**Budapesti Műszaki és Gazdaságtudományi Egyetem**

**Vilamosmérnöki és Informatikai Kar**

# Párhuzamos kieső érték keresés MapReduce alapokon

Tudományos Diákköri Konferencia Dolgozat

2015.

***Szerző:***

Lux Zoltán András
Neptun kód: WAMP78

***Konzulens:***

Dr. Ekler Péter

# Tartalomjegyzék

# 1. Bevezetés

Az elmúlt évtizedekben a tárolt adat mennyisége folyamatosan nőtt, sok esetben akkora méretű adathalmazok keletkeztek, amik már csak nagy nehézségekkel vagy egyáltalán nem kezelhetőek hagyományos relációs adatbázisokban. A hatalmas méretű adatok kezelésére jöttek létre a különféle *BigData* technológiák. Ezek közül jelenleg a legnépszerűbbek a különféle *Hadoop* alapú megoldások. A *Hadoop* egy nyílt forráskódú *Apache* projekt, tekinthető egy *BigData* management és elemző szoftver családnak, illetve sokan hívják a *BigData* elemzés operációs rendszerének. Egy *Hadoop* klaszter több ezer gépig is remekül skálázódik, *Petabyte* méretű adatok elemzésére is alkalmas. Fontos azonban kiemelni, hogy a *Hadoop*-os adatelemzés speciális lépéseket igényel, az általa használt *MapReduce* alapú programozási modell használatával automatikusan párhuzamosítható programok futtatását teszi lehetővé. Az automatizáltságnak azonban ára van, ez bizonyos megkötésekkel jár, és nem is minden megoldás valósítható meg rajta. Ez *természetesen* azt jelenti, hogy az adatelemzési és adatbányászati algoritmusokat újra implementálni kell a *MapReduce* paradigma szerint. Számos elemző algoritmus illetve módszer már elérhető nyílt forráskódú könyvtárakban, azonban még mindig sokkal kevesebb algoritmust tartalmaznak ezek mint például a hagyományos adatbányász és statisztikai szoftverek és könyvtárak. Napjaink egyik fontos adatbányászati feladata az *outlier* keresés. Ennek segítségével detektálhatunk többek között bankkártyás csalásokat, webes támadásokat, meghibásodásokat vagy sokféle anomáliát általában. Számos kieső értékeket észlelő algoritmus van, ezek több családba sorolhatók, attól függően, hogy miben nyilvánul meg az adatok egymástól és az adathalmaztól való különbözősége. Kutatásomban ezek közül a távolság és sűrűség alapú megközelítéseket választom, specifikusan a *DB(ε,π)-outlier modellt* és a *Local Outlier Factor* algoritmust. A távolságok megfelelő pontoktól való mérésének elvégzéséhez érdemes egy olyan elosztott adatszerkezetet létrehozni, ami az egyes pontokhoz térben közel található pontok hatékony megtalálását támogatja. Ez *Hadoop* klaszter esetén azt jelenti, hogy az egymáshoz közel elhelyezkedő pontok, amik esetében a távolságokat ki kell számolnunk,általában egy gépen legyenek, illetve kedvezőtlen esetben egy következő *MapReduce Job*-ban megtalálhatóak legyenek. A kutatás eredménye felhasználható nagy adathalmazokon hatékony kieső érték kereséshez, például online csalások, webes támadások és egészségügyi adatok esetén.

# 2. A kitűzött feladat

A TDK dolgozatomban *MapReduce* szemléletű skálázható implementációját tűztem ki közelség alapú kieső érték kereső algoritmusoknak, nevezetesen a *DB(ε,π)* outlierek keresését, valamint a *Local Outlier Factor* nevű algorimtus implementációját. Mind a két típusú outlier keresést hatékonyabbá teszi az valamilyen térbeli partícionálás használata, mivel mind a két algoritmus esetében az egyes pontok bizonyos számú szomszédjának megtalálása szolgál a kieső érték keresés alapjául, így a két algoritmus implementációja sok közös elemet tartalmaz, még ha némileg másként közelíti is meg egy pont kiesőségének meghatározását. Az adatok térbeli partícionálását és a megfelelő szomszédok keresését a *MapReduce* paradigma megkötéseit követve valósítottam meg, így sikerült elérni a nagy mértékű skálázhatóságot egy átlagos asztali gépen már nem feldolgozható méretű generált adatokon. A két kieső érték kereső algoritmust sikeresen felhasználtam hálózati támadások felderítésére.

# 3. Irodalomkutatás

Adatbányászati algoritmusok, illetve az ezekhez kapcsolódó hasznos adatstruktúrák elosztott, *MapReduce* alapú implementációjával számos cikk foglalkozik, plusz ezen algoritmusok más megközelítésű párhuzamosításával még többen és jóval régebb óta. Az általam vizsgált kieső érték kereső algoritmusok *MapReduce* alapú megvalósításához használt térbeli partícionálást a „Distributed Kd-Trees for Retrieval from Very Large Image Collections” [4] cikkből kiindulva valósítom meg. A [4] számú cikkben egy elosztott KD fát építenek fel, ami segítségével hatékonyan lehet több dimenziós pontok közeli szomszédait meghatározni. Az egyik módszer szerint az adatok nem kerülnek partícionlásra, az egyes gépeken egymástól független KD fákat építenek fel. A mi esetünkben érdekes második módszer mintavételezés segítségével felépíti az elosztott fastruktúra képzeletbeli tetejét, és az egyes gépeken csak részfákat tárol. Az elosztott *KD fa* tetejét használtam az adatok partíciókra osztásába, amit a [4] cikkben javasolt mintavételezéses módszerrel építettem fel. A távolság alapú *DB(ε,π) kieső érték modell* szerinti kieső érték keresés nagyban hasonlít a DBSCAN nevű klaszterezési algoritmusra, mivel a DBSCAN által zajnak minősített pontok felelnek meg a *DB(ε,π) kieső értékeknek*. Tehát témába illőnek találtam az „MR-DBSCAN: a scalable *MapReduce*-based DBSCAN algorithm for heavily skewed data” [5] cikket is. A DBSCAN *MapReduce* alapú implementációja során is az adatok térbeli eloszlásuk szerint lettek partícionálva (a [4] es cikkben említettől lényegesen eltérő módszerekkel), majd az egyes partíciókon belül szekvenciális DBSCAN algoritmust hajtottak végre. DBSCAN esetén a lokális klaszterezés nem elég, egy globális klaszter struktúra kialakítása szükséges a lokális eredmények felhasználásával, viszont ez az én esetemben nem szükséges, mivel én csak a kieső „noise”-nak megcímkézett értékeket keresem. A *Local Outlier Factor* alapú kieső érték keresés gyakorlati felhasználása során a „Data Mining for Network Intrusion Detection” [6] és a “A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection” [7] adnak támpontot és szolgálnak egyben referencia eredményként is. Illetve a kieső érték keresés ipari felhasználhatóságának vizsgálásához választott KDD Cup 1999 adathalmaz minőségének elemzéséhez, az ehhez hasonló módon létrehozott, ámde nem azonos adathalmaz hasznosságát tárgyaló „Usefulness of DARPA Dataset for Intrusion Detection System Evaluation” [7] publikáció is segít. A két kieső érték kereső algoritmus közül a *Local Outlier Factor* algoritmus a „OPTICS-OF: Identifying Local Outliers” [2], majd a végleges formája a „LOF: Identifying Density-Based Local Outliers” [1] cikkben lett publikálva, az utóbbi a „SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data” konferencia kiadványának a 93-104 oldalán is megtalálható. Az vizsgált algoritmusok a „Data Mining: Concepts and Techniques” [10] című könyv outlier kereséssel foglalkozó fejezetében is megtalálhatóak.

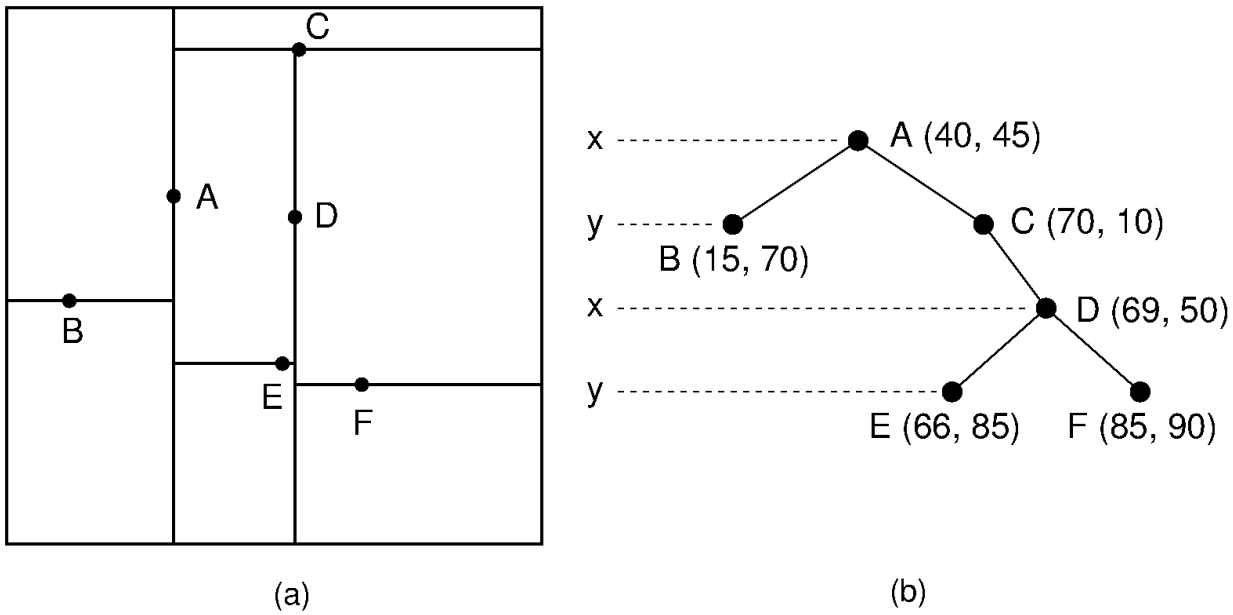
# 4. Adatstruktúrák

## 1.1 KD fa

A hatékony epszilon környezetben található szomszédok megtalálásához érdemes megfelelő adatstruktúrát választani, amivel ez a szekvenciális *O(n)* komplexitású keresésnél gyorsabb lesz. Ehhez érdemes valamilyen fa struktúrában gondolkozni. A *KD fa* a bináris fa egyfajta általánosítása több dimenzióra, ami átlagosan *O(k*log(n))* komplexitású k legközelebbi szomszéd lekérdezést tesz lehetővé, legrosszabb esetben *O(n)* idő alatt. Az epszilon szomszédság lekérdezés komplexitása nagyban függ az ezen a térrészben található pontok számától, azonban átlagosan ez is sokkal gyorsabban végrehajtható így, mint szekvenciális kereséssel. Szerencsére nem kell feltétlen az epszilon szomszédság keresést

használni, mivel megadott számú pontot keresünk ezen térrészen belül, azaz a *k legközelebbi szomszéd* lekérdezés is megfelelő.

Egy 2 dimenziós *KD fa* felépítését remekül lehet vizuálisan is szemléltetni.




*Ábra 1 - KD fa*

Ez használható lesz a partíciók esetén is, annak ellenére, hogy a pontok egy részénél a k legközelebbi szomszédtól való távolság nem feltétlen lesz globálisan helyes, mivel csak az számít, hogy nagyobb-e ez a távolság a megadott epszilon értéknél, és a partíciók kiterjesztése miatt ez mindig teljes bizonyossággal megadható. Azaz azt minden esetben biztosan meg tudjuk mondani, hogy az adott pont epszilon távolságán belül kellően sok pont található-e, vagyis kieső értéknek számít-e az éppen vizsgált pont.

A *KD fa* esetén lényeges lehet a fa felépítéséhez szükséges idő is, ha minél kiegyensúlyozottabb fát akarunk felépíteni. Az adatok minél egyenletesebb elosztásához a fában rendezésre van szükség, amihez *O(n*log(n))* komplexitású *merge sort*-ot használok.

A fa magassága kb. log(n), mivel a fa leveleiben általában viszonylag kevés pontot tárolunk. Az előbbiek alapján jelen esetben a fa konstrukciójának az ideje $O(n*log^2(n))$ lesz. A fa konstrukció rosszabbul skálázódik mint a k legközelebbi szomszéd keresés, ha k értéke kicsi, azonban bizonyos méretig jellemzően mégis rövidebb ideig tart. Az utóbbi két állítást az adatstruktúra implementálása során én is tapasztaltam, a futási idők ezt igazolták.

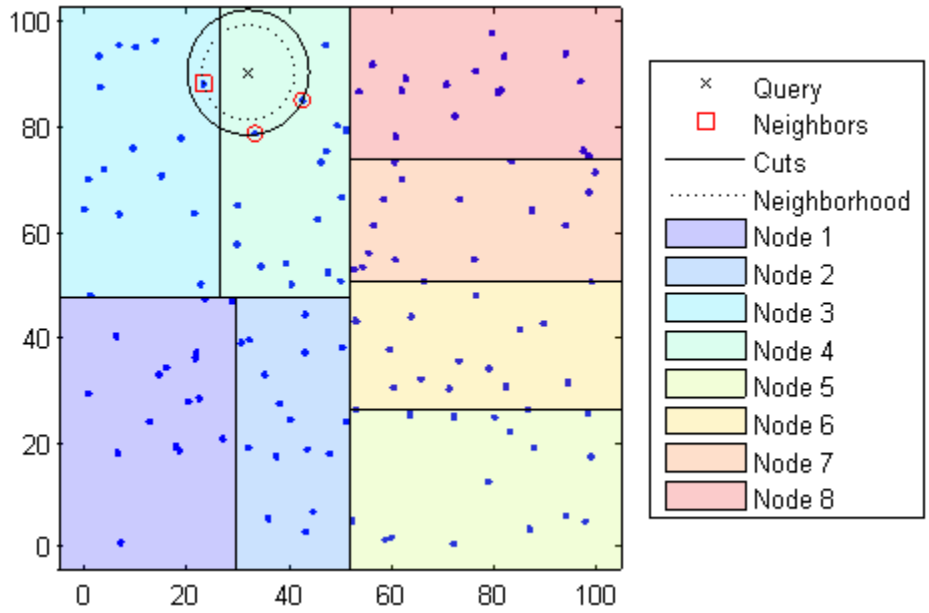
A *KD fa* ezen felül tartalmaz még beszúrás és törlés műveletet. Mind a kettő *O(log(n))* komplexitású. Beillesztésre csak a konstrukció folyamán van szükség jelen esetben, törlésre pedig egyáltalán nincs.

A K szomszédság definíciója fontos fogalom a továbbiakban, érdemes lehet a *k távolságon* keresztül megragadni:

*Egy adott pont k legközelebbi szomszédjának (k szomszédság) nevezzük azokat a pontokat, amik az adott ponttól maximum akkora távolságra találhatóak mint a k távolság.*

*P pont k távolsága az a legkisebb távolság, amire teljesül, hogy legalább k darab pont maximum ekkora távolságra található P ponttól.*

A k szomszédok megtalálását *KD fa* segítségével az alábbi kép szemlélteti:




*Ábra 2 - KD fa használata*

Az előbbi definíció jelentősége, hogy egy P pont szomszédjai lehetnek egyenlő távolságra P-től, és ha ez az eset pont a k távolság értékénél áll elő, akkor P pont *k szomszédsága* nagyobb lesz a k értéknél. Ennek az lesz a következménye, hogy szélsőséges esetben a *k szomszédság* rendkívül nagy lehet, akár az egész adathalmaz méretéhez képest sem elhanyagolható.

A *k legközelebbi szomszéd* keresése során tudnunk kell, hogy keresünk-e még szomszédokat, azaz megtaláltuk-e már a k legközelebbit. Az én KDFa implementációm esetében ebben a jelenlegi az aktuális *k távolság* és az egyes részfák, határoló egyenesei nyújtanak segítséget. Ha a jelenlegi *k távolság* az adott ponttól nem nyúlik be egy belső pont határvonalai közé, biztosak lehetünk benne, hogy az itt lévő pontok nem tartoznak bele a *k szomszédságba*. Ehhez érdemes kezdetben kellően nagy *k távolság* értéket választani, majd minden esetben aktualizálni, ha a *k szomszédság* mérete elérte a megengedett méretet. Fontos megjegyezni, hogy bizonyos esetekben érdemes lehet a kerekítési hibákra is odafigyelni, mivel azonos távolságra lévő pontok ezek miatt a hibák miatt eltérően távolinak mutatkozhatnak, holott egyenlő távolságra helyezkednek el az éppen vizsgált ponttól. Ez pl. abban az esetben vezethet téves eredményhez, ha a *k szomszédság* több mint k pontból áll, és pont a távolság szerinti növekvő sorrendben k-adik elem után még több egyenlően távoli pont létezik. Ez a probléma akár abban az esetben is előjöhet, amikor a határvonalaktól mérjük a távolságot.

Továbbá érdemes megjegyezni, hogy a *KD fa* kiegyensúlyozottsága, azaz a teljesítménye is, nagyban függ az adatoktól. Ha az adathalmaz kellően sok azonos értéket tartalmaz, nem lehet kiegyensúlyozott KD fát felépíteni, és ezáltal többek között a szomszéd keresést sem lehet kellően szűkíteni. Minél kevésbé

kiegyensúlyozott a fa, a szomszéd keresés teljesítménye átlagosan annál közelebb kerül az *O(n)* komplexitású szekvenciális kereséséhez. Ezen esetleg lehet segíteni például az azonos pontok csoportként kezelésével, de mindenképpen extra lépéseket igényel az adatok előkészítése terén, ha térben kellően azonos elhelyezkedésű pontokat hatékonyan akarunk keresni *KD fa* segítségével.

## 1.2 Hierarchikus rács KD fa alapján

A partícionáláshoz felhasznált hierarchikus térbeli rács a [4] alapján készült, az itt említett elosztott *KD fa* képzeletbeli tetejének felel meg lényegében, illetve az igényeinknek megfelelően kiegészíti például az ε környezet figyelembevételével.

A hierarchikus rács is lényegében tekinthető a partíciókra egyesével felépített KD fák tetejének, azonban ezt a tulajdonságát csak annyiban fogom kihasználni amennyire szükséges. A rács alapvető feladata a pontok partícionálása, utána már nincs szükség az ismeretére. A rács nem tárolja el azokat az elemeket, amibe beillesztünk, hanem csak egy hiperkockát, ami ezeket a pontokat körülöleli. Fontos megjegyezni, hogy a rács legszélső határait külön meg kell adni, mivel a mintavételezés miatt meg kell adni ennél a megközelítésnél, nehogy legyenek olyan pontok amik a rácson kívül helyezkednek el, amikor kialakítjuk a partíciókat. Ha nem ismernénk a szélső értékeket, akkor is lehetne a legközelebbi rács cellába tenni a kívül eső pontokat, ez viszont egy kevésbé egyenletes adat eloszlást eredményezne a partíciók között.

A rács konstrukciója a *KD fa* konstrukciójával lényegében teljesen megegyezik, azzal a különbséggel, hogy sokkal kisebb lesz a mélysége a pontok számához képest. Erre azért van szükség, hogy az egyes levél elemekbe kerülő összes pont száma a teljes adathalmazon a lehető legnagyobb mértékben megegyezzen. Fontos még, hogy itt nem mentjük el a pontokat a rácsba, ezekre csak a létrehozáshoz van szükségünk.

A rács konstrukciója után az a funkciója, hogy megmondja melyik pont melyik partícióba fog kerülni (igény szerint ε környezettel kiegészítve), valamint a *LOF* algoritmus *k* szomszédság lekérdezései közül a több partíciós keresések során lesz fontos szerepe.

A rács pontok partíciókba osztásánál lehetővé teszi, hogy egy pont több partícióba kerüljön, ha megadunk egy epszilon értéket, ami egy adott partíció epszilon távolsággal kibővített környezetében lévő pontokat sorolja ide, illetve extrém esetben pont a határvonalon elhelyezkedő pontokra is minden érintett levél rácsot visszaad.

A mintavételezés alkalmazásánál fontos, hogy megfelelő számú ponttal dolgozzunk. Ha túl nagy a minta, akkor nem férhet el a memóriában, ha túl kicsi nem lesz egyenletes az adatok eloszlása. Szerencsére a tapasztalatok alapján az adat méretéhez képest elhanyagolhatóan kicsi méretű minta is eddig minden esetben elegendőnek bizonyult, és kellően azonos méretű partíciókhoz vezetett. Ezzel együtt egy rossz minőségű rács elkészítése nem kizárható, de megfelelő mintavételezés esetén ez a helyzet nagyon kis valószínűséggel áll elő. Ennek a valószínűségnek a számítása rendkívül nehéz lehet, ha az adatok térbeli eloszlása nem ismert. Mivel ez egy kritikus lépésnek számít az algoritmus végrehajtása során, a munkám további részében is ki fogok térni erre, mérési eredményekkel alátámasztva a módszer hatékonyságát.

A rács kedvező tulajdonsága, hogy az adat méretéhez képest elhanyagolható méretű, mivel a levél cellák nagy számú pontot ölelnek körül, és csak az egyes dimenziókbeli legszélső koordinátáit tárolják el.

# 2. Kieső értékek

***Definíció**: A kieső értékek tekinthetők olyan adatoknak, amik annyira eltérnek az adathalmaz többi elemétől, hogy őket valószínűleg egy másféle, eltérő mechanizmus generálta, mint az adathalmaz átlagosnak tekinthető elemeit.*

Ennél fogva tekinthetők hibás értékeknek vagy zajnak, de bizonyos esetekben a kieső értékek akár a legérdekesebb elemei lehetnek az adathalmaznak. Tipikusan ilyen területek a csalás felderítés, vagy egészségügyi adatokban rendellenességek keresése.

# 3. Közelség alapú kieső érték keresés

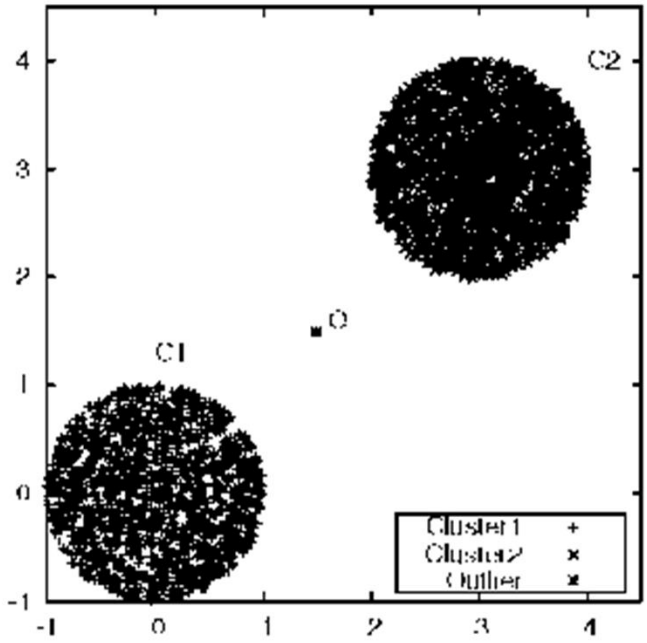
A közelség alapú outlier kereső algoritmusok két további csoprtba oszthatók, név szerint távolság és sűrűség alapúakra.

## 3.1 Távolság alapú kieső érték keresés

Távolság alapú kieső érték keresés esetén a k szomszédság méretét vesszük alapul. Azaz, ha az adathalmaz egy bizonyos része nem található kellően közel a vizsgált ponthoz, azt kiesőnek tekintjük.

### 3.1.1 DB(ε,π) kieső értékek

A [3] *DB(ε,π) kieső érték modell* jó eséllyel a legismertebb távolság alapú *outlier* kereső módszer. A *DB(ε,π)* modell szerint azok a pontok tekinthetőek kieső értéknek, amik ε környezetükben nem tartalmazzák a teljes adathalmaz legalább π hányadát.




*Ábra 3- Távolság alapú kieső érték*

A *DB(ε,π)* modell lehetővé teszi olyan kieső értékek megtalálását, amik értéke semmilyen dimenzióban sem számítnak extrém nagy vagy kis értéknek, hanem egy bizonyos méretű környezetük nem tartalmaz kellő számú pontot, azaz összességében eltérnek az adathalmaztól. A modell ezen tulajdonsága különösen jól jön, ha mondjuk az adatok nem kimondottan illeszkednek rá egy ismert eloszlásra.

## 3.2 Sűrűség alapú kieső érték keresés

Sűrűségen alapuló outlier modellek esetén az egyes pontok, a bizonyos szomszédjaiktól függően definiált, valamilyen sűrűség értéke alapján számoljuk a kiesőséget.

### 3.2.1 A Local Outlier Factor algoritmus

A *Local Outlier Factor* (röviden *LOF)* algoritmus [1],[2] talán a legismertebb és jó eséllyel a legelső sűrűség alapú outlier kereső algoritmus. Az eredeti *OPTICS-OF* [2] elnevezése is a végleges *LOF* elődjének arra utal, hogy az *OPTICS* nevű sűrűség alapú hierarchikus klaszterezési algoritmus motiválta. Ehhez érdemes hozzátenni, hogy a *LOF* által alkalmazott, viszonylag több lépésből álló és relatíve komplexnek nevezhető, kieső érték számolás nem lelhető fel a klaszterezési algoritmusban. Azonban nagy hasonlóság, hogy sem az *OPTICS* sem a *LOF* nem ad egy egyértelmű klaszter struktúrát vagy kieső értékeket, hanem ennek végleges kialakítását az algoritmus felhasználójára bízza. Az említett véglegesítő lépést mind két esetben egyszerűen el lehet végezni, és a tárgyalt algoritmusok használatát rendkívül felxibilissé teszi. Ezen felül a *LOF*-ban szereplő elérési távolság (*reachability-distance*) is megtalálható *OPTICS* algoritmusban.

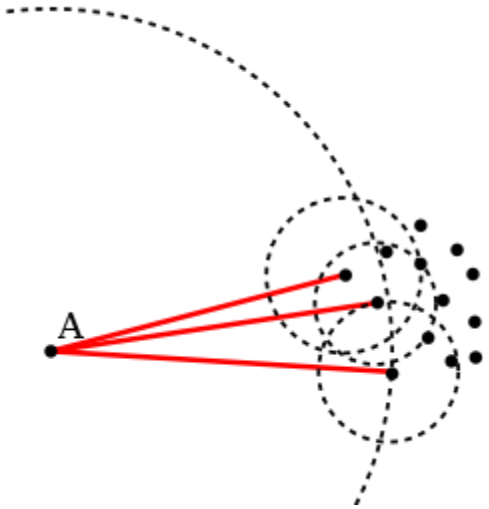
Az egyes pontok *LOF* pontszáma az alábbi képletek alapján lehet kiszámolni.

A pontnak B ponttól mért elérési távolsága a következő:

$$ReachabilityDistance_k(A,B) = \max\{kDistance(B), d(A,B)\}$$

ahol *d(A,B)* A és B pont távolsága, *k-distance(B)* pedig B pont *k távolsága*.

Az alábbi „Elérési távolság” című kép A pont elérési távolságait szemlélteti, és a szaggatott vonallal jelölt körök a *k távolságokat* jelölik (*k = 3* érték mellett).




*Ábra 4- Elérési távolság*

A *lokális elérési sűrűség* A ponthoz az alábbi módon számolható:

$$lrd(A) := 1/ \frac{\sum_{B \in N_k(A)} ReachabilityDistance_k(A,B)}{|\mathrm{N_k(A)}|}$$

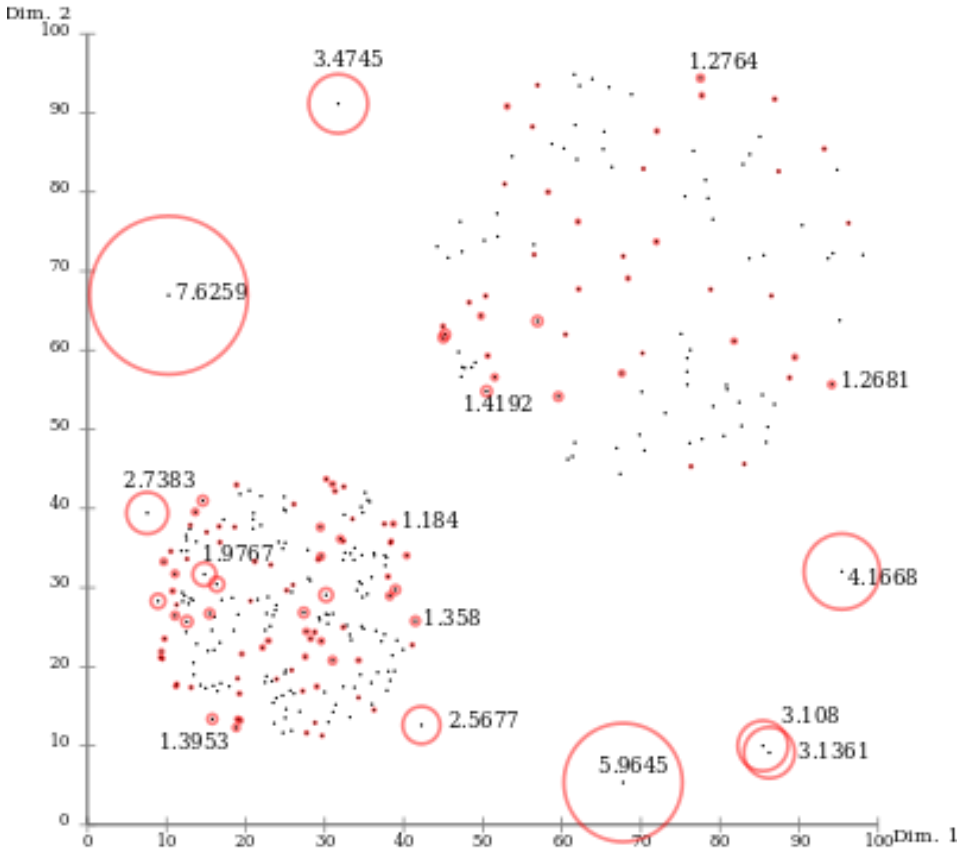
ahol $|\mathrm{N_k(A)}|$ a *k szomszédságba* tartozó pontok száma.

Végül a *LOF* pontszám értéke az alábbi képlet alapján kapható meg:

$$LOF_k(A) := \frac{\sum_{B \in N_k(A)} \frac{lrd(B)}{lrd(A)}}{|\mathrm{N_k(A)}|}$$

ahol $|\mathrm{N_k(A)}|$ a *k szomszédságba* tartozó pontok száma.

Az alábbi képen egy térben nem egyenletes sűrűségű adathalmazt láthatunk, ahol több eltérő sűrűségű csoport (klaszter) illetve más kieső értékek is vannak. A piros kör mérete a *LOF pontszámmal* van összefüggésben, illetve a nagy értékeket szám szerint is kiírja.




*Ábra 5 - LOF értékek*

A *Local Outlier Factor* algoritmus nagy előnye, hogy nem homogén sűrűségű adattérben is remekül lehet vele kieső értékeket keresni, mivel egy pont outlierségének a mértékét a szomszédjaihoz hasonlítva

mondja meg. Ezáltal a *Local Outlier Factor* gyakran remekül teljesít a való életből származó adathalmazokon.

A *LOF* algoritmus nem mondja meg egyértelműen, hogy egy pont kieső érték-e, hanem annak a mértékét mondja meg, hogy mennyire kieső érték. A képlete alapján, ha egy pont körüli *lokális elérési sűrűség* megegyezik a szomszédjainak az *lokális elérési sűrűségével*, akkor a *LOF pontszáma* egy körül lesz. Ha lényegesen kisebb az *lokális elérési sűrűsége*, mint a szomszédjainak, akkor jelentősen nagyobb lesz a *LOF pontszáma* 1-nél, azaz nagy mértékben kieső értéknek számít. Értelemszerűen, ha nagyobb a *lokális elérési sűrűség*e mint a szomszédjainak, akkor 1 alatti *LOF pontszámot* kap. Az eredmények ismeretében nekünk kell eldöntenünk mekkora pontszámtól minősítünk valamit kiugró értéknek, azonban pont ez teszi még flexibilisebbé az algoritmust. Fontos jól megválasztani  a vizsgált szomszédok számát. Ha túl kevés szomszédot nézünk könnyen lehet, hogy nem találjuk meg a kieső értékeket. Azonban ha túl sok szomszédot vizsgálunk, nem csak a futási idő lesz nagyobb, de elképzelhető, hogy az adathalmaz túl nagy és távoli részével is összehasonlítjuk az egyes pontokat, így az algoritmus szintén nem ad számunkra értékes eredményt.

# 4. Távolság alapú kieső érték kereső algoritmus párhuzamosítása

A *DB(ε,π)-outlier* modell szerinti kieső érték modellben az egyes pontokra végzett számítások egymástól függetlenek, nem függnek egymástól. Azonban nagy mértékben függnek az ε távolságon belül található szomszédok megtalálásának hatékonyságától. A hatékony szomszéd kereséshez érdemes a pontokat úgy szétosztani, hogy a térben közel található pontok egy partícióban legyenek. Fontos megjegyezni, hogy megkülönböztetünk belső és külső pontokat. A belső pontok a partíció határain belül találhatóak, a külső pontok pedig a partíciók határától epszilon távolságon belül, de a partíción kívül találhatóak. A partíciók létrehozásához az előbbiekben leírt hierarchikus rácsot használtam, ennek segítségével számunkra kedvező módon lehet partícionálni az adatokat, azaz minden egyes partíció kibővítve minden irányba epszilon távolsággal lehetővé teszi a térrészben található belső pontok epszilon környezetében található szomszédjainak a megtalálását. Ez abból a szempontból különösen kedvező, hogy a partíció alapból kisebb méretű a teljes adathalmaznál, és megfelelő epszilon paraméter megválasztásával a kibővített környezettel együtt is lényegesen kevesebb pontot tartalmaz az összes adatnál. A partíciókon belül hagyományos szekvenciális algoritmussal kereshetjük a kieső értékeket, annyit kiegészítéssel, hogy csak a belső pontok szomszédjait keressük.

# 5. *LOF* algoritmus párhuzamosítása

A *LOF* algoritmus három fázisra osztható:

- először a *k távolságok* meghatározására a *k* legközelebbi szomszéd megkeresésével,
- majd a *Local Reachability Density*, azaz *lokális elérési sűrűség* meghatározására, amihez az előző fázis eredményeivel kiegészített *k* legközelebbi szomszéd megtalálása szükséges
- végül a *LOF pontszám* kiszámítása az egyes pontokhoz, amihez ismét az előző fázis eredményeivel kiegészített k legközelebbi pont ismerete szükséges

A *LOF algoritmus* esetében hasonló módon párhuzamosítani lehet az algoritmus egyes fázisait, minimális kiegészítéssel. A teljes adathalmaz szintén partíciókra bontható, azonban a partíciókat nem kell meghosszabbítani epszilon környezettel. Viszont a legközelebbi szomszédjai a pontoknak, különösen a

partíciók szélén találhatóaknak, nem feltétlenül a partíción belül helyezkednek el, tehát partíciók közötti keresést is meg kell tudni valósítani. Azonban nagy partíciók esetén minden esetben meg tudunk találni lokálisan k legközelebbi szomszédot, azaz tudunk számolni egy lokális *k távolságot*. A lokális *k távolságnál* csak kisebb, vagy ezzel egyező lehet a globális *k távolság*, azaz az egész adathalmazra értelmezett *k távolság*. Ezt kihasználva megmondható, hogy a pontnak megtaláltuk-e a k legközelebbi szomszédját, ha megmérjük a partíciója határvonalaitól mért távolságait, és ezeket összehasonlítjuk a lokális *k távolságával*. Amennyiben a lokális *k távolság* kisebb, mint a bármelyik partíció-határ távolsága, akkor az eredmény globálisan helyes. Ha a *k távolság* átnyúl más partíciókba is, akkor ennek a pontnak a szomszédjait keresnünk kell azokban a partíciókban, amikbe átlóg a ponttól a *k távolság* sugarú hipergömb. Az adatok partícionálására használt hierarchikus térbeli rács segítségével megkereshetjük a k legközelebbi szomszéd kereséséhez szükséges partíciókat, és itt is megkeressük a k legközelebbi szomszédot, vagy optimálisabb esetben a *k távolságon* belül található részét az itteni k legközelebbi szomszédnak. Még egy harmadik lépésben össze kell gyűjtenünk a k legközelebbi szomszédokat az összes érintett partícióból az adott ponthoz, és ezek közül kiválasztjuk a globálisan is helyes k legközelebbi pontokat.

Tehát a *k* legközelebbi szomszéd keresésesét is 3 fázisra oszthatjuk:

- A partíciókon belül a lokális k szomszédok megkeresése, majd az egyes pontok saját partíciójuktól mért távolságuk felhasználásával megmondjuk, hogy a lokális *k szomszéd* globálisan is garantáltan *k szomszéd*-e
- Azoknak a pontoknak, amiknek lehetnek a *k szomszédságába* tartozó pontok más partíciókban is, megkeressük a *k legközelebbi szomszédját* azokban a partíciókban, amikbe átnyúl a pontok *k távolsága*
- Azoknak a pontoknak, amiknek nem csak a saját partíciójában kellett a *k szomszédjait* keresni, aggregálni kell az érintett partíciókból származó *k szomszédjait*, és ezek közül kiválasztani a globális *k szomszédságot*

A *k* legközelebbi pont megtalálása után minden fázisban elvégezhetőek a szükséges számítások, az előző fázis eredményeivel kibővített szomszédok esetében. A legelső fázisban természetesen nincs szükség korábbi részeredményre.

A *k szomszéd* keresés pl. Abban lenne másként elvégezhető másként a második és harmadik fő fázisában a *LOF algoritmusnak*, azaz az *LRD (lokális elérési sűrűség)* és *LOF pontszámok* számítása alatt, ha a *k távolságok* meghatározása során minden egyes ponttal együtt tároljuk azt is, hogy ő kiknek tartozik bele a *k szomszédságába*, majd a szomszédokat juttatjuk el valamilyen módon az egyes pontokhoz. Mivel az algoritmus komplexitását nem befolyásolja az előbb leírt három fázisú szomszéd keresés, mivel konstans háromszor kell megismételni, ezért a későbbiekben ennél az egyszerűbb módszernél maradtam.

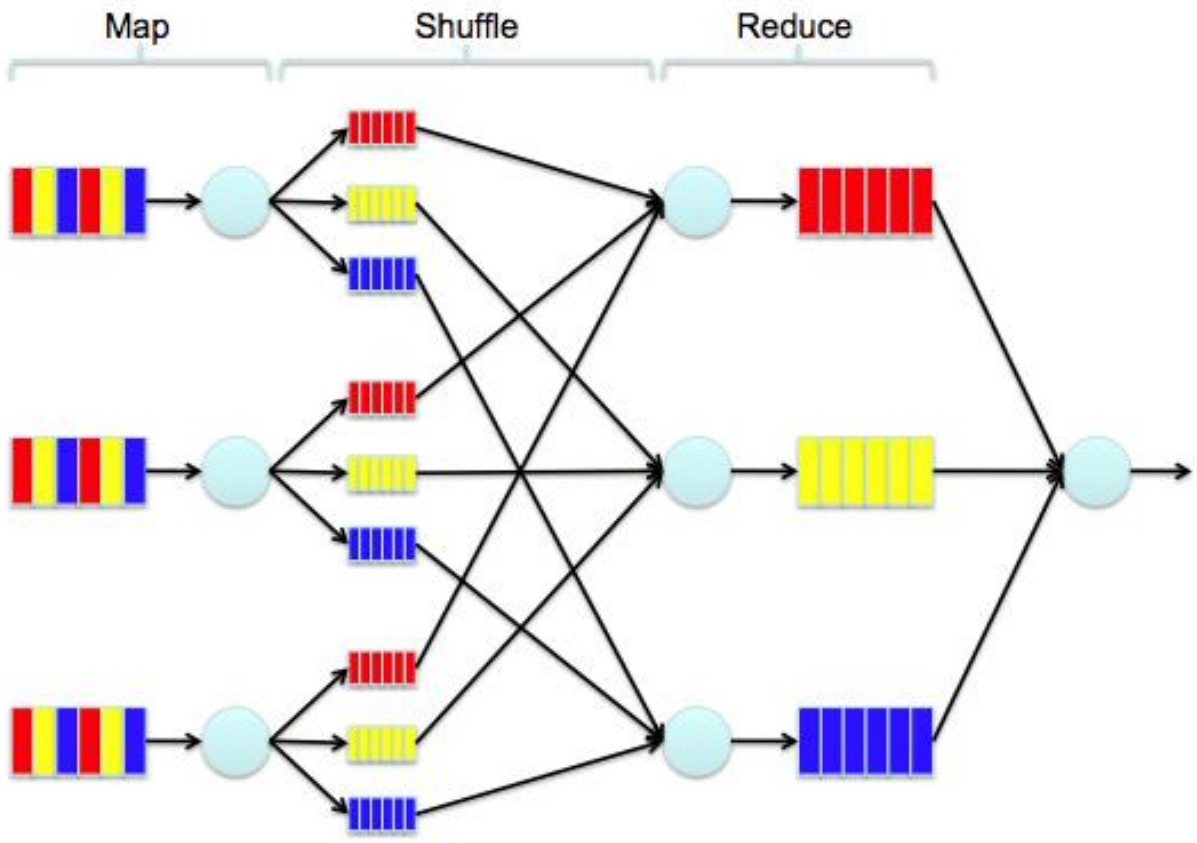
# 6. MapReduce alapú kieső érték keresés

## 6.1 A MapReduce paradigma ismertetése

A *MapReduce* programozási modell alapvetően két részének ismerete feltétlenül szükséges, ha valaki ezt használni akarja, a Map és a Reduce. A Map párhuzamosan végrehajtható a bemeneti adatok minden egyes elemére, és ez alapján visszaad egy kulcs-érték párost. Aztán a keretrendszer kulcsok szerint rendezi

és aggregálja a Map függvényekből kijövő értékeket. A Reduce részben az egyedi kulcsokat az összes hozzájuk tartozó érték listájával együtt kapjuk meg, majd ezeken végzünk számunkra tetsző műveleteket. Mivel az egyes Reduce feladatok között szintén nincs semmilyen egymásra hatás, ezek is párhuzamosan végezhetőek el. Ezzel a programozási modellel számos adatelemzési problémát hatékonyan párhuzamosítani lehet, azonban akadnak olyanok is, amik nem oldhatóak meg hatékonyan a MapReduce paradigma felhasználásával. A Map és a Reduce fázis között az értékek kulcsok szerinti rendezésére és csoportosítására van szükség, amit a jelenleg tárgyalt paradigmát megvalósító keretrendszernek biztosítania kell. Ezen felül lehetőség van lokális Reduce elvégzésére is, ha a Reduce által megvalósított redukció disztributív, azaz az azonos kulcsú értékek egy részhalmazán elvégzett redukciók végeredményét újra redukálva helyes eredményt kapunk, azaz a Reduce fázis az adatok részhalmazán is elvégezhető. Az alább ismertetett lokális Reduce-t Combine-nak szokás hívni, és az egyes megvalósításokban nem kell teljesen megegyezni a Reduce fázissal az általa elvégzett műveletek terén, így a fejlesztő nagyobb szabadságot kap. Illetve a Map, Reduce és Combine egyfajta Filter szerepet is elláthat, mivel nem kell feltétlenül kimenetet létrehozniuk, de természetesen egy bemenetre több kimenetet is létrehozhatnak.

A *MapReduce* működését vázlatosan szemlélteti az alábbi kép:




*Ábra 6- MapReduce szemléltetése*

Az egyes színek az egyes kulcsokat jelölik, a világoskék körök pedig az egyes Mappereknek és Reducereknek feleltethetőek meg.

Sok esetben fontos lehet MapReduce végrehajtása során, hogy valamilyen minimális mennyiségű extra információval rendelkezzünk, mint például egy vagy több paraméter. Ezeket általában különösebb nehézség nélkül el lehet juttatni szükség esetén az egyes fázisokba, *Hadoop* esetén például a *Job* konfigurációba, vagy a *Hadoop Distributed Cache* elérésével be lehet állítani.

## 6.2 A párhuzamos kieső érték kereső algoritmus leképezése a MapReduce modellre

A kieső érték keresés számos fázisa mind a sűrűség alapú *LOF algoritmus*, mind a távolság alapú kieső érték keresés csoportjába tartozó *DB(ε,π)* modell esetén azonos vagy hasonló.

Első lépésben partícionálnunk kell az adatokat, a hierarchikus térbeli rács segítségével. Mivel az egész adathalmaz könnyedén túl nagy lehet ahhoz, hogy egy gép memóriájában elférjen, ezért mintavételezéssel egy kellően nagy méretű, de a memóriában még könnyen kezelhető mintát kell létrehozni. Ehhez érdemes fix méretű mintát generálni, mivel ennél könnyebben meg tudjuk mondani, hogy mennyire fér el egy gép memóriájában, mint mondjuk ha az adatok egy bizonyos százalékával nagyjából megegyező méretű mintával dolgozunk, főleg ha nem tudjuk pontosan mennyi adattal is dolgozunk. Ráadásul az adatpontok mérete a diszken nem feltétlen egyezik meg a memória igényükkel.

Én az implementációmban a konstans méretű minták létrehozásához Heap-et használtam, amiben az adatok random generált prioritást kaptak. Ha a Heap túllépi a megengedett méretet, akkor annyi elemét eltávolítjuk, hogy újra megfelelő legyen a mérete.

*MapReduce* modellben a Map egyszerűen továbbadja a pontokat egy azonos kulccsal, amiken aztán a Combine és Reduce fázisokban Heap segítségével mintákat lehet generálni. A Reduce fázisban minden ha ismerjük az adattér kereteit, már minden információ a rendelkezésünkre áll, hogy létrehozzuk a térbeli rácsot, ezért ez azonnal elvégezhető közvetlenül a mintavételezés után a Reduce fázisban.

A megfelelő rács felépítéséhez ismerni kell a teljes adathalmaz kereteit, azaz minden dimenzióban a maximális és minimális koordinátákat. Ez könnyedén elvégezhető még a mintavételezés előtt, egy Map-Combine-Reduce fázisban.

A határoló egyenesek meghatározásához szükséges MapReduce pszeudokód:

```
Map:
            Emit(0,Pont)

      Output Key: 0
      Output Value: Pont
```

```
Combine:
      Input Key: 0
      Input Values: Pontok

            MinPont
            MaxPont
            Foreach Pont in Pontok:
                  Foreach dimenzió in Pont:
                        MinPont és MaxPont  aktualizálása d dimenzióban

      Output Key: 0
      Output Value 1: MinPont
      Output Value 2: MaxPont
```

```
Reduce:

      Input Key: 0
      Input Values: Pontok

            MinPont
            MaxPont
            Foreach Pont in Pontok:
                  Foreach dimenzió in Pont:
                        MinPont és MaxPont  aktualizálása

      Output Key: 0
      Output Value 1: MinPont
      Output Value 2: MaxPont
```

Az adattér szélső értékeinek ismeretében mintavételezéssel meghatározhatjuk az egyes partíciókat meghatározó hierarchikus térbeli rácsot. Ennek a *MapReduce* pszeudokódja:

```
Map:
      Emit(0,Pont és random prioritás)

      Output Key: 0
      Output Value: Pont és random prioritás
```

```
Combine:

      Input Key: 0
      Input Values: Pontok és a prioritásuk

            Heap létrehozása
            Foreach Pont in Pontok:
                  Ha( heap méret <= megengedett):
                        pont illesztése a heapbe a prioritásával
                  Különben:
                        remove min művelet végrehajtása a heapen
            Foreach Pont in Heap
                  Emit(0, Pont és a prioritása)

      Output Key: 0
      Output Values: pontok a prioritásukkal
```

```
Reduce:
      Input Key: 0
      Input Values: Pontok és a prioritásuk
            Heap létrehozása
            Pontok Listája

            Hierarchikus térbeli rács(Felső keret,Alsó keret)
            Foreach Pont in Pontok:
                  pont illesztése a heapbe a prioritásával
                  Ha( heap méret > megengedett):
                        remove min művelet végrehajtása a heapen

            Foreach Pont in Heap:
                  Pont beillesztése a listába
                  Hierarchikus térbeli rács létrehozása a listából és a
                  keretek ismeretében

      Output Key: 0
      Output Values: hierarchikus térbeli rács (szerializálva)
```

Miután létrehoztuk az adatok partícionálására használt hierarchikus térbeli rácsot mintavételezéssel, már csak el kell osztani a teljes adathalmazt a létrehozott partíciók között. Ennek a *MapReduce* pszeudokódja az alábbi módon néz ki:

```
Map:

      HTR: Hierarchikus térbeli rács
      Jó Partíciók <- Pontot tartalmazó levél elemek a HTR-ből
      Foreach Partíció in Jó Partíciók:
            Emit(partíció ID, Pont)

      Output Key: partíció ID
      Output Value: Pont
```

```
Reduce:

      Input Key: partíció ID
      Input Values: Pontok
            Foreach Pont in Pontok:
                  Emit(partíció ID, Pont)

      Output Key: Partíció ID
      Output Values: Pontok
```

## 6.3 Távolság alapú kieső érték keresés specifikus rész

A távolság alapú kieső érték keresés *MapReduce* pszeudo kódja:

```
Map:
      Emit(partícióID,Pont)

      Output Key: partícióID
      Output Value: Pont
```

```
Reduce:
      Input Key: partíció ID
      Input Values: Pontok

            Epszilon
            Pí
            PontLista: Pontok összegyűjtése
            KDFa konstrukció a PontLista elemeiből
            Foreach Pont in Pontlista:
                  Ha(Pont belső pont):
                        //Pí itt egy egész szám, nem pedig %
                        K távolság : KDFa.kTávolság(Pont,Pí)
                        Ha( K távolság > epszilon):
                              Emit(partícióID,Pont)

      Output Key: partíció ID
      Output Values: Pontok
```

## 6.4 Local Outlier Factor specifikus rész

A *LOF algoritmus MapReduce* alapú megvalósítása az általánosságban leírt *LOF* párhuzamosítással teljesen megegyezően 3 fázisban számolja ki a *LOF* értékeket, illetve az egyes fázisok szintén 3 részfázisra bonthatók, ami során 3 alkalommal megkeressük az épp szükséges adatokkal kibővített k legközelebbi szomszédot. Mivel a 3 fő fázis, azaz a *k távolságok*, a *lokális elérési sűrűség* (Local Reachability Density), és a *LOF* pontszámok kiszámolása csak minimálisan tér el egymástól, konkrétan abban, hogy épp mit számolunk ki, ezért általánosan fogom leírni a 3 főfázist.

A 3 főfázis általánosított pszeudokódja:

Az első fázisban megkeressük minden pont partícióján belüli k legközelebbi szomszédját, és megvizsgáljuk, hogy szükséges e további szomszéd keresés.

Fontos megjegyezni, hogy a pontok nem csak a pontok koordinátáit, hanem extra adatokat is tárolhatnak, mint pl. Azonosító, eredeti partíció azonosító, k-távolság, esetleg szomszédok, illetve a *lokális elérési sűrűség* és a *LOF pontszám* is végül.

```
Map:
      Emit(partícióID,Pont)

      Output Key: partícióID
      Output Value: Pont
```

```
Reduce:
      Input Key: partícióID
      Input Values: Pontok

      PontLista : Pontok összegyűjtése
      KDFa konstrukcó a PontLista elemeiből
      K: K-szomszédság (minimum) mérete
      Foreach Pont in Pontlista:
            K-szomszédok: KDFa.Kszomszédság(Pont,K)
            K-távolság: távolság a legtávolabbi K-szomszédtól
            HatárTávolság: távolság(Pont, KDFa legközelebbi határvonala)
            Ha( K-távolság >= HatárTávolság ):
                  Pont.k-szomszédok : K-szomszédok
                  Emit(partíció ID, Pont + nincs kész)
            Különben:
                  //A számításokat el lehet végezni, pl. LRD, LOF pontszám
                  //vagy k-távolság
                  Számítás(Pont, K-Szomszédok)
                  Emit(partíció ID, Pont + kész)

      Output Key: partíció ID
      Output Values: Pont + [ kész | nincs kész ]
```

A második fázisban a még nem kész *k szomszédságú* pontok k szomszédait keressük az érintett partíciokban:

```
Map:
      HTR: HierarchikusTérbeli rács
      Ha Pont nincs kész:
            Particiók: HTR.tartalmazóPartíciók(Pont, Pont.k-távolság)
            Foreach Partíció in Partíciók:
                  Ha Pont eredeti partíciója:
                        Emit(partícióID, Pont + vendég)
                  Különben:
                        Emit(partícióID,Pont + eredeti)
      Különben:
            Emit(partícióID, Pont + eredeti)

      Output Key: partíció ID
      Output Value: Pont + [eredeti | vendég ]
```

```
Reduce:
      Input Key: partíció ID
      Input Values: Pontok + a státuszuk ( [Eredeti | Vendég ] )

      VendégPontLista: Vendég Pontok összegyűjtése
      EredetiPontLista: Eredeti Pontok összegyűjtése
      KDFa: konstrukció EredetiPontLista elemeiből
      K: k-szomszédság mérete
      Foreach Pont in VendégPontLista:
            k-szomszédok: KDFa.k-szomszédság(Pont,K)
            Pont.k-szomszédok : k-szomszédok
            Emit(partíció ID, Pont + nincs kész)

      Output Key: partíció ID
      Output Value: Pont + státusz: [ nincs kész]
```

A harmadik és egyben utolsó fázisban a több partícióban való keresést igénylő pontok szomszédit gyűjtjük össze minden ponthoz, és ez alapján határozzuk meg a tényleges k szomszédokat, valamint elvégezzük a főfázisra jellemző számolást (*k-távolság*, *LRD* vagy *LOF* pontszám).

```
Map:
      Ha a pont nincs kész:
            PontID : Pont.ID
            Emit(PontID,Pont)

      OutputKey: PonID
      Output Value: Pont
```

```
Reduce:
      InputKey: PontID
      InputValues: Pontok

      ÖsszesSzomszéd
      Foreach Pont in Pontok:
            Foreach Szomszéd in Pont.Szomszédok:
                  ÖsszesSzomszéd.Insert(Szomszéd)

      K-Szomszédok: ÖsszesSzomszéd.K-Szomszédság kiszűrése
      Pont.K-Szomszédok : K-Szomszédok
      //K-Szomszédok ismeretében K-távolság, LRD és LOF érték számolása
      Számítás(Pont, K-Szomszédok)
      PartícióID : Pont.PartícióID
      Emit(PartícióID, Pont + kész)
```

```
    OutputKey: PartícióID
    OutputValue: Pont + státusz ([ kész ])
```

# 7. Az Outlier kereső algoritmus megvalósítása Hadoop-on

## 7.1 Hadoop ismertető

A *Hadoop* jelenleg az egyik, ha nem a legnépszerűbb, open source big data szoftver. Sokan tekintik a big data elemzés operációs rendszerének, mivel egy elosztott filerendszert és saját ütemezőt is megvalósít. *Hadoop* segítségével hatékonyan lehet nagy adathalmazokat kezelni, feldolgozni és elemezni. Egy *Hadoop* klaszter több ezer node-ig is skálázható, és Petabyte-os méretű adathalmazok tárolásával és kezelésével gond nélkül megbirkózik. Mivel a file-okat alkotó egyes adatblokkokat redundánsan több gépen is tárolja, az adatvesztés valószínűsége gyakorlatilag elhanyagolható. Az egyes gépek kiesését automatikusan kezeli, és az egyes adat blokkok több példányban való elosztott tárolása miatt ez kiesést sem okoz.

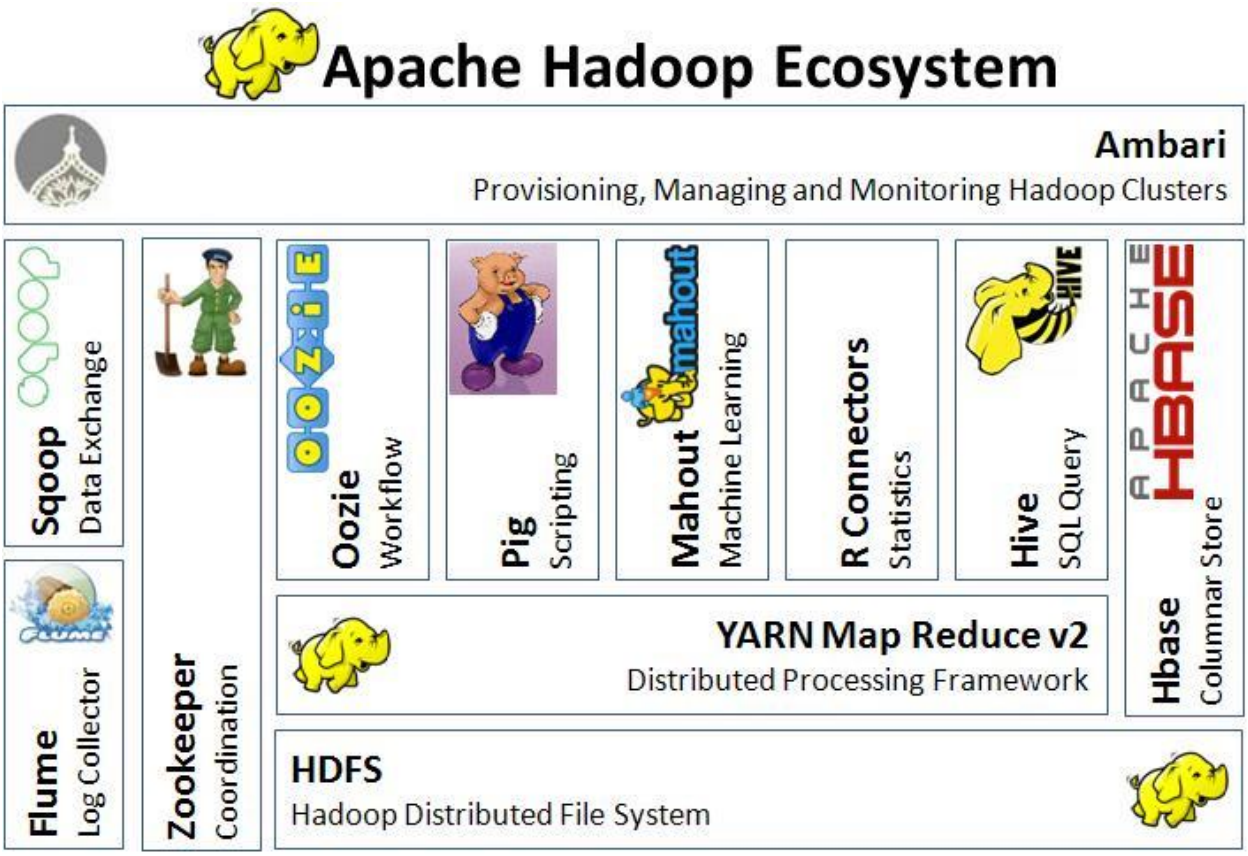
A *Hadoop* kezdetben a *MapReduce* keretrendszerből, valamint egy elosztott filerendszerből (*Hadoop Distibuted File System-ből*, röviden *HDFS*) állt. Mára ez egy egész ökoszisztémává nőtte ki magát, aminek számos eleme van. Ezek közül érdemes említeni a Hive-ot, ami segítségével a *Hadoop* klaszterben tárolt adatok SQL szerű lekérdezésekkel kezelhetőek, illetve SQL konnektorokon keresztül a klaszteren kívülről is könnyen elérhetővé teszi. További népszerű eszköz még a Pig nevű szkriptnyelv, ami többek között az adatfeldolgozást a hagyományos SQL nyelvnél egyszerűbben, könnyebben olvashatóan oldja meg. Egy másik széles körben alkalmazott szoftver az Oozie, amivel például Hive, Pig, *MapReduce Job*-okat szervezhetünk egységes munkamenetekbe.

A *MapReduce* keretrendszer lényegében a *MapReduce* paradigma implementációja, illetve a *Hadoop* motorja. Erre épül például a Hive és a Pig is, az ezekre készült szkriptek *MapReduce Job*-okra fordulnak le. A *MapReduce* 2.0-tól (más néven YARN) a korábbi verziók két nagy funkcionális egységét különválasztja, innentől kezdve az erőforrás kezelést és az ütemezést/monitorozást külön választja. Innentől kezdve egy alkalmazás nem feltétlen klasszikus értelemben vett MapReduce *Job*-okból áll, hanem ezek egy irányított körmentes gráfja is alkothatja.

A *MapReduce* kapcsán fontos még említeni a szintén Apache által fejlesztett Spark-ot, ami sokak szerint tekinthető az előbbi általánosításaként, ami idővel felváltja majd a *MapReduce*-t a *Hadoop* ökoszisztémában. A Spark lehetőség szerint próbál mindent a memóriában kezelni, és a fejlesztést is lényegesen egyszerűbbé kívánja tenni. Természetesen a Spark sem szakít a *MapReduce* paradigmával, az utóbbira készült algoritmusok könnyen megvalósíthatóak *Spark*-on is, vagyis a *MapReduce* keretrendszerre implementált programok átírhatóak. A legnépszerűbb *Hadoop* disztubúcióknak ma már a hagyományos *MapReduce* mellett a *Spark* is része.

Az *Hadoop* ökoszisztéma alappilérét, a *Hadoop Distributed File System-et* mindenképp hasznos külön is ismertetni. A *HDFS*-t alapvetően nem gyors hozzáférésre, hanem nagy áteresztőképességre optimalizálták. Az egyes file-ok blokkjai alapértelmezetten 64 MB-osak, hogy gyors szekvenciális olvasást biztosítsanak. A filerendszer minden egyes blokkot 3 (vagy opcionálisan több) példányban, külön gépeken tárol, ezáltal minimalizálva az adatvesztés valószínűségét.

A *Hadoop* architektúráját remekül lehet vizuálisan is szemléltetni, ami az alábbi képen látható:

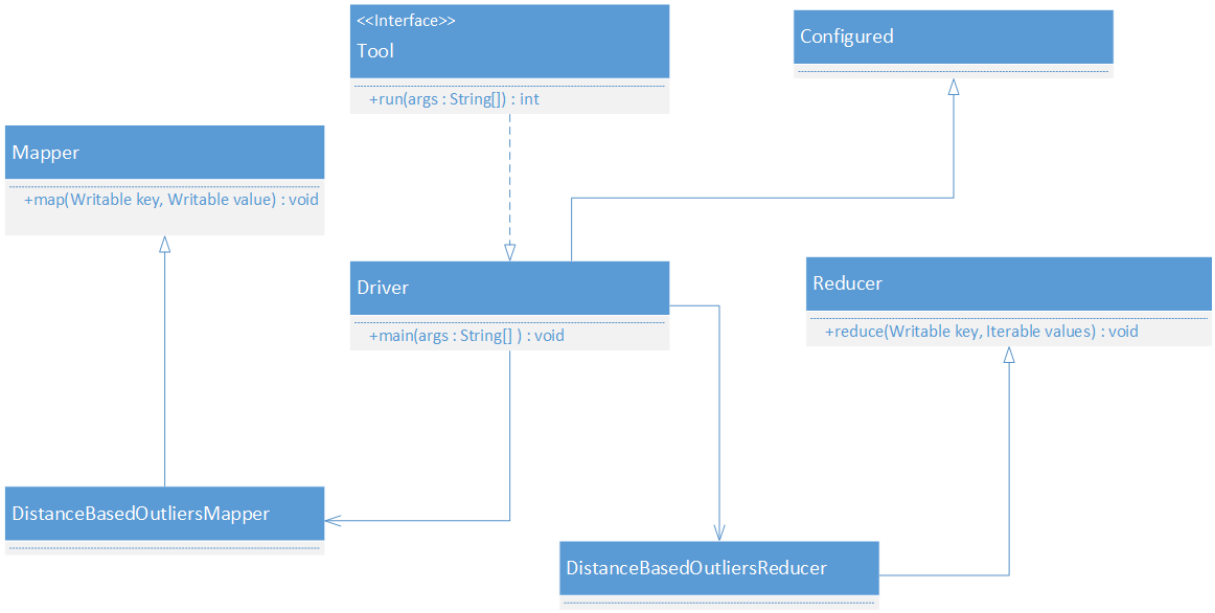



A piacon számos *Hadoop* disztribúció elérhető, amik az ismertetett ökoszisztéma számos elemét előre konfigurálva, könnyen telepíthető formában forgalmazza. Ezek közül a legelterjedtebb változatokat a *Cloudera* és a *Hortonworks* nevű cégektől származik.

## 7.2 Az implementáció architektúrája

A megvalósítás során alapvetően 3 féle osztályra van szükség a *KD fa*, a pontok és az ezekhez szükséges segéd osztályokon kívül: *Mapper*, *Reducer* és *Driver* osztályra. A *Mapper* osztályok Java nyelven a *Mapper* osztály leszármazottai, a különféle *Reducer* osztályok pedig a *Reducer* osztálynak. A *Driver* osztály akár egy mezei *Java* osztály is lehet main metódussal, azonban érdemes örökölni a *Configured* osztályból, valamint megvalósítani a *Tool interface*-t, a *Hadoop Command Line Option* parszolás lehetővé tétele miatt, és a *ToolRunner* osztályon keresztül futtatni az alkalmazást.

A *MapReduce Job*-ok Java nyelve implementációja az alábbi (egyszerűsített) osztálydiagramon látható felépítést követi:




*Ábra 7- MapReduce Alkalmazás felépítése*

## 7.3 Megvalósított adatstruktúrák ismertetése

### 7.3.1 KD fa megvalósítása az igényekre szabva

Mivel a *KD fa* nem minden definiált műveletére van szükség, ezeknek az implementációja elhagyható, viszont bizonyos lekérdezéseknek lehetőség nyílik optimalizált, azonban enyhén eltérő változatainak megalkotására is. Az elhagyható például a törlés művelet, illetve az egyesével való beillesztés is, amennyiben a fa konstrukciót nem számoljuk, mivel itt az összes pontot egyszerre illesztjük be.

Kibővített funkcionalitásra példa az ε körzet ellenőrzés, ami nem gyűjti össze az adott pontokat, csak megmondja, hogy kevesebb pont van-e az adott térrészben a meghatározottnál. Ez azt is lehetővé teszi, hogy egész részfákban való keresést kiváltsunk egy egész érték hozzáadásával, amennyiben az egész részfa a kérdéses térrészen belül helyezkedik el. Másfelől akkor se kell folytatnunk tovább a keresést, ha elértük a megadott limitet.

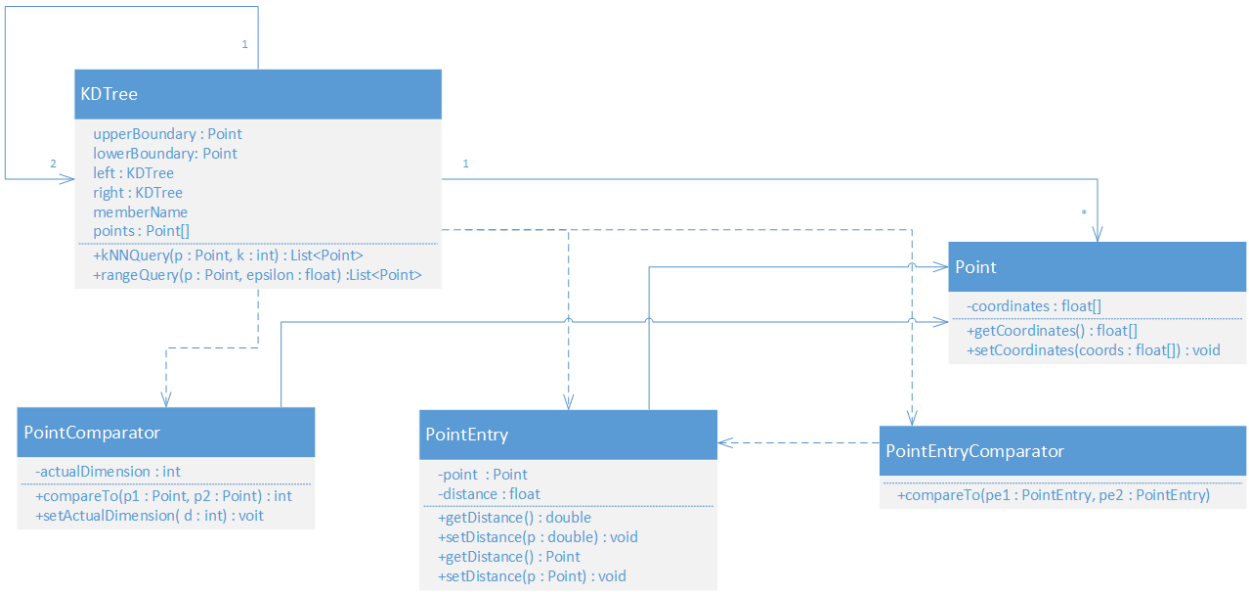
Kibővített funkcionalitásnak mondható még a k-távolság ellenőrzése, annyi kikötéssel, ami során ha találtunk k darab szomszédot, ami a limiten belül van, azonnal visszatérünk. Ez a lekérdezés típus különösen a távolság alapú kieső érték keresésnél jön jól. A fix térrészben való keresésnél azért tud ez lényegesen jobb lenni jelen esetben, mivel a nem kieső értékek az előbbi modellben lényegesen több szomszédot tartalmaznak az ε környezetükben, mint amennyit mi ténylegesen keresünk, tehát csak a k legközelebbi keresése már lényegesen kisebb térrészben való keresést jelent, mint az ε környezete a pontnak.

Az előbbiektől lényegesen eltérő jellegű a lebegőpontos számábrázolás jelentette problémák. A k legközelebbi szomszéd keresésénél pontok távolságát hasonlítjuk össze, érdemes megadni egy kerekítési

hibát, ami alatti eltérés esetén nem teszünk különbséget az egyes pontok között. Ha nem így teszünk, könnyen előfordulhat, hogy amúgy azonos távolságra lévő pontokat különbözően távolinak érzékelünk. Ez abban az esetben jelenthet problémát, ha ez pont a k-szomszédságot alkotó pontok távolság szerint növekvő sorrendben számított k-adik ponton túl jelentkezik, mivel ekkor nem fogunk minden szomszédot megtalálni. Ez a probléma különösen a *LOF algoritmus* során okozhat enyhén eltérő eredményeket az elvárttól.

Fontos még említeni a memóriahasználatra való optimalizációt, különösen az egyes pontok terén. Mivel a Reduce fázisban be kell férnie az egész rész adathalmaznak a memóriába, az egyes pontoknak a lehető legkisebb helyet kell foglalniuk. Ebből az okból a *float* típusú változók preferálandóak a *double* típusúakkal szemben, különösen a pont egyes koordinátáinak mentése során. Ráadásul a *double* változó a kerekítési hibákkal szemben épp úgy nem jelent védelmet, mint a *float* típusú, mivel ezt csak végtelen pontosságú számábrázolással lehet megoldani kerekítési hibák használata nélkül.

A *KD fa*, a pont osztály és a fontos segédosztályok az alábbi osztálydiagramon is láthatók:




*Ábra 8 - KD fa implementácó felépítése*

Az 5.-ös ábrán látható komparátor osztályok a Java nyelven az *Arrays* osztályban található rendezést, valamint a *PriorityQueue heap* alapú, prioritás szerinti rendezését teszik lehetővé. A *PointEntryComparator* osztály az összehasonlítástaz egyes pontok távolsága alapján végzi el. Az előbbi két említett rendezés komplexitása *O(n*log(n))*, tehát megfelelő kicsi lesz számunkra a lépésszámuk.

### 7.3.2 A rács megvalósítása

A rács megvalósítása során két fő gyakorlati szempont van:

- Viszonylag kis méret, maximum MB-os nagyságrendű
- Szerializálhatóság, csak a szükséges információk kiírásával

Mivel a rácsnak gond nélkül el kell férnie egy *Hadoop Job* konfigurációban, vagy esetleg az elosztott cache-ben optimálisan kByte-os nagyságrendű, de legrosszabb esetben sem lehet nagyobb 10 MB körüli értéknél. Ha az előbbi határt lényegesen túllépjük, akkor már az elosztott cache használata esetén is

érezhetően lassulhat a teljesítmény. A kis méret megvalósítása a bemeneti paramétereknek is függvénye, azonban ami mindenképp elkerülhető, az a mintavételezett pontok elmentése. Elég minden rács elemhez a határvonalait megadó egyeneseket eltárolni, ami két pont objektumban megoldható, amik közül az egyik a felső, a másik az alsó határoló koordinátát tartalmazza. Nem szükséges például az egyes levelek azonosítójának a mentése, ami a partíciók azonosítója lesz, ha a beolvasás és kiírás konzisztens egymással. Ezek betartásával 1024 partíció kialakítása esetén 146 kB a hierarchikus térbeli rács szerializált vátozata. Ez a méret még jelentős mértékben csökkenthető lenne szükség esetén a karakterláncos ábrázolások binárisra váltásával és az emberi szemmel olvashatóságot megkönnyítő ábrázolások jelölésével a rács elemeinek a típusánál, azonban erre nincs szükség.

Szerializáció során rekurzív szerializációt kell megvalósítani a fastruktúra kiírásához. A szerializált rácsot jól olvashatóvá teszi, ha emberi szemmel is olvasható karakter láncként írjuk ki. Ebben további könnyebbséget jelent a fa elem típusainak megnevezése, mint például gyökér, belső csúcs vagy levél. A jól olvashatóság a fejlesztés során a debuggolást is megkönnyítette, bár később elhagyható lenne. Ennek megfelelően természetesen a deszerializációt karakter láncból meg kell tudni oldani.

# 8. A kapott partíciók minőségének a vizsgálata

A mintavételezés kritikus lépesnek számít, ezért az így elért eredmények megfelelőségének vizsgálata kie melten fontos. Az értékeléshez igyekeztem egy viszonylag szélsőséges esetet kiválasztani először, ahol na gy számú partíció ehhez mérten viszonylag kisebb számú minta alapján lesz kialakítva. 240 millió adatpon t esetén 128 ezer mintavételezett pont alapján 1000 partíció jön létre, azaz egy partíciót átlagosan 128 ra ndom kiválasztott pont fog meghatározni, azaz 240 ezer pont közül átlagosan 128 kerül kiválasztása. Ebb en az esetben a partíciók átlagos méretének és szórásának a hányadosa 3 tizedes jegyre kerekítve 4.73 v olt, tehát a szórás még így is jelentősen kisebb volt az átlagos méretnél. A legnagyobb eltérés az átlaghoz képest 3.18 szórásra volt, százalékos értékben kifejezve 67%-al haladta meg az átlagos partíció méretét. Egy másik esetben 256 ezer pontot tartalmazó mintát hoztam létre véletlenszerűen 480 millió közül, ebb en az esetben 128 partíciót kialakítva. Ekkor az átlagos partícióméret a szórás 10.8 szorosa volt, a legnagy obb eltérés az átlagtól szórásban kifejezve 2.17, ugyanez százalékban kifejezve 20.1%. A minta méreténe k a rendelkezésre álló memória, Java specifikusan a heap space korlátozza. Ugyan a pont objektumok ált al igényelt memória mérete sok tényezőtől függ, azonban a tapasztalataim alapján milliós nagyságrendű minták kialakítása nem jelent gondot GB-os méretű memória rendelkezésre állása esetén. Jó összehason lítási alap ehhez, a távolság alapú kieső érték keresésénél a kieső érték keresés (utolsó *Job*) reduce fázisa , ahol az egész partíciónak kényelmesen el kell férnie a memóriában. A szóban forgó fázis OpenJDK 1.6 es etén (HDInsight specifikus) 2.5 GB méretű heap space esetén 10 millió 3 dimenziós ponttal gond nélkül m egbirkózott, tehát nagy valószínűséggel lehet állítani, hogy a mintavételezéssel nem lesznek problémáink , szinte biztosan kellő minőségű partícionálást tudunk kialakítani nagyságrendileg nagyobb méretű adath almazok esetén is.

# 9. Outlier keresés valós adatokon – felhasználási területek

## 9.1 Kieső érték keresés ipari felhasználása

Az anomáliák, avagy kieső értékek észlelése több területen is kiemelt fontosságú lehet. Ezek közül az fontosak például a csalásfelderítés,valamint a webes támadások észlelése.

### 9.1.1 Csalásfelderítés

A csalásnak sok jelentését meg lehet határozni, szoftveres területen egy informatikai (szoftver) rendszer kiskapuinak kihasználását jelenti, kártékony feladat céljából. A csalás felderítés több milliárd dolláros iparág, és évről évre nő. 2009-ben egy felmérés szerint a világ cégeinek 30%-a jelentette, hogy valamilyen csalás áldozatává vált. Csalás lehet például banki tranzakciók terén, vagy szolgáltatások ingyenes használata, vagy más egyéb kiskapuk használata. A csalások során egy vagy több szereplő titokban megkárosít valakit, illetve felhasznál valamit saját céljaira a szolgáltató szándékaitól szándékosan eltérő módon. A technika fejlődésével újabb és újabb csalási formák ütötték fel a fejüket, amit a szoftverek gyors fejlődése is elősegít. Az interneten keresztüli csalások száma nagyságrendileg több a hagyományos csalásokhoz képest. Ahogy egyre több problémát okoznak a rosszindulatú szereplők, egyre nagyobb hangsúlyt kapnak a biztonsági kérdések. Csalás detektálás sokféle módon történhet, például statisztikai, gépi tanulási vagy adatbányászati eszközökkel (az előbbiek között lehetnek átfedések). Adatbányászati technikák közül az egyik lehetséges opció a kieső értékek keresése, ahol a 'csalás' a normálistól bizonyos jellemzőiben kiugró módon eltér.

### 9.1.2 Webes támadások

A kieső érték keresés a webes támadások detektálása esetén a gyakorlatban is elterjedt módszer, többek között a klaszterezésen és döntési fák használatán alapuló módszerek mellett. A [6] és [7] alapján a *LOF* algoritmus megfelelő használata esetén megfelelő minőségű eredmények elérése az elvárás.

## 9.2 Kieső érték keresés online támadások észlelésére

### 9.2.1 Az adathalmaz

Az 1999-ben megrendezett KDD Cup versenyen a feladat online támadások azonosítása volt, ahol normális hálózati forgalmat és különböző támadási típusokat kellett megkülönböztetni. Az én munkámban csak normális forgalmat és támadást fogok megkülönböztetni. Az alapkoncepció, hogy a támadások egy jelentős része bizonyos jellemzői terén annyira különbözhet az átlagos forgalomtól, hogy kieső értéknek számít.

### 9.2.2 Az adatok jellemzése

Az 1999-es KDD Cup adathalmaz megfelelő előkészítése kiemelkedő fontosságú. Egyrészt az adathalmaz főleg támadásokból áll, és csak kisebb részben normális hálózati forgalomból. Az adatok nagyjából 20% átlagos hálózati forgalom. A támadások túlnyomó többségét két típus teszi ki, név szerint egy 'smurf' és egy 'neptun' nevű támadás, illetve nem elenyészően kis számú található 'satan' támadásból is.

Fontos megjegyezni, hogy az adatok szimulált adatok, mesterségesen létrehozott támadásokból és normális forgalomból áll. A való életben a rossz indulatú és átlagos csomagok közötti arány egyértelműen lényegesen eltérő, illetve az időbeli eloszlása a vizsgált adatoknak nem ismert. Ráadásul 1999-óta a támadások lényegesen szofisztikáltabbak és más lenyomatúak lehetnek. Azonban valódi hálózati adatok osztályozása rendkívül nehéz és időigényes feladat, ezáltal nagyon költséges is, ilyen adatok tudtommal nem elérhetőek nyilvánosan. Viszont az adathalamaz kezelhetően nagy méretű, és valóban támadások az ott rosszindulatúnak minősített adatok. Összességében a KDD Cup-ról származó adatok szerintem elegendően jó minőségűek, illetve nagyon kevés más alternatíva elérhető (az is hasonló módon eszközökkel szimulált). A [8] is kiáll a KDD Cup-nál régebbi, de hasonló eszközökkel összeállított DARPA adathalmaz mellett, feltehetően a KDD Cup-os adathalmazon elért eredmények is megfelelő iránymutatást adnak.

### 9.2.3 Az adatok előkészítése

Ha a rosszindulatú támadásokat outlierként akarjuk kezelni, akkor nem dolgozhatunk olyan adathalmazzal, aminek a döntő többségét ezek teszik ki, illetve a két leggyakoribb típusú támadás szinte minden paraméterében a saját típusától nagyon kicsiben eltérő,vagy gyakran nagyon nagy számú teljesen egyező adatot tartalmaz. A probléma orvoslására létre lehet hozni megfelelő mintavételezéssel egy olyan kisebb méretű adathalmazt, ahol már ritkának számítanak a támadások, és a normális adatok dominálnak. Az így generált rész adathalmaz tekinthető egyfajta tanító adathalmaznak, mivel ezt fogjuk használni az összes pont kiesőségének meghatározására.

További nehézséget jelent, hogy a normális hálózati forgalom is sok tekintetben számos majdnem teljesen egyező adatot tartalmaz, tehát a *KD fa* használata is csak relatíve kisebb teljesítmény növekedést okoz, ha nem szentelünk külön figyelmet az azonos értékek hatékony kezelésére.

Illetve érdemes még az adatokat valamilyen módon normalizálni, mivel a különböző numerikus jellemzők jelentősen eltérő értéktartománya nagyon erősen befolyásolja az euklideszi távolságok nagyságát az egyes adatok között. Én az alábbi képlet alapján normalizáltam minden egyes felhasznált oszlop adatait, ahol x egy cella értéke, X pedig az egész oszlopot jelenti:

$$Norm(x) = \frac{\big(x - mean(X)\big)}{std(X)}$$

A mean(X) az X oszlop aritmetikus közepe, az Std(X) pedig a szórása.

Valamint érdemes olyan dimenziókat kiválasztani a numerikusan értelmezhető dimenziók közül, amikben a kieső értékek a lehető legjobban eltérnek a normális hálózati forgalomtól. Azokat a dimenziókat, amik nem szolgálnak kellő információval, érdemes eldobni, mivel csak csökkentik a pontok közötti távolságok különbségét, azaz kevésbé fogjuk megtalálni a kieső értékeket.

### 9.2.4 A tanító adathalmaz jellemzése

A kieső értékek klasszifikálására használt tanító adathalmaz 48836 sort tartalmazott, vagyis a teljes adathalmaz nagyjából 1%-át. Ennek összetételét nagy precizitással, a teljes adathalmaztól eltérő jellegűre állítottam össze. A legjellemzőbb „smurf”, „neptune” és „satan” nevű támadásokból rendre 30 került bele a tanító adatok közé. Ez már csak azért sem problémás, mert a teljes adathalmaz több mint 80%-át kitevő „smurf” és „neptune” támadások elhanyagolható méretű eltérést mutatnak az osztályuknak megfelelő támadásoktól majdnem minden oszlopban (dimenzióban).

Az összesen több mint 40 oszlop közül 5-öt használtam, amiket az előző „Az adatok előkészítése” részben ismertetett képlet szerint normalizáltam.

Az oszlopok név szerint, és zárójelben a sorszámuk:

- Duration (1.)
- Src_bytes (5.)
- Count (23.)
- Srv_count (24.)

- Dst_host_same_src_port_rate (36.)

Az oszlopokat az alapján választottam ki, hogy melyekben lényegesen eltérő a támadások átlaga a szórásukkal mérve, ha az adatokat a támadási típusok szerint csoportokba rendezem, azaz „group by" műveletet hajtok végre rajtuk átlag és szórás aggregációs műveletekkel. További fontos szempont, hogy azok az oszlopok élveznek abszolút elsőbbséget ahol a legnépszerűbb támadási formák közül is lehetőleg minél több jelentősen eltér a normálisként megcímkézett érékektől. Más igények esetén, lehetne másként rangsorolni a különböző fajta támadási formák detektálásának fontosságát,mint egyszerűen az egyes típusok gyakoriságát szem előtt tartva. Természetesen ha csak minél akkurátusabban akarjuk észlelni az outliereket, nincs szükség ennél komplexebb modellre.

### 9.2.5 DB(ε,π) kieső értékek keresése

A távolság alapú kieső érték keresés a vártnál sokkal jobb eredményeket hozott az 1999-es KDD Cup adathalmazon. Jelen esetben a π értéke nem ez adatok hányadát, hanem azok kardinalitását fogja jelenteni, tehát csak egész értékű lehet.

Az ε paraméter értéknek 0.3-at választottam, és ezen a térrészen belül 50 darab szomszédnak kellett elhelyezkednie a vizsgált pont körül, hogy az ne számítson kieső értéknek.

Az alábbi két táblázatban a „pozitív" a kieső értéket jelenti, a „negatív" pedig a normál értéket. Az „valódi" jelző a jól meghatározottat, a „hamis" pedig a tévesen klasszifikált megfelelője.

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 3818530 | 953784 |
| **hamis** | 18997 | 107120 |

Az alábbi táblázatban a kerekített százalékos értékek, ahol minden oszlop összege 100-at ad ki:

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 99.5% | 90% |
| **hamis** | 0.5% | 10% |

Az eredményeken torzít, hogy a teszt adathalmaz (a teljes adathalmaz) több mint 80%-át támadások alkotják, mivel a valóságban feltehetően rendkívül ritka esetnek számíthat az előbbi arány.

Három további érdekes jellemző az angol terminológia szerinti accuracy, precision és recall:

- *Accuracy = (valódi pozitív + valódi negatív) / összes elem ≈ 0.97*
- *Precision = valódi pozitív / (valódi pozitív + hamis pozitív) ≈ 0.95*
- *Recall = valódi pozitív / (valódi pozitív + hamis negatív) ≈ 0.97*

### 9.2.6 LOF algoritmus alkalmazása

A *Local Outlier Factor* nevű algoritmussal való kieső érték keresés során több, eltérő paraméterezéssel eltérő eredményeket kaptam. Tisztán látszik, hogy a vizsgált szomszédok kiválasztásának száma nagy mértékben befolyásolja az eredményeket. A kipróbált 150-es, 200-as és 300-as *k szomszédság* méretek közül a nagyobbak enyhén csökkenő *LOF pontszám* határokkal jobbnak bizonyultak. Fontos megjegyezni, hogy ez csak egy bizonyos k értékig igaz, mert ha az egész adathalmaz jelentős hányada belekerül a *k szomszédságba*, a *LOF pontszámok* már kiegyenlítődnek a kieső és normál értékek között, mivel a *lokális elérési sűrűség*ek az egyes pontokhoz hasonlóak lesznek, ami szintén a *k távolságok* növekedésének és kiegyenlítődésének a következménye.

Mindenképpen fontos említeni, hogy az adathalmaz sok számos dimenzióban teljesen megegyező rekordot tartalmaz, ezért a *LOF algoritmus* működését figyelembe véve az egymástól 0 távolságra elhelyezkedő pontokat nem vettem bele egymás *k szomszédságába*. Ennek a lépésnek további érdekes aspektusai is vannak, amiket majd a Továbbfejlesztési lehetőségek című fejezetben részletesebben is kifejtek majd. Ami miatt az említett lépés nagy jelentőségű, hogy extrém esetben egy 0 értékű k-távolság végtelen *lokális elérési sűrűség*et jelent, ami a *LOF algoritmust* teljesen használhatatlanná teszi.

Az alábbi táblázatokban a „pozitív” a kieső értéket jelenti, a „negatív” pedig a normál (nem kieső) értéket. Az „valódi” jelző a jól meghatározottat, a „hamis” pedig a tévesen klasszifikált megfelelője.

#### *9.2.6.1 Eredmények 150 minimum elemszámú k szomszédság és minimum 3.0 LOF pontszám esetén:*

A legkisebb *k szomszédságú LOF* eredmények is relatíve jó eredményt nyújtanak, azonban ebben az esetben viszonylag több kieső értéket nem találunk meg. Azonban fontos megjegyezni, hogy a második táblázat adatai enyhén megtévesztő képet nyújtanak a kieső értékek észlelésének hatékonyságáról, mivel az adathalmaz 80% körüli arányban kieső értékekből áll, ezért a nem észlelt kieső értékek aránya relatíve nagyobb lesz a jól észlelt normál értékekhez képest. Amit le lehet olvasni a második táblázatról, hogy ha a klasszifikált értékek nagy része outlier, akkor ennek aránya a normális értékekhez képest csak nagyon akkurátus detektálás esetén lesz elhanyagolható a (jól detektált) normál értékekhez képest.

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 3108223 | 942517 |
| **hamis** | 30264 | 817427 |

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 99% | 53.6% |
| **hamis** | 1% | 46.4% |

Három további érdekes jellemző az angol terminológia szerinti accuracy, precision és recall:

- *Accuracy = (valódi pozitív + valódi negatív) / összes elem ≈ 82.7%*
- *Precision = valódi pozitív / (valódi pozitív + hamis pozitív) ≈ 99%*
- *Recall = valódi pozitív / (valódi pozitív + hamis negatív) ≈ 79%*

#### *9.2.6.2 Eredmények 200 minimum elemszámú k szomszédság és minimum 2.3 LOF pontszám esetén:*

A 150-es mérethez képest nagyobb *k szomszédság* alacsonyabb *LOF pontszámmal* is sokkal akkurátusabb kieső érték keresést tesz lehetővé, amit az alábbi táblázat és a három fontos mérőszám is szemléltet. Ugyan jelen paraméterek esetén több hamis pozitív érték szerepel, azaz kiesőnek detektált normális értékek, lényegesen kevesebb normálisként detektált outlier van. Ennek oka a szomszédság méretéhez képest nagyobb mértékben csökkentett *LOF pontszám*, ami a kieséként észleléshez szükséges.

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 3780112 | 916519 |
| **hamis** | 56262 | 145538 |

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 98.5% | 86.3% |
| **hamis** | 1.5% | 13.7% |

Három további fontos jellemző az angol terminológia szerinti accuracy, precision és recall:

- *Accuracy = (valódi pozitív + valódi negatív) / összes elem ≈ 95.8%*
- *Precision = valódi pozitív / (valódi pozitív + hamis pozitív) ≈ 98.5%*
- *Recall = valódi pozitív / (valódi pozitív + hamis negatív) ≈ 96.3%*

#### *9.2.6.3 Eredmények 300 minimum elemszámú k szomszédság és minimum 2.0 LOF pontszám esetén:*

Az 1999-es KDD Cup során végzett kieső érték keresési adatbányászati munkám során a legjobb eredményeket a 300 minimum nagyságú *k szomszédsággal* paraméterezett *Local Outlier Factor* algoritmussal értem el, ahol a 2.0 *LOF pontszámtól* számít egy adat kieső értéknek.

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 3908313 | 890647 |
| **hamis** | 82134 | 17337 |

| | **pozitív** | **negatív** |
|---|---|---|
| **valódi** | 98% | 98% |
| **hamis** | 2% | 2% |

- *Accuracy = (valódi pozitív + valódi negatív) / összes elem ≈ 98%*
- *Precision = valódi pozitív / (valódi pozitív + hamis pozitív) ≈ 98%*
- *Recall = valódi pozitív / (valódi pozitív + hamis negatív) ≈ 99.5%*

Az eredmények messze a legakkurátusabbak mind a *LOF algoritmus*, mind a *DB(ε,π)* kieső érték modell általi más próbálkozások esetén, mivel a detektált és nem detektált támadások aránya rendkívül jó, és a precizitás és a kiesőnek tartott értékek is remek arányban kiesőnek számítanak. A precizitás („Precision paraméter") kis mértékben ismét csökkent ugyan, azonban lényegesen kisebb mértékben a többi mutató javulásához képest, amin ha valamiért feltétlen javítanunk kellene, elég lenne enyhén megemelni a minimum *LOF pontszámot*.

### 9.2.7 A KDD Cup adathalmazon kapott eredmények összegzése

A kapott eredmények messzemenően fölülmúlták a várakozásaimat, és fölmerül a jogos kérdés, hogy vajon valós adatokon mennyire lenne hatásos az előbbi módszer. Sajnos valós (nem szimulált) adatok hiányában erre a kérdésre nem tudok válaszolni. Azonban a várakozásaim alapján a szimulált adatok bizonyos dimenzióban könnyen jellemezhető karakterisztikái a valóságban jó eséllyel kevésbé jönnek elő, tehát mind a két algoritmus teljesítménye feltehetően lényegesen rosszabb lenne, de még ezzel együtt is jó eredménynek számíthatnak.

Fontos megjegyezni, hogy más adathalmaz eltérő jellemzői miatt elképzelhető, hogy a módszereken is változtatni kellene.

A két vizsgált outlier modell közül a *Local Outlier Factor* bizonyult hatékonyabbnak megfelelő paraméterezés esetén. Mivel a *LOF algoritmus* a jobb eredmények eléréséhez nem csak nagyobb k-szomszédság vizsgálatára volt szüksége, de általánosságban véve is sokkal számítás igényesebb mint a *DB(ε,π)*-outlier modell szerinti kieső érték keresés. A *LOF* egyik nagyon kellemes tulajdonsága, hogy nem azt mondja meg, hogy mely pontok kieső értékek, hanem minden egyes ponthoz kiszámol egy értéket, amely jellemzi mekkora mértékben kieső érték. A megfelelő minimum pontszám meghatározása ugyan további számítás intenzív finomhangolást igényelhet, azonban az eredmények minőségét tovább javítja. Összességében a *LOF algoritmus* a TDK kutatásom során az általam megismert szakirodalommal [6], [7] összhangban hatékony online behatolás észlelő algoritmusnak bizonyult.

Érdemes ehhez hozzátenni, hogy bizonyos esetekben a *DB(ε,π)* kieső érték keresés is megfelelően jó minőségűnek bizonyulhat. Ennek további megerősítésére még több adatelemzésre és valós (nem szimulált támadásokat tartalmazó) adatokra is lenne szükség. Annyiban érdemes még tovább árnyalni a képet, hogy a *LOF* versus *DB(ε,π)* összehasonlításában ebben az esetben egy idő/minőség tradeoff látszik körvonalazódni, ami értelmezhető kisebb költség és jobb minőség közötti választásként.

# 10. Az algoritmus futási idejének vizsgálata

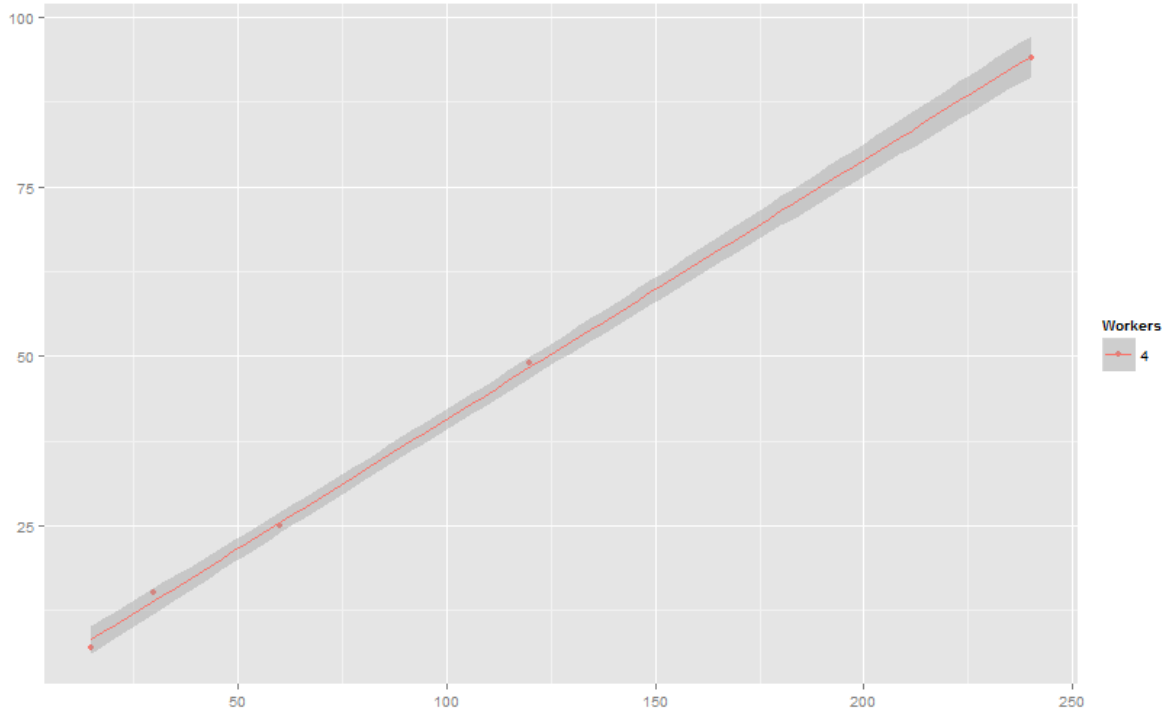
## 10.1 Futtatás Azure HDInsight Hadoop klaszteren

A futási idők mérése során az implementált algoritmusok skálázhatónak bizonyultak a vizsgált adatokon. Annyit érdemes ehhez hozzá tenni, hogy a *Hadoop* klaszter használatának overhead-je a kisebb adathalmazoknál jól kijött, például 15 és 30 millió adat pont esetében a futási idő jelentős részét a *MapReduce* keretrendszer és a HDFS használatának egyfajta minimum költsége teszi ki, tehát a 15 és 30 millió pont esetében kapott eredmények inkább érdekességnek számítanak, mint a skálázhatóság bizonyításának. A $\pi$ és a *k* paraméter mind a két algoritmus esetén 15 volt, és fontos hangsúlyozni, hogy itt a $\pi$ értéke is darabszámban van megadva. A skálázhatóság vizsgálatához generáltam az adatokat, így könnyedén létrehozhattam megfelelő méretű és eloszlású adatokat. A generált adatok térbeli eloszlása optimális a KD fával történő keresés hatékonysága szempontjából, ami a szomszéd keresés *O(k*log(n))* lépésszámát eredményezi az egyes pontokra.

### 10.1.1 Távolság alapú *DB(ε,π)* kieső érték keresés futtatása:

#### *4 Worker Node*

A 4 gépen 4 Reduce taszkkal való futási idő vizsgálatok mutatják talán a legszebben az adatmérettel való skálázódást:




*Ábra 9 – Távolság alapú kieső érték keresés futási idők 4 Worker esetén*

A pontos adatok az alábbi táblázatban tekinthetők meg.

| ***Pontok száma (millió)*** | ***Futási idő (perc)*** | ***Reduce taszkok száma*** | ***Partíciók száma*** |
|---|---|---|---|
| 15 | 7 | 4 | 4 |
| 30 | 15 | 4 | 8 |
| 60 | 25 | 4 | 16 |
| 120 | 49 | 4 | 32 |
| 240 | 94 | 4 | 64 |

*8 Worker Node*

| ***Pontok száma (millió)*** | ***Futási idő (perc)*** | ***Reduce taszkok száma*** | ***Partíciók száma*** |
|---|---|---|---|
| 30 | 12 | 8 | 1024 |
| 60 | 17 | 8 | 1024 |
| 120 | 27 | 8 | 1024 |
| 240 | 50 | 8 | 1024 |
| 240 | 35 | 8 | 64 |
| 480 | 32 | 32 | 128 |

### 10.1.2 LOF algoritmus futtatása:

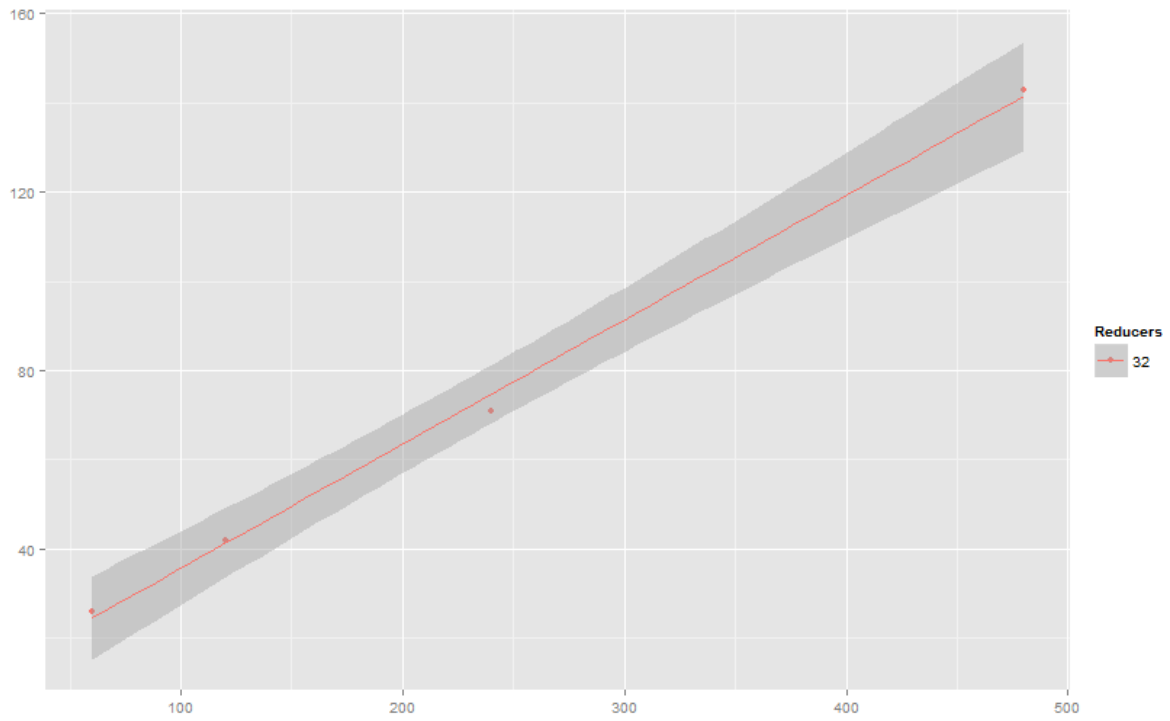
*4 Worker Node*

| ***Pontok száma (millió)*** | ***Futási idő (perc)*** | ***Reduce taszkok száma*** | ***Partíciók száma*** |
|---|---|---|---|
| 30 | 35 | 4 | 8 |
| 60 | 46 | 8 | 16 |
| 120 | 68 | 16 | 32 |

*8 Worker Node*

| ***Pontok száma (millió)*** | ***Futási idő (perc)*** | ***Reduce taszkok száma*** | ***Partíciók száma*** |
|---|---|---|---|
| 60 | 26 | 32 | 32 |
| 120 | 42 | 32 | 64 |
| 240 | 71 | 32 | 64 |
| 480 | 143 | 32 | 128 |

A 8 Worker node-on 32 Reduce Task-al elért eredmények az alábbi képen is megtalálhatók:




*Ábra 10 - LOF futási idő*

### 10.1.3 Fontos beállítások a megfelelő eredményekhez

#### *10.1.3.1 A Reduce taszkok száma*

A Reduce taszkok számát illetően érdemes a lehetőségekhez mérten a partíciók számával megegyező reduce taszkot indítani. Az egy worker Node által futtatható taszkok számát a rendelkezésre álló processzor magok és a memória mérete határozza meg. A Java Heap space mérete az egyes taszkok számára külön-külön rendelkezésre áll, a taszkok külön Java VM-ekben futnak. Az Apache ajánlása szerint a Reducer-ek számát sem érdemes indokolatlanul nagyra növelni, ha a partíciók túl gyorsan feldolgozhatóak, akkor az egyes taszkok elindítása jelentette overhead miatt is kaphatunk a vártnál rosszabb eredményeket futási idő tekintetében.

#### *10.1.3.2 Timeout esetleges beállítása*

A timeout megfelelő beállítása, alapvetően a kieső érték keresésé k szomszéd keresésese miatt. Alapértelmezetten, ha egy *Hadoop* (pl. *Azure HDInsight*) *Job* egy taszk-ja 10 perc alatt nem halad legalább 1%-ot illetve az összes érték feldolgozása után egy fázis befejezéséig, timeout lép fel. Ilyenkor az adott taszk újra lesz indítva, és megpróbálja ismét végrehajtani a timeoutolt taszk specifikus részt, alapbeállításként 3 alkalommal. Mivel a százalékban mért haladás a Reduce fázisban az értékeken való iteráció függvényében van mérve, és a kieső érték keresés során a *k szomszédság* meghatározásához a bemeneti értékeket egytől egyig el kell tárolja a memóriában, ezért a timeout megnövelése nélkül 10 perc áll rendelkezésre a *KD fa* konstrukcióra és a szomszéd keresésére együttvéve. Ez optimális adateloszlás esetén akár 10-20 millió pont méretű partíciók (ennél általában kisebb méretűvel dolgoztam) esetén a D12-es tíusú HDInsight Worker (Data) node-okkal ez nem jelent gondot. Azonban ha az adatok eloszlása nagyon nem egyenletes, akkor a *KD fa k szomszédság* lekérdezése az adott partíció minden elemére az $O(n^2)$ közeli futási idő miatt nagyságrendileg több időt igényel, mint az előző esetben. Természetesen nagyon nagy partíciók kialakítása miatt optimális esetben is meg kell növelni a timeout értékét.

#### *10.1.3.3 Partíciók számának jó kiválasztása*

Már az előbbi bekezdés tartalma is sugallhatja, hogy a túl kevés partíció nem csak a túl nagy memória igény, de a timeout érték átállítását is okozhatja. Azonban túl sok partíció különösen a *DB(ε,π)* implementáció esetén járhat jelentős overhead-el, mivel az ε környezettel való növekedés kis partíció méret esetén már nem elhanyagolható. A tapasztalatok alapján a valamivel több partíció a kelleténél nem okoz érezhető teljesítmény romlást, tehát jobb inkább több partíciót kialakítani, mint a szükségesnél kevesebbet. A túl nagy partíciók már a Java heap space megtelése előtt is lassuláshoz vezethetnek, mivel a memória nagy kihasználtsága is jelentős lassuláshoz vezethet.

#### *10.1.3.4 Megfelelő mintavételezés*

A mintavételezés megfelelő minősége szintén kiemelten fontos. Szerencsénkre az eredmények kellően robosztusnak bizonyultak, de ez sem zárja ki az alulmintavételezés lehetőségét. Érdemes legalább 2, de inkább 3 nagyságrenddel nagyobb mintaméret beállítása, mint a kialakítani kívánt partíciók száma, azaz 128 partíció esetén pl. 256 000 pont méretű minta. Természetesen nyugodtan lehet akár a kialakítani kívánt partíció mérettel akár megegyező méretű a minta, mivel ennek mindenképp el kell férnie a memóriában.

#### *10.1.3.5 Megfelelő Hadoop klaszter építése*

Az előbbiek alapján a megfelelő *Hadoop* klaszteren a Reducerek rendelkezésére álló memóriába az egyes partícióknak kényelmesen el kell férniük, azaz a Java heap sapce-nek kellően nagy méretűnek kell lennie. Mivel az outlier keresés, különösen a *LOF algoritmus* esetében, kellően számításigényes, előnyt jelent az erősebb *CPU* is. Harmadrészt, de nem utolsó sorban a használt *Java* verzió is sokat számít, a különböző *Java VM*-ek teljesítménye között jelentős különbségek lehetnek. Illetve ha virtualizált környezetben építjük a *Hadoop* klasztert, jól jön a virtualizációs overhead minimalizálása is. Az előbbi kijelentés persze általánosságban véve is igaz, azonban a használt algoritmusok nagy számításigénye felerősíti ezt a hatást.

## 10.2 Az algoritmusok bemeneti paramétereinek a hatása a teljesítményre

### 10.2.1 Local Outlier Factor

A keresett legközelebbi szomszédok száma egyenesen arányos az algoritmus futási idejével, ha minden más paramétert rögzítettnek tekintünk. Ez a *KD fa* várható teljesítményéből következik a k szomszédság lekérdezésekre (átlagosan *O(k*n*log(n))* ) lépésszámú megfelelő eloszlású adatokon ). Más befolyásoló

tényező még a partíciók száma. Extrém esetben ,nagy átlagos *k távolságok* és relatíve kisebb partíciók esetén, a partíciók számának nagy növekedése jelentősen megnövelheti azon pontok számát, amiknek a szomszédjait több partícióban is keresni kell. Azonban ez a hatás nagyságrendileg kisebb, minta a *DB(ε,π) outlier* modell szerinti kieső érték kereső algoritmus impelementációja esetén, tehát a *LOF* implementációja során sokkal jobban skálázódik kisebb partíciókkal is. 240 millió 3 dimenziós pont esetén 64 partícióval és *k = 15* értékkel körülbelül minden 20. pontnak kellett más partíciókban is keresni a szomszédjait. Mivel d dimenziós hiperkockákkal dolgozunk, úgy becsülhetjük a szomszéd keresések második fázisában a többlet pontok az előbbi arány *2*d*-szeresével (a kocka oldalszámával) egyezik meg várhatóan.

### 10.2.2 *DB(ε,π)* kieső értékek

*DB(ε,π)* kieső érték keresés esetén az ε környezet mérete és a partíciók száma egyaránt fontos tényező. Nagy ε értékek jelentette hatás ugyan a k-távolság számolásával enyhíthető, azonban a partíciók méretének növekedése a fa konstrukció idejét is növeli. Szélsőséges esetben a partíció *ε* környezete lényegesen nagyobb lehet a partíciónál, alapvetően a felhasználó választotta paramétereknek köszönhetően, ami a teljesítményt nagy mértékben csökkenti. A túl nagy *ε* értékek választása az algoritmus eredményeinek minőségét is negatív értékben befolyásolja, mivel ekkor az adathalmaz jelentős része eshet az egyes pontok ε környezetébe, mivel az ε sugarú hipergömb az adattér méretének nem elhanyagolható hányadát teszi ki. Az algoritmus optimális használata esetén a *π* értéke jelenti a legnagyobb hatást az algoritmus futási idejére a bemeneti paraméterek közül, mivel a k-távolság (lehetne akár *π* távolságnak is hívni jelen esetben) meghatározása ebben a kieső érték keresési modellben átlagosan sokkal rövidebb ideig tart mint a teljes ε környezetben található pontok számának meghatározása, mert az utóbbi száma általában sokkal nagyobb a π paraméterhez kapcsolódó k számnál. A k távolság meghatározásához szükséges lépésszám a k szomszédok megkereséséhez hasonlóan a k értékével egyenesen arányos.

# 11. Összefoglalás és továbbfejlesztési lehetőségek

A dolgozatom keretén belül implementált *LOF* és *DB(ε,π)* modell alapú algoritmusok a generált adatokon jól párhuzamosíthatónak és skálázhatónak bizonyult, és a vizsgált 1999 KDD Cup adathalmazon hatékonyan meg tudták különböztetni a kieső értékeket (támadásokat) az adathalmaz többi részétől.

A teljesítmény tovább javítható több módszerrel is. Először is maga az adathalmaz preparációjával, ha az sok azonos értéket tartalmaz. Ezeket az értékeket *Hadoop*-on hatékonyan csoportosítani lehetne, és nem egyesével kezelni őket. Ez a *KD fa* hatékonyságát nagy mértékben növelné, mivel a teljesítménye a szomszéd lekérdezések terén a szekvenciális O(n) legrosszabb esetű futási időtől (de még mindig érezhetően gyorsabbtól) a sokkal kellemesebb *O(k*log(n))*közelében lenne, amennyiben további kedvezőtlen speciális esetek nem lépnek közbe.

Az adatok pontonkénti aggregálása a *Local Outlier Factor* algoritmusnál egy különösen izgalmas téma, mivel ez az algoritmus viselkedését jelentősen befolyásolja. A *LOF* esetében a pont *lokális elérési sűrűség*ének számolásánál nem vesszük bele a pontot a saját szomszédságába. Azonban ha sok más adat belekerül, ami azonos koordinátákkal rendelkezik, akkor a sűrűség végtelen lehet, amennyiben az egész szomszédságot ezek a pontok alkotják. Ha pedig a *lokális elérési sűrűség* végtelen, a pont saját *LOF* pontszámának számítását értelmetlenné teszi, illetve minden más olyan pontét is, aminek ez az adott pont

a *k szomszédságában* található. Ez nem feltétlen számít speciális esetnek, amint láthattuk az 1999-es KDD Cup-os adatoknál is ez volt a helyzet. Meg lehet úgy is közelíteni, hogy minden azonos helyen elhelyezkedő pont megegyező, azonban ez nagyban torzíthatja az eredményeket. Vonzóbb lehetőségnek tűnik a pont csoportokhoz egy súly érték definiálása, ami az azonos koordinátákon elhelyezkedő pontok száma lehetne. Ez a *LOF algoritmusnak* minimális kiegészítését, módosítását jelentené, ami bizonyos esetekben jobb eredményeket jelenthet, az adatok eloszlásától függően.

További érdekes továbbfejlesztési lehetőség a párhuzamosítás további növelése, azaz a szomszéd keresés elvégzése egyszerre tetszőleges számú pontra egy partíción belül, mivel a k-szomszédok meghatározása során csak olvasásra van szükségünk, a KD fát nem módosítjuk. Ezt a párhuzamosítást több módon is meg lehet valósítani, ezek közül felsorolnék néhány lehetőséget:

- A legkönnyebben kivitelezhető, ha több partíció létrehozása helyett egyszerűen több gépen is azonos rész adathalmazzal dolgozunk, csak éppen elosztjuk köztük, hogy hol mely pontoknak a szomszédjait keressük meg. Ez abból a szempontból is jó, hogy kevesebb partíciók közötti keresésre van szükségünk a *Local Outlier Factor* esetében, mivel nagyobb partíciókkal dolgozunk. Ez természetesen a *KD fa* felépítésének az idejét megnöveli, ami a jelenlegi $O(n*\log^2(n))$ komplexitás miatt egy idő után már nem kifizetődő, a szomszédkeresés kisebb $O(k*n*\log(n))$ komplexitása miatt. Ez természetesen más fa konstrukciós módszerek bevezetésével orvosolható. A távolság alapú kieső érték keresésesnél még jobb a helyzet, mivel ott az ε környezettel való növelés miatti adat duplikáció csökkentése további teljesítmény növekedéshez vezet.
- Másik lehetőség lenne *Hadoop* vagy *Spark* plusz *GPU* használata a szomszédkereséshez, így akár ezres nagyságrendű pont szomszédjait kereshetnénk egyszerre, ami nagyságrendileg gyorsabb lenne a szomszédkeresés fázisában. Azonban a fa konstrukció párhuzamosítása *GPU*-n már kevésbé magától értetődően egyszerűen párhuzamosítható hasonlóan hatékonyan. *GPU* használatra akár *Hadoop* klaszterrel együtt is van lehetőség, ez egy gyakorlatban is alkalmazott módszer. Mivel a pontok partícionálása az eredmények alapján elég hatékonyan működött, ezt egy alapvetően jó opciónak tartom.

Erre több lehetőség lenne jelenleg akár *Java* nyelven is, például a *JCUDA* vagy az *AMD Aparapi* segítségével, amik segítségével Java nyelven lehet *GPU*-t programozni, ezáltal *Hadoop* vagy akár *Spark* klaszterben is könnyen használhatóvá teszi őket.

- Természetesen el lehet menni szín tiszta *GPU* klaszteres megoldás irányába is. Elképzelhető, hogy így lehetne a legjobb teljesítményt elérni, azonban ez jelentené a legtöbb újratervezést és implementációt

Az adatstruktúrák terén a már említett *KD fa* konstrukció teljesítményének a javításán, illetve az adatok térbeli partíciokba rendezéséhez használt heurisztikák használatával lehetne még jobb eredményeket elérni. Erre lehetőség például a valamilyen szempontból legjobb dimenzió kiválasztása az adattér kettévágásához. A konstrukció folyamán egyik opció a mediánok-mediánja algoritmus használata, ami segítségével *O(n)* komplexitással tudnánk a *pivot* elemet meghatározni *O(n*log(n))* helyett, így a fa konstrukció csak *O(n*log(n))* lenne. A mediánok mediánja csak közelíti a valós mediánt, azonban gyakran kellően kiegyensúlyozott fákhoz vezet. Másik lehetőség csupán egyszer rendezni minden egyes dimenzióban a pontokat, és ennek az eredményét eltárolni. Az előbbi módszer *O(d*n*log(n))* konstrukciós időt eredményez, viszont extra tárhely igénnyel jár.

# 12. Hivatkozások


[1] Markus M. Breunig†, Hans-Peter Kriegel†, Raymond T. Ng‡, Jörg Sander†. LOF: Identifying Density-Based Local Outliers. In: SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data, Pages 93-104, 2000.

[2] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, Jörg Sander. OPTICS-OF: Identifying Local Outliers. In: PROC. OF PKDD '99, PRAGUE, CZECH REPUBLIC, LECTURE NOTES IN COMPUTER SCIENCE (LNAI 1704), 1999.

[3] Edwin M. Knox and Raymond T. Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets. In: VLDB '98 Proceedings of the 24rd International Conference on Very Large Data Bases, Pages 392-403, 1998.

[4] Mohamed Aly, Mario Munich, Pietro Perona. Distributed Kd-Trees for Retrieval from Very Large Image Collections. In: Proceedings of the British Machine Vision Conference (BMVC) ,2011

[5] Yaobin HE, Haoyu TAN, Wuman LUO, Shengzhong FENG, Jianping FAN. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. In: Frontiers of Computer Science, February 2014, Volume 8, Issue 1, pp 83-99, 2014.

[6] Dokas, Levent Ertoz, Vipin Kumar, Aleksandar Lazarevic, Jaideep Srivastava, Pang-Nig Tan. Data Mining for Network Intrusion Detection Paul. In: *Proc. NSF Workshop on Next Generation Data Mining*. 2002. p. 21-30, 2002.

[7] Aleksandar Lazarevic, Levent Ertoz, Vipin Kumar, Aysel Ozgur, Jaideep Srivastava. A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In: *SDM*. 2003. p. 25-36, 2003.

[8] Ciza Thomas Vishwas Sharma N. Balakrishnan. Usefulness of DARPA Dataset for Intrusion Detection System Evaluation. In: *SPIE Defense and Security Symposium*. International Society for Optics and Photonics, 2008. p. 69730G-69730G-8, 2008.

[9] - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008, 51.1: 107-113, 2008.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In: *ACM SIGOPS operating systems review*. ACM, 2003. p. 29-43, 2003.

[11] Han, Jiawei, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques: concepts and techniques*. Elsevier, 2011.