NoSQL

NoSQL

- Két fő ok miatt születtek meg:
  - A nagy adattömeg kezelése kikényszerítette azt, hogy klaszterezéssel kapcsoljanak össze gépeket, és így nagy hardver platformokat építsenek.
  - Ez az igény azt is megnehezítette, hogy az alkalmazáskódok jól működjenek a relációs modellel.

NoSQL

A két ok miatt használunk NoSQL adatbázist:

- Az alkalmazásfejlesztés termelékenysége miatt: sok idő és erőfeszítés megy el arra, hogy a memóriastruktúrákat leképezzék relációs adatbázisra. A NoSQL adatbázisok olyan adatmodellt biztosítanak, amelyek jobban alkalmazkodnak az alkalmazások szükségleteihez. Leegyszerűsödik az adatbázis elérése, kevesebb kódot kell írni, nyomkövetni és javítani.
- A nagymennyiségű adat miatt: ma megéri több adatot tárolni és sokkal gyorsabban feldolgozni. Relációs adatbázissal ez drága vagy lehetetlen. A relációs adatbázisokat úgy tervezték, hogy egy gépen fussanak, azonban ma már általában gazdaságosabb sok kisebb, olcsóbb gép klaszterén végezni a nagymennyiségű adat feldolgozását. Sok NoSQL adatbázist úgy terveztek, hogy klasztereken futnak.

NoSQL

- NoSQL adatbázisok listája:
  - nosql-database.org
  - nosql.mypopescu.com/kb/nosql
  - https://db-engines.com/en/ranking
- A fő kategóriák:
  - Kulcs-érték (key-value) adatbázisok
  - Dokumentum adatbázisok
  - Oszlopcsalád (column-family) adatbázisok
  - Gráf adatbázisok
  - Objektum adatbázisok
  - XML adatbázisok
- Vannak nem tiszta adatbázisok is, amelyek kevernek két kategóriát.

NoSQL

- A NoSQL adatbázis a következőt jelenti:
  - többnyire open-source
  - többnyire a 21. század elején fejlesztették
  - többnyire nem használ SQL-t.

Aggregátum adatmodellek

- A kulcs-érték, a dokumentum és az oszlopcsalád adatmodelleket együtt aggregátum adatmodelleknek hívjuk, a közös jellemzőik miatt.

Aggregátum adatmodellek

- Azt támogatja, hogy gyakran akarunk olyan adatokon dolgozni, amelyeket olyan unitokba akarunk szervezni, amelynek bonyolultabb a struktúrája, mint amelyet a listák és a rekordstruktúrák egymásba ágyazása lehetővé tenne.

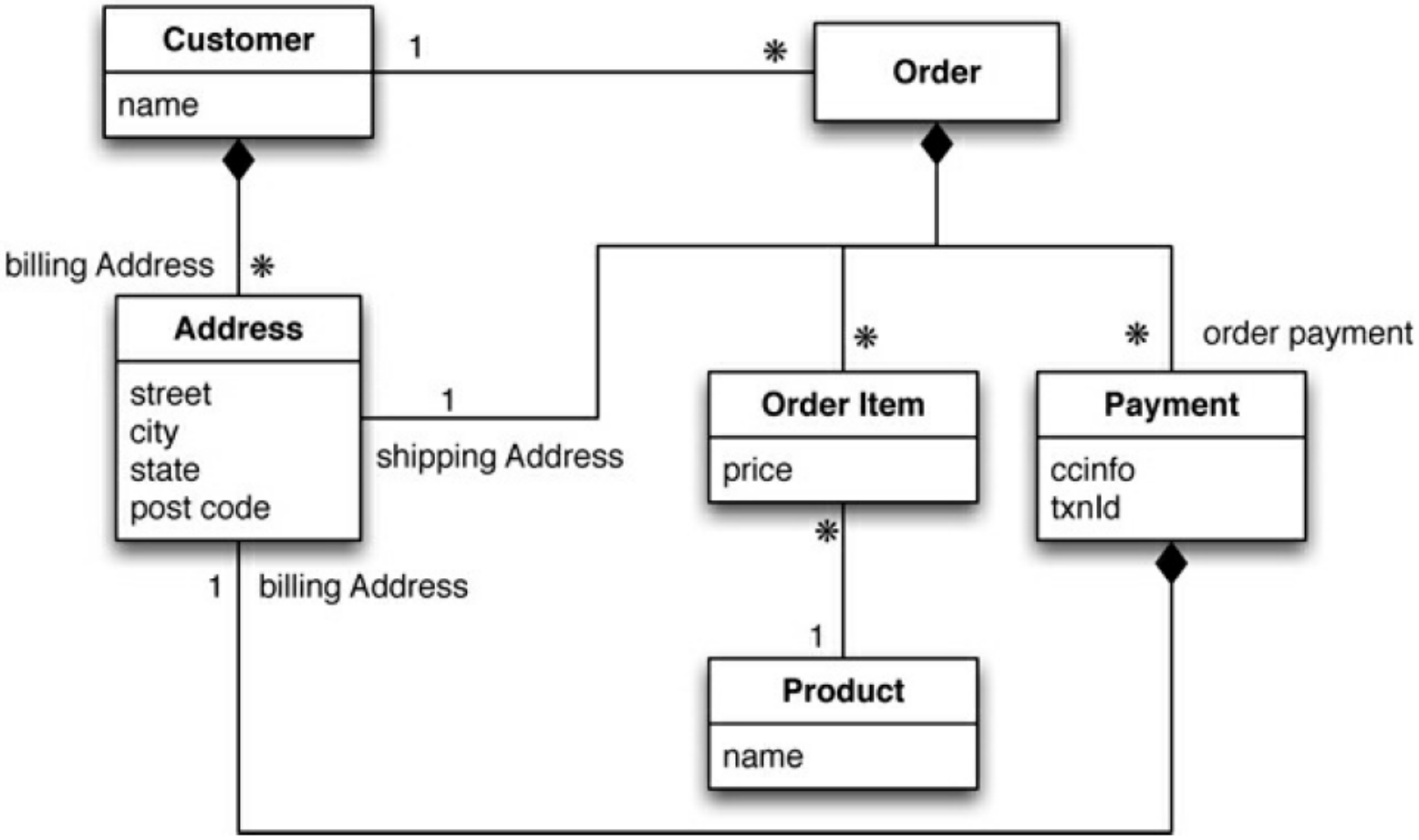
Aggregátum adatmodellek

- Az **aggregátum** kapcsolódó objektumok egy olyan gyűjteménye, amelyeket egy egységként szeretnénk kezelni. Ez egyben az adatmódosítás és a konzisztencia menedzsment egysége. Általában atomi műveletekkel szeretjük módosítani az aggregátumokat és aggregátumban kifejezve szeretünk kommunikálni az adattárolóval.

Aggregátum adatmodellek

- Az aggregátumok megkönnyítik az adatbázisoknak a klaszteren való műveletek kezelését, mert az aggregátum a replikáció és a sharding természetes egysége.
- Az aggregátumok megkönnyítik a programozók dolgát is, mert az gyakran az aggregátumokon keresztül módosítják.

Aggregátum példa



Aggregátum példa

In JSON format

```
// in customers
{ "id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
```

Aggregátum példa

```
// in orders
{ "id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
  }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
  "ccinfo":"1000-1000-1000-1000",
  "txnId":"abelif879rft",
  "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Aggregátum példa

- Egy egyszerű logikai cím rekord 3-szor jelenik meg a példában. Az ID-k használata helyett a címet egy értékként kezeli, és bemásolja mindenhova. Ez amiatt történhet, hogy nem akarjuk sem a shipping address, sem a payment’s billing address értékeket módosítani. Az aggregátumokkal az egész címstruktúrát bemásolhatjuk egy aggregátumba, ha szükséges.

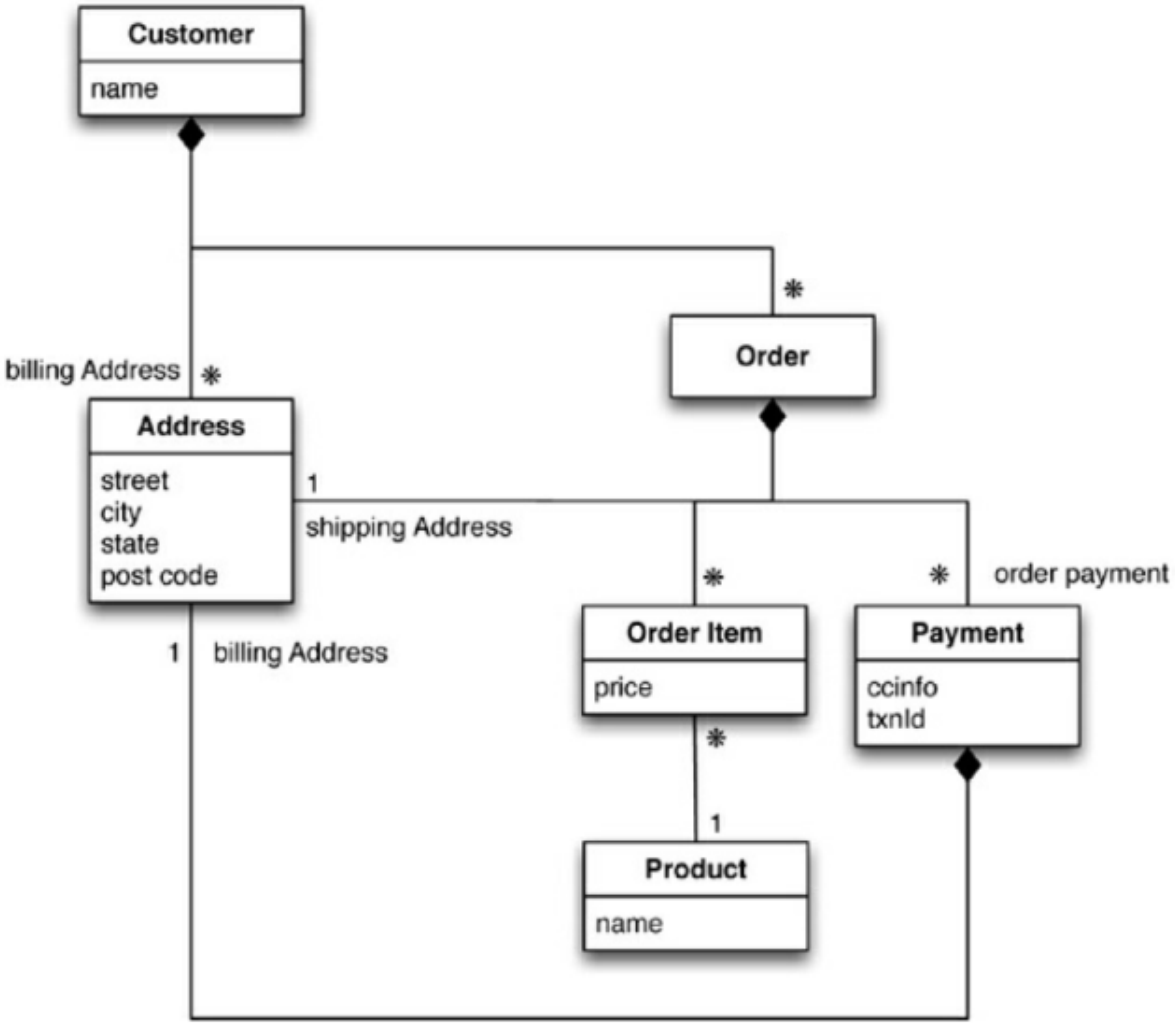
Aggregátum példa

- A customer és az order közötti kapcsolat az orders aggregátumon belül valósítja meg az aggregátumok közötti kapcsolatot.
- A product name az order részeként szerepel. Ez a fajta denormalizáció gyakori az aggregátumok világában, mert csökkenteni akarjuk azon aggregátumok számát, amelyeket egy adatkölcsönhatásban érünk el.

Aggregátum példa

- Nincs igazán meghatározva, hogyan húzzuk meg az aggregátumok határát (**aggregate boundary)**. Úgy tudjuk eldönteni, hogy hol legyen a határ, hogy megvizsgáljuk, hogy az adatokat milyen módon fogják használni az alkalmazások. Emiatt az aggregátumhatárokat másképp is meghúzhatjuk. A példánkban minden orders-t beletehetünk a customer aggregátumba.

Aggregátum példa



Aggregátum példa

```
// in customers
{"customer": {"id": 1,
              "name": "Martin",
              "billingAddress": [{"city": "Chicago"}],
              "orders":
                    [{ "id":99,
                       "customerId":1,
                       "orderItems":[{"productId":27,
                                      "price": 32.45,
                                "productName": "NoSQL Distilled"
                                     }],
                       "shippingAddress":[{"city":"Chicago"}]
                       "orderPayment":[
                         {"ccinfo":"1000-1000-1000-1000",
                          "txnId":"abelif879rft",
                          "billingAddress": {"city": "Chicago"}
                          }],
                     }]
}}
```

Aggregátum példa

- Nincs univerzális válasz arra a kérdésre, hogy hol húzzuk meg az aggregátumok határát. Csak attól függ, hogy hogyan fogjuk módosítani az adatot.
- Ha a customer-t az orders-sel együtt fogjuk módosítani, akkor egy aggregátummal fogunk dolgozni.
- Ha módosításkor egy megrendelésre fókuszálunk, akkor érdemes szétválasztani az aggregátumokat.
- Gyakran előfordul, hogy egy alkalmazás mindkét megközelítést is használná.

Aggregátum adatmodellek

- Előnyei
  - Nagyszerűen segít a klaszteren való futtatást, ahol minimalizálnunk kell az adatok lekérdezéshez szükséges csomópontok számát. Az aggregátumokkal egyértelműen megmondjuk az adatbázisnak, hogy mely adatokat fogjuk együtt módosítani. Ezeknek az adatoknak egy csomóponton kell lenniük.
- Hátrányai
  - Gyakran nehéz meghúzni a határt az aggregátumok között, különösen akkor, amikor ugyanazt az adatot több különböző esetben használjuk.
  - Az aggregátum struktúra sok típusú adatlekérdezésben / módosításban segíthet, azonban más típusú adatlekérdezésben / módosításban gátolhat.

Aggregátum adatmodellek

- Gyakran azt mondják, hogy a NoSQL adatbázisok nem támogatják az ACID tranzakciókat, és így feláldozza a konzisztenciát.
- Általában igaz, hogy az aggregátumorientált adatbázisoknak nincs olyan ACID tranzakciójuk, amely több aggregátumra kiterjed.
- Helyette **egy aggregátum egyetlen atomi módosítását** támogatják. Ez azt jelenti, hogy ha több aggregátumot szeretnénk atomi módon módosítani, azt nekünk az alkalmazáskódunkban kell kezelni.

Kulcs- érték és dokumentum adatmodellek

- Mindkét típusú adatbázis sok aggregátumból épül fel, ahol minden aggregátumnak van egy kulcsa vagy egy azonosítója, amelyet az adat elérésére használunk.

Kulcs- érték és dokumentum adatmodellek

Kulcs-érték adatbázisok:

- Az aggregátum **átlátszatlan** az adatbázisban. Az átlátszatlanság előnye az, hogy azt tárolhatunk az aggregátumban amit csak szeretnénk. Az adatbázisnak lehet, hogy van valamilyen méretkorlátozása, de ez több, mint ami a korlátozná a szabadságunkat.
- Egy aggregátumot csak a kulcsán keresztül kereshetünk ki.

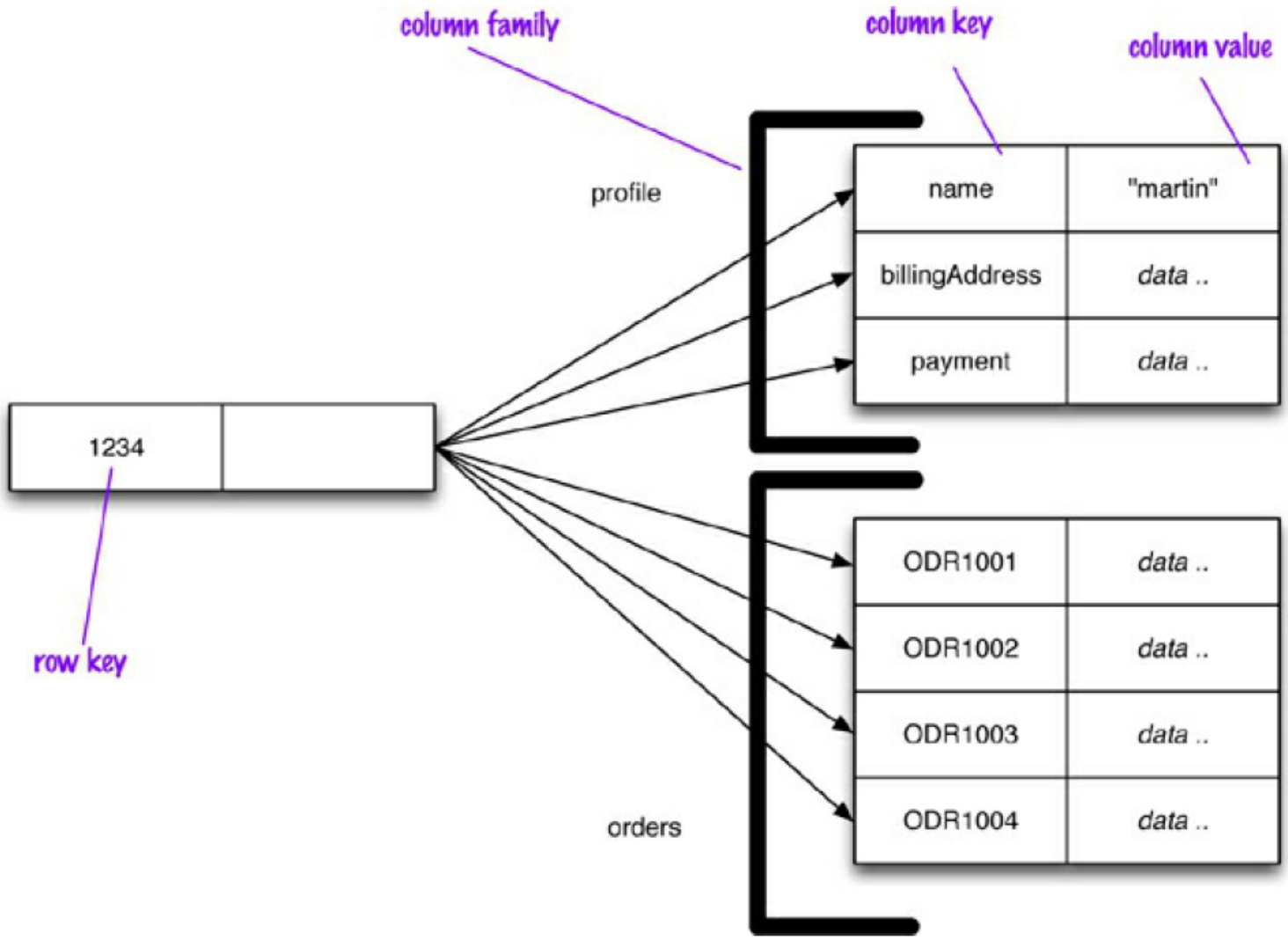
Dokumentum adatbázisok

- Képes látni az aggregátum **struktúráját**. A dokumentum adatbázis korlátozásokat adhat arra nézve, hogy milyen struktúrákat és típusokat helyezhetünk el egy aggregátumba. Cserébe az elérésnél nagyobb rugalmasságot biztosít.
- Lekérdezéseket írhatunk az aggregátum mezői alapján, kinyerhetjük az aggregátumok részeit a teljes aggregátum helyett, és indexeket hozhatunk létre az aggregátum tartalma alapján.

Oszlopcsalád tárak

- A legtöbb (relációs) adatbázisnak a tárolási alapegysége a sor. Ez az írási teljesítményt támogatja. Azonban sok olyan forgatókönyv van, ahol az írás kevés, de gyakori az olyan lekérdezés, ahol sok sornak néhány oszlopát kérjük le. Az ilyen esetekben a legjobb megoldás az, ha **a sorokhoz oszlopok csoportjait tároljuk**, és ezek a csoportok lesznek a tárolás egységei. Emiatt hívják őket oszloptáraknak.
- A legkönnyebb talán úgy megérteni az oszlopcsalád modellt, mint egy **kétszintű aggregátumstruktúrát**. Úgy, mint a kulcs-érték táraknál, az első kulcsot gyakran a sor azonosítójaként értelmezik, amellyel a keresett aggregátumot meg lehet fogni. A különbség az, hogy az oszlopcsalád táraknál a soraggregátum önmagában részletesebb értékek egy leképezését formázza. Ezeket a második szintű értékeket hívjuk oszlopoknak.

Oszlopcsalád tárak



Oszlopcsalád tárak

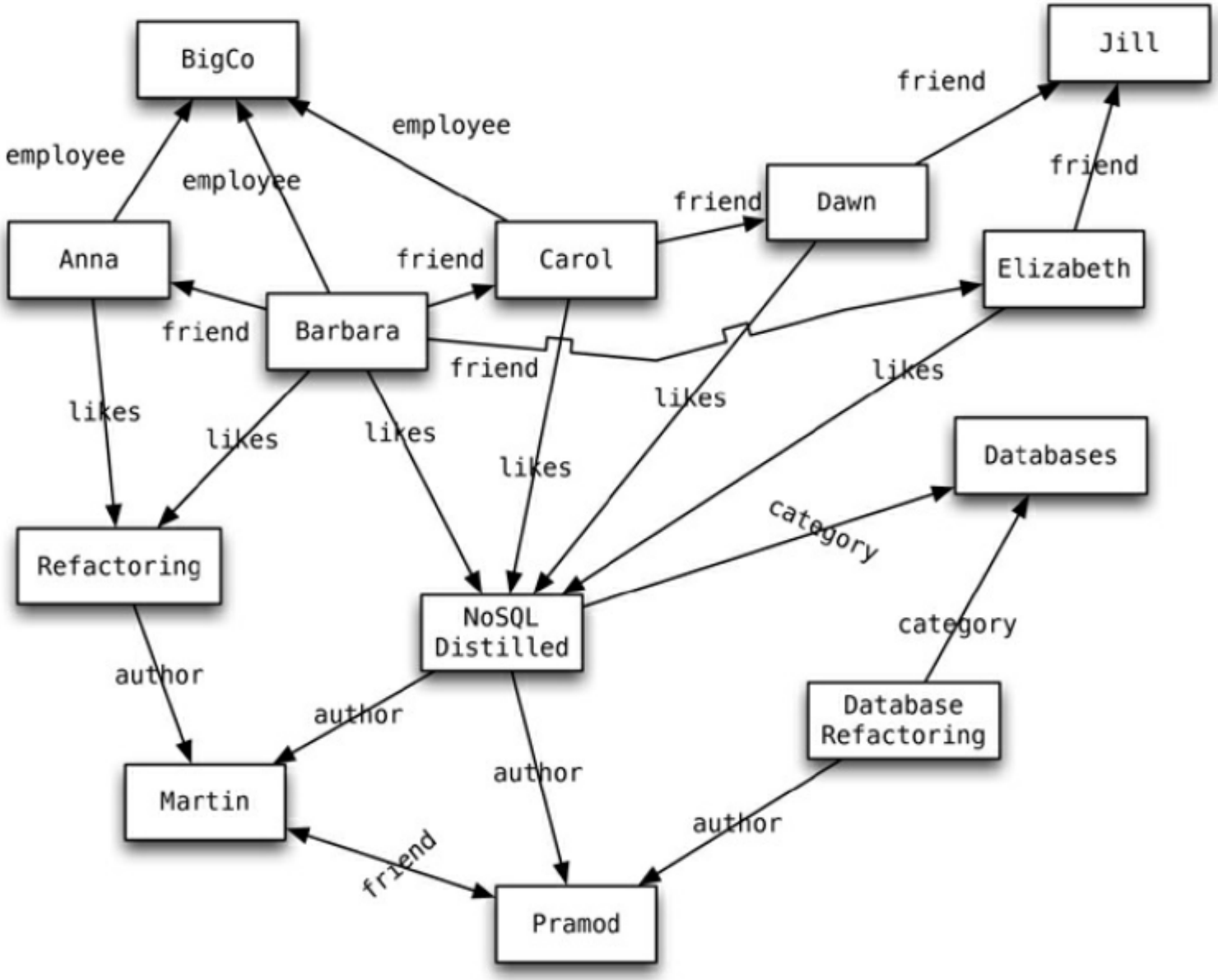
- Az oszlopcsalád adatbázisok az oszlopaikat **oszlopcsalád**okba szervezik. Minden oszlopnak egy oszlopcsalád részének kell lennie. Az oszlop az elérés egysége, azonban feltételezzük, hogy egy oszlopcsaládot általában együtt érjük el.
- Az adatokat kétféleképp strukturálhatjuk:
  - **Sororientáltan:** minden sor egy aggregátum (pl. customer, amelynek az ID-ja 1234), amelynek az oszlopcsaládjai hasznos adatcsonkokat tartalmaz az aggregátumon belül (profile, order history).
  - **Oszloporientáltan**: minden oszlopcsalád egy rekordtípust definiál (pl. customer profiles), ahol minden rekordhoz több sor tartozik. Így minden oszlopcsaládban úgy gondolhatsz egy sorra, mint rekordok összekapcsolása.

Oszlopcsalád tárak

- **Bármilyen oszlophoz bármilyen sort hozzá lehet adni**, így a soroknak különböző oszlopkulcsai lehetnek. Új oszlopokat adhatunk a sorokhoz a szokásos adatbázis használat alatt. Azonban az új oszlopcsaládok definiálása ritka, és ehhez gyakran le kell állítani az adatbázist.
- A **sovány sorok**nak **(Skinny rows)** kevés oszlopuk van, és sok különböző sorban ugyanazokat az oszlopokat használják. Ebben az esetben az oszlopcsalád egy rekordtípust definiál, minden sor egy rekord, és minden oszlop egy mező.
- A **széles sor**nak **(wide row)** has sok oszlopa van (akár több ezer), és a soroknak nagyon különböző oszlopai vannak. Ebben az esetben a széles oszlopcsalád egy listát modellez, ahol minden oszlop egy elem a listában.

Gráf adatbázisok

- A gráf adatbázisokat az olyan kis rekordok motiválták, amelyek között bonyolult kapcsolatok állnak fenn.
- Ebben az értelemben a gráf egy gráfadatstruktúra, élekkel (edge) összekapcsolt csomópontok (node).
- Ebben a struktúrában olyan kérdéseket tehetünk fel, mint „keressük azt a könyvet az Adatbázis kategóriában, amelyet olyan valaki írt, akit valamelyik barátom kedvel”.
- Ideális megoldás olyan esetben, amikor összetett kapcsolatokat tartalmazó adatokat szeretnénk tárolni, mint a szociális hálók (social networks), vagy termékajánlások.


BigCo
Jill
friend
employee
employee
employee
friend
Dawn
friend
Anna
friend
Carol
Elizabeth
friend
Barbara
friend
likes
likes
likes
likes
likes
likes
Databases
category
Refactoring
NoSQL
Distilled
category
author
author
Database
Refactoring
author
Martin
author
friend
Pramod

Gráf adatbázisok

- A relációs adatbázisok a kapcsolatot külső kulcs segítségével valósítják meg. Az összekapcsolások, amelyek a navigációhoz szükségesek meglehetősen drágák lehetnek, amely azt jelenti, hogy a teljesítmény gyakran gyenge a sok kapcsolattal rendelkező adatmodellekben.
- A gráfadatbázisok navigálása a kapcsolatokon keresztül nagyon olcsó. A gráfadatbázisok a navigálás munkájának a nagyrészét beszúráskor végzi el és nem a lekérdezéskor. Ez természetesen akkor kifizetődő, amikor a lekérdezés teljesítménye sokkal fontosabb, mint a beszúrásé.
- Ezek az adatbázisok főleg egy gépen futnak és nem klasztereken elosztva.

NoSQL adatbázisok

- Sémamentes adatbázisok:
  - A kulcs-érték tár lehetővé teszi, hogy bármilyen adatot tároljunk egy kulcs alatt.
  - A dokumentum adatbázis hatékonyan végzi ugyanezt, mivel a tárolt dokumentum struktúrájára nincs megszorítás.
  - Az oszlopcsalád adatbázisok lehetővé teszik, hogy bármilyen adatot bármelyik oszlop alatt tároljunk.
  - A gráfadatbázisok lehetővé teszi, hogy szabadon adjunk éleket az adatbázishoz, és tulajdonságokat az élekhez és a csomópontokhoz.

NoSQL adatbázisok

- Sémamentes adatbázisok:
  - Egyszerűen azt tárolhatjuk, amire szükségünk van.
  - Lehetővé teszik, hogy egyszerűen **módosítsuk** az adattárunkat a projektünk előrehaladtával. Ha új dolgot fedezünk fel, egyszerűen hozzáadhatjuk.
  - Ha rájövünk, hogy valamit nem szükséges tárolnunk, akkor egyszerűen **törölhetjük**.
  - Egyszerűvé teszik, hogy **nemegységes** adatokkal (olyan adatokkal, amelyeknél a rekordok más mezőkből épülnek fel) dolgozzuk. Lehetővé teszik, hogy a rekordok pontosan azt tartalmazzák, amire szükségünk van, nem többet és nem kevesebbet.

NoSQL adatbázisok

- Sémamentes adatbázisok:
  - A probléma: tény, hogy amikor adatokat használó programot írunk, akkor a program majdnem mindig bízik az implicit séma valamilyen formájában.
  - Attól, hogy az adatbázisunk sémamentes, általában van egy implicit séma. Ez az implicit séma olyan feltételezések halmaza, amelyek az adatokat manipuláló kódban lévő adatstruktúráról szólnak.
  - Az alkalmazáskódban szereplő implicit séma eredményez néhány problémát. Ha meg akarjuk érteni, hogy mit jelent az adat, akkor az alkalmazáskódunkban kell ásni.

Elosztási modellek

- A NoSQL adatbázisok elsődleges érdekeltsége az a képesség volt, hogy az adatbázis egy nagy klaszteren fusson.
- Az aggregátum megközelítés jól illeszkedik a skálázáshoz, mert az aggregátum az elosztáshoz használt természetes egység.

Elosztási modellek

- Két út van az adatok elosztásához:
  - Replikáció (A replikáció ugyanazt az adatot több csomópontra helyezi.)
  - sharding (A sharding különböző adatokat helyez a csomópontokra.)
- A replikáció és a sharding ortogonális technikák: lehet használni az egyiket vagy akár mindkettőt.
- A replikáció két formában jelenhet meg: master-slave és peer-to-peer.

Elosztási modellek

- Egyedülálló szerver
  - A legegyszerűbb elosztási modell
  - nincs elosztás
  - szeretjük, mert kiküszöböli az összes komplexitást, amellyel a többi modell megvalósítás jár
  - A gráf adatbázisok tartoznak ebbe a kategóriába, mivel legjobban az egy szerveres konfigurációban működnek.
  - Ha az adathasználatunk leginkább aggregátumok feldolgozásáról szól, akkor egy egyszerveres dokumentum vagy kulcs-érték tár jó választás lehet, mert egyszerűbbek a fejleszlesztőknek.

Elosztási modellek

- Sharding
  - A sharding különböző adatokat különböző csomópontokra tesznek, minden csomópont saját maga végzi az írást és az olvasást.
  - Az aggregátumokat úgy tervezzük, hogy olyan adatokat tartalmazzanak, amelyeket általában együtt érünk el, így az aggregátum lesz az elosztás nyilvánvaló egysége.

Elosztási modellek

- Sharding
  - Néhány tényező javíthatja a teljesítményt:
    - Az adatokat ahhoz közel helyezzük el, ahol használni fogják.
    - Próbáljuk meg úgy elhelyezni az aggregátumokat, hogy egyenlően legyenek elosztva a csomópontokat, amelyek mindegyike egyenlő terhelést kap.
    - Néhány esetben hasznos, ha az aggregátumokat együtt helyezzük el, ha úgy gondoljuk, hogy sorban fogjuk felolvasni őket (A Bigtable azt javasolja, hogy a sorokat abc sorrendben tároljuk, és a webcímeket a megfordított domain neveik alapján rendezzük (pl: com.martinfowler). Így több oldal adatát érhetjük el egyszerre, amivel javítjuk a feldolgozás hatékonyságát.)
    - Sok NoSQL adatbázis ajánl **auto-sharding**-ot, ahol az adatbázis vállalja a felelősségét annak, hogy kiutalja a shard-ot az adathoz és biztosítsa, hogy az adat a megfelelő shard-ra kerül.

Elosztási modellek

- Sharding
  - A sharding különösen értékes a teljesítmény tekintetében, mert az írási és az olvasási teljesítményt is növelheti. A sharding horizontálisan skálázható írást biztosít.

Elosztási modellek

- Master-Slave replikáció
  - Több csomópontra másoljuk az adatot.
  - Egy csomópontot kijelölünk **master** vagy elsődleges csomópontnak. A master a hiteles forrása az adatoknak és általában ő a felelős az adatok **módosítás**áért. A többi csomópont **slave** vagy másodlagos csomópont. A replikációs folyamat **szinkronizálja** a slave-eket a master-rel.
  - Master-slave replikáció leginkább akkor hasznos, amikor **olvasás intenzív** adathalmazt skálázunk. Több olvasási kérés kiszolgálásához horizontálisan skálázhatunk, ha **több slave csomópontot adunk** a rendszerhez és biztosítjuk, hogy minden olvasási kérés a slave-ekhez fusson be.
  - Nem jó választás, ha az adathalmazunkon sok írási forgalom történik. Korlátozva vagyunk a master írási képességével és azzal a képességgel, hogy továbbadja a módosításokat.

Elosztási modellek

- Master-Slave replikáció
  - A másik előnye az **olvasási rugalmasság**: ha a master meghibásodik, a slave-k még mindig tudják kezelni az olvasási kéréseket. Ez akkor hasznos, ha a legtöbb adatelérés olvasás. A **master meghibásodása** lehetetlenné teszi az írások kezelését addig, amíg a master helyre nem áll vagy egy másik mastert ki nem jelölünk. Mivel a slave-k a master másolatai, a master meghibásodása utáni **helyreállás** felgyorsul, mivel egy slave-t gyorsan ki lehet jelölni új masternek.
  - A mastert ki lehet jelölni kézzel és automatikusan.
    - A kézzel való kijelölés tipikusan azt jelenti, hogy amikor konfiguráljuk a klasztert, akkor egy csomópontot masternek konfigurálunk.
    - Az automatikus kijelölésnél létrehozzuk a csomópontok klaszterét és ők választanak ki egyet maguk közül masternek.

Elosztási modellek

- Master-Slave replikáció
  - A hátránya az inkonzisztencia.
    - Az a veszély, hogy különböző kliensek különböző slave-ket olvasva, **különböző értékeket** fognak látni, mivel nem minden változás adódik tovább azonnal a slave-knek. Legrosszabb esetben egy kliens nem tud olvasni egy írást, amit most készült.
    - Ha master-slave replikációt használunk **forró mentés**hez, ez az eset akkor is fennáll, mivel ha a master meghibásodik, minden olyan módosítás, amelyet nem továbbított a mentésnek, elveszik.

Elosztási modellek

- Peer-to-Peer replikáció
  - Minden csomópont fogadhat írást és olvasást minden adatra.
  - Minden csomópontnak egyenlő súlya van, mindenki fogadhat írást, és ha valamelyiküket elvesztjük, az nem akadályozza meg az adattár elérését.
  - A meghibásodásokat könnyen áthidalhatjuk úgy, hogy az adatokhoz végig hozzáférünk.
  - Könnyen adhatunk hozzá új csomópontot, hogy javítsuk a teljesítményt.
  - A legnagyobb bonyodalom a konzisztencia. Ha két különböző helyet írhatunk, akkor megkockáztatjuk azt, hogy két felhasználó ugyanazt a rekordot ugyanabban az időben próbálja módosítani, amely írás-írás konfliktust okoz. Az olvasási inkonzisztencia is vezethet problémákhoz, de azok legalább relatívan átmenetiek.
  - Konzisztenciamegoldások: később

Elosztási modellek

- A sharding és a replikáció kombinációja
  - Ha master-slave replikációt és shardingot használunk egyszerre, az azt jelenti, hogy **több master**unk van, de egy adatelem csak egy masteren van. Konfigurációtól függően kiválaszthatsz egy csomópontot masternek és a többit slave-nek vagy választhatsz több mastert és több slavet.
  - **Peer-to-peer replikáció és sharding** használata az oszlopcsalád adatbázisoknál szokásos stratégia.
  - Jó kiindulási pont a peer-to-peer replikációkhoz, ha a replikációs faktor 3, így minden shard 3 csomóponton jelenik meg. Ha egy csomópont meghibásodik, akkor a shard-jai a többi csomóponton felépülnek.

Konzisztencia

- **Módosítási konzisztencia**
  - **Írás-írás konfliktus**: két ember ugyanabban az időpontban ugyanazt az adatelemet módosítja.
  - Amikor az írás eléri a szervert, akkor a szerver sorbaállítja (**serialize**) őket, azaz eldönti, hogy melyiket hajtja végre elsőként majd másodikként.
  - **Elveszett módosítás:** Martin módosítása után Péter azonnal felülírja az adatelemet, akkor a Martin módosítása elveszett. Konzisztenciahibának látjuk, mert Péter módosítása azon az állapoton hajtódott végre, amire Martin módosított, bár azt várjuk, hogy az eredeti állapoton hajtódjon végre.

Konzisztencia

- **Módosítási konzisztencia**
  - A konzisztencia kezelését kétféleképp szokták megközelíteni: pesszimista kezelés és optimista kezelés.
  - A **pesszimista** megközelítésben megelőzzük a konfliktusokat; az **optimista** megközelítésben megengedjük a konfliktusokat, de észrevesszük őket és intézkedéseket teszünk az eltávolításukra.
  - Az írási konfliktusokhoz a szokásos **pesszimista** megközelítés az **írási zárak** elhelyezése, azaz ha egy értéket meg akarunk változtatni, előtte zárat kell elhelyezni rajta. A rendszer biztosítja, hogy egyszerre csak kliens tehet zárat egy adatelemre. Ha Martin és Péter zárat kérne, akkor csak Martin (az első) járna sikerrel. Péter így látná Martin írását, mielőtt eldöntené, hogy elvégzi-e a módosítását.

Konzisztencia

- **Módosítási konzisztencia**
  - A szokásos **optimista** megközelítés a **feltételes módosítás**, ahol bármely kliens (amelyik módosítást végez) a közvetlenül a módosítás előtt leteszteli, hogy a módosítandó érték változott-e az utolsó olvasása óta. Ebben az esetben Martin módosítása sikeres lenne, Péteré meghiúsulna. A hibából Péter tudná, hogy újra meg kell néznie az értéket és eldönteni, hogy akar-e módosítani.
  - Egy másik **optimista** megközelítés az írás-írás konfliktus kezelésére, amikor **elmentjük mindkét módosítást** és rögzítjük, hogy konfliktusban állnak. Ez ismerős lehet a verziókezelő rendszerekből. A következő lépésben a két módosítást valahogyan **össze** kell **olvasztani** (kézzel vagy automatikusan).

Konzisztencia

- **Módosítási konzisztencia**
  - A párhuzamos programozás alapvető velejárója, hogy mérlegelni kell a biztonság (a hibák, mint a módosítási konfliktusok kerülése) és a gyorsaság (gyorsan válaszoljunk a kliensnek) között.

Konzisztencia

- **Olvasási konzisztencia**
  - **Inkonzisztens olvasás** vagy **olvasási-írási konfliktus:** Képzeljük el, hogy van egy megrendelésünk, amelyek több tétel van, és szállítási költség tartozik hozzá, amelyet a tételekre vonatkozóan számolnak ki. Ha a megrendeléshez hozzáadunk egy új tételt, akkor újra kell számolni és módosítani kell a szállítási költséget. Egy relációs adatbázisban a szállítási költség és a rendelések külön táblában lennének. Az inkonzisztencia úgy állhat elő, hogy Martin **ad egy új sort** a megrendeléséhez, majd Péter **olvassa a tételeket és a szállítási költséget**, végül Martin **módosítja** a szállítási költséget.

Konzisztencia

- **Olvasási konzisztencia**
  - **Logikai konzisztencia**: biztosítja, hogy a különböző adatelemeknek **együtt van értelmük**. A logikai inkonzisztencia elkerülését a relációs adatbázis a tranzakciókkal végzi. Ha Martin két módosítása egy tranzakcióban lenne, a rendszer garantálná, hogy Péter vagy a módosítások előtti vagy a módosítások utáni adatokat olvasná.
  - A NoSQL adatbázisok nem támogatják a tranzakciókat, de:
    - A gráfadatbázisok hajlamosak támogatni az ACID tranzakciókat
    - Az aggregátum-orientált adatbázisok egy aggregátum atomi módosítását támogatják. Ez azt jelenti, hogy egy aggregátumon belül van logikai konzisztencia, de az aggregátumok között nincs.

Konzisztencia

- **Olvasási konzisztencia**
  - Természetesen nem lehet minden adatot egy aggregátumban elhelyezni, így minden olyan módosítás, amely több aggregátumot érint, hagy egy időablakot, amikor a kliensek inkonzisztens olvasást tudnak végezni. Azt az időtartamot, amikor az inkonzisztencia fennáll **inkonzisztencia ablak**nak hívják. Egy NoSQL rendszernek lehet nagyon rövid inkonzisztenciaablaka: Az Amazon dokumentációja azt írja, hogy a SimpleDB inkonzisztenciaablaka általában kevesebb, mint egy másodperc.

Konzisztencia

- **Olvasási konzisztencia**
  - **Másolási (replikációs) konzisztencia**: biztosítja, hogy ugyanannak az adatelemnek ugyanaz az értéke, amikor különböző másolatokról olvassuk.
    - Képzeljük el, hogy van egy utolsó hotelszoba egy adott időpontra. A hotel foglalási rendszere sok csomóponton fut. Martin és Cili együtt (mert ők egy pár) szeretnék lefoglalni a szobát, de Martin Londonban van, Cili Budapesten. Telefonon beszélgetnek. Közben Péter Tokióból lefoglalja a szobát. Ez módosítja a másolt szoba elérhetőségét, de a módosítás Budapestre hamarabb odaér, mint Londonba. Amikor Martin és Cili frissítik a böngészőjüket, hogy lássák, hogy a szoba elérhető-e még, Cili foglaltnak látja,míg Martin szabadnak.

Konzisztencia

- **Olvasási konzisztencia**
  - **Egyszer majd valamikor a jövőben (eventually) konzisztencia:** bármely időben lehetnek a csomópontok másolási inkonzisztensek, de ha nincs több módosítás, akkor végső soron minden csomópont ugyanarra az értékre fog módosulni.
  - **Lejárt (stale):** az adat elavult (amely emlékeztet minket arra, hogy a cache a replikáció egy másik formája)

Konzisztencia

- **Olvasási konzisztencia**
  - **Olvasom az írásom (read-your-writes) konzisztencia**, amely azt jelenti, hogy ha egyszer módosítottunk valamit, akkor azt a módosítást garantáltan látni fogjuk.
    - Előfordulhat a következő szituáció: az írásunkat az egyik csomópont fogadja és a klaszteren néhány perces az inkonzisztenciaablak. Az utolsó írásunk után frissítjük a böngészőnket, ami egy másik csomópontra dob át, amelyhez még nem jutott el a módosítás. Így úgy tűnhet, mintha elveszítettük volna az írásunkat.

Konzisztencia

- **Olvasási konzisztencia**
  - Az egyik módja, hogy elérjük az olvasom az írásom (read-your-writes) konzisztenciát egy egyébként Egyszer majd valamikor a jövőben (eventually) konzisztens rendszerben, ha biztosítjuk a **munkamenet (session) konzisztenciát**: egy felhasználói munkamenet olvasom az írásom (read-your-writes) konzisztens.
  - Több módszer létezik a munkamenetkonzisztencia biztosítására. Az általános és gyakran a legkönnyebb módszer a **ragadós** (**sticky) munkamenet**: a munkamenet egy csomóponthoz van kötve (ezt hívják **session affinity**-nek is). A hátránya az, hogy a load balancer nem tudja olyan jól végezni a munkáját.

Konzisztencia

- **Olvasási konzisztencia**
  - A munkamenetkonzisztencia másik megközelítése a **verzióbélyeg** (**version stamps)** használata. Minden adattárbeli érintkezéskor használjuk a munkamenet által látható utolsó időbélyeget. A szerver csomópontoknak biztosítaniuk kell, hogy meglegyen nekik az a módosítás, amely a megfelelő verzióbélyeget tartalmazza, mielőtt egy kérésre válaszol.

Konzisztencia

- **The CAP Theorem**
  - CAP: konzisztencia (consistency), elérhetőség (availability), és partíciótolerancia (partition tolerance)
  - Az elmélet szerinte csak kettőt lehet megkapni egy rendszerben.
    - Az **elérhetőség (availability)** különleges jelentése van ebben a környezetben, azt jelenti, hogy ha tudsz beszélni egy csomóponttal a klaszteben, akkor az tud adatot olvasni és írni.
    - A **partíciótolerancia (partition tolerance)** azt jelenti, hogy a klaszter túl tud élni olyan problémákat, amikor kommunikációs kiesés miatt a klaszter több partícióra esik szét, és azok nem tudnak egymással kommunikálni (ezt a szituációt úgy ismerik, mint **split brain**)

Konzisztencia

- **The CAP Theorem**
  - Az egyedülálló szerver nyilvánvaló példa a **CA rendszerre**: a rendszer konzisztens és elérhető, de nem partíciótoleráns. Ebben a világban él a legtöbb relációs adatbázisrendszer.
  - A CAP theorem valódi lényege: Gyakran azt mondják, hogy az CAP elmélet szerinte “csak kettőt kaphatunk meg a háromból”, a gyakorlatban ez úgy hangzik, hogy **ha egy rendszer partíciókból áll**, mint az elosztott rendszerek, **akkor egyensúlyozni kell a konzisztencia és az elérhetőség között**. Ez nem egy bináris döntés, gyakran kis konzisztencia mellett döntünk, hogy jobb elérhetőséget kapjunk. Az eredmény rendszer nem lesz tökéletesen konzisztens és nem lesz tökéletesen elérhető, de egy olyan kombinációja lesz, amely alkalmas a rendszer szükségleteihez.

Konzisztencia

- **The CAP Theorem**
  - A NoSQL követői azt mondják, hogy a relációs tranzakciók ACID tulajdonságai helyett a NoSQL rendszerek a **BASE tulajdonságokat** követik **(Basically Available, Soft state, Eventual consistency) – (alapvetően elérhető, lágy állapot, egyszer majd valamikor a jövőben konzisztens)**
    - Sem a “basically available” sem a “soft state” nem jól definiált.

Konzisztencia

- **The CAP Theorem**
  - Gyakran jobb a **konzisztencia és a lappangás között egyensúlyozására** gondolni a konzisztencia és az elérhetőség egyensúlyozása helyett. Elosztott környezetben a konzisztenciát úgy javíthatjuk, ha több csomópontot vonunk be az interakcióba (azaz az írásba vagy az olvasásba), azonban minden hozzáadott csomópont növeli az interakció válaszidejét. Az **elérhetőség**re így úgy gondolhatunk, mint a **lappangás** olyan korlátozására, amelyet még tolerálni tudunk, azaz ha a lappangás túl magas lesz, feladjuk és az adatot elérhetetlenként kezeljük. Ez illeszkedik a CAP-ban lévő definícióhoz.

Konzisztencia

- **Nyugalmi tartósság (relaxing durability)**
  - Lehet, hogy egyensúlyozni akarunk a tartósság és a teljesítmény között (nagyobb teljesítmény, kevesebb tartósság)
    - Ha az adatbázisunk leginkább a memóriában fut, a módosításokat a memóriabeli reprezentáción végezzük, és időszakonként kiírjuk a változásokat a lemezre. Ez a megoldás lényegesen gyorsabb válaszidőt eredményez. Az az ára, hogy ha a szerver összeomlik, akkor az utolsó lemezreírás óta történt minden módosítás elveszik.
    - Az egyik példa, ahol megéri ezt alkalmazni, az a felhasználói munkamenetek állapotának tárolása. Egy nagy weboldalnak sok felhasználója lehet, és minden felhasználóhoz tárolhat ideiglenes információkat a munkamenet állapotáról. Egy állapotban nagyon sok tevékenység lehet, amely hatással lehet a weboldal válaszolókészségére. Tény, hogy a munkamenet adatainak az elvesztése nem olyan nagy tragédia, bosszúsággal jár, de lehet, hogy kevesebbel, mint ami egy lassú weboldallal jár. Gyakran a tartósság szükségletét hívásról hívásra határozzák meg, azaz ha fontos az írás, akkor kikényszerítik a lemezreírást.
    - Másik példa a nyugalmi tartósságra a fizikai eszközök telemetrikus adatainak elkapása. Inkább gyorsabban szerezzük meg az adatot, amely azzal a költséggel jár, hogy az utolsó módosítások hiányozhatnak, ha a szervernek le kell állnia.

Konzisztencia

- **Nyugalmi tartósság (relaxing durability)**
  - A tartósság egyensúlyozás másik osztálya a másolt adatokhoz kapcsolódik. **Másolási tartósság**i hiba lép fel, amikor egy csomópont egy módosítást dolgoz fel, de meghibásodik, mielőtt a módosítás egy másik csomópontra kerülne.
    - Ennek egy egyszerű esete akkor történhet, amikor egy master-slave elosztási modellben a slave-ek egy új mastert jelölnek ki automatikusan, mert az eredeti master meghibásodott. Ha a master meghibásodik, akkor azok az írások, amelyek nem kerültek át a másolatokra, elvesznek. Amikor a master ismét elérhető lesz, ezek a módosítások konfliktusba kerülnek az azóta történt írásokkal. Erre a problémára tartóssági problémaként gondolunk, mert azt gondoljuk, hogy a módosítás sikeresen végbement, mivel a master visszaigazolta azt, de a master csomópont meghibásodása miatt elveszett.

Konzisztencia

- **Quorum-ok**
  - Minél több csomópontot bevonunk egy kérésbe, annál nagyobb az esélyünk, hogy elkerüljük az inkonzisztenciát. Ez természetesen felveti a kérdést: hány csomópontot kell bevonnunk, hogy erős konzisztenciát kapjunk?
  - Az **írási quorum**-ot a W>N/2 egyenlőtlenséggel fejezhetjük ki, ahol W az írásban résztvevő csomópontok száma, N a másolatokba bevont csomópontok száma, amelyet úgy is hívnak, hogy **replikációs faktor**.

Konzisztencia

- **Quorum-ok**
  - **Olvasási quorum**: hány csomóponttal kell kapcsolatot létesítened ahhoz, hogy biztos legyél abban, hogy a legfrissebb adatot kapod. Az olvasási quorum egy kicsit komplikáltabb, mert attól függ, hogy hány csomópont igazolta vissza az írást. Akkor van erősen konzisztens olvasásunk, ha R + W > N, ahol R az olvasáskor elérendő csomópontok száma, W csomópont igazolta vissza az írást, és a replikációs faktor N.
  - Ezt az egyenlőtlenséget a peer-to-peer elosztási modell esetén tartsuk észben. Ha egy master-slave elosztási modellünk van, akkor csak a mastert kell írni, hogy elkerüljük az írás-írás konfliktust és hasonlóan a masterről kell olvasni, hogy elkerüljük az olvasás-írás konfliktust.

Konzisztencia

- **Quorum-ok**
  - A replikációs faktornak legtöbbször 3-as értéket javasolnak, amely elegendő egy jó rugalmassághoz. Ha az egyik csomópont meghibásodik, akkor még mindig marad kettő szavazó az írások és az olvasások kezelésére. Ha automatikus rebalancing van, akkor a klaszternek nem kell sok idő, hogy létrehozza a harmadik másolatot, így csekély annak a lehetősége, hogy elveszítünk még egy másolatot, mielőtt a meghibásodott gépet helyreállítanánk.

Map-Reduce

- A map-reduce minta egy olyan módszer, amellyel a feldolgozást úgy szervezhetjük, hogy kihasználjuk egy klaszter több gépből adódó előnyét, miközben az adatokat és annak feldolgozását egy csomóponton tartjuk.
- A “map-reduce” név a map (leképez) és a reduce (csökkent) műveletekből jön, amelyeket a funkcionális programozási nyelvek a kollekciókon végeznek.

Map-Reduce

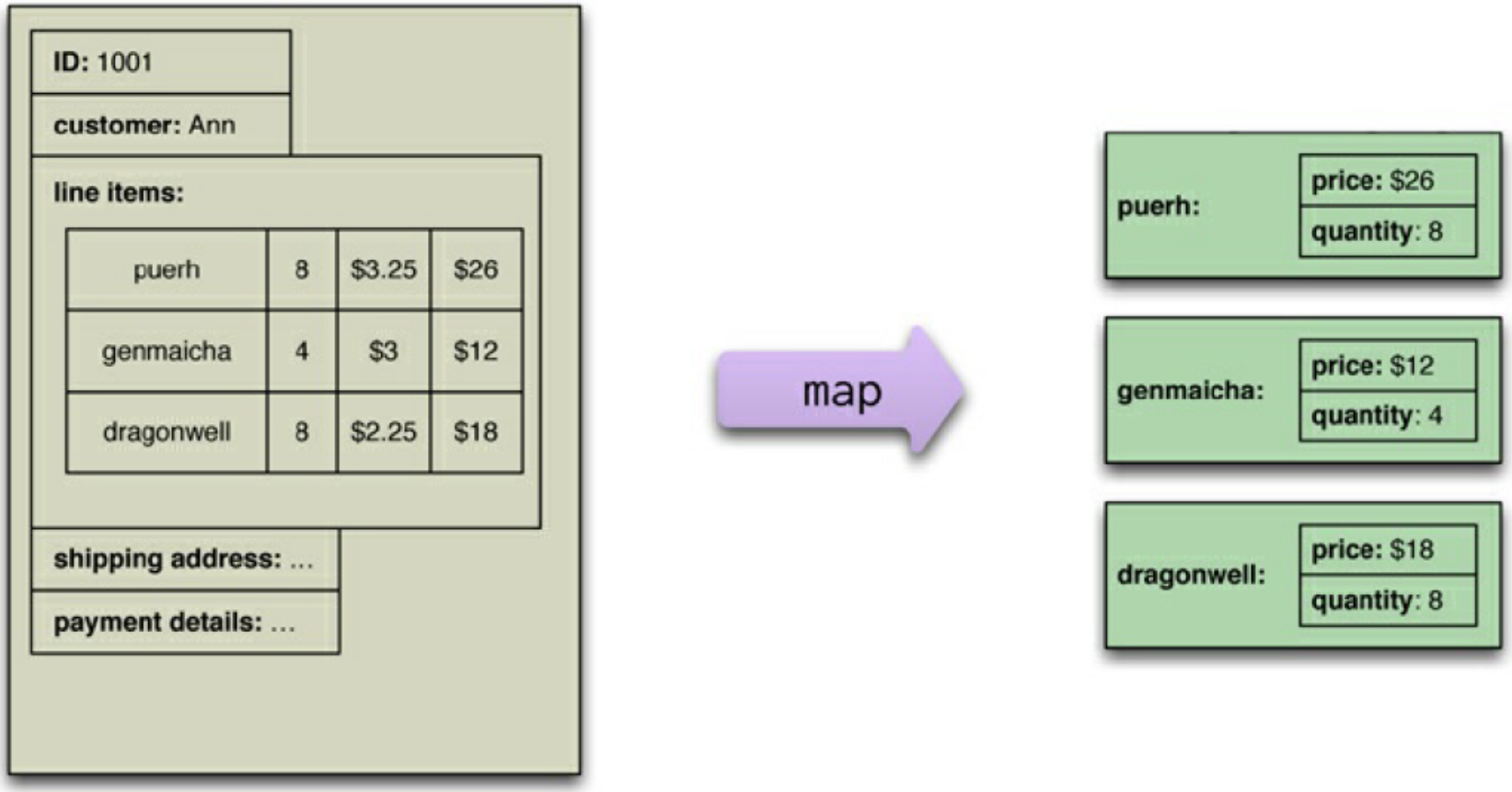
- Tegyük fel, hogy az aggregátumunk egy megrendelést tartalmaz, minden megrendelések több tétel szerepel. Minden tételnek van azonosítója, mennyisége és ára. Általában annak van értelme, hogy egy megrendelést akarunk egyszerre látni. Mivel sok megrendelésünk van, az adathalmazunkat sok gépre szórjuk szét.
- Az értékesítési elemzők a termékeket és a hozzájuk kapcsolódó összbevételt szeretnék látni az elmúlt 7 napra.
- Ez a riport nem illeszkedik az aggregátumstruktúránkba, amely az aggregátum használatának egy rossz tulajdonsága. A riport elkészítéséhez sok rekordot kell átvizsgálni, amelyek a klaszter bármely gépén lehetnek.
- Ez pontosan egy olyan szituáció, ahol a map-reduce-t kell használni.

Map-Reduce

- A map-reduce-ban az első szakasz a **map**. A map egy függvény, amelynek az inputja egy aggregátum, és amelynek az outputja egy csomó kulcs-érték pár. Ebben az esetben az input egy megrendelés. Az output a tételeknek megfelelő kulcs-érték párok. A kulcs a termék azonosító, az értékben a mennyiség és az ár szerepel.
- A map függvény használata független a többi használattól. Ez teszi lehetővé, hogy biztonságosan lehessen párhuzamosan használni, hogy a map-reduce keretrendszer hatékonyan tudja végezni a map-et az egyes csomópontokon és szabadon használja a map-et az egyes megrendeléseken. Ez nagymértékű párhuzamosítását eredményez.

Map-Reduce

- A **reduce** függvény több ugyanolyan kulcsú map outputot kap és kombinálja az értékeiket.
- Így a map függvény 1000 tételt eredményezhet a “Database Refactoring” megrendelésből, a reduce függvény ezt lecsökkentené egyre, amelyben a mennyiség és a bevétel összege lenne. A map függvény egy aggregátum egy adatán dolgozik, a reduce függvény az egy kulcshoz tartozó összes értéket használja.


ID: 1001
customer: Ann
line items:
puerh 8 $3.25 $26
genmaicha 4 $3 $12
dragonwell 8 $2.25 $18
shipping address: ...
payment details: ...
map
puerh:
price: $26
quantity: 8
genmaicha:
price: $12
quantity: 4
dragonwell:
price: $18
quantity: 8


puerh:
price: $26
quantity: 8
price: $36
quantity: 12
price: $44
quantity: 14
reduce
puerh:
price: $106
quantity: 34

Map-Reduce

- A map-reduce keretrendszer elrendezi, hogy a map feladat a megfelelő csomópontokon fusson, hogy feldolgozza az összes aggregátumot és eljuttatja az adatokat a reduce függvényhez. A reduce függvény megírásának megkönnyítésére a keretrendszer összegyűjti egy pár minden értékét, és meghívja a reduce függvényt egyszer a kulcsra és az ahhoz a kulcshoz tartozó összes értékre. Egy map-reduce feladat elvégzéséhez ezt a két függvényt kell megírni.

NoSQL

Az adatbázis típusok

Kulcs-érték adatbázis

- A kulcs-érték (key-value) tár egy egyszerű **hash tábla**, elsősorban akkor használják, amikor az adatbázis minden elérését az elsődleges kulcson keresztül végeznek.
- Képzeljünk el a relációs adatbázisban egy táblát, amelynek két oszlop van, mint ID és NAME. Az ID az elsődleges kulcs, a NAME tárolja az értékeket. Egy relációs adatbázisban a NAME oszlop csak string típusú adatokat tárol. Az alkalmazás így egy ID és VALUE értékpárt tárolhat; ha az ID már létezik, akkor az aktuális értéket felülírja, egyébként egy új bejegyzést készít.

Kulcs-érték adatbázis

- A kulcs-érték tár a legegyszerűbben használható NoSQL adattár egy API nézőpontjából. A kliens vagy kap egy értéket egy kulcshoz vagy egy kulcshoz berak egy értéket vagy töröl egy kulcsot az adatbázisból. Az érték egy blob (binary large object), amelyet az adattár csak tárol anélkül, hogy törődne vele vagy tudná, hogy mi van benne. A tartalmáért az alkalmazás felelős, azaz ő érti meg mi van tárolva benne. Mivel a kulcs-érték tár mindig elsődleges kulcs szerint érik el adatbázist, ezért nagyon jó teljesítményt nyújtanak és könnyen skálázhatóak.

Kulcs-érték adatbázis

- Népszerű kulcs-érték adatbázisok:
  - Riak,
  - Redis,
  - Memcached DB és változatai,
  - Berkeley DB,
  - HamsterDB,
  - Amazon DynamoDB,
  - Project Voldemort

Kulcs-érték adatbázis

- Néhány kulcs-érték tár (mint a Redis) esetén a tárolt aggrgátumnak nem kell feltétlenül egy tartományobjektumnak lennie, lehet valamilyen adatstruktúra. A Redis támogatja a lista, halmaz, hashtábla struktúrákat és a range, diff, union, intersect műveleteket. Ez a jellemző lehetővé teszi, hogy a Redist több különböző módon lehessen használni, ne csak kulcs-érték tárként.

Kulcs-érték adatbázis

- Ez a jegyzet leginkább a Riak adatbázisra fókuszál. A Riak lehetővé teszi, hogy a kulcsokat vödrökben (bucket) tárolja, amely egy módja a kulcsok csoportosításának - gondoljunk a vödörre, mint kulcsok egy névterére.
- Ha a Riakban munkamenet adatokat, vásárói kosár információt és a felhasználó preferenciáit akarunk tárolni, akkor ezeket mindet tárolhatjuk ugyanabban a vödörben egy egyszerű kulccsal és ezekhez az objektumokhoz egy értékkel. Így egy olyan egyszerű objektumunk lesz, amely minden adatot tárol és egy vödörben helyez el.
- A kulcs-érték tárak, mint a Redis különböző adatstruktúrák tárolását is támogatja, mint set, hash tábla, string, stb.

Kulcs-érték adatbázis

- **Konzisztencia**
  - A konzisztencia csak az egy kulcshoz tartozó műveletekre értelmezhető, mivel a művelet egy get, put, vagy delete egy kulcson. Optimista írást lehet végrehajtani, de nagyon drága megvalósítani, mert az adattár nem tudja megállapítani az adatváltozást.
  - Egy elosztott kulcs-érték tár, mint a Riak az *eventually consistent* konzisztenciamodellt valósítja meg. Mivel az értéket már lehet, hogy más csomópontra másolták, a Riaknak két lehetősége van, hogy feloldja a módosítási konfliktust: vagy a legújabb írás nyer és a régebbi veszít, vagy mindkét (összes) értéket visszaküldi, hogy a kliens oldja fel a konfliktust. A Riak-ban ezt az opciót be lehet állítani a vödör létrehozásakor.

Kulcs-érték adatbázis

- **Tranzakció**
  - A különböző kulcs-érték táraknak különböző tranzakcióspecifikációi vannak. Általában nincs garancia az írásra. Sok adattár sok különböző módon valósítja meg a tranzakciókat. A Riak a quorum fogalmat használja.
- **Lekérdezési lehetőségek**
  - Minden kulcs-érték tár le tud kérdezni a kulcs szerint, és általában ennyi lehetőség van. Ha az érték oszlop más attribútuma szerint szeretnénk lekérdeni, azt az adatbázis nem teszi lehetővé: az alkalmazásnak ki kell olvasni az értéket, hogy kitalálja, hogy mely attribútumok felelnek meg a feltételeknek.

Kulcs-érték adatbázis

- **Lekérdezési lehetőségek**
  - A kulcs szerinti lekérdezésnek van egy érdekes mellékhatása. Mi van akkor, ha nem ismerjük a kulcsot például hibakeresés alatti ad-hoc lekérdezés alatt? A legtöbb adattár nem adja vissza az összes elsődleges kulcs listáját, vagy ha mégis, a kulcsok listájának a kinyerése és az érték lekérdezése nagyon fáradtságos lenne. Néhány kulcs-érték adatbázis megkerüli ezt azzal, hogy lehetővé teszi az értéken belüli keresést. A **Riak Search** lehetővé teszi, hogy lekérdezd az adatot éppen úgy, mintha a Lucene indexelő motor használatával kérdeznéd le az adatot.

Kulcs-érték adatbázis

- **Adatstruktúra**
  - A kulcs-érték adatbázis nem törődik azzal, hogy mit tárol a kulcs-érték pár érték részében. Az érték lehet blob, text, JSON, XML, stb.
- **Skálázás**
  - Sok kulcs-érték tár sharding használatával skáláz. Sharding esetén a kulcs értéke határozza meg, hogy a kulcsot melyik csomóponton tárolja. Tegyük fel, hogy a kulcs első karaktere alapján shardingolunk. Ha a kulcs f4b19d79587d, akkor f-fel kezdődik, akkor az más csomópontra kerül, mint a ad9c7a396542 kulcs. Ez a sharind megoldás növelheti a teljesítményt, ha több csomópontot adunk a klaszterhez.
  - A sharding néhány problémával is jár. Ha az f-et tároló csomópont meghibásodik, a csomóponton lévő adatok elérhetetlenné válnak és nem lehet új f-fel kezdődő adatot sem tárolni.

Kulcs-érték adatbázis

- **Skálázás**
  - Az adattárak, mint a Riak lehetővé teszik, a CAP elmélet eszközeit vezéreljük:
    - N (annak a száma, hogy egy kulcs-érték másolatot (replikák) hány csomóponton tárolunk)
    - R (az a szám, ahány csomópontról az adatokat be kell olvasni, hogy az olvasást sikeresnek tekintjük), és
    - W (az a szám, ahány csomópontokra az írást el kell végezni, hogy az írást sikeresnek tekintsük).
  - Tegyük fel, hogy egy 5 csomópontos Riak klaszterünk van. Ha N-et 3-ra állítjuk, az azt jelenti, hogy minden adatot legalább 3 csomópontra másolunk. Ha R-et 2-re állítjuk, az azt jelenti, egy GET kérésre valamelyik 2 csomópontnak kell válaszolnia, hogy sikeres legyen. És ha W-t 2-re állítjuk, akkor a PUT kérést 2 csomópontra kell írni mielőtt az írást sikeresnek tekintjük.

Kulcs-érték adatbázis

- **Skálázás**
  - Ezek a beállítások teszik lehetővé, hogy finomhangoljuk a csomópont hibákat az írás és az olvasás műveletekhez. Az igényeink szerint változtathatunk ezeken az értékeken a jobb olvasási elérhetőség és a jobb írási elérhetőség érdekében. Általában a W értékét a konzisztenciaigényekhez választjuk. Ezeket az értékeket a vödör létrehozásakor alapértelmezettként be lehet állítani.

Kulcs-érték adatbázis

- **Alkalmas használati esetek**
  - **Munkamenet információk tárolása**
    - Általában minden web munkamenet egyedi és egy egyedi munkamenet azonosítót kap. Azoknak az alkalmazásoknak, amelyek a munkamenet azonosítót lemezen vagy relációs adatbázisban tárolják, nagyon jót fog tenni, ha a munkamenet azonosítókat egy kulcs-érték tárba helyezzük át, mivel minden munkamenettel kapcsolatos információt egy PUT kéréssel tárolhatunk és egy GET kéréssel kinyerhetünk. Ez az egyszerű kérés művelet nagyon gyorssá teszi, és a munkamenethez kapcsolódó minden információ egy egyszerű objektumban lesz tárolva. A Memcached-hez hasonló megoldásokat sok webalkalmazás használja. A Riak-ot akkor használják, ha az elérhetőség fontos.

Kulcs-érték adatbázis

- **Alkalmas használati esetek**
  - **Felhasználói profilok, preferenciák**
    - Majdnem minden felhasználónak van egyedi felhasználói azonosítója, neve és más jellemzője, mint kedvelt nyelv, szín, időzóna, mely termékekhez van a felhasználónak hozzáférése, stb. Ezt mindet be lehet tenni egy objektumba, így egy egyszerű GET művelettel ki tudjuk nyerni ezeket az infókat az adatbázisból. Hasonlóan a termék profilokat is lehet tárolni.
  - **Vásárlói kosár adatok**
    - E-commerce weboldalaknak van a felhasználóhoz kötött kosara. Ha azt akarjuk, hogy a kosár mindig elérhető legyen, akár más böngészőből, más gépről, más munkamenetből, akkor a vásárlási információkat belerakhatjuk egy értékként az adatbázisba, ahol a kulcs a felhasználó azonosítója lesz. Egy Riak klaszter lenne a legalkalmasabb az ilyen alkalmazásra.

Kulcs-érték adatbázis

- **Mikor ne használjuk**
  - **Kapcsolatok az adatok között**
    - Ha különböző adathalmazok közötti kapcsolatokra van szükségünk, vagy kulcsok különböző halmazai közötti viszonyra, akkor a kulcs-érték tár nem a legjobb megoldás. Bár néhány kulcs-érték tár biztosít link-walking eszközöket.
  - **Többműveletes tranzakciók**
    - Ha több kulcsot mentünk és az egyik mentése alatt hiba történik és helyre akarjuk állítani a többi műveletet, akkor a kulcs-érték tár nem a legjobb megoldás.

Kulcs-érték adatbázis

- **Mikor ne használjuk**
  - **Adatok szerinti lekérdezés**
    - Ha a kulcs-érték pár érték részében valami fontos alapján szeretnénk kulcsra keresni, azt a kulcs-érték tár nem fogja jól végezni. Nincs mód arra, hogy az értéket az adatbázis oldalon vizsgáljuk, kivéve néhány terméket, mint a Riak Search vagy az indexelő motorok, mint a Lucene vagy Solr.
  - **Műveletek kulcshalmazokon**
    - A műveleteket egyidőben csak egy kulcson lehet végezni, nincs mód arra, hogy egyidőben több kulcson végezzünk műveletet. Az ilyen esetet kliens oldalról kell megoldani.

Dokumentum adatbázisok

- A dokumentum adatbázisokban a fő fogalom a dokumentum. A dokumentum tár dokumentumokat tárol és nyer ki, amely lehet XML, JSON, BSON (binary JSON), stb. Ezek a dokumentumok önleíró, hierarchikus fa adat struktúrák, amelyek tartalmazhatnak map-eket, kollekciókat és skalár értékeket. A tárolt dokumentumok hasonlóak egymáshoz, de nem kell, hogy teljesen ugyanazok legyenek. A dokumentum adatbázisban egy kulcshoz egy dokumentumot tárolunk. Azaz a dokumentum adatbázisra gondolhatunk úgy, mint egy kulcs-érték adatbázis, ahol az érték vizsgálható.

Dokumentum adatbázisok

```
{ "firstname": "Martin",
  "likes": [ "Biking",
             "Photography" ],
  "lastcity": "Boston",
  "lastVisited":
}
```

Dokumentum adatbázisok

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London",
                     "Pune", "Bangalore" ],
  "addresses": [
     { "state": "AK",
       "city": "DILLINGHAM",
       "type": "R"
     },
     { "state": "MH",
       "city": "PUNE",
       "type": "R" }
   ],
  "lastcity": "Chicago"
}
```

Dokumentum adatbázisok

- Ha megnézzük az előző dokumentumokat, láthatjuk, hogy hasonlóak, de vannak különbségek az attribútumnevekben. A dokumentum adatbázis ezt lehetővé tesz. Az adat sémája más-más lehet az egyes dokumentumokban, de ezek a dokumentumok ugyanahhoz a kollekcióhoz tartozhatnak (a relációs modellben egy tábla minden sorának követnie kell ugyanazt a sémát). A meglátogatott városok listáját tömbbel reprezentáljuk vagy a címek egy listáját a fődokumentumba ágyazott dokumentumok listájával. A dokumentumba gyermekobjektumok, mint alobjektumok beágyazása könnyű elérést és jobb teljesítményt biztosít.

Dokumentum adatbázisok

- Ha megnézzük a dokumentumokat, láthatjuk, hogy néhány attribútum hasonló, mint a firstname vagy a city. Ugyanakkor a második dokumentumban vannak attribútumok, amelyek nem léteznek az elsőben, mint a címek, míg like-ok vannak az elsőben, de a másodikban nem.
- A dokumentumokban nincsenek üres attribútumok, ha egy attribútum nem található, akkor feltételezhetjük, hogy az adott dokumentumnál nem adták meg vagy nem fontos. A dokumentumok lehetővé teszik, hogy új attribútumokat hozzunk létre anélkül, hogy definiáljuk őket vagy a meglévő dokumentumokat változtatnánk.

Dokumentum adatbázisok

- Néhány népszerű dokumentum adatbázis
  - MongoDB,
  - CouchDB,
  - Terrastore,
  - OrientDB,
  - RavenDB ,
  - és a jól ismert és gyakran szidott Lotus Notes.

Dokumentum adatbázisok

- A MongoDB-t fogjuk használni, hogy bemutassuk a család jellemzőit.
- Minden MongoDB példánynak több *adatbázisa* van, és minden adatbázisnak több *kollekciója*.
- Amikor egy dokumentumot tárolunk, ki kell választanunk, hogy megy adatbázisba és kollekcióba tartozik.
- Az _id egy speciális mező, amely a Mongo minden dokumentumában megtalálható, hasonlóan az Oracle ROWID-jához. A MongoDB-ben az _id-t a felhasználó adhatja meg, amíg az egyedi.

Dokumentum adatbázisok

- **Konzisztencia**
  - A MongoDB konzisztenciáját **replica sets**-ek használatával annak a kiválasztásával lehet beállítani, hogy várjunk-e arra, hogy az írások minden slave-re vagy egy megadott számú slave-re átmásolódjanak. Minden írás meghatározhatja azon szerverek számát, amelyeken az írásnak meg kell történnie, mielőtt sikeresként tér vissza.

Dokumentum adatbázisok

- **Konzisztencia**
  - Egy olyan parancs, mint a db.runCommand({ getlasterror : 1 , w : "majority" }) megmondja az adatbázisnak, hogy milyen erős konzisztenciát akarsz. Például, ha van egy szerverünk és a w-t többségnek állítjuk be, akkor az írás azonnal visszatér, mert csak egy csomópont van. Ha három csomópont van a replica set-ben és a w-t többségnek állítjuk be, akkor az írásnak legalább két csomóponton meg kell történnie, mielőtt sikeresnek jelenti az adatbázis. Növelhetjük a w-t az erősebb konzisztencia miatt, de sérülni fog az írási teljesítmény, mivel az írásnak több csomóponton is teljesülnie kell.

Dokumentum adatbázisok

- **Konzisztencia**
  - A Replica set-ek azt is lehetővé teszik, hogy növeljük az olvasási teljesítményt azzal, hogy lehetővé tesszük a slave-kről való olvasást a slaveOk; beállításával, amely paramétert be lehet állítani a kapcsolathoz, az adatbázishoz, a kollekcióhoz vagy minden művelet esetén.
  - Beállíthatjuk a slaveOk-ot műveletenként, így eldönthetjük, hogy mely műveletek dolgozhatnak együtt a slave csomópontokon lévő adatokkal.
  - Az íráshoz is meg lehet adni paramétereket. Beállíthatunk erős írási konzisztenciát. Alapértelmezés szerint egy írást sikeresnek tekintünk, ha az adatbázis megkapta. Ez változtathatjuk úgy, hogy várjon a szerver arra, hogy az írás szinkronizálva legyen a lemezzel vagy két vagy több slave-re rámásolódjon. Ezt úgy ismerjük, mint WriteConcern: biztossá tesszük, hogy bizonyos írások a masterre és néhány slave-re íródnak azzal, hogy a WriteConcern-t REPLICAS_SAFE-re állítjuk. .

Dokumentum adatbázisok

- **Tranzakció**
  - Az egyszerű dokumentum szintű tranzakciókat atomi tranzakcióként ismerjük.
  - Alapértelmezés szerint minden írást sikeresnek jelent a rendszer. Az írás feletti finomabb vezérlést a WriteConcern paraméter használatával lehet elérni.
  - WriteConcern.REPLICAS_SAFE használatával biztosítjuk, hogy a megrendelést egynél több csomópontra írja a rendszer mielőtt sikeresnek jelenti. A WriteConcern különböző szintjei lehetővé teszik, hogy egy biztonsági szintet válasszunk az íráshoz. Például amikor napló bejegyzéseket írunk, akkor elég a legalacsonyabb biztonsági szint, WriteConcern.NONE.

Dokumentum adatbázisok

- **Elérhetőség**
  - A dokumentum adatbázisok adatmásolással próbálják fejleszteni az elérhetőséget, amihez master-slave beállítást használnak. Ugyanaz az adat több csomóponton is elérhető és a kliensek akkor is megkaphatják az adatot, ha az elsődleges csomópont nem működik. Általában az alkalmazáskódnak nem kell megállapítania, hogy az elsődleges csomópont elérhető-e vagy sem. A MongoDB replikációt valósít meg, a magas elérhetőséget a **replica set**ek használatával biztosítja.
  - Egy replica setben két vagy három csomópont van, amelyek egy aszinkron master-slave replikációban vesznek részt. A replica-set csomópontok maguk közül szavazzák meg a master-t vagyis az elsődleges csomópontot. Feltételezve, hogy minden csomópontnak ugyanolyan szavazójoga van, néhány csomópont kitüntetett lehet azzal, hogy közelebb vannak más szerverekhez, vagy több RAM-juk van, stb. A felhasználók ezt befolyásolhatják azzal, hogy prioritást rendelnek a csomópontokhoz, ami egy 0 és 1000 közötti szám.

Dokumentum adatbázisok

- **Elérhetőség**
  - Minden kérés a master csomóponthoz megy, és az adatot a slave csomópontokra másolja a DBMS. Ha a master csomópont leáll, a replica set maradék csomópontja új mastert szavaz maguk közül. Ezután minden kérés az új masterhez kerül és a slave csomópontok az új mastertől veszik az adatot. Ha a hibás csomópont ismét elérhető lesz, kapcsolódik és átveszi a többi csomópontról az adatokat, így az adatai aktuálisak lesznek.

Dokumentum adatbázisok

- **Elérhetőség**
  - Az alkalmazások az elsődleges (master) csomópontot írják és olvassák. Amikor a kapcsolat létrejön, az alkalmazásnak a replica setben csak egy csomóponthoz kell kapcsolódnia (mindegy, hogy az az elsődleges vagy nem), és a többi csomópontot automatikusan felfedezi. Amikor az elsődleges csomópont leáll, a driver a replica set által megszavazott új elsődleges csomóponttal fog beszélni. Az alkalmazásnak nem kell kezelnie a kommunikációs hibát és nem kell csomópontot választania. A replica setek használatával egy magas elérhetőségű dokumentum adattárunk lehet.
  - A replica set-eket általában adatredundanciára, automatikus hibaáthidalásra, olvasás skálázására, leállás nélküli szerverkarbantartásra és katasztrófahelyreállításra használják. Hasonló elérhetőségi beállítások érhetőek el a CouchDB-vel, a RavenDB-vel, a Terrastore-ral, és más termékekkel.

Dokumentum adatbázisok

- **Lekérdezési lehetőségek**
  - A dokumentumadatbázisok különböző lekérdezési eszközöket biztosítanak. A CouchDB lehetővé teszi a nézeteken keresztüli lekérdezést, amelyek összetett lekérdezések a dokumentumon, és amelyek lehetnek materializált nézetek vagy dinamikusak. Ha egy termék összes megtekintését vagy átlagos értékelését szeretnénk megkapni, a CouchDB-vel létrehozhatunk egy nézetet, amelyben map-reduce-t használunk.
  - Ha sok kérés van, akkor nem akarjuk mindne kéréshez kiszámolni a darabszámot és az átlagot, hanem egy materializált nézetet hozunk létre, amely előre kiszámolja az értékeket és az eredményt az adatbázisban tárolja. Ezek a materializált nézek módosulnak, amikor lekérdezik őket.

Dokumentum adatbázisok

- **Lekérdezési lehetőségek**
  - A dokumentum adatbázisok egyik jó tulajdonsága (összehasonlítva a kulcs-érték tárakkal), hogy a kérdezhetünk le dokumentumon belüli adatokat anélkül, hogy az egész dokumentumot a kulcsa szerint ki kellene nyerni és utána a dokumentumot kellene vizsgálni. Ez a jellemző ezeket az adatbázisokat közelebb hozza az RDBMS lekérdező modelljéhez.
  - A MongoDB-nek vagy egy lekérdező nyelve, amelyet JSON-on keresztül fejezi ki magát és olyan szerkezetei vannak, mint a $query a where feltételhez, a $orderby a rendezéshez, és $explain, amely megmutatja a lekérdezés lekérdezéstervét (explain plan). Egy MongoDB lekérdezés létrehozásához sok ehhez hasonló szerkezet van.

Dokumentum adatbázisok

- **Skálázás**
  - Erős olvasási terheléshez történő skálázás elérhető több olvasási slave hozzáadásával, így minden olvasást a slave-khez lehetne irányítani. Adott egy olvasás igényes alkalmazás, és egy 3 csomópontos replica-set klaszter, ekkor mi több olvasási kapacitást adhatunk a klaszterhez, ha több slave csomópontot adunk a rendszerhez, hogy azok olvasásokat végezzenek a slaveOk flaggel. Ez az olvasás horizontális skálázása.
  - Amikor egy új csomópontot hozzáadunk a rendszerhez, az szinkronizálódni fog a többi csomóponttal, kapcsolódik a replica set-hez, mint másodlagos csomópont és elkezd olvasási kéréseket kiszolgálni. Ennek a beállításnak az előnye, hogy nem kell újraindítani egyik csomópontot sem, így az alkalmazás sem áll le.

Dokumentum adatbázisok

- **Skálázás**
  - Amikor az íráshoz szeretnénk skálázni, akkor el kell kezdeni sharding-olni az adatot. A shardingban az adatot egy bizonyos értelemben felosztjuk, de aztán különböző Mongo csomópontokra mozgatjuk őket. Az adat dinamikusan mozog a csomópontok között, hogy biztosítsa, hogy a shard-ok mindig ki vannak egyensúlyozva. A klaszterhez adhatunk több csomópontot, hogy növeljük az írható csomópontok számát, lehetővé téve az írás horizontális skálázását.

Dokumentum adatbázisok

- **Skálázás**
  - Ha adunk egy új shardot a már létező shardolt klaszterunkhoz, az adat már 4 shard között lesz egyensúlyba hozva 3 helyett. És amikor ez az adatmozgatás és infrastruktúra átalakítás történik, az alkalmazás nem fog leállni, bár a klaszter talán nem fog optimálisan teljesíteni, amikor nagy adatmennyiséget mozgat át, hogy kiegyensúlyozza a shardokat.
  - A shard kulcsa fontos szerepet játszik.
  - Ha a shardokat a felhasználóihoz közel szeretnénk elhelyezni, akkor a felhasználó helyén alapuló shardolás egy jó ötlet lehet.

Dokumentum adatbázisok

- **Alkalmas használati esetek**
  - **Eseménynaplózás**
    - A dokumentum adatbázisok mindenféle típusú események tudnak tárolni és egy központi adattárként szolgálhatnak az események tárolásához. Ez különösen igaz akkor, ha az esemény által rögzített adatok típus állandóan változik.
  - **Tartalommenedzsment rendszer, Blog platform**
    - Mivel a dokumentumadatbázisoknak nincs előre definiált sémájuk, és általában értik a JSON dokumentumokat, jól működnek a tartalommendzsment rendszerekben vagy olyan alkalmazásokban, amelyek: for publishing websites, managing user comments, user registrations, profiles, web-facing documents.

Dokumentum adatbázisok

- **Alkalmas használati esetek**
  - **Web Analitika vagy valós idejű elemzés**
    - A dokumentum adatbázisok adatokat tárolhatnak a valós idejű elemzésekhez; mivel a dokumentum részeit módosítani lehet, nagyon könnyű oldalnézeteket vagy egyedi látogatókat tárolni, és sémamódosítás nélkül lehet új metrikákat hozzáadni.
  - **E-Commerce alkalmazások**
    - Az e-commerce alkalmazásoknak gyakran rugalmas sémára van szükségük a termékekhez és a megrendelésekhez, aza képesnek kell lenniük az adatmodellüket fejleszteni (javítani) drága adatbázis átalakítás és adatmigráció nélkül.

Dokumentum adatbázisok

- **Mikor ne használjuk**
  - **Összetett tranzakciók, amelyek különböző műveleteket tartalmaznak**
    - Ha olyan atomi műveletre van szükségünk, amely több dokumentációt is érint, akkor a dokumentumadatbázis valószínűleg nem lesz alkalmas számunkra.
  - **Lekérdezések különböző aggregátumstruktúrákra**
    - A rugalmas séma azt jelenti, hogy az adatbázis a sémán nem kényszerít ki semmilyen megszorítást. Az adatot az alkalmazás entitásaként menti. Ha ad hoc módon akarjuk lekérdezni ezeket az entitásokat, akkor a lekérdezéseink mindig változni fognak. Mivel az adat aggregátumként van mentve, ha az aggregátum terve mindig változik, akkor az aggregátumot a szemcsézettség legalacsonyabb szintjén kell menteni, azaz normalizálni kell az adatot. Ebben a forgatókönyvben a dokumentumadatbázis nem működik.

Oszlopcsalád tárak

- Pl:
  - Cassandra,
  - HBase,
  - Hypertable, és
  - Amazon SimpleDB,
- Az oszlopcsalád tárak lehetővé teszik, hogy olyan adatokat tároljunk, amelyek esetén kulcsokhoz értékeket rendelünk, és az értékeket több oszlopcsaládba csoportosítsunk.
- A Cassandráról fogunk beszélni:
  - A Cassandra úgy jellemezhető, hogy gyors és könnyen skálázható egy klaszteren az írási műveletekhez. A klaszternek nincs master csomópontja, így minden írást és olvasást bármely csomópont kezelhet.

Oszlopcsalád tárak

- Az oszlopcsalád adatbázisok az adatokat az oszlopcsaládokban, mint olyan sorokban tárolják, amelyeknek egy sorkulcshoz rendelve sok oszlopuk van. Az oszlopcsaládok kapcsolódó adatok csoportja, amelyeket általában együtt érnek el. Egy ügyfélhez (Customer) gyakran ugyanakkor el akarjuk érni a Profile információját, de nem a megrendeléseit (Order).

Oszlopcsalád tárak

- A Cassandra tárolásának az alapegysége az oszlop. Egy Cassandra oszlop egy név-érték párból áll, ahol a név kulcsként is viselkedik.
- Minden kulcs-érték pár egy oszlop és mindig egy időbélyeg értékkel van együtt tárolva. Az időbélyeget arra használja, hogy jelölje az adat frissességét, írási konfliktusokat oldjon fel, a régi adattal foglalkozni, stb.
- Ha egy oszlopadatot már nem használnak, a helye a „compaction” fázisban felszabadítható.

Oszlopcsalád tárak

```
{
  name: "fullName",
  value: "Martin Fowler",
  timestamp: 12345667890
}
```

- Az oszlopnak a kulcsa a fullName, az értéke a Martin Fowler, és egy időbélyeget adtak hozzá.
- Egy sor oszlopok kollekciója, amelyek egy kulcshoz vannak rendelve. Hasonló sorok kollekciója lesz egy oszlopcsalád. Amikor egy oszlopcsalád oszlopai egyszerű oszlopok, az oszlopcsaládot úgy hívják, hogy **standard oszlopcsalád** (**standard column family)**.

Oszlopcsalád tárak

```
//column family
{ //row
  "pramod-sadalage" : {
     firstName: "Pramod",
     lastName: "Sadalage",
     lastVisit: "2012/12/12"
  }
//row
  "martin-fowler" : {
     firstName: "Martin",
     lastName: "Fowler",
     location: "Boston"
  }
}
```

Oszlopcsalád tárak

- Az egyes oszlopcsaládokat össze lehet hasonlítani az RDBMS-ek tábláiba lévő sorok tárolójával, ahol a kulcs azonosítja a sort és a sor több oszlopot tartalmaz. A különbség az, hogy a különböző soroknak nem kell, hogy ugyanolyan oszlopai legyenek és az oszlopokat bármelyik sorhoz bármikor hozzá lehet adni anélkül, hogy azokat másik sorokhoz is hozzáadnánk.
- A pramod-sadalage és a martin-fowler soroknak különböző oszlopai vannak, és mindkettő egy oszlopcsalád sora.
- Amikor egy oszlop oszlopok egy leképezéséből áll, akkor szuper oszlopunk (**super column**) van. Egy szuperoszlop egy névből és egy értékből áll, ahol az érték oszlopok egy leképezése. Gondoljunk egy zsuper oszlopra úgy, mint oszlopok egy tára.
- Amikor szuper oszlopokat használunk egy oszlopcsalád létrehozására, akkor egy szuper oszlopcsaládot kapunk (**super column family)**.

```
//super column family
{ //row
name: "billing:martin-fowler",
value: {address: {name: "address:default",
          value: {fullName: "Martin Fowler",
                  street:"100 N. Main Street",
                  zip: "20145" }},
          billing: {name: "billing:default",
                    value: {creditcard: "8888-8888-8888-8888",
                            expDate: "12/2016"}}}
//row
name: "billing:pramod-sadalage",
value: {address: {name: "address:default",
           value: {fullName: "Pramod Sadalage",
                   street:"100 E. State Parkway",
                   zip: "54130"}},
           billing: {name: "billing:default",
                     value: {creditcard: "9999-8888-7777-4444",
                             expDate: "01/2016"}}}
}
```

Oszlopcsalád tárak

- A szuper oszlopcsaládok arra jók, hogy a kapcsolód adatokat együtt tartsák, de amikor néhány oszlopra a legtöbbször nincs szükség, az oszlopokat a Cassandra betölti és feldolgozza, amely valószínűleg nem optimális.
- A Cassandra a standard és a szuper oszlopcsaládokat kulcsterekbe (**keyspace**) bepakolja. Egy kulcstér hasonló egy RDBMS-beli adatbázishoz, ahol az alkalmazáshoz kapcsolódó minden oszlopcsaládat tárol a rendszer. A kulcstereket létre kell hozni, hogy az oszlopcsaládokat hozzájuk lehessen rendelni..

Oszlopcsalád tárak

- **Konzisztencia**
  - Amikor a Cassandra egy írást kap, az adatot először a naplóba (commit log) írja, aztán egy memóriabeli struktúrába amit **memtable**-nek hívnak. Egy írási műveletet sikeresnek tekintjük, ha ki van írva a naplóba és a memtable-be. Az írásokat összegyűjti a memóriába és időközönként kiírja az **SSTable**-nak hívott struktúrába. Az SSTable-ket nem módosítja a rendszer, ha már egyszer kiírta. Ha módosítás történik az adaton, akkor egy új SSTable-t ír. A nem használt SSTable-kat helyreállítja a tömörítő (**compactation)**.

Oszlopcsalád tárak

- **Konzisztencia**
  - Vizsgáljuk meg, hogy az olvass műveletet hogyan befolyásolják a konzisztencia beállítások. Ha konzisztenciabeállításnak ONE-t adtunk meg alapértelmezett értékként minden olvasásművelethez konzisztenciabeállításnak, akkor amikor az olvasáskérés létrejön, a Cassandra az első replica adatával tér vissza, akkor is, ha az adat régi. Ha az adat régi, a következő olvasások az utolsó (legújabb) adatot kapják. Ezt a folyamatot **read repair**-nek hívják. Az alacsony konzisztenciaszintet akkor jó használni, ha nem foglalkozunk azzal, hogy az adat régi-e és/vagy magas olvasás teljesítménybeli követelményeink vannak.

Oszlopcsalád tárak

- **Konzisztencia**
  - Hasonlóan, ha írási műveletet végzünk, a Cassandra csak egy csomópont commit logját írná és visszatérne a válasszal a klienshez. A ONE konzisztencia akkor jó, ha nagyon magas írási teljesítménybeli követelményeink vannak és nem bánjuk ha néhány írás elveszik, amely akkor történhet ha a szerver leáll mielőtt az írást egy másik csomópontra másolnánk.

Oszlopcsalád tárak

- **Konzisztencia**
  - Ha a QUORUM konzisztenciabeállítást használjuk az írási és az olvasási műveletekhez is, akkor biztosítjuk, hogy a csomópontok többsége válaszoljon az olvasásra és a legújabb időbélyeggel rendelkező oszlop érkezzen vissza a klienshez, miközben azokat a replikákat, amelyek nem tárolják a legújabb adatot, az olvasási művelet segítségével feljavítjuk (odamásoljuk az újabb adatot). Az írási műveletek alatt, a QUORUM konzisztencia beállítás azt jelenti, hogy az írási műveletet a csomópontok többségére propagálni kell mielőtt sikeresnek tekintjük és a klienst értesítjük.

Oszlopcsalád tárak

- **Konzisztencia**
  - Az ALL konzisztencia szint használata azt jelenti, hogy minden csomópontnak válaszolnia kell az írásokra és az olvasásokra, amely miatt a klaszter nem lesz hibatűrő – ha csak egy csomópont leáll, az írások és az olvasássok meghiúsulnak. A rendszer tervező feladata, hogy a konzisztencia szintet az alkalmazás követelményeihez igazítsa.
  - Ugyanazon az alkalmazáson belül a konzisztencia követelmények különbözőek lehetnek. A konzisztencia szintet lehet változtatni az egyes műveletekhez. Például a felhasználói kommentek megmutatásának más konzisztencia követelményei vannak, mint a felhasználó utolsó megrendelésének a státusza.

Oszlopcsalád tárak

- **Konzisztencia**
  - A kulcstér (**keyspace)** létrehozása alatt, megadhatjuk, hogy hány adatreplikát szeretnénk tárolni. Ez a szám meghatározza az adat replikációs faktorát (azaz hány helyre lesz másolva). Ha a replikációs faktor 3, az adatot 3 helyre másolja a rendszer. Ha írunk és olvasunk a Cassandrával, és a konzisztencia értéket 2-esre állítjuk, akkor azt kapjuk, hogy a R + W érték nagyobb, mint a replikációs faktor (2 + 2 > 3), amely az írások és az olvasások alatt jobb konzisztenciát ad.

Oszlopcsalád tárak

- **Konzisztencia**
  - Futtathatjuk a csomópont javító parancsot a kulcstéren (keyspace) és kényszeríthetjük a Cassandrát, hogy hasonlítson össze minden felelős kulcsot a többi másolattal. Mivel ez a művelet drága, megjavíthatunk csak egy bizonyos oszlopcsaládot vagy oszlopcsaládok egy listáját.
  - Amíg a csomópont nem működik, azt az adatot, amelyről azt gondoljuk, hogy azon a csomóponton van, más csomópontok átveszik. Ahogy a csomópont ismét elérhető lesz, az adaton történt változások visszakerülnek a csomópontra. Ezt a technikát **hinted handoff**-nak (célzott leadás) hívják, amely lehetővé teszi a hibás csomópont gyorsabb helyreállítását.

Oszlopcsalád tárak

- **Tranzakció**
  - Cassandraban az írás atomi sorszinten, amely azt jelenti, hogy egy adott sorkulcshoz az oszlopok beszúrása és módosítása egy egyedülálló írásként lesz tekintve és vagy sikerül vagy nem. Az írásokat először a commit log-ba (naplóba) és a memtables-be írja a rendszer. Az írást akkor tekintjük sikeresnek, ha a naplóba írás és a memtables-be írás sikeres. Ha egy csomópont leáll, a naplót használja a rendszer arra, hogy a változásokat megvalósítsa.

Oszlopcsalád tárak

- **Elérhetőség**
  - A Cassandra-t magas elérhetőségre tervezték, nincs master a klaszterben, ezért mindenki egyenrangú. Egy klaszter elérhetősége növelhető ha a kérések konzisztenciaszintjét csökkentjük.
  - Az elérhetőséget a (R + W) > N formula irányítja, ahol
    - W azon csomópontok minimum száma, amelyeket sikeresen kell írni,
    - R azon csomópontok minimum száma, amelyeknek egy írásra sikeresen kell válaszolniuk,
    - N azon csomópontok száma, amelyek részt vesznek az adatok másolásában.
  - Az elérhetőséget az R és W értékek változtatásával hangolhatjuk rögzített számú N esetén.

Oszlopcsalád tárak

- **Elérhetőség**
  - Egy 10 csomópontos Cassandra klaszterben a keyspace (kulcstér) replikációs faktora 3 (N=3), ha W = 2-t és R = 2 állítunk be, akkor (2 +2) > 3-t kapunk. Ebben az esetben ha egy csomópont leáll, az elérhetőség nem nagyon érintett, mivel az adatokat a másik két csomópontról ki lehet nyerni.
  - Ha W = 2 és R = 1, és két csomópont leáll, a klaszter nem lesz elérhető az írásra, de még tudunk olvasni.
  - Hasonlóan, ha R = 2 és W = 1, akkor tudunk írni, de az olvasásra nem érhető el a rendszer.
  - Az R + W > N egyenlőtlenséggel, tudatos döntést hozunk a konzisztencia kérdésről.

Oszlopcsalád tárak

- **Lekérdezési lehetőségek**
  - Amikor a Cassandrában adatmodellt tervezünk, tanácsos az oszlopokat és az oszlop családokat úgy létrehozni, hogy az az adatok olvasásához legyen optimalizálva, mivel nincs gazdag lekérdező nyelve, és mivel az oszlop családba beszúrt adatot az egyes sorokban oszlopnevenként rendezi a rendszer. Ha van egy oszlopunk, amelyet sokkal többször lekérdezünk, mint más oszlopokat, a teljesítmény szempontjából jobb, ha azt az értéket használjuk sorkulcsként.

Oszlopcsalád tárak

- **Lekérdezési lehetőségek**
  - Az alap lekérdezések alatt, amelyeket a Cassandrában használhatunk, a GET-et, SET-et és a DEL-t értjük.

```
SET Customer['mfowler']['city']='Boston';
SET Customer['mfowler']['name']='Martin Fowler';
SET
Customer['mfowler']['web']='www.martinfowler.com';

GET Customer['mfowler'];
GET Customer['mfowler']['web'];
DEL Customer['mfowler']['city'];
DEL Customer['mfowler'];
```

Oszlopcsalád tárak

- **Lekérdezési lehetőségek**
  - A Cassandrának van egy lekérdező nyelve, amely SQL szerű utasításokat támogat, úgy hívják, hogy Cassandra Query Language (CQL).

```
CREATE COLUMNFAMILY Customer (
KEY varchar PRIMARY KEY,
name varchar, city varchar, web varchar);
INSERT INTO Customer (KEY,name,city,web)
VALUES ('mfowler', 'Martin Fowler', 'Boston',
'www.martinfowler.com');
SELECT * FROM Customer
SELECT name,web FROM Customer
```

- CQL-nek sok eszköze van az adatok lekérdezésére, azonban korántsem annyi, mint az SQL-nek.
- CQL nem tesz lehetővé joinokat vagy allekérdezéseket, és a where feltétele általában egyszerű.

Oszlopcsalád tárak

- **Skálázás**
  - Egy létező Cassandra klaszter skálázása több csomópont hozzádását jelenti. Mivel nincs master csomópont, amikor csomópontot adunk hozzá a klaszterhez, javítjuk a klaszter kapacitását, hogy több olvasást és írást tudjon végezni. A horizontális skálázás ezen típusa lehetővé teszi a maximális működési időt, mivel a klaszter akkor is kiszolgálja a kéréseket, amikor új csomópontokat adunk a klaszterhez.

Oszlopcsalád tárak

- **Alkalmas használati esetek**
  - **Eseménynaplózás**
    - Az oszlopcsalád adatbázisok azzal a képességükkel, hogy képesek bármilyen adatstruktúrát tárolni, egy nagyszerű választás eseményinformációk tárolására, mint alkalmazás állapotok vagy az alkalmazások hibái. Egy cégen belül minden alkalmazás írhatja az eseményeit a Cassandrába a saját oszlopaiba és appname:timestamp formájú sorkulcsaiba. Mivel az írásokat skálázhatjuk, a Cassandra ideális működik esemény naplózó renszerként.
  - **Tartalommenedzsment rendszer (Content Management Systems), Blogging Platforms**
    - Az oszlopcsaládok használatával blog bejegyzéseket tárolhatunk tag-ekkel, kategóriákkal, linkekkel és trackback-ekkel különböző oszlopokban. A kommentek lehetne ugyanabban a sorban vagy egy másik keyspace-ben tárolva, Hasonlóan a blog felhasználóit és az aktuális blogokat különböző oszlop családokban tárolhatjuk

Oszlopcsalád tárak

- **Alkalmas használati esetek**
  - **Számlálók**
    - Gyakran a web alkalmazásokban meg kell számolnunk és kategóriákba kell sorolnunk egy oldal felhasználóit, hogy analitikát számoljunk. Használhatjuk a CounterColumnType-t egy oszlopcsalád létrehozásakor. Ha egy oszlopcsalád elkészült, minden, az alkalmazáson belül meglátogatott oldalhoz és minden felhasználóhoz tetszőleges oszlopaink lehetnek.

```
CREATE COLUMN FAMILY visit_counter
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND
comparator=UTF8Type;
INCR visit_counter['mfowler'][home] BY 1;
INCR visit_counter['mfowler'][products] BY 1;
INCR visit_counter['mfowler'][contactus] BY 1;
UPDATE visit_counter SET home = home + 1 WHERE
KEY='mfowler'
```

Oszlopcsalád tárak

- **Alkalmas használati esetek**
  - **Expiring Usage**
    - Lehet, hogy a felhasználóknak demo elérést biztosítunk vagy reklámokat (ad banner) akarunk megjeleníteni egy weboldalon egy ideig. Ezt megtehetjük a „lejáró” oszlopok (**expiring columns)** használatával: a Cassandra lehetővé teszi, hogy olyan oszlopaink legyenek, amelyek egy idő után automatikusan törlődnek. Ezt az időt úgy hívják, hogy TTL (Time To Live) és másodpercben van megadva. Amikor a TTL lejár, az oszlopot törli a rendszer. Ha az oszlop nem létezik, a jogot vissza lehet vonni vagy a bannert el lehet távolítani.

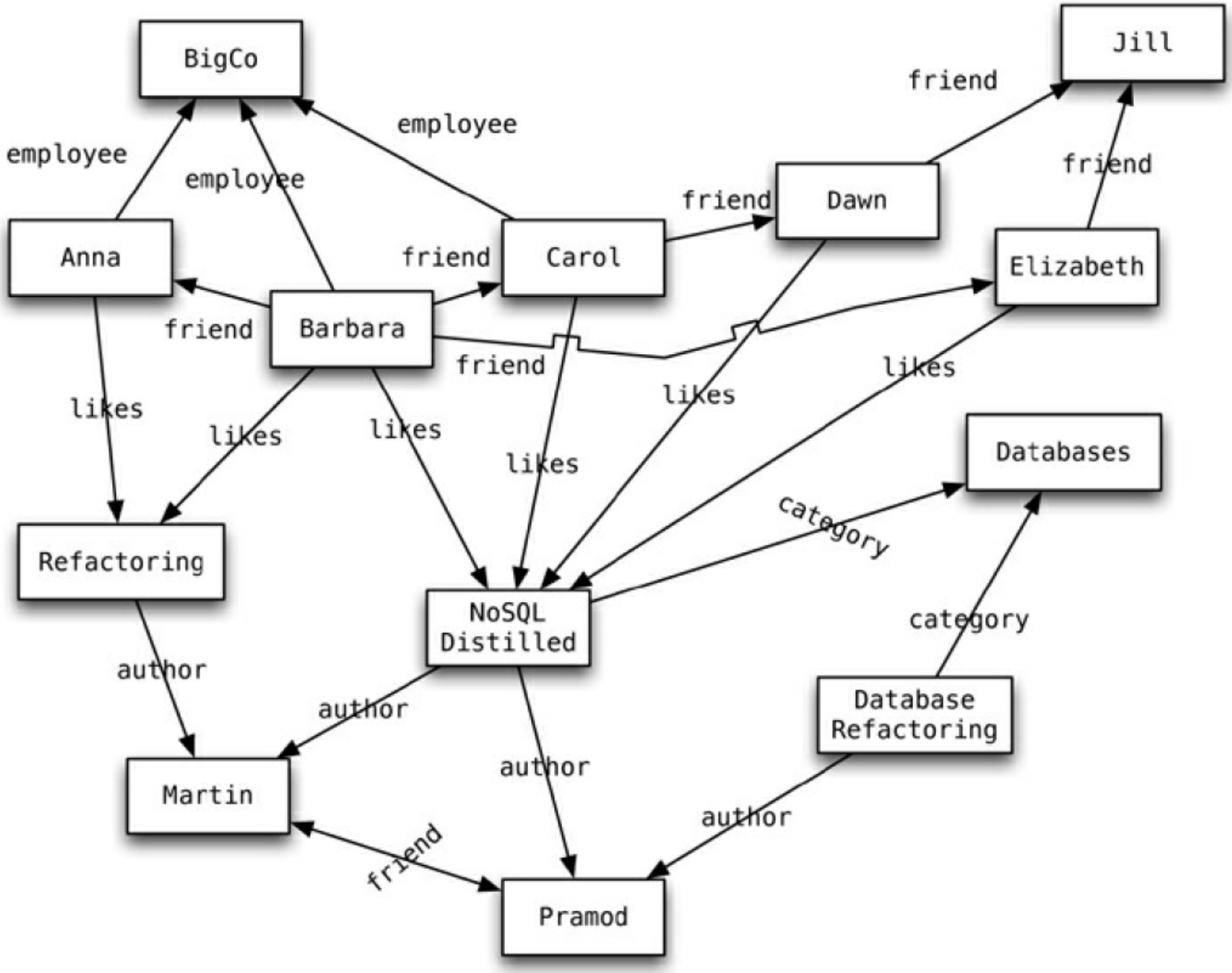
```
SET Customer['mfowler']['demo_access'] = 'allowed'
   WITH ttl=2592000;
```

Oszlopcsalád tárak

- **Mikor ne használjuk**
  - Van néhány probléma, amelyekre az oszlopcsalád adatbázisok nem a legjobb megoldások, mint az olyan rendszerek, amelyeknek ACID ttranzakciókra van szükségük az írásokhoz és az olvasásokhoz.
  - Ha az adatokat lekérdezések használatával aggregálni szeretnénk (mint SUM vagy AVG), ezt a kliens oldalon kell megtenni úgy, hogy a kliens által kinyert összes sort használjuk.
  - A Cassandra nem a legjobb a korai prototípusokhoz vagy az initial tech spikes-hoz: A korai fázisok alatt nem vagyunk biztosak abban, hogy a lekérdezési minták hogyan fognak változni, és ha a lekérdezési minták változnak, akkor az oszlop család terveinket is változtatni kell. Ez súrlódást okoz a termék innovációs csapatnál és lelassítja a fejlesztési produktivitást. Az RDBMS-ben a séma változtatás magas költségű, amelyet ellensúlyoz a lekérdezés változtatásának alacsony költsége. A Cassandrában a költségek magasabbak lehetnek a lekérdezés változások esetén összehasonlítva a séma változásokkal.

Gráf adatbázisok

- A gráf adatbázisok lehetővé teszik, hogy egyedeket és az egyedek közötti kapcsolatokat tároljuk.
- Az egyedeket csomópontokként is ismerhetjük, amelyeknek tulajdonságai vannak. Gondoljunk úgy egy csomópontra, mint az alkalmazásban egy objektum példányára.
- A kapcsolatokat élekként (edges) is ismerhetjük, amelyeknek tulajdonsága vannak. Az éleknek van irányítottsági jelentősége; a csomópontokat a kapcsolatok szerint rendszerezi a rendszer, amely lehetővé teszi, hogy érdekes mintákat fedezzünk fel a csomópontok között.
- A gráf szervezet lehetővé teszi, hogy az adatokat egyszer tároljuk és a kapcsolatokra alapozva különböző módokon értelmezzük.


BigCo
Jill
friend
employee
employee
employee
friend
friend
Dawn
Anna
friend
Carol
Elizabeth
friend
Barbara
friend
likes
likes
likes
likes
likes
likes
Databases
category
Refactoring
NoSQL
Distilled
category
author
author
Database
Refactoring
author
Martin
author
friend
Pramod

Gráf adatbázisok

- A csomópontok egyedek, amelyeknek tulajdonságai vannak (A Martin csomópont egy csomópont, amelynek egy Martinra állított név tulajdonsága van.
- Az éleknek van típusa, mint kedveli, írta.
- A Martin és a Pramod csomópontokat összeköti egy él, amelynek a típusa a barátok.
- Az éleknek több tulajdonsága lehet.
- A kapcsolat típusoknak irányítási jelentősége van, a barátok kapcsolat kétirányú, míg a kedveli egyirányú.

Gráf adatbázisok

- Ha létrehoztuk a gráfunkat a csomópontokkal és az élekkel, a gráfot sokféleképp lekérdezhetjük, mint „keressük az összes olyan csomópontot, akit a Big Co alkalmazott és szereti a NoSQL Distilled-et”.
- A gráf egy lekérdezését úgy is nevezik, hogy **traversing** (átkelés, bejárás) a gráfon. A gráfadatbázisok egy előnye, hogy változtathatjuk a lekérdezési követelményeket anélkül, hogy a csomópontokon vagy az éleken változtatni kellene. Ha „minden olyan csomópontot meg akarunk kapni, amely szereti a NoSQL Distilled-et”, akkor megkaphatjuk anélkül, hogy a meglévő adatokat vagy az adatmodellt változtatni kellene, mert úgy járjuk be a gráfot, ahogy akarjuk.
- A gráfadatbázisokban a kapcsolatok bejárása nagyon gyors. A csomópontok közötti kapcsolatot nem a lekérdezési időben számolja a rendszer, hanem ténylegesen kapcsolatként van tárolva. A tárolt kapcsolatok bejárása gyorsabb, mint minden lekérdezéshez kiszámolni.
- A csomópontokat különböző típusú kapcsolatok köthetik össze, lehetővé téve, hogy az egyedek közötti kapcsolatok reprezentálását, és azt, hogy a dolgok között második kapcsolat is létrejöjjön. Mivel nincs korlát arra, hogy egy csomópontnak hány és milyen fajta kapcsolata lehet, így mindet ugyanabban a gráf adatbázisban lehet reprezentálni.

Gráf adatbázisok

- Gráfadatbázisok:
  - Neo4J,
  - Infinite Graph,
  - OrientDB,
  - FlockDB (amely egy speciális eset: egy gráf adatbázis, amely csak egymélységű kapcsolatokat vagy adjacent list (rendezetlen listák kollekciója) listákat támogat.

Gráf adatbázisok

- Neo4J-ben, egy gráf létrehozása olyan egyszerű, mint létrehozni két csomópontot és aztán egy kapcsolatot.

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");
Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");

martin.createRelationshipTo(pramod, FRIEND);
pramod.createRelationshipTo(martin, FRIEND);
```

Gráf adatbázisok

- A kapcsolatok „first-class citizens” a gráf adatbázisokban; a gráfadatbázisok értékének a nagyrészt a kapcsolatokból származik.
- A kapcsolatoknak nem csak egy típusuk, kezdő és végcsomópontjuk van, hanem van saját tulajdonságuk is (property). Ezeket a tulajdonságokat használva értelmet adhatunk a kapcsolatokhoz. Pl. mikor lettek barátok, két csomópont között mennyi a távolság. A kapcsolatok tulajdonságait használhatjuk amikor lekérdezzük a gráfot.
- Mivel a gráfadatbázisok legnagyobb ereje a kapcsolatok és azok tulajdonságaiból jön, sok gondolkodás és tervezői munka szükséges, hogy megtervezzük a kapcsolatokat az adott területen, amelyekkel dolgozni szeretnénk. Egy új kapcsolattípus hozzáadása egyszerű. Azonban a meglévő csomópontok és azok kapcsolatának a változtatása hasonló az adatmigrációhoz, mert ezeket a változtatásokat el kell végezni minden egyes meglévő csomóponton és minden egyes meglévő kapcsolaton.

Gráf adatbázisok

- **Konzisztencia**
  - Mivel a gráfadatbázisok összekapcsolt csomópontokon dolgoznak, a legtöbb gráfadatbázis megoldás nem támogatja a csomópontok különböző szervereken történő elosztását. Van néhány gráfadatbázis, amelynek van ilyen lehetősége, pl. Infinite Graph.
  - Egy szerveren belül az adatok mindig konzisztensek, különösképpen a Neo4j-ben, amely teljesen megfelel az ACID tulajdonságoknak. Ha a Neo4j-t klaszterben futtatjuk, a master írása végül is (eventually) szinkronizálva lesz a slave-ekkel, míg a slave-ek mindig elérhetőek olvasásra. A slave-k írása lehetséges és azonnal szinkronizálva vannak a masterrel; más slave-ek nem azonnal lesznek szinkronizálva, mivel várniuk kell az adatra, amelyet a mastertől kapnak.
  - A gráf adatbázisok tranzakciókon keresztül biztosítják a konzisztenciát. Nem tesznek lehetővé lógó kapcsolatokat: a kezdő és cég csomópontoknak mindig létezniük kell, és a csomópontokat csak akkor lehet törölni, ha nincs kapcsolódó kapcsolatuk.

Gráf adatbázisok

- **Tranzakció**
  - A Neo4j megfelel az ACID tulajdonságoknak.
  - Mielőtt valamelyik csomópontot megváltoztatjuk vagy egy meglévő csomóponthoz új kapcsolatokat adunk, egy tranzakciót kell kezdenünk. Ha a műveleteket nem csomagoljuk tranzakciókba, akkor egy NotInTransactionException kivételt kapunk.
  - Olvasást lehet végezni tranzakció kezdés nélkül.
  - Egy tranzakciót sikeresnek kell megjelölni, különben a Neo4j feltételezi, hogy hibás volt és visszagörgeti ha befejeztük. A sikeresnek megjelölés befejezés nélkül még nem commitálja az adatot az adatbázisban.

Gráf adatbázisok

```
Transaction transaction = database.beginTx();
try {
  Node node = database.createNode();
  node.setProperty("name", "NoSQL Distilled");
  node.setProperty("published", "2012");
  transaction.success();
} finally {
  transaction.finish();
}
```

Gráf adatbázisok

- **Elérhetőség**
  - Neo4J (ez a könyv az 1.8-as verzióról beszél, de már 3.0-nál járunk) magas elérhetőséget és el replicált slave-ek biztosításával.
  - Ezek a slave-ek írást is tudnak kezelni: Amikor írják őket, akkor szinkronizálnak az aktuális masterrel és az írást először a masteren commitálják aztán a slave-en. Más slave-ek végül (eventually) megkapják a módosítást.
  - Más gráf adatbázisok, mint az Infinite Graph vagy FlockDB, a csomópontoknak elosztott tárat biztosít.

Gráf adatbázisok

- **Lekérdezési lehetőségek**
  - A gráf adatbázisoknak vannak lekérdezőnyelvei, mint a Gremlin.
  - A Gremlin egy tartományspecifikus nyelv a gráf lekérdezésére; Minden gráfadatbázist le tud kérdezni, amely a Blueprints property gráfot valósítja meg.
  - A Neo4J-nek is van lekérdező nyelve: a Cypher. A lekérdező nyelveken kívül a Neo4j lehetővé teszi, hogy nyelvi kötések (language bindings) segítségével kérdezzük le a gráfot.
  - A csomópontok tulajdonságaira (properties) lehet indexet tenni. Hasonlóan az élet tulajdonságaira is, így a csomópont vagy az él megtalálható az értéke alapján.
  - Az indexeket kell lekérni ha meg akarjuk találni a kezdő csomópontot, hogy egy bejárást elkezdjünk.

Gráf adatbázisok

- **Lekérdezési lehetőségek**

```
Node node=nodeIndex.get("name", "Barbara").getSingle();
Node martin=nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships();
incomingRelations =
       martin.getRelationships(Direction.INCOMING);
incomingRelations =
       martin.getRelationships(Direction.OUTGOING);
```

Gráf adatbázisok

- **Lekérdezési lehetőségek**
  - A gráfadatbázisok nagyon erőteljesek, amikor valamilyen mélységben be akarod járni a gráfokat és meghatározol egy kezdő csomópontot a bejáráshoz. Ez különösképpen hasznos, amikor meg akarsz találni csomópontokat, amelyek több, mint egy szinttel lefelé kapcsolódnak a kezdő csomóponthoz.
  - Ahogy a gráf mélysége nő, egyre több értelme lesz a kapcsolatok bejárásának egy Traverser használatával, ahol megadhatjuk, hogy bejövő, kimenő vagy mindkét típusú kapcsolatot keresünk.
  - Létrehozhatsz traversert, ami fentről lefelé halad (mélységi bejárás) vagy oldalra halad (szélességi bejárás) a gráfban az Order érték meghatározásával (BREADTH_FIRST vagy DEPTH_FIRST). A bejárásnak valamilyen csomópontnál kell kezdődnie.

Gráf adatbázisok

- **Lekérdezési lehetőségek**

```
Node barbara =
     nodeIndex.get("name","Barbara").getSingle();
Traverser friendsTraverser =
      barbara.traverse(
                  Order.BREADTH_FIRST,
                  StopEvaluator.END_OF_GRAPH,
                  ReturnableEvaluator.ALL_BUT_START_NODE,
                  EdgeType.FRIEND,
                  Direction.OUTGOING);
```

Gráf adatbázisok

- **Lekérdezési lehetőségek**
  - A Neo4J a **Cypher** lekérdező nyelvet is biztosítja az adatok lekérdezésére.
  - A Cyphernek szüksége van egy csomópontra, ahonnan elkezdi a lekérdezést (START). A kezdőcsomópontot meg lehet határozni a csomópont ID-jával, csomópont ID-k listájával vagy index alapú kereséssel.
  - A Cypher a MATCH kulcsszót használja minták illesztésére a kapcsolatokban; (ő a lekérdezés kezdete)
  - A WHERE kulcsszó megszűri a kapcsolat és a csomópont tulajdonságát.
  - A RETURN kulcsszó megadja, hogy mivel térjen vissza a lekérdezés: csomópontokkal, kapcsolatokkal, vagy csomópontok vagy kapcsolatok mezőivel.
  - A Cypher metódusokat biztosít, hogy az adatokat rendezzük(ORDER), aggregáljuk (AGGREGATE), mellőzzük (SKIP), vagy korlátozzuk (LIMIT).

Gráf adatbázisok

- **Lekérdezési lehetőségek**

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)--(connected_node)
RETURN connected_node
```

- Ha az irányítottság fontos:

```
MATCH (barbara)<--(connected_node)
MATCH (barbara)-->(connected_node)
```

- Le lehet kérdezni kapcsolatokat a :RELATIONSHIP_TYPE meghatározásával:

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

Gráf adatbázisok

- **Lekérdezési lehetőségek**

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[relation]->(related_node)
WHERE type(relation) = 'FRIEND' AND relation.share
RETURN related_node.name, relation.since
```

Gráf adatbázisok

- **Skálázás**
  - A gráfadatbázisokkal a sharding nehéz, mivel a gráf adatbázis nem aggregátum orientált sem kapcsolat orientált. Mivel bármelyik csomópont kapcsolatban lehet más csomóponttal, a gráfbejáráshoz jobb, ha a kapcsolódó csomópontokat ugyanazon a szerveren tároljuk. Teljesítmény szempontjából nem jó egy gráf bejárása, ha a csomópontjai más gépeken vannak.

Gráf adatbázisok

- **Skálázás**
  - 3 módja van a gráfadatbázis skálázásának:
    - Mivel a gépek ma már sok memóriával rendelkeznek, adjunk elegendő memóriát a szervernek, hogy a csomópontok és a kapcsolatok azon halmaza, amin dolgozunk elférjen a memóriában. Ez a technika csak akkor segít, ha az adathalmaz, amin dolgozunk elfér egy valós méretű memóriában.
    - Az olvasási skálázást fejleszthetjük, ha több csak olvasható slave-et adunk a szerverhez, és minden írás a masterhez kerül. Ez nagyon hasznos, amikor az adathalmaz elég nagy, hogy ne férjen el egy gép memóriájában, de elég kicsi, hogy több gépre lehessen másolni. A slave-k hozzájárulhatnak az elérhetőséghez és az olvasási skálázáshoz, mivel úgy konfigurálhatóak, hogy soha ne legyenek masterek, mindig csak olvashatóak maradnak.
    - Megoszthatjuk (shard) az alkalmazás oldali adatokat tartomáynspecifikus ismeretet használva. Például a csomópontokat, amelyek Észak-Amerikához kapcsolódnak létrehozhatjuk az egyik szerveren, míg az Ázsiához kapcsolódó csomópontokat egy másikon hozzuk létre. Az alkalmazásszintű sharding esetén a csomópontok fizikailag különböző adatbázisokban vannak tárolva.

Gráf adatbázisok

- **Alkalmas használati esetek**
  - **Kapcsolódó adatok**
    - A közösségi hálók (social networks) területen lehet a gráfadatbázisokat nagyon hatékonyan használni. A közösségi hálók nem csak a barátságról szólnak, hanem reprezentálhatunk dolgozókat, a tudásukat, vagy a projekteken kikkel dolgoztak együtt. Minden link-gazdag területre jól illeszthető a gráf adatbázis.
    - Ha egy adatbázisban több terület (közösségi, térbeli (spatial), kereskedelmi) egyedei között van kapcsolat, akkor ezek a kapcsolatok még értékesebbé tehetők azzal, ha lehetővé tesszük, hogy a területeken átívelően járjuk be a gráfot.

Gráf adatbázisok

- **Alkalmas használati esetek**
  - **Útvonaltervezés, áruszállítási menetirányító, és hely alapú szolgáltatások**
    - Minden hely vagy cím, amelyről szállítanak, egy csomópont. És minden csomópont, ahová a szállítónak (ember) teljesítenie kell a szállítást egy gráfcsomópontjént modellezhető. A csomópontok közötti kapcsolatoknak lehet távolságtulajdonságuk, így lehetővé teszik a termékek hatékony módú kiszállítását. A távolság és a hely tulajdonságokat is lehet használni érdekes helyek (point of interest) gráfjaiban, így az alkalmazásunk javaslatokat tehet közeli jó éttermekre vagy szórakozó helyekre. Létrehozhatunk eladási pontokhoz is csomópontokat, mint könyvesbolt, vagy étterem, és figyelmeztethetjük a felhasználót, ha az egyikhez közel van, hely alapú szolgáltatásokat biztosítva.

Gráf adatbázisok

- **Alkalmas használati esetek**
  - **Ajánlói motorok**
    - Mivel csomópontokat és kapcsolatokat hozunk létre a rendszerben, használhatjuk őket ajánlások létrehozására, mint „a barátaid is megvásárolták ezt a terméket” vagy “amikor megrendelték ezt az árut, akkor ezeket az árukat is általában megrendelték.” Vagy arra is lehet őket használni, hogy ajánlatokat tegyünk az utazóknak megemlítve, hogy amikor más látogatók Barcelonába ennek akkor gyakran meglátogatják az Antonio Gaudi alkotásait.
    - A gráfadatbázisok egy érdekes mellékhatása az ajánlatokra, hogy ahogy az adatméret nő, az ajánlatok számára elérhető csomópontok és kapcsolatok száma gyorsan növekszik. Ugyanezeket az adatokat lehet használni információ bányászatra, például: mely adatokat vásárolták együtt. Mint más ajánlói motorokat, a gráf adabázisokat is lehet arra használni, hogy kapcsolatokban mintákat keressünk vagy csaló tranzakciókat fedezzünk fel.

Gráf adatbázisok

- **Mikor ne használjuk**
  - Ha az egyedek mindegyikét vagy egy részét szeretnénk módosítani - például egy analitikai megoldásban, ahol lehet, hogy minden egyednek egy változás tulajdonságát kell módosítani – akkor a gráf adatbázisok nem biztos, hogy optimálisa, mert egy tulajdonság minden egyeden történő változása nem egy egyszerű művelet. Még akkor is ha az adatmodell működik a problémakörre, néhány adatbázis képtelen lehet kezelni a sok adatot, például egy globális gráf műveletben (amely a teljes ráfot magában foglalja).

Redis

Reference

- Josiah L. Carlson: Redis in Action
- https://redis.io/documentation

Redis

- Redis is used for both a primary and a secondary storage medium for data
- Redis has two different forms of persistence available for writing in-memory data to disk in a compact format.
  - a point-in-time dump either when certain conditions are met (a number of writes in a given period) or when one of the two dump-to-disk commands is called.
  - an append-only file that writes every command that alters data in Redis to disk as it happens. Depending on how careful you want to be with your data, append-only writing can be configured to never sync, sync once per second, or sync at the completion of every operation.

Redis

- Redis supports master/slave replication where slaves connect to the master and receive an initial copy of the full database.
- Redis data are always fast, because data is always in memory.
- Queries to Redis don’t need to go through a typical query parser/optimizer.

Connection commands

- **AUTH password**
  - Request for authentication in a password-protected Redis server. Redis can be instructed to require a password before allowing clients to execute commands. This is done using the requirepass directive in the configuration file.
  - If password matches the password in the configuration file, the server replies with the OK status code and starts accepting commands. Otherwise, an error is returned and the clients needs to try a new password.
  - **Note**: because of the high performance nature of Redis, it is possible to try a lot of passwords in parallel in very short time, so make sure to generate a strong and very long password so that this attack is infeasible.

Connection commands

- **ECHO message**
  - Returns message.
- **QUIT**
  - Ask the server to close the connection. The connection is closed as soon as all pending replies have been written to the client.

Other commands

- **TIME**
  - The TIME command returns the current server time as a two items lists: a Unix timestamp and the amount of microseconds already elapsed in the current second. Basically the interface is very similar to the one of the gettimeofday system call.
  - Return value
    - Array reply, specifically: A multi bulk reply containing two elements: unix time in seconds and microseconds.

Other commands

- **KEYS pattern**
  - Returns all keys matching pattern.
  - NOTE: For example, Redis running on an entry level laptop can scan a 1 million key database in 40 milliseconds.
  - **Warning**: Don't use KEYS in your regular application code. If you're looking for a way to find keys in a subset of your keyspace, consider using SCAN or sets.
  - Supported glob-style patterns:
    - h?llo matches hello, hallo and hxllo
    - h*llo matches hllo and heeeello
    - h[ae]llo matches hello and hallo, but not hillo
    - h[^e]llo matches hallo, hbllo, ... but not hello
    - h[a-b]llo matches hallo and hbllo
  - Use \ to escape special characters if you want to match them verbatim.
  - Examples
    - redis> **KEYS *name***
    - redis> **KEYS a??**
    - redis> **KEYS ***

Redis data structures

- Redis allows us to store keys that map to any one of five different data structure types:
  - STRING,
  - LIST,
  - SET,
  - HASH,
  - ZSET.
- Each of the five different structures have
  - some shared commands, as well as
  - some commands that can only be used by one or two of the structures.

Redis data structures

- What it contains Structure read/write ability
- STRING:
  - Can contain Strings, integers, or floatingpoint values
  - You can operate on the whole string, parts, increment/decrement the integers and floats
- LIST
  - Can contain Linked list of strings
  - You can push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value
- SET
  - Can contain onordered collection of unique strings
  - You can add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items
- HASH
  - Can contain unordered hash table of keys to values
  - You can add, fetch, or remove individual items, fetch the whole hash
- ZSET (sorted set)
  - Can contain ordered mapping of string members to floating-point scores, ordered by score
  - Add, fetch, or remove individual values, fetch items based on score ranges or member value

Redis

- You can use redis-cli console to interact with Redis.

String in Redis

- A key-value pair can be stored in it
- Commands (redis-cli):
  - SET: Sets the value stored at the given key
    - Set hello world
  - GET:Fetches the data stored at the given key. If the key does not exist the special value nil is returned.
    - Get hello
  - DEL: Deletes the value stored at the given key (works for all types)
    - Del hello
- Key- hello, value- world

List in Redis

- It supports a linked list structure: ordered sequence of strings.
- Commands (redis-cli):
  - RPUSH: Pushes the value onto the right end of the list
  - LPUSH: Pushes the value onto the left end of the list
  - LRANGE: Fetches a range of values from the list
  - LINDEX: Fetches an item at a given position in the list
  - LPOP: Pops the value from the left end of the list and returns it
  - RPOP: Pops the value from the right end of the list and returns it

List in Redis

- rpush list-key item
- rpush list-key item2
- rpush list-key item3
- lrange list-key 0 -1
  - (It fetches the entire list by passing a range of 0 for the start index and -1 for the last index.)
- lindex list-key 1
  - (It fetches individual items from the list. The index of the first element is 0) (item2)
- lpop list-key
  - (It pops an item from the list makes it no longer available. It writes out item)
- lrange list-key 0 -1
  - (item2, item3)

Set in Redis

- SETs are a sequence of strings,
- SETs use a hash table to keep all strings unique (though there are no associated values).
- Sets are unordered
- Commands
  - SADD: Adds the item to the set
  - SMEMBERS: Returns the entire set of items (it can be slow for large SETs)
  - SISMEMBER: Checks if an item is in the set
  - SREM: Removes the item from the set, if it exists

Set in Redis

- sadd set-key item
- sadd set-key item2
- sadd set-key item3
- sadd set-key item
- smembers set-key
  - 1) "item"
  - 2) "item2"
  - 3) "item3"
- sismember set-key item4
  - (integer) 0
- sismember set-key item
  - (integer) 1
- srem set-key item2
- srem set-key item2
- smembers set-key
  - 1) "item"
  - 2) "item3"

Hash in Redis

- It stores a key-value pair where the value is sequence of items. An item is also a key-value pair, where the value can store the same thing as the value of the STRING structure.
- Hash is a miniature version of Redis
- Commands:
  - HSET: Stores the value at the key in the hash
  - HGET: Fetches the value at the given hash key
  - HGETALL: Fetches the entire hash
  - HDEL: Removes a key from the hash, if it exists

Hash in Redis

- hset hash-key sub-key1 value1
- hset hash-key sub-key2 value2
- hset hash-key sub-key1 value1
- hgetall hash-key
  - 1) "sub-key1"
  - 2) "value1"
  - 3) "sub-key2"
  - 4) "value2"
- hdel hash-key sub-key2
- hdel hash-key sub-key2
- hget hash-key sub-key1
  - "value1"
- hgetall hash-key
  - 1) "sub-key1"
  - 2) "value1"

Sorted set in Redis

- ZSETs hold a type of key and value. The keys (called *members*) are unique, and the values (called *scores*) are limited to floating-point numbers.
- ZSETs have the unique property in Redis of being able to be accessed by member (like a HASH), but items can also be accessed by the sorted order and values of the scores.
- Commands
  - ZADD: Adds member with the given score to the ZSET
  - ZRANGE: Fetches the items in the ZSET from their positions in sorted order
  - ZRANGEBYSCORE: Fetches items in the ZSET based on a range of scores
  - ZREM: Removes the item from the ZSET, if it exists

Sorted set in Redis

- zadd zset-key 728 member1
- zadd zset-key 982 member0
- zadd zset-key 982 member0
- zrange zset-key 0 -1 withscores
  - 1) "member1"
  - 2) "728"
  - 3) "member0"
  - 4) "982„
  - (It fetches all of the items in the ZSET, which are ordered by the scores.)
- zrangebyscore zset-key 0 800 withscores
  - 1) "member1"
  - 2) "728„
  - (It fetches a subsequence of items based on their scores.)
- zrem zset-key member1
- zrem zset-key member1
- zrange zset-key 0 -1 withscores
  - 1) "member0"
  - 2) "982"

Other commands

- **KEYS pattern**
  - Returns all keys matching pattern.
  - NOTE: For example, Redis running on an entry level laptop can scan a 1 million key database in 40 milliseconds.
  - **Warning**: Don't use KEYS in your regular application code. If you're looking for a way to find keys in a subset of your keyspace, consider using SCAN or sets.
  - Supported glob-style patterns:
    - h?llo matches hello, hallo and hxllo
    - h*llo matches hllo and heeeello
    - h[ae]llo matches hello and hallo, but not hillo
    - h[^e]llo matches hallo, hbllo, ... but not hello
    - h[a-b]llo matches hallo and hbllo
  - Use \ to escape special characters if you want to match them verbatim.
  - Examples
    - redis> **KEYS *name***
    - redis> **KEYS a??**
    - redis> **KEYS ***

Other commands

- **DEL key [key ...]**
  - Removes the specified keys. A key is ignored if it does not exist.

Other commands

- **EXISTS key**
  - Returns if key exists.
  - Return value
    - 1 if the key exists.
    - 0 if the key does not exist.

Other commands

- **RENAME key newkey**
  - Renames key to newkey. It returns an error when key does not exist. If newkey already exists it is overwritten, when this happens RENAME executes an implicit DEL operation, so if the deleted key contains a very big value it may cause high latency.

Other commands

- **TYPE key**
  - Returns the string representation of the type of the value stored at key. The different types that can be returned are: string, list, set, zset, hash and stream.

SCAN cursor [MATCH pattern] [COUNT count]

- The SCAN command and the closely related commands SSCAN, HSCAN and ZSCAN are used in order to incrementally iterate over a collection of elements.
  - SCAN iterates the set of keys in the currently selected Redis database.
  - SSCAN iterates elements of Sets types.
  - HSCAN iterates fields of Hash types and their associated values.
  - ZSCAN iterates elements of Sorted Set types and their associated scores.
- Since these commands allow for incremental iteration, returning only a small number of elements per call, they can be used in production without the downside of commands like KEYS or SMEMBERS that may block the server for a long time (even several seconds) when called against big collections of keys or elements.
- However while blocking commands like SMEMBERS are able to provide all the elements that are part of a Set in a given moment, The SCAN family of commands only offer limited guarantees about the returned elements since the collection that we incrementally iterate can change during the iteration process.
- Note that SCAN, SSCAN, HSCAN and ZSCAN all work very similarly, so this documentation covers all the four commands. However an obvious difference is that in the case of SSCAN, HSCAN and ZSCAN the first argument is the name of the key holding the Set, Hash or Sorted Set value. The SCAN command does not need any key name argument as it iterates keys in the current database, so the iterated object is the database itself.

SCAN cursor [MATCH pattern] [COUNT count]

- **SCAN basic usage**
  - SCAN is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.
  - An iteration starts when the cursor is set to 0, and terminates when the cursor returned by the server is 0.
  - The **SCAN return value** is an array of two values: the first value is the new cursor to use in the next call, the second value is an array of elements.
  - Starting an iteration with a cursor value of 0, and calling SCANuntil the returned cursor is 0 again is called a **full iteration**.

SCAN cursor [MATCH pattern] [COUNT count]

- **SCAN basic usage**

```
redis 127.0.0.1:6379> scan 0
1) "17"
2) 1) "key:12„
   2) "key:8„
   3) "key:4„
   4) "key:14„
   5) "key:16„
   6) "key:17„
   7) "key:15„
   8) "key:10„
   9) "key:3„
   10) "key:7"
   11) "key:1"
```

```
redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18„
   3) "key:0„
   4) "key:2„
   5) "key:19„
   6) "key:13„
   7) "key:6„
   8) "key:9„
   9) "key:11"
```

  - In the example above, the first call uses zero as a cursor, to start the iteration. The second call uses the cursor returned by the previous call as the first element of the reply, that is, 17.
  - Since in the second call the returned cursor is 0, the server signaled to the caller that the iteration finished, and the collection was completely explored.

SCAN cursor [MATCH pattern] [COUNT count]

- **Scan guarantees**
  - The SCAN command, and the other commands in the SCAN family, are able to provide to the user a set of guarantees associated to full iterations.
  - A full iteration always retrieves all the elements that were present in the collection from the start to the end of a full iteration. This means that if a given element is inside the collection when an iteration is started, and is still there when an iteration terminates, then at some point SCANreturned it to the user.
  - A full iteration never returns any element that was NOT present in the collection from the start to the end of a full iteration. So if an element was removed before the start of an iteration, and is never added back to the collection for all the time an iteration lasts, SCAN ensures that this element will never be returned.
  - However because SCAN has very little state associated (just the cursor) it has the following drawbacks:
  - A given element may be returned multiple times. It is up to the application to handle the case of duplicated elements, for example only using the returned elements in order to perform operations that are safe when re-applied multiple times.
  - Elements that were not constantly present in the collection during a full iteration, may be returned or not: it is undefined.

SCAN cursor [MATCH pattern] [COUNT count]

- **Number of elements returned at every SCAN call**
  - SCAN family functions do not guarantee that the number of elements returned per call are in a given range. The commands are also allowed to return zero elements, and the client should not consider the iteration complete as long as the returned cursor is not zero.
  - However the number of returned elements is reasonable, that is, in practical terms SCAN may return a maximum number of elements in the order of a few tens of elements when iterating a large collection, or may return all the elements of the collection in a single call when the iterated collection is small enough to be internally represented as an encoded data structure (this happens for small sets, hashes and sorted sets).
  - However there is a way for the user to tune the order of magnitude of the number of returned elements per call using the **COUNT** option.

SCAN cursor [MATCH pattern] [COUNT count]

- **The COUNT option**
  - While SCAN does not provide guarantees about the number of elements returned at every iteration, it is possible to empirically adjust the behavior of SCAN using the **COUNT** option. Basically with COUNT the user specified the *amount of work that should be done at every call in order to retrieve elements from the collection.* This is **just a hint** for the implementation, however generally speaking this is what you could expect most of the times from the implementation.
  - The default COUNT value is 10.
  - When iterating the key space, or a Set, Hash or Sorted Set that is big enough to be represented by a hash table, assuming no **MATCH** option is used, the server will usually return *count* or a bit more than *count*elements per call.
  - When iterating Sets encoded as intsets (small sets composed of just integers), or Hashes and Sorted Sets encoded as ziplists (small hashes and sets composed of small individual values), usually all the elements are returned in the first SCAN call regardless of the COUNT value.
  - Important: **there is no need to use the same COUNT value** for every iteration. The caller is free to change the count from one iteration to the other as required, as long as the cursor passed in the next call is the one obtained in the previous call to the command.

SCAN cursor [MATCH pattern] [COUNT count]

- **The MATCH option**
  - It is possible to only iterate elements matching a given glob-style pattern, similarly to the behavior of the KEYS command that takes a pattern as only argument.
  - To do so, just append the MATCH <pattern> arguments at the end of the SCAN command (it works with all the SCAN family commands).
  - It is important to note that the **MATCH** filter is applied after elements are retrieved from the collection, just before returning data to the client. This means that if the pattern matches very little elements inside the collection, SCAN will likely return no elements in most iterations. An example is shown below:
  - scan 0 MATCH *11*

SCAN cursor [MATCH pattern] [COUNT count]

- **Terminating iterations in the middle**
  - Since there is no state server side, but the full state is captured by the cursor, the caller is free to terminate an iteration half-way without signaling this to the server in any way. An infinite number of iterations can be started and never terminated without any issue.
- **Calling SCAN with a corrupted cursor**
  - Calling SCAN with a broken, negative, out of range, or otherwise invalid cursor, will result into undefined behavior but never into a crash. What will be undefined is that the guarantees about the returned elements can no longer be ensured by the SCAN implementation.
  - The only valid cursors to use are:
  - The cursor value of 0 when starting an iteration.
  - The cursor returned by the previous call to SCAN in order to continue the iteration.

SCAN cursor [MATCH pattern] [COUNT count]

- **Guarantee of termination**
  - The SCAN algorithm is guaranteed to terminate only if the size of the iterated collection remains bounded to a given maximum size, otherwise iterating a collection that always grows may result into SCAN to never terminate a full iteration.
  - This is easy to see intuitively: if the collection grows there is more and more work to do in order to visit all the possible elements, and the ability to terminate the iteration depends on the number of calls to SCAN and its COUNT option value compared with the rate at which the collection grows.

Strings

- STRINGs are used to store three types of values:
  - Byte string values
  - Integer values
  - Floating-point values
- Integers and floats can be incremented or decremented by an arbitrary numeric value (integers turning into floats as necessary).
- Integers have ranges that are equivalent to the platform's long integer range (signed 32-bit integers on 32-bit platforms, and signed 64-bit integers on 64-bit platforms), and floats have ranges and values limited to IEEE 754 floating-point doubles.

Increment and decrement commands for Strings

- INCR key-name: Increments the value stored at the key by 1
- DECR key-name: Decrements the value stored at the key by 1
- INCRBY key-name amount: Increments the value stored at the key by the provided integer value
- DECRBY key-name amount: Decrements the value stored at the key by the provided integer value
- INCRBYFLOAT key-name amount: Increments the value stored at the key by the provided float value (available in Redis 2.6 and later)

- If you try to increment or decrement a key that doesn't exist or is an empty string, Redis will operate as though that key's value were zero. If you try to increment or decrement a key that has a value that can't be interpreted as an integer or float, you'll receive an error.

Substring manipulation commands

- APPEND key-name value: Concatenates the provided value to the string already stored at the given key
- GETRANGE key-name start end: Fetches the substring, including all characters from the start offset to the end offset, inclusive
- SETRANGE key-name offset value: Sets the substring starting at the provided offset to the given value

Substring manipulation commands

- **GETSET key value:** Atomically
  sets key to value and returns the old value stored at key. Returns an error when key exists but does not hold a string value.

  GETSET can be used together with INCR for counting with atomic reset. For example: a process may call INCR against the key mycounter every time some event occurs, but from time to time we need to get the value of the counter and reset it to zero atomically. This can be done using GETSET mycounter "0".

Substring manipulation commands

- **SETNX key value:** Set key to hold string value if key does not exist. In that case, it is equal to SET. When key already holds a value, no operation is performed. SETNX is short for "**SET** if **N**ot e**X**ists". Return value is 1 if the key was set; 0 if the key was not set.
- **STRLEN key:** Returns the length of the string value stored at key. An error is returned when key holds a non-string value. 0 is returned when key does not exist.

- **MGET key [key ...]:** Returns the values of all specified keys. For every key that does not hold a string value or does not exist, the special value nil is returned. Because of this, the operation never fails.

Lists

- LISTs allow you to push and pop items from both ends of a sequence, fetch individual items, and perform a variety of other operations that are expected of lists.
- LISTs by themselves can be great for keeping a queue of work items, recently viewed articles, or favorite contacts.
- It stores an ordered sequence of STRING values.

Some commonly used **LIST** commands

- **RPUSH key value [value ...]:** Insert all the specified values at the tail of the list stored at key. If key does not exist, it is created as empty list before performing the push operation. When key holds a value that is not a list, an error is returned.
- It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the tail of the list, from the leftmost element to the rightmost element. So for instance the command RPUSH mylist a b c will result into a list containing a as first element, b as second element and c as third element.

Some commonly used **LIST** commands

- **LPUSH key value [value ...]:** Insert all the specified values at the head of the list stored at key. If key does not exist, it is created as empty list before performing the push operations. When key holds a value that is not a list, an error is returned.
- It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the head of the list, from the leftmost element to the rightmost element. So for instance the command LPUSH mylist a b c will result into a list containing c as first element, b as second element and a as third element.

Some commonly used **LIST** commands

- **RPOP key:** Removes and returns the last element of the list stored at key.
- **LPOP key:** Removes and returns the first element of the list stored at key.
- **LINDEX key index:** Returns the element at index **index** in the list stored at key. The index is zero-based, so 0 means the first element, 1 the second element and so on. Negative indices can be used to designate elements starting at the tail of the list. Here, -1 means the last element, -2 means the penultimate and so forth. When the value at key is not a list, an error is returned.

Some commonly used **LIST** commands

- **LRANGE key start stop:** Returns the specified elements of the list stored at key. The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element and so on. These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, -1 is the last element of the list, -2 the penultimate, and so on.
  - Out of range indexes will not produce an error. If start is larger than the end of the list, an empty list is returned. If stop is larger than the actual end of the list, Redis will treat it like the last element of the list.

Some commonly used **LIST** commands

- **LTRIM key start stop:** Trim an existing list so that it will contain only the specified range of elements specified. Both start and stop are zero-based indexes, where 0 is the first element of the list (the head), 1 the next element and so on.
  - For example: LTRIM foobar 0 2 will modify the list stored at foobar so that only the first three elements of the list will remain.
  - start and end can also be negative numbers indicating offsets from the end of the list, where -1 is the last element of the list, -2 the penultimate element and so on.
  - Out of range indexes will not produce an error: if start is larger than the end of the list, or start > end, the result will be an empty list (which causes key to be removed). If end is larger than the end of the list, Redis will treat it like the last element of the list.

Some commonly used **LIST** commands

- **LINSERT key BEFORE|AFTER pivot value:** Inserts value in the list stored at key either before or after the reference value pivot. When key does not exist, it is considered an empty list and no operation is performed. An error is returned when key exists but does not hold a list value.
- **LLEN key:** Returns the length of the list stored at key. If key does not exist, it is interpreted as an empty list and 0 is returned. An error is returned when the value stored at key is not a list.
- **LPUSHX key value:** Inserts value at the head of the list stored at key, only if key already exists and holds a list. In contrary to LPUSH, no operation will be performed when key does not yet exist.
- **RPUSHX key value:** Inserts value at the tail of the list stored at key, only if key already exists and holds a list. In contrary to RPUSH, no operation will be performed when key does not yet exist.

Some commonly used **LIST** commands

- **LREM key count value:** Removes the first count occurrences of elements equal to value from the list stored at key. The count argument influences the operation in the following ways:
  - count > 0: Remove elements equal to value moving from head to tail.
  - count < 0: Remove elements equal to value moving from tail to head.
  - count = 0: Remove all elements equal to value.
- For example, LREM list -2 "hello" will remove the last two occurrences of "hello" in the list stored at list.
- Note that non-existing keys are treated like empty lists, so when key does not exist, the command will always return 0.

Some commonly used **LIST** commands

- **LSET key index value:** Sets the list element at index to value. An error is returned for out of range indexes.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- BLPOP key-name [key-name ...] timeout: Pops the leftmost item from the first non-empty LIST, or waits the timeout in seconds for an item
- BRPOP key-name [key-name ...] timeout: Pops the rightmost item from the first non-empty LIST, or waits the timeout in seconds for an item
- RPOPLPUSH source-key dest-key: Pops the rightmost item from the source and LPUSHes the item to the destination, also returning the item to the user
- BRPOPLPUSH source-key dest-key timeout: Pops the rightmost item from the source and LPUSHes the item to the destination, also returning the item to the user, and waiting up to the timeout if the source is empty

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BLPOP key [key ...] timeout:** BLPOP is a blocking list pop primitive. It is the blocking version of LPOP because it blocks the connection when there are no elements to pop from any of the given lists. An element is popped from the head of the first list that is non-empty, with the given keys being checked in the order that they are given.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BLPOP key [key ...] timeout:**
  Non-blocking behavior
  - When BLPOP is called, if at least one of the specified keys contains a non-empty list, an element is popped from the head of the list and returned to the caller together with the key it was popped from.
  - Keys are checked in the order that they are given. Let's say that the key list1 doesn't exist and list2 and list3 hold non-empty lists. Consider the following command:

    BLPOP list1 list2 list3 0
  - BLPOP guarantees to return an element from the list stored at list2 (since it is the first non empty list when checking list1, list2 and list3 in that order).

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BLPOP key [key ...] timeout:**
  Blocking behavior
  - If none of the specified keys exist, BLPOP blocks the connection until another client performs an LPUSH or RPUSH operation against one of the keys.
  - Once new data is present on one of the lists, the client returns with the name of the key unblocking it and the popped value.
  - When BLPOP causes a client to block and a non-zero timeout is specified, the client will unblock returning a nil multi-bulk value when the specified timeout has expired without a push operation against at least one of the specified keys.
  - **The timeout argument is interpreted as an integer value specifying the maximum number of seconds to block**. A timeout of zero can be used to block indefinitely.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BLPOP key [key ...] timeout:**
  Return value
  - A nil multi-bulk when no element could be popped and the timeout expired.
  - A two-element multi-bulk with the first element being the name of the key where an element was popped and the second element being the value of the popped element.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BLPOP key [key ...] timeout:**
  Reliable queues
  - When BLPOP returns an element to the client, it also removes the element from the list. This means that the element only exists in the context of the client: if the client crashes while processing the returned element, it is lost forever.
  - This can be a problem with some application where we want a more reliable messaging system. When this is the case, please check the BRPOPLPUSH command, that is a variant of BLPOP that adds the returned element to a target list before returning it to the client.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BRPOP key [key ...] timeout:** BRPOP is a blocking list pop primitive. It is the blocking version of RPOP because it blocks the connection when there are no elements to pop from any of the given lists. An element is popped from the tail of the first list that is non-empty, with the given keys being checked in the order that they are given.
  - See the BLPOP documentation for the exact semantics, since BRPOP is identical to BLPOP with the only difference being that it pops elements from the tail of a list instead of popping from the head.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **RPOPLPUSH source destination:** Atomically returns and removes the last element (tail) of the list stored at source, and pushes the element at the first element (head) of the list stored at destination.
  - For example: consider source holding the list a,b,c, and destination holding the list x,y,z. Executing RPOPLPUSH results in source holding a,b and destination holding c,x,y,z.
  - If source does not exist, the value nil is returned and no operation is performed. If source and destination are the same, the operation is equivalent to removing the last element from the list and pushing it as first element of the list, so it can be considered as a list rotation command.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **RPOPLPUSH source destination:**

  Pattern: Reliable queue

  - Redis is often used as a messaging server to implement processing of background jobs or other kinds of messaging tasks. A simple form of queue is often obtained pushing values into a list in the producer side, and waiting for this values in the consumer side using RPOP (using polling), or BRPOP if the client is better served by a blocking operation.
  - However in this context the obtained queue is not *reliable* as messages can be lost, for example in the case there is a network problem or if the consumer crashes just after the message is received but it is still to process.
  - RPOPLPUSH (or BRPOPLPUSH for the blocking variant) offers a way to avoid this problem: the consumer fetches the message and at the same time pushes it into a *processing* list. It will use the LREM command in order to remove the message from the *processing* list once the message has been processed.
  - An additional client may monitor the *processing* list for items that remain there for too much time, and will push those timed out items into the queue again if needed.

Some **LIST** commands for blocking **LIST** pops and moving items between **LIST**s

- **BRPOPLPUSH source destination timeout:** BRPOPLPUSH is the blocking variant of RPOPLPUSH. When source contains elements, this command behaves exactly like RPOPLPUSH. When used inside a MULTI/EXEC block, this command behaves exactly like RPOPLPUSH. When source is empty, Redis will block the connection until another client pushes to it or until timeout is reached. A timeout of zero can be used to block indefinitely.

Sets

- SETs hold unique items in an unordered fashion.

Some commonly used **SET** commands

- **SADD key-name item [item ...]**: Add the specified members to the set stored at key. Specified members that are already a member of this set are ignored. If key does not exist, a new set is created before adding the specified members. An error is returned when the value stored at key is not a set. Return value is the number of elements that were added to the set, not including all the elements already present into the set.
- **SREM key-name item [item ...]**: Remove the specified members from the set stored at key. Specified members that are not a member of this set are ignored. If key does not exist, it is treated as an empty set and this command returns 0. An error is returned when the value stored at key is not a set. Return value is the number of members that were removed from the set, not including non existing members.
- **SISMEMBER key-name item**: Returns if item is a member of the set stored at key.
- **SCARD key-name**: Returns the number of items in the SET

Some commonly used **SET** commands

- **SMEMBERS key-name**: Returns all the members of the set value stored at key.
- **SPOP key [count]:** Removes and returns one or more random elements from the set value store at key.
- **SMOVE source-key dest-key item**: If the item is in the source, removes the item from the source and adds it to the destination, returning if the item was moved. In every given moment the element will appear to be a member of source **or** destination for other clients. If the source set does not exist or does not contain the specified element, no operation is performed and 0 is returned. An error is returned
  if source or destination does not hold a set value.

Some commonly used **SET** commands

- **SRANDMEMBER key-name [count]**: Returns one or more random items from the SET. When count is positive, Redis will return count distinct randomly chosen items, and when count is negative, Redis will return count randomly chosen items that may not be distinct. Elements are **not removed** from the Set.
- **SSCAN key cursor [MATCH pattern] [COUNT count]**: SCAN

Operations for combining and manipulating **SET**s in Redis

- **SDIFF key-name [key-name ...]**: Returns the items in the first SET that weren't in any of the other SETs (mathematical set difference operation)
- **SDIFFSTORE dest-key key-name [key-name ...]**: This command is equal to SDIFF, but instead of returning the resulting set, it is stored in destination. If destination already exists, it is overwritten.
- **SINTER key-name [key-name ...]**: Returns the items that are in all of the SETs (mathematical set intersection operation)
- **SINTERSTORE dest-key key-name [key-name ...]**: This command is equal to SINTER, but instead of returning the resulting set, it is stored in destination. If destination already exists, it is overwritten.
- **SUNION key-name [key-name ...]**: Returns the items that are in at least one of the SETs (mathematical set union operation)
- **SUNIONSTORE dest-key key-name [key-name ...]**: This command is equal to SUNION, but instead of returning the resulting set, it is stored in destination. If destination already exists, it is overwritten.

Hashes

- HASHes in Redis allow you to store groups of key-value pairs in a single higher-level Redis key.
- Functionally, the values offer some of the same features as values in STRINGs and can be useful to group related data together.

Operations for adding and removing items from **HASH**e

- **HSET key field value:** Sets field in the hash stored at key to value. If key does not exist, a new key holding a hash is created. If field already exists in the hash, it is overwritten.
  - Return value:
    - 1 if field is a new field in the hash and value was set.
    - 0 if field already exists in the hash and the value was updated.
- **HGET key field:** Returns the value associated with field in the hash stored at key.

Operations for adding and removing items from **HASH**e

- **HMSET key field value [field value ...]:** Sets the specified fields to their respective values in the hash stored at key. This command overwrites any specified fields already existing in the hash. If key does not exist, a new key holding a hash is created.
- **HSETNX key field value:** Sets field in the hash stored at key to value, only if field does not yet exist.
  If key does not exist, a new key holding a hash is created. If field already exists, this operation has no effect.
  - Return value
    - 1 if field is a new field in the hash and value was set.
    - 0 if field already exists in the hash and no operation was performed.

Operations for adding and removing items from **HASH**e

- **HMGET key field [field ...]:** Returns the values associated with the specified fields in the hash stored at key. For every field that does not exist in the hash, a nil value is returned. Because non-existing keys are treated as empty hashes, running HMGET against a non-existing key will return a list of nil values.

Operations for adding and removing items from **HASH**e

- **HGETALL key:** Returns all fields and values of the hash stored at key. In the returned value, every field name is followed by its value, so the length of the reply is twice the size of the hash.
- **HKEYS key:** Returns all field names in the hash stored at key.
- **HVALS key:** Returns all values in the hash stored at key.

Operations for adding and removing items from **HASH**e

- **HLEN key:** Returns the number of fields contained in the hash stored at key.
- **HSTRLEN key field:** Returns the string length of the value associated with field in the hash stored at key. If the key or the field do not exist, 0 is returned.
- **HEXISTS key field:** Returns if field is an existing field in the hash stored at key.
  - Return value
    - 1 if the hash contains field.
    - 0 if the hash does not contain field, or key does not exist.

Operations for adding and removing items from **HASH**e

- **HDEL key field [field ...]:** Removes the specified fields from the hash stored at key. Specified fields that do not exist within this hash are ignored.
  If key does not exist, it is treated as an empty hash and this command returns 0.

Operations for adding and removing items from **HASH**e

- **HINCRBY key field increment:** Increments the number stored at field in the hash stored at key by increment. If key does not exist, a new key holding a hash is created. If field does not exist the value is set to 0 before the operation is performed.
- **HINCRBYFLOAT key field increment:** Increment the specified field of a hash stored at key, and representing a floating point number, by the specified increment. If the increment value is negative, the result is to have the hash field value **decremented** instead of incremented. If the field does not exist, it is set to 0 before performing the operation. An error is returned if one of the following conditions occur:
  - The field contains a value of the wrong type (not a string).
  - The current field content or the specified increment are not parsable as a double precision floating point number.

Operations for adding and removing items from **HASH**e

- **HSCAN key cursor [MATCH pattern] [COUNT count]**
  - See SCAN for HSCAN documentation.

Sorted sets

- ZSETs offer the ability to store a mapping of members to scores (similar to the keys and values of HASHes).
- These mappings allow us to manipulate the numeric scores, and fetch and scan over both members and scores based on the sorted order of the scores.
- Sorted sets are sorted by their score in an ascending way. The same element only exists a single time, no repeated elements are permitted. The score can be modified both by ZADD that will update the element score, and as a side effect, its position on the sorted set, and by ZINCRBY that can be used in order to update the score relatively to its previous value.
- The current score of an element can be retrieved using the ZSCORE command, that can also be used to verify if an element already exists or not.

Sorted sets

- While the same element can't be repeated in a sorted set since every element is unique, it is possible to add multiple different elements *having the same score*. When multiple elements have the same score, they are *ordered lexicographically* (they are still ordered by score as a first key, however, locally, all the elements with the same score are relatively ordered lexicographically).
- The lexicographic ordering used is binary, it compares strings as array of bytes.
- If the user inserts all the elements in a sorted set with the same score (for example 0), all the elements of the sorted set are sorted lexicographically, and range queries on elements are possible using the command ZRANGEBYLEX (Note: it is also possible to query sorted sets by range of scores using ZRANGEBYSCORE).

ZSET commands

- **ZADD key [NX|XX] [CH] [INCR]score member [score member ...]:** Adds all the specified members with the specified scores to the sorted set stored at key. It is possible to specify multiple score / member pairs. If a specified member is already a member of the sorted set, the score is updated and the element reinserted at the right position to ensure the correct ordering.
  - The score values should be the string representation of a double precision floating point number. +inf and -inf values are valid values as well.
  - Return value
    - The number of elements added to the sorted sets, not including elements already existing for which the score was updated.

ZSET commands

- **ZADD key [NX|XX] [CH] [INCR]score member [score member ...]:** ZADD options (Redis 3.0.2 or greater): ZADD supports a list of options, specified after the name of the key and before the first score argument. Options are:
    - **XX**: Only update elements that already exist. Never add elements.
    - **NX**: Don't update already existing elements. Always add new elements.
    - **CH**: Modify the return value from the number of new elements added, to the total number of elements changed (CH is an abbreviation of *changed*). Changed elements are **new elements added** and elements already existing for which **the score was updated**. So elements specified in the command line having the same score as they had in the past are not counted. Note: normally the return value of ZADD only counts the number of new elements added.
    - **INCR**: When this option is specified ZADD acts like ZINCRBY. Only one score-element pair can be specified in this mode. The return value will be the new score of member (a double precision floating point number), represented as string.

ZSET commands

- **ZREM key member [member ...]:** Removes the specified members from the sorted set stored at key. Non existing members are ignored.
  - Return value: The number of members removed from the sorted set, not including non existing members.
- **ZCARD key:** Returns the sorted set cardinality (number of elements) of the sorted set stored at key.
- **ZCOUNT key min max:** Returns the number of elements in the sorted set at key with a score between min and max (including elements with score equal to min or max).
- **ZLEXCOUNT key min max:** When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command returns the number of elements in the sorted set at key with a value between min and max. The min and max arguments have the same meaning as described for ZRANGEBYLEX.

ZSET commands

- **ZINCRBY key increment member:** Increments the score of member in the sorted set stored at key by increment. If member does not exist in the sorted set, it is added with increment as its score (as if its previous score was 0.0). If key does not exist, a new sorted set with the specified member as its sole member is created.
  - The score value should be the string representation of a numeric value, and accepts double precision floating point numbers. It is possible to provide a negative value to decrement the score.
- **ZSCORE key member:** Returns the score of member in the sorted set at key. If member does not exist in the sorted set, or key does not exist, nil is returned.

ZSET commands

- **ZRANK key member:** Returns the rank of member in the sorted set stored at key, with the scores ordered from low to high. The rank (or index) is 0-based, which means that the member with the lowest score has rank 0. If member does not exist in the sorted set or key does not exist, the return value is nil.
- **ZREVRANK key member:** Returns the rank of member in the sorted set stored at key, with the scores ordered from high to low. The rank (or index) is 0-based, which means that the member with the highest score has rank 0.

ZSET commands

- **ZRANGE key start stop [WITHSCORES]:** Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with equal score.
  - Both start and stop are zero-based indexes, where 0 is the first element, 1 is the next element and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with -1 being the last element of the sorted set, -2 the penultimate element and so on.
  - start and stop are **inclusive ranges**, so for example ZRANGE myzset 0 1 will return both the first and the second element of the sorted set.
  - Out of range indexes will not produce an error. If start is larger than the largest index in the sorted set, or start > stop, an empty list is returned. If stop is larger than the end of the sorted set Redis will treat it like it is the last element of the sorted set.
  - It is possible to pass the WITHSCORES option in order to return the scores of the elements together with the elements.
- **ZREVRANGE key start stop [WITHSCORES]:** Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the highest to the lowest score. Descending lexicographical order is used for elements with equal score.

ZSET commands

- **ZRANGEBYLEX key min max:** When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command returns all the elements in the sorted set at key with a value between min and max.
  - If the elements in the sorted set have different scores, the returned elements are unspecified.
  - The elements are considered to be ordered from lower to higher strings as compared byte-by-byte using the memcmp() C function. Longer strings are considered greater than shorter strings if the common part is identical.
  - Valid *start* and *stop* must start with (or [, in order to specify if the range item is respectively exclusive or inclusive. The special values of + or - for *start* and *stop* have the special meaning or positively infinite and negatively infinite strings, so for instance the command **ZRANGEBYLEX myzset - +** is guaranteed to return all the elements in the sorted set, if all the elements have the same score.
  - Details on strings comparison: Strings are compared as binary array of bytes. Because of how the ASCII character set is specified, this means that usually this also have the effect of comparing normal ASCII characters in an obvious dictionary way. However this is not true if non plain ASCII strings are used (for example utf8 strings).

ZSET commands

- **ZREVRANGEBYLEX key max min [LIMIT offset count]:** When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command returns all the elements in the sorted set at key with a value between max and min.
  - Apart from the reversed ordering, ZREVRANGEBYLEX is similar to ZRANGEBYLEX.

ZSET commands

- **ZRANGEBYSCORE key min max [WITHSCORES]**: Returns all the elements in the sorted set at key with a score between min and max (including elements with score equal to min or max). The elements are considered to be ordered from low to high scores.
  - The elements having the same score are returned in lexicographical order (this follows from a property of the sorted set implementation in Redis and does not involve further computation).
  - The optional WITHSCORES argument makes the command return both the element and its score, instead of the element alone.
  - Exclusive intervals and infinity: min and max can be -inf and +inf, so that you are not required to know the highest or lowest score in the sorted set to get all elements from or up to a certain score.
  - By default, the interval specified by min and max is closed (inclusive). It is possible to specify an open interval (exclusive) by prefixing the score with the character (. For example:
    - ZRANGEBYSCORE zset (1 5
    - ZRANGEBYSCORE zset (5 (10

ZSET commands

- **ZREVRANGEBYSCORE key max min [WITHSCORES]:**Returns all the elements in the sorted set at key with a score between max and min (including elements with score equal to max or min). In contrary to the default ordering of sorted sets, for this command the elements are considered to be ordered from high to low scores. The elements having the same score are returned in reverse lexicographical order.
  - Apart from the reversed ordering, ZREVRANGEBYSCORE is similar to ZRANGEBYSCORE.

ZSET commands

- **ZREMRANGEBYLEX key min max:** When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command removes all elements in the sorted set stored at key between the lexicographical range specified by min and max. The meaning of min and max are the same of the ZRANGEBYLEX command. Similarly, this command actually returns the same elements that ZRANGEBYLEX would return if called with the same min and max arguments.
- **ZREMRANGEBYRANK key start stop:** Removes all elements in the sorted set stored at key with rank between start and stop. Both start and stop are 0 -based indexes with 0 being the element with the lowest score. These indexes can be negative numbers, where they indicate offsets starting at the element with the highest score. For example: -1 is the element with the highest score, -2 the element with the second highest score and so forth.
- **ZREMRANGEBYSCORE key min max:** Removes all elements in the sorted set stored at key with a score between min and max(inclusive).

ZSET commands

- **ZPOPMAX key [count]:** Removes and returns up to count members with the highest scores in the sorted set stored at key.
  - When count unspecified, the default value for count is 1. Specifying a count value that is higher than the sorted set's cardinality will not produce an error. When returning multiple elements, the one with the highest score will be the first, followed by the elements with lower scores.
- **ZPOPMIN key [count]:** Removes and returns up to count members with the lowest scores in the sorted set stored at key.

ZSET commands

- **BZPOPMIN key [key ...] timeout**
  - It is the blocking variant of the sorted set ZPOPMIN primitive.
  - It is the blocking version because it blocks the connection when there are no members to pop from any of the given sorted sets. A member with the lowest score is popped from first sorted set that is non-empty, with the given keys being checked in the order that they are given.
  - The timeout argument is interpreted as an integer value specifying the maximum number of seconds to block. A timeout of zero can be used to block indefinitely.
  - See the BLPOP documentation for the exact semantics, since BZPOPMIN is identical to BLPOP with the only difference being the data structure being popped from.
  - Return value
    - A nil multi-bulk when no element could be popped and the timeout expired.
    - A three-element multi-bulk with the first element being the name of the key where a member was popped, the second element being the score of the popped member, and the third element being the popped member itself.

ZSET commands

- **BZPOPMAX key [key ...] timeout**
- BZPOPMAX is the blocking variant of the sorted set ZPOPMAX primitive.
- See the BZPOPMIN documentation for the exact semantics, since BZPOPMAX is identical to BZPOPMIN with the only difference being that it pops members with the highest scores instead of popping the ones with the lowest scores.

ZSET commands

- **ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]:**
  Computes the union of numkeys sorted sets given by the specified keys, and stores the result in destination. It is mandatory to provide the number of input keys (numkeys) before passing the input keys and the other (optional) arguments.
  - By default, the resulting score of an element is the sum of its scores in the sorted sets where it exists.
  - If destination already exists, it is overwritten.
- Using the WEIGHTS option, it is possible to specify a multiplication factor for each input sorted set. This means that the score of every element in every input sorted set is multiplied by this factor before being passed to the aggregation function. When WEIGHTS is not given, the multiplication factors default to 1.
- With the AGGREGATE option, it is possible to specify how the results of the union are aggregated. This option defaults to SUM, where the score of an element is summed across the inputs where it exists. When this option is set to either MIN or MAX, the resulting set will contain the minimum or maximum score of an element across the inputs where it exists.

ZSET commands

- **ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]:**
  - Using the WEIGHTS option, it is possible to specify a multiplication factor for each input sorted set. This means that the score of every element in every input sorted set is multiplied by this factor before being passed to the aggregation function. When WEIGHTS is not given, the multiplication factors default to 1.
  - With the AGGREGATE option, it is possible to specify how the results of the union are aggregated. This option defaults to SUM, where the score of an element is summed across the inputs where it exists. When this option is set to either MIN or MAX, the resulting set will contain the minimum or maximum score of an element across the inputs where it exists.

ZSET commands

- **ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]:**
  Computes the intersection of numkeys sorted sets given by the specified keys, and stores the result
  in destination. It is mandatory to provide the number of input keys (numkeys) before passing the input keys and the other (optional) arguments.
  - By default, the resulting score of an element is the sum of its scores in the sorted sets where it exists. Because intersection requires an element to be a member of every given sorted set, this results in the score of every element in the resulting sorted set to be equal to the number of input sorted sets.
  - If destination already exists, it is overwritten.

ZSET commands

- **ZSCAN key cursor [MATCH pattern] [COUNT count]**
  - See SCAN for ZSCAN documentation.

Expiring keys

- When writing data to Redis, sometimes the data is only going to be useful for a short period of time. We can manually delete this data after that time has elapsed, or we can have Redis automatically delete the data itself by using key expiration.
- When writing data into Redis, there may be a point at which data is no longer needed. We can remove the data explicitly with DEL, or if we want to remove an entire key after a specified timeout, we can use what's known as *expiration*. When we say that a key has a *time to live*, or that it'll *expire* at a given time, we mean that Redis will automatically delete the key when its expiration time has arrived.
- Having keys that will expire after a certain amount of time can be useful to handle the cleanup of cached data.
- For the types of structures that are used; few of the commands we use offer the ability to set the expiration time of a key automatically. And with containers (LISTs, SETs, HASHes, and ZSETs), we can only expire entire keys, not individual items (this is also why we use ZSETs with timestamps in a few places).

Commands for handling expiration

- **PERSIST key:** Remove the existing timeout on key, turning the key from *volatile* (a key with an expire set) to *persistent* (a key that will never expire as no timeout is associated).
  - Return value is 1 if the timeout was removed, 0 if key does not exist or does not have an associated timeout.
- **TTL key:** Returns the remaining time to live of a key that has a timeout. This introspection capability allows a Redis client to check how many seconds a given key will continue to be part of the dataset.
  - Return value is an integer value, which is TTL in seconds, or a negative value in order to signal an error. The command returns -2 if the key does not exist. The command returns -1 if the key exists but has no associated expire.
- **PTTL key:** Like TTL this command returns the remaining time to live of a key that has an expire set, with the sole difference that TTL returns the amount of remaining time in seconds while PTTL returns it in milliseconds.

Commands for handling expiration

- **EXPIRE key seconds:** Set a timeout on key. After the timeout has expired, the key will automatically be deleted. A key with an associated timeout is often said to be *volatile* in Redis terminology.
  - Note that calling EXPIRE/PEXPIRE with a non-positive timeout or EXPIREAT/PEXPIREAT with a time in the past will result in the key being deleted rather than expired (accordingly, the emitted key event will be del, not expired).
  - Refreshing expires: It is possible to call EXPIRE using as argument a key that already has an existing expire set. In this case the time to live of a key is *updated* to the new value. There are many useful applications for this, an example is documented in the *Navigation session* pattern section below.
  - Return value:
    - 1 if the timeout was set.
    - 0 if key does not exist.

Commands for handling expiration

- **EXPIRE key seconds:**
  - The timeout will only be cleared by commands that delete or overwrite the contents of the key, including DEL, SET, GETSET and all the *STORE commands. This means that all the operations that conceptually *alter* the value stored at the key without replacing it with a new one will leave the timeout untouched. For instance, incrementing the value of a key with INCR, pushing a new value into a list with LPUSH, or altering the field value of a hash with HSET are all operations that will leave the timeout untouched.
  - The timeout can also be cleared, turning the key back into a persistent key, using the PERSIST command.
  - If a key is renamed with RENAME, the associated time to live is transferred to the new key name.
  - If a key is overwritten by RENAME, like in the case of an existing key Key_A that is overwritten by a call like RENAME Key_B Key_A, it does not matter if the original Key_A had a timeout associated or not, the new key Key_A will inherit all the characteristics of Key_B.

Commands for handling expiration

- **EXPIREAT key timestamp**
  - EXPIREAT has the same effect and semantic as EXPIRE, but instead of specifying the number of seconds representing the TTL (time to live), it takes an absolute Unix timestamp (seconds since January 1, 1970). A timestamp in the past will delete the key immediately.
- **PEXPIRE key milliseconds:** This command works exactly like EXPIRE but the time to live of the key is specified in milliseconds instead of seconds.
- **PEXPIREAT key milliseconds-timestamp:** PEXPIREAT has the same effect and semantic as EXPIREAT, but the Unix time at which the key will expire is specified in milliseconds instead of seconds.

Commands for handling expiration

- **SETEX key seconds value:** Set key to hold the string value and set key to timeout after a given number of seconds. This command is equivalent to executing the following commands:

  SET mykey value

  EXPIRE mykey seconds

  - SETEX is atomic, and can be reproduced by using the previous two commands inside an MULTI/ EXEC block. It is provided as a faster alternative to the given sequence of operations, because this operation is very common when Redis is used as a cache.
- **PSETEX key milliseconds value:** PSETEX works exactly like SETEX with the sole difference that the expire time is specified in milliseconds instead of seconds.
  -

Commands for handling expiration

- **SET key value [expiration EX seconds|PX milliseconds][NX|XX]**
- Set key to hold the string value. If key already holds a value, it is overwritten, regardless of its type. Any previous time to live associated with the key is discarded on successful SET operation.
- SET supports a set of options that modify its behavior:
  - EX *seconds* -- Set the specified expire time, in seconds.
  - PX *milliseconds* -- Set the specified expire time, in milliseconds.
  - NX -- Only set the key if it does not already exist.
  - XX -- Only set the key if it already exist.
-

SET and locking

- The command **SET resource-name anystring NX EX max-lock-time** is a simple way to implement a **locking system with Redis**.
- A client can acquire the lock if the above command returns OK (or retry after some time if the command returns Nil), and remove the lock just using DEL.
- The lock will be auto-released after the expire time is reached.

SCAN cursor [MATCH pattern] [COUNT count]

- The SCAN command and the closely related commands SSCAN, HSCAN and ZSCAN are used in order to incrementally iterate over a collection of elements.
  - SCAN iterates the set of keys in the currently selected Redis database.
  - SSCAN iterates elements of Sets types.
  - HSCAN iterates fields of Hash types and their associated values.
  - ZSCAN iterates elements of Sorted Set types and their associated scores.
- Since these commands allow for incremental iteration, returning only a small number of elements per call, they can be used in production without the downside of commands like KEYS or SMEMBERS that may block the server for a long time (even several seconds) when called against big collections of keys or elements.
- However while blocking commands like SMEMBERS are able to provide all the elements that are part of a Set in a given moment, The SCAN family of commands only offer limited guarantees about the returned elements since the collection that we incrementally iterate can change during the iteration process.
- Note that SCAN, SSCAN, HSCAN and ZSCAN all work very similarly, so this documentation covers all the four commands. However an obvious difference is that in the case of SSCAN, HSCAN and ZSCAN the first argument is the name of the key holding the Set, Hash or Sorted Set value. The SCAN command does not need any key name argument as it iterates keys in the current database, so the iterated object is the database itself.

SCAN cursor [MATCH pattern] [COUNT count]

- **SCAN basic usage**
  - SCAN is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.
  - An iteration starts when the cursor is set to 0, and terminates when the cursor returned by the server is 0.
  - The **SCAN return value** is an array of two values: the first value is the new cursor to use in the next call, the second value is an array of elements.
  - Starting an iteration with a cursor value of 0, and calling SCANuntil the returned cursor is 0 again is called a **full iteration**.

SCAN cursor [MATCH pattern] [COUNT count]

- **SCAN basic usage**

```
redis 127.0.0.1:6379> scan 0
1) "17"
2) 1) "key:12„
   2) "key:8„
   3) "key:4„
   4) "key:14„
   5) "key:16„
   6) "key:17„
   7) "key:15„
   8) "key:10„
   9) "key:3„
   10) "key:7"
   11) "key:1"
```

```
redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18„
   3) "key:0„
   4) "key:2„
   5) "key:19„
   6) "key:13„
   7) "key:6„
   8) "key:9„
   9) "key:11"
```

  - In the example above, the first call uses zero as a cursor, to start the iteration. The second call uses the cursor returned by the previous call as the first element of the reply, that is, 17.
  - Since in the second call the returned cursor is 0, the server signaled to the caller that the iteration finished, and the collection was completely explored.

SCAN cursor [MATCH pattern] [COUNT count]

- **Scan guarantees**
  - The SCAN command, and the other commands in the SCAN family, are able to provide to the user a set of guarantees associated to full iterations.
  - A full iteration always retrieves all the elements that were present in the collection from the start to the end of a full iteration. This means that if a given element is inside the collection when an iteration is started, and is still there when an iteration terminates, then at some point SCANreturned it to the user.
  - A full iteration never returns any element that was NOT present in the collection from the start to the end of a full iteration. So if an element was removed before the start of an iteration, and is never added back to the collection for all the time an iteration lasts, SCAN ensures that this element will never be returned.
  - However because SCAN has very little state associated (just the cursor) it has the following drawbacks:
  - A given element may be returned multiple times. It is up to the application to handle the case of duplicated elements, for example only using the returned elements in order to perform operations that are safe when re-applied multiple times.
  - Elements that were not constantly present in the collection during a full iteration, may be returned or not: it is undefined.

SCAN cursor [MATCH pattern] [COUNT count]

- **Number of elements returned at every SCAN call**
  - SCAN family functions do not guarantee that the number of elements returned per call are in a given range. The commands are also allowed to return zero elements, and the client should not consider the iteration complete as long as the returned cursor is not zero.
  - However the number of returned elements is reasonable, that is, in practical terms SCAN may return a maximum number of elements in the order of a few tens of elements when iterating a large collection, or may return all the elements of the collection in a single call when the iterated collection is small enough to be internally represented as an encoded data structure (this happens for small sets, hashes and sorted sets).
  - However there is a way for the user to tune the order of magnitude of the number of returned elements per call using the **COUNT** option.

SCAN cursor [MATCH pattern] [COUNT count]

- **The COUNT option**
  - While SCAN does not provide guarantees about the number of elements returned at every iteration, it is possible to empirically adjust the behavior of SCAN using the **COUNT** option. Basically with COUNT the user specified the *amount of work that should be done at every call in order to retrieve elements from the collection.* This is **just a hint** for the implementation, however generally speaking this is what you could expect most of the times from the implementation.
  - The default COUNT value is 10.
  - When iterating the key space, or a Set, Hash or Sorted Set that is big enough to be represented by a hash table, assuming no **MATCH** option is used, the server will usually return *count* or a bit more than *count*elements per call.
  - When iterating Sets encoded as intsets (small sets composed of just integers), or Hashes and Sorted Sets encoded as ziplists (small hashes and sets composed of small individual values), usually all the elements are returned in the first SCAN call regardless of the COUNT value.
  - Important: **there is no need to use the same COUNT value** for every iteration. The caller is free to change the count from one iteration to the other as required, as long as the cursor passed in the next call is the one obtained in the previous call to the command.

SCAN cursor [MATCH pattern] [COUNT count]

- **The MATCH option**
  - It is possible to only iterate elements matching a given glob-style pattern, similarly to the behavior of the KEYS command that takes a pattern as only argument.
  - To do so, just append the MATCH <pattern> arguments at the end of the SCAN command (it works with all the SCAN family commands).
  - It is important to note that the **MATCH** filter is applied after elements are retrieved from the collection, just before returning data to the client. This means that if the pattern matches very little elements inside the collection, SCAN will likely return no elements in most iterations. An example is shown below:
  - scan 0 MATCH *11*

SCAN cursor [MATCH pattern] [COUNT count]

- **Terminating iterations in the middle**
  - Since there is no state server side, but the full state is captured by the cursor, the caller is free to terminate an iteration half-way without signaling this to the server in any way. An infinite number of iterations can be started and never terminated without any issue.
- **Calling SCAN with a corrupted cursor**
  - Calling SCAN with a broken, negative, out of range, or otherwise invalid cursor, will result into undefined behavior but never into a crash. What will be undefined is that the guarantees about the returned elements can no longer be ensured by the SCAN implementation.
  - The only valid cursors to use are:
  - The cursor value of 0 when starting an iteration.
  - The cursor returned by the previous call to SCAN in order to continue the iteration.

SCAN cursor [MATCH pattern] [COUNT count]

- **Guarantee of termination**
  - The SCAN algorithm is guaranteed to terminate only if the size of the iterated collection remains bounded to a given maximum size, otherwise iterating a collection that always grows may result into SCAN to never terminate a full iteration.
  - This is easy to see intuitively: if the collection grows there is more and more work to do in order to visit all the possible elements, and the ability to terminate the iteration depends on the number of calls to SCAN and its COUNT option value compared with the rate at which the collection grows.

Publish/subscribe

- The concept of publish/subscribe, also known as pub/sub, is characterized by listeners *subscribing* to channels, with publishers sending binary string messages to channels.
- Anyone listening to a given channel will receive all messages sent to that channel while they're connected and listening.
- You can think of it like a radio station, where subscribers can listen to multiple radio stations at the same time, and publishers can send messages on any radio station.

Commands for handling pub/sub

- SUBSCRIBE channel [channel ...]: Subscribes to the given channels
- UNSUBSCRIBE [channel [channel ...]]: Unsubscribes from the provided channels, or unsubscribes all channels if no channel is given
- PUBLISH channel message: Publishes a message to the given channel
- PSUBSCRIBE pattern [pattern ...]: Subscribes to messages broadcast to channels that match the given pattern
- PUNSUBSCRIBE [pattern [pattern ...]]: Unsubscribes from the provided patterns, or unsubscribes from all subscribed patterns if none are given

PUBSUB subcommand [argument [argument ...]]

The PUBSUB command is an introspection command that allows to inspect the state of the Pub/Sub subsystem. It is composed of subcommands that are documented separately. The general form is:

- PUBSUB <subcommand> ... args ...
- **PUBSUB CHANNELS [pattern]**
  - Lists the currently *active channels*. An active channel is a Pub/Sub channel with one or more subscribers (not including clients subscribed to patterns).
- **PUBSUB NUMSUB [channel-1 ... channel-N]**
  - Returns the number of subscribers (not counting clients subscribed to patterns) for the specified channels. The format is channel, count, channel, count, ..., so the list is flat. The order in which the channels are listed is the same as the order of the channels specified in the command call.
- **PUBSUB NUMPAT**
  - Returns the number of subscriptions to patterns (that are performed using the PSUBSCRIBEcommand). Note that this is not just the count of clients subscribed to patterns but the total number of patterns all the clients are subscribed to.

Sorting

- **SORT key [BY pattern]**
  [LIMIT offset count]
  [GET pattern [GET pattern ...]]
  [ASC|DESC]
  [ALPHA][STORE destination]:
  Returns or stores the elements contained in the list, set or sorted set at key. By default, sorting is numeric and elements are compared by their value interpreted as double precision floating point number.
- SORT mylist
- SORT mylist DESC

Sorting

- **SORT key [BY pattern]**
  [LIMIT offset count]
  [GET pattern [GET pattern ...]]
  [ASC|DESC]
  [ALPHA][STORE destination]:
- When mylist contains string values and you want to sort them lexicographically, use the ALPHA modifier:

  SORT mylist ALPHA
- The number of returned elements can be limited using the LIMIT modifier. This modifier takes the offset argument, specifying the number of elements to skip and the count argument, specifying the number of elements to return from starting at offset. The following example will return 10 elements of the sorted version of mylist, starting at element 0 (offset is zero-based):

  SORT mylist LIMIT 0 10
- Almost all modifiers can be used together. The following example will return the first 5 elements, lexicographically sorted in descending order:

  SORT mylist LIMIT 0 5 ALPHA DESC

Sorting by external keys

- Sometimes you want to sort elements using external keys as weights to compare instead of comparing the actual elements in the list, set or sorted set. Let's say the list mylist contains the elements 1, 2 and 3 representing unique IDs of objects stored in object_1, object_2 and object_3. When these objects have associated weights stored in weight_1, weight_2 and weight_3, SORT can be instructed to use these weights to sort mylist with the following statement:

  SORT mylist BY weight_*

- The BY option takes a pattern (equal to weight_* in this example) that is used to generate the keys that are used for sorting. These key names are obtained substituting the first occurrence of * with the actual value of the element in the list (1, 2 and 3 in this example).

Skip sorting the elements

- The BY option can also take a non-existent key, which causes SORT to skip the sorting operation. This is useful if you want to retrieve external keys (see the GET option below) without the overhead of sorting.
- SORT mylist BY nosort

Retrieving external keys

- Our previous example returns just the sorted IDs. In some cases, it is more useful to get the actual objects instead of their IDs (object_1, object_2 and object_3). Retrieving external keys based on the elements in a list, set or sorted set can be done with the following command:

  SORT mylist BY weight_* GET object_*

- The GET option can be used multiple times in order to get more keys for every element of the original list, set or sorted set.
- It is also possible to GET the element itself using the special pattern #:
- SORT mylist BY weight_* GET object_* GET #

Storing the result of a SORT operation

- By default, SORT returns the sorted elements to the client. With the STORE option, the result will be stored as a list at the specified key instead of being returned to the client.

  SORT mylist BY weight_* STORE resultkey

- An interesting pattern using SORT ... STORE consists in associating an EXPIRE timeout to the resulting key so that in applications where the result of a SORT operation can be cached for some time. Other clients will use the cached list instead of calling SORT for every request. When the key will timeout, an updated version of the cache can be created by calling SORT ... STORE again.
- Note that for correctly implementing this pattern it is important to avoid multiple clients rebuilding the cache at the same time. Some kind of locking is needed here (for instance using SETNX).

Using hashes in BY and GET

- It is possible to use BY and GET options against hash fields with the following syntax:

  SORT mylist BY weight_*->fieldname
  GET object_*->fieldname

- The string -> is used to separate the key name from the hash field name. The key is substituted as documented above, and the hash stored at the resulting key is accessed to retrieve the specified hash field.

Basic Redis transactions

- Redis has five commands that help us operate on multiple keys without interruption: WATCH, **MULTI**, **EXEC**, UNWATCH, and DISCARD.
- A basic transaction involving MULTI and EXEC is meant to provide the opportunity for one client to execute multiple commands A, B, C, ... without other clients being able to interrupt them.
- To perform a transaction in Redis, we first call MULTI, followed by any sequence of commands we intend to execute, followed by EXEC. When seeing MULTI, Redis will queue up commands from that same connection until it sees an EXEC, at which point Redis will execute the queued commands sequentially without interruption.

Basic Redis transactions

- **MULTI:** Marks the start of a transaction block. Subsequent commands will be queued for atomic execution using EXEC.
- **EXEC:** Executes all previously queued commands in a transaction and restores the connection state to normal.
  - When using WATCH, EXEC will execute commands only if the watched keys were not modified, allowing for a check-and-set mechanism.
  - Return value: Array reply: each element being the reply to each of the commands in the atomic transaction. When using WATCH, EXEC can return a Null reply if the execution was aborted.

Basic Redis transactions

- **DISCARD:** Flushes all previously queued commands in a transaction and restores the connection state to normal. If WATCH was used, DISCARD unwatches all keys watched by the connection.
- **WATCH key [key ...]:** Marks the given keys to be watched for conditional execution of a transaction.
- **UNWATCH:** Flushes all the previously watched keys for a transaction. If you call EXEC or DISCARD, there's no need to manually call UNWATCH.

Transaction

- All the commands in a transaction are serialized and executed sequentially. It can never happen that a request issued by another client is served **in the middle** of the execution of a Redis transaction. This guarantees that the commands are executed as a single ***isolated*** operation.
- Either all of the commands or none are processed, so a Redis transaction is also ***atomic***. The EXEC command triggers the execution of all the commands in the transaction, so if a client loses the connection to the server in the context of a transaction before calling the MULTI command none of the operations are performed, instead if the EXEC command is called, all the operations are performed.
  - When using the append-only file Redis makes sure to use a single write(2) syscall to write the transaction on disk. However if the Redis server crashes or is killed by the system administrator in some hard way it is possible that only a partial number of operations are registered. Redis will detect this condition at restart, and will exit with an error. Using the redis-check-aof tool it is possible to fix the append only file that will remove the partial transaction so that the server can start again

Errors inside a transaction

- During a transaction it is possible to encounter two kind of command errors:
  - A command may fail to be queued, so there may be an error ***before*** **EXEC** is called. For instance the command may be syntactically wrong (wrong number of arguments, wrong command name, ...), or there may be some critical condition like an out of memory condition (if the server is configured to have a memory limit using the maxmemorydirective).
  - A command may fail ***after*** **EXEC** is called, for instance since we performed an operation against a key with the wrong value (like calling a list operation against a string value).
- Clients used to sense the first kind of errors, happening **before the EXEC call**, by checking the return value of the queued command: if the command replies with QUEUED it was queued correctly, otherwise Redis returns an error. **If there is an error while queueing a command, most clients will abort the transaction discarding it.**
- When Redis executes the commands, it's important to note that **even when a command fails, all the other commands in the queue are processed** – Redis will *not* stop the processing of commands. ()

Optimistic locking using check-and-set

- WATCH is used to provide a check-and-set (CAS) behavior to Redis transactions.
- WATCHed keys are monitored in order to detect changes against them. If at least one watched key is modified before the EXEC command, the whole transaction aborts, and EXEC returns a Null reply to notify that the transaction failed.

- For example, imagine we have the need to atomically increment the value of a key by 1 (let's suppose Redis doesn't have INCR).
- The first try may be the following:
- val = GET mykey
- val = val + 1
- SET mykey $val

Optimistic locking using check-and-set

- This will work reliably only if we have a single client performing the operation in a given time. If **multiple clients** try to increment the key at about the same time there will be a **race condition**. For instance, client A and B will read the old value, for instance, 10. The value will be incremented to 11 by both the clients, and finally SET as the value of the key. So the final value will be 11 instead of 12.
- Thanks to WATCH we are able to model the problem very well:
- WATCH mykey
- val = GET mykey
- val = val + 1
- MULTI
- SET mykey $val
- EXEC

Optimistic locking using check-and-set

- Using the above code, if there are race conditions and another client modifies the result of val in the time between our call to WATCH and our call to EXEC, the transaction will fail.
- We just have to repeat the operation hoping this time we'll not get a new race. This form of locking is called *optimistic locking* and is a very powerful form of locking. In many use cases, multiple clients will be accessing different keys, so collisions are unlikely – usually there's no need to repeat the operation.

WATCH explained

- WATCH is a command that will make the EXEC conditional: we are asking Redis to perform the transaction only if none of the WATCHed keys were modified. Otherwise the transaction is not entered at all. (Note that if you WATCH a volatile key and Redis expires the key after you WATCHed it, EXEC will still work.)
- WATCH can be called multiple times. Simply all the WATCH calls will have the effects to watch for changes starting from the call, up to the moment EXEC is called. You can also send any number of keys to a single WATCH call.
- When EXEC is called, all keys are UNWATCHed, regardless of whether the transaction was aborted or not. Also when a client connection is closed, everything gets UNWATCHed.
- It is also possible to use the UNWATCH command (without arguments) in order to flush all the watched keys. Sometimes this is useful as we optimistically lock a few keys, since possibly we need to perform a transaction to alter those keys, but after reading the current content of the keys we don't want to proceed. When this happens we just call UNWATCH so that the connection can already be used freely for new transactions.

Using pipelining to speedup Redis queries

Request/Response protocols and RTT

- Redis is a TCP server using the client-server model and what is called a *Request/Response* protocol. This means that usually a request is accomplished with the following steps:
  - The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
  - The server processes the command and sends the response back to the client.
- Clients and Servers are connected via a networking link. Such a link can be very fast (a loopback interface) or very slow (a connection established over the Internet with many hops between the two hosts). Whatever the network latency is, there is a time for the packets to travel from the client to the server, and back from the server to the client to carry the reply.
- This time is called RTT (Round Trip Time). It is very easy to see how this can affect the performances when a client needs to perform many requests in a row (for instance adding many elements to the same list, or populating a database with many keys). For instance if the RTT time is 250 milliseconds (in the case of a very slow link over the Internet), even if the server is able to process 100k requests per second, we'll be able to process at max four requests per second.
- Fortunately there is a way to improve this use case.

Using pipelining to speedup Redis queries

Redis Pipelining

- A Request/Response server can be implemented so that it is able to process new requests even if the client didn't already read the old responses. This way it is possible to send *multiple commands* to the server without waiting for the replies at all, and finally read the replies in a single step.
- This is called pipelining, and is a technique widely in use since many decades. For instance many POP3 protocol implementations already supported this feature, dramatically speeding up the process of downloading new emails from the server.

Redis-py

- https://redis-py.readthedocs.io/en/latest/

- #!/usr/bin/env python3

- # step 1: import the redis-py client package
- import redis

- # step 2: define our connection information for Redis
- redis_host = *"192.168.0.101"*
- redis_port = 6379
- redis_password = *"student"*

- # step 3: create the Redis Connection object
- # The decode_repsonses flag here directs the client to convert the responses from Redis into Python strings
- # using the default encoding utf-8. This is client specific.
- r = redis.StrictRedis(host=redis_host, port=redis_port, password=redis_password, decode_responses=True)
-
- r.set(*"msg:hello", "Hello Redis!!!")*
- msg = r.get(*"msg:hello")*
- print(msg)
- r.hmset(*"hashp",{"apple":1, })*

Redis replication

Replication

- At the base of Redis replication there is a very simple to use and configure *leader follower* (master-slave) replication: it allows replica Redis instances to be exact copies of master instances. The replica will automatically reconnect to the master every time the link breaks, and will attempt to be an exact copy of it *regardless* of what happens to the master.

Replication

- This system works using three main mechanisms:
  - When a master and a replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica, in order to replicate the effects on the dataset happening in the master side due to: client writes, keys expired or evicted, any other action changing the master dataset.
  - When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization: it means that it will try to just obtain the part of the stream of commands it missed during the disconnection.
  - When a partial resynchronization is not possible, the replica will ask for a full resynchronization. This will involve a more complex process in which the master needs to create a snapshot of all its data, send it to the replica, and then continue sending the stream of commands as the dataset changes.

Replication

- Redis uses by default asynchronous replication, which being low latency and high performance, is the natural replication mode for the vast majority of Redis use cases.
- However Redis replicas asynchronously acknowledge the amount of data they received periodically with the master.
- So the master does not wait every time for a command to be processed by the replicas, however it knows, if needed, what replica already processed what command. This allows to have optional synchronous replication.

Replication

- Synchronous replication of certain data can be requested by the clients using the WAIT command.
  - However WAIT is only able to ensure that there are the specified number of acknowledged copies in the other Redis instances, it does not turn a set of Redis instances into a CP system with strong consistency: acknowledged writes can still be lost during a failover, depending on the exact configuration of the Redis persistence. However with WAIT the probability of losing a write after a failure event is greatly reduced to certain hard to trigger failure modes.

Replication

- The following are some very important facts about Redis replication:
  - Redis uses asynchronous replication, with asynchronous replica-to-master acknowledges of the amount of data processed.
  - **A master can have multiple replicas.**
  - Replicas are able to accept connections from other replicas. Aside from connecting a number of replicas to the same master, replicas can also be connected to other replicas in a **cascading-like structure**. Since Redis 4.0, all the sub-replicas will receive exactly the same replication stream from the master.
  - Redis replication is non-blocking on the master side. This means that **the master will continue to handle queries** when one or more replicas perform the initial synchronization or a partial resynchronization.

Replication

- The following are some very important facts about Redis replication:
  - Replication is also largely non-blocking on the replica side. While the replica is performing the initial synchronization, it can handle queries using the old version of the dataset, assuming you configured Redis to do so in redis.conf. Otherwise, you can configure Redis replicas to return an error to clients if the replication stream is down. However, after the initial sync, the old dataset must be deleted and the new one must be loaded. The replica will block incoming connections during this brief window (that can be as long as many seconds for very large datasets). Since Redis 4.0 it is possible to configure Redis so that the deletion of the old data set happens in a different thread, however loading the new initial dataset will still happen in the main thread and block the replica.

Replication

- The following are some very important facts about Redis replication:
  - Replication can be used both for scalability, in order to have multiple replicas for read-only queries, or simply for improving data safety and high availability.
  - It is possible to use replication to avoid the cost of having the master writing the full dataset to disk: a typical technique involves configuring your master redis.conf to avoid persisting to disk at all, then connect a replica configured to save from time to time, or with AOF enabled. However this setup must be handled with care, since a restarting master will start with an empty dataset: if the replica tries to synchronized with it, the replica will be emptied as well.

Replication

- Every Redis master has a replication ID: it is a large pseudo random string that marks a given story of the dataset. Each master also takes an offset that increments for every byte of replication stream that it is produced to be sent to replicas, in order to update the state of the replicas with the new changes modifying the dataset. The replication offset is incremented even if no replica is actually connected, so basically every given pair of:
  - Replication ID, offset
- Identifies an exact version of the dataset of a master.
- When replicas connect to masters, they use the PSYNC command in order to send their old master replication ID and the offsets they processed so far. This way the master can send just the incremental part needed. However if there is not enough *backlog* in the master buffers, or if the replica is referring to an history (replication ID) which is no longer known, than a full resynchronization happens: in this case the replica will get a full copy of the dataset, from scratch.

Replication

- This is how a full synchronization works in more details:
  - The master starts a background saving process in order to produce an RDB (Redis Database) file. At the same time it starts to buffer all new write commands received from the clients. When the background saving is complete, the master transfers the database file to the replica, which saves it on disk, and then loads it into memory. The master will then send all buffered commands to the replica. This is done as a stream of commands and is in the same format of the Redis protocol itself.

- As already said, replicas are able to automatically reconnect when the master-replica link goes down for some reason. If the master receives multiple concurrent replica synchronization requests, it performs a single background save in order to serve all of them.

Replication ID explained

- If two instances have the same replication ID and replication offset, they have exactly the same data.
- A replication ID basically marks a given *history* of the data set. Every time an instance restarts from scratch as a master, or a replica is promoted to master, a new replication ID is generated for this instance. The replicas connected to a master will inherit its replication ID after the handshake. So two instances with the same ID are related by the fact that they hold the same data, but potentially at a different time. It is the offset that works as a logical time to understand, for a given history (replication ID) who holds the most updated data set.
- For instance if two instances A and B have the same replication ID, but one with offset 1000 and one with offset 1023, it means that the first lacks certain commands applied to the data set. It also means that A, by applying just a few commands, may reach exactly the same state of B.

Replication ID explained

- The reason why Redis instances have two replication IDs is because of replicas that are promoted to masters. After a failover, the promoted replica requires to still remember what was its past replication ID, because such replication ID was the one of the former master. In this way, when other replicas will synchronize with the new master, they will try to perform a partial resynchronization using the old master replication ID. This will work as expected, because when the replica is promoted to master it sets its secondary ID to its main ID, remembering what was the offset when this ID switch happened. Later it will select a new random replication ID, because a new history begins. When handling the new replicas connecting, the master will match their IDs and offsets both with the current ID and the secondary ID (up to a given offset, for safety). In short this means that after a failover, replicas connecting to the new promoted master don't have to perform a full sync.
- In case you wonder why a replica promoted to master needs to change its replication ID after a failover: it is possible that the old master is still working as a master because of some network partition: retaining the same replication ID would violate the fact that the same ID and same offset of any two random instances mean they have the same data set.

Configuration

- To configure basic Redis replication is trivial: just add the following line to the replica configuration file:
  - replicaof 192.168.1.1 6379
- Alternatively, you can call the REPLICAOF command and the master host will start a sync with the replica.

Redis persistence

Redis persistence

- Redis provides a different range of persistence options:
  - The RDB (Redis Database File) persistence performs point-in-time snapshots of your dataset at specified intervals.
  - The AOF (Append Only File)persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to rewrite the log in the background when it gets too big.
  - If you wish, you can disable persistence completely, if you want your data to just exist as long as the server is running.
  - It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

RDB advantages

- RDB is a very compact single-file point-in-time representation of your Redis data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.
- RDB is very good for disaster recovery, being a single compact file that can be transferred to far data centers (possibly encrypted).
- RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest. The parent instance will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.

RDB disadvantages

- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

AOF advantages

- Using AOF Redis is much more durable: you can have different fsync policies:
  - no fsync at all,
  - fsync every second,
  - fsync at every query.

  With the default policy of fsync every second write performances are still great (fsync is performed using a background thread and the main thread will try hard to perform writes when no fsync is in progress.) but you can only lose one second worth of writes.
- The AOF log is an append only log, so there are no seeks, nor corruption problems if there is a power outage. Even if the log ends with an half-written command for some reason (disk full or other reasons) the redis-check-aof tool is able to fix it easily.

AOF advantages

- Redis is able to automatically rewrite the AOF in background when it gets too big. The rewrite is completely safe as while Redis continues appending to the old file, a completely new one is produced with the minimal set of operations needed to create the current data set, and once this second file is ready Redis switches the two and starts appending to the new one.
- AOF contains a log of all the operations one after the other in an easy to understand and parse format. You can even easily export an AOF file. For instance even if you flushed everything for an error using a FLUSHALL command, if no rewrite of the log was performed in the meantime you can still save your data set just stopping the server, removing the latest command, and restarting Redis again.

AOF disadvantages

- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- AOF can be slower than RDB depending on the exact fsync policy. In general with fsync set to *every second* performance is still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.

AOF disadvantages

- In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to not reproduce exactly the same dataset on reloading. These bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is fine. However, these kind of bugs are almost impossible with RDB persistence.

The solution

- The general indication is that you should use both persistence methods if you want a degree of data safety comparable to what PostgreSQL can provide you.
- If you care a lot about your data, but still can live with a few minutes of data loss in case of disasters, you can simply use RDB alone.
- There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.

Snapshotting

- By default Redis saves snapshots of the dataset on disk, in a binary file called dump.rdb. You can configure Redis to have it save the dataset every N seconds if there are at least M changes in the dataset, or you can manually call the SAVE or BGSAVE commands.
- For example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:
  - save 60 1000
- This strategy is known as *snapshotting*.

Snapshotting

- How it works
- Whenever Redis needs to dump the dataset to disk, this is what happens:
  - Redis forks. We now have a child and a parent process.
  - The child starts to write the dataset to a temporary RDB file.
  - When the child is done writing the new RDB file, it replaces the old one.
- This method allows Redis to benefit from copy-on-write semantics.
- Snapshotting is not very durable. If your computer running Redis stops, your power line fails, or you accidentally kill -9 your instance, the latest data written on Redis will get lost. While this may not be a big deal for some applications, there are use cases for full durability, and in these cases Redis was not a viable option.

Append-only file

- The *append-only file* is an alternative, fully-durable strategy for Redis. You can turn on the AOF in your configuration file:
  - appendonly yes
- From now on, every time Redis receives a command that changes the dataset (e.g. SET) it will append it to the AOF. When you restart Redis it will re-play the AOF to rebuild the state.

Append-only file

- Log rewriting
  - As you can guess, the AOF gets bigger and bigger as write operations are performed. For example, if you are incrementing a counter 100 times, you'll end up with a single key in your dataset containing the final value, but 100 entries in your AOF. 99 of those entries are not needed to rebuild the current state.
  - So Redis supports an interesting feature: it is able to rebuild the AOF in the background without interrupting service to clients. Whenever you issue a BGREWRITEAOF Redis will write the shortest sequence of commands needed to rebuild the current dataset in memory. Newer versions of Redis is able to trigger log rewriting automatically.

Append-only file

- How durable is the append only file?
  - You can configure how many times Redis will fsync data on disk. There are three options:
    - appendfsync always: fsync every time a new command is appended to the AOF. Very very slow, very safe.
    - appendfsync everysec: fsync every second. Fast enough, and you can lose 1 second of data if there is a disaster.
    - appendfsync no: Never fsync, just put your data in the hands of the Operating System. The faster and less safe method. Normally Linux will flush data every 30 seconds with this configuration, but it's up to the kernel exact tuning.
- The suggested (and default) policy is to fsync every second. It is both very fast and pretty safe.
  The always policy is very slow in practice, but it supports group commit, so if there are multiple parallel writes Redis will try to perform a
  single fsync operation.

Redis Partitioning

- Partitioning is the process of splitting your data into multiple Redis instances, so that every instance will only contain a subset of your keys.

- Partitioning in Redis serves two main goals:
  - It allows for much larger databases, using the sum of the memory of many computers. Without partitioning you are limited to the amount of memory a single computer can support.
  - It allows scaling the computational power to multiple cores and multiple computers, and the network bandwidth to multiple computers and network adapters.

Redis Partitioning

- Imagine we have four Redis instances R0, R1, R2, R3, and many keys representing users like user:1, user:2, ... and so forth, we can find different ways to select in which instance we store a given key. In other words there are different systems to map a given key to a given Redis server.

1. Range partition

- One of the simplest ways to perform partitioning is with range partitioning, and is accomplished by mapping ranges of objects into specific Redis instances. For example, I could say users from ID 0 to ID 10000 will go into instance R0, while users form ID 10001 to ID 20000 will go into instance R1 and so forth.
- This system works and is actually used in practice, however, it has the disadvantage of requiring a table that maps ranges to instances. This table needs to be managed and a table is needed for every kind of object, so therefore range partitioning in Redis is often undesirable because it is much more inefficient than other alternative partitioning approaches.

Redis Partitioning

- Imagine we have four Redis instances R0, R1, R2, R3, and many keys representing users like user:1, user:2, ... and so forth, we can find different ways to select in which instance we store a given key. In other words there are different systems to map a given key to a given Redis server.

2. Hash partitioning

- This scheme works with any key and is as simple as:
- Take the key name and use a hash function (e.g., the crc32 hash function) to turn it into a number. For example, if the key is foobar, crc32(foobar) will output something like 93024922.
- Use a modulo operation with this number in order to turn it into a number between 0 and 3, so that this number can be mapped to one of my four Redis instances. 93024922 modulo 4 equals 2, so I know my key foobar should be stored into the R2 instance. Note: the modulo operation returns the remainder from a division operation, and is implemented with the % operator in many programming languages.

3. Others

- There are many other ways to perform partitioning, but with these two examples you should get the idea. One advanced form of hash partitioning is called consistent hashing and is implemented by a few Redis clients and proxies.

Redis Partitioning

- **Different implementations of partitioning**
- Partitioning can be the responsibility of different parts of a software stack.
  - **Client side partitioning** means that the clients directly select the right node where to write or read a given key. Many Redis clients implement client side partitioning.
  - **Proxy assisted partitioning** means that our clients send requests to a proxy that is able to speak the Redis protocol, instead of sending requests directly to the right Redis instance. The proxy will make sure to forward our request to the right Redis instance according to the configured partitioning schema, and will send the replies back to the client. The Redis and Memcached proxy Twemproxy implements proxy assisted partitioning.
  - **Query routing** means that you can send your query to a random instance, and the instance will make sure to forward your query to the right node. Redis Cluster implements an hybrid form of query routing, with the help of the client (the request is not directly forwarded from a Redis instance to another, but the client gets *redirected* to the right node).

Redis Partitioning

- **Disadvantages of partitioning**
- Some features of Redis don't play very well with partitioning:
  - Operations involving multiple keys are usually not supported. For instance you can't perform the intersection between two sets if they are stored in keys that are mapped to different Redis instances (actually there are ways to do this, but not directly).
  - Redis transactions involving multiple keys can not be used.
  - The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set.
  - When partitioning is used, data handling is more complex, for instance you have to handle multiple RDB / AOF files, and to make a backup of your data you need to aggregate the persistence files from multiple instances and hosts.
  - Adding and removing capacity can be complex. For instance Redis Cluster supports mostly transparent rebalancing of data with the ability to add and remove nodes at runtime, but other systems like client side partitioning and proxies don't support this feature. However a technique called *Pre-sharding* helps in this regard.

Redis Partitioning

Implementations of Redis partitioning

What system should you use in practice?

1. Redis Cluster

- Redis Cluster is the preferred way to get automatic sharding and high availability. Once Redis Cluster is available, and if a Redis Cluster compliant client is available for your language, Redis Cluster will be the de facto standard for Redis partitioning.
- Redis Cluster is a mix between *query routing* and *client side partitioning*.

2. Twemproxy

- Twemproxy is a proxy developed at Twitter for the Memcached ASCII and the Redis protocol. It is single threaded, it is written in C, and is extremely fast. It is open source software released under the terms of the Apache 2.0 license.
- Twemproxy supports automatic partitioning among multiple Redis instances, with optional node ejection if a node is not available (this will change the keys-instances map, so you should use this feature only if you are using Redis as a cache).
- Basically Twemproxy is an intermediate layer between clients and Redis instances, that will reliably handle partitioning for us with minimal additional complexities.

3. Clients supporting consistent hashing

- An alternative to Twemproxy is to use a client that implements client side partitioning via consistent hashing or other similar algorithms. There are multiple Redis clients with support for consistent hashing (Redis-py is a client for Redis).

Redis Cluster

Redis Cluster

- Redis Cluster provides a way to run a Redis installation where data is **automatically sharded across multiple Redis nodes**.
- Redis Cluster also provides **some degree of availability during partitions**, that is in practical terms the ability to continue the operations when some nodes fail or are not able to communicate. However the cluster stops to operate in the event of larger failures (for example when the majority of masters are unavailable).
- You will get with Redis Cluster:
  - The ability to **automatically split your dataset among multiple nodes**.
  - The ability to **continue operations when a subset of the nodes are experiencing failures** or are unable to communicate with the rest of the cluster.

Redis Cluster TCP port

- Every Redis Cluster node requires two TCP connections open. The normal Redis TCP port used to serve clients, for example 6379, plus the port obtained by adding 10000 to the data port, so 16379 in the example.
- This second *high* port is used for the Cluster bus, that is a node-to-node communication channel using a binary protocol. The Cluster bus is used by nodes for failure detection, configuration update, failover authorization and so forth. Clients should never try to communicate with the cluster bus port, but always with the normal Redis command port, however make sure you open both ports in your firewall, otherwise Redis cluster nodes will be not able to communicate.
- The command port and cluster bus port offset is fixed and is always 10000.

Redis Cluster data sharding

- Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an **hash slot**.
- There are 16384 hash slots in Redis Cluster, and to compute what is the hash slot of a given key, we simply take the CRC16 of the key modulo 16384.
- Every node in a Redis Cluster is responsible for a subset of the hash slots, so for example you may have a cluster with 3 nodes, where:
  - Node A contains hash slots from 0 to 5500.
  - Node B contains hash slots from 5501 to 11000.
  - Node C contains hash slots from 11001 to 16383.
- This allows to add and remove nodes in the cluster easily. For example if I want to add a new node D, I need to move some hash slot from nodes A, B, C to D. Similarly if I want to remove node A from the cluster I can just move the hash slots served by A to B and C. When the node A will be empty I can remove it from the cluster completely.

Redis Cluster data sharding

- Because moving hash slots from a node to another does not require to stop operations, adding and removing nodes, or changing the percentage of hash slots hold by nodes, does not require any downtime.
- Redis Cluster supports multiple key operations as long as all the keys involved into a single command execution (or whole transaction, or Lua script execution) all belong to the same hash slot. The user can force multiple keys to be part of the same hash slot by using a concept called *hash tags*.
- Hash tags are documented in the Redis Cluster specification, but the gist is that if there is a substring between {} brackets in a key, only what is inside the string is hashed, so for example this{foo}key and another{foo}key are guaranteed to be in the same hash slot, and can be used together in a command with multiple keys as arguments.

Redis Cluster master-slave model

- In order to remain available when a subset of master nodes are failing or are not able to communicate with the majority of nodes, Redis Cluster uses a master-slave model where every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes).
- In our example cluster with nodes A, B, C, if node B fails the cluster is not able to continue, since we no longer have a way to serve hash slots in the range 5501-11000.
- However when the cluster is created (or at a later time) we add a slave node to every master, so that the final cluster is composed of A, B, C that are masters nodes, and A1, B1, C1 that are slaves nodes, the system is able to continue if node B fails.
- Node B1 replicates B, and B fails, the cluster will promote node B1 as the new master and will continue to operate correctly.
- However note that if nodes B and B1 fail at the same time Redis Cluster is not able to continue to operate.

Redis Cluster consistency guarantees

- Redis Cluster is not able to guarantee **strong consistency**. In practical terms this means that under certain conditions it is possible that Redis Cluster will lose writes that were acknowledged by the system to the client.
- The first reason why Redis Cluster can lose writes is because it uses asynchronous replication. This means that during writes the following happens:
  - Your client writes to the master B.
  - The master B replies OK to your client.
  - The master B propagates the write to its slaves B1, B2 and B3.
- As you can see, B does not wait for an acknowledgement from B1, B2, B3 before replying to the client, since this would be a prohibitive latency penalty for Redis, so if your client writes something, B acknowledges the write, but crashes before being able to send the write to its slaves, one of the slaves (that did not receive the write) can be promoted to master, losing the write forever.

Redis Cluster consistency guarantees

- This is **very similar to what happens** with most databases that are configured to flush data to disk every second, so it is a scenario you are already able to reason about because of past experiences with traditional database systems not involving distributed systems. Similarly you can improve consistency by forcing the database to flush data on disk before replying to the client, but this usually results into prohibitively low performance. That would be the equivalent of synchronous replication in the case of Redis Cluster.
- Basically there is a trade-off to take between performance and consistency.

Redis Cluster consistency guarantees

- Redis Cluster has support for synchronous writes when absolutely needed, implemented via the WAIT command, this makes losing writes a lot less likely, however note that Redis Cluster does not implement strong consistency even when synchronous replication is used: it is always possible under more complex failure scenarios that a slave that was not able to receive the write is elected as master.

Redis Cluster consistency guarantees - node timeout

- There is another notable scenario where Redis Cluster will lose writes, that happens during a network partition where a client is isolated with a minority of instances including at least a master.
- Take as an example our 6 nodes cluster composed of A, B, C, A1, B1, C1, with 3 masters and 3 slaves. There is also a client, that we will call Z1.
- After a partition occurs, it is possible that in one side of the partition we have A, C, A1, B1, C1, and in the other side we have B and Z1.
- Z1 is still able to write to B, that will accept its writes. If the partition heals in a very short time, the cluster will continue normally. However if the partition lasts enough time for B1 to be promoted to master in the majority side of the partition, the writes that Z1 is sending to B will be lost.

Redis Cluster consistency guarantees – node timeout

- Note that there is a **maximum window** to the amount of writes Z1 will be able to send to B: if enough time has elapsed for the majority side of the partition to elect a slave as master, every master node in the minority side stops accepting writes.
- This amount of time is a very important configuration directive of Redis Cluster, and is called the **node timeout**.
- After node timeout has elapsed, a master node is considered to be failing, and can be replaced by one of its replicas. Similarly after node timeout has elapsed without a master node to be able to sense the majority of the other master nodes, it enters an error state and stops accepting writes.

Redis Cluster

On higher level

Redis Cluster – write safety

- Redis Cluster uses asynchronous replication between nodes, and **last failover wins** implicit merge function. This means that the last elected master dataset eventually replaces all the other replicas. There is always a window of time when it is possible to lose writes during partitions. However these windows are very different in the case of a client that is connected to the majority of masters, and a client that is connected to the minority of masters.

Redis Cluster – availability

- Redis Cluster is not available in the minority side of the partition. In the majority side of the partition assuming that there are at least the majority of masters and a slave for every unreachable master, the cluster becomes available again after NODE_TIMEOUT time plus a few more seconds required for a slave to get elected and failover its master (failovers are usually executed in a matter of 1 or 2 seconds).
- Thanks to a Redis Cluster feature called **replicas migration** the Cluster availability is improved in many real world scenarios by the fact that replicas migrate to orphaned masters (masters no longer having replicas). So at every successful failure event, the cluster may reconfigure the slaves layout in order to better resist the next failure.

Redis Cluster – Why merge operations are avoided

- Redis Cluster design avoids conflicting versions of the same key-value pair in multiple nodes as in the case of the Redis data model this is not always desirable. Values in Redis are often very large; it is common to see lists or sorted sets with millions of elements. Also data types are semantically complex. Transferring and merging these kind of values can be a major bottleneck and/or may require the non-trivial involvement of application-side logic, additional memory to store meta-data, and so forth.

Redis Cluster – Why merge operations are avoided

- Redis Cluster design avoids conflicting versions of the same key-value pair in multiple nodes as in the case of the Redis data model this is not always desirable. Values in Redis are often very large; it is common to see lists or sorted sets with millions of elements. Also data types are semantically complex. Transferring and merging these kind of values can be a major bottleneck and/or may require the non-trivial involvement of application-side logic, additional memory to store meta-data, and so forth.

Overview of Redis Cluster main components

Keys distribution model

- The key space is split into 16384 slots, effectively setting an upper limit for the cluster size of 16384 master nodes (however the suggested max size of nodes is in the order of ~ 1000 nodes).
- Each master node in a cluster handles a subset of the 16384 hash slots. The cluster is **stable** when there is no cluster reconfiguration in progress (i.e. where hash slots are being moved from one node to another). When the cluster is stable, a single hash slot will be served by a single node (however the serving node can have one or more slaves that will replace it in the case of net splits or failures, and that can be used in order to scale read operations where reading stale data is acceptable).

Keys hash tags

- There is an exception for the computation of the hash slot that is used in order to implement **hash tags**. Hash tags are a way to ensure that multiple keys are allocated in the same hash slot. This is used in order to implement multi-key operations in Redis Cluster.
- In order to implement hash tags, the hash slot for a key contains a "{...}" pattern. Only the substring between { and } is hashed in order to obtain the hash slot. It is possible that there are multiple occurrences of { or }. (See the documentation.)

Cluster nodes attributes

- Every node has a unique name in the cluster. The node will save its ID in the node configuration file, and will use the same ID forever, or at least as long as the node configuration file is not deleted by the system administrator, or a *hard reset* is requested via the CLUSTER RESET command.
- The node ID is used to identify every node across the whole cluster. It is possible for a given node to change its IP address without any need to also change the node ID. The cluster is also able to detect the change in IP/port and reconfigure using the gossip protocol running over the cluster bus.

Cluster nodes attributes

- The node ID is not the only information associated with each node, but is the only one that is always globally consistent. Every node has also the following set of information associated:
  - Some information is about the cluster configuration detail of this specific node, and is eventually consistent across the cluster.
  - Some other information, like the last time a node was pinged, is instead local to each node.

Cluster nodes attributes

- Every node maintains the following information about other nodes that it is aware of in the cluster:
  - The node ID, IP and port of the node, a set of flags, what is the master of the node if it is flagged as slave, last time the node was pinged and the last time the pong was received, the current *configuration epoch* of the node (explained later in this specification), the link state and finally the set of hash slots served.
- The CLUSTER NODES command can be sent to any node in the cluster and provides the state of the cluster and the information for each node according to the local view the queried node has of the cluster.

The Cluster bus

- Every Redis Cluster node has an additional TCP port for receiving incoming connections from other Redis Cluster nodes. This port is at a fixed offset from the normal TCP port used to receive incoming connections from clients. To obtain the Redis Cluster port, 10000 should be added to the normal commands port. For example, if a Redis node is listening for client connections on port 6379, the Cluster bus port 16379 will also be opened.
- Node-to-node communication happens exclusively using the Cluster bus.

Nodes handshake

- Nodes always accept connections on the cluster bus port, and even reply to pings when received, even if the pinging node is not trusted. However, all other packets will be discarded by the receiving node if the sending node is not considered part of the cluster.

Nodes handshake

- A node will accept another node as part of the cluster only in two ways:
  - If a node presents itself with a MEET message. A meet message is exactly like a PING message, but forces the receiver to accept the node as part of the cluster. Nodes will send MEET messages to other nodes **only if** the system administrator requests this via the following command: CLUSTER MEET ip port
  - A node will also register another node as part of the cluster if a node that is already trusted will gossip about this other node. So if A knows B, and B knows C, eventually B will send gossip messages to A about C. When this happens, A will register C as part of the network, and will try to connect with C.

Nodes handshake

- This means that as long as we join nodes in any connected graph, they'll eventually form a fully connected graph automatically. This means that the cluster is able to auto-discover other nodes, but only if there is a trusted relationship that was forced by the system administrator.
- This mechanism makes the cluster more robust but prevents different Redis clusters from accidentally mixing after change of IP addresses or other network related events.

Redirection and resharding

MOVED Redirection

- A Redis client is free to send queries to every node in the cluster, including slave nodes. The node will analyze the query, and if it is acceptable (that is, only a single key is mentioned in the query, or the multiple keys mentioned are all to the same hash slot) it will lookup what node is responsible for the hash slot where the key or keys belong.
- If the hash slot is served by the node, the query is simply processed, otherwise the node will check its internal hash slot to node map, and will reply to the client with a MOVED error, like in the following example:
  - GET x
  - -MOVED 3999 127.0.0.1:6381

MOVED Redirection

- The error includes the hash slot of the key (3999) and the ip:port of the instance that can serve the query. The client needs to reissue the query to the specified node's IP address and port. Note that even if the client waits a long time before reissuing the query, and in the meantime the cluster configuration changed, the destination node will reply again with a MOVED error if the hash slot 3999 is now served by another node. The same happens if the contacted node had no updated information.

MOVED Redirection

- So while from the point of view of the cluster nodes are identified by IDs we try to simplify our interface with the client just exposing a map between hash slots and Redis nodes identified by IP:port pairs.
- The client is not required to, but should try to memorize that hash slot 3999 is served by 127.0.0.1:6381. This way once a new command needs to be issued it can compute the hash slot of the target key and have a greater chance of choosing the right node.

Cluster live reconfiguration

- Redis Cluster supports the ability to add and remove nodes while the cluster is running. Adding or removing a node is abstracted into the same operation: moving a hash slot from one node to another. This means that the same basic mechanism can be used in order to rebalance the cluster, add or remove nodes, and so forth.
  - To add a new node to the cluster an empty node is added to the cluster and some set of hash slots are moved from existing nodes to the new node.
  - To remove a node from the cluster the hash slots assigned to that node are moved to other existing nodes.
  - To rebalance the cluster a given set of hash slots are moved between nodes.

ASK redirection

- While MOVED means that we think the hash slot is permanently served by a different node and the next queries should be tried against the specified node, ASK means to send only the next query to the specified node.
- The full semantics of ASK redirection from the point of view of the client is as follows:
  - If ASK redirection is received, send only the query that was redirected to the specified node but continue sending subsequent queries to the old node.
  - Start the redirected query with the ASKING command.
  - Don't yet update local client tables to map hash slot 8 to B.

Clients first connection and handling of redirections

- While it is possible to have a Redis Cluster client implementation that does not remember the slots configuration (the map between slot numbers and addresses of nodes serving it) in memory and only works by contacting random nodes waiting to be redirected, such a client would be very inefficient.
- Redis Cluster clients should try to be smart enough to memorize the slots configuration. However this configuration is not *required* to be up to date. Since contacting the wrong node will simply result in a redirection, that should trigger an update of the client view.
- Clients usually need to fetch a complete list of slots and mapped node addresses in two different situations:
  - At startup in order to populate the initial slots configuration.
  - When a MOVED redirection is received.

Clients first connection and handling of redirections

- Note that a client may handle the MOVED redirection by updating just the moved slot in its table, however this is usually not efficient since often the configuration of multiple slots is modified at once (for example if a slave is promoted to master, all the slots served by the old master will be remapped). It is much simpler to react to a MOVED redirection by fetching the full map of slots to nodes from scratch.
- CLUSTER SLOTS command provides an array of slots ranges, and the associated master and slave nodes serving the specified range.

Scaling reads using slave nodes

- Normally slave nodes will redirect clients to the authoritative master for the hash slot involved in a given command, however clients can use slaves in order to scale reads using the READONLY command.
- READONLY tells a Redis Cluster slave node that the client is ok reading possibly stale data and is not interested in running write queries.
- When the connection is in readonly mode, the cluster will send a redirection to the client only if the operation involves keys not served by the slave's master node. This may happen because:
- The readonly state of the connection can be cleared using the READWRITE command.

Additionals

- **Fault Tolerance**
- **Configuration handling, propagation, and failovers**

https://redis.io/topics/cluster-spec

Redis Cluster

Redis Cluster

- Redis Cluster provides a way to run a Redis installation where data is **automatically sharded across multiple Redis nodes**.
- Redis Cluster also provides **some degree of availability during partitions**, that is in practical terms the ability to continue the operations when some nodes fail or are not able to communicate. However the cluster stops to operate in the event of larger failures (for example when the majority of masters are unavailable).
- You will get with Redis Cluster:
  - The ability to **automatically split your dataset among multiple nodes**.
  - The ability to **continue operations when a subset of the nodes are experiencing failures** or are unable to communicate with the rest of the cluster.

Redis Cluster TCP port

- Every Redis Cluster node requires two TCP connections open. The normal Redis TCP port used to serve clients, for example 6379, plus the port obtained by **adding 10000 to the data port**, so 16379 in the example.
- This second *high* port is used for the **Cluster bus**, that is a node-to-node communication channel using a binary protocol. The Cluster bus is used by nodes for failure detection, configuration update, failover authorization and so forth. Clients should never try to communicate with the cluster bus port, but always with the normal Redis command port, however make sure you open both ports in your firewall, otherwise Redis cluster nodes will be not able to communicate.
- The command port and cluster bus port offset is fixed and is always 10000.

Redis Cluster data sharding

- Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an **hash slot**.
- There are 16384 hash slots in Redis Cluster, and to compute what is the hash slot of a given key, we simply take the CRC16 of the key modulo 16384.
- Every node in a Redis Cluster is responsible for a subset of the hash slots, so for example you may have a cluster with 3 nodes, where:
  - Node A contains hash slots from 0 to 5500.
  - Node B contains hash slots from 5501 to 11000.
  - Node C contains hash slots from 11001 to 16383.
- This allows to add and remove nodes in the cluster easily. For example if I want to add a new node D, I need to move some hash slot from nodes A, B, C to D. Similarly if I want to remove node A from the cluster I can just move the hash slots served by A to B and C. When the node A will be empty I can remove it from the cluster completely.

Redis Cluster data sharding

- Because moving hash slots from a node to another does not require to stop operations, adding and removing nodes, or changing the percentage of hash slots hold by nodes, does not require any downtime.
- Redis Cluster supports multiple key operations as long as all the keys involved into a single command execution (or whole transaction, or Lua script execution) all belong to the same hash slot. The user can force multiple keys to be part of the same hash slot by using a concept called *hash tags*.
- Hash tags are documented in the Redis Cluster specification, but the gist is that if there is a substring between {} brackets in a key, only what is inside the string is hashed, so for example this{foo}key and another{foo}key are guaranteed to be in the same hash slot, and can be used together in a command with multiple keys as arguments.
- {foo}apple, {foo}banana

Redis Cluster master-slave model

- In order to remain available when a subset of master nodes are failing or are not able to communicate with the majority of nodes, Redis Cluster uses a master-slave model where every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes).
- In our example cluster with nodes A, B, C, if node B fails the cluster is not able to continue, since we no longer have a way to serve hash slots in the range 5501-11000.
- However when the cluster is created (or at a later time) we add a slave node to every master, so that the final cluster is composed of A, B, C that are masters nodes, and A1, B1, C1 that are slaves nodes, the system is able to continue if node B fails.
- Node B1 replicates B, and B fails, the cluster will promote node B1 as the new master and will continue to operate correctly.
- However note that if nodes B and B1 fail at the same time Redis Cluster is not able to continue to operate.

Redis Cluster consistency guarantees

- Redis Cluster is not able to guarantee **strong consistency**. In practical terms this means that under certain conditions it is possible that Redis Cluster will lose writes that were acknowledged by the system to the client.
- The first reason why Redis Cluster can lose writes is because it uses asynchronous replication. This means that during writes the following happens:
  - Your client writes to the master B.
  - The master B replies OK to your client.
  - The master B propagates the write to its slaves B1, B2 and B3.
- As you can see, B does not wait for an acknowledgement from B1, B2, B3 before replying to the client, since this would be a prohibitive latency penalty for Redis, so if your client writes something, B acknowledges the write, but crashes before being able to send the write to its slaves, one of the slaves (that did not receive the write) can be promoted to master, losing the write forever.

Redis Cluster consistency guarantees

- This is **very similar to what happens** with most databases that are configured to flush data to disk every second, so it is a scenario you are already able to reason about because of past experiences with traditional database systems not involving distributed systems. Similarly you can improve consistency by forcing the database to flush data on disk before replying to the client, but this usually results into prohibitively low performance. That would be the equivalent of synchronous replication in the case of Redis Cluster.
- Basically there is a trade-off to take between performance and consistency.

Redis Cluster consistency guarantees

- Redis Cluster has support for synchronous writes when absolutely needed, implemented via the WAIT command, this makes losing writes a lot less likely, however note that Redis Cluster does not implement strong consistency even when synchronous replication is used: it is always possible under more complex failure scenarios that a slave that was not able to receive the write is elected as master.

Redis Cluster consistency guarantees - node timeout

- There is another notable scenario where Redis Cluster will lose writes, that happens during a network partition where a client is isolated with a minority of instances including at least a master.
- Take as an example our 6 nodes cluster composed of A, B, C, A1, B1, C1, with 3 masters and 3 slaves. There is also a client, that we will call Z1.
- After a partition occurs, it is possible that in one side of the partition we have A, C, A1, B1, C1, and in the other side we have B and Z1.
- Z1 is still able to write to B, that will accept its writes. If the partition heals in a very short time, the cluster will continue normally. However if the partition lasts enough time for B1 to be promoted to master in the majority side of the partition, the writes that Z1 is sending to B will be lost.

Redis Cluster consistency guarantees – node timeout

- Note that there is a **maximum window** to the amount of writes Z1 will be able to send to B: if enough time has elapsed for the majority side of the partition to elect a slave as master, every master node in the minority side stops accepting writes.
- This amount of time is a very important configuration directive of Redis Cluster, and is called the **node timeout**.
- After node timeout has elapsed, a master node is considered to be failing, and can be replaced by one of its replicas. Similarly after node timeout has elapsed without a master node to be able to sense the majority of the other master nodes, it enters an error state and stops accepting writes.

Creating and using a Redis Cluster

- To create a cluster, the first thing we need is to have a few empty Redis instances running in **cluster mode**. This basically means that clusters are not created using normal Redis instances as a special mode needs to be configured so that the Redis instance will enable the Cluster specific features and commands.
- The following is a minimal Redis cluster configuration file:

```
port 6379
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

Creating and using a Redis Cluster

- Every instance also contains the path of a file where the configuration for this node is stored, which by default is **nodes.conf**. This file is **never touched by humans**; it is simply generated at startup by the Redis Cluster instances, and updated every time it is needed.
- Note that the **minimal cluster** that works as expected requires to contain **at least three master nodes**. For your first tests it is strongly suggested to start a six nodes cluster with three masters and three slaves.

Creating and using a Redis Cluster

- You need at least 3 Redis database in cluster mode.
- As you can see from the logs of every instance every node assigns itself a new ID. This ID will be used forever by this specific instance in order for the instance to have a unique name in the context of the cluster. Every node remembers every other node using this IDs, and not by IP or port. IP addresses and ports may change, but the unique node identifier will never change for all the life of the node. We call this identifier simply **Node ID**.

Creating and using a Redis Cluster

- To create a cluster with 3 nodes:
- redis-cli --cluster create 172.22.207.171:6379 172.22.207.172:6379 172.22.207.173:6379 --cluster-replicas 0
- In this case we has 3 masters.

- To create a cluster with 6 nodes:
- redis-cli --cluster create 172.22.207.171:6379 172.22.207.172:6379 172.22.207.173:6379 172.22.207.174:6379 172.22.207.175:6379 172.22.207.176:6379 --cluster-replicas 1
- In this case we has 3 masters and 3 slaves.

Creating and using a Redis Cluster

- The option --cluster-replicas 1 means that we want a slave for every master created. The other arguments are the list of addresses of the instances I want to use to create the new cluster.
- Redis-cli will propose you a configuration. Accept the proposed configuration by typing **yes**.
- The cluster will be configured and *joined*, which means, instances will be bootstrapped into talking with each other.

Creating and using a Redis Cluster

- redis-cli -c
- You can connect to the Redis with the previous statement.

- The redis-cli cluster support is very basic so it always uses the fact that Redis Cluster nodes are able to redirect a client to the right node.

Creating and using a Redis Cluster

Connect to a redis cluster with a programming language:

- python:
  - Needs redis-py-cluster (but it is old, doesn't support the new Redis version)
- C#:
  - FreeRedis worked, we tried out
- Java:
  - Lettuce (I should try out, it looks good)

At this stage one of the problems with Redis Cluster is the lack of client libraries implementations.

Creating and using a Redis Cluster

You can:

- Reshard the cluster
  - redis-cli --cluster reshard
- Adding a new node
  - redis-cli --cluster add-node
- Adding a new node as a replica
  - redis-cli --cluster add-node 127.0.0.1:7000 --cluster-slave
- Removing a node
  - redis-cli --cluster del-node
- Upgrading nodes

Creating and using a Redis Cluster

Some statements in redis-cli:

- CLUSTER INFO provides information about Redis Cluster vital parameters.
- CLUSTER NODES**:** Each node in a Redis Cluster has its view of the current cluster configuration, given by the set of known nodes, the state of the connection we have with such nodes, their flags, properties and assigned slots, and so forth. CLUSTER NODES provides all this information, that is, the current cluster configuration of the node we are contacting.
- Note that normally clients willing to fetch the map between Cluster hash slots and node addresses should use CLUSTER SLOTS instead. CLUSTER NODES, that provides more information, should be used for administrative tasks, debugging, and configuration inspections. It is also used by redis-trib in order to manage a cluster.
- **CLUSTER FORGET node-id:** The command is used in order to remove a node, specified via its node ID, from the set of *known nodes* of the Redis Cluster node receiving the command.
- **CLUSTER REPLICATE node-id:** The command reconfigures a node as a replica of the specified master.
- **CLUSTER REPLICAS node-id:** The command provides a list of replica nodes replicating from the specified master node.
- CLUSTER SLOTS returns details about which cluster slots map to which Redis instances.

Creating and using a Redis Cluster

- **Redis Cluster configuration parameters**
- Let's introduce the configuration parameters that Redis Cluster introduces in the redis.conf file.
- **cluster-enabled <yes/no>**: If yes, enables Redis Cluster support in a specific Redis instance. Otherwise the instance starts as a stand alone instance as usual.
- **cluster-config-file <filename>**: Note that despite the name of this option, this is not a user editable configuration file, but the file where a Redis Cluster node automatically persists the cluster configuration (the state, basically) every time there is a change, in order to be able to re-read it at startup. The file lists things like the other nodes in the cluster, their state, persistent variables, and so forth. Often this file is rewritten and flushed on disk as a result of some message reception.
- **cluster-node-timeout <milliseconds>**: The maximum amount of time a Redis Cluster node can be unavailable, without it being considered as failing. If a master node is not reachable for more than the specified amount of time, it will be failed over by its slaves. Notably, every node that can't reach the majority of master nodes for the specified amount of time, will stop accepting queries.

Creating and using a Redis Cluster

- **Redis Cluster configuration parameters**
- **cluster-slave-validity-factor <factor>**: If set to zero, a slave will always consider itself valid, and will therefore always try to failover a master, regardless of the amount of time the link between the master and the slave remained disconnected. If the value is positive, a maximum disconnection time is calculated as the *node timeout* value multiplied by the factor provided with this option, and if the node is a slave, it will not try to start a failover if the master link was disconnected for more than the specified amount of time. For example, if the node timeout is set to 5 seconds and the validity factor is set to 10, a slave disconnected from the master for more than 50 seconds will not try to failover its master.
- **cluster-migration-barrier <count>**: Minimum number of replicas a master will remain connected with, for another replica to migrate to a master which is no longer covered by any replica.
- **cluster-require-full-coverage <yes/no>**: If this is set to yes, as it is by default, the cluster stops accepting writes if some percentage of the key space is not covered by any node. If the option is set to no, the cluster will still serve queries even if only requests about a subset of keys can be processed.
- **cluster-allow-reads-when-down <yes/no>**: If this is set to no, as it is by default, a node in a Redis Cluster will stop serving all traffic when the cluster is marked as failed, either when a node can't reach a quorum of masters or when full coverage is not met. This prevents reading potentially inconsistent data from a node that is unaware of changes in the cluster. This option can be set to yes to allow reads from a node during the fail state, which is useful for applications that want to prioritize read availability but still want to prevent inconsistent writes.