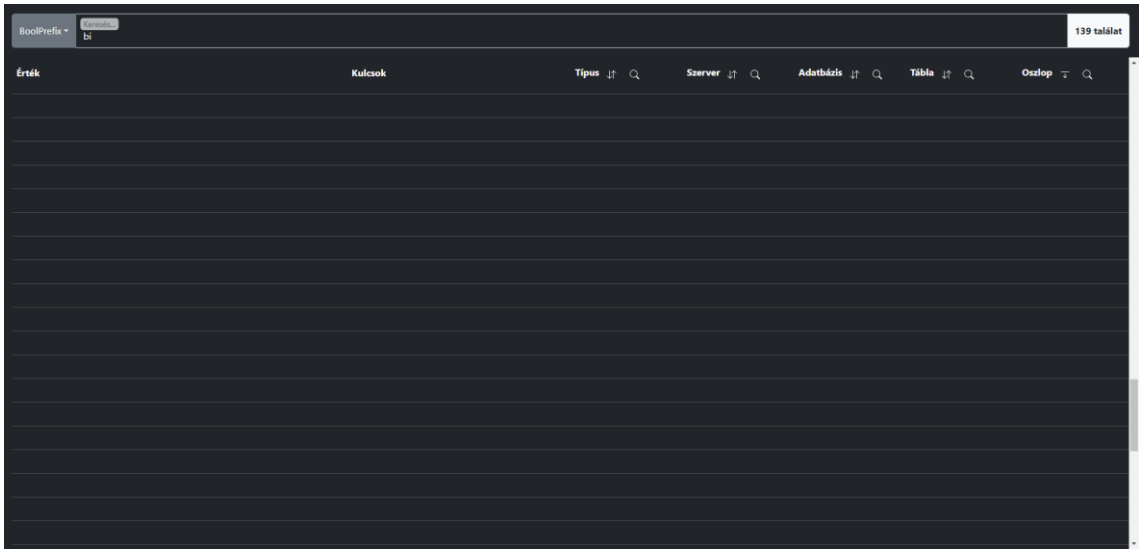
```
var bulkRequestDescriptor = new BulkRequestDescriptor();
bulkRequestDescriptor.IndexMany(_documents,
    (descriptor, document) => descriptor.Id(CreateId(document)));

private Id CreateId(ElasticDocument document)
{
    var idHashFactory = new IdHashFactory(elasticDocumentIdHashDto);
    return idHashFactory.Create();
}

public string Create()
{
    string json = JsonSerializer.Serialize(_elasticDocumentIdHashDto);
    byte[] jsonBytes = Encoding.UTF8.GetBytes(json);

    using var hashAlgorithm = MD5.Create();
    var hashedBytes = hashAlgorithm.ComputeHash(jsonBytes);
    string hash = string.Concat(hashedBytes.Select(b => b.ToString("X2")));

    return hash;
}
```

Az elkészült listát az ElasticsearchClient BulkAsync függvényének átadva az utolsó szinkronizáció óta létrejött, illetve frissített dokumentumok áttöltődnek az Elasticsearch adatbázisába. A sikeres áttöltést követően az eljárás hasonlóan történik a törlendő dokumentumok esetében is.

3.5. Megjelenítés

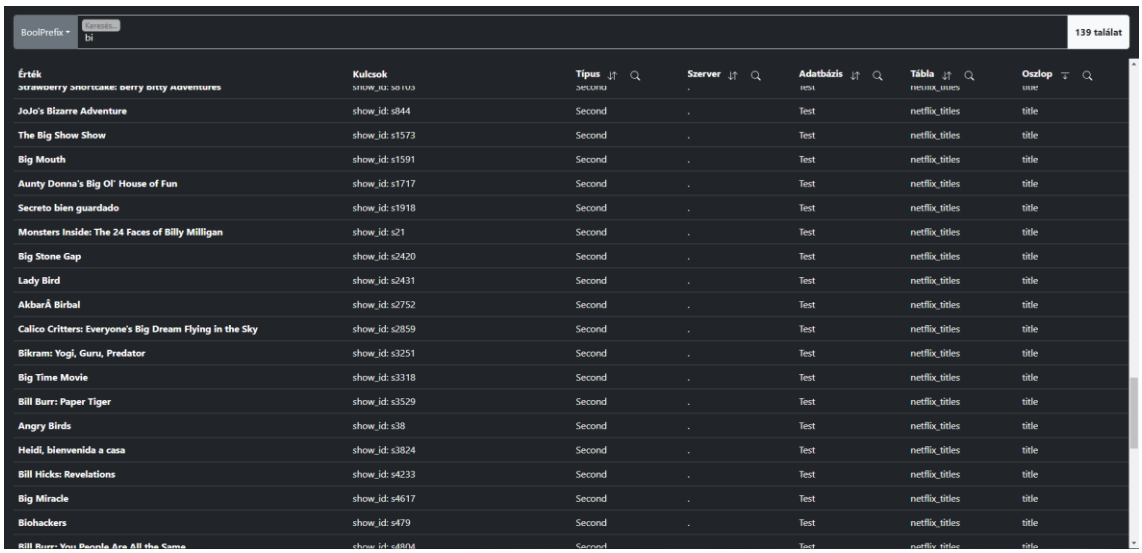
3.5.1. Quick Grid

A találatok megjelenítésében a QuickGrid [26] Blazor NuGet csomag segít. A komponens lehetővé tesz olyan paginálást, amely mindig csak a képernyőn egyszerre megjeleníthető találatokat tölti be. A táblázat görgetés hatására lekérdezi a hiányzó sorokba tartozó adatokat.



3.1. ábra: Táblázat Lazy Loading közben

A találatok ElasticDocument-ek formájában a táblázat megfelelő soraiba betöltve tovább szűrhetők oszlopaik alapján.

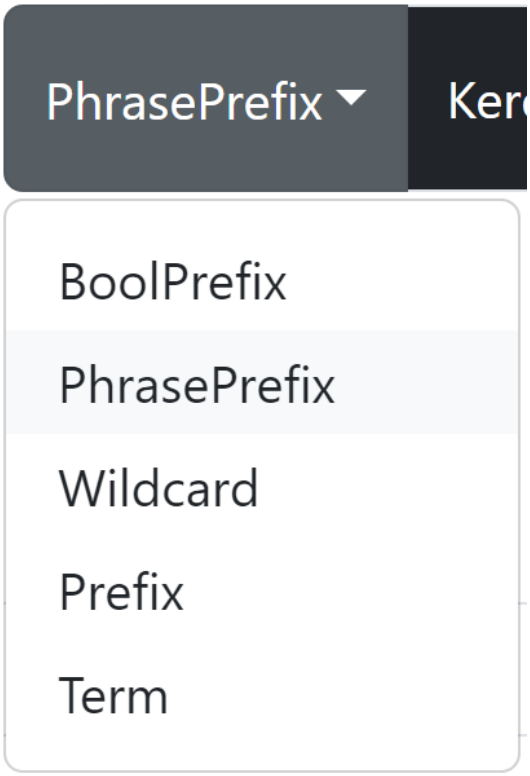


3.2. ábra: Táblázat betöltés után

3.5.2. Műveletek

A keresés és annak típusa a beviteli mező, illetve a legördülő lista segítségével adható meg. A keresési folyamat bármely gépeléssel vagy kattintással történő változás esetén lefut.

3.3. ábra: Keresési érték beviteli mező



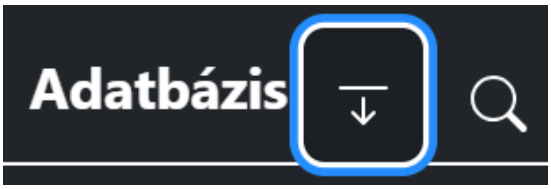
3.4. ábra: Kereső algoritmust kiválasztó lista

A BoolPrefix egy olyan összetett algoritmus, amely szóközök mentén szavakra darabolja a keresendő kifejezést és a keresést két különböző módszerrel végzi. Az első módszer olyan szövegeket keres, amelyekben sorrendtől függetlenül megtalálható a feldarabolt szavak közül a legutolsó kivételével bármelyik. Az ezt követő módszer találatként olyan szövegeket fogad el, amelyekben szerepel a legutolsó feldarabolt szóval megegyezően kezdődő szó. A találati listát a két módszer együttes eredménye képezi.

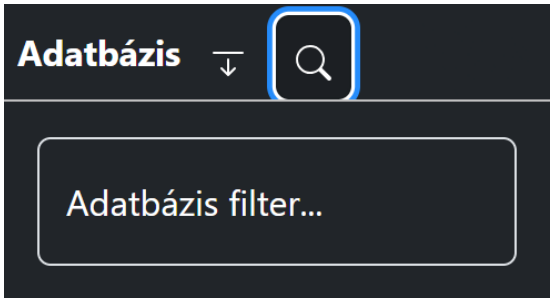
A PhrasePrefix az előbb említett algoritmushoz hasonlóan működik azzal a különbséggel, hogy ez figyelembe veszi a feldarabolt szavak sorrendjét is. A Wildcard kereső

algoritmus olyan szövegeket eredményez, amikben a keresendő kifejezés pozíciótól függetlenül teljes egészében megtalálható. A Prefix algoritmus a keresendő kifejezéssel kezdődő, Term algoritmus pedig a vele megegyező szövegeket találja meg.

Az oszlopokhoz tartozó fel-le nyíl gomb megnyomásával rendezés, a nagyító gomb megnyomásával pedig szűrés végezhető a találatokon.



3.5. ábra: Adatbázis oszlop rendezése



3.6. ábra: Adatbázis oszlop szűrése

3.5.3. SearchRequestBuilder

A keresési kérést a SearchRequestBuilder alakítja SearchRequest-té. Láncolható függvényei segítségével könnyedén létrehozható a kívánt objektum, amely átadható a keresést végző szolgáltatásnak.

```
    var searchRequest = searchRequestBuilder
        .Filters(searchFilters)
        .Ands(searchAnds)
        .Ors(searchOrs)
        .Sorts(searchSorts)
        .Take(request.Count ?? 20)
        .Skip(request.StartIndex)
        .Build();

    return _elasticsearchService.SearchAsync(searchRequest);
}
```

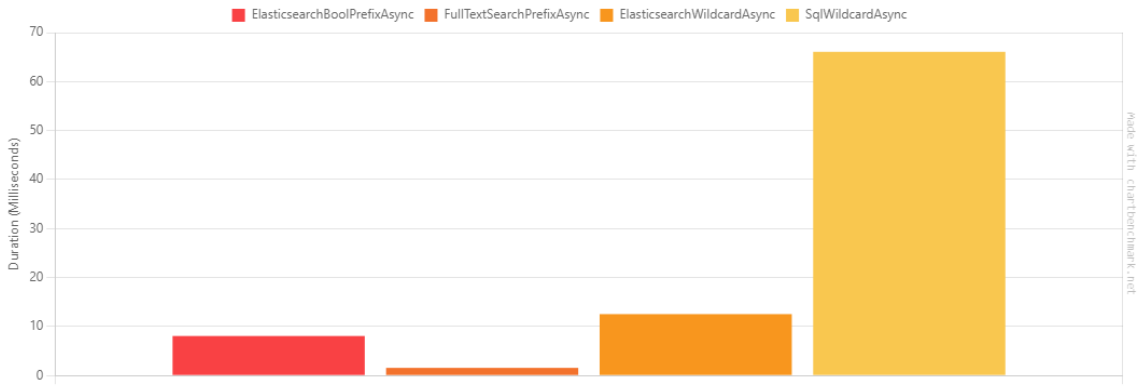
4. Eredmények

Az Elasticsearch mellett egy másik szabadszöveges keresésre szintén alkalmas módszer az SQL Server részét képező Full-Text Search. A következőkben a két módszer telepíthetőség, üzemeltethetőség, teljesítmény és funkcionalitás szempontjából kerül összehasonlításra.

Telepíthetőségüket és üzemeltethetőségüket tekintve, az Elasticsearch előre elkészített csomagokat kínál, amelyek telepítése általában nem okoz különösebb nehézséget. A Docker [27] használata az olyan összetettebb beállításoknak is megkönnyítheti a konfigurálását, mint a terheléseloszlás. Az SQL Server Full-Text Search funkciójának beállítása az adatbázis motor telepítésén kívül nem igényel egyéb konfigurálást.

Az Elasticsearch rugalmas, elosztott architektúrája miatt az alkalmazás a keresési feladatok megoldására horizontálisan is skálázható. A keresési teljesítménye a fordított indexelésnek köszönhetően nagy adatmennyiség kezelése esetén is kiemelkedő. A Full-Text Search ezzel szemben az adatbázis motorba való integráltsága miatt az adatbázis szerver erőforrásait használja. Kereséseinek sebessége a szerver leterheltségétől és a kezelendő adatmennyiségtől függően általában jóval lassabb.

Teljesítményük összehasonlítása benchmark segítségével történt. A statisztika készítése során a keresések kb. 60 ezer szöveges rekordon 1000 alkalommal kerültek megismétlésre. A benchmark futtatása során mindkét szerver teljes erőforrással rendelkezésre állt.



4.1. ábra: Benchmark diagram

Az eredményt a ChartBenchmark [28] vizualizálta egy egyesített diagramon. A grafikonon két kereső algoritmus sebességének összehasonlítása látható lineáris skálán áb-

rázolva Elasticsearch, illetve Full-Text Search esetében. Az Elasticsearch a Prefix keresést ötször lassabban, a Wildcard keresést viszont ötször gyorsabban hajtotta végre a Full-Text Search-nél. Megfigyelhető továbbá, hogy amíg az Elasticsearch keresési performanciáiban nagyságrendi változás nem vehető észre, Full-Text Search esetén ez nem mondható el.

A keresési képességeik szempontjából a Full-Text Search meglehetősen korlátozott, mindössze néhány szabadszöveges kereső algoritmust kínál. Az elkészült alkalmazás az ennél jóval rugalmasabb Elasticsearch keresési képességeit kihasználva egyszerűen bővíthető. A bővítési lehetőségei közé tartozik többek között a fuzzy logika, vagy a reguláris kifejezéseket használata is.

Összegezve az SQL Server Full-Text Search funkciója könnyebben telepíthető és nem igényel többlet üzemeltetési költséget. Mindössze néhány egyszerűbb keresési algoritmust biztosít, amelyek gyorsan integrálhatók az alkalmazásokba. Ezek az algoritmusok az Elasticsearch-csel szemben skálázhatóság és teljesítmény terén is alul maradnak. Ideális választás lehet kisebb alkalmazások esetében, amelyeknél nincsen szükség bonyolultabb keresési algoritmusok használatára. Az olyan nagyobb rendszerekben, ahol fontos a skálázhatóság, a teljesítmény, vagy a széleskörű keresés, az Elasticsearch egyértelműen jobb választásnak bizonyul.

Irodalomjegyzék

[1] Elasticsearch: https://www.elastic.co/elasticsearch, 2024.05.18.

[2] Stack Overflow: https://stackoverflow.com, 2024.05.18.

[3] Netflix: https://www.netflix.com, 2024.05.18.

[4] Match boolean prefix query: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-bool-prefix-query.html, 2024.05.18.

[5] Inverted index: https://en.wikipedia.org/wiki/Inverted_index, 2024.05.18.

[6] REST: https://en.wikipedia.org/wiki/REST, 2024.05.18.

[7] HTTP: https://hu.wikipedia.org/wiki/HTTP, 2024.05.18.

[8] NEST: https://www.elastic.co/guide/en/elasticsearch/client/net-api/current/nest.html, 2024.05.18.

[9] .NET: https://learn.microsoft.com/hu-hu/dotnet/core/introduction, 2024.05.18.

[10] Kibana: https://www.elastic.co/kibana, 2024.05.18.

[11] Kibana dashboard: https://www.elastic.co/kibana/kibana-dashboard, 2024.05.18.

[12] DALL-E 3: https://openai.com/index/dall-e-3, 2024.05.18.

[13] Temporal table: https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver16, 2024.05.18.

[14] Microsoft SQL Server: https://www.microsoft.com/en-us/sql-server, 2024.05.18.

[15] Blazor: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor, 2024.05.18.

[16] Layered Architecture: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html, 2024.05.18.

[17] Dependency Injection: https://en.wikipedia.org/wiki/Dependency_injection, 2024.05.18.

[18] JSON: https://www.json.org/json-en.html, 2024.05.18.

[19] Provider Pattern: https://en.wikipedia.org/wiki/Provider_model, 2024.05.18.

[20] Elasticsearch index: https://www.elastic.co/blog/what-is-an-elasticsearch-index, 2024.05.18.

[21] SQL Server - Full-Text Search: https://learn.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver16, 2024.05.18.

[22] Hosted Service: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-8.0&tabs=visual-studio, 2024.05.18.

[23] Cron: https://en.wikipedia.org/wiki/Cron, 2024.05.18.

[24] Connection String: https://en.wikipedia.org/wiki/Connection_string, 2024.05.18.

[25] MD5 Hash: https://en.wikipedia.org/wiki/MD5, 2024.05.18.

[26] Quick Grid: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/quickgrid?view=aspnetcore-8.0, 2024.05.18.

[27] Docker: https://www.docker.com, 2024.05.18.

[28] Charts for BenchmarkDotNet: https://chartbenchmark.net, 2024.05.18.

Nyilatkozat

Alulírott Révész Gergő László, Programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Programtervező informatikus MSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2024. május 18.

.....

aláírás

Köszönetnyilvánítás

Szeretném köszönetemet és hálámat nyilvánítani Kaposvári Dánielnek, aki felkészített a diplomamunka megírására és számtalan tanácsot adott a szakmai nehézségek megoldásához.

Köszönetet szeretnék mondani családomnak, akik feltétel nélkül támogattak és a lehető legjobb körülményeket biztosították számomra tanulmányaim során.

Külön köszönet barátaimnak, akik kikapcsolódások, illetve feltöltődések formájában segítettek átvészelni a nehezebb időszakokat.