Webalkalmazások teljesítményoptimalizálása

Előadó

- Kovács Ferenc (Tyrael)
- Fejlesztési vezető @ Escalion

Definíciók

- Teljesítmény: egységnyi idő alatt végzett hasznos munka.
- Teljesítményoptimalizáció: a rendszer teljesítményének növelése, az elvárt működés (jelentősebb) megváltoztatása nélkül.
- Szűk keresztmetszet(bottleneck): a részegységek átlagteljesítményétől jelentősen elmaradó teljesítményű komponens, mely így az egész rendszer teljesítményét korlátozza.

Definíciók

- Jó teljesítmény jellemzői:
  - Alacsony válaszidő (latency)
  - Alacsony szórás a válaszidőkben (variance)
  - Nagy áteresztőképesség (throughput)
  - Takarékos erőforráskihasználás (res. utilization)
  - Lineáris skálázódás
- Általában a latency és a latency variance szemben áll a throughput-tal, ha az egyikre optimalizálsz, akkor a másik romlik.

Definíciók

- Scale up: A szerver teljesítményének a növelésével növeli a rendszer kapacitását.
  - + Nem igényel különösebb módosítást, vagy speciális tervezést a szoftveren.
  - - Exponenciálisan növekszik a nagyobb kapacitású szerverek ára, és nem növelhető végtelenségig a kapacitás.
- Scale out: A szerverek számának a növelésével növeli a rendszer kapacitását:
  - + Lineárishoz közeli a kapacitásnövelés költsége hardver oldalon.
  - - A szoftvernek támogatnia kell a clusterben való működést.

Idézetek

- "You can’t control what you can't measure.”
  -- *Tom DeMarco*

- "Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.”
  -- *Rob Pike*

Idézetek

- *"*We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified”

  -- Donald E. Knuth

Tippek

- Mindig szisztematikusan álljunk neki az optimalizációnak:
  - Mérjük ki, hogy mi a jelenlegi teljesítménye a rendszernek, és próbáljuk megfogalmazni, hogy mi az amit elfogadhatónak tekintenénk.
  - Keressük meg azt a komponenst ami leginkább felelős a jelenlegi elfogadhatatlan teljesítményért (bottleneck).
  - Javítsuk ki a problémás részt ügyelve arra, hogy megtartsuk az eredeti működést (vagy módosítsuk az elvárásainkat…).
  - Mérjük ki a módosított rendszer teljesítményét, és ha megfelelő, akkor keressük meg a következő bottleneck-et, egészen addig, amíg el nem érjük a kívánt teljesítményt, vagy el nem értük a rendszer maximális teljesítményét.

Tippek

- A tervezési fázisban már érdemes felmérni, hogy a szoftver mely részei lesznek sebességkritikusak és ott figyelni az optimalizációra.
- A pareto-elv (80-20 szabály) a teljesítményoptimalizálásra is igaz, általában a kód 20%-a teszi ki a futási idő 80%-át.
- Az egyszerű rendszer általában könnyen optimalizálható, illetve jól skálázódik, az optimalizált rendszer általában komplex, és nehéz skálázni, ezért törekedjünk először mindig az egyszerű megoldásokra.

Tippek

- A leggyorsabb működés az, ha nem csinálunk semmit:
  - Ha valamit nem kell végrehajtani, ne hajtsuk végre.
  - Ha valamit végre kell hajtani, hajtsuk végre az utolsó pillanatban, hátha mégse lesz rá szükség (lazy-loading pl.).
  - Ha valamit már végrehajtottunk egyszer, és megtehetjük, akkor ne hajtsuk többször végre(caching).
  - Ha több dolgot kell végrehajtani, akkor mindig próbáljuk meg kötegelve végrehajtani, mert az gazdaságosabb.
  - Viszont mivel webalkalmazásokról beszélünk, ezért próbáljunk meg minden időigényes feladatot kiszervezni háttérfolyamatnak, csökkentve ezáltal a felhasználók által tapasztalható töltési időt.

Tippek

- Meg lehet, és kell is különböztetni a valós és az érzékelt teljesítményt, a felhasználók általában nem stopperrel mérik az oldalt, hanem „érzésből”.
- Nem elég ha alacsonyak a válaszideink, a varianciára is figyeljünk oda.
- A felhasználók által érzékelt teljesítménynek általában csak kis részéért felelős közvetlenül a szerver oldali alkalmazás.
- Ezért is nagyon fontos, hogy mérjük, és monitorozzuk a teljes lapletöltést, ezáltal kimérhetjük, hogy hol vannak még optimalizációs lehetőségek.

Tippek

- Általában mindig tudunk némi „ingyen” teljesítménynövekedést kicsikarni, csupán azzal, hogy a legfrissebb verziót használjuk a szoftvereinkből, és minden általunk nem használt featuret kikapcsolunk, vagy bele sem forgatjuk az alkalmazásba (a statikus linkelés és tud még dobni egy kicsit).
- Ha nem akarunk csúnya meglepetéseket, akkor a build környezetünk mintájára csináljunk egy olyan környezetet, ahol folyamatosan monitorozzuk a rendszerünk teljesítményét egy fix tesztkészlettel, ezáltal azonnal észrevesszük, ha egy kódmódosítás, vagy szoftverfrissítés megváltoztatja a rendszer teljesítményét.

Hogy működik a web?

- A felhasználó beírja a böngészőjébe a www.example.com címet.
- A böngésző értelmezi az url-t, és megnézi, hogy cacheből kiszolgálható-e.
- Amennyiben nem, akkor megpróbálja feloldani a domaint egy IP címre, előbb a saját cache-éből, majd az operációs rendszer segítségével, ami rossz esetben egy címfeloldást fog kiváltani, ami egy network roundtrip legjobb esetben a legközelebbi DNS szerverig, legrosszabb esetben a saját NS, .com NS, example.com NS útvonalon.
- Miután megvan a címfeloldás, a böngésző létrehoz egy TCP connection-t a www.example.com domain IP címére a megfelelő porton (ez megint egy network roundtrip), majd ha nincs szerencsénk, és ssl kapcsolatot kell létrehozni, ami még +2 network roundtrip, valamint a cert kulcsméretétől függően némi plusz számolgatás.
- Ezután végre elküldtük a HTTP kérésünket. (network roundtrip a kérés válasz)
- A favicon.ico-ra nem térek ki, de az is egy vicces történet. :P

Hogy működik a web?

- A webszerver fogadja a kérést, ha kell kicsomagolja, decrypteli, parseolja a HTTP fejléceket.
- A webszerver a kapott fejlécek(host, url, etc) alapján bemappeli a kérést egy erőforráshoz majd ha kell, akkor átadja a végrehajtást a php értelmezőnek.
- A php értelmező a kapott fejlécek, környezeti változók alapján inicializálja a névteret (szuperglobális változók, etc.), majd betölti a megadott php scriptet.
- Ha nem használunk valamilyen opcode cache-t, akkor itt egy context switch történik, valamint ha nincs benne az OS cache-ében a keresett php script, akkor egy lemezművelet. Ha használunk opcode cache-t, és benne volt a keresett fájl, akkor megspóroltuk a fenti procedúrát, sőt még a fájl interpretálását is.
- Ezután a php értelmező végrehajta az opcode-ot, ha szükséges, akkor további fájlokat tölt be, majd script futása befejeződik, a script kimenete és a vezérlés visszakerül a webszerverhez.
- A webszerver a visszakapott kimenetet a HTTP response formájában visszaküldi a kliensnek (esetleg tömörít és vagy titkosít).

Hogy működik a web?

- Itt azt gondolhatnánk, hogy vége a történetnek, de most jön csak a java.
- A kliens felparse-olja a html kódot, ez alapján további fájlokat kér le a webszervertől (képek, css, js), amik ha nincs szerencsénk, még csak nem is párhuzamosan töltődnek le (maximalizálva van az egy időben egy domainről letölthető fájlok száma, illetve a js letöltődése általában blokkol minden más letöltést az adott oldalhoz, sőt az oldal renderelését is).
- Itt ugye megint az összes letöltésnél megtörténik az ssl handshake, hacsak nincs bekapcsolva a keepalive.
- Mindeközben a böngésző próbálja felépíteni illetve renderelni a dom-ot.
- És közben a látogató csak annyit ért az egészből, hogy lassú az oldal.

Hogy működik a web?

- Az általam vázoltak nyomon követésére a következő eszközöket tudom ajánlani:
  - tcpdump / ngrep / wireshark :)
  - Yslow
  - Page Speed
  - Speed Tracer
  - gtmetrix.com
- Jól látszik, hogy nagyon fontos, hogy ne valami ismeretlen, zárt dobozként tekintsünk az egyes rétegekre, tudjuk hogy mi történik a színfalak mögött, és tudjuk monitorozni, mérni az eseményeket.

Hogyan optimalizáljunk teljesítményt

- A problémákra a kötelező irodalmat röviden összefoglalva:
  - Vigyük közel a tartalmat a látogatóhoz: geobalancing/cdn megoldások
  - Csökkentsük a HTTP kérések számát (cache, inline képek, css sprite-ok, js/css combine)
  - Csökkentsük a fájlok méretét: minify js, css, statikus tartalmak tömörítése.
  - Csökkentsük a dns feloldások számát, de használjunk egynél több domaint a statikus fájlok linkelésére, ezáltal párhuzamosan több fájlt tudunk tölteni, csak ne vigyük túlzásba.
  - Csökkentsük a HTTP kérések méretét, pl. statikus képeket kiszolgáló domainre ne küldjük el a 4k-s sütijeinket.
  - Szervezzük jól a layoutot, css a <head>be, a js behúzások a </body> elé.
  - Csökkentsük a DOM fa méretét, sokat tud gyorsítani a dom műveleteken.
  - Ha szükséges, adjunk valami vizuális visszajelzést a felhasználónak, hogy lássa, hogy semmi pánik, történik valami (animált forgó fogaskerék, etc.).

Szerver oldalon

- Persze mégis előfordulhat, hogy a szerveroldalon lassú valami:
  - Ismerjük az eszközeinket(általában ez a LAMP stacket jelenti), illetve tudjuk itt is nyomon követni az eseményeket.
  - strace: nagyon hasznos kis bináris, a rendszerhívások monitorozására szolgál, futó processzekbe is bele tudunk kapaszkodni vele.
  - valgrind+callgrind: gyönyörű callgraphokat tudunk a kimenetéből rajzoltatni az így futtatott processzről.
  - Xdebug/xhprof: php alkalmazásainkat tudjuk többféle szempont szerint mérni, analizálni (callgraph-ot is tud mindkettő).
  - gdb: debugger, ha tényleg könyékig kell turkálni, mert mondjuk segfault-ol a php és a logokban nincs semmi.
  - iostat, vmstat, mpstat, netstat: ha IO, virtuális memória, processzor, vagy hálózat jellegű teljesítményproblémát kell monitorozni.
  - OProfile: ha balszerencsés módon kernel szintű low level analizísre lenne szükségünk.

Linux

- Itt nagyon sok apróság van, de általában ritkán szokott a default config szűk keresztmetszet lenni.
- Ahol lehet, nyugodtan támaszkodhatunk arra a tényre, hogy a linux híresen szereti a szabadon téblábolóó memóriát a gyakran használt fájlok cachelésére használni.
- Az io ütemezők közül érdemes lehet kipróbálni a cfq helyett a noop-ot vagy a deadline-t.

Apache

- Ha lehet, akkor cseréljük le valami lightweight-ebb cuccra, mondjuk nginxre, de ha csak dinamikus tartalmat szolgálunk ki vele, akkor akár meg is felelhet az igényeinknek.
- Statikus fájlokat ne apache-ból szolgáljuk ki, arra tényleg nem ő a nyerő.
- AllowOverride none, különben minden lapletöltésnél keresni fogja a .htaccess file-t rekurzívan a könyvtárakban, ami lassú.
- DirectoryIndex tartalmazza a lehető legkevesebb érteket, a használt gyakoriság sorrendjében.
- Favicon.ico legyen, megfelelően cachelve.
- Sokat tudunk dobni a teljesítményen, ha berakunk egy reverse proxyt a content szerverek elé, modjuk egy(több) Varnisht.

Apache

- A webszerverünk teljesítményét több eszközzel is tudjuk mérni:
  - ab
  - siege
  - WBox
  - multi-mechanize
  - Pylot

- A shared-nothing architechtúrából kifolyólag a webszervereink általában jól skálázhatóak, szemben mondjuk az alkalmazásszerveren futó rendszerekkel, ahol nagyobb overheadje van a cluster üzemeltetésének(persze ez általában nálunk is jelentkezik, de inkább a perzisztencia rétegben).

Mysql

- Ez is megérne egy külön előadássorozatot (nem akar valaki tartani? :))
- Többek között a google a facebook és a percona jóvoltából rengeteg fejlesztés, illetve tudásanyag került közkézre, érdemes csemegézni.
- 5.1-es verziónál régebbivel már ne kezdjünk új projectet, ha innodb-n gondolkozunk, akkor érdemes a beépített engine helyett a plugin-nal indulni, rengeteg fejlesztés van benne az engine-hez képest.
- Ha a dataset belefér memóriába, akkor kis hackeléssel akár egy memcache-t is odaver a kulcs-érték alapú lekérdezésekkel.
- Egy aksival megtámogatott write-cache enabled raid vezérlőt SAS, vagy egy SSD raid tömb nem árt azért ha kéznél van, és akkor még az írás is tud egész gyors lenni.

Mysql

- A percona jóvoltából több toolkit is született, ami segít nekünk a teljesítményoptimalizálásban:
  - Maatkit
    - **mk-index-usage: log alapján analizálja az indexek hasznosságát.**
    - **mk-query-advisor: szintén logból analizálja a queryket, és jelzi a problémákat.**
    - **mk-query-profiler: hasonló csak elsősorban teljesítményspecifikus infókat ad**
    - **mk-visual-explain: segít könnyen érthető formába önteni az explain eredményét**
  - Aspersa
    - **diskstats: /proc/diskstats kimenete kicsit jobban formázva**
    - **ioprofile: lsof+strace kimenetet összegezve egy áttekintést nyújt**
    - **mext: show global status monitorozásához egy hasznos eszköz**
    - **mysql-summary: segít felmérni egy mysql szerver konfigurációját**
    - **usl: tool a kapacitástervezéshez**

Mysql

- Egyéb mysql specifikus eszközök:
  - innotop: innodb specifikus mysql monitoring tool
  - tuning-primer.sh: segít a szerver konfiguráció helyes beállításában
  - mysqltuner: ugyanez pepitában
- Proxyk:
  - Mysql Proxy: lua-ban írt mysq proxy, támogat read-write splittinget, és particionálást.
  - Spock Proxy: gyakorlatilag Mysql Proxy klón, csak C/C++ a Lua helyett.
  - Mysql Cluster: az alkalmazás számára transzparens, hivatalos mysql cluster megoldás, NDB/NDBCLUSTER engine alapokon.

Mysql

- Teljesítmény optimalizációs technikák:
  - Az első lépés általában a gépbővítés, meg a normális configolás.
  - Majd mindenki schemát meg sql-t optimalizál.
  - Utána általában elkezdődik a logika áthelyezése a backendekre, ezáltal a query-k leegyszerűsítése, amivel tehermentesíthető az adatbázisszerver.
  - Ezzel kb. egyidőben lehet növelni az olvasási kapacitást replikák bevezetésével, de ez sajnos nem segít az írási műveleteken.
  - Egy idő után nem lehet növelni az írási kapacitást, ilyenkor általában jön a „bohóckodás” a shardolással, vagy valamilyen NoSQL megoldásra való áttérés.
  - Szóba kerülhet a MySQL Cluster-re való migrálás is, ennek vannak bizonyos megkötései, de az alkalmazás szempontjából még ez jár a legkisebb látható változással.

PHP

- Az első és legfontosabb lépés a php alkalmazásunk teljesítményoptimalizálása kapcsán, egy opcode cache telepítése (apc, xcache, etc.).
- A következő lépés az kellene hogy legyen, hogy minden hibát, warningot kijavítunk a kódban, mert a php sajátossága, hogy hiába van elnyomva, figyelmen kívül hagyva egy hiba, akkor is lefut rá a teljes belső hibakezelés, stack trace generálás, etc.
- Az include_path –ünk legyen mindig jól beállítva, és mindig helyesen hivatkozzunk a fájlok/könyvtárak elérési útjaira.
- open_basedir szintén elég negatív hatással bír lenni a teljesítményre, minden könyvtárra végigstat-olná rekurzívan az összes parent könyvtárat is (jelenleg úgy tűnik, hogy egy bug miatt ezt mindig így csinálja, de realpath_cache_size, realpath_cache_ttl növelése segíthet).

PHP

- Jelentősen meg tudjuk dobni a teljesítményt, ha a direkt include/require helyett autoloaderrel töltjük be a nem minden lapletöltésnél behúzandó fájljainkat.
- Ha viszont mindig szükségünk van arra a fájlra, akkor gondolkozzunk el rajta, hogy összebuildeljük ezeket a fájlokat egy darab fájlba, ez is sokat tud gyorsítani a végrehajtáson.
- Serializalás valamint a serializált string validálása alapesetben lassú, ne vigyük túlzásba ezt (ez vonatkozik mind a serialize, mind a json_encode, mind a xml függvények túlzott használatára).
- Legtöbb time() hívás kiváltható egy $_SERVER[’REQUEST_TIME’] változóval.

PHP

- Gyakran hívott metódusok eredményeit cachelhetjük egy statikus változóba, ugyanígy az adatbázis lekérdezések eredményeit is tehetjük mind ide, mind akár lapletöltéseken átívelően egy memcache vagy redis tárolóba.
- Ha nem használunk túl sok dinamikus nyelvi feature-t, illetve extensiont, akkor szóba jöhet még akár a HipHop használata is, de azért ez még eléggé kísérleti terep.
- Érdemes odafigyelni, hogy bár a memcache/redis nagyságrendileg gyorsabb, mint a mysql, de ha sok apró kérésre használjuk, akkor a hálózati latency itt is lassíthat, ez ellen kétféleképpen védekezhetünk:
  - Multiget feature használatával batch műveletben kérdezzük le a key-value store-ból az értékeket.
  - Lokálisan, hálózat helyet socket-en keresztül csatlakozunk (Redis slave.)

Összefoglalás

- Eléggé pörgős terület a teljesítményoptimalizálás, rengeteg új dolog jön ki nap, mint nap, emiatt nem is könnyű vele lépést tartani, de néhány egyszerű szabály örök érvényű.
- Nincs gyorsabb, mint ha nem csinálsz semmit.
- Ha csinálsz valamit, akkor lehetőleg memóriában csináld.
- Rakd a rendszereidet a lehető legközelebb egymáshoz, és batch kérésekben kommunikáljanak egymással, ezáltal minimalizálva a latency-ből eredő veszteségeket.
- A felhasználó felé legyen minden a lehető leggyorsabb, sokszor megengedhetjük, hogy ez akár a tartalom „frissességének” a rovására is menjen.
- Óvakodj a túlságosan komplex rendszerektől, nehéz túljárni egy olyan rendszer eszén, ami okosabb mint a fejlesztője ☺
- Ne foglalkozz túl sokat az optimalizációval addig, amíg nem működik a rendszered.

Vég(r)e!

Kérdések?

Köszönöm a figyelmet!

Kapcsolat

- tyra3l@gmail.com
- http://tyrael.hu/
- http://twitter.hu/tyr43l
- http://slideshare.net/tyrael