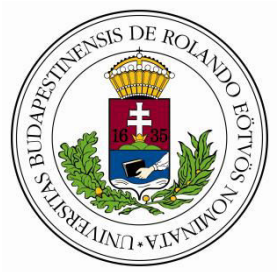


EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

INFORMÁCIÓS RENDSZEREK TANSZÉK

Térinformatikai adatok korszerű feldolgozása

Polgár Ákos

Programerverző Informatikus MSc

Témavezető:
Gombos Gergő
Doktorandusz

Budapest, 2015

Tartalomjegyzék

Absztrakt

A relációs adatbázis-kezelő rendszerek töretlen sikernek örvendenek már hosszú ideje. Azonban bizonyos feladatok megoldására vagy adatok ábrázolására nem tökéletesek. Ezen adatbázis-kezelők alternatívájaként jelentek meg a NoSQL rendszerek. A NoSQL egy gyűjtőnév a nem relációs adatmodellt használó adattárolásra.

Diplomamunkám célja egy klasszikus relációs adatbázis összevetése NoSQL adatbázisokkal, és ezáltal feltárni azon területeket, amelyekben hatékonyabban használhatóak a NoSQL rendszerek a relációs adatbázisoknál vektoros térinformatikai adatok feldolgozására. Olyan tradicionális relációs adatbázis-kezelőket fogok vizsgálni, mint a PostgreSQL, és ezeket fogom összevetni a modern NoSQL adatbázis-kezelőkkel, mint a MongoDB. Ezen összehasonlítások célja elsősorban térinformatikai adatok feldolgozását vizsgálni. A kutatás célja meghatározni bizonyos térinformatikai feladattípusokat, és ezekhez megtalálni a legmegfelelőbb eszközt a megvizsgált technológiák közül. Az eszközök meghatározása összehasonlításokon alapszik, az adatbázisok működésének sebessége mellett szempont kell, hogy legyen a használhatóságuk is. Emiatt nem csak egyszerű lekérdezéseken alapuló kéréseket futtatok, hanem saját funkciókat is írok. Az adatbázisokból való közvetlen lekérdezések helyett Node.js-ben megírt szerver alkalmazások segítségével végzem a lekérdezéseket. Ugyanazokat az adathalmazokat használom fel minden tesztkörnyezetben.

1. Bevezető

Adatbázisok

Az **adatbázis**ok rendezett adatok gyűjteményei. Emellett az adatbázis szót használva gyakran az **adatbázis-kezelő rendszer**ekre (DBMS – Database Management System) utalunk, amely a tároláson túl kezeli is az adatokat, azaz hálózatos környezetben biztosít hozzáférést (lekérdezést és szerkesztést) az adatokhoz egyidejűleg több felhasználó számára. Az adatbázis-kezelő rendszereknek több fajtája létezik.

Az adatbázisok egyik legelterjedtebb csoportja a **relációs adatbázis**ok (teljes nevén relációs adatbázis-kezelő rendszer, angolul Relational database management system, röv: RDBMS). Ezek olyan adatbázis-kezelő rendszerek, amelyek relációs modellen alapulnak, azok szoftveres implementációi. Másképp fogalmazva, egy relációs adatbázis olyan táblák halmaza, amelyek adott kategóriájú, előre definiált típusú adatokkal vannak feltöltve. Minden **tábl**ának vannak sorai és oszlopai, ahol az oszlopok egy adott adattípusú attribútumok, a sorok pedig a táblán belül azonos szerkezetű rekordok. Az **SQL** (structured query language) lekérdező nyelv segítségével lehet elérni ezen adatokat. Ebben meghatározhatjuk, hogy egy adott táblának mely soraira és mely oszlopaira vagyunk kíváncsiak, illetve az adott táblák adatait össze is kapcsolhatjuk a bennük levő adatok alapján. Erre gyakran **kulcs**okat használunk, amelyek a tábla egy vagy több sorából állnak, és minden rekordban értelmezzük őket. Egy kulcs lehet egyszerű vagy összetett, illetve elsődleges (primary) vagy külső (foreign) kulcs. Az **elsődleges** kulcs értéke a táblán belül egyedi kell, hogy legyen, ezáltal biztosítva a rekordok egyértelmű azonosíthatóságát. A külső kulcs célja, hogy csak olyan adatok szerepeljenek benne, amellyel egyértelműen beazonosítható egy másik tábla rekordja. Ezekből adódóan a kulcsok értéke sosem lehet NULL. Az **egyszerű** kulcs egyetlen attribútumból áll, az **összetett** kulcsot több oszlop kombinációja alkotja.

Az adatbázisokban levő adatok lekérdezési sebességét növelhetjük **index**ek használatával, melyek segítségével egy lekérdezés esetén nem kell a tábla minden sorát megvizsgálni. Az indexek olyan adatszerkezetek, amelyek a tábla valamely mezőinek értékét tárolják el egy

könnyen bejárható formában az értékek szerint rendezve. Megkülönböztetünk **egyszerű** és **összetett** indexeket, akárcsak az egyszerű és az összetett kulcsok esetében. Az indexek hátránya, hogy az írási, módosítási műveletek ideje megnő, hiszen az indexeket is tárolnunk kell az adatbázisban.

Gyakran használjuk a B-fa adatszerkezetet indexek tárolására. A **B-fa** olyan fa adatszerkezet, ahol a levelekben tároljuk fizikailag az adatot, amelyek egy fán belül egyenlő távolságra vannak a gyökértől. A többi pontban mutatókat tárolunk csak. Az adatokat rendezetten tárolja, az adatok mennyiségének növekedésével a beillesztés és törlés műveletigénye logaritmikusan nő. A belső pontokban a keresés lineáris vagy logaritmikus, van, hogy a levélen belül is keresnünk kell. A B-Fa csúcsai az előre meghatározott tartományban változó mennyiségű gyerekcsúcsot tartalmazhatnak. Más elnevezései: r-áris, r-ed rendű keresőfa, r-ed rendű rendezőfa, ahol r a belső csúcsok maximális gyerekszáma, és egy adott fára rögzített. Angol elnevezése B-tree.

Mivel egyetlen számítógép gyakran nem elég egy többfelhasználós alkalmazás kiszolgálására, ezért szükség lehet elosztott adatbázis-kezelőkre. Az **elosztott (vagy osztott) adatbázis** egy fizikailag megosztott, de logikailag egységes adatbázis. A számítógépek egységét, amelyek egy elosztott adatbázis részét képezik, **klaszter**nak (vagy cluster, fürt) nevezzük. Amikor egy adatbázist logikailag felosztunk, vagy az azt alkotó elemek különálló, független részekre bontjuk, **particionálás**nak hívjuk. Amikor egy új partíciót hozunk létre, az történhet a jobb kezelhetőség, a teljesítmény növelése vagy a rendelkezésre állás javítása céljából.

A **skálázhatóság** a rendszer azon képessége, hogy a rendszer teljesítménye a rendelkezésre álló erőforrásokkal mennyire változik lineáris arányban.

Egy adott alkalmazás skálázására alkalmas módszereket két kategóriába sorolhatjuk, a **függőleges** irányú skálázásra (Scale vertically, scale up) és a **vízszintes** irányú skálázásra (Scale horizontally, scale out). A függőleges skálázás során a rendszer egy kiválasztott eleméhez adunk új erőforrást, például egy számítógépet bővítünk hardveresen. A vízszintes skálázás esetén a rendszert bővítjük egy új elemmel, például számítógéppel vagy

routerrel.

A rendszer skálázása és logikai particionálása azonban néhány hiba lehetőségét is magában hordozza, és minden egyes igénynek egyetlen adatbázis-kezelő sem tud eleget tenni. Az Eric Brewer *(Browne, 2009)* által megfogalmazott **CAP tétel** kimondja, hogy egy elosztott rendszer a három alapvető képesség közül (konzisztencia, rendelkezésre állás, particionálás-tűrés) legfeljebb kettőt tud egyszerre teljesíteni.

A **konzisztencia** azt jelenti, hogy csak konzisztens adatok kerülnek tárolásra, azaz egy elosztott adatbázisban a tárolt adatok mindenhol egységesek. Ez garantálja, hogy nem történhet olyan eset, hogy a rendszer egyes elemei különböző válaszokat szolgáltatnak a klienseknek. Az adatbázis állapotváltozásai csak egyszerre történhetnek a csomópontokon, ezért a csomópontoknak szinkron működésűnek kell lenniük. Akkor nevezhető egy adatbázis konzisztens rendszernek, ha egy adott lekérdezésre egy adott időpontban az adatbázis bármely csomópontjából ugyanazon információt kapja vissza a kliens. A konzisztencia különösen a kritikus rendszereknél fontos (például bank, közlekedés).

A hozzáférhetőség vagy **rendelkezésre állás** százalékban kifejezhető érték, amely megmutatja, hogy az adott rendszer az esetek hány százalékában volt elérhető. Több csomópont esetén ennek mérése bonyolult, ugyanis elképzelhető, hogy a rendszer nagy része elérhető, de a benne található egyes csomópontok kiesése miatt a teljes adattartalom mégsem elérhető. Akkor mondjuk, hogy egy rendszer teljesíti a rendelkezésre állást, ha ennek értéke száz százalék.

A **particionálás-tűrés** akkor merül fel egy rendszernél, amikor a csomópontjai közötti hálózatban hiba keletkezik. Ekkor az adatbázis két különálló hálózatra szakad. A két hálózat ilyenkor nem tud egymás működéséről, teljesen függetlenül változnak a kapcsolat megszakadása után. Amikor a hiba megszűnik, és a két hálózat újra egyesül, a rendszernek képesnek kell lennie az adatokat szinkronba hozni, hogy megszakítás nélkül folytatható legyen az adatbázis működése.

Habár ezek közül minden adatbázis-kezelő csak kettőt tud garantálni, ettől függetlenül egy jó adatbázis-kezelő nem hagyja el teljesen a harmadikat sem, közelíteni próbál annak megvalósításához. A relációs adatbázisok a konzisztenciát és a rendelkezésre állást biztosítják, a NoSQL rendszerekben ez a rendszer típusától függően változó.

A **replikáció** egy adatbázis duplikáció készítésének és kezelésének folyamata. Célja hogy biztosítva legyen a konzisztencia a redundáns erőforrások közt, és hogy növeljük a megbízhatóságot, a hibatűrést és a hozzáférhetőséget. A gyakorlatban a replikáció több telepített szerver, amelyek egymás klónjai.

A vízszintes skálázás és az egyre több replika készítése egyre növekvő számú hálózati kapcsolatot is jelent, ezáltal megnövelve a hibalehetőségek esélyét. Az **integritás** biztosítja, hogy egy rendszeren belül egy tranzakció akkor is kivitelezhető lesz, ha a művelet – általában hálózati problémák miatt – megszakad. Ezt az SQL adatbázisok igen, de a NoSQL adatbázisok nem feltétlenül garantálják. Ahhoz, hogy egy adatbázis integritása garantálva legyen, az adatábizsnak ACID tulajdonságúnak kell lennie. Adatbázisok esetén az **ACID** mozaikszó az Atomicity (atomicitás), Consistency (konzisztencia), Isolation (izoláció), és Durability (tartósság) rövidítése. Ezek az adatbázis-kezelő rendszer tranzakciófeldolgozó képességeinek alapelemei.

Atomitás: A tranzakciónak minden egyes atomi eleme teljesül, vagy egyik sem.

Konzisztencia: Az adatbázis megőrzi az integritását, sosem kerül érvénytelen állapotba a tranzakció előtt és után. Ez a konzisztencia nem teljesen egyezik a CAP tétel esetében definiálttal, ugyanis ott egy klaszter szintű konzisztenciáról, az ACID esetében pedig egy csomópont szintű konzisztenciáról van szó.

Izoláció: Az egyes tranzakciók a többitől elhatárolva futnak le, a többi tranzakcióból nem láthatóak annak hatásai. Biztosítja, hogy az egy időben zajló tranzakciók olyan állapothoz vezetnek, mint amilyet sorban végrehajtott tranzakciók érnének el.

Tartósság: A változásokat egy tartós adattárolón kell tárolni, hogy a szoftver vagy a hardver meghibásodása, vagy egyéb hiba esetén is megmaradjanak az adatok.

Az ACID tulajdonságok megvalósításához gyakran szükség van zároló műveletekre. A **zár** (angolul: lock) egy szinkronizációs módszer, mely limitálja a hozzáférést olyan erőforrásokhoz vagy objektumokhoz (például: táblák, kapcsolatok, adatblokkok), amelyek a futás közben több szálon is eléretőek. Léteznek különböző zárak, melyek az adott erőforrások vagy objektumok biztonságát szolgálják. Az egyik ilyen a „transactional locking", amely ahhoz szükséges, hogy egy adatbázis ACID tulajdonságát fenntartsuk az olyan esetekben, amikor egyidejűleg több felhasználó képes hozzáférni az adatbázishoz. Ha valami hiba történik egy tranzakció folyamán, akkor az adatbázis visszaállítódik a legutolsó konzisztens állapotára. Ennek a műveletnek a neve **rollback**.

Térinformatikai adatok, műveletek

Egy **térinformatikai adatbázis-kezelő rendszer** (röviden: térinformatikai adatbázis, angolul Spatial database management system, röv: SDBMS) egy olyan adatbázis, mely optimalizálva van olyan objektumokkal való feladatok elvégzésére, melyeknek a térbeli helyzete definiálva van. A leggyakoribb ilyen geometriák a pont, a vonal és a poligon. Ezen geometriák legfőbb tulajdonsága, hogy őket nem az euklideszi síkban illetve térben helyezzük el, hanem valamilyen **szférikus koordinátarendszer**ben. A szférikus koordinátarendszerek közös tulajdonsága, hogy a Föld alakját közelítik. A Föld elméleti alakja a **geoid,** amely a világóceánon nagyjából azonos a tengerszinttel, a kontinensek alatt pedig helyzete geofizikai és geodéziai mérésekből számítható ki. A geoidot helyettesítő normál szferoid méretét jól megközelítő szabályos matematikai (geometriai) felület, a **forgási ellipszoid**. A **szférikus geometria** (másnéven gömbi geometria) egy olyan nemeuklideszi geometria, amely alapeleme a (Föld)gömb, amelyet a síkra kivetítünk, így kapva egy koordináta-rendszert.

A forgási ellipszoid paramétereit az egyre pontosabb mérések segítségével többször is meghatározták. Kérdés, hogy a meghatározott méretű ellipszoid hogyan helyezkedik el a Föld elméleti alakjához, a geoidhoz viszonyítva. Ezt nevezzük **geodéziai dátum**nak. Magyarországon jelenleg két ilyen dátumot használunk, az egyik az IUGG/1976 ellipszoid Magyarországi alkalmazásokhoz meghatározott, lokális elhelyezését leíró **HD72** (Szilárd,

2002) jelű dátum (Hungarian Datum 1972), és a nemzetközileg meghatározott, globális, geocentrikus elhelyezésű **WGS 84** *((IfEN), 1998)* (World Geodetic System 1984) dátum (ezt alkalmazzák pl. a GPS mérések esetén).

Az **ortodroma**, vagy ortodromikus távolság, a földfelszín két pontja közötti legrövidebb távolság, amit a Föld felszínén a két pontot összekötő geodetikus vonal (gömb esetén főkör) mentén mérünk. Az euklideszi geometriában a legrövidebb távolságot a két pontot összekötő egyenes mentén mérjük.

Ezen távolságok kiszámítására több formula is létezik, a három leggyakrabban használt az algoritmus bonyolultsági sorrendjében növekvően: 'Spherical law of cosines', 'Haversine' *(Sinnott, 1984)* és 'Vincenty' *(Vincenty, 1975)*. Az első kettő feltételezi, hogy a Föld gömb alakú, míg a 'Vincenty'azt tételezi fel, hogy az egy ellipszoid, így közelebb járva az igazsághoz. Kisebb távolságok mérésére az első két formula nagyon apró eltérésekkel dolgozik. A'Vincenty'eljárás különbségei a leghangsúlyosabbak *(Hijmans, 2014)*.

Legyen θ_1, λ_1 és θ_2, λ_2 a két pot latitude és longitude koordinátája, és $\Delta\theta, \Delta\lambda$ az ő abszolút különbségeik. Ekkor $\Delta\sigma$, a köztük levő központi szög (radiánban).

'Spherical law of cosines'

$$\Delta\sigma = arccos(sin\theta_1 sin\theta_2 + cos\theta_1 cos\theta_2 cos\Delta\lambda)$$

'Haversine'

$$\Delta\sigma = 2arcsin\left(\sqrt{sin^2\left(\frac{\Delta\theta}{2}\right) + cos\theta_1 cos\theta_2 sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

'Vincenty'

A Vincenty formula két, egymáshoz kapcsolódó iteratív módszer, ahol az inverz módszerrel számolhatjuk ki két pont közt a távolságot.

$$\Delta\sigma = arctan\left(\frac{\sqrt{(cos\theta_2 sin\Delta\lambda)^2 + (cos\theta_1 sin\theta_2 - sin\theta_1 cos\theta_2 cos\Delta\lambda)^2}}{sin\theta_1 sin\theta_2 + cos\theta_1 cos\theta_2 cos\Delta\lambda}\right)$$

Ezen formulákból végül a következőképp kapjuk meg a távolságot:

$$d = r\Delta o$$

ahol a távolság d, a sugár r.

Több térinformatikai művelet létezik, a fejezetben csak a diplomamunkában előforduló műveletek definíciója szerepel. A térinfomatikai műveleteket a **térinfomatikai adattípusok**on hajtjuk végre, melyek a következő geometriák (ADT [1]szinten): Point, Polyline, Complex polyline, Polygon, Polygon set, Mixed.

Az **intersects** azt dönti el, hogy két geometriának van-e közös metszete.

A **within** azt dönti el, hogy egy adott geometria teljes mértékben benn van-e egy másik geometriában. Ahhoz, hogy ez a művelet értelmezhető legyen, az utóbbi geometriának poligonnak kell lennie.

A **distance** két geometria közti legkisebb távolságot adja vissza.

A **bounding box** (másnéven: bounding rectangle) egy olyan téglalap, melynek oldalai párhuzamosak a koordináta-tengelyekkel. Általában a téglalap bal alsó és jobb felső ponjának megadásával definiáljuk. Gyakori alkalmazása egy bounding box definiálásának, hogy intersects vagy within műveletekben használjuk fel, mint az egyik input geometria.

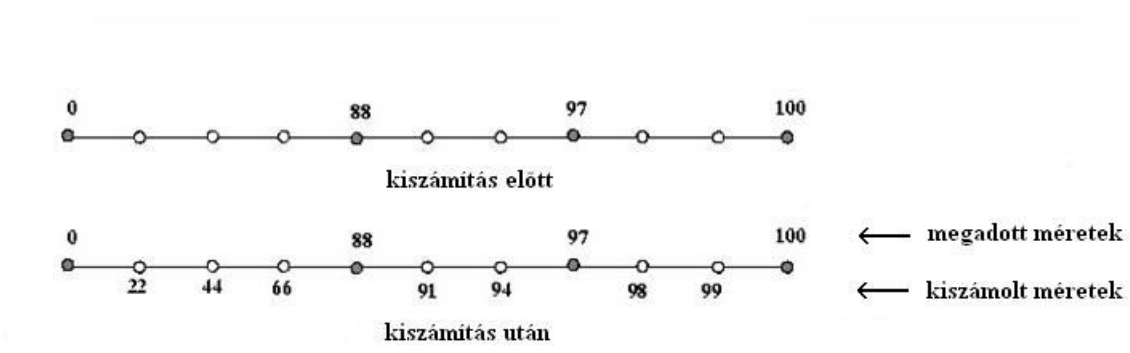
Az útvonalaknak (polyline-ok) nem feltétlenül csak a koordinátáikkal mérhetőek. Az **LRS** (Linear referencing system – magyarul: lineáris hivatkozási rendszer) azt teszi lehetővé, hogy útvonal jellegű alakzatoknak, azaz polyline-ok egyes pontjait mérőszámokkal lássuk el. Olyan ez, mintha egy út két kilométerköve közt egy köztes pontnak is adnánk egy paramétert, meghatározva a távolságát a két kilométerkőtől. Ezt a paramétert **méret**nek

[1] Absztrakt adattípus (a rövidítés az angol Abstract data type elnevezésből adódik)

hívjuk. Tehát ha megadunk egy pontot, visszakapjuk a méretet. Fordítva használva ezt, egy pontot meghatározhatunk két egyetlen paraméterrel.

Az útvonalnak mindig van egy kezdő és egy végpontja. A méret információnak az út mentén haladva mindig monoton módon kell változnia, lehet növő vagy csökkenő, nem kell szigorúan monotonnak lennie. Az útvonalaknak van iránya, ez a monotonitásból meghatározható. Egy LRS útvonalnak a megadása hasonló, mint a hagyományos útvonalaké, menetirány szerint felsoroljuk a pontok koordinátáit. Ezen felül tetszőlegesen megadhatunk a pontokhoz tartozó méret információt. Azokat a pontokat, amelyekben méret információt is megadtunk, **LRS pont**oknak nevezzük. Amikor egy hagyományos útvonalat adunk meg, akkor a végpontok koordinátáit adjuk meg. Ezek a pontok definiálják tulajdonképpen az útvonalat. Az LRS utak esetén ezeket a definiáló pontokat mindenképpen meg kell adnunk, de ezen kívül előfordulhat, hogy valamely további pontban szeretnénk méret információt megadni. Az útvonalat definiáló pontokat és az LRS pontokat együttesen úgynevezett **shape pont**oknak nevezzük. Előfordulhat tehát, hogy egy szakasz belső pontjában megadunk méret információt (ez egy LRS pont lesz), egy másik szakasz végpontjában pedig nem (ez egy nem LRS, de shape pont lesz).

Ha a méret csak a szakasz vagy útvonal hosszát jelenthetné csupán, akkor csak a kezdő és a végpont méret információját volna értelme megadni. Valójában azonban tetszőleges értékeket megadhatunk méret információként, olyat is, aminek semmi köze nincs a hosszúsághoz. Nézzük meg a következő példát, melyet az 1. ábra illusztrál.



1. ábra - LRS pontok méret információinak kiszámítása *(Nikovits, 2012)*

Az útvonalnak négy pontjára adtunk meg méret információt, de az első, a második és a harmadik rész mérete (88, 9 és 3) nincs arányban a hosszukkal. Ennek lehet célja útvonalak súlyozása. Képzeljük el, hogy egy útvonalon a menetidőt vagy a költségeket tároljuk el méret információként, mert útszakaszonként a sebesség vagy a költség eltér. Ilyenkor az LRS információk segítségével kérdezhetjük le, hogy az útnak egy adott pontjára mennyi idő alatt, illetve milyen költséggel juthatunk el. A mérettel nem rendelkező közbeeső pontok méret információját a szomszédos LRS pontok méretei alapján kell számolnunk úgy, hogy ez esetben már a szomszédos LRS pontok által meghatározott szakasz tényleges hosszát vesszük figyelembe, ahogy az az 1. ábrán is látható.

GIS

A **földrajzi információs rendszer** egy olyan számítógépes rendszer, melyet földrajzi helyhez kapcsolódó adatok gyűjtésére, tárolására, kezelésére, elemzésére, az információk megjelenítésére, a földrajzi jelenségek megfigyelésére, modellezésére használható. Nevezik térinformatikai, geoinformációs rendszernek, angolul Geographic information system-nek, ami rövidítve **GIS**. A GIS egyetlen rendszerbe integrálja a térbeli és a leíró információkat – alkalmas keretet biztosít a földrajzi adatok elemzéséhez.

Amikor összeállítunk egy alkalmazást, nagyon fontos, hogy előre meghatározzuk, hogy az milyen **komponens**ekből fog állni. Egy szoftver komponens egy olyan modul, amely egységbe zárja az egymáshoz kapcsolódó metódusok és adatok gyűjteményét. Minden folyamat különböző komponensben helyezkedik el. Egy GIS rendszerben ezeket a komponenseket külön-külön **GIS szoftver**eknek hívjuk. A GIS szoftverek a következő kategóriákba sorolhatóak *(Stefan Steiniger, 2011)*:

Térinformatikai adatbázisok (SDBMS): A kutatás szenpontjából legfontosabb komponens, elsődleges funkciója adatok tárolása fájlokban. Ezek gyakorlatilag adatbázisok, ebből kifolyólag önmagukban nem elegendőek egy GIS rendszer létrehozására, de más – nem feltétlenül térinformatikai – eszközökkel összekapcsolva már képesek GIS rendszerként működni.

desktop GIS: Adatok importálására, létrehozására és módosítására használhatóak lokális

környezetben.

Server GIS: Egy olyan szoftver, mely web protokollokon keresztül funkcionalitásokhoz biztosít távoli hozzáférést, és képes adatokat küldeni illetve fogadni más web alapú szolgáltatásoknak illetve szolgáltatásoktól.

Web map server: Egy speciális webes szerver alkalmazás, amely biztosít Web Mapping Services (WMS; ISO 19128) *(de la Beaujardiere, 2004)*, Web Feature Services (WFS; ISO 19142) *(Vretanos, 2005)*, vagy Web Coverage Services (WCS) *(Baumann, 2008)* szolgáltatásokat. Ezek mind térinformatikai adatok, de amíg a WMS képeket ad vissza, addig a WFS és a WCS feldolgozható adatokat ad vissza (például egy XML-t). A WFS és a WCS közti különbség, hogy a WCS térben és időben reprezentálja az adatokat.

Web GIS client: Hozzáférést biztosítanak térinformatikai adatokhoz webes környezetben.

Mobil GIS: Olyan mobil eszközön futó szoftver, amely támogatja a térinformatikai adatok lekérdezését, korrigálását (redlining), és frissítését.

GIS könyvtárak és kiterjesztések: Speciális funkciókat biztosítanak, ezzel kibővítve az alap rendszer adta lehetőségeket.

NoSQL

A NoSQL kifejezésnek kezdetben nem SQL volt a definíciója, ma már inkább Not only SQL, azaz nem csak SQL adatbázis kezelő rendszerek összefoglalóját jelenti. Ezen adatbázisok nem szükségszerűen relációsak. Gyakran egy speciális problémára adnak egy különösen jó megoldást, cserébe azonban más feladat megoldásában gyengék. Ahogy láthattuk, a definíció nem teljesen egzakt, mégis vannak tulajdonságok, melyek minden NoSQL adatbázisról elmondhatók.

Nem relációs adatmodellek, ha van is séma-beli megkötés, az gyenge. A rendszerek (horizontálisan) skálázhatóra tervezettek, az adatok replikációja, particionálása támogatott. A rendszer általában valamilyen konzisztencia modellt használ, de nem feltétlenül ACID. A NOSQL adatbázisok esetében gyengébb elvárások jelentek meg a

konzisztencia felé, mint az SQL adatbázisok esetében. A NoSQL adatbázisokat használó rendszereknél gyakran nem okoz problémát, ha egy adatsor nem a legfrissebb változatát kapja vissza a kliens (kritikus rendszerek esetén nem alkalmaznak NoSQL adatbázist).

Valójában a NoSQL azért nem egy teljesen megfogalmazható fogalom, mert az ebbe a besorolásba eső rendszerek négy nagyobb kategóriába sorolhatók. Ezen kategóriák mindegyike különböző igényeket elégít ki.

NoSQL adatbázisok típusai

Gráf	Oszlop-család	Dokumentum	Persistent Kulcs-érték	Volatile Kulcs-érték
neo4j	BigTable (Google)	MongoDB (~BigTable)	Dynamo (Amazon)	memcached
FlockDB (Twitter)	HBase (BigTable)	CouchDB	Voldemort (Dynamo)	Hazelcast
InfiniteGraph	Cassandra (Dynamo + BigTable)	Riak (Dynamo)	Redis	
	Hypertable (BigTable)		Membase (memcached)	
	SimpleDB (AmazonAWS)		Tokyo Cabinet	

2. ábra – NoSQL kategóriák és a kategóriákba tartozó adatbázisok [2]

Kulcs-érték

A kulcs-érték adatbázisok sémája asszociatív tömb, az értékeket az egyedi kulcsok segítségével lehet elérni illetve felülírni. Ezen felül sémabeli megkötéseket nem tartalmaz. Bizonyos esetekben a kulcsokat rendezhetjük. Ezen felül általában kevés műveletet

[2] https://bigdatanerd.wordpress.com/2012/01/04/why-nosql-part-2-overview-of-data-modelrelational-nosql/

támogatnak. Ahogy a 2. ábrán látható, két alkategóriába sorolhatóak a kulcs-érték adatbázisok, azonban ezek működése csak az adatok tárolásának helyében tér el egymástól. A Persistent típusúak a szokásos módon, a fizikai diszkeken, míg a kulcs-érték adatbázisok a memóriában tárolják az adatot. Utóbbiak előnye a megnövekedett sebesség, hátránya a gyenge skálázhatóság és hogy az adatvesztés bekövetkezhet egy szerver összeomlásakor, vagy egyéb hibából kifolyólag.

Oszlop-család tárolók

Hatalmas méretű adatok feldolgozására szolgál elosztott környezetben. Hasonlóak a kulcs-érték adatbázisokhoz, de itt egy érték valójában oszlopok halmaza. A relációs adatbázisok analógiájára rájuk erőltethetjük az ott használt fogalmakat, egy „oszlop-család” felel meg a táblának, míg egy kulcs-érték pár egy rekordnak. Minden oszlop az oszlopnév, érték, időbélyeg hármasokat tárolja. Hívják még big table-nek is.

Dokumentum-tárolók

A dokumentum-tárolók központi koncepciója a dokumentum. A dokumentum valamilyen formátumban tárolt információ. A gyakran használt formátumok az XML, JSON, BSON, YAML. A relációs adatbázisok táblázatainak rekordjaitól abban térnek el, hogy a dokumentumok teljesen eltérő struktúrával rendelkezhetnek, nincs előre definiált sémájuk. Különböző szoftverek különböző módszereket használnak a dokumentumok rendszerezésére, mint pl.: gyűjtemények, címkék, metaadatok, könyvtár hierarchiák.

Gráf

A gráf-adatbázisok olyan adatok tárolására specializálódtak, amelyek jól modellezhetőek gráfként, azaz az adatok határozatlan számú kapcsolattal vannak összekötve. Ilyen adatszerkezetek például a közösségi hálók, tömegközlekedési térképek, hálózati topológiák.

NoSQL Adatbázisok összehasonlítása

A térinformatikai adatok kezelésére létező NoSQL megoldások egy részét az adott eszközzel közvetlenül, míg másokat valamilyen külső plugin segítségével érhetünk el. Mivel ma már rengeteg NoSQL adatbázis létezik, ezért el kell döntenünk, hogy ezek közül

melyek azok a rendszerek, melyeket érdemes megvizsgálnunk. Több technológia létezik, ezekből négyet vizsgáltam meg, melyekről egy összefoglaló táblázatot mutat a 3. ábra.

CouchDB with the GeoCouch extension

Dokumentum-tároló típusú adatbázis. Gyenge a teljesítménye, ezen a lekérdezésekben használt MapReduce segítségével javíthatunk. Kevés térinformatikai támogatás, ezek a bounding box, radius és polygon keresések. Könnyű a használata, azonban ritkán frissítik, ekkor is csak minimális javításokkal.

MongoDB

Szintén dokumentum-tároló. Magas teljesítmény. Limitált térinformatikai támogatás, azonban gyakran adnak ki új verziót, melyek folyamatosan egyre nagyobb térinformatikai támogatást biztosítanak.

Neo4j using the Neo4j-Spatial plugin

Gráf adatbázis, ennek megfelelően az adatok tárolása tulajdonsággráfban történik. Ebből következőleg a bennfoglalás műveletek gyengébbek, mint a relációs adatbázisok esetében, azonban a modellhez illeszkedő műveletek az adat mennyiségétől függetlenül gyorsak. Bizonyos területeken lehet előnyös a használata, ilyenek a közelségi keresések, vagy útvonaltervezés.

Big Table with the GeoModel and GeoDataStore extension

Kulcs-érték adatbázis. Limitált támogatottság, összetett geometriákat nem tudja kezelni, csak a distance funkciója lett implementálva. Előnye, hogy hatalmas adatmennyiséget kezel, masszívan skálázható.

	CouchDB	MongoDB	Neo4j	BigTable
Adatbázis típusa	Dokumentum tároló	Dokumentum tároló	Gráf	Oszlop-család tároló
Indexelés	R-fa (csak 2D)	2D, 2Dsphere (B-fa alapú)	R-fa (2D, részleges 3D)	B-fa (csak 2D)
Vektoros adattípusok	Teljes	Teljes	Teljes alaptípusok és limitált multigeometriák	Teljes alaptípusok
Topológiai funkciók	Csak Within() és Contains()	Csak Within() és Contains()	Majdnem teljes	Nem támogatott
Analitikus és metrikus funkciók	Csak Distance()	Csak Distance(pont)	Teljes	Csak Distance()
Halmaz funkciók	Nem támogatott	Csak Intersection()	Teljes	Nem támogatott
Bemeneti és kimeneti formátumok	Input: .SHP Output: .KML .CSV .GeoJSON	Input: .GeoJSON Output: .GeoJSON	Input: .SHP .OSM Output: SLD styled PNG	Input: .KML .JSON Output: .KML .JSON

3. ábra - Geografikus NoSQL rendszerek összehasonlítása *(Pourabbas, 2014)*

Korábbi kutatások *(Baas, 2012)* alapján a Neo4j tűnik a legígéretesebbnek, azonban ezen kutatások óta a MongoDB adatbázisnak jelent meg több újabb verziója, amelyekben bővültek a lehetőségek. Mivel a MongoDB-t már ez a kutatás is a második helyre sorolta,

ezért érdemes lehet megvizsgálni az új lehetőségeit. A 2013 márciusában megjelent 2.4-es MongoDB térinformatikai támogatása jelentősen megnőtt, ekkor jelent meg a 2DSphere indexelés és újabb térinformatikai műveletek. Amíg régen a MongoDB csak pont típusú adatokat kezelt, a 2.6-os verzióra ez a lista kibővült.

2. MongoDB

A MongoDB a NoSQL adatbázisokon belül a dokumentum-tároló típusú adatbázisokhoz tartozik. Saját lekérdező nyelve van. A MongoDB az adatokat BSON dokumentumokban tárolja. A BSON a bináris reprezentációja a JSON-nak, köztük bijektív leképezés van. A dokumentumokban egy mező típusa bármilyen BSON típus lehet, beleértve dokumentumokat, tömböket, vagy dokumentumok tömbjét. A MongoDB minden dokumentumot gyűjteményekben tárol. Egy gyűjtemény egymáshoz kapcsolódó, egy célt szolgáló dokumentumok csoportja, analóg az RDBMS-ből ismert tábla fogalmával. A gyűjteményeknek nincs sémájuk, ebből kifolyólag az egy gyűjteményben tárolt dokumentumoknak eltérő mezőik lehetnek. A CAP tétel három tulajdonsága közül a kozisztenciát és a partícionálás-tűrést teljesíti. Ez természetesen nem jelenti azt, hogy nincs rendelkezésre állása, csupán az nem száz százalékos.

A dokumentumok a relációs adatbázisok esetében levő táblákkal feleltethetőek meg, azonban ezeknek a sémájuk nem annyira kötött. Az adatbázisoknak azonban van némi struktúrája. A „system namespace” tartalmaz explicit listákat a gyűjteményekről és indexekről. A rekordoknak a BSON dokumentumok feleltethetőek meg. A MongoDB saját lekérdező függvényeinek segítségével érjük el a dokumentumokat. Megadhatunk feltételeket, melyek azonosítják, hogy a MongoDB mely dokumentumokat adja vissza. Egy lekérdezés tartalmazhat projekciót, melyben meg kell adnunk a dokumentum visszaadni kívánt mezőit. Lehetőség van a lekérdezések módosítására a limits, skips, és sort parancsokkal. Minden dokumentumnak van egy _id nevű mezője, mely akkor is automatikusan létrejön, ha a dokumentum létrehozásakor azt nem definiáljuk. Az automatikusan létrejövő _id értéke egy speciális 12 bájtos BSON objektum, amely random[3] jön létre. Az _id mező értéke egyedi egy gyűjteményen belül.

Adatainkat megváltoztathatjuk létrehozás, módosítás és törlés műveletekkel. Egy ilyen

[3] Timestamp, machine ID, process ID, és a process-local inkrementális számláló alapján

művelet egyetlen gyűjteményt változtathat csak meg. A módosítás és törlés esetén feltételek segítségével tudjuk kiválasztani a módosítani vagy törölni kívánt dokumentumokat.

Join

Az SQL nyelvekben található összekapcsolások (join) teljesen hiányoznak a nyelvből. A MongoDB a join analógiájára két lehetőséget kínál, az egyik, hogy az adatok struktúráját úgy alakítjuk ki, hogy a dokumentumokat beágyazva tároljuk. Ez az oka az összekapcsolások hiányának, hisz a MongoDB célja a gyűjtemények olyan kialakítása, ahol nincs szükségünk az összekapcsolás műveletre, az összefüggő dokumentumokat egy gyűjteményben tároljuk.

A másik lehetőségünk, hogy adatbázis referenciákat használunk. Az adatbázis referenciák használatára két lehetőség kínálkozik: a kézi referenciák és a DBRefs. A kézi referencia esetén az adatokban ki kell alakítani egy olyan struktúrát, hogy a különböző gyűjteményekben levő dokumentumokat a továbbiakban össze tudjuk kapcsolni szükség esetén. Az adatok létrehozásakor készíthetünk egy mezőt, ami alapján a dokumentumok egymásra mutatnak, egy idegen kulcshoz hasonló megoldás segítségével. Például:

```
original_id = ObjectId()//Generálunk egy egyedi azonosítót

db.places.insert({
   "_id": original_id,     //legyen az azonosító a generált elem
   "name": "Broadway Center",
   "url": "bc.example.net"
})

db.people.insert({ //Itt nem adunk meg _id-t, a MongoDB generál
   "name": "Erin",
   "places_id": original_id, //ez alapján összekapcsolhatjuk a
places-el
   "url":  "bc.example.net/Erin"
})
```

Természetesen utólag is frissíthetjük a dokumentumainkat, és létrehozhatunk kézi referenciákat.

A DBRef-ek a konvencionális reprezentációi egy dokumentumnak. Tartalmazza egy

gyűjtemény nevét, a hivatkozott dokumentum _id mezőjének értékét, és bizonyos esetekben az adatbázis nevét is.

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                "$ref" : "creators",
                "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                "$db" : "users"
               }
`
```

A fenti példában a DBRef egy dokumentumra mutat a users adatbázisban található creators gyűjteményben, ahol az _id mező értéke: `ObjectId("5126bc054aed4daf9e2ab772")`. A DBRef hátránya, hogy nem minden driver támogatja. A MongoDB ennek használatát akkor ajánlja, ha többszörös hivatkozást szeretnénk a dokumentumok közt, egyébként inkább a kézi referenciát javasolja.

Függvények

A MongoDB egyik speciális lehetősége, hogy függvényeket is írhatunk, melyeket az adatbázisban el is menthetünk, és futtathatunk. A függvényeket mindenképp a system.js nevű speciális gyűjteménybe kell menteni. A dokumentum _id elemére kell hivatkoznunk, hogy a továbbiakban elérjük az adott függvényt. A dokumentum value mezőjébe kell a függvényünket megírni, melynek lehetnek paraméterei, írhatunk benne adatbázis műveleteket. Az alkalmazás során szükség volt függvények használatára, a mérések illetve az LRS implementáció fejezetekben fogok példákat adni függvényekre.

Indexelés

Alapjában véve a MongoDB indexek hasonlóak, mint a relációs adatbázisrendszerek indexei. A MongoDB az indexeléshez a B-fa adatszerkezetet használja. A MongoDB gyűjtemény szinten definiálja az indexeket, és azon belül bármely mező, vagy almező indexelhető. Létrehozhatunk a relációs adatbázisokhoz hasonlóan egyszerű és összetett indexeket. Egy almezőre a következőképpen hozhatunk létre egy egyszerű indexet:

```
db.cities.ensureIndex( {"properties.CITY_NAME" : 1});
```

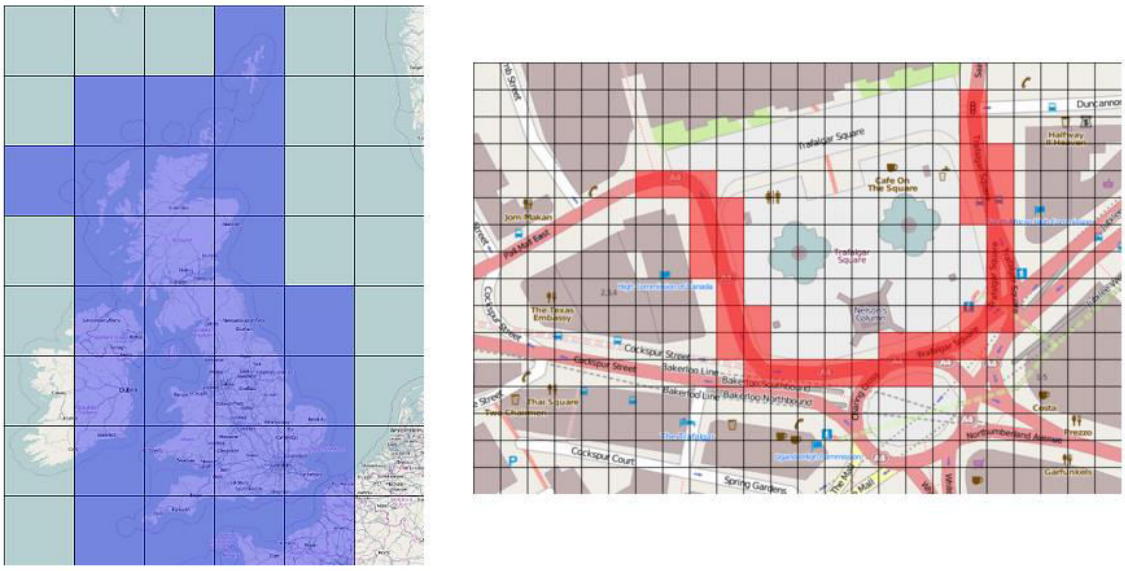
Az indexeknek különböző típusai lehetnek, melyek bizonyos típusú adatok és lekérdezések esetében hatékonyak. Az _id mezők indexe létrejön egy gyűjtemény létrehozásakor. Speciális indexelési struktúrák még a tömbökön használható multikey indexek, a térbeli (geospatial) indexek, string tartalom keresésére szolgáló text indexek, és hashed indexek.

Térbeli inexek

Két féle létezik. A 2D indexeléssel csak pontokat indexelhetünk. Később, a 2.4-es verzióban (2013 március) vezették be a 2Dsphere indexelést. A két indexelés közti fő különbség a használt koordinátarendszer. A 2Dsphere a WGS84 dátumot használja, ami egy Földgömb-szerű koordinátarendszer, amíg a 2D az Euklideszi síkot veszi koordinátarendszerül. Ebből adódóan a 2Dsphere indexelést használom a MongoDB adatbázisban található összes geometrián. Ezen felül az indexeléstől függ, hogy melyek az elérhető lekérdezések, bizonyos indexekkel csak bizonyos térinformatikai lekérdezések futtathatóak (5. ábra).

A 2dsphere indexek esetén előszűrhetünk egy gyűjteményt a nem geometriai mezői alapján, ami gyakran növeli a hatékonyságot. Támogatja az adatok tárolását GeoJSON objektumokban és koordináta párokként.

A MongoDB a térinformatikai indexeléshez a C++ egyik nyílt forráskódú könyvtárát, az s2-t használja. Ennek segítségével választja ki cellák minimális halmazát, amely teljesen lefed egy geometriát. Ezen grid cellák neve **covering**, és a cellák mérete dinamikus (0,5-100km oldalú négyzetek), a lefedni kívánt poligon méretétől függően, ahogy a 4. ábrán is látható. Az indexeket B-fákban tároljuk. Minden covering cellát hozzáadunk egy kulccsal együtt a B-fa indexhez, így a földrajzi elhelyezkedések könnyen számolhatók. Az ugyanazon területen levő finomabb felbontású (kisebb) cellák prefixe egyezni fog a nagyobbakéval.



4. ábra – Különöbző méretű covering cellák, balra az Egyesült Királyság, jobbra a Trafalgar tér[4]

A MongoDB-ben a 2Dsphere indexnek két verziója létezik, melyek az 1-es és 2-es, a különbség az, hogy a korábbi (1-es) verzió nem enged beszúrni olyan elemet, amelynek az indexelt mezője NULL vagy egy üres tömb. Azon dokumentumokat azonban, ahol az indexelt mező nem létezik, beszúrja a gyűjteménybe, és az indexelt lesz a dokumentum. Az új (2-es) verzióban, ha az indexelt mező nem létezik, vagy létezik, de értéke NULL vagy üres tömb, a dokumentum bekerül a gyűjteménybe, de nem lesz indexelt a dokumentum. Ez jelen esetben nem releváns, mert minden dokumentum tartalmaz geometriát, így a 2-es verziót használtam, ami a MongoDB 2.6-os verziója óta az alapértelmezett verzió.

Térinformatikai adattípusok

A MongoDB jelenleg a következő geometriák használatát támogatja. Zárójelben a MongoDB azon verziószáma látható, amikor bekerült a geometria támogatása.

- Point **(2.0)**
- LineString**(2.4)(ez a polyline-nak felel meg ADT szinten)**

4 http://blog.MongoDB.org/post/50984169045/new-geo-features-in-MongoDB-2-4

- Polygon**(2.4)**
- MultiPoint**(2.6)**
- MultiLineString**(2.6)** **(ez a complex polyline-nak felel meg ADT szinten)**
- MultiPolygon**(2.6)** **(ez a polygon set-nek felel meg ADT szinten)**
- GeometryCollection**(2.6)** **(ez a mixed-nek felel meg ADT szinten)**

Egy pontot értelemszerűen két koordinátával, egy vonalat pedig pontok tömbjével reprezentálunk menetirány szerint. Egy poligont a pontok tömbjeinek tömbjével reprezentáljuk. Minden tömbnek ugyanaz kell, hogy legyen a kezdő és a végpontja. Az első tömb határozza meg a külső geometriát, a többi tömb pedig a lyukakat írja le a poligon belsejében. Természetesen ezek a lyukak nem metszhetik egymást. Ha a poligon több, nem összefüggő külső geometriából áll (pl. szigetek) akkor multipoligonnak hívjuk.

Térinformatikai műveletek

A térbeli indexelés után egyszerűvé válik a geometriák keresése például within, intersects műveletekkel. A geometriáknak generálódik egy covering, és ezen covering celláit vetjük össze a B-fákban levő cellákkal, így leszűkítve azokat az egyáltalán szóba jöhetőkre, melyeket a szűkítés után részletesebben is vizsgálunk, hogy az adott kritérium tényleg teljesül-e.

Ahogy a NoSQL fejezetben leírtam, a MongoDB térinformatikai szempontból nem ad akkora támogatottságot a térbeli lekérdezésekhez, mint a relációs adatbázisok térinformatikai moduljai. Bizonyos lekérdezéseket térbeli indexek nélkül is futtathatunk, habár a hatékonyságuk így csökken valamivel. A két (2D, 2Dsphere) különböző index különböző térbeli lekérdezések esetén garantálja azt, hogy a lekérdezések hatékonyabbak lesznek, mintha azokat index nélkül futtatnánk. Egyes lekérdezések azonban csak az adott indexelés létezésekor érhetőek el, ahogy az az 5. ábrán látható. A táblázat tartalmazza a 2D indexek által támogatott lekérdezéseket is.

Lekérdezések, amelyek visszatérési értéke point, line és/vagy polygon	Lekérdezés feltételének geometriája	Felszín típusa	Lekérdezés számítási egysége	Ezzel az index-el támogatott
`{ $geoWithin : { $geometry : <GeoJSON Polygon> } }`	poligon	sphere	méter	2dsphere
`{ $geoIntersects : { $geometry : <GeoJSON> } }`	pont, vonal vagy polygon	sphere	méter	2dsphere
`{ $near : { $geometry : <GeoJSON Point>, $maxDistance : d } }`	pont	sphere	méter	2dsphere Az index kötelező!
Lekérdezések, amelyek visszatérési értéke minden esetben point	**Lekérdezés feltételének geometriája**	**Felszín típusa**	**Lekérdezés számítási egysége**	**Ezzel az index-el támogatott**
`{ $geoWithin : { $box : [[x1, y1], [x2, y2]] } }`	négyzet	flat	flat egység	2d
`{ $geoWithin : { $polygon : [[x1, y1], [x1, y2], [x2, y2], [x2, y1]] } }`	poligon	flat	flat egység	2d
`{ $geoWithin : { $center : [[x1, y1], r], } }`	körkörös régió	flat	flat egység	2d

Lekérdezések, amelyek visszatérési értéke point, line és/vagy polygon	Lekérdezés feltételének geometriája	Felszín típusa	Lekérdezés számítási egysége	Ezzel az index-el támogatott
`{ $geoWithin : { $geometry : <GeoJSON Polygon> } }`	poligon	sphere	méter	2dsphere
`{ $geoWithin : { $centerSphere : [[x, y], radius] } }`	körkörös régió	sphere	radián	2d 2dsphere
`{ $near : [x1, y1], $maxDistance : d }`	pont	flat / flat units	flat egység	2d Az index kötelező!

5. ábra- Térbeli lekérdezések kompatibilitása[5]

Within (Bennfoglalás) és Intersects (Metszés)

A bennfoglalást a $geoWithin operátor valósítja meg, amely visszaad minden olyan geometriát, amely teljes mértékben az adott poligonon belül van. Korlátja, hogy multipolygon esetén nem használható.

A metszést a $geoIntersects operátor valósítja meg, visszaad minden olyan geometriát, mely metszi az adott geometriát (de nem a két geometria metszetét adja vissza).

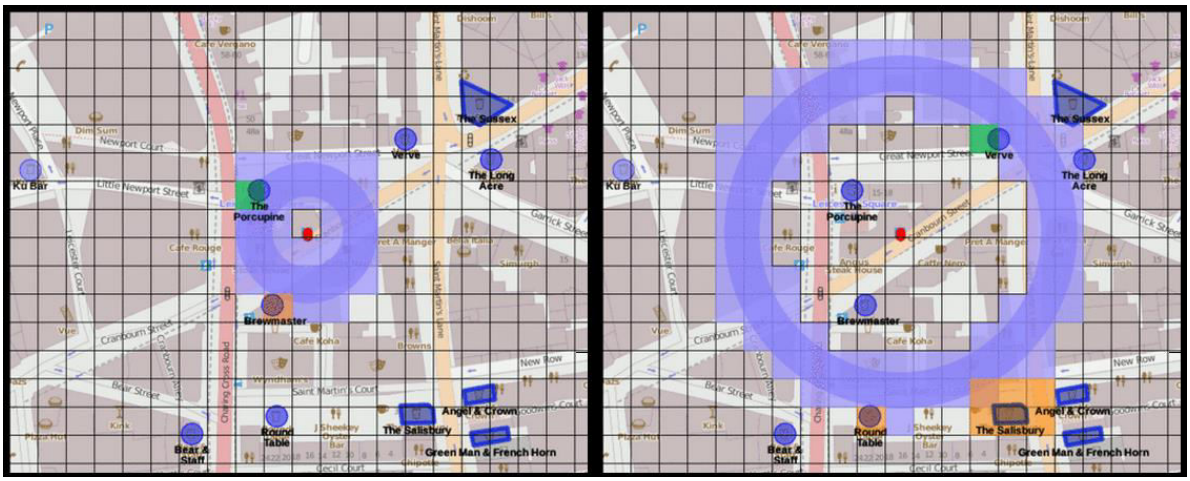
A bennfoglalás és metszés műveletek indexelés nélkül is működnek.

Közelség

A $near illetve a $geoNear operátorok valósítják meg, segítségükkel a közeli geometriák kérdezhetők le. Egy adott sugarú körön belül található objektumok, vagy egy tetszőleges

[5] http://docs.MongoDB.org/manual/reference/operator/query-geospatial/

darab legközelebb eső objektumot kaphatunk vissza (automatikusan sorba rendezve) a megadott gyűjteményből. A $near művelet egy közelségi keresés, ami valójában $within lekérdezéseket hajt végre koncentrikusan növekvő fánk (tórusz) alakzatokon (kör alakú poligonok kör alakú lyukakkal). A 6. ábrán láthatjuk, ahogy kezdetben a kiinduló pont körüli covering cellákban keresi meg a geometriákat, majd a későbbi állapotokban az egyre távolabb cellákban keres. Mért nem kérdezzük akkor csak le egy egyszerű $within segítségével a közelben levő elemeket? A $within és a $near műveletek közti különbség abban van, hogy a $near közelség szerint sorba rendezve adja vissza a dokumentumokat, hiszen a fánkokat sorban bejárva a megtalált geometriák is sorban kerülnek bele az eredmények tömbjébe. Ebből kifolyólag természetesen a $within a gyorsabb művelet, de itt a sorbarendezés nem garantált.



6. ábra- Koncentrikusan növő fánkok. Balra egy kezdeti, jobbra egy későbbi állapot.[6]

Ha nem csak a közelben levő objektumokra vagyunk kíváncsiak, hanem azok távolságára is, annak meghatározása kissé nehézkes, ehhez mindenképpen aggregációt kell használnunk. Az aggregation framework része a $geoNear, ezt kell használnunk, ha az adott geometriákon kívül szükségünk van olyan diagnosztikákra, mint a distance, vagy a location matched. A következő példa visszaadja a legközelebbi folyót és annak távolságát a (47,5;19) ponthoz képest:

6 http://blog.MongoDB.org/post/50984169045/new-geo-features-in-MongoDB-2-4

```
db.rivers.aggregate([ { $geoNear: { near: [19, 47.5],
distanceField: "distance", spherical: true, limit: 1 }}]);
```

A visszakapott objektum a legközelebbi objektum lesz, kiegészítve a „distance” nevű mezővel, mely a távolságot tartalmazza radián fokban, ezt kell átváltanunk kilométerre. Az átváltás egyik módja, hogy az aggregációban megadunk még egy paramétert: „distanceMultiplier:6371000”, ahol 6371000 a Föld sugara, amely méterben megadva. A másik lehetőségünk egy MongoDB „bug” kihasználása, ugyanis, akkor is méterben kapunk eredményt, ha a near paramétert a következőképpen adjuk meg: „`near: { type: "Point", coordinates: [19, 47.5]}`”. Ez esetekben az egyenlítői sugarat veszi a számítás alapjául a függvény. A spherical mező azt dönti el, hogy a MongoDB hogyan számolja ki a távolságot két pont között. Amennyiben a mező értéke igaz, akkor szférikus, különben pedig euklideszi geometriát használ a távolság kiszámítására. A 2dsphere index esetén muszáj a spherical-nal true értéket adni (ennek default értéke false), különben a lekérdezés nem fog lefutni. A limit-tel szűrjük az eredményhalmazt egy eleműre, így csak a legközelebbi elemet kapjuk meg.

Egy pont és egy meghatározott objektum távolságának kiszámítására is van lehetőségünk, azonban ehhez már egy összetettebb aggregációra van szükségünk. A $geoNear aggregációkat szűkíthetjük a query mezővel, mely szintaxisa a klasszikus MongoDB lekérdezésekével egyezik meg, így megadhatjuk az adott objektum nevét/id-ját, melyre keresni szeretnénk.

```
db.rivers.aggregate([ { $geoNear: { near: [39, 7.5],
distanceField: "distance", spherical: true, limit: 1,
query:{"properties.NAME":"Danube"}  } }]);
```

A bevezetőben tárgyalt ortodromikus távolságok kiszámítása helyett a MongoDB egy saját módszert alkalmaz, amelyet **n-vector**-nak hívunk. Ennek lényege, hogy a latitude és longitude koordinátákat (szélességi és hosszúsági fokokat) egy három elemű vektorra cseréljük. Ez a vektor a következő:

$$n^e = \begin{bmatrix} cos(latitude)cos(longitude) \\ cos(latitude)sin(longitude) \\ sin(latitude) \end{bmatrix}$$

Ezt a vektort használva a korábbi képletek leegyszerűsödnek.

Spherical law of cosines: $\Delta\sigma = arccos(n_a \cdot n_b)$

Haversine: $\Delta\sigma = arcsin(|n_a \times n_b|)$

Ennek a két képletnek a kombinálásából kapjuk meg a Vincenty formulát:

$$\Delta\sigma = arctan\left(\frac{|n_a \times n_b|}{n_a \cdot n_b}\right)$$

ahol n_a és n_b a két pontot reprezentáló *n*-vector.

3. Probléma

A relációs adatbázisok megfelelnek majdnem minden adatbázis feladat elvégzésére. Ebben a fejezetben három témakörbe csoportosítva írom le, hogy mik a relációs adatbázisok gyengeségei és azok milyen esetekben jelentkeznek.

„impedance mismacth"

A procedurális nyelvek és a relációs adatbázisok lekérdező nyelvei különböző szemantikus alapokon és optimalizációs stratégiákon alapulnak. Az objektum-orientált paradigma szoftverfeljesztési, a relációs paradigma viszont matematikai alapelveken nyugszik. Ezen különbségeket nem hivatalosan „impedance mismacth"-ként szokás hívni, melyek a következőek: Imperatív programok a deklaratív lekérdezésekkel szemben, fordító optimalizálás a lekérdezés optimalizálással szemben, algoritmusok és adatszerkezetek szemben relációkkal és indexekkel, szálak a tranzakciókkal szemben, a hiányzó adat reprezentálására NULL pointerek a NULL értékkel szemben, és különböző megközelítési módjai a modularitásnak és információ elrejtésnek *(Cook, 2006)*.

Az „impedance mismatch" egyik megoldatlan problémája, hogy amíg az objektum-orientált nyelvekben lényegében primitív típusok, tömbök és objektumok segítségével modellezünk minden entitást, addig a relációs adatbázisok táblái skalár értékeket kapnak (String, int, stb…). Természetesen léteznek eszközök, amelyek leképezik a két rendszer közt az adatokat, ezeket Objektum-relációs leképező rendszereknek (ORMs) hívjuk. Ezen eszközök azonban nem adnak igazi megoldást a problémára, mert a programozók beavatkozását igénylik ahhoz, hogy a leképezések ésszerű módon jöjjenek létre.

Séma megkötések

A sémák által nyújtott előnyök gyengeséggé válnak, ha olyan adatokkal dolgozunk, melyek struktúrája nem előre definiált, vagy nehezen illeszthető relációs táblákba. Ha egy relációs adatbázisban szükség van egy új oszlopra valamelyik táblában, akkor változtatni kell a sémát. Tipikusan a nem kötött sémájú adatok a felhasználó-központú adatok, melyek a rendszer működése közben jönnek létre. De egy térinformatikai adatbázisban is előfordulhat, hogy a különböző geometriákhoz különböző struktúrájú metaadatok

kapcsolódnak. Például van egy user táblánk, ahol a user azonosítóját, keresztnevét és vezetéknevét tároljuk. Bizonyos esetekben azonban ezen felül szeretnénk még egyéb adatokat is tárolni (pl. középső név, egy külső azonosító, ami csak egy adott kampányban résztvevőknek van). Ekkor az egyéb adatokra vagy felveszünk külön oszlopokat, melyek nagyon sok helyen NULL értékűek lesznek, és csak néhány helyen lesznek kitöltve, vagy egy külön táblában tároljuk őket, ekkor viszont minden hozzáférés ezen adatokhoz egy extra join-nal jár. Egy harmadik megoldás lehet, ha csak egyetlen oszlopot veszünk fel, melybe ömlesztve, egy text típusú oszlopba JSON-ként tároljuk az egyéb adatokat. Ennek hátránya, hogy így az adatok nehezen indexelhetőek, és az adatbázisból nem érhetjük el külön-külön őket, csak egy programozási nyelv segítségével a szerver oldalon. Emellett egy rosszul megtervezett relációs adatbázis szükségtelenül komplex lehet, és korlátozhatja a további fejlesztéseket, így egy jó adatbázis struktúrájának megtervezése sok időt igényel. Emellett a relációs táblák létrehozására használt modellezési folyamat az egyed-kapcsolat modellezés, amely nem feltétlenül tükrözi a valóságban szereplő adatokat.

Skálázhatóság

Egy rendszer megtervezésénél szempont kell, hogy legyen a skálázhatóság. Ha az adatok túlnőnek egy szerver kapacitásán, akkor azt muszáj darabolni. Ez optimális esetben lineáris skálázhatóságot jelent, vagyis a processzorok duplázásával duplájára nő a terhelhetőség is. Ez a gyakorlatban azonban elérhetetlen. Relációs adatbázisok használata esetében oda kell figyelnünk rendszerünk tervezésekor, implementációjakor és a hardverek megválasztásakor, hogy az jól skálázható legyen.

Továbbá az egyes műveletek több szerver közti végrehajtása, mint például az outer join, amelyek több táblával is dolgoznak, problémás lehet. Egy relációs adatbázis nem tud egy lekérdezést darabokra szedni, különböző szervereken lefuttatni a részeket, majd az egyesített eredményt elküldeni. Ez a probléma akkor léphet fel, ha nagy adatokkal dolgozó analitikus lekérdezéseket (pl. statisztikák) kell végeznünk.

A vízszintes skálázás ugyan megvalósítható a relációs adatbázisokkal, de általában ez olyan bonyolult eszközök használatával érhető el, mint a Storage area network (röviden: SAN, magyarul: tárolóhálózat).

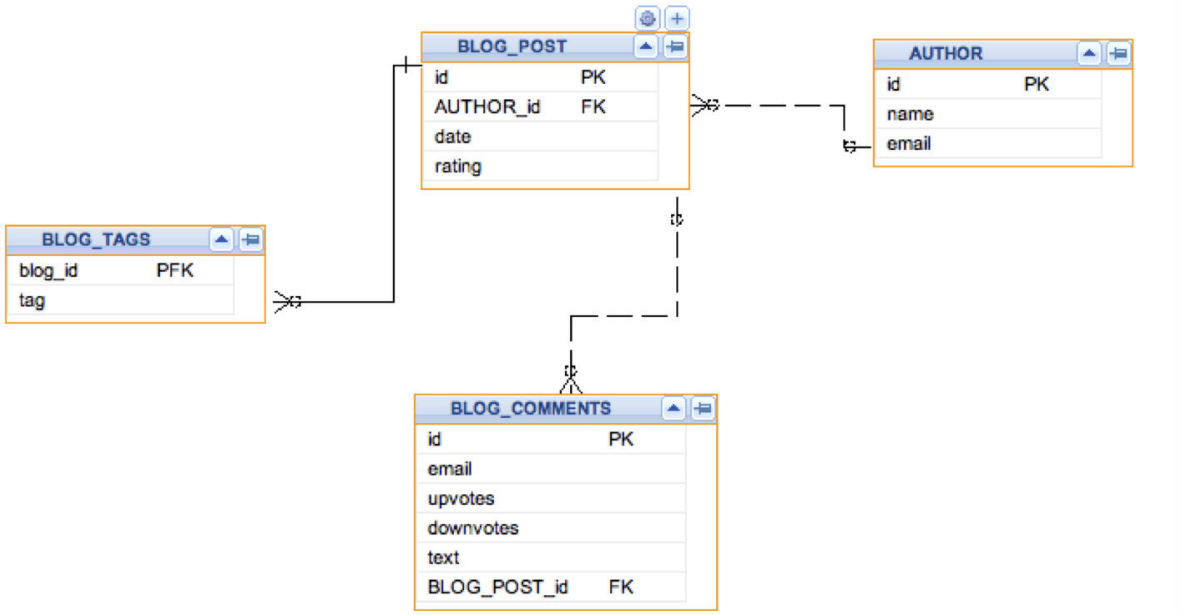
A relációs adatbázisok nem teljesítik a paticionálás-tűrést, amely bizonyos vízszintesen skálázott rendszerekben fontos lehet.

4. Megoldási terv

A programozási nyelvek és adatbázisok közti különbségek, a kötött sémák használata és a bizonyos szint feletti skálázhatóság hiánya vezettek a NoSQL adatbázisok vizsgálatához.

„impedance mismatch"

Vegyük példának a blog posztokat. Legyen egy "blog post" osztályunk a programozási nyelvünkben, melynek van egy bizonyos struktúrája. Ahhoz, hogy ezt a struktúrát követve tárolni tudjuk az ilyen típusú adatainkat az adatbázisban, egyetlen objektum is szétosztva kerül tárolásra több tábla közt, ahogy azt a 7. ábra szemlélteti.



7. ábra-A blog post osztályhoz tartozó táblák[7]

A feljesztőnek tudnia kell, hogy hogyan konvertálunk egy "Blogpost" objektumot ahhoz, hogy a táblákban elhelyezhető legyen. Egy dokumentum-tároló adatbázisban nincs szükség bonyolult leképezésekre, szinte pontosan ugyanúgy tárolhatók az adatok, mint ahogy a programban reprezentáljuk őket. A 7. ábrán található struktúra alapján a

7 http://blog.mongolab.com/2012/08/why-is-MongoDB-wildly-popular/

következőképpen hozhatnánk létre egy MongoDB dokumentumot, amelyben az objektum összes adata tárolható:

```
{
     _id: 1234,
     author: { name: "Bob Davis", email : "bob@bob.com" },
     post: "In these troubled times I like to …",
     date: { $date: "2010-07-12 13:23UTC" },
     location: [ -121.2322, 42.1223222 ],
     rating: 2.2,
     comments: [
        { user: "jgs32@hotmail.com",
          upVotes: 22,
          downVotes: 14,
          text: "Great point! I agree" },
        { user: "holly.davidson@gmail.com",
          upVotes: 421,
          downVotes: 22,
          text: "You are a moron" }
     ],
     tags: [ "Politics", "Virginia" ]
 }
```

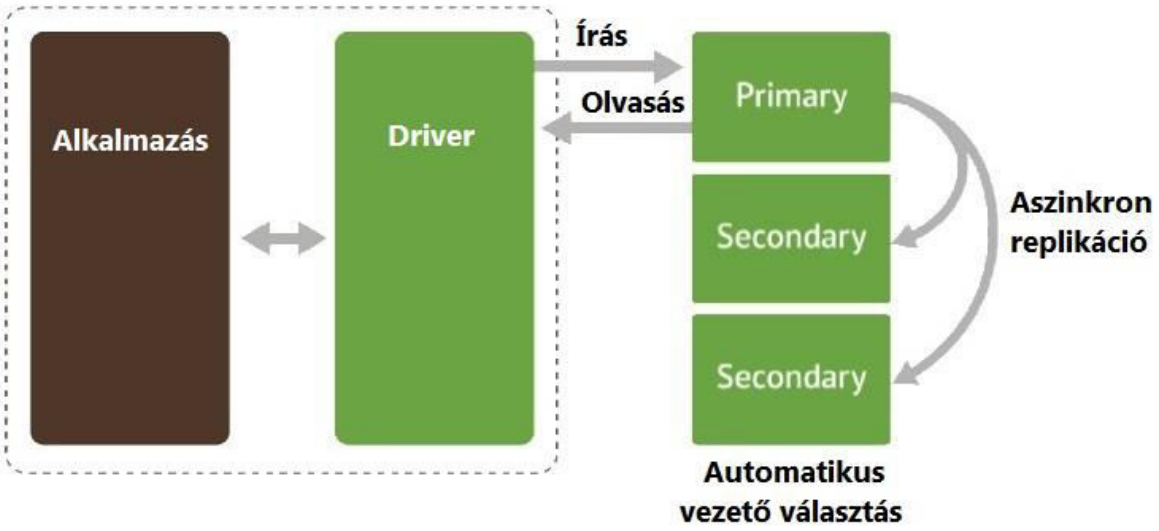
Séma megkötések

Mivel a MongoDB BSON formátumban tárolja az adatokat, ezért gyakran tárolhatunk egyetlen gyűjteményben olyan adatokat (tömbök, objektumok), amelyek számos táblát igényelnének egy relációs adatbázisban. Erre jó példa az imént említett Blog post. A MongoDB esetében az adatbázis megtervezése nem igényel különösebb feladatot. A séma migrációk nagyon egyszerűen elvégezhetőek. Egy adatbázis szerkezetének módosítása szinte automatikus. Ha például a városok gyűjteményhez hozzá szeretnénk adni egy középhőmérséklet mezőt, hozzáadva az attribútumot csak frissítjük azt. Ez egy jól megírt kódban a szerver oldalon a továbbiakban nem okozhat problémát, ahol az attribútum nem létezik, ott NULL értéket kapunk.

Skálázhatóság és particionálás-tűrés

A MongoDB teljesíti a particionálás-tűrést. A szerverek replikációját konfigurálhatjuk úgy, mint egy „primary-secondary” rendszer, ahol egyetlen elsődleges szerver van, a többi szerver pedig másodlagos. A primary szerver fog minden írás és log műveletet kezelni, a secondary szerverek ezt másolják le. A secondary szerverek képesek a többi secondary

server műveleteinek olvasására, ezáltal levéve a terhet a primary szerverről. Ez látható az 8. ábrán. Ez a nagyobb megbízhatóság mellett a kliens alkalmazások olvasási műveleteinek teljesítmény növekedésével is jár. Ez nagy hatással lehet a teljesítményre, azonban magában hordozza a lehetőségét annak, hogy a kliensek felé küldött adat nem mindig lesz a legfrissebb. Ennek oka az aszinkron replikációs modell, amely biztosítja, hogy egy secondary szerver folytathassa működését, ha egy másik replika kiesik, de nem garantálja az ACID tulajdonságok közül a konzisztencia teljesülését. Amikor egy primary szerver működése leáll, akkor egy szavazás dönti el, hogy mely secondary vegye át a helyét. A 2.6-os MongoDB 12, a 3.0-ás verzió 50 darabban korlátozza a replikák számát.



8. ábra-Több replikából álló MongoDB fürt működése[8]

A MongoDB támogatja az „auto-sharding”-ot, azaz az automatikus vízszintes skálázást, ami azt jelenti, hogy natív módon és automatikusan terjeszti szét az adatot tetszőleges számú szerver közt, anélkül, hogy az alkalmazásnak ismernie kellene a szerverpark fizikai felépítését. A terhelések automatikusan egyensúlyba jönnek a szerverek közt, és amikor egy szerver leáll, az nem okoz zavart az alkalmazásban. Az egyértelmű teljesítmény-növekedés ellenére a replikáció jobban ajánlott a sharding-nál, mert az biztosítja a

8 http://www.slideshare.net/onomi/onomi-mongodb-introduction

megbízhatóságot is.

A lekérdezéseket darabokra szedhetjük, és különböző szervereken lefuttatva azokat, összesíthetjük a részeredményeket. Erre a MapReduce programozási modell ad lehetőséget.

Célok

A NoSQL adatbázisok kínálhatnak előnyöket a klasszikus relációs adatbázisokhoz képest, azonban nem feltétlenül rendelkeznek mindennel, ami egy relációs adatbázistól elvárható. Diplomamunkám célja egy klasszikus relációs adatbázis, a PostgreSQL összevetése egy NoSQL adatbázissal, a MongoDB-vel, ezáltal feltárni azon területeket, amelyekben a MongoDB hatékonyabban használható a relációs adatbázisoknál vektoros térinformatikai adatok feldolgozására. Két, egymáshoz hasonló szerver alkalmazást készítek, melyek az imént említett adatbázisokat használják. Ennek célja, hogy a két rendszer térinformatikai funkcióinak sebességét lemérjem, majd összehasonlíthassam.

A MongoDB egy olyan rendszer, mely bizonyos problémákra a felépítéséből adódóan megoldásokat biztosít, emellett sok térinformatikai adattípust kezel és elégséges számú térinformatikai funkcióval rendelkezik. Ezen tulajdonságai miatt egy jó alternatívája lehet a relációs adatbázisoknak egy GIS rendszeren belül.

Ahogy a relációs adatbázisok nem képesek mindenre, melyek a NoSQL adatbázisok vizsgálatát indokolják, természetesen ez visszafelé is igaz. A MongoDB egy gyorsan működő adatbázis, azonban nem teljesíti a rendelkezésre állást, nem teljesen ACID, ezért nem alkalmazható kritikus rendszerekben. A MongoDB térinformatikai lekérdezései limitáltak. Ezért munkám része egy olyan térinformatikai funkció megvalósítása is, amely csak relációs adatbázisokban lett eddig implementálva. A választásom a lineáris hivatkozási rendszerre esett. Ennek célja elsősorban az, hogy megvizsgáljam, mennyire körülményes saját függvények létrehozása egy NoSQL adatbázisban. Ezen felül azt is vizsgálom, hogy az architektúra mely részén hatékonyabb egy adott feldolgozó művelet megvalósítása. Itt a futási idő mellett figyelembe kell venni a kód implementálhatóságát is. Ezen vizsgálat célja egy optimális megvalósítás a saját funkciók implementálásához.

A diplomamunka tehát két feladatra bontható. Az egyik feladat megtervezni és implementálni relációs és NoSQL adatbázisokat használó szerver alkalmazásokat. Ezen adatbázisokban térinformatikai adatok találhatóak, melyekkel az alapvető térbeli műveleteik mellett olyan, általam implementált műveleteket is végezhetünk, mint az LRS függvények. Az így létrehozott eszközök a kutatásomról készült beszámoló elkészítéséhez elengedhetetlenek. A második, legfontosabb feladat a beszámolóm elkészítése, mely tartalmazza a témához kapcsolódó szakirodalmat, a tervezés és fejlesztés menetét, és a mérés eredményeket, melyeket az implementált eszközök segítségével kaptam meg. Célom, hogy valós képet kapjunk arról, hogy a MongoDB mennyire lehet alternatívája egy relációs adatbázisnak egy térinformatikai adatokat kezelő rendszer komponenseként.

5. Rendszer implementáció

Környezet

Architektúra

Habár terminálból is futtathatóak a lekérdezések, az adatbázisokat egy szerver oldali alkalmazás segítségével szeretném elérni. Ez azért jó, mert így a lekérdezések írása helyett, csak kéréseket kell küldeni a szervernek, aki azokat továbbítja az adatbázisok felé. Ezen felül a szerver oldalon rengeteg dolgot tudok definiálni, amire az adatbázisban nem feltétlenül van lehetőségem. A terminált a külső eszközök használatakor, például adatbázis importáláskor is kellett használnom. Ezen felül a MongoDB esetében több műveletet kellett terminálból végeznem, például a gyűjtemények indexelésekor vagy az LRS függvények implementálásakor.

El kell dönteni, hogy mely GIS komponensek legyenek az alkalmazások részei. Térinformatikai adatbázisokra mindenképp szükség van, ők a vizsgálat tárgyai. Már létező Server GIS és web map server szoftverekre nincs szükség a vizsgálat szempontjából, hiszen a létrehozott alkalmazások ezek feladatait fogják ellátni. Az alkalmazások lényegében egy Web Feature Service típusú web map server-nek felelnek meg, mert az adatokat szöveges, feldolgozható formában jelenítik meg. A GIS könyvtárak és kiterjesztések használatára is szükség van, mert az általam választott eszközök közül több nem rendelkezik térinformatikai támogatottsággal. Ezen felül használtam desktop GIS kategóriájú eszközöket is, melyek adatok importálására, létrehozására és módosítására használhatóak. A web GIS client szoftverek az adatok megjelenítéséért, böngészéséért felelősek, a mobile GIS szoftverek pedig mobil eszközökre valók, ezekre nem koncentráltam a diplomamunka elkészítésekor.

Node.js

A Node.js egy olyan szoftverrendszer, amellyel JavaScript nyelven írhatóak aszinkron, skálázható, eseményvezérelt alkalmazások. Segítségével szerver oldali alkalmazásokat készíthetünk, ahol a túlterhelés minimalizálva és a skálázhatóság maximalizálva van. Rengeteg eszköz található a Node.js-hez, mely segíti a fejlesztést. A különböző csomagok a

node package manager (NPM) segítségével könnyen telepíthetőek, ami a Node.js-el együtt települő csomagkezelő.
A kliens oldali alkalmazás megírására nem akartam nagy hangsúlyt fektetni, mivel az nem szerepelt a céljaim közt. Emiatt a jade nevű template nyelvet választottam a user interface megvalósításához, amely segítségével HTML kódokat generálhatunk. A jade egy csomagja a Node.js-nek. Ennek segítségével egy böngészőn keresztül könnyedén kommunikálhatunk az alkalmazással.

Ezen kívül szükség van még az Express.js csomagra, amely egy csupasz Node.js alkalmazást felruház minden olyan tulajdonsággal, amely egy webszervertől elvárható.

A következő Node.js csomagok mindkét alkalmazás megírásakor telepítve lettek:

Express.js version 4.0 - Webes keretrendszer

jade version 0.31.0 - HTML sablon leíró

morgan version 1.0.0 - Logger (sztem kidobható)

body-parser version 1.0.1 - express-be beépíthető értelmező, pl. JSON-t értelmez

geolib version 2.0.14 - Node.js csomag, biztosít néhány alapvető térinformatikai műveletet

Az alkalmazással böngészőből vagy terminálból REST API-n keresztül tudok kommunikálni. Egy REST achitektúra kliensekből és szerverekből áll. A kliensek kéréseket küldenek a szerverek felé, a szerverek ezen kéréseket dolgozzák fel és a megfelelő választ küldik vissza a kliensnek, akitől a kérés jött. Az alkalmazásaimban gyakran elég egy URL megadása a böngészőbe ahhoz, hogy elküldjek egy kérést, bizonyos esetekben (például ha bonyolultabb paraméterezést igényelne így küldeni az adatokat) pedig a jade által generált HTML kód segítségével tudok kérést küldeni.

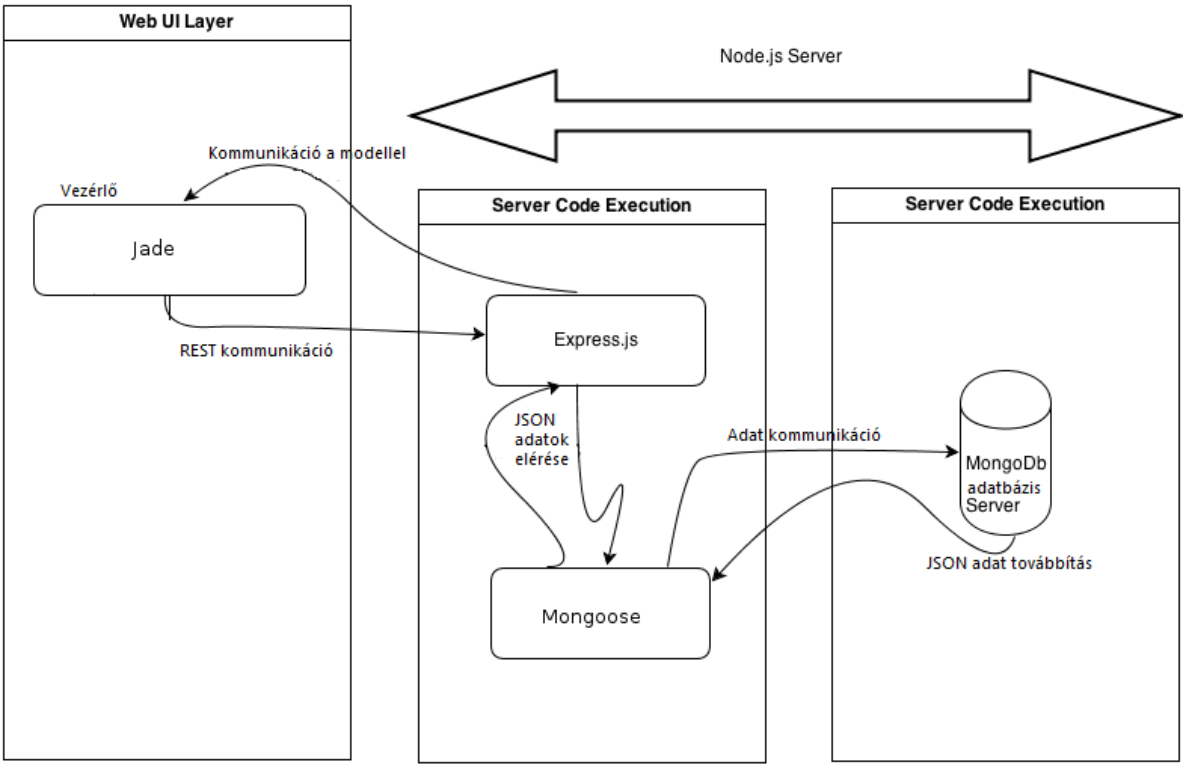
MongoDB

A MongoDB által használt BSON objektumok könnyen dekódolható JSON objektumokká. A JSON (javaScript Object Notation) egy nyitott, ember és gép által is könnyen olvasható

adat, mely az XML mellett a modern webes adatcsere fő formátuma. A JSON támogatja a fő adatformátumokat, és a primitív típusok mellett a tömbök és objektumok kezelését is. A MongoDB térinformatikai objektumai a GeoJSON objektumok, amely egy nyílt szabvány a térinformatikai objektumok leírására azok nem térinformatikai adataival együtt JSON objektumokként. A JSON a JavaScript nyelv szintaxisát használja, emiatt nagyon könnyen objektumokká alakíthatók a JavaScript nyelvben. Emiatt esett a Node.js rendszerre a választásom a szerver oldal megvalósításakor. A Node.js MongoDB-vel való integrációja több csomag segítségével is elérhető, két hivatalos csomag létezik, a MongoDB és a Mongoose. Több Node.js MongoDB Driver eszközt (MongoDB, Mongolian, Mongoskin) is kipróbáltam, azonban ezek nem rendelkeztek számomra elegendő függvénykészlettel. A választásom a Mongoose csomagra esett, amely tartalmaz séma alapú funkciókat, aszinkron validálást, objektum életciklus menedzsmentet, pszeudo-join műveleteket, és gazdag lekérdezés builder támogatottságú. A Mongoose megköveteli, hogy létrehozzunk séma definíciókat, ezzel elveszítve a schemaless előnyét a MongoDB-nek. Azonban a séma típusok közt szerepel a Mixed, ami egy "anything goes" séma típus, azaz az értéke bármi lehet, habár ezzel a Mongoose elveszíti az automatikus detektáló képességét. Használatára egy példa:

```
var Any = new Schema({ any: Schema.Types.Mixed });
```

A MongoDB adatbázis használó alkalmazásom architektúrája a 9. ábrán látható.



9. ábra-MongoDB adatbázist használó szerver architektúra

Az alábbi eszközök lettek telepítve:

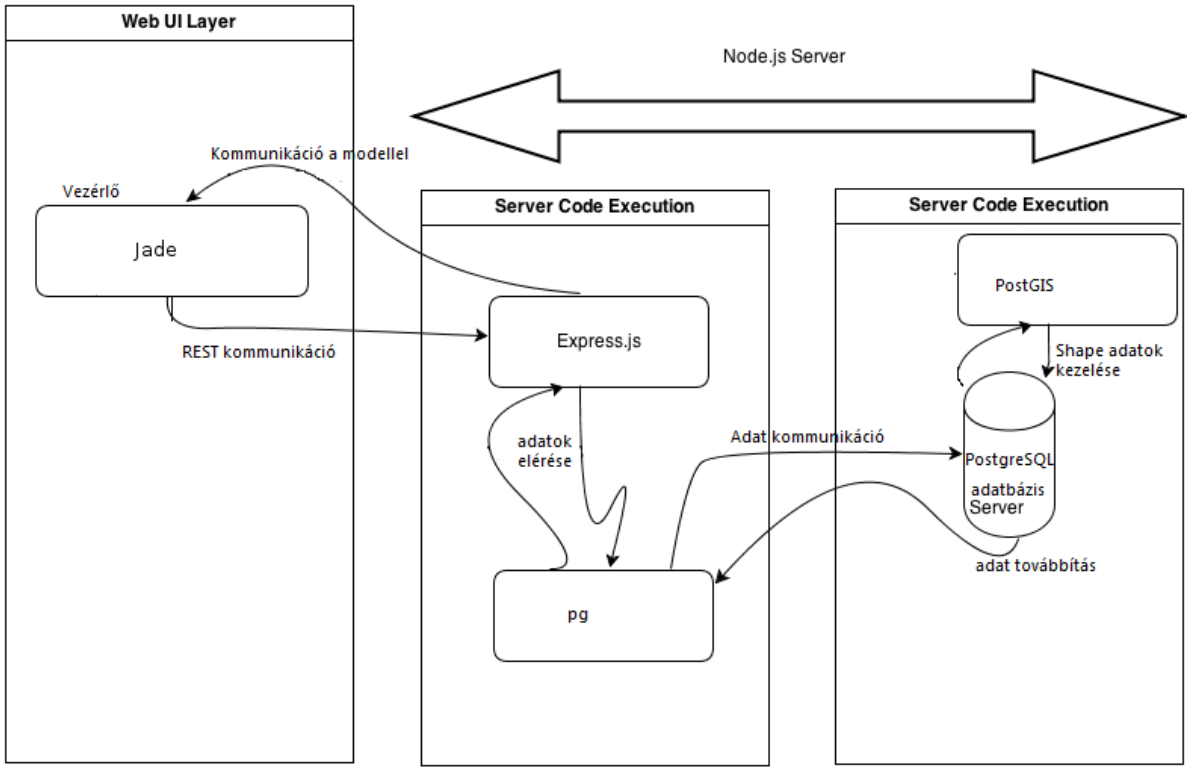
MongoDB 2.6 - adatbázis

Mongoose version 4.0.2 - Node.js csomag, hozzáférést biztosít a MongoDB adatbázishoz

PostGIS

Az Oracle, habár integrálható a Node.js-el, azonban az így kapott rendszer a térinformatikai adatokat képtelen volt feldolgozni a node-oracle package segítségével. Emellett fontos szempont az is, hogy az Oracle nem open source. Így esett a választás a PostgreSQL adatbázisra, mely a PostGIS térinformatikai kiterjesztéssel reprezentálja kutatásomban az SQL adatbázisokat, melynek több ígéretes tulajdonsága is van. A térinformatikai adatbázisok közül feltehetőleg a PostGIS-nek van a legnagyobb felhasználói bázisa. Bizonyos, csak pontokra kiterjedő vizsgálatok *(Matuszka, 2014)* alapján a leggyorsabb az SQL adatbázisok közül. Lekérdező nyelve az SQL.

A PostGIS térinformatikai objektumai a GIS objektumok, amely szabványt az OpenGIS Consortium (OGC) definiálta. A PostGIS kiterjeszti a standard szabványt 3DZ,3DM és 4D koordináták használatával, mi ezeket nem használjuk. Az OpenGIS két módját adja meg a tér-adatok leírásának: a Well-Known Text (WKT) forma és a Well-Known Binary (WKB). Mindkettő tartalmazza az információkat a geometria típusáról és koordinátáiról. A két típus közti eltérés, hogy a WKT egy ember számára olvashatóbb típus. Hasonlóan a GeoJSON formátumhoz, itt is megjelennek a LAT és LON koordináták. A két típus közt az átjárhatóság igen egyszerű, bármelyiket a másikká transzformáhatjuk egy egyszerű paranccsal, például a WKB-ből WKT-vé alakítás a `ST_AsText(geom)` paranccsal történik.



10. ábra-PostGIS adatbázist használó szerver architektúra

Az architektúrák kialakításakor fontos szempont volt a mérések torzítatlansága miatt, hogy a szerver oldalon a legkevésbé kelljen változtatni. Mivel a Node.js PostgreSQL-el való integrációja elérhető a PG csomag segítségével, ezért lényegében csak az adatbázist, és a Mongoose csomagot kellett lecserélni a PG csomagra ebben az architektúrában. A 10.

ábrán látható ezen alkalmazás architektúrája. Az alábbi eszközök lettek telepítve:

PostgreSQL version 9.4.1 - adatbázis

PostGIS version 2.1.5 r13152 - adatbázis eszköz, térinformatikai támogatást biztosít

shp2pgsql version 1.1.5 - importáló eszköz

PG version 4.2.0 - Node.js csomag, hozzáférést biztosít a PostgreSQL adatbázishoz

Adatok

A térinformatikai adatokkal a valóságot sokféle aspektusból közelíthetjük. El kell – az adott feladattól függően – dönteni, hogy a valóság mely adatai fontosak számunkra. Ezen adatokat többféleképpen is leírhatjuk, használhatunk vektoros és raszteres, 2 vagy 3 dimenziós adatokat. A raszteres adatok esetén minden képpontot eltárolunk, emiatt hatalmas adatmennyiségek születnek. Ezen adatok nem interpretálhatóak, nehezen kapcsolhatók adatbázisrendszerekhez.

A vektoros adatok felépítésükben közelebb állnak logikailag az adatbázisrendszerekhez, ezért könnyen összekapcsolhatóak azokkal. Kezelhetjük a térbeli elemeket objektumokként, melyeknek koordinátáik vannak. Az objektumainkat könnyen csoportosíthatjuk rétegek szerint, vagy logikailag elkülönítve egymástól (épületek, határok, vizek, erdők, utak, stb…).

Minden térinformatikai objektum geometriáját a Latitude és Longitude koordinátáival reprezentálok, így valójában az adataim 3 dimenziósak, hiszen a MongoDB közelség fejezetben leírtak szerint könnyen x, y és z koordinátákká transzformálhatjuk az adatokat.

Ugyanazon adatokat használtam mindkét adatbázis esetén. Az adatokat átkonvertáltam shape fájlokból JSON formátumúakra is, hiszen mindegyik rendszer csak az egyik típusú adatot képes kezelni. A shape fájlok GeoJSON formátumúvá konvertálásához a Geospatial Data Abstraction Library egyik Utility programját, az ogr2ogr eszközt használtam. Az adatok dátuma a nemzetközi WGS84 volt, melynek megfelelően az SRID számuk a 4326

volt. A PostGIS adatbázisba WKB formában kerülnek be az adatok importálás után.

Az adatok különböző típusúak voltak:

1, európa országainak, nagyobb folyóinak és városainak koordinátáit tartalmazták

2, Magyarország épületei, vasutai, útjai, természeti területei, füldművelési területei, nevezetes pontjai[9].

A shape fájlok PostgreSQL adatbázisba történő importálásához először egy sql fájlt kell létrehoznunk. Ezután ezt az sql fájlt importálhatjuk. Ez a két parancs sorban a következőképp néz ki:

```
shp2pgsql -s 4326 cities.shp cities spatial >cities.sql

psql -f cities.sql -h localhost -d spatial -U postgres
```

A cities lesz a tábla neve, localhost a host, spatial az adatbázis neve, postgres a user, a 4326 alapján pedig a WGS84 dátum lesz beazonosítva. A két lépést össze is fűzhetjük a unix terminálban. A shape fájlok beimportálása a következőképp néz ki egy lépésben PostgreSQL adatbázisba, ha karakterkódolást is szeretnénk megadni:

```
shp2pgsql -s 4326 -W "latin1" -d -g geom -I cities.shp cities |
psql postgres
```

Emellett léteznek grafikus eszközök is a beimportálásra, mint például a shp2pgsql-gui.

A JSON fájlok beimportálása a MongoDB adatbázisba a következőképp érhető el, ahol spatial az adatbázis neve, a countries pedig a kívánt gyűjteményé:

```
mongoimport --db spatial --collection countries --file
countries.json --jsonArray
```

9 http://download.geofabrik.de/

A következő táblák/gyűjtemények lettek részei a rendszernek:

Cities – Európa nagyobb városai(Shape 623KB, Json 846KB)

Countries – Európa 20 országa (Shape 190KB, Json 305KB)

Rivers – Európa folyói (Shape 1.9MB, Json 3.1MB)

Pontok – Magyarország (Shape 1.8MB, Json 19.1MB)

Waterways – Magyarország vizei (Shape 10.3MB, Json 15.5MB)

A MongoDB gyűjteményeknek kell sémát definiálnunk. A MongoDB sémák és a PostGIS táblák a függelékben találhatóak egy rövid leírással.

6. Mérések

Hardver

A mérésekhez egy Lenovo ThinkPad SL510 laptopot használtam, melyben Intel Core 2 Duo 2.1GHz teljesítményű processzor, és 4GB 800MHz sebességű belső memória volt.

A hardverre telepített operációs rendszer egy Ubuntu 12.04 (precise pangolin) Linux rendszer volt, melynek termináljából a UNIX parancsokat és az adatbázisokat is elértem.

Lekérdezések

A lekérdezéseket igyekeztem úgy összeválogatni, hogy azok különbözősége mellett különböző méretű táblákat/gyűjteményeket használjanak, illetve, hogy a visszakapott adat mérete is eltérő legyen. Ezen felül igyekeztem olyan eseteket is vizsgálni, mikor az objektumok különböző távolságra voltak egymástól a térben.

Spatial bounding box megszámlálás

Számoljuk meg a geometriák számát egy gyűjteményből/táblából egy adott téglalapon belül.

Három-három vizsgálatot csináltam, ahol két esetben mindkét adatbázis ugyanazon eredményeket adta:

A (45;15) (50;20) közt hány geometria található a cities gyűjteményben/táblában? 21 darab

A (45;15) (50;20) közt hány geometria található a pontok gyűjteményben/táblában? 69872 darab

A harmadik eset azonban eltért:

A (45;15) (50;20) közt hány geometria található a waterways gyűjteményben/táblában?

A MongoDB 0, míg a PostGIS 16677 darab eredménnyel tért vissza. Ebből látszik, hogy a MongoDB esetében a $box metódus csak pont geometriákon értelmezhető, ahogy az a MongoDB fejezetben található 5. ábrán is látható volt. A többi lekérdezés azonban

ugyanúgy működött, mint a PostGIS esetében. Ugyan a $box metódus a 2D indexek esetében támogatott csak, azonban azok használata nem kötelező, így akár egyáltalán nem indexelt dokumentumokon is végrehajthatjuk, a művelet hasonló teljesítménnyel fut le, mint indexelve. A 12. és 19. ábrákon emiatt hiányzik adat a MongoDB futási idejéről.
Egy érdekes kérdés, hogy ez a művelet kiváltható-e úgy, hogy a bounding box helyett egy téglalap alakú poligont adunk meg. Ennek kényelmetlensége, hogy kettő pont helyett öt ponttal (a négyzet négy pontja plusz a kezdőpont még egyszer végpontként) tudjuk csak reprezentálni a bounding box-ot, cserébe azonban a pont mellett polyline és polygon geometriákat is kereshetünk egy adott területen.

Spatial bounding box

Kérjük le az összes geometriát egy gyűjteményből/táblából egy egy adott téglalapon belül. Három-három vizsgálatot csináltam, ahol mindkét adatbázis ugyanazon eredményeket adta (az eredményeket itt nagy méretük miatt nem közlöm):

A (45;15) (50;20) közt milyen városok találhatók?
A (45;15) (50;20) közt milyen pontok találhatók?
A (16.11;46.987) (17;48.123) közt milyen pontok találhatók?

Spatial Within

Kérjük le az összes geometriát, amely benne van egy adott geometriában. Ehhez olyan lekérdezéseket írtam, melyek azt vizsgálták, hogy mely folyók vannak benn egy adott országban. A within definíciójából adódóan csak olyan folyókat kapunk vissza, amelyek teljes mértékben az adott ország területén találhatóak, azaz nem folynak át az ország határain. Az adott ország lekérdezéséhez a Countries ISO_3DIGIT mezőjét adtam meg az országnak, mellyel egyértelműen azonosítható egy ország. Zárójelben feltüntetve látható, hogy melyik azonosító melyik országot jelenti.

Négy-négy vizsgálatot csináltam, ahol az adatbázisok eltérő eredményeket adtak. Az első három esetben egyeztek az eredmények:

HUN (Magyarország): Szamos, Tisza, Sio, Raba

LUX (Luxemburg): [] (Nincs ilyen)

POL (Lengyelország): Vistula

A negyedik esetben mást kaptam:

MongoDB

NLD (Hollandia): A multipoligont sajnos a MongoDB nem tudja a spatial within esetén használni (14., 21. és 29. ábra). Amíg például a $box műveletet polyline-okon (waterways) végrehajtva egyszerűen nem adott vissza egyetlen geometriát sem, illetve azok megszámlálására 0 eredményt adott, itt a program futása hibát adott az adatbázisból, amit a szerver oldalon kivételkezéssel kellett megoldanom. Hibaüzenetet írattam ki, ha az ország típusa nem Poligon volt.

PostGIS

NLD (Hollandia): [] (Nincs ilyen) A PostGIS rendesen tudta kezelni a multipolygont.

Vannak bizonyos esetek, mikor nem tárolhatunk minden adatot egyetlen MongoDB gyűjteményben. Ennek az oka, hogy az egyes gyűjteményekben a logikailag összefüggő adatokat szeretnénk tárolni. Ilyen a mostani eset is, nem tárolhatom a folyókat és az országokat egy gyűjteményben, azonban ahhoz, hogy lekérdezzem, hogy mely folyó van benn egy országban, vagy mely folyó folyik át egy ország határán, mindkét gyűjteményre szükségem van egyetlen lekérdezésben.

Habár a Node.js-ben van lehetőségünk referenciák használatára (ugyanis nem minden nyelvből elérhetőek), azonban ezek használatához az adatok szerkezetét meg kell változtatnunk. Ez jelen esetben igen körülményes lenne a sok dokumentum miatt, melyeket kézzel kéne kibővíteni a $ref, $id és $db mezőkkel.

Fejlesztésem első szakaszában ezen nyelvi “hiányosságot” úgy küszöböltem ki, hogy a szerver oldalon két lekérdezést végeztem. Az első lekérdezés egy GeoJSON objektumot adott vissza, melyet a második lekérdezés feltételében helyeztem el. Ennek azonban több

hátránya is van. A megvalósításkor figyelnünk kell arra, hogy a két lekérdezés egymás után fusson le, hiszen a Node.js lekérdezései aszinkron futnak le. Emellett dupla kommunikációra van szükség az adatbázis és a szerver oldal közt.

A lekérdezés sebességén javíthatunk, ha az adatbázis oldalon végezzük el a műveleteket, és csak a végeredményt küldjük vissza az alkalmazásnak. Függvények használatával elérhető, hogy ne kelljen kétszer kommunikálni az adatbázis és a szerver közt, egyetlen feltétele, hogy a függvényeinket a MongoDB-ben előre definiálnunk kell.

Az alábbi függvénnyel írhatjuk át az országon belül levő folyók lekérdezését.

```
db.system.js.save({
  _id: "within",value: function (x) {
      var $geom = db.countries.find(
        {'properties.ISO_3DIGIT' :x }, {geometry:1, _id:0}
      );
      return db.rivers.find({
        geometry: {
            $geoWithin: {
              $geometry: {
                  type: 'Polygon', coordinates:
                    $geom[0].geometry.coordinates
              }
            }
        }
      }).toArray();
   }
})
```

Ez a függvény paraméterül a vizsgált ország 3 jegyű azonosítóját várja. A visszatérési érték egy lekérdezés, amit tömbbé kell alakítani, különben magával a lekérdezéssel térne vissza, nem annak eredményével. Látható, hogy a függvény belső logikáján nem sokat kellett változtatni. A függvényt meghívhatjuk a MongoDB adatbázisból a következő paranccsal:

```
db.eval("within('<ISO_3_DIGIT>')")
```

ahol az `<ISO_3_DIGIT>` a vizsgált ország 3 jegyű azonosítója. A szerver oldali alkalmazásból természetesen ez is elérhető.

Spatial Intersection

Kérjük le az összes geometriát, amelynek van közös pontja egy adott geometriával.

Négy-négy vizsgálatot csináltam, ahol az adatbázisok egyező eredményeket adtak. Az első két lekérdezésben azt kérdeztem, hogy egy adott folyó mely országok határán folyik át (polyline intersects polygon):

Tisa intersects with: Hungary

Dnieper intersects with: Ukraine

A másik két lekérdezésben azt kérdeztem, hogy a folyók mely más folyókkal metszik egymást (polyline intersects polyline):

Tisa intersects with: Danube, Sava

Dnieper intersects with: Desna, Psel, Sozh, Berezina, Pripyat

A within analógiájára készítettem el az intersection műveletek MongoDB-ben tárolt függvényeit. Az alábbi kód meghatározza, hogy mely folyót metsz egy adott folyó.

```
db.system.js.save({
      _id: "intersect", value: function (x) {
            var $geom = db.rivers.find({
                  'properties.NAME' :x  }, {geometry:1, _id:0}
            ) ;
            return db.rivers.find({
                  geometry: {
                        $geoIntersects: {
                              $geometry: {
                                    type: 'LineString', coordinates:
                                    $geom[0].geometry.coordinates
                              }
                        }, "properties.NAME": {
                              $ne: x
                        }
                  }
            }).toArray();
      }
})
```

Ez hasonló, mint a függvény a within esetében. A logikai felépítésében abban tér el az előző függvénytől, hogy meg kell adnunk azt a feltételt, hogy önmagát ne dobja ki az eredmények közt. Ezt a „`properties.NAME": {$ne: x}` kódrészlettel érhetjük el.

Paraméterül egy folyó nevét várja.

A függvényt meghívhatjuk az adatbázisból a következő paranccsal:

`db.eval("intersect('<NAME>')")` //ahol a `<NAME>` a vizsgált folyó neve

Az alábbi kód meghatározza, hogy mely országhatáron folyik át egy adott folyó.

```
db.system.js.save({
      _id: "intersectCountry",value: function (x) {
            var $geom = db.rivers.find({
                  'properties.NAME' :x  }, {geometry:1, _id:0}
            );
            return db.countries.find({
                  geometry: {
                        $geoIntersects: {
                              $geometry: {
                                    type: 'LineString', coordinates:
                                    $geom[0].geometry.coordinates
                              }
                        }
                  }
            }).toArray();
      }
})
```

A függvényt meghívhatjuk az adatbázisból a következő paranccsal:

`db.eval("intersectCountry('<NAME>')")` //ahol a `<NAME>` a vizsgált folyó neve

Az intersects művelet működött multipolygon esetében is, ellenben a within művelettel. Ehhez azt kértem le a MongoDB-ből, hogy a Maas folyó mely országhatárokat keresztezi, az eredményben Belguim mellett Hollandia is szerepelt.

Legközelebbi pontok gyűjteménye

Találjuk meg azon pontokat közelség szerint sorba rendezve, amelyek adott távolságra vannak egy adott ponttól. Az eredményeket itt nagy méretük miatt nem közlöm.

Két-két vizsgálatot csináltam, ahol mindkét adatbázis ugyanazon eredményeket adta:

A (46;16) koordinátához melyek a 300000 méteren belül eső városok?

A (46.381245;18.74549) koordinátához melyek a 30000 méteren belül eső pontok?

Legközelebbi pont és távolsága

Találjunk egy pontot, amely a legközelebb esik egy megadott koordinátához, és határozzuk meg a pont és a koordináta közti távolságot.

Négy-négy vizsgálatot csináltam, ahol mindkét adatbázis ugyanazon pontokat adta vissza, azonban más mértékegységben számoltak:

MONGODB (rádiusz mértékegység)

A (46,16) koordinátához melyik a legközelebb eső város(cities collection)? Zagreb, distance":0.003396760050806629

A (54,5) koordinátához melyik a legközelebb eső város(cities collection)? Leeuwarden, distance":0.016182576462687025

A (90,80) koordinátához melyik a legközelebb eső város(cities collection)?Longyearbyen, distance":0.20594890409520855

A (46.381245;18.74549) koordinátához melyik a legközelebb eső pont(pontok collection)? Budapest Pécs "distance":1.745329245305404e-8

PostGIS (méter mértékegység)

A (46,16) koordinátához melyik a legközelebb eső város(cities tábla)? Zagreb, "distance":21640.767809517

A (54,5) koordinátához melyik a legközelebb eső város(cities tábla)? Leeuwarden, "distance":103099.31245501

A (90,80) koordinátához melyik a legközelebb eső város(cities collection)? Longyearbyen, "distance":1312102.2802091

A (46.381245;18.74549) koordinátához melyik a legközelebb eső pont(pontok collection)? Budapest Pécs "distance":0.090269319

A MongoDB méter, míg a PostGIS rádiuszt ad vissza a lekérdezés eredményéül. Mivel a méter egy általánosabban használt mértékegység, és könnyebben átváltható más hosszmértékegységekre (km, mérföld), ezért úgy döntöttem, hogy a rádiuszt alakítom

méterré. Ahhoz, hogy a rádiuszt, azaz a sugarat méterre tudjuk váltani az adott koordináták ismerete nélkül, amelyek közt a távot ki szeretnénk számolni, ismerünk kell a Föld sugarát. Ez a hossz azonban sajnos attól függ, hogy éppen melyik szélességi körön hajtjuk végre a számolást. Az egyenlítői sugár 6378,1370 km, míg a poláris sugár 6356,7523 km. A kettő közti szélességi fokokon a két sugár mérete közt vannak a sugarak hosszai. Ez láthatóan nagy differencia ahhoz, hogy az egyiket választva azt mondjuk, vele kell számolnunk. Két lehetőségünk a számolásra, az egyik, hogy kikeressük az adott szélességi körhöz tartozó sugarat, a másik, hogy veszünk egy átlagos sugarat, és azzal számolunk. Azonban a kapott eredmények még így sem lesznek 100% pontosságúak. Az átlagos sugár 6372,797 km, a középérték pedig 6371 km. Én a középértékkel számoltam.

Ahhoz, hogy átváltsuk a sugarat méterbe, a MongoDB fejezetben leírtak szerint a distanceMultiplier használata szükséges:

```
db.cities.aggregate({ $geoNear:  { near: [ <lon>, <lat>],
distanceField: "distance", spherical: true, distanceMultiplier:
6371000,limit: 1  } }) //ahol <lon> és <lat> az adott koordináták.
```

Ez az aggregáció MongoDB-ben a következő eredményeket hozta:

A (46,16) koordinátához melyik a legközelebb eső város(cities collection)? Zagreb, distance": 21640.758283689032

A (54,5) koordinátához melyik a legközelebb eső város(cities collection)? Leeuwarden, distance": 103099.19464377903

A (90,80) koordinátához melyik a legközelebb eső város(cities collection)?Longyearbyen, distance": 1312100.4679905737

A (46.381245;18.74549) koordinátához melyik a legközelebb eső pont(pontok collection)? Budapest Pécs "distance": 0.11119492621840728

Így láthatóan már a két alkalmazás eredményei nagyon hasonlóak voltak. Az

eredményeket a „Great-circle distance between two points”[10] segítségével ellenőriztem az eredményeket, melyek a következők lettek:

Zágráb: **21.64** km

Leeuwarden: **103.1** km

Budapest Pécs: **0.0001112** km

Az utolsó lekérdezés eredményéből látszik, hogy a MongoDB néhány centiméterrel pontosabb eredményt hoz. Így, ha ennyire pontos eredményeket szeretnénk, a MongoDB használata ajánlott. Ennek hátránya, hogy ez esetben meg kell tudnunk jól választani a sugarat. A PostGIS függvénykönyvtár két függvényt is kínál számunkra a távolságok lekérdezésére. Én a tesztek esetében az elsőt használtam:

ST_Distance_Sphere: gyors, de kevésbé pontos. Mindig 6370986 méterrel számol, ez nagyjából megfelel a Föld sugarának középértékének.
ST_Distance_Spheroid: lassabb, de pontosabb. Itt kézzel kell megadni a használt sugarat, akárcsak a MongoDB esetében. A sugár mellett meg kell adni az ‘inverse flattening’-et, mely a Föld laposságát vizsgálja (azaz, hogy mennyire ellipszoid egy gömb formához képest a Föld alakja).

Az utóbbi függvény azonban rosszabb eredményeket hozott a középértékkel számolva, mint a sima ST_Distance_Sphere, és ebből következőleg a MongoDB-nél is. Zágráb esetében 21612.2404578662 méter távolságot kaptam, amely már nem apró egy méteren belüli eltérés, hanem közel 28 méter.

Legközelebbi poligon

Találjunk egy poligont, amely a legközelebb esik egy megadott koordinátához, és határozzuk meg a poligon és a koordináta közti távolságot.

Két-két vizsgálatot csináltam, ahol az eredmények az első esetben egyeztek, és a

[10] http://www.movable-type.co.uk/scripts/latlong.html

másodikban is hasonlóak voltak:

A (49.1876;17.1123) koordinátához melyik a legközelebb eső ország? Czech Republic, distance: 0 (a pont benne volt az országban)

A (87.12;66.5111) koordinátához melyik a legközelebb eső ország? Poland. A távolság a MongoDB esetében 3703208.8594238884, míg a PostGIS esetében 3703213.99245726 volt. A MongoDB lekérdezésben szerepelt a within-ben is használt distanceMultiplier paraméter.

Indexek használata

Ahogy a bevezetőben említettem, az adatbázisok lekérdezéseinek hatékonysága indexeléssel növelhető. A mérések során összehasonlítottam azt a két esetet, amikor használtam indexeket, és amikor nem.

A PostgreSQL különöbző index típusokat biztosít, melyek a MongoDB-hez hasonlóan különböző típusú lekérdezések esetében a leghatékonyabbak. Ezek a típusok a B-fa, Hash, GiST és GIN. A leggyakoribb a B-fa indexek használata, mely a legtöbb szituációban hatékony. A GiST egy generalizált keresőfa, azaz egy olyan kiegyensúlyozott fa sablon, amellyel tetszőleges indexelések implementálhatóak (például B-fa, R-fa, stb…). Előnye, hogy lehetővé teszi egyedi adattípusok használatát, emiatt a térbeli objektumok is GiST indexekkel indexelődnek.

Arról, hogy az adott index használatban van-e egy lekérdezés során, mindkét adatbázis esetében megbizonyosodhatunk. A MongoDB esetében az `explain` függvényt kell meghívnunk a lekérdezésre:

```
db.cities.find( {"properties.CITY_NAME" :  Budapest"}).explain();
```

A PostGIS esetében pedig a lekérdezésünk elé kell írni az EXPLAIN ANALYZE kulcsszavakat:

```
EXPLAIN ANALYZE SELECT * FROM cities WHERE city_name = 'Budapest';
```

Ha futtatjuk ezeket a lekérdezéseket, akkor egyéb információkat is megkapunk a lekérdezésről, mint például a becsült futási időt, illetve hogy hány

rekordot/dokumentumot vizsgál a lekérdezés során az adatbázis. A MongoDB sajnos csak a find típusú lekérdezéseken tudja meghívni az explain függvényt. Ez az eredmények kielemzésekor többször is problémát okozott. Az aggregate típusú lekérdezések paraméterlistájában megadhatjuk ugyan az explain opciót, azonban ez kevés információt ad. A saját funkciókat és a count lekérdezéseket egyáltalán nem tudjuk elemezni.

A PostGIS esetében az látható, hogy a GiST indexek csak akkor vannak használatban, ha a lekérdezéseinket jól írtuk meg. Vegyük példának az ST_Within lekérdezést. Habár a legfontosabb alapelv, hogy a térbeli lekérdezések a WHERE kulcsszó után szerepeljenek, egyáltalán nem mindegy, hogy a lekérdezéseinket hogyan írjuk meg. A Következő két lekérdezés közül a második fog gyorsabban lefutni:

```
SELECT gid,name, ST_AsText(geom) FROM rivers WHERE ST_Within(geom, (SELECT geom FROM countries WHERE iso_3digit ='HUN'));
```

```
SELECT rivers.gid,name, ST_AsText(rivers.geom) FROM rivers, countries WHERE ST_Within(rivers.geom, countries.geom) and countries.iso_3digit ='HUN';
```

A térbeli indexek mellett B-fa indexek használatának nem lett volna értelme. Látható, hogy a within és az intersects lekérdezéseken kívül minden esetben a geometriákon szűrünk. A within és az intersects esetében először valamelyik szöveges mező (név, iso_3digit) alapján kikeresünk egy objektumot, majd ezen objektum geometriájával adunk ki egy újabb lekérdezést. Ha külön-külön kielemezzük ezt a két lekérdezést, azt látjuk, hogy a második a hangsúlyosabb. A MongoDB esetében Magyarország geometriájának becsült lekérdezése kevesebb, mint 1ms, de a geometria belsejében levő folyók lekérdezésének ideje 77ms. Az is látható az explain eredményéből, hogy a 2dsphere indexnek köszönhetően 15 elemet vizsgál meg, mint lehetséges elemei az eredménynek, amelyből végül 4 darabot ad vissza. A PostGIS hasonló tendenciát mutat, Magyarország geometriájának lekérdezése esetében a cost 0.00-7.25, míg a geometriában levő folyók esetében 0.00-235.43 volt a cost.

Mérés eredmények

A következő táblázatok a mérések eredményeinek idejét mutatják millisecundumban. A város a cities, a pont a pontok, a folyó a rivers, az ország a countries, a víz a waterways táblára/gyűjteményre értendő. Az összes mérés során - a megszámlálást leszámítva - mindig az eredményül kapott rekordok összes adatát kiírattuk, beleértve a geometriákat is. Minden mérést ötször végeztem el, és átlagoltam az eredményeket.

Importálás

Az adatok beimportálása UNIX terminálból történt. Az időt a time parancs segítségével mértem, és a parancs eredményei közül a real értéket vettem figyelembe, ami stopperóra szerűen működik. A különböző méretekből adódóan az adatbázisok beimportált táblái/gyűjteményei is különböző méretűek lettek.

Gyűjtemény/Tábla neve	PostGIS mérete	MongoDB mérete	PostGIS futási ideje (ms)	MongoDB futási ideje (ms)
Cities	664KB	1227KB	308	724
Countries	2136KB	5322KB	86	3256
Rivers	208KB	570KB	680	814
Pontok	17MB	23MB	34558	773372
Waterways	11MB	25MB	9672	150926

11. ábra- Adatok mérete az adatbázisban és az importálás ideje

A JSON egy emberek által is olvasható fájltúpus, emiatt viszont a mérete nagyobb a shape fájlokénál. A JSON fájlok nagyobbak, a MongoDB importok ehhez képest is aránytalanul lassan futottak le. Meg kell jegyezni, hogy a PostGIS esetén első lépésben szükségünk van a shape fájlokból sql fájlok létrehozására, ennek idejét a táblázat nem tartalmazza. Azonban ez lényegesen kicsi az importálás méretéhez képest, ennek ideje 100(countries)-1500(pontok) millisecundum közt volt. A MongoDB esetében pedig a 2Dsphere indexeket létre kell hozni még az importálás után, ami szintén plusz idő. A PostGIS-ben importálni csak shape fájlokat tudunk, de a lekérdezések geometriáját GeoJSON-ná alakíthatjuk.

Cold és Warm futások

Ha egy adatbázison végzünk egy műveletet, akkor az adatokat a memóriában, az azokra mutató referenciákat pedig a cahce-ben tárolja a számítógép a továbbiakban a gyorsabb elérés érdekében *(Justin J. Levandoski, 2013)*. A gyakran használt adatokat hívjuk **warm**, a nem használtakat pedig **cold** adatoknak (rekordoknak). Természetesen a rendszertől függően nem csak két állapotunk van, lehetnek **lukewarm** adataink is, amelyek a kettő közt helyezkednek el. Az adatbázis egy újraindítás, vagy egy adott idejű használatlanság után emiatt lassabban működik, mint normál esetben, hiszen nincs adat a memóriában, csupán a fizikai meghajtókon. Ezt a kezdeti lassú működést hívjuk warm-up résznek, amikor a számítógép betölti a szükséges adatokat a memóriába. Ezen jelenség miatt külön mértem az alkalmazások teljesítményét cold és warm állapotban. A JavaScript motor (amely a Node.js motorja is) az inline caching optimalizálási technikát használja.

A PostGIS induláskor 8KB méretű cache bejegyzéseket hoz létre, melyek kezdetben üresek. Később ezeket tölti fel adatokkal, melyek mutatók a fájlokra, flagek, melyek az adatok állapotát és a lock-okat írják le, és egyéb adatok. A cache méretét meg lehet adni. A lekérdezések futtatásakor az általam beállított cache mérete 128MB (defalut) volt. A méréseket elvégezve 512MB cache méret mellett az eredmények nagyon hasonlóak voltak.

A MongoDB automatikusan használja a számítógép szabad memóriáját cahce-elésre. A relációs adatbázisokkal szemben a MongoDB nem használ különböző adatreprezentációt a különböző cache szinteken. Ez azt jelenti, hogy az adatok felhasználható formában kerülnek letárolásra a fizikai szinten is. A MongoDB minden gyakran használt adatot a memóriában tart. Ha lettek indexek létrehozva a lekérdezésekhez, és az adatok belefférnek a RAM-ba, akkor a MongoDB minden lekérdezést a memóriából fog kiszolgálni. A lekérdezés cache nincs implementálva, a MongoDB minden lekérdezést az indexekből és az adatfájlokból hajt végre.

Ahhoz, hogy az adatok lekérdezését szimulálni tudjuk cold állapotban, sorban a következőket kell tenni: Elindítani az adatbázis szervert, elindítani az alkalmazást, futtatni

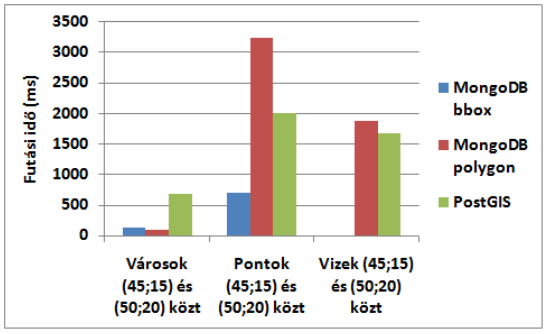
a tesztet, leállítani az alkalmazást, leállítani az adatbázis szervert, törölni a memóriát. Ezen folyamatok automatizálására bash nyelven írtam shell scripteket.

Ahhoz, hogy az adatok lekérdezését szimulálni tudjuk warm állapotban, hasonlóképp kell eljárnunk, azzal a különbséggel, hogy a teszt futtatása előtt “felmelegítjük” az adatbázist. Ez a MongoDB esetén egy db.gyűjtemény.find() , míg a PostGIS esetén pedig egy SELECT * FROM TÁBLA futtatását jelenti.

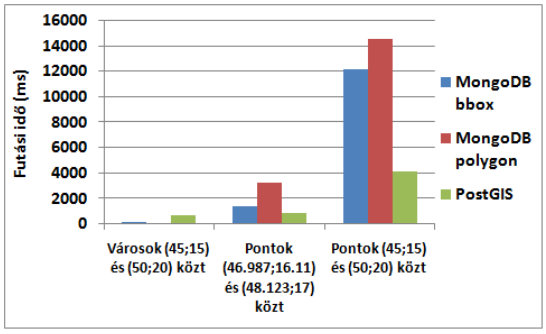
A lekérdezéseket egy harmadik módon is elvégeztem, úgy hogy két lekérdezés közt a cache nem lett törölve, csupán ötször egymás után futtatva ugyanaz a lekérdezés.

A mérések elvégzésekor a PostGIS adatbázis az importáláskor létrejövő GiST indexeket, a MongoDB adatbázis pedig a kézzel létrehozott 2Dsphere indexeket tartalmazta.

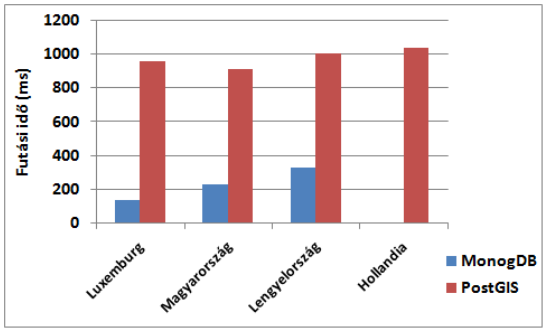
A következő táblázatok a cold futásokat mutatják:



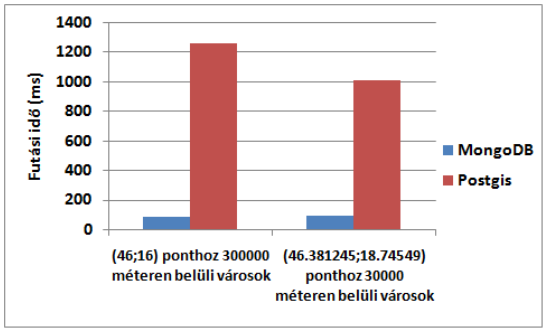
12. ábra-Boxon belüli geometriák megszámlálása



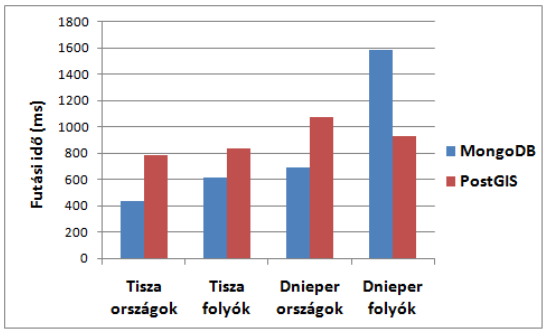
13. ábra- Boxon belüli geometriák lekérése



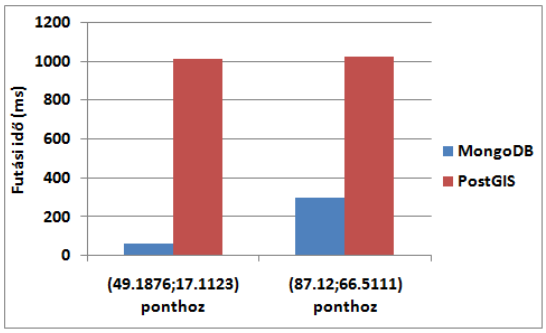
14. ábra-Folyók az adott országhatáron belül



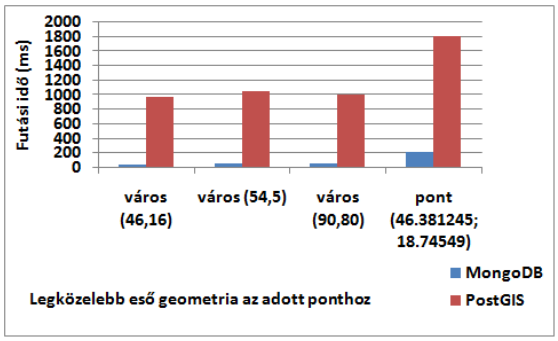
15. ábra-Közelség szerint rendezett városok adott távon belül



16. ábra-Folyók és határok, melyeken átmegy folyó



17. ábra-Adott ponthoz legközelebbi ország



18. ábra-Adott ponthoz legközelebbi pont, és a két pont távolsága

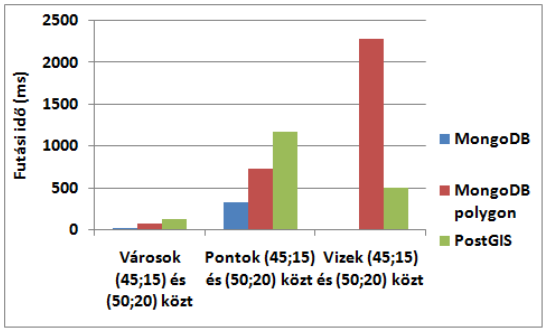
Ahogy az várható volt, a MongoDB a legtöbb esetben sokkal gyorsabb cold állapotban, hiszen ő az adatokat közvetlenül a diszkekről is képes feldolgozni, amíg a PostGIS-nek azokat először mindenképpen be kell töltenie a memóriába. Ahol a PostGIS gyorsabb teljesítményt mutat a MongoDB-nél, azok a bounding box műveletek (12. ábra) és az olyan intersects műveletek, ahol olyan polyline-okat kerestünk, amelynek van metszete egy adott polyline-al (15. ábra). Azonban itt is csak a nagyobb táblák/gyűjtemények vagy rekord/dokumentum (pontok table, illetve hosszabb folyó) esetében. A bounding box esetében a MongoDB lassúsága nem is meglepő, hisz ezeket a lekérdezéseket nem támogatja 2Dsphere index esetén. Látható, hogy különböző méretű gyűjtemények esetében a sebességek is különböznek mindkét alkalmazás esetében. Emellett az is

megfigyelhető, hogy ugyanazon lekérdezéskor az eredmény rekord méretének eltértéseire a MongoDB sokkal érzékenyebben reagált, mint a PostGIS (12., 13. és 15. ábra). Habár a PostGIS esetében is megfigyelhető ez a jelenség, itt sokkal kisebb százalékban.

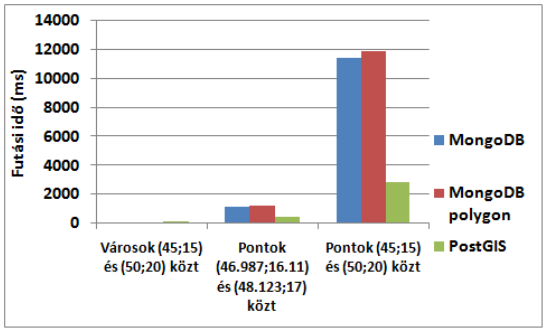
A MongoDB esetében a legközelebbi polygon meghatározása lassabb volt egy olyan pont esetében, amelytől távol esett minden polygon (16. ábra). Ezt a jelenséget nem tudtam kielemezni, ugyanis aggregációval valósítható csak meg ez a művelet. A legközelebbi pont és annak távolsága művelet mindkét rendszerben függött a táblában/gyűjteményben levő pontok számától (18. ábra), és a MongoDB ebben gyorsabbnak bizonyult. A 14. ábrán látható, hogy a MongoDB sebessége erősebben függ a visszaadott geometria méretétől, a PostGIS ehhez képest stabil tendenciát mutatott.

A bounding box műveleteket elvégeztem a MongoDB adatbázist használó alkalmazásban úgy is, hogy a $box helyett egy négyzet alakú polygon belsejében levő elemeket kérdeztem le. A lekérdezések eredményei ugyan mutattak kisebb eltéréseket, azonban ezek nem voltak nagyobbak, mint 0,1%. Ennek az oka az, hogy a $geoWithin a $box esetén euklideszi, míg a $geometry esetén szférikus geometriát használ. A polygon belsejében levő lekérdezések sebessége ez esetekben kissé megnőtt a $box lekérdezésekhez képest (11. és 12. ábra).

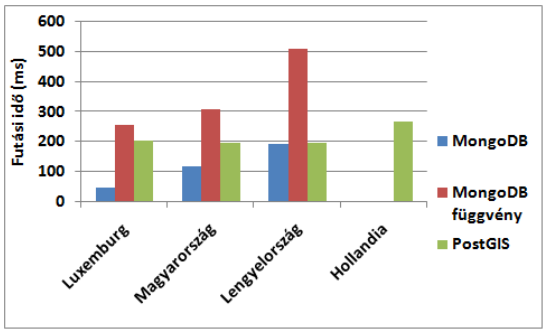
A következő táblázatok a warm futásokat mutatják:



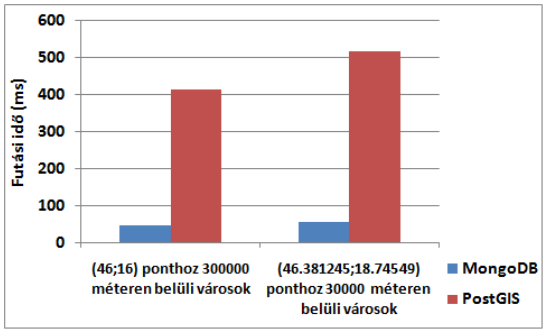
19. ábra-Boxon belüli geometriák megszámlálása



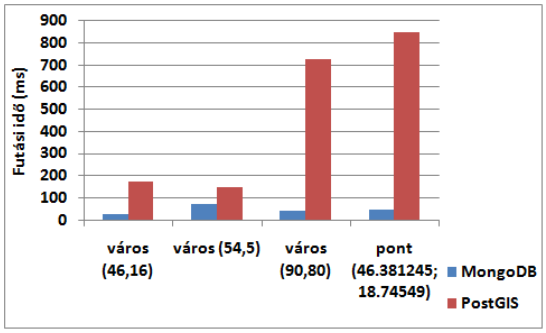
20. ábra-Boxon belüli geometriák lekérése



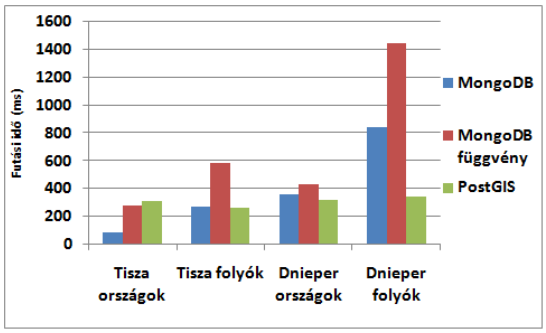
21. ábra-Folyók az adott országhatáron belül



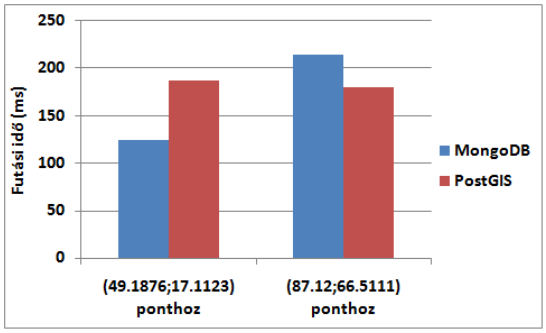
22. ábra-Közelség szerint rendezett városok adott távon belül



23. ábra-Ponthoz legközelebbi pont, és távolságuk



24. ábra-Folyók és határok, melyeken átmegy folyó



25. ábra-Adott ponthoz legközelebbi ország

A táblázatokból az látszik, hogy ahol a PostGIS eddig erős volt, azaz a bounding box (19. és 20. ábra), és az intersects két polyline közt (24. ábra) esetén, ott most még erősebb, mert a MongoDB sebességén a warm-up ezen területeken (és a többin is) kevesebbet javított, mint a PostGIS. A PostGIS itt az intersects két polyline közt művelet esetében már kisebb méretű polyline vizsgálata esetén is gyorsabb, és az egyik esetben a két legközelebbi

polygon meghatározásakor is. A within műveletek esetében is közel jár a MongoDB-hez a Lengyelországon belüli folyók megkeresésekor (21. ábra). A bejárt táblák/gyűjtemények és a lekérdezés eredményének mérete valamint a polygonok távolsága egy adott ponttól hasonlóképp befolyásolja vagy nem befolyásolta a sebességet, mint a cold futások esetében. Azonban most mindkét alkalmazásban megfigyelhetjük, hogy egy adott ponthoz képest a legközelebbi város megkeresése és távolságának meghatározása függ attól, hogy mennyire esik távol a legközelebbi város a ponttól (23. ábra). A 17. és a 25. ábrán is látható, hogy a legközelebbi ország típusú keresések a MongoDB esetében függtek az eredmény geometria méretétől, a PostGIS-ben ez nem tapasztalható.

A lekérdezéseket a szerver újraindítása nélkül újrafuttatva a MongoDB egyedül a közelségi lekérdezésekben tudott jobb lenni a PostGIS-nél. Erre egyszerű magyarázat, hogy a MongoDB nem használ lekérdezés cache-t, ami azt jelenti, hogy habár a használt adatok tárolva vannak a cache-ben, a lekérdezéseket minden egyes alkalommal újra kell terveznie. Emiatt a MongoDB csak minimális javulást mutat ugyanazon kérések ismétlésekor. Ezzel szemben a PostGIS jelentős javulást mutat.

	MongoDB	**PostGIS**
Cold->Warm	2,521236	3,560357
Warm->Újrafuttatott	3,560215	36,04811

26. ábra-A sebesség javulása az alkalmáazásokban

A 26. ábra azt mutatja, hogy a különböző futási módok közt hányszorosak voltak az eltérések az alkalmazásokban. A számok a futási idők átlag eltéréseinek átlagát mutatják. Ebből látszik, hogy a PostGIS teljesítménye warm állapotban jobban növekszik, mint a MongoDB teljesítménye, de még így is ritkán haladja meg azt. A PostGIS kimagaslóan megnövekedett teljesítményt csak a kérések ismétlésekor mutat.

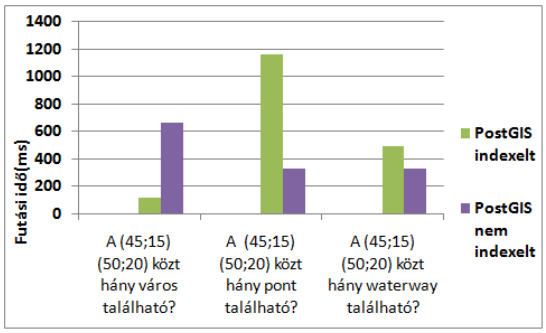
Szerver és adatbázis oldal

Összehasonlítva a MongoDB szerver oldali és az adatbázis oldali feldolgozását a within és intersects esetén, azt tapasztaljuk, hogy a szerver oldali feldolgozás gyorsabb annak ellenére, hogy több kommunikáció van az adatbázis és a szerver közt. Ennek oka, hogy a

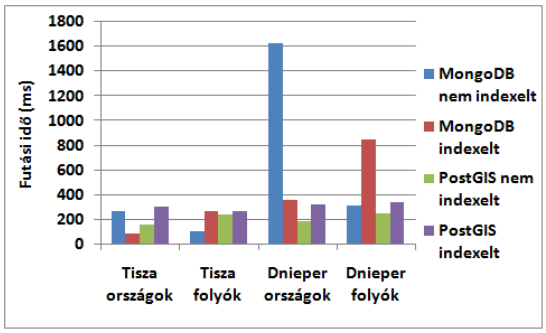
MongoDB-ben definiált függvényeink valójában egy beépített javascript környezetben futnak le, nem pedig a MongoDB C++ kódjában. Amellett, hogy a javascript futási ideje jelentősen lassabb a C++ futási idejénél, az eval függvények esetében a javascript motor elindul, keletkezik egy szál, majd ezután a C++ kód adja át a futást a javascriptnek. A szerver oldali feldolgozás emellett könnyebben implementálható, és dolgozhatunk bármilyen fejlesztőeszközzel, nem csak a terminálból. Emellett az eval függvények hátulütője még, hogy csak a primary szervereken fut le, a replikákon nem, és egy rosszul megírt függvény esetén lehetőséget adhatunk egy injection támadásra is.

Az indexek szerepe a sebességben

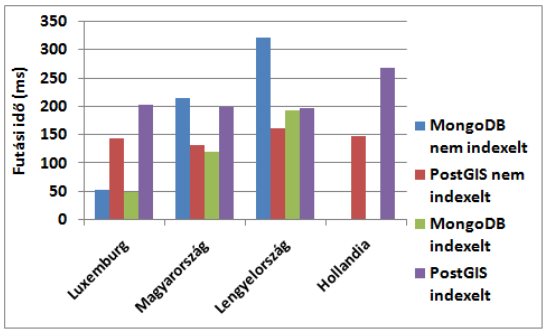
A mérések az előző részben leírt warm futásokkal megyegyező módon történtek, azzal a különbséggel, hogy az indexek törölve lettek. A MongoDB esetében ez a bounding box típusú keresésekben nem okozott eltérést. Ahogy a MongoDB dokumentációja is írja, a bounding box típusú lekérdezéseket csak 2d indexekkel gyorsíthatjuk. A near lekérdezések nem működtek. Emiatt eltéréseket csak a within és az intersects típusú műveletek esetében van értelme mérni a MongoDB esetében. Egy folyó más folyókkal való metszete (28. ábra) esetén az indexek csak rontottak a lekérdezések sebességén, amíg a folyó mely országhatárokon folyik át (28. ábra), és a mely folyók vannak benn egy geometriában (29. ábra) lekérdezések javultak az indexek használatától.



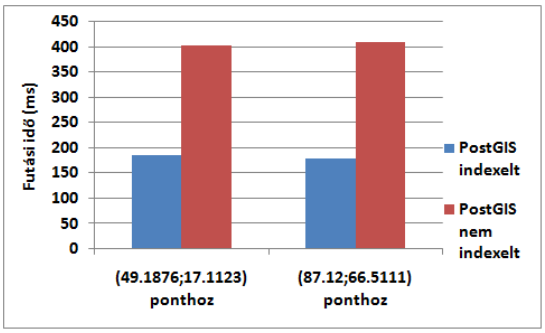
27. ábra-Boxon belüli geometriák megszámlálása



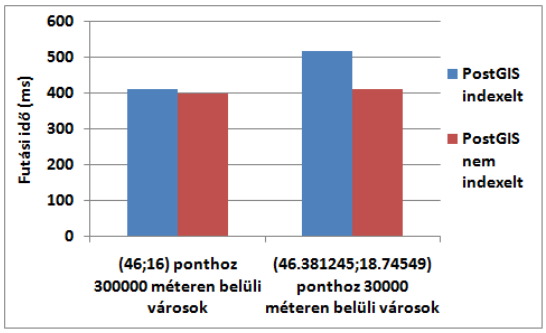
28. ábra-Folyók az adott országhatáron belül



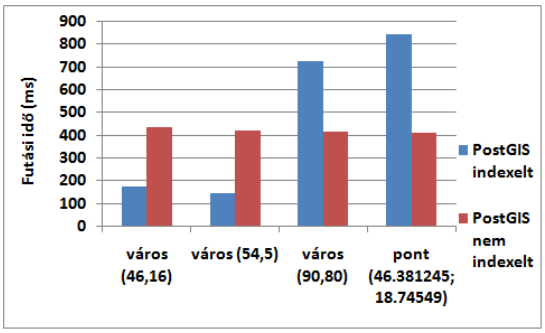
29. ábra-Folyók és határok, melyeken átmegy egy folyó



31. ábra-Közelség szerint rendezett városok adott távon belül



30. ábra-Adott ponthoz legközelebbi ország



32. ábra-Ponthoz legközelebbi pont, és távolságuk

A PostGIS esetében azt látjuk, hogy bizonyos esetekben a térbeli indexek csak rontották a lekérdezéseket. Ennek oka, hogy a térbeli indexek csak akkor javítanak egy lekérdezésen, ha a lekérdezésekben az indexelt geometriákat a WHERE feltételben szűrjük. A lekérdezések megírásakor igyekeztem optimális kódokat írni, azonban erre nem mindig volt lehetőség. A PostGIS-ben a közelségi lekérdezéseket megírhatjuk úgy, hogy a közeli objektumok távolságait is visszaadják, azonban ez esetben már nem fogják az indexeket úgy kihasználni, mint optimális esetben. A közelségi lekérdezések esetében több dolog is megfigyelhető. A legközelebbi polygon (30. ábra) esetében az indexek használata javíthat a lekérdezés sebességén, de a legközelebbi pont és annak távolsága (32. ábra) esetében inkább az tapasztalható, hogy az indexek használata csak akkor segít, ha a legközelebbi pont nincs túl messze, ellenkező esetben inkább ront az eredményen, ahogy az például a MongoDB esetében alapvetően megfigyelhető.

7. LRS

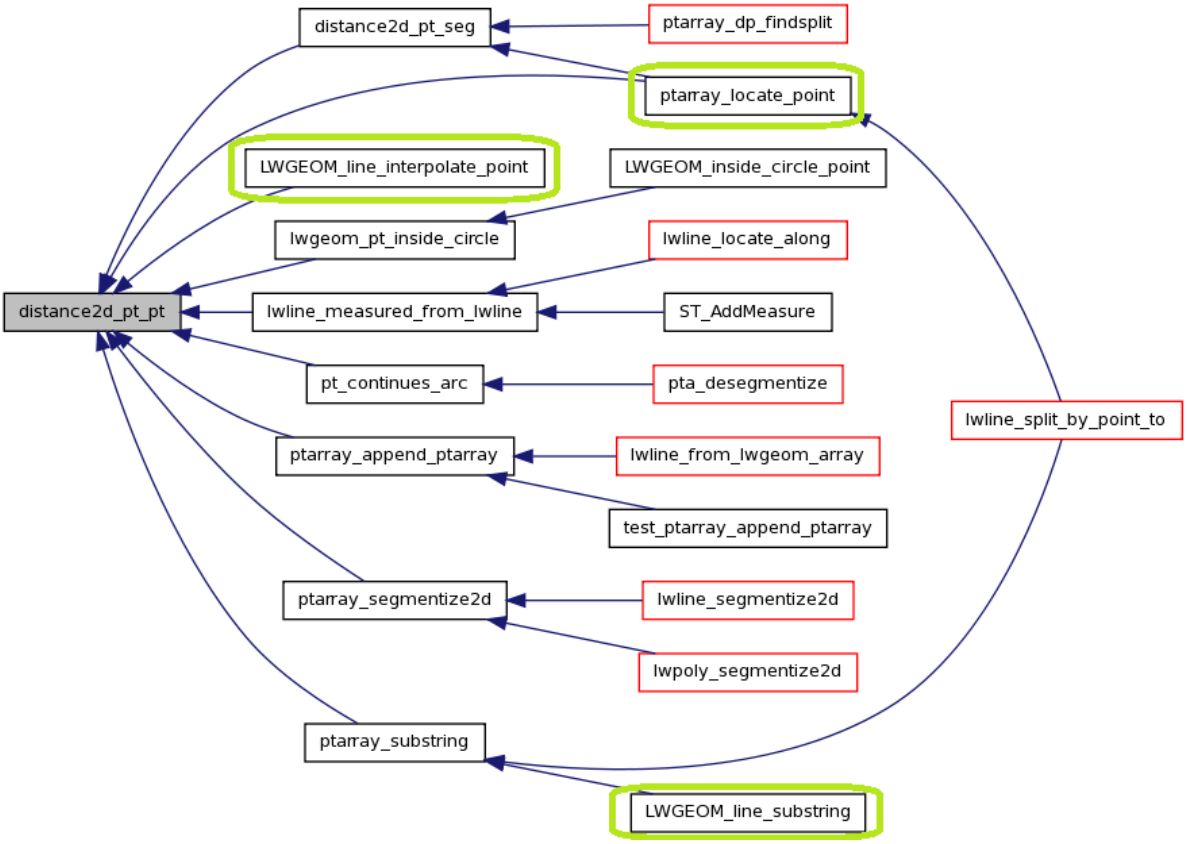
Amíg a PostGIS az LRS-el, és ahhoz kapcsolódóan több funkcióval is rendelkezik, addig a MongoDB ezen funkciókat egyáltalán nem támogatja. Kísérletet tettem ennek megvalósítására, és emellett egy általános függvény megírására, amely egy általánosított alakja az egyik PostGIS függvénynek. Ennek az elsődleges céjla, hogy szubjektív módon is mérhető legyen a MongoDB azon tulajdonsága, hogy mennyire használható rugalmasan újabb funkciók implementálására.

Létező PostGIS függvények

A PostGIS egy – pl. az Oracle-höz *(Oracle, 2002)* képest – kisebb LRS támogatást tartalmaz, amely nyolc függvényt foglal magába. Ebből három lett implementálva a MongoDB adatbázist használó alkalmazásban.

Ahhoz, hogy egy PostGIS geometrián linear referencing lekérdezéseket végezzünk, a geometria típusú input paraméternek linestring-nek kell lennie. Az adatbázisban az útvonalak automatikusan MultiLineString típusúként jöttek létre annak ellenére, hogy valójában összefüggő útvonalakról van szó. Emiatt át kell őket transzformálni az ST_LineMerge függvénnyel.

Az általam választott függvények az ST_line_interpolate_point, az ST_line_locate_point és az ST_line_substring volt. A 33. ábrán látható, hogy mindhárom függvény közvetetten ugyanazt, a distance_2d_2d nevű függvényt használja a távolságok kiszámítására, amely az euklideszi távolság alapján dönti el két pontról, hogy milyen messze vannak egymástól, és csak az x és y koordinátákkal dolgozik. Az algoritmusok hasonlóan működnek, mint az általam implementált esetekben, azonban én a LAT és LON koordinátákat 3 dimenziós pontokká transzformálom, és így számolom ki két pont távolságát, majd az eredményeket szükség esetén visszatranszformálom. Ebből adódóan a két alkalmazás eredményei közt kisebb (akár 1-2 százalékos) eltérések előfordulhatnak. Ez a jelenség magyarázatot ad azon különbségekre, amelyek a PostGIS bounding box, és a MongoDB boxot reprezentáló within műveletek eredményei közt tapasztaltunk.



33. ábra- A distance2d_pt_pt hívási grafikonja, kiemelve az érintett függvények[11]

ST_line_interpolate_point(utvonal, location)

Visszaad egy pontot interpolálva az utvonal paraméteren. A location paraméter egy szám kell, hogy legyen 0 és 1 közt, amely reprezentálja a pont helyzetét a linestring-en vett teljes útvonalon. A következőképp néz ki a Duna középpontjának lekérdezése PostGIS-ben:

```
SELECT ST_AsText(ST_LineInterpolatePoint(ST_LineMerge(geom),  0.5)
) from rivers where name= 'Danube '
```

MongoDB-ben az alábbi algoritmussal valósítottam meg a függvényt.

[11] http://PostGIS.refractions.net/documentation/PostGIS-doxygen/da/de7/liblwgeom_8h_19a875e067d3c38e761a9328a9d524ff.html#19a875e067d3c38e761a9328a9d524ff

```
INIT OsszTav = pontTav = pontTavElotti = 0

FOR (i=0; i<geoms.length-1;i++)

INIT tav = tavolsag(geoms[i], geoms[i+1])

     SET OsszTav = OsszTav + tav

INIT I = 0

WHILE (pontTav/OsszTav<UtvonalArany)

     SET tav = tavolsag(geoms[I], geoms[I+1])

     SET pontTavElotti = pontTav

     SET pontTav = pontTav + tav

     I++

SET SzakaszArany = (UtvonalArany - (pontTavElotti/OsszTav )) / (
(pontTav/OsszTav) - (pontTavElotti/OsszTav) )

SET P1 = geoms[c-1].convertXYZ()

SET P2 = geoms[c].convertXYZ()

SET P3 = (P1+SzakaszArany*(P2+P1)).convertLATLNG()

RETURN P3
```

helyzetét is megkeresi a szakaszon. Ehhez a lat és lon koordinátákat X, Y és Z koordinátákká kell alakítani, majd az így meghatározott pontot visszatranszformálni lat és lon koordinátákra.

ST_line_substring(utvonal, start, end)

Visszaad egy útvonalat, amely egy része utvonal-nak. A start és az end paraméterek számok 0 és 1 közt, akárcsak az ST_line_interpolate_point esetében. Ez a két paraméter határozza meg, hogy utvonal-at mely két ponton törjük meg. A következőképp néz ki a Duna egynegyede és háromnegyede közti rész lekérdezése PostGIS-ben:

```
SELECT ST_AsText( ST_Line_Substring(ST_LineMerge(geom), 0.25, 0.75
) ) from rivers where name='Danube'
```

A MongoDB megoldás hasonló, mint az ST_line_interpolate_point esetében, azonban itt egy helyett két pontot határozunk meg utvonal-on, és az ezek közé eső pontokkal összefűzve adjuk vissza az új polyline-t. Így a visszatért polyline kezdő és végpontja az eredeti polyline valamely szakaszainak belső pontjai, és a köztes pontok (amennyiben vannak) az eredeti polyline pontjainak részhalmaza.

ST_line_locate_point(utvonal, Point)

Visszaad egy számot 0 és 1 közt, amely reprezentálja az utvonal legközelebbi pontját Point-hoz, mint egy arány az utvonal paraméter teljes hosszán véve. A következőképp néz ki a Tisza (46;20) ponthoz képest legközelebbi pontjának lekérdezése a PostGIS-ben:

```
SELECT ST_LineLocatePoint(ST_LineMerge(geom),
ST_SetSRID(ST_MakePoint(20, 46), 4326) ) from rivers where
name='Tisa'
```

MongoDB-ben a következő algoritmussal valósítottam meg a függvényt:

```
INIT keresettPontok = GeoPontok(polyline, point)
INIT OsszTav = pontTav = 0
INIT elertuk = false
FOR (i=0; i<geoms.length-1;i++)
     INIT tav = tavolsag(geoms[i], geoms[i+1])
     SET OsszTav = OsszTav + tav
     IF geoms[i]=KeresettPontElottiPont()
          SET elertuk = false
     IF !elertuk
          SET pontTav = pontTav + tav
SET pontTav = pontTav + tavolsag(keresettPontok.korabbi,
keresettPontok.pont)
RETURN pontTav/OsszTav
```

A GeoPontok függvény két pontot ad vissza. Az első (pont) a point paraméterhez

legközelebb eső pont a polyline-on, a második (korabbi) pedig polyline azon szakaszának az első pontja, amelyen az első pont van. Az algoritmus szummázza az útvonal pontjait a 'korabbi' pontig, és ehhez még hozzáadjuk 'korabbi' és 'pont' távolságát. Az így kapott távolságot osztja el az útvonal teljes hosszával.

Az ST_Line_locate_point általánosítása

Ez az általam mindkét alkalmazásban implementált függvény a képernyőn felsorolja egy kiválasztott folyó összes meghatározó LAT és LON koordinátáját, amellyel definiálva van az objektum. Ezek lesznek a shape pontjaink. Az első és az utolsó pont méret információja kötelezően megadandó, az összes többi pontról pedig eldönthetjük, hogy megadjuk-e vagy sem (LRS pont legyen-e vagy sem). Miután elküldjük kérésünket a szervernek, ő kiszámolja a hiányzó méret információkat, majd a képernyőre megjeleníti az összes shape pont LAT és LON koordinátája mellett azok méret információit is.

A funkciót megvalósíthatjuk az architektúra különböző komponenseiben. A jade, mint a kliens oldali HTML megjelenítő, nem alkalmas számítások végrehajtására, de nem is lenne értelme ennyi adatot feldolgozásra küldeni a kliensnek, és a céltól is eltérne. Annak eldöntésére, hogy adatbázis vagy szerver oldalon érdemesebb-e megvalósítani egy számítást, a következőket kell figyelembe venni:

Számítások bonyolultsága: Mikor csak vertikálisan skálázható adatbázisok léteztek, akkor a bonyolult számításokat érdemesebb volt szerver oldalon végezni, melyek horizontálisan is skálázhatóak. Ma már ezt a tulajdonságot sok adatbázis tudja, ezért ez a tényező csak bizonyos adatbázisok esetén számít, mint például a relációs adatbázisok.

Az adat mennyisége: Az adatok hálózaton való küldése lassíthatja a futási időt.

Kényelem: Figyelembe kell vennünk, hogy az adott adatbázis-kezelő nyelvén mennyire körülményes megírnunk egy számítást, ha egyáltalán az megvalósítható.

MongoDB esetében a szerver oldalon és az adatbázisban is megírhatjuk a kódot. Két pont közt levő pontok méret információját ki kell számolni, akármelyiket módszert választjuk.

```
FUNCTION INTERPOLATION(Number[]TAVOK, Number[] MERETEK)
  INIT STARTPOINT = 0
  INIT I = 0
  WHILE I< TAVOK.length
  INIT AKTTAV = 0
  DO
      SET AKTTAV = AKTTAV + TAVOK [I]
```

A bemenő paraméterek a TAVOK és a MERETEK. A TAVOK a két pont közti távolságot tartalmazza, a MERETEK a méret információkat tartalmazza, az első és az utolsó eleme mindig meg van adva, a köztes elemek nem mindenhol tartalmaznak értéket. A cél a MERETEK tömb hiányzó elemeit kiszámolni úgy, hogy két egymást követő ismert elem közt vesszük a távolságot, és lineáris interpolációval meghatározzuk a köztes, nem ismert elemeket. A továbbiakban négy különböző módszert mutatok be az ST_Line_locate_point általános alakjának implementálására, ahol mind a négy esetben ugyanezt az interpolációs módszert használtam. A kód ezen részét nem kellett újraírnom, hiszen a szerver és az adatbázis oldalon is futtatható a javascript.

A függvény megvalósítása MongoDB-ben

Szerver oldali megvalósítás Geolib package használatával

A szerver oldalon egyszerű dolgunk van, a Node.js különböző csomagjai segítségével könnyen megvalósíthatjuk a függvényt.

Egy lehetséges megoldás a geolib package használata, mely rengeteg függvényt biztosít, nekünk a getDistance-re van szükségünk, amely méter pontossággal adja meg az eredményt. Annyi dolgunk van, hogy a 2-2 latitude és longitude koordinátát paraméterként átadjuk a metódusnak, és egy távot kapunk vissza méterben. Ez ugyan a Vincenty formulát használja, de levágja a tizedesjegyeket, így nem kapunk feltétlenül pontos eredményt, elveszítjük a formula által nyújtott előnyöket. A csomag

forráskódjában egyszerűen átírható a kerekítést végző részt, csupán pár helyen a kerekítő függvényeket kell kitörölnünk a getDistance metódusból. Az előző három LRS függvényt is a szerver oldalon valósítottam meg, és ezzel a csomaggal határoztam meg a távokat.

Szerver oldali megvalósítás Haversine módszer használatával

A geolib package mellett megírtam a haversine metódust, mely eredménye méterben 9 tizedesjegy pontosságig egyezett a később adatbázis oldalon megírt függvények eredményével, néhány kilométeres nagyságrendű hosszakra is.

Adatbázis oldali megvalósítás Remove és Insert metódusokkal

Az adatbázisban ezen számítások megvalósíthatóak, azonban több okból is körülményesek. A MongoDB fejezetben megvizsgált aggregáció segítségével számolhatjuk ki 2 pont közt a távolságot. Mivel most a pontok egy polyline koordinátáiból jönnek, ezért azokat fel kell dolgoznom. Habár a MongoDB-ben definiálhatunk objektumokat, azonban azokra lekérdezéseket nem futtathatunk, csak ha azokat gyűjteményekben tároljuk. Ezen gyűjteményeket létre kell hozni, mert ugyan a MongoDB a nem definiált gyűjteményeket automatikusan létrehozza, ha frissíteni próbálunk egy nem létező gyűjteményt, de jelen esetben szükségünk van térbeli indexelésre is, amit csak kézzel hozhatunk létre.

Egy megoldás lehet az adatbázis oldali feldolgozásra, ha a segédtáblát az összes koordinátával sorszámozva egyszerre feltöltjük, majd ezután számítjuk ki a pontok közti távolságot. Mivel a MongoDB-ben több parancsot is ki kell adni, érdemes azt egyetlen MongoDB függvénybe megírni, megspórolva a túl sok kommunikációt a szerver és az adatbázis közt, ahogy a within és intersects tesztekben láttuk.

```
db.system.js.save( { _id: "LRS", value: function(river, sizeArr){
      var $geom =
      db.rivers.find({'properties.NAME':river},{geometry:1,
      _id:0})[0].geometry.coordinates;//Keresett geometria
      db.ponts.remove({}); //Segédtábla, mindig ürítjük és
      feltöltjük
      for (var i=0; i<$geom.length; i++) {
            db.ponts.insert({ _id:i, geometry: {type:'Point',
            coordinates: $geom[i]  }   })
      }
      var hosszok = new Array(); //Polyline pontjai közti hosszok
      for (var i=1; i<$geom.length;i++){
            hosszok[i-1] = db.ponts.aggregate([{$geoNear: { near:
             $geom[i-1] , distanceField: "distance", spherical:
            true, distanceMultiplier:6371000 , query:{ _id : i} }
              }]).firstBatch[0].distance
      }
      var meretArr = sizeArr.split(',');    //Konverziós műveletek
```

Ez azonban lassan fut le. Az adatbázis-műveletek statisztikáit a db.setProfilingLevel(2) parancs kiadása után naplózza az adatbázis a system.profile gyűjteménybe. A paraméterül kapott 2 azt jelenti, hogy minden műveletet naplózzon. A dokumentumok millis mezője tartalmazza a művelet lefutási idejét, így rászűrhetünk az, egy adott időnél hosszabb lefutású műveletekre. ebből látszik, hogy az insert és remove műveletek miatt lesz lassú a lekérdezés, melyek egyértelműen az adatbázis-kezelők leglassabb műveletei, ez a MongoDB esetében kifejezetten igaz.

Adatbázis oldali megvalósítás Haversine módszer használatával

Úgy gyorsíthatunk az adatbázis oldali megvalósításon, ha megírjuk a saját távolság metódusunkat, de itt igazából el is veszítettük a MongoDB által nyújtott térinformatikai támogatást, hisz többé nem az általa nyújtott térinformatikai függvényeket használjuk. Szerencsére csak az előző kódból kell kitörölni a remove és insert műveleteket, és betenni néhány részt a szerver oldalon megírt kódból. A toRad függvényt itt meg kell írni, a MongoDB nem ismerte fel ezt az egy soros JavaScript függvényt.

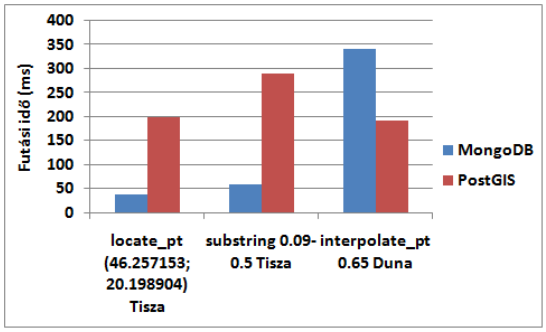
```
db.system.js.save( { _id: "LRSWithoutWrite", value:
function(river, sizeArr){
var $geom = db.rivers.find({'properties.NAME':river},{geometry:1,
_id:0})[0].geometry.coordinates;
function toRad(deg) { return deg * (Math.PI/180) }
function haversine(lat1, lat2, lon1, lon2) {
      var R = 6371000; // metres
      var φ1 =toRad(lat1);
      var φ2 =toRad(lat2);
      var Δφ = toRad(lat2-lat1);
      var Δλ = toRad(lon2-lon1);
      var a = Math.sin(Δφ/2) * Math.sin(Δφ/2) +
            Math.cos(φ1) * Math.cos(φ2) *
            Math.sin(Δλ/2) * Math.sin(Δλ/2);
      var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
      return R * c;
}
```

A függvény megvalósítása PostGIS-ben

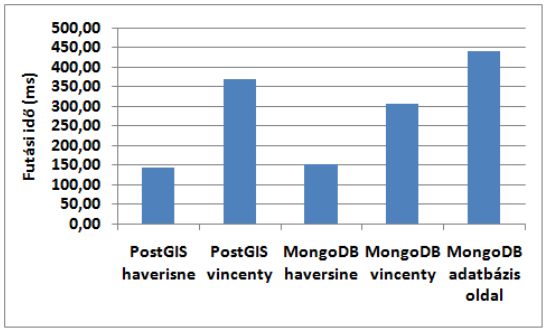
A PostGIS egy open source eszköz, amely C programozási nyelven íródott. Van hozzá fejlesztői dokumentáció, de egy kód megírása mély ismeretét igényli az egész alkalmazás forráskódjának. Függvényeket is definiálhatunk az adatbázisban, azonban ezek megírása nem a procedurális nyelvek, hanem az SQL struktúráját követi és több megszorítást is tartalmaz. Ezek miatt a PostGIS adatbázis oldali LRS függvény implementálása igen körülményes lenne, ezért csak a szerver oldalon írtam meg ezt a függvényt.

Ahhoz, hogy a függvény implementálható legyen, elsőnek le kell kérnünk az adott folyó geometriáját. A probléma fejezetben tárgyalt „impedance mismatch” a PostGIS esetében ekkor előjön, mert ő egy String-el tér vissza, amelyet egy 2 dimenziós tömbbé kell alakítanom miután a felesleges metaadatokat (pl. geometria típusa) kivágtam a string-ből. A MongoDB esetében erre nincs szükség, ő alapvetően egy tömbbel tér vissza, amely elemei két-két számot tartalmazó tömbök, amelyek a LAT és LON koordináták. Tehát a MongoDB egy olyan objektumot küld a szerver alkalmazásnak, ami bármilyen transzformáció nélkül továbbküldhető a kliens alkalmazásnak. Ezt leszámítva azonban a MongoDB esetében használt szerver oldali kódot átemelhettem a PostGIS szerver alkalmazásba, az algoritmus ugyanaz, csupán ezt a konverziót kellett elvégeznem.

Mérés eredmények



34. ábra – LRS függvények futási ideje



35. ábra – Általánosított locate point futási ideje

A MongoDB LRS függvényei viszonylag gyorsak voltak, de függtek a bemenő geometriák méretétől. A 34. ábrán látszik, hogy a kisebb geometriákon (Tisza) gyorsabb, mint a nagyobbakon (Duna), és a lekérdezések hasonló idővel futnak. A PostGIS ehelyett inkább

arra érzékenyebb, hogy melyik függvényt hívjuk meg.

A 35. ábrán nem szerepel azon MongoDB implementáció futási ideje, amely írt is az adatbázisba. Ez jelentősen sok, 21630 ms volt. Az adatbázis oldalon megírt műveletek ez esetben is alulmúlták a szerver oldaliakat. Az általam megírt távolság számító haversine algoritmus bizonyult a leggyorsabbnak, amely gyakran elegendő pontosságú eredményt ad vissza. A PostGIS szerver alkalmazás hasonló sebességű volt, mint a MongoDB.

Ha valami oknál fogva mégis az adatbázis oldalon szeretnénk mindenképp implementálni a kódot, akkor a számításokat felgyorsíthatjuk a mapReduce vagy pedig a $where operátor használatával. Egy másik mód lehet a számításhoz szükséges adatok (azaz az alakzatok koordinátáinak) permanens tárolása, melyet az adatbázis-kezelőben állítunk elő. Ez akkor lehet célravezető, ha az alakzataink nem fognak változni a továbbiakban.

8. Konklúzió

A legtöbb támogatott és általam implementált lekérdezés esetén is felveszi a MongoDB a versenyt a relációs adatbázisokkal. Vannak lassabb és gyorsabb lekérdezések, a PostGIS-hez viszonyítva, általánosságban versenyképesnek mondható vele. Erőssége a within, kizárólag pontok esetében pedig a near, illetve a bounding box count típusú lekérdezésekben mutatkozik meg. A lekérdezések esetenként pontosabb eredményt is hoztak, mint a PostGIS. Gyors „cold" működés figyelhető meg minden lekérdezés típus futtatásakor, nincs szükség az adatokat a memóriába tölteni a lekérdezésekhez.

A MongoDB lekérdező nyelve könnyen tanulható, nincs szükség mélyebb informatikai vagy halmazelméleti ismeretekre. A lekérdezéseket általában nem kell optimalizálni. Ha szükséges, javascript nyelven írhatunk saját függvényeket, azonban ezeket ajánlatosabb szerver oldalon megvalósítani, mert lassúak lehetnek. Az adatbázis struktúrája nem igényel tervezést, és előre definiált sémákat. A sémák nem kötött struktúrájúak, egyedi adatokat is könnyen tárolhatunk benne. Nincs szükség az adatbázis és a szerver alkalmazás közt az objektumok konverziójára.

A MongoDB egyszerűen skálázható vízszintesen, nincs szükség külső eszközök használatára. Az egyetlen dolog, amit skálázáskor ki kell választanunk, hogy a gyűjtemény dokumentumait hogy osztjuk el a csomópontok közt. A MongoDB a particionálás-tűrést biztosítja, cserébe a rendelkezésre állást nem teljesíti a CAP tétel három eleméből. A MongoDB nem teljesen ACID, így kritikus rendszerekben nem használhatjuk.

A MongoDB fejlesztését 2007 októberében kezdték, azóta hirtelen nagy felhasználói bázisra tett szert. A dokumentációk szinte mindenre választ adnak. Gyakoriak a frissítések, a legújabb, 3.0-ás verzió a diplomamunkám elkészítése közben, 2015 márciusában jelent meg. Habár jelenleg a MongoDB térinformatikai támogatottsága kicsi, ígéretesnek tűnik az egyre fejlődő rendszer.

A tradicionális térbeli lekérdezések közül nem mind van megvalósítva a MongoDB-ben. Habár a hiányzó lekérdezések implementálása elképzelhető az OCG szabvány szerint,

azonban ez megfelelő jártasságot igényel és körülményes. Nincs legrövidebb útvonal művelet. A within (bennfoglalás) művelet multipoligon esetén nem értelmezhető, hibát kapunk vissza, ezt a dokumentáció nem említi. A bounding box művelet lassú. A bounding box within csak pontokra működik. Ez a probléma kiküszöbölhető úgy, ha a boxot szimuláljuk polygonnal, azonban így a művelet sebessége nagyon lassú lesz.

A MongoDB csak a WGS84 dátumot támogatja, ami ugyan a legnépszerűbb nemzetközi szabvány, azonban más országokban - pl. Magyarországon – nem feltétlenül a legelterjedtebb. Más dátumban levő adatokat előbb át kell alakítani egy külső eszközzel erre a dátumra importálás előtt. Csak a GeoJSON formátumot támogatja mind kimenetként, mind bemenetként.

Csak a find lekérdezések kielemzésére van lehetőség. A MongoDB 3.0 ugyan már kínál lehetőséget a count lekérdezések kielemzésére is, de az aggregáció és a saját függvények kielemzésére még nincs lehetőség.

Lassú betöltési idő, dokumentumok beszúrása, módosítása sokáig tarthat. Nincs lekérdezés cache, emiatt az újrafuttatott lekérdezések futási ideje nem javul jelentősen az első futáshoz képest.

A MongoDB egy teljesen más tervezési alapokon álló rendszer, mint a relációs adatbázisok. Ebből adódóan sok előnnyel és hátránnyal rendelkezik azokhoz képest. Nem mondhatjuk azt, hogy ezentúl használjunk MongoDB adatbázist a relációs adatbázisok helyett, azonban egy rendszer megtervezésekor érdemes megfontolni használatának lehetőségét, összevetve az előnyöket a hátrányokkal. Bizonyos esetekben célravezető lehet a MongoDB használata. Térinformatikai szempontból a MongoDB használata akkor ajánlott, ha a lekérdezéseink típusai a near, a within és az intersects. Ezen belül is a near típusúak esetén teljesít a legjobban, ahol a sebesség mellett a távolság pontos meghatározásában is felülmúlta a PostGIS-t. Ezen felül használata akkor is javasolt, ha az adatbázis megtervezésébe nem szeretnénk túl sok munkát belefektetni, de mégis szeretnénk egy skálázható vagy bonyolult struktúrájú objektumokat kezelő rendszert kapni.

9. Referenciák

(IfEN), E. O. (1998). *WGS 84 IMPLEMENTATION MANUAL.*

Baas, B. (2012). NoSQJ spatial: Neo4j versus PostGIS. *NoSQJ spatial: Neo4j versus PostGIS Master thesis* . Hollandia.

Baumann, P. C. (2008). *Web coverage service (WCS) implementation standard.* Open Geospatial Consortium.

Browne, J. (2009). *Brewer's CAP Theorem*. Letöltés dátuma: 2015. 5 14, forrás: julianbrowne.com: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

Cook, W. R. (2006). *Integrating programming languages and databases: What is the problem?* odbms.org, Expert Article: Department of Computer Sciences, University of Texas at Austin.

de la Beaujardiere, J. (2004). *OpenGIS web map service (WMS) implementation specification.* Open Geospatial Consortium.

Hijmans, R. J. (2014. september). Introduction to the "geosphere" package (Version 1.3-11).

Justin J. Levandoski, P.-A. L. (2013. april). Identifying Hot and Cold Data in Main-Memory Databases.

Matuszka, T. -A. (2014). *Experimental Evaluation of Some Geodata Management Systems.*

Pourabbas, E. (ed) (2014). *Geographical Information Systems: Trends and Technologies,.* CRC Press Inc/ Taylor & Francis Group. ISBN 9781466596931 http://www.crcpress.com/product/isbn/9781466596931 Letöltés dátuma: 2015. 5 14

Sinnott, R. (1984). Virtues of the Haversine. *Sky and Telescope* , old.: 159.

Stefan Steiniger, A. J. (2011). Free and Open Source GIS Software for Building a Spatial Data Infrastructure.

Szilárd, T. G.–M.–P. (2002. jan). A WGS84 és HD72 alapfelületek közötti transzformáció Molodensky-Badekas-féle (3-paraméteres) meghatározása a gyakorlat számára. *Geodézia és Kartográfia* , old.: 11.

Tibor, N. (2012). Az Oracle adatbázis-kezelő térinformatikai lehetőségei.

Vincenty, T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Surwey review* , 88-93.

Vretanos, P. A. (2005). *Web feature service implementation specification.* Open Geospatial Consortium.

Elek, I. (2014). *Topologic data structures*

Mélykúti, G. (2010) *Topográfia 1*., Térképészeti alapfogalmak

PostGIS 2.0 Manual. http://PostGIS.net/docs/manual-2.0/ Letöltés dátuma: 2015. 2 27

MongoDB 2.6 Documentation 2015 Letöltés dátuma: 2015. 4 18

http://docs.MongoDB.org/manual/applications/geospatial-indexes/ Letöltés dátuma: 2015. 5 12

http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=welcome Letöltés dátuma: 2015. 3 16

Sunday, Dan (2001), *Inclusion of a Point in a Polygon*, http://geomalgorithms.com/a03-_inclusion.html Letöltés dátuma: 2015. 4 26

Rigaux, P. (2002) - *Spatial Databases: With Application to GIS*

http://www.dba-oracle.com/t_lru_latches.htm Letöltés dátuma: 2015. 5 03

http://PostGIS.net/stuff/PostGIS-1.5.pdf - GIS objektumok (környezet fejezet) Letöltés dátuma: 2015. 5 11

Hadjigeorgiou, C. (2013 August) *RDBMS vs NoSQL: Performance and Scaling Comparison*. Master thesis. The University of Edinburgh

Függelék

A MongoDB sémákat a következő táblázatok mutatják. Emellett természetesen az összes gyűjteménynek van _id mezője, ezek az azonosítók az importáláskor automatikusan generálódnak. A sémában ezt nem kötelező feltüntetni.

City séma:

```
properties: {
        CITY_NAME : String,          //A város neve
        GMI_ADMIN : String,          //
        ADMIN_NAME : String,         //
        FIPS_CNTRY : Boolean,        //
        CNTRY_NAME : String,         //Melyik országban található
        STATUS :  String,            //Települési jogállás
        POP_RANK : Number,           //Lakosságszám
        POP_CLASS : String,          //
        PORT_ID : Number             //
},
geometry: {
        type: Object,                //Geometria típusa (ez point, de a mongoose nem ismeri a
        index: '2dsphere'            //térinformatikai típusokat, ezért Object-et adtam meg)
}
```

Country séma:

```
properties: {
        ISO_2DIGIT : String,     //Ország 2 betűs azonosítója (Magyarország pl. HU)
        ISO_3DIGIT : String,     //Ország 3 betűs azonosítója (Magyarország pl. HUN)
        CNTRY_NAME : String,     //Ország neve
        POP_CNTRY : Number       //Lakosságszám
   },
geometry: {
        type: Object,            //Geometria típusa (ez polygon, de a mongoose nem ismeri a
        index: '2dsphere'        //térinformatikai típusokat, ezért Object-et adtam meg)
}
```

River séma:

```
properties: {
        NAME : String            //Folyó neve
   },
geometry: {
        type: Object,            //Geometria típusa (ez polyline, de a mongoose nem ismeri a
        index: '2dsphere'        //térinformatikai típusokat, ezért Object-et adtam meg)
}
```

Pontok séma:

```
properties: {
    osm_id: Number,              //OpenStreetMap ID
    timestamp: Date,             //Adat rögzítésének ideje
    name: String,                //Pont neve (ha van)
    type: String                 //Pont típusa (pl. traffic signal, milestone)
```

```
    },
geometry : {
      type: Object                    //Geometria típusa (ez point, de a mongoose nem ismeri a
    , index: '2dsphere'               //térinformatikai típusokat, ezért Object-et adtam meg)
}
```

Waterway séma:

```
properties: {
        osm_id : Number,              //OpenStreetMap ID
        name : String,                //Víz neve
        type : String,                //Víz típusa (pl. lake,river,canal,dam stb…)
        width : Number,               //Objektum szélessége méterben
},
geometry : {
        type: Object                  //Geometria típusa (ez point, de a mongoose nem ismeri a
        , index: '2dsphere'           //térinformatikai típusokat, ezért Object-et adtam meg)
}
```

PostGIS

A MongoDB-hez hasonló az adatok struktúrája, csak itt az adatok táblákban tárolódnak. A gid oszlopok az egyedi azonosítók. Egymás mellett látható az oszlop neve, és típusa. Értelemszerűen az oszlopokban tárolt adatok ugyanazt az adatot reprezentálják mind a MongoDB egyező nevű gyűjteményének egyező nevű mezőjében, de az adatok típusa eltérhet.

rivers tábla:

gid	integer
name	character varying
geom	USER-DEFINED

cities tábla:

gid	integer
city_name	character varying
gmi_admin	character varying
admin_name	character varying
fips_cntry	character varying
cntry_name	character varying
status	character varying
pop_rank	smallint
pop_class	character varying
port_id	integer
geom	USER-DEFINED

countries tábla:

gid	integer
iso_2digit	character varying
iso_3digit	character varying
cntry_name	character varying
pop_cntry	numeric
geom	USER-DEFINED

pontok tábla:

gid	integer
osm_id	character varying
timestamp	character varying
name	character varying
type	character varying
geom	USER-DEFINED

waterways tábla:

gid	integer
osm_id	character varying
name	character varying