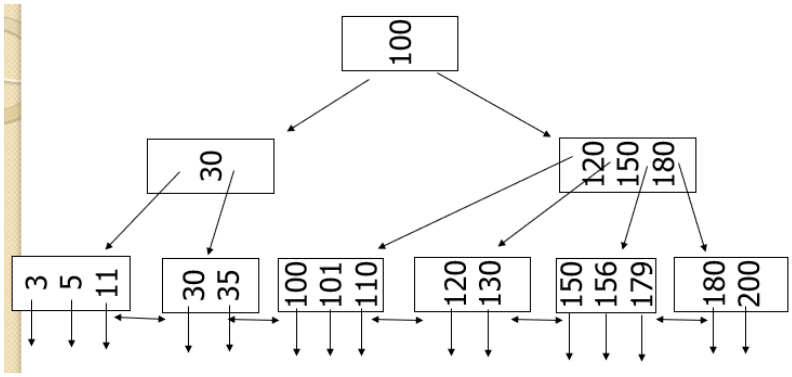
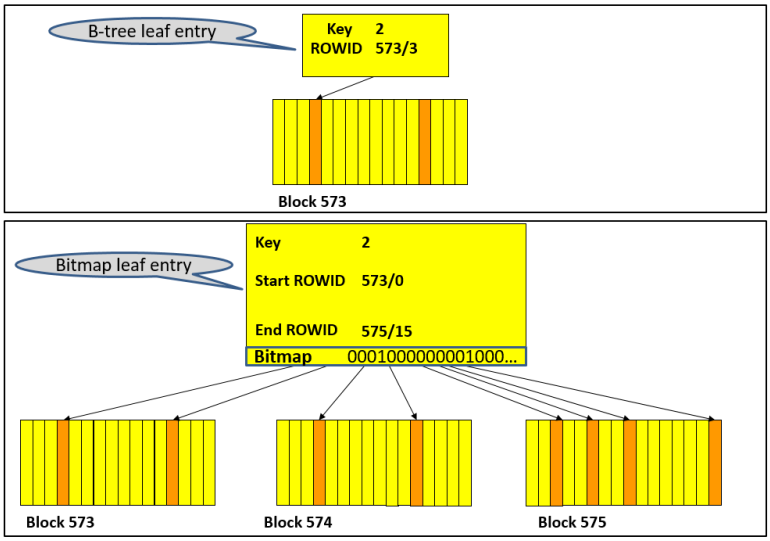
1. **B-fa, bitmap indexek szerkezetének és használatának ismerete**

B-fa index: a hagyományos index B-fa szerkezetű. Két fajta csúcsa van: levélcsúcsok és belső csúcsok (gyökér elem is). Belső csúcsok a navigációt segítik (hogy eljussak az adott kulcsértékhez). A levél csúcs tartalmazza a kulcselemeket (tábla oszlopának/oszlopainak tartalma), illetve a kulcselemeket tartalmazó rekordok ROWID-jait. Tehát a levél csúcs mutatja meg a keresett sor helyét a táblában. A levélben a kulcselemek növekvően vagy csökkenően (sorrendben, rendezetten) helyezkednek el.



Bitmap index: itt is kulcsok vannak a levelekben, de itt nem ROWID-k, hanem Bitmap-ek kapcsolódnak hozzá. Minden sornak egy bit felel meg, ami azokra a sorokra lesz 1-es, amelyek az adott értéket tartalmazzák.



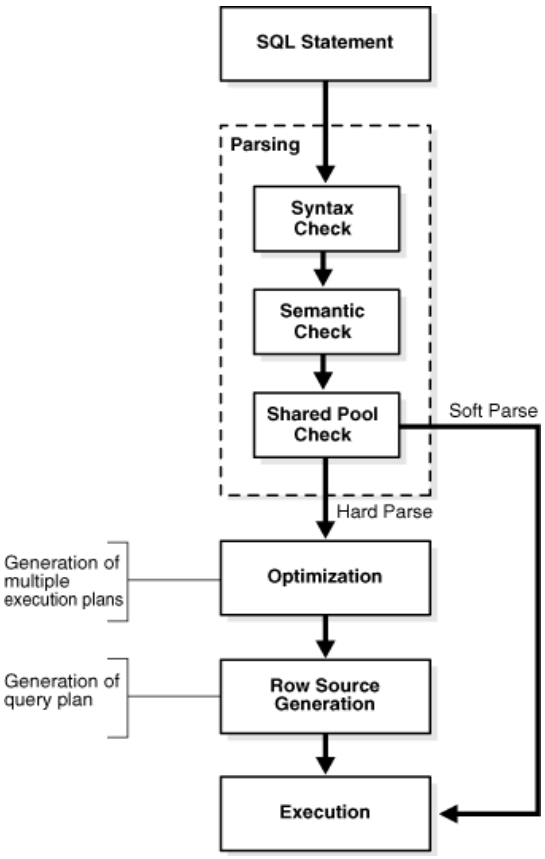
Minden sorhoz van egy bejegyzés, ahol 0-ás van, ott nem 2-es számú érték szerepel abban a sorban, ahol 1-es van ott 2-es számú érték szerepel a sorban. A bitmap indexben minden egyes különböző kulcs értékhez csak egyetlen egy bejegyzés van, míg egy B-fa indexben ha egy 2-es többször szerepel a táblában, akkor minden sorhoz külön-külön megtalálható a bejegyzése.

2. **Adatszótár: mit tárol, kinek a sémájában van, adatszótár nézetek és prefixei**

Központi, csak olvasható táblák és nézetek gyűjteménye. Az Oracle az adatbázisleíró metainformációkat tárolja (táblák neve, oszlopok neve). A Data Dictionary a SYS user birtokában van, aki a teljes Oracle adatbázist testesíti meg. Három csoportja van az adatszótáraknak:

- "DBA_"-val kezdődő: lát mindent (az összes view-t)
- "ALL_"-al kezdődő: csak amikre van jogosultsága azt látja
- "USER_"-el kezdődő: arról van információ, ami a felhasználó birtokában van

3. SQL lekérdezés végrehajtásának folyamatát leíró ábra ismerete



Szintaktikai ellenőrzése a kiadott utasításnak.

Szemantikai ellenőrzése (létező objektumokat akarok-e lekérdezni, értelmes-e logikailag).

Shared Pool Check: küldéskor generálódik egy egyedi kliens process, ami kivált egy egyedi server processt. Utóbbi a felelős azért, hogy az utasítást a szerver elvégezze és az eredményt vissza tudja adni a kliensnek. Generál egy Hash kódot az utasításhoz. Ezt utána megvizsgálja, hogy megtalálható-e a már létező ID-k között, azaz volt-e már korábban kiadva az utasítás. Akkor az SQL szerver nem generálja le a végrehajtási tervet, mert régebben már megtette és letárolta. Ilyenkor előveszi a meglévő végrehajtási tervet és ezt hívjuk **Soft Parse**-nek. Így jut eredményre.

Ha nincs letárolva ilyen végrehajtási terv, akkor optiamlizálnia kell, le kell generálnia egy row source-t és úgy ad eredményt. Ezt hívjuk **Hard Parse**-nak.

Optimiziation: itt megtervezi az optimális végrehajtást a végrehajtási terv segítségével. Ide kapcsolódik a CBO (Cost-Based Optimization), mely egy futási módja az SQL szervernek, ez az alapértelmezett Oracle-nél.

Row Source Generation: az optimális végrehajtási tervhez legenerálja a kódot, amit ha lefuttat, megkapjuk a végeredményt.

4. Shared pool check, access path, CBO, költség, row source, row source tree

Access path: egy hozzáférési mód, hogy hogyan olvassuk ki a Row source-t. Ez egy unáris művelet: egy Row source a kimenet és egy Row source a bemenet. A JOIN műveletnek viszont 2 row source a bemenete és 1 row source a kimenete. Ezért a Hash JOIN például nem access path.

CBO: költség alapú optimalizáció, mely során az adatbázis motor a végrehajtási tervekhez költséget számol. Több végrehajtási terv készül el ilyenkor és a legkisebb költségűt választja ki. Költséget a tárolt adatok statisztikái alapján számolja ki.

Költség: adott környezetben egy becsült erőforrásigény számértéke

Row Source: rekordok halmaza (pl.: táblák sorai, nézetek sorai, JOIN műveletek, GROUPING műveletek eredménye.

5. Access path típusok ismerete

//A kifejezetten fontosakat pirossal jelzem, ezeknek a működésével tisztában kell lenni.

Heap-organized tables: Full Table Scans, Table Access by Rowid, Sample Table Scans
B-Tree indexes: Index Unique Scans, Index Range Scans, Index Full Scans, Index Fast Full Scans, Index Skip Scans, Index Join Scans

Bitmap indexes: Bitmap Index Single Value, Bitmap Index Range Scans, Bitmap Merge Table clusters: Cluster Scans, Hash scans

6. Join végrehajtásának típusai (működésük és választásuk)

Nested loop join: van két for ciklus (egymásba ágyazott): az első táblának fogja az első sorát és megnézi hogy ahhoz mennyi hozzátartozó sor van a másik táblában. Általában a kisebb row source-t választja külsőnek. CBO:

- ha a row source-ok kevés rekordból állnak (kis tábla)
- ha FIRST_ROWS módban vagyunk (nem a teljes eredményhalmazt hanem csak az első néhányat akarom visszakapni)
- ha hatékonyan tudjuk olvasni a belső row source-t (arra van index készítve)

Sort-merge join: rendezi a feltételre vonatkozóan a táblákat és egy ideiglenes táblába eltárolja a két táblát. Ezeken a rendezett táblákon megy végig. CBO:

- nagy adathalmazokra alkalmazza, ha:
- van megfelelő index a külső row source-ra, ezért nem kell rendezni
- vagy ha amúgy rendezni kell, azt itt fel tudja használni
- vagy ha nem equijoinról van szó, hanem egyenlőtlenségi feltétellel kapcsoljuk össze

Hash join: a kisebb táblára join kulcs szerint generál egy hash táblát, amikor keresi a másik (nagyobb) táblához tartozó sort a másik táblában, akkor veszi a hash alapján letárolt értéket. CBO:

- nagyobb, de különböző méretű adathalmazoknál
- ha equijoin-t használunk
- ha a kisebbik adathalmaz belefér a memóriába, akkor különösen gazdaságos

Cartesian join (cross join): olyan join, ahol nincs összekapcsolási feltétel, minden sort minden sorral párosítunk (Descaretes-szorzat). Nem önálló módszer, az előzőek közül valamelyikkel állítja elő. CBO:

- ha nincs join feltétel
- ha olyan sorrendet alakítunk ki több tábla között, hogy a szomszédos tábláknak nincs közvetlen kapcsolata
- ha ez a hatékony megoldás (ha két pici táblánál join-olni kell mindkettőt egy nagy táblára)
- ha ilyet észlelünk, érdemes átvizsgálni a lekérdezést, hogy biztos jól írtuk-e meg

7. Statisztika szerepe és tartalma, hisztogramok, index clustering factor

Statisztika: mindenről tárol plusz információt, hogy az optimalizálót segítse a helyes döntésben:

- táblák: hány sora van, hány blokkban tárolódnak az adatok
- oszlopok: különböző értékek (Number of Differenc Values) - ha egy oszlopra van WHERE feltétel és van rajta index is, NULL-ok száma, adatok eloszlása
- indexek: levél blokkok száma, famagasság stb.
- rendszerről: I/O, CPU teljesítmény és kihasználtság

Hisztogramok: a CBO alapból egyenletes eloszlásúnak feltételezi az adatokat, azaz az előforduló különböző értékekből kb. ugyanannyi van egy oszlopban. Ha ez nem így van, akkor a JOIN és szűrő kifejezések becslése pontatlan lehet. A hisztogram azt mutatja meg, hogy melyik érték milyen gyakori

egy oszlopban. Ez segítséget ad a kardinalitás becslésében. Kardinalitás: hány sorból fog állni az adott művelet eredményeként kapott row source.

Index clustering factor: a diszken lévő adatok csoportosítása mennyire kapcsolódik az indexbeli struktúrához (0 és 1 közötti szám). Ha 0-hoz van közel, akkor jó indexhez nyúlni, mert fizikailag az azonos indexkulcsú rekordok vannak egy halmazban. Ha látja, hogy a faktor 1, nem fog az indexhez nyúlni. Méri a sorok fizikai csoportosulását az idex értékéhez mérten. Segít az optimalizálónak eldönteni, hogy egy index scan vagy egy full table scan lenne e hatékonyabb az adott lekérdezésnél. Alacsony clustering factor index scan hatékonyságára utal.

8. Hintek szerepe, általános szintaktikája, 5 darabot tudni a jelentésével együtt

Ki tudunk kényszeríteni bizonyos fajta eljárást a végrehajtási tervben, pl. access path választást, join módszer választást, join sorrend választást. Azaz adhatunk tippeket az optimalizálónak. Inkább csak tesztelésre használjuk. Ez egy speciális komment:

SELECT /*+ hint_text */ ... írhatjuk: SELECT, UPDATE, INSERT, MERGE vagy DELETE után.

A + jel elé nem kerülhet szóköz. Csak egy hint komment lehet egy lekérdezésben, de lehetőket kombinálni szóközökkel elválasztva. Hiba esetén a rendszer nem jelez.

INDEX_FFS: index fast full scan
INDEX_SS: index skip scan
FULL: full table scan a megadott táblára
USE_MERGE: sort-merge join
USE_HASH: hash join

9. Oracle AB 2 fő része: adatbázis (vagyis a fizikai struktúrák rendszere), példány (instance), vezérlő fájl (control file), helyrehozó napló fájlok (redo log files), adatfájlok, Paraméterfájl, Jelszófájl, System Global Area (SGA) terület, szerverfolyamatok, 69. dia ábra, PGA

Amikor egy kliens folyamat megszólítja az adatbázist, akkor az Oracle szerver elindít egy szerverfolyamatot, amely lehetővé teszi az alkalmazás utasításainak végrehajtását. Az Oracle egy példány indításakor háttérfolyamatokat is elindít, amelyek kommunikálnak egymással és az operációs rendszerrel. A háttérfolyamatok kezelik a memóriát, puffereket, végrehajtják az írási, olvasási műveleteket a lemezen, karbantartásokat végeznek.

Vezérlő fájl: az adatbázis konfigurációját tárolja. Egy kontrolfájlnak kell léteznie egy adatbázisról. Az instance indításakor az adatbázis rákapcsolásához be kell olvasni a vezérlő fájlokat. Automatikusan módosul, ha új fájlt adunk az adatbázishoz. Érdemes legalább három másolatot tartani. A szinkronizálást az Oracle automatikusan elvégzi.

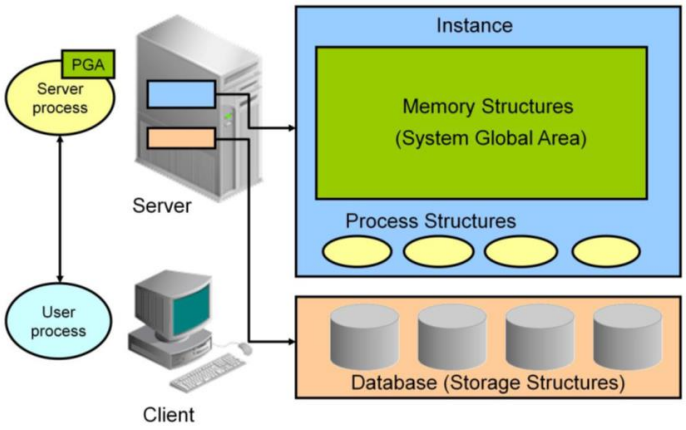
Helyreállítási log fájlok: a helyreállításhoz szükséges napló információkat tároljuk benne. Érdemes különböző lemezeken több másolatot tartani belőle. A naplóíró folyamat (log writer process) felel azért, hogy a memóriából a napló adatok kiíródjanak a diszkre. Ezeket a fájlokat csoportokba sorolják, másolatok készülnek. Minden csoportnak van azonosítószáma.

Adatfájlok: az adatbázisban tárolt adatok szerepelnek benne

Paraméterfájl: olyan paraméterek, melyek befolyásolják egy példány méretét és tulajdonságait (pl.: SGA memória részeinek mérete)

Jelszófájl: a rendszergazdák (SYSDBA) jelszavát, a felhasználók, akik kapcsolódhatnak, elindíthatnak egy Oracle példányt

Példány: a memóriában lefoglalt (SGA System Global Area) terület és azok a szerverfolyamatok, melyek az adatbázis-műveletek végrehajtásáért felelnek.



PGA (program global area): memóriaterület a szerveren, ami kizárólag az adott szerver processhez tartozik.

SGA (system global area): közös memóriaterület és minden server process el tudja érni. Itt kerül tárolásra az összes lekérdezés hash kódja, amiket az utóbbi időben futtattunk.

10. Oracle memória szerkezete: SGA (database buffer cache, redo log buffer, shared pool), PGA

Database Buffer Cache: itt tárolja le a diszkről kiolvasott adatokat (memóriaterület) és ezt dolgozza fel. Ha az adat módosul, ide menti le, majd ennek a tartalmát írja ki a diszkre. Ha valami adat kell, előbb megnézi, hogy benne van-e, ha nincs itt, csak akkor olvassa ki a diszkről. Ha betelik, a legrégebben használt adatblokk törlődik, annak a helyére olvassa be az újat.

Redo Log Buffer: az a memóriaterület, ahova a redo log naplóadatok először beíródnak. Körkörös szerkezete van, ez mentődik időnként a diszkre. Szerverfolyamat indulásakor lefoglalja ezt a területet, befejezéskor pedig felszabadítja.

Shared Pool:

- **Library Cache:** ha megkap egy SQL lekérdezést a szerver, ahhoz hash-el generál egy ID-t és az a lekérdezés az ID-val tárolódik a Shared Pool-ban és a hozzá tartozó Execution Plan is. Ezért van, hogy ha újra megérkezik a lekérdezés a szerverhez, nem kell újra kiszámolnia.
- **Data Dictionary Cache:** a data dictionary-t nagyon gyakran kell lekérdezni, ezért a feléje érkezett lekérdezések és eredmények itt eltárolódnak, hogy ha új kérés jön, ami egy ugyanolyan DD adatotkér, akkor innen ki tudja olvasni.
- **Server Reult Cache:** egy lekérdezés eredményét is tárolja, így nem kell az Execution Plan-t se lefuttatni, innen ki tudja olvasni.

PGA (program global area): memóriaterület a szerveren, ami kizárólag az adott szerver processhez tartozik.

SGA (system global area): közös memóriaterület és minden server process el tudja érni. Itt kerül tárolásra az összes lekérdezés hash kódja, amiket az utóbbi időben futtattunk.

11. Háttérfolyamatok (DBWn, LGWR ismerete)

Van néhány háttérfolyamat, melyek közül vannak kötelező processek és opcinális processek. A háttérfolyamatok kezelik a memóriát, puffereket, végrehajtják az írási, olvasási műveleteket a lemezen, karbantartásokat végeznek.

- **DBWn:** database writer process, azért felel, hogy a memóriából kiíródjanak a tábla adatok a diszkre.
- **LGWR:** log writer, a naplóinformációk kiírásááért felelős.

12. Táblatér, szegmens, extens, adat block

Egy tábla adatai **adatblokkokban** vannak tárolva és valameddig találunk fizikailag egymás után elhelyezkedő területeket a diszken, ez fogja felépíteni az **extenst**. A **szegmens** több extensből épül fel, de ugyanahhoz az objektumhoz tartoznak. A szegmenseket **táblatérben** tároljuk. Az adatbázis több táblatérből épül fel. Egy táblateret egy fizikai fájlban vagy ha nem fér el, több fájlban is lehet tárolni. Az adatfájlok mindig csak egy táblatérhez tartoznak. A táblatér a logikailag összetartozó adatbázis objektumokat tárolja.

13. SQL utasítás végrehajtásának folyamata: példány, felhasználói folyamat, szerver folyamat

Fut egy listener a szerveren, az látja, hogy a user (user process) csatlakozni akar. Elindítja a szerver process-t és a szerver process fog az instance-al kommunikálni és adja vissza az eredményeket a usernek.

14. Relációs ABKR-ek esetén mi a tranzakció fogalma

Tranzakció: az adatbázisműveletek végrehajtási egysége, mely DML-beli utasításokból áll. Egy tranzakció mindig egy konzisztens adatbázis állapotból indul ki és a módosítások egy olyan sorozatát tartalmazza, melyek végén ismét egy konzisztens adatbázis tartalom áll elő.

15. ACID tulajdonságok

Atomosság: a tranzakció összes elemi utasítását vagy végrehajtjuk vagy egyet sem (mindent vagy semmit). Másképp: A tranzakcióba bevont DML utasításokat egy egységként kell kezelnie az adatbázis-kezelőnek.

Konzisztencia: a tranzakciót követően is teljesülnek az adatbázisban előírt konzisztencia megszorítások (integritás megszorítások), azaz az adatelemekre és a közöttük lévő kapcsolatokra vonatkozó elvárások.

Elkülönítés: ha több tranzakció fut párhuzamosan, egyszerre és ezek az utasítások keverednek, a tranzakciók végén az állapot olyan, mintha nem keveredtek volna (úgy kell lefutnia, mintha azalatt semmilyen másik tranzakciót nem hajtanánk végre)

Tartósság: ha befejeződött egy tranzakció, annak az adatbázisban kifejtett hatása nem veszhet el. Másképp: A lezárt tranzakciók eredménye nem veszhet el.

16. Rendszerhiba, helyreállítás, naplózás, archiválás

Kiindulás: Tartósan tároljuk az adatokat a diszken, feldolgozáskor beolvassuk a memóriába. Ha a memória (áramszünet, rendszer elszáll) tartalma sérül/törlődik, akkor a tranzakció bizonyos utasításainak a változtatásai ki lettek írva a diszkre, bizonyos utasítások elvesztek a memóriából.

Rendszerhibánál a memória-tartalom veszik el. Kiváltó ok pl. áramszünet vagy rendszer összeomlás. Védekezés formája a naplózás, melynek 3 módszere van:

- **semmisségi (undo):** régi érték tárolódik el a naplóbejegyzésben, hogy vissza tudjunk görgetni
- **helyreállító (redo):** új értékek tárolódnak le a naplóban, nem visszagörgetünk, hanem le kell játszani az új utasításokat a lemezről hiba esetén
- **a kettő együtt (undo/redo):** régi és új értéket is tároljuk a naplóbejegyzésben.

Helyreállításkor a naplóbejegyzések segítségével újra konzisztens állapotba hozzuk a rendszert. Vagy visszagörgetéssel (rollback) vagy az utasítások újra játszásával a kimentett naplóadatok segítségével.

Archiválás: ilyenkor nem a memória, hanem a diszk sérülését akarjuk orvosolni. Biztonsági mentéssel tudunk védekezni azzal, hogy másik diszken tárolunk egy biztonsági másolatot az adatbázisról. Részleges mentés: csak az utolsó mentés óta változott adatokat mentjük, így mentési időt és helyet takaríthatunk meg.

17. Classic Database Architecture are based on SMP (Symmetric Multi Processor) Systems. Vázolja fel ezen rendszerek 3 fő jellemzőjét!

Klasszikus adatbázis rendszer SMP-re épül, SMP alapú. Több processzor szolgál ki egy időben több process-t. Mindegyik process egy processzorhoz van rendelve, de a processzorok közösen használják a system bus-t, a memóriát, tárolóegységet. Mindegyik processzornak megvan a saját cache-e, viszont ezeket közösen használják.

Vertikális skálázhatóság alapú: egy vashoz kapcsolódó komponenseket tudjuk fejleszteni, de nem tudjuk a végtelenségig csinálni.

18. **Massive Parallel Databases are based on a „shared-nothing“ Architecture. Sorolja fel ezen rendszerek 3 fő jellemzőjét!**

Ennek az egyik képviselője a Teradata adatbáziskezelő rendszer. Az MPP architektúrában van a

- Controller Node: csak egy van belőle, metadata-t tárol és a data dictionary-t. Megmondja, melyik Data Node tárol egy adatot, amit szeretnénk elérni.
- Data Node: azért van több DN, mert ők csomópontok, amik az adatokat szétosztva tárolják.
- Shared-nothing: ezek a DN-ok különálló egységek és nem osztoznak egymás között a memórián, diszken. Minden DN-nek megvan a saját CPU-ja, ami több CPU-t tartalmaz, memóriája és diszkje.
- A DN-ek egyszerre dolgozhatnak ugyanazon a lekérdezésen. Több processzor párhuzamosan futtat egy folyamatot.
- **Horizontálisan skálázható:** újabb Node-okat tudunk hozzácsatolni a rendszerhez és így növeljük az erőforrást.

19. **Milyen felhasználási területe van az MPP adatbázis rendszereknek?**

Az MPP-t nagy cégek használják, tőkeerős cégek, mert nagyon drága rendszer. Ha rengeteg adatról van szó, analitikus rendszereknél, adattárházak támogatására. Pl.: itthon T-Systems. Olyan komplett rendszerekként értékesítik, melyekben adatbázis-kezelésre optimalizált hardver és szoftver (operációs rendszer, adatbázis szerver) elemek egyaránt megtalálhatók.

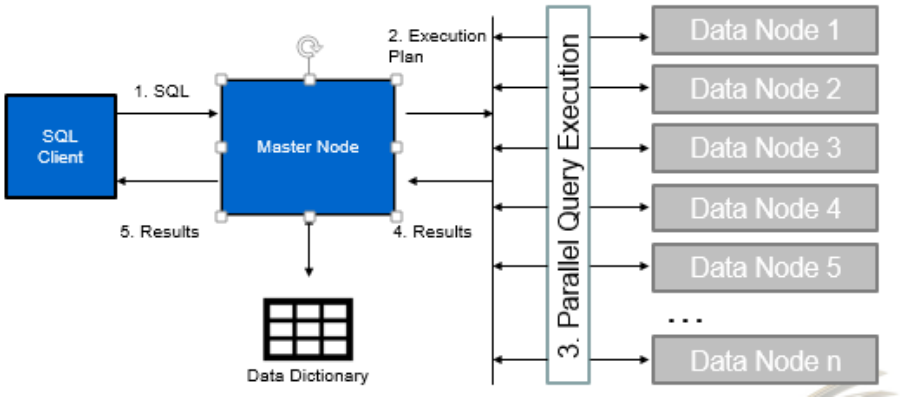
20. **Milyen data distribution stratégiákat ismer MPP rendszerek esetén? Ismertesse ezek jellemzőit!**

A Head Node szét akarja osztani egy hatalmas tábla adatait a Data Node-okra:

- **Hash distribution:** lefuttatok rajta egy hash algoritmust és annak az eredménye határozza meg, hogy melyik DN-ra kerüljön a rekord. A fejlesztő kiválaszt egy vagy több oszlopot a táblából (distribution column), közlik az adatbázissal, hogy ennek az oszlopérték alapján dobja szét az adatbázist a DN-ekre, lefuttatja rajta a hash funkciót és az eredmény határozza meg hogy melyik DN-re kerül az adat. Ha egy oszlopban 2 rekordnak ugyanaz az értéke, akkor ugyanarra a DN-re kerülnek.
- **Even distribution / Round robin:** fogja az MPP a táblát és a rekordokat szétosztja sorban, egyenletesen. i-edik rekordot az i mod n lemezre írja, ezzel biztosítja a rekordok egyenletes eloszlását a rendelkezésre álló lemezekre. Fogok egy rekordot, 1-es megy az 1-es DN-re, 2-es a DN-re, ha 2 DN-em van, a 3-asnál újra az 1-es DN-re és így tovább. Egyenletesen szétosztom a rekordokat.
- **Range partition:** fogok egy distribution column-t, ha a rekord érték egy adott intervallumban van, akkor rakd egy adott DN-re, ha más értéke, másik DN-re stb. Itt intervallumokat adok meg és a DB ez alapján tárolja el. Pl.: dátumok szerint, kezdőbetű alapján stb. Tartomány partícionálás, a rekordokat bizonyos tartományok alapján osztja szét a lemezekre. Pl. nevek, A-H-ig 1.lemezre, I-Z-ig 2. lemezre stb.
- **Script on all:** kis tábláknál érdemes, azaz egy tábla rekordjait írd az összes DN-re. Tehát bebiztosítom, hogy egy tábla adatai mindegyik DN-re felkerülnek.

21. **Milyen rendszert ábrázol az alábbi ábra! Ismertesse az ábrán látható adat elérési folyamatot!**

MPP rendszer: a kliens felveszi a kapcsolatot a Controller Node-al. A CN az adatkatalógusa alapján (data dictionary) meghatározza az Execution Plant-t és a kérést továbbítja a DN-eknek. A DN-ek párhuzamosan végrehajtják a saját adataikon a lekérdezést., majd visszaküldik az eredményt a CN számára, ami összegyűjti az eredményeket, majd továbbítja a kliensnek. Az írás és az olvasás párhuzamosan.történik.



22. **Soroljon fel 3 relációs MPP adatbázist!**

Self managed adatbázisok: amikor ott van a vas a cégnél, azon van az RDBMS, az adatok is itt vannak. Pl.: Teradata, HP vertica, Netezza

On demand adatbázisok: felhős megoldás. Pl.: Snowflake, Google Big Query, Microsoft Azure Data Warehouse, Amazon Redshift

23. **Írjon 8 kiemelten jellemző tényt a Teradata DBMS-ről!**

1. MPP architektúrán alapul
2. Relációs adatbáziskezelő rendszer,
3. SQL nyelvet értelmez
4. adattárházakban használják
5. saját hardvere van, amit meg kell vásárolni (eredetileg Linux-ot futtatott, ma már szélesebb körű támogatottsága)
6. shared nothing architektúrája van
7. lineárisan skálázható
8. nagyon fejlett az optimalizáló komponense
9. hatékony adatbetöltő eljárásokat ismer (adattárházaknál ez fontos)
10. automatikus adatdisztribúciót alkalmaz.

24. Ismerje az alábbi ábrán felvázolt Teradata rendszer-komponenseket, ismertesse ezek szerepét és jellemzőit.

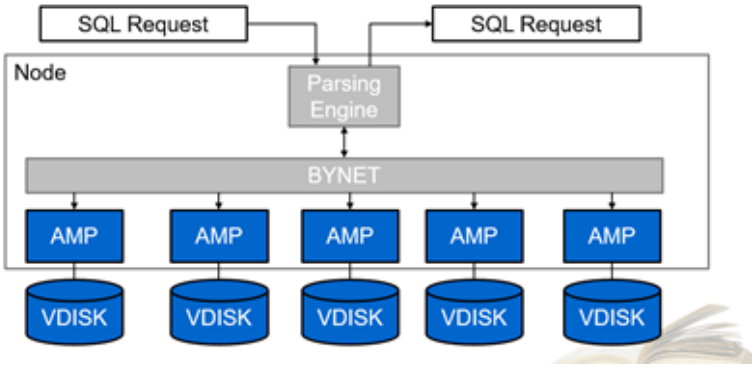
Node: alapegysége a TeraData-nak. Minden Node-nak van saját OS-e, processzora, memóriája, diszk komponense van. Több node egy kabinetté csatolható.

PE: CN helyett Parsing Engine van, elemei: Session Controller, Parser, Optimalizáló, Diszpécser. A PE egy virtuális processzor, itt van a session felügyelete. A PE átveszi az SQL utasítást, megnézi, hogy helyes-e, benne van az optimalizáló. Ő dobja szét íráskor az adatokat is az AMP-kre.

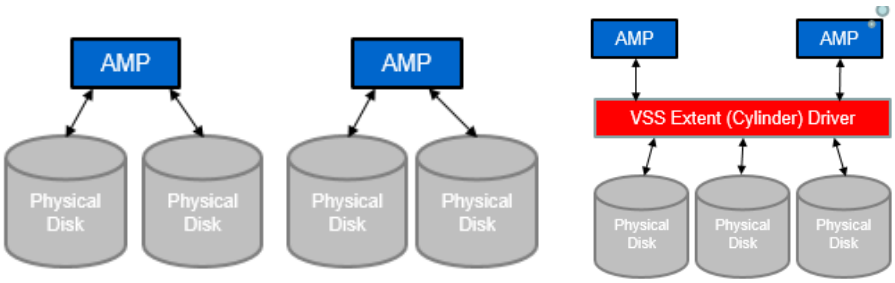
BYNET: összeköti az DN-eket a CN-el. Nagy sebességű hálózat, összekapcsolja és kommunikál az összes AMP-vel és PE-vel a rendszerben. Nagy teljesítményű, Hibatűrő, kiegyensúlyozott terhelésű, skálázható.

AMP (hozzáférési modul processzor): Ha valaki dolgozik adatokon, akkor az AMP felel az adott táblarészért, hogy azokra a sorokra vagy sorra egy lock történjen addig, amíg tart a tranzakció. Az AMP-ek előkészítik az adathalmazt, amit visszaküldenek a PE-nek, az előszortírozás, aggregálás is az AMP-en történik. Az AMP-k úgy működnek, mint egy DN, mindent párhuzamosan tudnak csinálni.

VDISK: lemezek, adatokat tárolnak, saját AMP-hez kapcsolódnak



25. Miben különbözik az alábbi 2 architektúrájú Teradata DBMS? Írja le a fő jellemzőiket és a különbséget!



1. Ábra: a Teradata alap felállása. Van az AMP (Data Node) és a hozzátartozó diszkek. Minden egyes AMP-nek megvan a saját diszkje, amit tud használni. Ezeken nem osztoznak, a Node-ok klikkekben vannak.

2. Ábra: Továbbfejlesztett verzió az ún TVS (Teradata Virtual Storage). Az AMP-k osztozzanak a diszkeken, azaz a tárolókapacitáson közösen az igényeknek megfelelően, amennyire szüksége van az AMP-nek annyi tárolóegység lesz hozzárendelve. Ezt az igényt a Multi-Temperature Data írja le. Három típust különböztetünk meg:

- az adat, amiket gyakran kérdeznek le (hot adatok)
- kevésbé gyakran (warm)
- nagyon ritkán (cold)

Cél: a gyakori adatok kiszolgálása gyors HDD-ről történjen. Magától történik ezeknek a meghatározása a lekérdezési statisztikák alapján. Lehet egy architektúrában HDD és SDD is.

A drágább diszkekre a gyakrabban lekérdezendő adatokat teszik, a ritkábban lekérdezőket pedig egy olcsóbbra.

26. TVS típusú Teradata esetén mit jellemez a „Data Temperature" fogalma? Milyen 3 kategóriát ismer, és mit jelentenek ezek a fogalmak? Miért különböztetik meg ezt a 3 kategóriát az adatok tekintetében?

Azokat az adatokat, amiket sűrűn olvasnak fel a felhasználók vagy egy gyorsan elérhető részen akarjuk tárolni vagy egy olyan tárolóegységen tartjuk, ami gyors (pl.: SSD).

- az adat, amiket gyakran kérdeznek le (hot adatok)
- kevésbé gyakran (warm)
- nagyon ritkán (cold)

Cél: a gyakori adatok kiszolgálása gyors HDD-ről történjen. Magától történik ezeknek a meghatározása a lekérdezési statisztikák alapján. Lehet egy architektúrában HDD és SDD is.

27. Teradata data distribution folyamata.

Teradata hash algoritmust használ hogy véletlenszerűen elossza a tábla sorait az AMP-k között. Az elsődleges indexoszlop értékét használják a Teradatában az AMP-eken való tárolás meghatározásához hash algoritmus segítségével. A Primary Index kiválasztása meghatározza, hogy egy tábla sora egyenlően vagy egyenlőtlenül lesz elosztva az AMP-k között. Egyenlően elosztott tábla sorok, egyenlően elosztott Workload-okat eredményeznek. Teradata automatizálja a fizikai Data Location-ök elhelyezését és karbantartását.

28. Teradata esetén mit jelentenek a következő fogalmak: primary key, primary index, partitioned primary index, secondary index.

Primary Key: ami ugyanaz a Teradata esetén, mint az RDBMS-ben. Ez azonosítja egyértelműen a sorokat a táblában. Erre alapból nem épül fel az index, külön meg kell adni. Nem muszáj PK-t megadni.

Primary index: egyrészt index és megmutatja, hogy melyik AMP-re lesz elosztva az adott rekord. Az indexérték hash képe (row hash) mutatja meg, hogy melyik AMP-re kerül a sor. Muszáj definiálni.

- Unique Primary index: egy oszlopban egy érték csak egyszer fordulhat elő
- Non-Unique Primary Index: egy oszlopban egy adott érték többször fordulhat elő

Azt befolyásolja, hogy a Non-Unique esetében kialakulhat egy nem kiegyenlített elosztás. Ha van sok sorom az eredeti táblában, melynek mind ugyanaz a PI helyén lévő értéke, a sok sor egy helyre fog képződni. Az AMP-n a sorok rendezve vannak a row hash alapján.

Partitioned primary index: megadjuk a Primary Indexet, meg a Partition column-t. Az AMP-n a sorok először a Partition Column alapján vannak elrendezve és utána a row hash alapján.

Secondary index: lehet más oszlopokra indexet felépíteni. Ha ezt megadjuk, a fizikai helyét mutatja meg az adatnak, de itt kell egy plusz adatstruktúra, ahol le van tárolva a plusz információ. Ez a Primary index esetében nem kellett. Nem muszáj definiálni. A teljesítmény javítására használják, elkerülve ezzel a full table scan-t. Olyan, mint egy primary index, mivel lehetővé teszi a rendszer számára a táblázat egy sorának keresését. Nincs hatással a sorok elosztására az AMP-eken. A másodlagos indexkeresés általában olcsóbb, mint egy full table scan. Plusz költség a lemezterületen és a karbantartásban, de szükség esetén hozzáadhatók vagy eldobhatók.

29. **Emeljen ki 6 olyan tulajdonságot, ami az SQLite adatbázis-kezelőre jellemző! Adjon meg 3 nagyon gyakori alkalmazási területet SQLite választásához! Milyen hiányosságai vannak az SQLite-nak (3 db-ot írjon)!**

Egyszerű, egy fájlból álló adatbázis, ott történik a feldolgozás (nincs kliens-szerver architektúra). Kompakt, ezért nagyon szeretik (pl.: mobiltelefonokba alapból benne van). Annak ellenére, hogy pici, az ACID-nek teljesen megfelel (tranzakciókat kezel). C-ben írták. Mindössze egy fájl, így könnyű biztonsági mentést készíteni (egy fájl egy db). A legtöbb programozási nyelv támogatja. Gyors, mivel nem kell sokfelé keresgélni. Pár KB-ot eszik memóriaszinten.

Alkalmazási területei: Telefonok, beépített IoT rendszerek, Website-ok alá is berakják (viszonylag statikus, kis forgalmú weblapok), Egy komplex feladatot be lehet rajta mutatni, hogy egy SQL adatbázisban ez hogy nézne ki, SQL oktatásra is kiváló.

Hiányosságok: Kevésbé hatékony, ha egyszerre sokan akarnak írni/olvasni, egy fájlnál ez nehéz. Egy adatbázis (fájl) kb. 80-100Mb-os lehet max. Jogosultságkezelés nincs benne

30. **Fogalmazza meg a HIVE adatbázis-kezelőről legfontosabb tanultakat 5 mondatban.**

A Hadoop keretrendszer része, HDFS (hadoop distributed file system) fájlrendszeren tárolt adatokat strukturált formában tudja kezelni. A lekérdezés feldolgozásában használja a mapreduce algoritmust. Lekérdező nyelve a HiveQL, ami az SQL nyelvhez hasonlít, ezáltal a relációs adatbázisokban megszokott módszerekkel tudjuk az adatokat kinyerni. A Hiveban megírt lekérdezések MapReduce jobokra fordulnak le. A Hive driver felelős a megírt lekérdezések fordításáért, optimalizálásáért, illetve végrehajtásáért. Gyakran alkalmazzák vállalati adattárházak kiegészítéseként, az adatok betöltésére az ETL folyamatokat használja (kinyerjük az adatokat, transzformáljuk és betöltjük).

(A 2000-es évek elején Facebooknál rájöttek, hogy van egy nagy hadoop rendszerük, aminek egy részét fel tudják dolgozni, de valamit kezdeni kellett a többi adattal is. Mapreduce keretrendszert kevesen tanulták meg akkoriban. Egy rakás elemzőjük volt, aki jól tudott SQL-ül. Ezért találták ki, hogy a HDFS rétegre egy SQL-szerű környezetet húznak, ahol nagyon hasonló, mint egy SQL és innentől kezdve az adatelemző csapat teljes egészében képes működni, így jött létre a HIVE.)

31. **Hasonlítsa össze a MySQL és PostgreSQL adatbázis-kezelőket 12 db (előadáson vett) szempont szerint**

- Szlogen: legnépszerűbb / legfejlettebb
- Nyílt forráskódúak
- ACID megközelítés: többféle db engine, de az innoDB ACID kompatibilis / nagyon ACID
- Replikációkezelés: Master-Master, Master-Slave / Csak Master-Master
- Teljesítmény: OLTP, magas intenzitású rendszerek alá / OLAP rendszerek, analitikus feladatokra
- Biztonság: mindkettőnél biztonságos az adattovábbítás, jogok kezelése is megoldott

- Materializált nézet: ideiglenes táblák alapján / natívan támogatja
- Common Table Expression: ma már mindkettő tudja
- Index: B-fa, hash, cluster indexek (utóbbi csak innoDB) / b-fa, hash, partial index
- Felhő szolgáltatást mindkettő támogatja
- JSON és XML fájlok támogatása: némi lemaradással tudja / régebb óta támogatja
- Támogatják a JOIN-okat, adattípusokat, tárolómotorokat, megkötéseket

32. Jellemezze a 3Vs tulajdonságokat!

Az RDBMS-ek általában egy drága vasra települnek fel, ami rengeteg adatot kell, hogy feldolgozzon. 1-2 év után ha kiderült, hogy gyenge a hardver (userek, performancia, adatmennyiség), nagyon sokba került bővíteni. Nem lehetett előre látni, hogy egy ilyen scale up megoldja-e a problémát vagy csak enyíti. Tehát a **volume** kritériumnak egyre nehezebben tud megfelelni ez a struktúra.

A **variaty** egy elég gyenge pontja a relációs adatbázisoknak, mert ott előre meg kell tervezni, létre kell hozni az adatbázist, így a modellt folyamatosan módosítani kell az adatok átalakulása során. Ez sokszor adatmigrációval jár, ami elég körülményes.

A **velocity**: ha egy relációs adatbázist elárasztjuk adatokkal (write) a lekérdezési performancia jelentősen romlik. Ha sokat olvasunk (read), nem fogunk tudni jól írni bele. Inkább csak az egyiket választhatjuk. Rengeteg adat érkezik, melyeken szeretnénk analitikus függvényeket futtatni.

33. Milyen felhasználási területekről érkezett az első igény a nem tisztán relációs adatbázisok iránt?

Elsősorban a közösségi alkalmazások/oldalak felől volt igény rá

34. Soroljon fel igényeket az új rendszerek felé?

Social app: szeretnénk építeni egy olyan rendszert, amiben eltároljuk, hogy a személyek ismerik egymást. A személyekről információt és a kapcsolatot a személyek között. Relációs modellben önmagára mutatna az egyed. Macerás valaki barátainak a lekérdezése, mert 3 tábla kell hozzá ez még kezelhető, a probléma akkor kezdődik, ha valaki barátainak a barátait akarom megkeresni. Minél mélyebbre megyek, egyre kezelhetetlenebb, olvashatatlanabb lesz.

Üzleti felhasználás: egy cég gyakran módosíthatja az igényeit. Pl.: ügyfelekről tárolunk adatokat. Ha a megrendelő szeretné, hogy vonalas és mobil szám is legyen az ügyfeleknek. Módosítani kell a táblát, új oszlopot kell hozzáadni. Két hét múlva megint változtatna, egy ügyfélnek legyen magán telefonszáma és üzleti telefonszáma is, megint táblát kell módosítani. Később úgy gondolják, hogy egy ügyfélnek lehessen bármennyi telefonszáma. Ehhez fel kell borítani a modellt, létre kell hozni egy új táblát, hogy a telefonszámokat el tudjuk tárolni.

35. Mit tud a CAP tételről?

Az elosztott rendszerek esetén az alábbi három követelmény közül maximum kettőt tudunk mindig garantálni:

- **Consistency:** egy clusternél az adatok minden csomóponton egy adott pillanatban egyformák (a különböző node-okon egyforma adatokat találunk – replikáció)
- **Avaibility:** egy clusterben mindig kapok választ az adatlekérdezésre. (a response time megfelelően gyors, ez persze függ attól, hogy a feladat alapján milyen válaszidőket várunk el)

- **Partition tolerance:** ha van egy cluster és a clusternek egyes csomói elhalnak, akkor elérhető-e a rendszer vagy nem. A Reddis-nél pl.: ha elhal egy node, a választ még megkapja egy másiktól.

36. Mit jelent a BASE kifejezés? Jellemezze a 3 fogalmat 1-1 mondatban!

BASE: túl szigorú az ACID a NOSQL technológiákra, így kialakítottak egy új dolgot. Ez sokkal kevésbé szigorú a követelmény a tranzakciókra vonatkozóan.

- **Basic Avabiality:** az adatbázis alapvetően elérhető
- **Soft state:** nem kell mindig konzisztensnek lennie az adatoknak a masterrel
- **Evential Consistent:** a replikáció később is történhet, nem azonnal → a master nem azonnal dobja tovább a tranzakciót a slave-eknek, hanem majd megpróbáljuk továbbdobni és majd valamikor konzisztens lesz az egész cluster

37. Milyen 2 fő replikáció típust ismer? Jellemezze őket 2-3 mondatban!

Sharding: az adatokat szétosztja különböző node-okra, de úgy hogy minden nodera egyedi adathalmaz kerül (nem replikálja, hanem szétosztja) – MPP. Pl.: tábla rekordjai közül 5 rekord az egyikre, 5 a másikra.

Replication: az adatokat kompletten mindegyik node-re rámásolja (teljes adathalmaz másolat).

- **Master-Slave:** van egy master ahová írni lehet, ahonnan olvasni is lehet és vannak slave-ek melyekről csak olvasni lehet, ez kezeli a slave szerverek szinkronizációját
- **Peer to Peer:** mindegyik node egyforma rangú és mindegyikre lehet írni és olvasni is róluk. A node kezeli a másolatainak a szinkrinozálását

Van olyan rendszer, ami mindkettőt tudja pl.: Redis is ilyen.

38. 4 fő NoSQL rendszer kategória

Key-Value store: kulcs-értékpár tároló pl.: Redis
Column store: oszloptároló pl.: HBase
Document store: dokumentumtároló pl.: MongoDb
Graph stroe: gráf adatbázis pl.: Neo4J

39. Hasonlítsa össze a vertical scaling és a horizontal scaling fogalmakat!

Vertical scaling: egy vashoz kapcsolódó komponenseket tudjuk fejleszteni, de nem tudjuk a végtelenségig csinálni. Több erőforrást(cpu,memoria) adunk a géphez.

Horizontal scaling: újabb Node-okat tudunk hozzácsatolni a rendszerhez és így növeljük az erőforrást.

40. Soroljon fel tipikus use case-eket Key Value store használatára!

Caching: adatokat átmenetileg tárolok (ramban tárolja a Reddis is)
Session store: session kezelés lefejlesztésére is jó
Messaging channels: mint egy Twitter funkció (van egy publisher, subscriber, a publisher valamit egy kulcs-érték mögé ír, minden feljelentkező megkapja a kulcs-értéket)
Creating secondary indexes: indexelésre is használható

41. Mondjon 3 Key vaue store ABKR-t!

Redis, Amazon DynamoDB, Memcached, Microsoft Azure Cosmos DB, Hazelcast, Aerospike

42. Milyen programozási nyelvekhez van API a Redis esetében (min. 5-öt!)

Java, Javascript, Scala, Python, Ruby, R, C, C++, C#, R

43. Miért in-memory típusú rendszer a Redis?

Az egész adatbázis a RAM-ban van, ez a fő előnye, azaz nagyon gyors. Képes akár 100 000 set / sec-re. Benchmark teszt hajtható végre saját paranccsal, hogy tudjuk hogy set-eknél, get-eknél mennyire gyors.

44. Soroljon fel 5 parancsot a Redis esetében! Magyarázza el a jelentésüket.

set mykey myvalue / get mykey
incr / incrby testnumber
publish mychannel "Hello"
keys my*
del mykey / flushdb / flushall

45. Milyen adattípusokat ismer Redis esetén?

Key-Values, Hashes, Sets, Sorted Sets, Lists (készíthető queue vagy stack), Channels

46. HDFS és HBase rendszerek kapcsolata és különbsége

HDFS: elosztott fájlrendszer, ami eltárolja a nagy fájlokat. A gyors keresést nem támogatja. A batch feldolgozást nem támogatja. Szekvenciálisan tud csak csatlakozni az adatokhoz.

HBase: egy adatbázis, ami a fenti tárolón fut. Nagyon gyorsan lehet vele keresni egy nagy táblában. Támogatja a batch feldolgozást. Random csatlakozik az adatokhoz, indexeket használ az hfile-ok tárolásához.

47. HBase is more a "Data Store" than "Data Base". Miért?

Oszloporientált. Flexibilis a sémája, oszlopokat menet közben is adhatunk hozzá. Nem kell adatokat eltárolni minden CF-ben. Használ mapreduce-t, azaz neki szolgáltat adatot. Horizontálisan skálázható. A nem strukturált adatokat is jól kezeli. Denormalizáltan tárolja a táblákat, gyorsabban működik. Több verziót tárolhat el egy column family-ben.

48. HBase logikai adatmodell (row key, column family, column, column qualifier, cell value, version/timestamp). Sparse table, dynamic columns.

Tábla létrehozásakor megadjuk a **column family**-t, a keyspace pedig a séma. Amikor táblát készítünk, mindig meg kell adni a keyspace-t. Ez az egyetlen kötöttség.

A column family-k sorokban állnak és minden egyes sornak van egy sor azonosítója (**row key**) és több oszlopot tartalmazhat. Minden oszlopnak van egy neve, értéke és egy timestamp-je.

A column family a logikailag összetartozó adatoknak a csoportja. A **timestamp** azért fontos, mert ez alapján határozza meg hogy melyik volt a legutolsó rekord, mert mindig azt fogja visszaadni. Ha egy adatból több változat van, fordítva rendezi, így a legfrissebbet tudja szolgáltatni.

Column: minden oszlopnak van neve, értéke és időbélyege.

49. HBase fizikai adatmodell (HFile tartalma, regions, table cell tartalma), region server.

A régió szerverek tartalmazzák az adatokat. Az HMaster szerver a karmester, ő hajtja végre a DDA műveleteket, ill. az adatok felosztását a szerverek között. A felosztott darabok régiókban tárolódnak a régió szervereken és a master végzi ezeknek a szétosztását.

A régió szervert egy HDFS datanode szolgálja ki. Ennek a tetején vannak rajta az HBase szolgáltatásai. A régió szerveren belül vannak régiók, melyek tartalmaznak column family darabokat. Az egyes column family-khez tartozik egy Memstore és HFiles. Amikor valaki be akar írni a column familybe, megkeresi, hogy hová kell írni, azaz hol tárolódik az adat.

A HDFS fájlrendszer egy elosztott fájlrendszer. Van nameNode és dataNode. Előbbi a könyvtáros, tudja mi hol található, a dataNode-ok ténylegesen tárolják az adatokat és ők önmagukban képesek a replikációra is.

Hfile: a fájl adatblokkokra van osztva (kulcstól kulcsig), ezeknek az adatoknak van egy indexe. A fájlokba blokkokban egy rootfilter is van, ami adatkereséskor meg tudja mondani, hogy az adott rowkey ebbe a blokkban van-e vagy nem, ha nem, azonnal megy tovább. Így nem kell minden blokkot végig néznie.

50. HBase architecture. NameNode, ZooKeeper, Hmaster, Region server, Meta table, WAL, BlockCache, Memstore, Hfile, Region Flush, Minor compaction, Major compaction, Data replication, Region split, Recovery.

Name node: a HDFS fájlrendszerben van, egyfajta könyvtáros szerepet tölt be, tudja mi hol található.

ZooKeeper: egy olyan elosztott felügyeleti szolgáltatás, mely képes ellenőrizni, hogy a sok szerver él-e, elérhető-e. Vannak az ún. heartbeat csomagok, melyeket figyel. Amikor valakitől nem kap választ, tudja hogy van valami gond. Ő gondoskodik arról, hogy egy másik szerver átvegye a helyét annak, ami kiesett.

HMaster: a szerver, ami a karmester feladatát látja el, tudja a DDA (tábla létrehozás, módosítás, törlés) műveleteit, illetve ő végzi az adatok felosztását a szerverek között.

Region server: a régiók a táblákat horizontálisan osztják szét a régiószerverek között. Max. 1 gigásak, ők a slave-ek, adatokkal dolgoznak. A régió szerveren belül vannak régiók, melyek tartalmaznak column family darabokat. Az egyes column family-khez tartozik egy Memstore és HFiles.

Meta table: az összes régiószerver elhelyezkedését tartalmazza, ill. hogy melyik régiószerver milyen kulcsokat tartalmaz.

WAL (writeahead log): ide íródik be a kért módosítás (amit a kliens akar). Ennek az a szerepe hogy ha a szerver elszállna, mielőtt mentésre kerülne az adat, akkor belőle vissza lehessen állítani azokat az adatokat amik nem kerültek fájlszinten mentésre.

Block Cache: régiószerverszinten van. Gyakran olvasott dolgok vannak benne.

Memstore: a memóriában egy struktúra. Van neki egy korlátozott mérete, addig gyűjti a módosítási kéréseket. Ha eléri az előre meghatározott méretet, elmenti egy HFile-ba. Ez azért jó, mert a sokat, mostanában használt adatok a memóriában lesznek, nem kell fájlból kiolvasni.

HFile: ha a memstore megtelik, ilyen formában írja ki a HDFS-re a fájlokat.

Region split: egy régió ha megtelik, automatikusan szétvágja két alrégióra. Itt az HMaster feladata, hogy ezeket a régiókat ismét régiószerverekhez rendelje.

Data replication: klienstől jön az adat (írási kérés), beíródik a writer logba, ezután beíródik a memstore-ba. A primary datanode-on, egy másodlagos datanode-on és egy harmadlagos data node-on is eltárolásra kerül az adat. Kicsi az esély, hogy mindhárom helyen elvesszenek az adatok.

Recovery: ha egy régiószerver meghibásodik, a ZooKeeper értesíti az HMastert. Ő tud intézkedni a helyreállításról. Tehát azok az adatok helyreállítása a cél, amik a memóriában vannak még.

Minor compaction: fogja a fájlokat és egy fájlba gyűjti az egy column family-be tartozó adatokat.

Major compaction: automatikusan fogja a kisebb fájlokat és beleteszi a nagyobb fájlokat (helyet szabadít meg).

51. **Milyen típusú adatokat tárol? (BSON)**
52. **Mit jelent a BSON? Hogyan kapcsolódik ez a fogalom a Mongodb-hez?**

A BSON a JSON bináris formátuma. Valójában ez egy univerzális fájlformátum, sok nyelv támogatja, külön jó hogy az XML-el szemben emberi szemmel is könnyű olvasni és átlátható.

Több adattípusa van (JSON: 8-10, BSON: 23). Nagyobb fájl, mert plusz meta infókat tárol, de gyorsabb emiatt, így egy alkalmazás gyorsabban tud benne keresni. Max. 16Mb lehet.

53. **Soroljon fel 5 különböző adattípust Mongodb-ben!**

Null, Int, Long, String, Object, Array, Document, JavaScript, Binary data, Boolean, Date, timestamp, RegExp

54. **Soroljon fel 5 operátort!**

$eq, $gt, $gte, $in, $nin; $lte, $lt, $or, $nor, $and, $not, $all

55. **Sorolja fel az RDBMS szerinti database, table, row, column, index, join fogalmak megfelelőit MongoDB-ben!**

Database: kollekciók összessége.
Table: kollekció
Row: dokumentum
Column: mező
Index: nagyjából ugyanaz, mint az RDBMS-nél. Felpakolhatjuk a doksik oszlopaira, így gyorsabb lesz.
Join: embedded dokumentumot rakunk bele egy dokumentumba. Ez végtelen mélységig mehet így. Az 1-N kapcsolatok így működnek

56. **Jellemezze a MongoDB architektúra komponenseit!**

Legfelül van az **alkalmazási réteg**, amit kiszolgál.

Ez után van az **MQL réteg** van, ami kiszolgálja a CRUD operációkat. Ide csapódnak be a kliens/applikációs kérések.

Ezután a kliens letolja a **Document Data Model** rétegbe, az felel a kérések kiszolgálásáért és az adatstruktúra szervezéséért, indexek karbantartásával, foglalkozik a replikációs és sharding feladatokkal.

A **Storage layer**, alacsony szintű operációs szolgáltatások ill. a diszkkel való kommunikáció.

57. **Primary, Secondary, Arbiter, Hidden node, Router, Config server**

Primary-secondary architektúra van. Nincs külön node balancer, hanem minden kérés automatikusan a primeryhez fut be és ő adja tovább a megkapott kéréseket a különböző secondary nodeoknak. Aszinkron a kommunikáció, azaz mindig van egy kis idő, amíg átkerülnek a primaryról a secondary-ra.

Routing secondary node: nem kap adatokat a primary-től és nem fogad el, egy dolga van: biztosítani a többségi szavazatokat valamelyik secondary számára. Akkor nyerhet valaki, ha a szavazatok több mint felével rendelkezik.

Shardingnál van egy mongos modul, ami **router** feladatokat lát el és fogadja a kéréseket az applikációktól. Ez a **config szerver**hez megy, ami szintén egy mongo adatbázis, ő tárolja, hogy melyik shardon melyik szelet található. A shard nodeokon van egy ún. primary shard ami próbálja egyenlő szeletekre felosztani a darabokat.

58. Replica set, Shard

Replikálás: ha valami történik az egyik adatbázissal, ne legyen teljes leállás az adatbázisban, alkalmazásban, hanem azonnal átállhassunk másik adatbázisra és kiszolgálhassuk a kéréseket. Redundánsan tároljuk el.

Sharding: felosztjuk az adatbázist, azaz felosztjuk a diszk méreteket vagy hozzá rakunk új diszkeket és szétosztjuk az adatbázist. Különböző szerverekre fog futni az adatbázis tartalmának egy része. Tehát egy adatbázis több szerverre kerül. Ezeket lehet clusterekbe szervezni és megvalósulhat a high avaibility.

59. Melyik node-ra írunk, melyik node-ról olvasunk

0. szint: az applikáció betolja a mongonak az adatot, onnan magára hagyja. Nem kér visszaigazolást. Tárolás nem érdekli. Írási szempontból jó, gyorsan ki tudja írni. Ha adunk neki egy 2-es számot, akkor legalább 2 mp alatt ki kell írni, akkor igazolja vissza a primary nodenak.

Olvasásánál be lehet állítani szinteket itt is: hány node-on legyen ugyanaz a konzisztens adat. Pl.: read consens legyen 0, mert a primary-n lévő adatot elfogadom igaznak, de kérhetem hogy legalább 2 node-on vagy a secondary node-ok többségén ugyanazt akarom látni és végig kell menni az olvasási műveletnek és megnézni hogy melyek egyeznek már.

60. Mikor nem elérhető a rendszer?

Ha a clusterben több mint fele node kiesik, akkor mindenki átmegy seconderybe, mert ott nagy gond lehet és nem lehet az adatokat írni. Ilyenkor csak kézzel lehet beavatkozni.

61. Oplog, szavazás, shard key, write concern, read concern

Oplog: fizikailag egy collection. Maga a teljes adatbázis 5%-át állítják be rá. Tőle függ hogy meddig képes egy cluster túlélni. A primary beír az oplogba, ezeket tovább másolják a secondary-k és úgy végzik el. Az opog csak olyat utasítást tartalmaz, ami CRUD-nak megfelelő módosító tranzakciót tartamaz.

Szavazás: ha a primary node kiesik, valamelyik secondary észreveszi, választanak maguk közül egy új primary-t. A szavazás úgy megy hogy megnézik az oplogot (amibe tárolják a tranzakciókat). Kinek van a legfrissebb oplogja, és az lesz az új primary. Ez a node úgy is fog viselkedni. Ha visszatér a régi primary, nem veszi vissza a primary státuszát, secondaryként fog tovább viselkedni.

Write concern: 0. szint: az applikáció betolja a mongonak az adatot, onnan magára hagyja. Nem kér visszaigazolást. Tárolás nem érdekli. Írási szempontból jó, gyorsan ki tudja írni. Ha adunk neki egy 2-es számot, akkor legalább 2 mp alatt ki kell írni, akkor igazolja vissza a primary nodenak.

Read concern: olvasásánál be lehet állítani szinteket itt is: hány node-on legyen ugyanaz a konzisztens adat. Pl.: read consens legyen 0, mert a primary-n lévő adatot elfogadom igaznak, de kérhetem hogy legalább 2 node-on vagy a secondary node-ok többségén ugyanazt akarom látni és végig kell menni az olvasási műveletnek és megnézni hogy melyek egyeznek már.

Shard key: shardingnál a darabolás a shared key alapján történik. Mi is kiválaszthatjuk de a mongora is bízhatjuk. Nem lehet olyan típusú mező amely túl nagy darab. Pl.: egy oszlopba csak 3 kategória van, akkor az nem jó. Olyan kulcsot kell választani ami egyenlő részre felbontható és egyenlő disztribúció biztosítható.

62. Soroljon fel 3 példát olyan alkalmazásra, ahol érdemes lehet gráf adatbázist alkalmazni. Milyen típusú adatok esetén célszerű alkalmazni?

Tömegközlekedési hálózat, térképek, navigáció, IT infrastruktúra (eszközök kapcsolata), családfa. Akkor érdemes használni, ha az adatok közötti kapcsolat áll a központban.

63. Mondjon egy olyan példát, amit hatékonyabb gráf adatbázisban modellezni, mint relációsban

Ajánló rendszerek: ismerjük, hogy milyen vevőnek milyen tipikus kapcsolatmintái vannak termékekhez, így következtethetünk rá, hogy mások ilyen kapcsolatnál miket vásároltak még, így valószínűleg ő is vásárolhat ilyeneket, ha ajánljuk neki.

64. Native graph processing jelentése

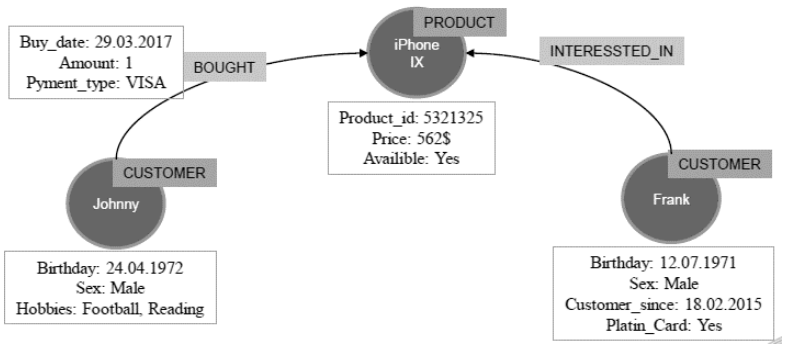
A gráf adatbázis minden csúcsnál eltárolja, hogy milyen más csúcsokkal van kapcsolata. Ez egy gyors fizikai kapcsolat a szomszédos adatokkal. Ha szeretném felolvasni a gráf egy részét van egy kiindulópont és onnan megyünk tovább a következő csúcsokra. Itt nem indexek vannak, hanem csúcsonként el van tárolva a referencia (pointer) a szomszédos csúcsokra.

65. Index free adjacency jelentése

Amikor felolvassa az adatokat csak a pointereket olvassa fel. Nem kell indexeket felolvasni meg táblákat. Ezt hívjuk „index-free adjacency”-nek. Egyik csúcsból megyünk a másikba, mert minden csúcs tudja, hogy kik a szomszédai. A relációs modellben úgy találhatjuk meg egy személy ismerőseit, ha indexelést használunk.

66. Labeled property graph adatmodell jellemzése

Ez a modell is a csúcsokból és élekből indul ki. Ami plusz, hogy a csúcsoknak és a kapcsolatoknak lehetnek tulajdonságaik (kulcs-érték párok). Egy csúcshoz tartozhat többféle címke is. Ezzel csoportosítani lehet a csúcsokat. Pl.: Béla nem csak alkalmazott lehet, hanem ügyfél is. Könnyen hozzáférhető, könnyen értelmezhető és sok mindent lehet vele modellezni. Nem kötött a séma, egy elemhez változó mennyiségű tulajdonságok kapcsolódhatnak. Pl.: Neo4J



67. RDF adatmodell jellemzése. RDF jelentése, RDF adatmodell, mint gráf

Dupla tárolók (RDF): ez egy adatcsere formátum. Anno a weblapok inkább statikusok voltak. Az RDF-nek is hasonló célja volt, mint az XML-nek, azaz az adatok automatikus feldolgozása. Három komponensből áll, mint egy egyszerű angol mondat:

- SUBJECT: Johnny, egy forrás, egy weblap (node)
- PREDICATE: mit szeretnék mondani a tárgyról (csúcs, él)
- OBJECT: egy másik erőforrás vagy egy literált pl.: Johnny's ID 1234 (node)
- Saját lekérdezőnyelve: SPARQL

68. Soroljon fel gráf típusokat

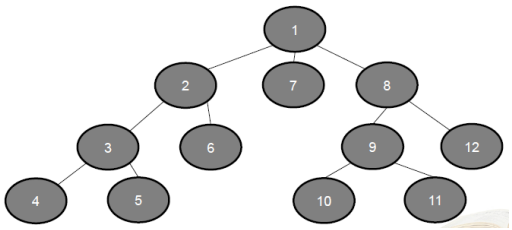
Irányított vagy irányítatlan (van iránya vagy nincs az éleknek)
Ciklikus vagy aciklikus (kiköthetünk-e a kiindulópontban vagy nem)
Súlyozott vagy súlyozatlan (az éleknek van-e értékük)
Sűrű vagy ritka (egy csúcsnak egy vagy több kapcsolata van)

69. Gráf bejárás jelentése. 2 alapvető gráf bejáró algoritmus

A gráf bejárása: egy gráfban elkezdek egy adott csúcstól felkutatni szomszédos csúcsokat. A pointerek alapján könnyen bejárhatók. Két alapvető stratégia:

Szélességi keresés: megnézem az összes szomszéd csúcsot, azaz az 1-es kit ismer. Ezután visszamegyek a 2-eshez és tovább megyek a 2-es szomszédaira: 3-6, utána a 3-as szomszédai és így tovább.

Mélységi keresés: az 1-esből kiindulok és keresünk egy célt. Végigmegyek egy sávon a legmélyebb pontig. Amikor eljutok, visszamegyek és megint mélységbe keresek amíg egy zsákutcához nem érek.



70. Gráf adatmodell tervezésének szempontjai

Whiteboard model. Egy gráfot könnyen fel lehet vázolni egy papírra, whiteboard-ra. Három alapelem: csúcs, él és esetleg a tulajdonságok. Ha ezt le akarjuk fejleszteni a Neo4J-be, nem kell másik modellre átalakítanunk, hanem ahogy megterveztük, úgy tudjuk létrehozni az adatbázisban. Ha ez megvan, megbeszéljük az analistával hogy számára megfelelő-e a modell. Ha nem, akkor bővíteni kell:

- egy személy milyen szerepben lehet a rendszerben
- végig kell gondolni hogy egy csúcsnak megvan-e minden tulajdonsága
- kapcsolatok elnevezése, kapcsolatok iránya, darabszáma, plusz tulajdonságok

71. **Relációs modell konvertálása labeled property graph modellbe**

Tábla: csúcsok a tábla nevével
Sor: egy csúcs az egyed címkéjével
Oszlop: a cúcs tulajdonsága
PK, FK kulcsok: élek
N:M kapcsolatok: élek tulajdonságai

72. **Mit tud a Neo4j native graph storage-ról?**

A gráf adatbázis minden csúcsnál eltárolja, hogy milyen más csúcsokkal van kapcsolata. Ez egy gyors fizikai kapcsolat a szomszédos adatokkal. Ha szeretném felolvasni a gráf egy részét van egy kiindulópont és onnan megyünk tovább a következő csúcsokra. Itt nem indexek vannak, hanem csúcsonként el van tárolva a referencia (pointer) a szomszédos csúcsokra.

73. **Van-e Neo4j-ben sharding?**

Nincs

74. **Mit jelent a replication Neo4j-ben?**
75. **Mit tud a Clustering architecture-ről Neo4j esetében?**

Két funkciója van a clusternek:

- **Klasszikus:** A gráf részeit nem lehet különböző szerverekre elhelyezni. Itt csak replikáció van, azaz minden egyes szerverre rápakoljuk a komplett adatbázist. Egyrészt így létrehozható egy HA rendszer, azaz ha kiesnek gépek, elérhető az applikáció, az adatbázis és működnek a lekérdezések. Másrészt ha nagyon sok lekérdezés, erőforrásigény van, nem csak egy gépről tudnak olvasni hanem akár 100-ról is, így az erőforráseloszlás biztosítva van.
- **Specifikus:** a központban van 3-5 core szerver, mindegyik OLTP-s tranzakció itt fog végrehajtódni. Amelyik szerverhez megérkezik a tranzakció, megpróbálja a core szerverek többségére ráírni, ha a többség visszaigazolja, hogy megérkezett a tranzakció (update), akkor igazolja vissza az eredetileg megcélzott szerver a kliensnek, hogy végrehajtódott a tranzakció. Ez addig megy, amíg van elég core szerver. Ha van 3 és 1 elhal belőle, akkor kritikus a dolog, hisz átvált readonly módba a cluster. Elérhető de csak olvasásra.

76. **Cypher nyelv jellemzése (diák alapján): főbb parancsok (ismerjen néhányat fejből), megszorítások, indexek**

Elsősorban a Neo4J-re készült. A Cypher is egy leírónyelv, mint az SQL. ASCII Art logikára épül, azaz az utasítás egyfajta rajz a csúcsokról és kapcsolatokról. Létre lehet vele hozni, módosítani, törölni, felolvasni objektumokat.

Pl.: Az a ember kedveli b embert: (a:person)-[:likes]->(b:person)