ÓBUDAI EGYETEM

Keleti Károly Gazdasági Kar
Vállalkozásfejlesztés és Infokommunikációs Intézet

SZAKDOLGOZAT

ÓE-KGK
2023

Hallgató neve: **Biró Adorján Péter**
Hallgató törzskönyvi száma: T007947/FI12904/G

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Keleti Károly Gazdasági Kar
Vállalkozásfejlesztés és Infokommunikációs Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Biró Adorján Péter
Szakdolgozat száma: [redacted]
Törzskönyvi száma: T007947/FI12904/G **Neptun kódja:** [redacted]

Szak: Gazdaságinformatikus
Specializáció: Vállalatinformatikai specializáció

A dolgozat címe: Gyorsítótár egyedi alkalmazása egy vizsgáztatórendszer optimalizálásához
A dolgozat címe angolul: Applying a custom cache to optimize an examination software
A feladat részletezése:
1. Készítsen kutatást a gyorsítótárak szükségességéről és alkalmazhatóságáról a vizsgáztatórendszerek kapcsán!
2. Szakirodalom alapján mutassa be a gyorsítótárak működését és vizsgálja meg a lehetséges alternatívákat!
3. Készítsen empirikus kutatást az egyedi problémára a kigyűjtött adatok felhasználásával!
4. Értelmezze az elért eredményeket, adjon meg javaslatokat azok alapján!

Intézményi konzulens neve: Dr. Szikora Péter Gábor

A kiadott téma elévülési határideje:2024. 12. 15.
Beadási határidő: 2023. 12. 15.

A szakdolgozat: Nem titkos.

Óbudai Egyetem Keleti Károly Gazdasági Kar Vállalkozásfejlesztés és Infokommunikációs Intézet 1. Budapest P.H.

Kiadva: Budapest, 2023. 11. 07.

.....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom:

.....
belső konzulens

.....
külső konzulens

HALLGATÓI NYILATKOZAT

Alulírott Biró Adorján Péter (Neptunkód: [redacted]) hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Budapest, 2023.12.12.

Biró Adorján

hallgató aláírása

Tartalomjegyzék

BEVEZETÉS

Napjainkban az egész világon, így Magyarországon is egyre inkább elterjedtek az elektronikus oktatást támogató rendszerek. Hazánkban az összes köz- és felsőoktatási intézmény valamilyen formában használ már *e-learning* platformokat, melyeket már nem csak a fiatalabb korosztályok veszik előszeretettel igénybe. Ezek használata egyre elterjedtebb, hiszen már lassan majdnem minden ember rendelkezik okostelefonnal, számítógéppel, azaz olyan digitális eszközzel, melyek rendelkezésre állása lehetővé teszi az online megtalálható oktató anyagok feldolgozását.

Ez a növekvő tendencia egyre nagyobb igényt teremt az elektronikus számonkérést biztosító ezközök fejlesztésére is. Ezen megoldások használata akkor válik meglehetősen bonyolulttá, amikor már a bizonyító erejű adatok szolgáltatása is elvárásként fogalmazódik meg velük szemben. Eddig a pontig csak egyszerű kérdőívszerkesztő alkalmazásokról beszélhetünk, amelyek feladata csupán annyi, hogy valamilyen kevésbé bonyolult formátumban rendelkezésre bocsájtsa a kitöltő személyek válaszait. Abban az esetben azonban, amikor már ezeken az eredményeken több múlhat, ezeknek a weben elérhető szoftvereknek komoly elvárásoknak kell megfelelniük, mint többek között a résztvevők jogosultságának ellenőrzése, valamint a kitöltők tevékenységeinek szigorú naplózása. A felmerülő problémák kezelése érdekében érthető módon egyre komplexebb megoldások látnak napvilágot, melyek működtetése komoly erőforrás igényt generál, valamint ezen erőforrások optimalizálásának megvalósítása problémáját veti föl.

Az elektronikus oktatás és az ezzel szorosan összefüggő bizonyos szervezetek számára elengedhetetlen elektronikus számonkérés, mint folyamat, számottevő mértékű adat létrejöttét és mozgását idézi elő, melyek tárolásának és manipulálásának kihívásai a mostani digitális korszak egyik legnagyobb nehézsége. Ezt a problémát megoldandó, az informatika majdnem összes területén új technológiák jönnek létre, valamint a már meglévő módszerek folyamatos javítása, finomhangolása zajlik. A terület fejlődésének talán egyik legfontosabb eszköze a szűk keresztmetszetek, illetve a kapacitás csökkenés feltárása és ezek megszüntetése, mely procedúrát manapság a legtöbbször különböző analitikai szoftverekkel végzik.

A vizsgáztató rendszerek témakörében a kapacitáshiány megjelenése a hatalmas mennyiségben képződő adatok keletkezése nyomán alakul ki. Az adatállományok hatékony kezelésére számos megoldást fejlesztettek ki, melyek közül – aki az egyre növekvő elvárások mellett érvényesülni akar – a létező összes technológiát egyidejűleg, megfelelő arányban kell igénybe vennie. Az utóbbi időben a Magyarországon végbemenő társadalmi, technológiai változások, és a pandémiás helyzet kialakulása miatt a távoktatás és az elektronikus számonkérés felé támasztott kapacitásigény nagy mértékben megnövekedett.

A szakdolgozatomban elsőként bemutatom a távoktatás történetét magyarországi és világviszonylatban, ismertetem a témakörbe tartozó alapvető fogalmakat és a távoktatásnak a dolgozat fő témájával való kapcsolatát. Ismertetem röviden a pandémiás időszak hazai online oktatásra gyakorolt rövid- és hosszútávú hatásait. Ezek után bemutatom a szakdolgozatban fölvetett problémát és ennek feltételezett megoldását. A későbbiekben bevezetem a gyorsítótár fogalmát, valamint behatóan feltárom annak működését, illetve részletezem a típusait, változatait.

A szakdolgozat további részében bemutatom az általam készített tesztprogramot, mely az előzőekben a felvázolt problémának egy lehetséges megoldását hivatott szemléltetni. Ezen felül a lebonyolított kísérlet által kapott eredményeket elemzem, melyekből levont konklúziók által javaslatot adok a kérdéses probléma teljeskörű orvoslására, valamint bemutatok néhány alternatívát, melyek bizonyos mértékben kész megoldást kínálnak a vizsgáztató szoftverek témakörében fellépő nehézségek elhárítására.

A szakdolgozat elkészítésében segítséget nyújt számomra az *Unipoll* vizsgáztató alkalmazást készítő fejlesztői csoport. Mind technológiai, mind tapasztalati úton segítik a kutató munkám és az általam megvalósítandó tesztalkalmazás létrejöttét.

1. KÉSZÍTSEN KUTATÁST A GYORSÍTÓTÁRAK SZÜKSÉGESSÉGÉRŐL ÉS ALKALMAZHATÓSÁGÁRÓL A VIZSGÁZTATÓ RENDSZEREK KAPCSÁN!

Az alábbi fejezetben megvizsgálom a vizsgáztató rendszerek megjelenésének előzményeit és okait globális és lokális (magyarországi) viszonylatban. Röviden leírom a működésük mibenlétét, valamint elemzem a hiányosságaikat és ezen hiányosságok következményeit. Ezen felül kitérek arra is, hogy a vizsgáztató alkalmazások bizonyos problémáira milyen mértékben, és miként jelent megoldást a gyorsítótárak alkalmazása.

1.1 Távoktatás és elektronikus számonkérés kapcsolata

A távoktatás egy technológiai koncepció, amelynek lényege, hogy az ismeretanyag feldolgozására, elsajátítására vállalkozó személyek időben és térben különböző és kötetlen, választásuknak leginkább megfelelő módon vegyenek részt egy tanulási folyamatban, mely folyamat során személyesen nem találkoznak az oktatóikkal. Ezen módszer sarkalatos pontja a technológia eszközök hatékony használata. Ha a távoktatás koncepcióját kiegészítjük azzal, hogy mindenki számára elérhetővé tesszük, akkor a távoktatást ezáltal az úgynevezett nyílt vagy nyitott oktatás *(open learning)* metódusává terjesztjük ki (Monolescu – Schifter, 2003; Holmberg, 1995; Bates, 2005; Moore – Anderson, 2003).

A távoktatás *(distance education)* első feljegyzett változata a 19. század végén jelent meg az Amerikai Egyesült Államok területén (Penn State University). Ekkor még csak kizárólag papír alapon működött az elgondolás, majd a későbbiekben különböző technológiák létrejöttével, előbb a rádió (1920-as évek), a televízió (1950-es évek), később kábeltelevízió (1970-es évek), majd videókonferencia (1980-as évek) használatával egyre többen be tudtak kapcsolódni a különféle távolsági képzésekbe (Monolescu – Schifter, 2003).

A szakirodalom a távoktatás lebonyolításának módját két alapvető csoportra osztja. Az egyik az egyirányú, azaz aszinkron változat, ahol nincs kommunikáció a tananyagot átadók és készítők, valamint az azt elsajátítók között, illetve nincs lehetőség személyre szabott foglalkozásra. Ebben az esetben az oktatást előkészítő szervezet vagy személyek a tervezésre fektetnek nagy hangsúlyt, hiszen a hallgatók, tanulók visszacsatolására

egyáltalán nem építhetnek. A másik változat alapvetése a kétirányú dialógus a képzésben résztvevők között, kölcsönös interakciók segítségével alakítják az oktatás menetét és javítják a minőségét (Holmberg, 1995).

Egy másik, némileg vitatott tagolás szerint a távoktatás fejlődését három osztályra vagy generációra lehet felosztani.

Az első generáció esetében a képzés főként egyetlen technológiára fókuszál, emellett nélkülözi a személyes kapcsolatteremtést a tanulók és tanárok között (aszinkron), valamint a foglalkozás végén nem kapnak a résztvevők semmilyen bizonyítékot arra vonatkozóan, hogy sikeresen elvégezték a képzést. Ebbe a csoportba tartoznak a szaklapok által közétett, továbbá az oktató jellegű televíziós műsorokon és a rádión keresztül sugárzott fejlesztő tudásanyagok. Itt még nem beszélhetünk semmilyen jellegű számonkérésről, hiszen a tárgyalt képzés teljesen egy irányú, nincs visszacsatolás az olvasók, hallgatók felé (Bates, 2005).

A második generációja a távoktatásnak előre megfontoltan, több médium használtára épít, amely a nyomtatott és sugárzott forrásokat használja fel. A tudásanyagot úgy alakították ki, hogy a távoktatás céljainak a leginkább megfeleljen. Ebben az esetben már valamilyen szinten megvalósul a kétirányú kommunikáció az oktatók révén – akik jellemzően nem a tananyag készítői –, habár általában a cél az volt, hogy elkerüljék, legalábbis minimalizálják a szereplők közötti kontaktust. A második generációs távoktatás sok esetben nagyon nagyszámú embert szolgál ki, akár százezreket is egy meglehetősen költséghatékony módon. A vizsgáztatás, valamint a szigorú keretek között történő számonkérés, ha valamilyen szinten meg is valósul, itt még a technológia adottságok nem teszik lehetővé az elektronikus úton történő számonkérést. A második generációs távoktatást másképpen ipari természetű távoktatásnak is nevezik (Bates, 2005).

A harmadik generáció merőben más alapokon nyugszik, mint az előző kettő, tekintettel arra, hogy ebben az esetben a kétirányú kommunikációra helyeződik a hangsúly. Ezt a fajta információcserét az internet betörésével megjelenő kommunikációs platformok teszik lehetővé, ide sorolva a weboldalak használatának, valamint a videókonferenciák létesítésének lehetőségét. Ezek, az oktatásban újszerű technológiák hozzájárulnak a tanulók és a tanítók, és – talán még fontosabb – a tanulók egymás közötti akadálytalan,

nagy távolságból is létesíthető kommunikációjához is, amelyeket csoportosan és egyénileg is le lehet bonyolítani. Ennek a generációnak a jellemzője, hogy kicsi, független csoportokban történő, a felek párbeszédén alapuló, flexibilis képzési folyamat alakul ki. A fentiekben leírt módszert alkalmazó szervezetek közé tartoznak a hagyományos értelemben vett egyetemek, kisebb különböző képzésekkel foglalkozó szervezetek. Ezt a harmadik generációt tudásalapú *(knowledge-base),* vagy posztindusztriális *(post-industrial)* távoktatásnak is nevezik (Bates, 2005).

Az *e-learning* fogalom jelentése lényegesen többet takar az *online* oktatásnál, lényegében minden tanulási forma, amely elektronikus segédeszközöket vesz igénybe az elektronikus oktatás témaköréhez tartozik (Naidu, 2006).

Az *e-learning* fogalma az 1990-es években jelenik meg először az Egyesült Államokban, valamint az évtized végén Európában is. Ez utóbbi tanulási metódus esetén beszélhetünk először elektronikus úton, lokális hálózat vagy internetkapcsolat felhasználásával történő számonkérésről, vizsgáztatásról, hiszen ennek a lehetőségnek a kiaknázásához ettől fogva hozzáférhető bárki számára a megfelelő technikai környezet. Napjainkban azonban ezen módon történő tudás ellenőrzést az egy- és kétirányú (szinkron – aszinkron) távoktatás során is előszeretettel alkalmazzák. (Holmberg, 1995; Moore – Anderson 2003).

1.2 Távoktatás hazánkban

Magyarországon a távoktatás az 1950-es évektől kezdődően jelent meg, melynek első formái a felsőoktatásban induló levelező tagozatok voltak. Később távoktatási módszertani kísérletek folytak Pécsett 1973 és 1980 között. A próbálkozás a Tudományos Ismeretterjesztő Társulat (TIT), valamint a Magyar Televízió és Rádió közreműködésével zajlott. 1974-ben Távoktatás Konferencia zajlott Tihanyban, majd 1976-ban Távoktatási Szakértői Tanácskozás tartottak Sopronban. A magánszférában az 1980-as években jelent meg a távoktatás. 1989 után egyre több felsőoktatási intézmény vezetett be távoktatás segítségével működő tagozatot. 1991-ben létrejön a Nemzeti Távoktatási Tanács (NTT), mely célja a hazai távoktatás támogatása, koordinálása ösztönzése volt, majd 1994-ben Budapesten megalakult a Nemzeti Távoktatási Központ (NTK). A Duna Televízió 1996-tól közvetített távoktatási műsorokat (Lengyel, 2007).

E-learning rendszereket és hozzájuk kapcsolható vizsgáztató rendszereket a magyar felsőoktatásban több, mint egy évtizede – már 2006-tól – alkalmaztak, ám használatukra komoly igény csak a koronavírus világjárvány Magyarországra való betörése után alakult ki. A „karantén-korszak" előtt, mintegy alternatívaként kezelték az online térben való oktatást és számonkérést, használatunk azonban nem volt kötelező, sok esetben legfeljebb egy, bizonyos esetekben kézenfekvő eszközként tekintettek rá. A vizsgáztató rendszereknek megvoltak, jelenleg is megvannak a maga korlátai, azonban meghatározott keretek között nagyon is hasznosnak bizonyultak, kisebb csoportok esetében előszeretettel vették igénybe a jelenléti, tantermi számokérés akadályoztatása esetén (Lannert, 2022).

A pandémia hazánkba való betörése miatt az online oktatás fontos részeként a vizsgáztató rendszerek nélkülözhetetlen elemei lettek a köz- és felsőoktatásnak. Ez idő tájt az oktatási rendszer működésénekegyetlen fennmaradó alternatívája a távoktatás volt, melyet határozatlan időre kötelezővé is tettek. Ekkor az igény olyannyira megugrott az online oktatást és vizsgáztatást kiszolgáló szoftverekre és web szolgáltatásokra, hogy a már meglévő termékek és szolgáltatók nem is tudták maradéktalanul, valamint zökkenőmentesen kiszolgálni a hirtelen megnőtt keresletet. A vírushelyzet után a távoktatás népszerűsége, és felhasználhatósága jelentős mértékben nőtt Magyarországon, így a továbbiakban lényeges részévé vált – különösképp a felsőoktatásban és egyéb felnőtképzési platformok esetében – a hazai oktatásnak. (Lannert, 2022)

1.3 A vizsgáztató rendszerek és a koronavírus vizsgáztató rendszerekre gyakorolt hatása

Az elektronikus vizsgáztató rendszerek olyan szoftverek, melyek segítségével az oktatóknak lehetőségük van lebonyolítani a tanulók, hallgatók számonkérését, értékelését egy tantárgy vagy kurzus ismeretanyagára vonatkozóan. Ezeknek a rendszereknek hiteles információt kell szolgáltatniuk a kérdőívek kitöltésének eredményeiről, a kitöltők válaszairól, hiszen például a felsőoktatási intézmények esetében a hallgatóknak ezen múlhat a félév végi eredménye. Ez utóbbi miatt tökéletesen megbízhatóan és konzisztens módon kell működniük minden egyes alkalommal. Bonyolultságuk emellett abban is rejlik, hogy egy ilyen jellegű szoftver

akkor lesz széleskörűen alkalmazható, ha számos különböző kompetencia, szakismeret számonkérésére nyílik lehetőség alkalmazásuk által. Ehhez azonban nagyszámú, eltérő kérdéstípust és válaszadási módot kell támogatnia. Ezen felül a használat során további széleskörű dokumentációt is kell generálnia az applikációnak a visszakövethetőség miatt, valamint nem utolsó sorban adat- és információbiztonsági okokból kifolyólag is. Ezeket a vizsgáztató programokat különféle oktatási intézmények és gazdasági szervezetek is igénybe veszik, mely szoftvereket egymástól gyakran eltérő és egyedi módon alkalmazzák. Vannak szervezetek, ahol nagy hangsúly helyeződik a kitöltők anonimitására, máshol pedig egyedi szolgáltatásokat, elvárásokat kérnek a szoftver működtetőjétől, készítőjétől, úgy, mint adatfeldolgozás, kérdőív lebonyolítása, egyedi megjelenés készítése. Az elektronikus úton történő számonkérésnek szóbeli változata is van, ám ez a kérdéskör nem kapcsolódik szervesen a szakdolgozatom témájához, ezért bővebb kifejtésére jelen dokumentumban nem fogok sort keríteni (Reddy, 2023).

A koronavírus hatásaként az *e-learning* és a hozzá kapcsolható vizsgáztató rendszerek leterheltsége rövid időn belül nagyságrendekkel növekedett meg. Az elsődleges probléma a robbanás-szerűen kibővült felhasználói kör méretével és az általa generált többlethasználattal az volt, hogy a fent említett rendszereket nem készítették fel ilyen méretű terhelés elviselésére és ennek okán komoly változtatások kieszközölése vált szükségessé (Jose, 2021).

A vizsgáztató szoftverek témakörében legtöbb esetben a szűk keresztmetszetet az úgynevezett párhuzamos vagy szimultán kitöltés száma jelenti, mely lényegében azt a mennyiséget összegzi, amely megmutatja, hogy egyszerre hány felhasználó képes egyidejűleg kitölteni egy adott vizsgát vagy kérdőívet. Ezen felül kihívás elé állítja a kérdéses programokat az is, hogy részletes, gyors és megbízható adatfeldolgozást, valamint adatkiértékelést legyenek képesek lebonyolítani. A felsorolt problémák megvalósításának színvonala sok mindentől függhet, többek között az informatikai eszközök specifikációjától, a hálózati kapcsolatok sávszélességétől, és a felhasznált szoftverelemek hatékonyságától is. (Yang, 2022)

A teljesítmény vagy kapacitás a számítástechnikai eszközöktől való függése egy jóval komplexebb probléma, mint azt elsőre gondolnánk. Egy vizsgáztató szoftver általában több szoftverkomponensből áll, melyeknek más és más erőforrásigényük van. Ezen

erőforrások alatt a hardverek, szerverek és összetevőik teljesítményét értjük, amelyeken a szóban forgó vizsgáztató rendszer részei futnak. A fent említett igények mérése sokszor nem egyszerű feladat, hiszen ezeket vizsgálva számos esetben azt tapasztalhatjuk, hogy a szoftverkomponensek futásának sebessége nagyban függ a működésük közben történő, egymásra gyakorolt hatásuktól. A kérdéses számokérést támogató rendszereket gyakran több, egymástól független számítógépen (hardveren) szimultán futtatják, annak érdekében, hogy megfelelő kapacitást tudjanak szolgáltatni nagy számú felhasználó esetén, illetve annak érdekében is, hogy a folyamatos rendelkezésreállást tudják biztosítani: tehát, ha valamelyik gép meghibásodik, vagy bármilyen oknál fogva nem tud megfelelően üzemelni, legyen tartalék eszköz annak helyettesítésére. Bizonyos esetekben az adott szoftveregyüttes elemei különböző készülékeken futnak a megfelelő teljesítmény biztosítása, valamint a biztonságos működés érdekében. Az egymástól esetlegesen elszeparálva operáló komponensek jellemzően a web és adatbázisszerverek, ezenfelül a kiegészítő szolgáltatásért felelős modulok például eltérő feladatok ütemezését végző, különféle kimutatásokat készítő szegmensek. A fentiek alapján belátható, hogy abban az esetben, ha csak a szoftveregyüttes futásáért felelős eszközöket vizsgáljuk, akkor is meglehetősen időigényes és bonyolult feladat meghatározni, hogy a kapacitásbővítéshez, azaz a párhuzamosan történő kitöltések számának növeléséhez pontosan melyik berendezés, melyik összetevőjének teljesítményét kell fokozni. A probléma megoldásához sok esetben a terméket fejlesztő csapatnak érdemes együtt dolgozni a témát jól átlátó szakértőkkel (Yang, 2022; Hammar, 2017).

A hálózati kapcsolatok sávszélessége is nagyban befolyásolhatja egy vizsgáztató rendszer kapacitását, ám ez a szolgáltatást igénybe vevő intézmény technológiai felkészültségétől, eszközparkjának fejlettségétől és földrajzi elhelyezkedésétől nagyobb mértékben függ, mint a szoftver működésének hatékonyságától. Az elsődleges szűk keresztmetszetet az internetszolgáltató által kínált hálózati hozzáférés minőségének és teljesítményének mértéke jelenti. Ez egy olyan változó, amelyet sem az alkalmazást készítő vállalkozás, sem a felhasználó szervezet nem tud nagy mértékben befolyásolni. Az adatátviteli sebességet továbbá befolyásolhatja, hogy a kérdéses intézmény belső hálózati kommunikációjáért felelős eszközei, szoftverei mennyire korszerűek, valamint

az is, hogy rendelkezik-e a társaság megfelelő számú és szakképzetségű, a hálózat működtetésért felelős alkalmazottal (Hammar, 2017).

A fejlesztői csapat munkája értelemszerűen jelentősen befolyásolja egy vizsgáztató rendszer működésének hatékonyságát. Az alkalmazás készítésekor az üzleti logikát lebonyolító kód futásának sebessége számottevően hat a program kapacitására, mely erőteljesen függ a szoftver tervezésének minőségétől, valamint a felhasznált technológiák, fejlesztési minták, koncepciók korszerűségétől, és ezek megfelelő implementálásától. Talán ennél is hangsúlyosabb tényező a vizsgáztató rendszer és annak adatbázis szerverei közötti kommunikáció, az adatok áramlásának, tárolásának és kezelésének hatékonysága, azaz az adatok lekérdezésének *(select)*, új rekordok hozzáadásának *(insert)*, az adatok módosításának (*update*), valamint ezen adatok törlésének (*delete*) módja. Az imént felsorolt műveleteket együttesen adatbázis-szkripteknek (*script*) nevezzük. Hatékony adatbázis-szkriptek írásával az adatbázis szerverek válaszideje, valamint írási és olvasási sebessége nagy mértékben csökkenthető, mely tényező jelentősen hozzájárulhat az alkalmazás hatékony működéséhez. Az adatbázisok manipulálásának témaköre egy önálló szakterülete az informatikának, ezért szinte nélkülözhetetlen, hogy az olyan szoftvertermékek kialakításakor, amelyek használatához elengedhetetlen nagy mennyiségű adat tárolása és feldolgozása, – így egy vizsgáztató rendszer esetében is – rendelkezzen különálló munkacsoporttal, mely erre a szakterületre van specializálva. Egy „átlagos" szoftverfejlesztőnek is kell érteni az adatbázis-szkriptek írásához, ám jellemzően közel sem tud olyan hatékony megoldásokat készíteni, mint azok a személyek, akik a kérdéses témakörre specializálódtak (Fowler – Wesley, 2002; Adel F, 2019).

A vizsgáztató rendszerek a legtöbb esetben rétegzett alkalmazások, jellemzően rendelkeznek megjelenítési réteggel (*presentation layer*), üzleti logikai réteggel (*logic/domain tier/layer*), és adat réteggel (*data tier/layer*). A megjelenítési réteg felelős a felhasználók és a szoftver közötti interakciók lebonyolításáért. Elsődleges funkciója az információk megjelenítése az felhasználók számára, valamint a felhasználók által adott utasítások az üzleti logikai réteg számára feldolgozható információkká alakítása. Az üzleti logikai réteg a megjelenítési rétegtől kapott információkon számításokat végez, feldolgozza, hitelesíti (validálja) azokat, ezen felül az adatrétegből kapott és a saját maga által feldolgozott adatokat továbbítja a megjelenítési rétegnek. Az adatrétegbe

történő információk mentéséért is ez a réteg felelős. Az adatréteg legtöbbször egy relációs adatbázissal való kommunikációt végzi, mely réteg biztosítja, hogy az adatok megfelelő struktúrában legyenek eltárolva. Előfordul azonban, hogy az üzleti logikai réteget két részre osztják: alkalmazásrétegre és üzleti modell rétegre. Az alkalmazásréteg egy API-ként (*Application Programming Interface*) funkcionál illetve. A *tier* és *layer* kifejezések között a lényegi különbség az, hogy a *tier* fizikailag elkülönített réteg – konkrétan különböző hardveren fut a réteg –, a *layer* viszont csak logikai elkülönítettséget jelent (Fowler – Wesley, 2002).

1.4 A gyorsítótár által megoldani kívánt probléma

A többrétegű alkalmazások esetében legtöbbször a kapacitás szűk keresztmetszete a relációs adatbázis szerverek válaszideje. A kérdéses architektúrájú szoftverek esetén a felhasználói interakciók hatására esetenként akár több tíz vagy száz adatbázis lekérdezést is kezdeményezhet az üzleti logikai alkalmazás, melyek aztán nagy mértékben leterhelik az adatbázis szervereket, lassítva az írási és olvasási sebességüket. Az adatbázis szerverek általában véve drága, nagy teljesítményű szerverek, melyekből egy alkalmazás működése során jellemzően csak egyet darabot használnak. Ezekből a nagy kapacitású eszközökből csak a nagyon sok felhasználó által használt szoftverek esetében használnak többet. Abban az esetben, ha nagyobb kapacitásra van szükség, elsősorban az alkalmazás szerverek számát szokták növelni, amelyek viszonylag olcsóbb eszközök (Larson – Goldstein – Zhou 2004). Ez a megoldás nem feltétlenül teszi lehetővé a skálázhatóságot, valamint, csak addig működik, amíg az adatbázis szerver megfelelő mennyiségű erőforrással rendelkezik az alkalmazás szerverek kiszolgálásához (Yang, 2022).

Az alkalmazás szerverek számának növelésével az elosztott működésüket is lehetővé kell tenni, azaz a felhasználó kéréseit valamilyen módon el kell osztani a szerverek között. Ezen utóbbi problémára jelent megoldást az úgynevezett *load balancing*, mely egy olyan eljárás, amely során a terhelés újraelosztásra kerül egy elosztott rendszerben a csomópontok (szerverek) között az erőforrás hasznosítás és a válaszidő javítása érdekében. Használata továbbá segít elkerülni az olyan állapotot is, amikor bizonyos csomópontok erősen terheltek míg mások kihasználatlanok. Legtöbb esetben a *load*

balancer egy speciális, erre a célra kialakított hardveren vagy virtuális eszközön futó algoritmus (Alakeel, 2010).

A kisebb szoftverek esetén az üzemeltetők lehetőség szerint elkerülik az újabb adatbázis szerverek igénybevételét, hiszen ezek bevezetése komoly költségekkel jár, ezen felül azért is próbálják mellőzni ezt a megoldást, mert elosztott módon működő használatuk megvalósítása rendkívül komplikált. Sok esetben egyéb, skálázható módon próbálják lehetővé tenni a nagyobb kapacitás elérését. A legnépszerűbb megoldás a gyorsítótár (*cache*) használata, amely viszonylag olcsón, az alkalmazás komolyabb megváltoztatása nélkül képes számottevő teljesítmény növekedés elérésére. Az eljárásnak számos különböző típusa van, melyek a szoftver működési elvétől függően alkalmazhatóak. A *cache* kifejezés alatt nagyon sok mindent érthetünk, ezért érdemes itt leszögezni, hogy a vizsgáztató rendszerek témakörében nem a processzorok működéséhez kapcsolódó kis méretű, gyors elérésű adattárolóról van szó, hanem a rétegzett alkalmazások adatmanipulációs folyamatait optimalizáló, főként szoftver jellegű megoldásáról. Az utóbb említett témához kapcsolódó gyorsítótár esetében is több fajtáról lehet beszélni, melyek típusait a következő fejezetben részletekbe menően ki fogom fejteni (Modi, 2021)

Ahogy az a szakdolgozatomban már említésre került, a vizsgáztató rendszerek esetében legtöbbször a párhuzamos kitöltések maximális száma mutatja meg, hogy milyen mértékű terhelés mellett képes még az alkalmazás a folytonos és megbízható működésre. Természetesen érdemes még más mészőrszámokat és a követelmények teljesülését is megvizsgálni, ám a szimultán kitöltések számossága a legmérvadóbb aspektus, mely elsősorban az adatbázis szerver és az alkalmazás szerverek kapacitásától függ, illetve – ha ez fennál – ezek elosztott működésének hatékonyságától. Az említett szerverek teljesítménye a hardver összetevőik specifikációján múlik. Ezen felül nagy kapacitásnövekedést eredményezhet a gyorsítótárak használata, illetve elosztott rendszerek esetén használatos változatai segítik elkerülni az adatok inkonzisztens állapotát, továbbá lehetővé teszik az erőforrások nagymértékű skálázhatóságát. (Paneri, 2023)

Egy vizsgáztató rendszer esetén a gyorsítótárak használta elsősorban az adatbázis szerver felé érkező kérések számának csökkentését eredményezi olyan módon, hogy a

közelmúltban egy vagy több felhasználó által kért adatok, objektumok (például kérdőíveket) ideiglenesen egy gyorsabban elérhető tárhelyen (memória) kerülnek eltárolásra, így egyrészt az adatbázis szervernek nem kell újra és újra azonos lekérdezések alapján összeválogatni a megfelelő memóriacímekről a kért információkat, másrészt az adatkérés gyorsabban végbemegy az adat „közelsége” miatt. Az, hogy fizikailag hol, milyen típusú tárhelyen, milyen formában és hogy mennyi ideig kerül tárolásra az adat, az a *cache* típusától, kialakításától és beállításaitól nagyban függ. Az eljárás vésző soron azt eredményezi, hogy minimális erőforrás-bővítés mellett a vizsgáztatórendszer kapacitása nagy mértékben növekszik, ilyen módon több felhasználó számára teszi lehetővé egy adott kérdőív vagy vizsga kitöltését. (Modi, 2021; Paneri, 2023; Yang, 2022)

2. SZAKIRODALOM ALAPJÁN MUTASSA BE A GYORSÍTÓTÁRAK MŰKÖDÉSÉT ÉS VIZSGÁLJA MEG A LEHETSÉGES ALTERNATÍVÁKAT!

Ebben a fejezetben szakirodalom felhasználásával bemutatom a különböző típusú gyorsítótárak működését és előnyeit. Ezen felül megvizsgálom, hogy egy vizsgáztató rendszer számára milyen fajta *cache* megoldások lehetnek hasznosak, melyek segítik elő kapacitásának növekedését.

2.1 A gyorsítótárak működése és típusai

A gyorsítótár (*cache*) egy kis méretű, gyors elérésű adattároló hardver vagy szoftver komponens. Az *cache*-n eltárolt adatok általában egy korábbi művelet eredményei, vagy lassabban elérhető adatok másolatai, melyek ideiglenes tárolásának célja, hogy egy meghatározott időablakon belül ezeket gyorsabban el lehessen érni. A gyorsítótárat használó alkalmazások a felhasználni kívánt adatokat először a gyorsítótár tárhelyén keresik, abban az esetben, ha kívánt információ nem lelhető fel itt, csak akkor folytatódik a keresés az adatbázis szervereken vagy egyéb lassabb elérésű háttértárakon. A *cache* használatának előnye, hogy csökkenti a hálózati sávszélesség felhasználását, a felhasználók által érzékelt várakozási időt, valamint az adatbázis szerver terhelését (Podlipnig – Böszörmenyi, 2003). A gyorsítótár működése során, ha egy kért adatcsomag a *cache* memóriájában megtalálható, akkor az alkalmazás nem indít legkérdezéseket az adatbázis szerver felé. Ezt úgy nevezi a szakirodalom, hogy *cache hit,* illetve *cache miss*-nek nevezi azt, amikor a kívánt tartalom nem található meg a gyorsítótárban. Ebben az utóbbi esetben az adatbázistól lekérdezésre kerülnek az adatok, és a *cache*-ben is letárolásra kerülnek. Amennyiben a szükséges információk megtalálhatóak a gyorsítótárban, abban az esetben sokkal gyorsabb adatelérés valósítható meg. A *cache* -ben és az adatbázis szerveren elért adatok arányát *cache hit ratio*-nak nevezi a szakirodalom. A gyorsítótárak annál hatékonyabbak, minél nagyobb ez az arányszám. Ennek az aránynak a maximalizálása többek között a felhasználók által frekventáltan látogatott tartalmak előzetes letöltésével, ezen kívül a gyorsítótárak tartalmának előre definiált kiürítésével, esetlegesen a felhasználók számára elérhető megosztott *cache* használatával valósítható meg (Liang – Xueqian – Yong – Mengting, 2020).

A gyorsítótárakat többféle felosztás szerint lehet csoportosítani, melyek közül talán a legelemibb felosztás alapján beszélhetünk hardver (*hardware cache*), hálózati (*In-network cache*) és szoftver (*software cache*) gyorsítótárakról. A hardver és a hálózati *cache* csoportokat nem lehet igazán további elméleti kategóriákra bontani, ezzel ellentétben a szoftver *cache*-t számos nagyobb osztályra lehet tagolni, attól függően, hogy egy alkalmazás melyik szegmensében működik, vagy melyik részére hat.

2.1.1. Hardver típusú cache-ek

A hardver *cache*-ek a legtöbb esetben valamilyen számításokat végző eszköz számára fenttartott kisméretű, ám annál gyorsabb elérésű adattároló memóriája. Ezek a *cache* megoldások nagyban hozzájárulnak a különböző processzorok számítási kapacitásához, használatuk csökkenti az adatok elérésének „árát" (idő, energia), így rendkívül hatékony adatelérést tesz számukra lehetővé. Ebben a megoldásban az adatok kétdimenziós tömbökben kerülnek eltárolásra, kulcs- és értékpárok formájában. A hardver gyorsítótárakat a szakirodalom csoportosítja elérési sebességük, tárolókapacitásuk (szintek: *L1, L2,* stb.), és funkciójuk alapján (például *D-cache, I-cache, TLB, MMU*) (BasuMallick, 2022; Torres, 2007)

2.1.2 In-network típusú cache

Az *In-network cache* működésének lényege, hogy a sűrűn látogatott internetes tartalmak eltárolásra kerülnek egy erre a célra kialakított gyorsítótár memóriájában, ezzel megszüntetve a redundáns adatforgalom kialakulását. Ezen típusú gyorsítótár segít optimalizálni a hálózati forgalmat, mely megoldás abban az esetben tud működni, ha a kérdéses hálózat rendelkezik a megfelelő hálózati eszközökkel, amelyek lehetővé teszik a megfelelő protokollok, és így a hálózati *cache* üzemelését (Cisco Network Caching 2000).

2.1.3 Szoftver típusú cache-ek

A szoftver *cache*-ek csoportjába tartozik talán a legtöbb gyorsítótár típus. A *cache*-ek ezen osztályába tartozó eljárások – ahogy a nevük is mutatja – valamilyen szoftvercsomagok, futtatható programok, amelyek nem feltétlenül igényelnek speciális hardvereket, jellemzően bármilyen személyi számítógépen vagy szervergépen képesek

futni. Ezeknek a megoldásoknak a nagy részét bármilyen vállalat vagy szervezet által használt webes, lokálisan futó vagy rétegzett alkalmazása képes felhasználni. A legtöbb, nagyobb felhasználói körrel rendelkező szoftver futása során szinte elengedhetetlen, hogy alkalmazzon is ilyen jellegű megoldásokat, ha megfelelő fokú rendelkezésre állás mellett szeretne magas színvonalú szolgáltatást nyújtani (Shinder, 2008; Fountis, 2017; Messaoud, 2009).

A *szoftver cache* megoldások egyik fajtája a *web cache*, mely magában foglalja a kliens oldali és a szerver oldali web gyorsítótárat. Ennek a megoldásnak a lényege, hogy webtartalmak (*HTML*, *css*, *Javascript*, kép, videó), azaz weboldalak reprezentációjának ideiglenes eltárolásával gyorsítja, folyamatosabbá teszi a felhasználók számára a böngészést (Shinder, 2008; Fountis, 2017).

A *web cache*-nek a kliens oldali megvalósítására a szakirodalom *browser cache* vagy *forward cache* elnevezéssel hivatkozik. Ezek a szoftverek a tartalom megosztók webszerverein kívül, tehát webböngészőkön, lokális hálózatokon, illetve az internet szolgáltatók (*ISP*) szerverein futnak. A sávszélesség-felhasználás csökkentése érdekében sűrűn felkeresett weboldalak állományait tárolják el, így amikor a hálózaton belüli – ideértve a böngészőket is – felhasználók felkeresik a *cache*-elt tartalmakat, nem indítanak felesleges kéréseket a webszerverek felé. A *cache* működése során, amikor egy felhasználó kérést indít annak érdekében, hogy egy webes tartalmat meg tudjon jeleníteni, akkor az adatok mentésre kerülnek a gyorsítótárban. Ha a lokális hálózatban vagy azonos böngészőn újból lekérik a már mentett tartalmat, akkor az a *cache*-ből kerül kiolvasásra, nem pedig a szolgáltató webszerveréről (Shinder, 2008).

A *reverse cache*, másnéven *gateway cache* vagy *surrogate cache* úgynevezett *proxy* szerverek beiktatásával szintén a webszervereken csökkenti, illetve a szolgáltató belső hálózatán próbálja mérsékelni a tartalom igénylés által megnövekedett terhelést. Az utóbb említett gyorsítótár megoldásnak továbbá hatalmas előnye még, hogy abban az esetben, ha a szolgáltató webszerverei leállnak, vagy csak minimális szolgáltatás kiesés fordul elő szoftverhiba, áramszünet, vagy akár rendszeres karbantartás miatt, abban az esetben is ki tudja szolgálni a hálózaton kívülről, az internetről érkező felhasználói adatigénylést. A *web cache* megoldások hatékonysága nagyban függhet a memóriájuk feltöltésének és kiürítésének rendszerességétől, módjától, melyeket különböző

komplikált – az adott tartalom megosztó vállalat működésétől függő – algoritmusok végzik, illetve eltérő *HTTP header*-ek használatával viszik véghez. Sokszor több *web cache* szervert is használnak elosztott rendszerekben, ilyen esetekben számos, az elosztott működést támogató protokollt vesznek igénybe. Ezek segítségével tudnak megfelelően kommunikálni és ezáltal hatékonyan üzemelni a *proxy* szerverek egymással párhuzamos módon (Shinder, 2008; Fountis, 2017; Messaoud, 2009).

A *disk cache* is a szoftver típusú gyorsítótárak osztályába tartozik, melynek funkciója, hogy személyi, illetve szerverszámítógépeken a lassabb háttértárakon a frekventáltan használt adatok számára gyorsabb elérést biztosítson. Másnéven *page cache*-nek is nevezik. Működése során a sűrűn kért adatokat, gyorsabb elérésű adattárolón, jellemzően a *RAM* (*Random-Access Memory*)-on tárolja el. Írási és olvasási műveletek során „lap" méretű kötegeket tárol, neve a szöveges fájlok írásának és olvasásának támogató folyamatából származik (Love, 2011; BasuMallic, 2022).

A *content deliver network* (*CDN*) szintén az említett kategóriába tartozó *cache* megoldás, mely eredetileg weboldalak elérhetőségét javította, statikus tartalmak gyorsítótárban való eltárolásával. Mára már eltérő magas minőségű dinamikus tartalmak, nagy felhasználói elérést produkáló, elosztott rendszerű webalkalmazások és valós idejű médiatartalmak megosztásában érdekelt weboldalak performanciájának növelését is támogatja. A megoldás hatékony működéséhez jelentős hardver (*edge*-szerver) támogatásra is szüksége van a *CDN* rendszernek. Az *edge*-szerver fogalom arra utal, hogy ezek a szerverek vagy szervercsoportok az internet és a végfelhasználók hálózatának határán helyezkednek el, melyek így a globális internethálózat irányából tekintve a legközelebb található a felhasználókhoz. Ez a *cache* eljárás nem csak a weboldalak kiszolgáló kapacitását növeli, hanem lehetővé teszi a különféle web és *stream*-szolgáltatások rendelkezésre állását, elérhetőségének megbízhatóságát, valamint hálózati tartalmak kiszolgálásának skálázhatóságát is. Fontos elve a tárgyalt gyorsítótár megoldásnak, hogy az elosztott kiszolgáló szerverek minél „közelebb" legyenek a végfelhasználókhoz. A közelség fogalom itt arra utal, hogy földrajz és hálózati topológia szempontjából is ideális esetben az összes felhasználó internet szolgáltatójának (*ISP*) hálózatában legyen egy a *CDN* rendszerét kiszolgáló szerver annak érdekében, hogy minél kevésbé kelljen a végfelhasználóknak a nagy távolságú adatkapcsolatokra hagyatkoznia. A kiszolgáló szervereket a szakirodalom *node*-oknak

nevezi, melyek számossága és a közöttük lévő minél sűrűbb kapcsolat lehetővé teszi az adatáramlási „utak” optimalizálását. Ezen *node*-ok vagy a csomópontok az átfolyó adatmennyiség folyamatos monitorozásával a hálózat működtetőinek lehetősége nyílik szükség esetén bármikor új hálózati „térképet” kialakítani, mely segítségével az adattovábbítás sebessége optimális marad. Az optimalizálás során nagy hangsúlyt fektetnek a hiba detektálásra, hiszen a hálózati komponensek bizonyos időközönként meghibásodhatnak (legfőképpen a hardver eszközök, melyeken belül a legnagyobb rendszerességgel a mozgó, kopó alkatrészekkel rendelkező berendezések). Ideális esetben, ha megfelelő mértékű a hálózati utak redundanciája, a végfelhasználók nem érzékelik az esetlegesen meghibásodó komponensek miatti kapacitás kiesést. A *CDN* rendszerű *cache* használatának célkitűzése, hogy a lehető legmegbízhatóbb, és legnagyobb kapacitású adatelérést tegyen lehetővé a költséghatékonyság fokozott figyelembevételével (Nygren – Sitaraman – Sun, 2012; Dilley et al., 2002).

A szoftver-*cache* megoldások egy hardver közelibb variánsa az úgynevezett *memoization,* amely egy optimalizációs eljárás, melynek folyamán költséges számítások eredményei kerülnek átmeneti letárolásra. Később, ha az adott függvény azonos bemeneti értékeket kap, akkor megtörténik a mentett eredmények kiolvasása. Az eljárás célja, hogy a számítástechnikai programok futását gyorsabbá tegyék vele. A *memoization* kifejezést először Donald Michie használta 1968-ban, felhasználását tekintve leggyakrabban dinamikus programozási algoritmusokat igénybe vevő architektúrák részeként alkalmazzák (Norvig 1991).

A felsorolt *cache* változatokon kívül még számos másik megvalósítás is létezik, amelyek működése azonban vagy nagyon hasonló elvek alapján valósul meg, vagy alkalmazási területük meglehetősen speciális, így a szakdolgozatomban nem kívántam ezeket feltüntetni. A bemutatott gyorsítótár típusok közötti határok sokszor elmosódnak, számos alkalommal több kategóriába is beletartozhatnak a megvalósítások, nem minden esetben lehet egy konkrét csoportba besorolni az adott *cache* megoldást. A szoftveres gyorsítótárak hatékony működéséhez a legtöbb esetben szükség van egyfajta hardveres támogatásra, értelemszerűen mindegyik típus valamilyen szerveren, vagy speciális hardvereszközön tud csak futni. Gyakran a különböző gyorsítótár implementációk párhuzamosan, egyszerre működnek. A modern számítástechnikai berendezések szinte már mind rendelkeznek valamilyen fizikai vagy szoftveres *cache* mechanizmussal,

ugyanígy az internet szolgáltatók is szinte kivétel nélkül igénybe veszik valamelyik típus legalább egyikét, de jellemzően szimultán, többet is (BasuMallic, 2022; Stasoz, 2021; Sandu, 2022).

Egy másik csoportosítás a gyorsítótárakat *In-memory caching*, azaz csak az eszköz saját memóriában való, vagy *Distribute caching*, tehát elosztott rendszerben történő adattárolási mód szerint osztja föl. Mind a kettő megoldás esetében a szerverszámítógép *RAM*-jában kerül letárolásra a *cache*-elni kívánt adatállomány. Az előzőleg tárgyalt felosztásban bemutatott gyorsítótár változatok közül bizonyos fajták a jelenlegi csoportosítás szerinti mindkét osztályába beletartoznak, mások csak az egyik architektúra fennállása esetén értelmezhetőek, megvalósíthatóak. Sok esetben a *cache* implementációk elosztott környezetben való alkalmazása az *In-memory* változat hálózaton való, emellett több *node vagy* csomópont általi kiterjesztéseként alakítható ki (Stasoz, 2021; Sandu, 2022).

Az *In-memory caching* alapvetése, hogy a gyorsabb szoftverműködés vagy tartalom megjelenítés elérése érdekében átmentileg eltárolandó adathalmaz a merevlemez, valamint adatbázis helyet a szervergépek *RAM*-jában kerül ideiglenes letárolásra, így biztosítva az állományok gyorsabb elérését az alkalmazások számára. Ezen megoldás olyan szoftverek esetén hasznos különösen, amelyek számára kritikus a lehető legrövidebb időn belül megtörténő adatelérés. Az *In-memory caching* számottevően javítja az alkalmazások teljesítményét azáltal, hogy a működésük során indított adatbázis kérések és a lemezolvasások számát jelentősen csökkenti. Egy erőteljes hátránya a lehet a megoldásnak, hogy a *RAM*-on lévő információk egy esetleges, nem tervezett áramkimaradás vagy az adott eszköz futását valamilyen okból megszakító esemény miatt elveszhetnek, hiszen könnyen lehet, hogy ezeknek az adatoknak biztonsági okokból történő redundáns tárolása nem megoldott (Stasoz, 2021; Sandu, 2022; Chengjian et al., 2015).

A *Distribute caching* vagy elosztott rendszerű gyorsítótár implementáció működése során számos szerver és csomópont hálózatán keresztül tart gyorsabban elérhető „helyen” adatállományokat. Ez a típusú *cache* megoldás lehetővé teszi, hogy a webalkalmazások teljesítménye skálázható módon legyen növelhető, illetve, hogy nagyobb mennyiségű adatot lehessen *cache*-elni. Jellemzően ezen típusú gyorsítótárak a

jelentős méretű adathalmazt kisebb részekre bontja, és az így kapott adat „szilánkokat” (*shard*) szétosztja a hálózat csomópontjai között, ezáltal az eltérő szerverek közötti terhelés is megosztásra kerül, így mérsékelve az egyes szerverek és kapcsolataik által létrejövő erőforrás korlátokat. Az elosztott rendszerű *cache* megoldás azoknak az alkalmazásoknak előnyös különösen, amelyek futása közben kritikus a folyamatos rendelkezésre állás, illetve az, hogy az adatok mindig több eszköz memóriájában redundánsan megjelenjenek, így küszöbölve ki a kulcsfontosságú információk elvesztésének lehetőségét (Stasoz, 2021; Sandu 2022; Chengjian et al., 2015).

A tárgyalt osztályozás szerint még lehetne *client side* nevű csoport is, bár ez igen nagy átfedést mutatna a már előzőleg kifejtett *web cache* – ahogy a neve is mutatja – kliens oldali változatával.

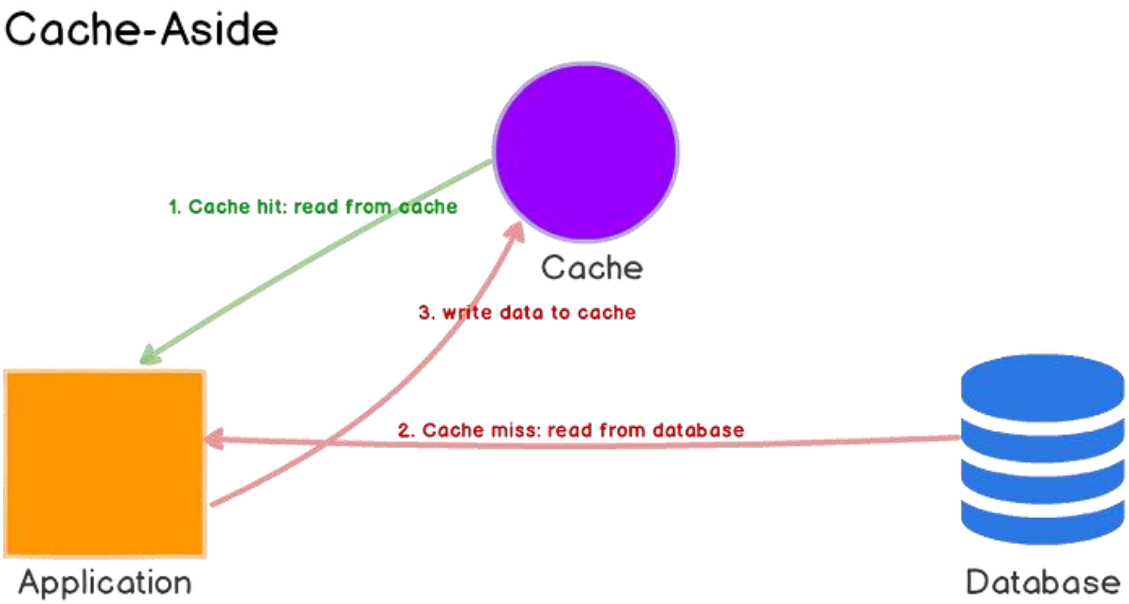
2.2 Gyorsítótár stratégiák

A gyorsítótárak témakörébe tartoznak még az úgynevezett *cache* stratégiák (*cache strategies*), melyek meghatározzák, hogy az információáramlás során az ideiglenes adattárolás topológiai szempontból hol helyezkedjen el. Ezeket a mintázatokat sokszor *caching pattern*-nek is nevezik. Az alapvető kérdés, amit ezek a megoldások meghatároznak, hogy mikor kerüljenek az adatok a gyorsítótárba, illetve, hogy az olvasás folyamata közben az adatbázis szervert milyen irányból éri el az alkalmazás vagy webszerver. A megfelelő stratégia nagyban befolyásolja a *cache* működésének hatékonyságát, valamint a *cache hit* arányt, valamint az adatállományok konzisztenciájának mértékét.

2.2.1 Cache-aside, lazy-loading

Az egyik ilyen gyorsítótár minta a *cache-aside* [*write-aside* és *read-aside* (*lazy-loading*)], melynek használata során az alkalmazás felelős a gyorsítótár irányításáért. Amikor az applikáció adatot igényel az adatbázis szervertől, akkor a kívánt állományok meglétét először a gyorsítótár memóriájában keresi, majd, ha az adatok itt megtalálhatóak, akkor az alkalmazás innen gyűjti be ezeket (*cache-hit*), ellenkező esetben, ha itt nem fellelhetőek, akkor fordul valójában az adatbázishoz (*cache-miss*). Ezt követően, amennyiben az adatokat kiolvasta az adatbázisról, akkor a *cache*

memóriájába is kiírja ezeket, annak érdekében, hogy ha egy meghatározott időn belül újból szükség lenne az adott állományokra, akkor rendelkezésre álljon itt, és ennek köszönhetően lényegesebben hamarabb megkaphatja az igényelt információkat az alkalmazás. Ennek az eljárásnak az előnyei, hogy csak azok az adatok kerülnek a *cache*-be, melyeket az alkalmazás valóban használt már, így kifejezetten „költséghatékony" a futása, valamint működése kiváltképp egyszerű. Használata során nagy figyelmet kell szentelni annak, hogy mindig az aktuális adatok legyenek rajta tárolva, hiszen, ha egy *cache*-elt adatot változtatunk, akkor könnyen megeshet, hogy az adat időközben valamilyen mértékben megváltozik, és ilyenkor kezelni kell azt a problémát, hogy a gyorsítótárról érkező adatok nem minden esetben a legfrissebbek. Egyéb problémája még az lehet ennek a rendszerű *cache*-nek, hogy csak *cache-miss* esetén kerülnek az átmeneti memóriába a kért állományok, illetve enyhén lassítja az adatigénylést az, hogy az alkalmazás – ha feleslegesen is – minden alkalommal „beolvas" a *cache*-be (Mansoor, 2017; Rahul, 2022).

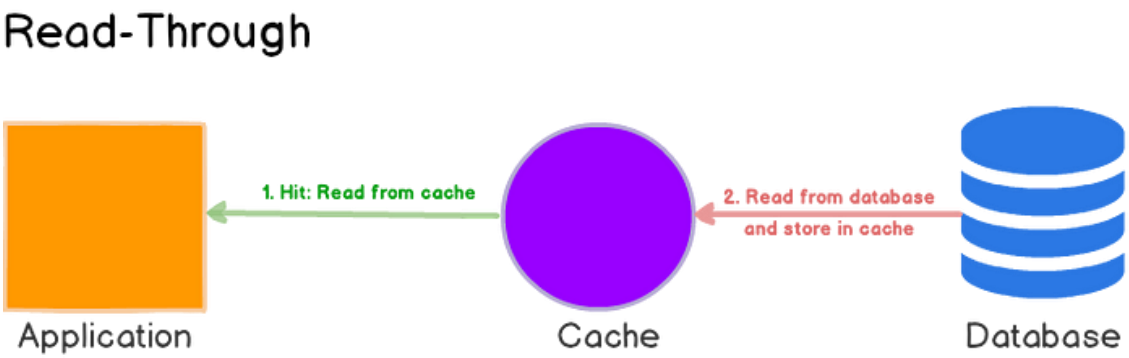


1. ábra Cache-aside stratégia

Forrás: https://akasht.medium.com/cache-layer-in-application-c9e15e9b173b

2.2.2 Read through

A *read-through* egy különböző *cache* stratégia, melynek alkalmazása során az alkalmazás számára maga a gyorsítótár a közvetlen adatforrás. A program ilyenkor nem kommunikál az adatbázis szerverrel, hanem kérés esetén a *cache*-t címzi meg. Abban az esetben, ha az igényelt információk nem találhatóak az átmeneti tárban, akkor a gyorsítótár az, ami elkéri az állományokat az adatbázistól, majd mentésre kerül a *cache*-ben és ezután küldi tovább az applikációnak. Ezen stratégia implementálása esetében az adatok lényegesen könnyebben tarthatók konzisztens állapotban, tehát mindig a megfelelő adatot kapja a felhasználó alkalmazás, azonban használata valamivel komplikáltabb. A megoldás abban az esetben hatékony, ha az információk olvasására sokszor kerül sor, viszont kevés alkalommal történik ezek frissítése (Mansoor, 2017; Rahul, 2022).



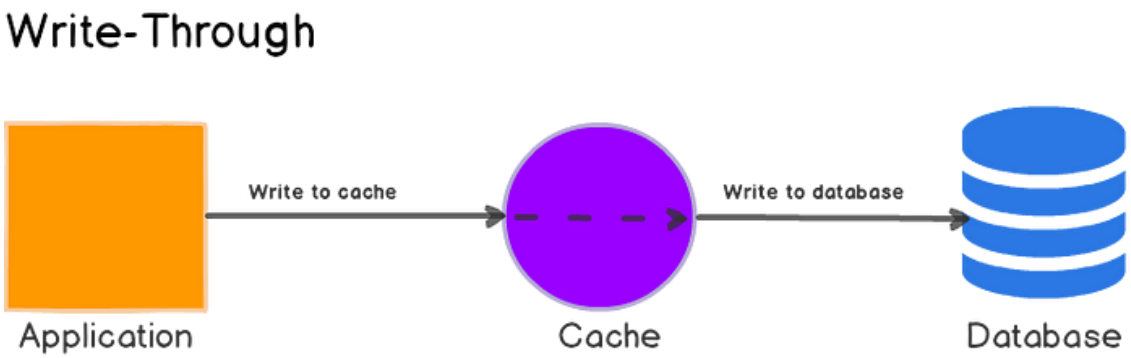
2. ábra Read-through stratégia

Forrás: https://akasht.medium.com/cache-layer-in-application-c9e15e9b173b

2.2.3 Write-through

Egy másik gyorsítótár stratégia a *write-through*, mely megoldás esetén a *read-through*-hoz hasonlóan a *cache* felelős az adatbázisban lévő információk mentéséért, valamint frissítéséért. Az állományok frissítése esetén mind a gyorsítótár memóriájába, mind az adatbázis szerver tárhelyére egyidejűleg kiírásra kerül. Ennek a mintának a használata során az írási folyamat valamivel lassabban megy végbe, viszont az adatok állapota minden esetben megfelel a valóságnak, a *cache* memóriájában lévő állományok tökéletesen megegyeznek az adatbázis tárhelyén lévővel. Ez a megoldás akkor a leghatékonyabb, amennyiben az alkalmazás számára nagyon fontos az adatok

konzisztenciája, emellett nem történik frekventáltan adatok írása az adatbázisba (gridgain.com, 2023; Alex DeBrie 2023; Mansoor, 2017; Rahul, 2022).

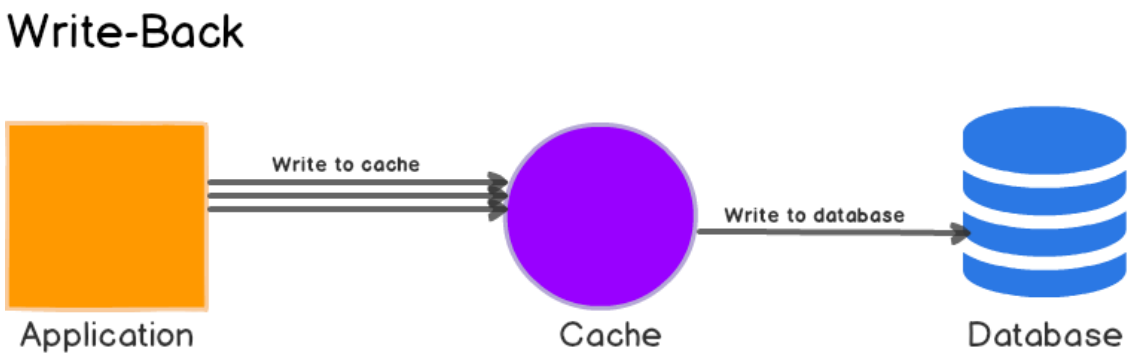


3. ábra Write-through stratégia

Forrás: https://akasht.medium.com/cache-layer-in-application-c9e15e9b173b

2.2.4 Write-behind

A *write-behind caching* egy olyan gyorsítótár stratégia, ami esetében az adatok írása először csak a gyorsítótár tárhelyére történik, majd egy későbbi időben, amikor gyorsítótár terhelése számottevően kisebb mértékű, akkor az adatbázis szerver háttértárjába is megtörténik az állományok mentése. A *cache* a működése során követi az adatváltozásokat, ennek köszönhetően tudja az állományok keletkezését követően eltérő időpontban elvégezni ezek háttértárra való kiírását. Ennek jelentős előnye, hogy lényegesen gyorsabban tudja lebonyolítani az alkalmazás az információk érkezését, jóllehet az adatok konzisztenciája jóval sérülékenyebb ebben az estben (Binieli 2023; Mansoor, 2017; Rahul, 2022)



4. ábra Write-behind/back stratégia

Forrás: https://medium.com/@SaiRahulAkarapu/caching-caching-caching-everywhere-8855ebe56f57

2.3 Gyorsítótár alternatívák egy vizsgáztató rendszer számára

A többrétegű alkalmazások témakörében – így a vizsgáztató rendszerek esetében is – a gyorsítótárak alapvető funkciója, hogy adatok ideiglenes eltárolásával a szoftver bizonyos komponenseit tehermentesítse, ezáltal növelve az egész szoftvercsomag kapacitását. Napjainkban a komplex webalkalmazások a bárki számára elérhető informatikai technológiáknak köszönhetően jellemzően szinte minden esetben több *cache* megoldást is egyszerre használnak. A vizsgáztató rendszerek teljesítményét legtöbbször az adatbázis kérések mennyiségének csökkentésével lehet javítani, melyhez számos, főként szoftveres gyorsítótár megoldás hozzájárulhat. Az első dolog, amit ki kell választani, hogy a szoftver adatainak kezelését elősegítő *cache* rendszer *In-memory* típusú vagy elosztott struktúrában megvalósított működési elvű legyen (Modi, 2021; Paneri, 2023; Yang, 2022).

2.3.1 In-memory változat

Ez az opció egy lényegesen egyszerűbb és olcsóbb alternatíva, melynek hátránya, hogy felhasználásának meglehetősen korlátoltak a mennyiségi mérőszámai, az igazán sok végfelhasználót kiszolgáló vizsgáztató rendszerek esetében egy szinten túl nem tud javítani a teljesítményén. Előnye azonban, hogy a konzisztens adattárolás megoldása nem túl bonyolult feladat, valamint, hogy ennek üzemeltetése egy költséghatékony megoldás, hiszen a legtöbb keretrendszer rendelkezik valamilyen beépített *In-memory cache* megoldással. Ezen kívül nincs szükség semmilyen hardveres (*cache* szerver) támogatásra. Egy, a témakörhöz kötődő speciális hátulütője, hogy ha párhuzamosan több alkalmazás szerveren akar egy intézmény egy kérdőívet kitöltetni a nagyobb teljesítmény érdekében, akkor, ha a kérdőívek a szerverek gyorsítótáraiba való bekerülése után változtatnak rajtuk, abban az esetben az összes szerveren külön-külön be kell futtatni a változtatást, különben az adott kérdőívek eltérhetnek egymástól, így az állományok konzisztens állapotának fenntartása meglehetősen komplikálttá válik. Ennek eredményeképpen az alkalmazás kapacitásának skálázhatósága nem megvalósítható az alkalmazás szerverek számának növelésével, amely a legköltséghatékonyabb eljárások közé tartozik (Stasoz, 2021; Sandu, 2022; Chengjian et al., 2015).

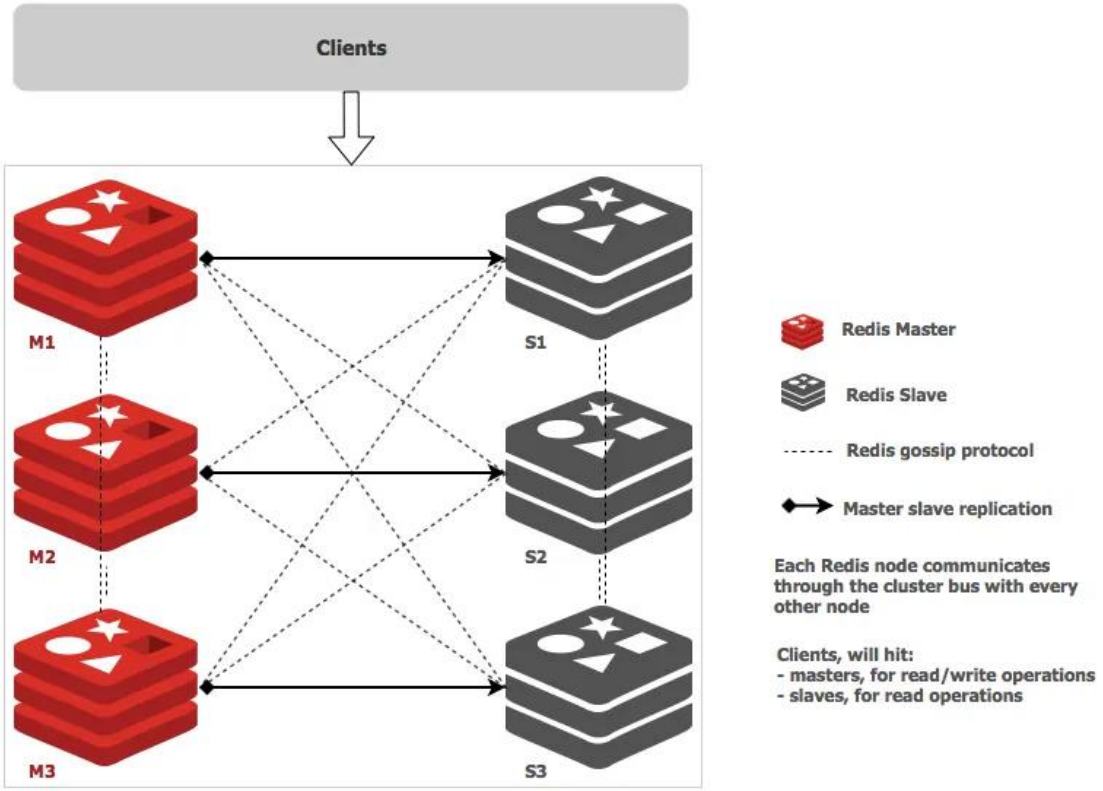
2.3.2 Elosztott rendszerszerű változat

Ha egy vizsgáztató rendszer tevékenysége során elosztott rendszerű gyorsítótárat implementál, abban az esetben teljes mértékben skálázható lesz a teljesítménye, illetve a működése által – ugyan nem a legegyszerűbb módon, de – garantálható az általa használatos adatok konzisztens állapota. Az elosztott *cache* megoldás esetén számos topológiai elrendezésnek is van létjogosultsága (Stasoz, 2021; Sandu, 2022; Chengjian et al., 2015).

A szakdolgozatomban három alapvető megvalósítást mutatok be. Az első kivitelben a gyorsítótár egy külön, erre a célra kijelölt szerveren fut, melynek előnye, hogy a kérdéses hardvert semmilyen más alkalmazás nem használja, így kapacitásának száz százalékát *cache* szoftver működésére tudja fordítani, továbbá a konzisztens állapot fenntartása lényegesen egyszerűbb, mint a többi elrendezés esetén. Hátránya, hogy egy külön eszközt kizárólag erre a funkcióra kell felhasználni, így a beszerzés, illetve az üzemeltetés plusz költséget jelent. Abban az esetben (második eset), ha a gyorsítótár több alkalmazás szerver közül az egyiken fut, akkor nyilvánvalóan ez egy költséghatékonyabb megoldást eredményez, a konzisztencia és a teljesítmény rovására. A harmadik megvalósítás során mindegyik alkalmazás szerver futtat egy önálló gyorsítótár példányt, melyek működésük során úgy tartják fenn az adatok konzisztens állapotát, hogy tranzakciók segítségével üzemelnek, így az adatbázissal való kommunikációjuk során a létrejövő tranzakciók gondoskodnak arról, hogy mindegyik *cache* példány memóriájában a legfrissebb információk szerepeljenek. Ha a tranzakció valamilyen hiba miatt nem tud végbe menni, akkor adott esetben semelyik gyorsítótár entitásban lévő adatok nem változnak meg.

A különböző alternatívák bármelyik *caching* stratégiával képes megfelelően működni, a támogatni kívánt alkalmazás architektúrájától függ, hogy melyik opciót érdemes felhasználni. Egy vizsgáztató rendszer esetén véleményen szerint leginkább a *cache-aside* megoldást érdemes használni, hiszen például egy vizsgáztatás során jellemzően nagy számú végfelhasználó egyszerre viszonylag kevés adatot használ. Az adatok konzisztens állapotának állandóságát ebben az esetben garantálja az elosztott rendszer működtetéséért felelős szoftver, így a *cache-aside* hátrányát ez kiküszöböli.

Az elosztott rendszerű gyorsítótár megvalósítások esetében úgynevezett csomópontokat (*node*-okat) is használunk, melyekből alapesetben kétfélét különböztetünk meg: *master* és *slave*. Ezen architektúra fennállása során a *slave node*-ok a *master node*-ok másolatai, melyek akkor válnak hasznossá, amikor a *master node*-ok elérése valamilyen oknál fogva meghiúsul. Például a szerver (csomópont) üzemelés szünetel annak meghibásodása vagy esetleg áramszünet miatt, akár hálózati meghibásodásból kifolyólag. Bizonyos kialakítású rendszerek esetén a *master node* felelős az adatok írásáért és módosításáért, míg a *slave node*-ok az állományok olvasását teszik lehetővé (Williams 2019).



5. ábra Redis csomópont architektúra

Forrás: https://medium.com/@123williams93/distributed-caching-in-redis-8ff882bf79ac

3. KÉSZÍTSEN EMPIRIKUS KUTATÁST AZ EGYEDI PROBLÉMÁRA A KIGYŰJTÖTT ADATOK FELHASZNÁLÁSÁVAL!

Az alábbi fejezetben bemutatok egy konkrét vizsgáztató rendszert, az *Unipoll*-t, valamint ennek jelenlegi gyorsítótár implementációját. Ezen felül ismertetem az általam készített teszt szoftvert, és ennek segítségével produkált eredményeket. A fejezetben továbbá megtervezem a vizsgáztató rendszer kapacitásának növelése érdekében elkészíthető ideálisabb *cache* megvalósítást, valamint részletezem az optimális megoldás implementálásának problémáit.

3.1 Az *Unipoll* vizsgáztató rendszer

Az *Unipoll* egy online vizsgáztató rendszer, melyet minden magyarországi felsőoktatási intézmény, valamint számos „független", privát intézmény használ valamilyen formában. Az alkalmazás segítségével kérdőíveket, valamint vizsgákat lehet szerkeszteni, készíteni számos kérdéstípus felhasználásával az egyszerű, „bepipálós" kérdésektől kezdve a táblázatos, topográfiai, vagy fájlfeltöltéses kérdésekig. Az alkalmazás fejlesztése 2009-ben kezdődött, mely folyamat indító motivációja egy „diplomás pályakövetési" rendszer kialakítása volt. Az *Unipoll* vizsgáztató rendszer egy többnyelvű webalkalmazás, mely a begyűjtött adatokból statisztikát, valamint riportokat is képes készíteni. Az ily módon kinyert információkat különféle táblázatos formátumban is elő tudja állítani, melyek használata megkönnyíti az adatok feldolgozását (www.sdainformatika.hu/termekek). A vizsgáztató rendszer egy rétegzett alkalmazás, mely rendelkezik megjelenítési réteggel (*presentation layer*), üzleti logikai réteggel (*logic/domain tier/layer*) és adat réteggel (*data tier/layer*). A szoftver három különböző módban üzemel (Fowler – Wesley, 2002).

Az egyik ilyen módja a vizsgáztató alkalmazásnak a magán vállalatok saját belső környezetein üzemelő, úgynevezett „privát" változat, mely kisebb vállalkozásoktól egészen nagy cégekig, különböző felhasználási igények mellett, változó számú végfelhasználó által történő alkalmazással üzemel. A szolgáltatást alkalmazó vállalatok jellemzően belső működésük támogatására használják a szoftvert. Az egyidejűleg történő kérdőív kitöltések számáról ez esetben nem rendelkezik a készítő csapat, de legnagyobb valószínűséggel egy-két száznál nem lehet több ez az érték.

A második változata az alkalmazásnak az *unipoll.hu*-n elérhető, bárki által igénybe vehető kérdőívszerkesztő webapplikáció. A szolgáltatás használatához különböző időintervallumokra lehet jogosultságot vásárolni. Általában magán személyek, vagy a szoftver használatában rövid távon érdekelt szervezetek veszik igénybe ezt a módozatot. Jellemzően ezen a „lábon" keletkezik a legkevesebb felhasználó általi adatforgalom az adatbázis szerver és a webszerver között. A fejlesztő csoport ennek a változatnak a működését tudja a legjobban monitorozni, hiszen ebben az esetben az alkalmazás üzemeléséhez szükséges informatikai rendszert a készítő cég tartja fennt.

A harmadik verziója a szoftvernek a magyarországi felsőoktatási intézmények által kizárólagosan használt *Neptun* névre hallgató „Egységes Tanulmányi Rendszerrel" összekapcsoltan működik. Ebben az úgynevezett „integrált" módban van kitéve a legnagyobb terhelésnek az alkalmazás futásához használatos informatikai infrastruktúra, mely alatt az alkalmazás és adatbázis szervereken kívül még a lokális, illetve internet hálózatot értjük. Ezen változat működése során bizonyos esetekben a szimultán kitöltések száma elérheti akár az ezret is. A felsőoktatási intézmények informatikai rendszerüket saját preferenciáik alapján alakítják ki, illetve a *Neptun*-t, valamint az *Unipoll*-t fejlesztő cég megfogalmaz egy ajánlást a két alkalmazás számára szükséges informatikai környezettel kapcsolatban, amelyet érdemes az egyetemeknek figyelembe venniük.

A vizsgáztató rendszer használatának gyakorisága a Koronavírus világjárvány betörésével számottevően megnövekedett (Lannert, 2022), ebből kifolyólag az alkalmazás üzemeltetésének szűk keresztmetszetei is kirajzolódtak. Az egyetemek ekkor számos esetben a hallgatók vizsgáztatásának lebonyolítását teljes egészében az *online* térbe, vagyis a vizsgáztató rendszerekre helyezték át. A hirtelen fellépő igény kielégítése, a nagy mennyiségű hallgató által abszolválandó vizsga lebonyolítása ekkor komoly logisztikai problémának bizonyult, hiszen csak a hallgatók egymástól eltérő időben történő számonkérésével tudták megoldani szituációt. A járványhelyzet lezárultával az egyetemi vizsgáztatás módja közel sem került teljes mértékben vissza a korábbi, „normális" kerékvágásba, ugyanis a felsőoktatási intézmények nagy számban továbbra is az *Unipoll* rendszerén keresztül bonyolítják számonkérési folyamataik egy jelentős részét. A fentiekből könnyen belátható, hogy a vizsgáztató alkalmazás folyamatos kapacitás növelésére a továbbiakban is nagy hangsúlyt kell fordítani. A

szoftver kapacitásának bővítését számottevően elősegítheti a meglévő gyorsítótár megvalósításának fejlesztése.

3.2 A vizsgáztató rendszer jelenlegi gyorsítótár megoldása

Az *Unipoll online* vizsgáztató rendszer működése során egy *In-memory* típusú gyorsítótár megvalósítást alkalmaz, mely a *.NET* keretrendszerbe integrált *System.Runtime.Caching.MemoryCache* névtér alatt érhető el. Ez a *cache* implementáció a *Framework 4.0*-ás verziójával került be a keretrendszerbe. Az *In-memory cache* megoldás az adott alkalmazás vagy webszerver saját *RAM*-jában tárolja el ideiglenesen az adatokat annak érdekében, hogy gyorsabban ki tudja szolgálni a felhasználói kéréseket. Ez a gyorsítótár megvalósítás nem egy elosztott rendszerű változat, melyből következik, hogy egy szerveren tud biztonságosan, az adatok konzisztens állapotát megtartva üzemelni. Ha több alkalmazás szervert párhuzamosan használ egy intézmény, és az adatbázisban úgynevezett natív szkript segítségével változtatnak például egy vizsgában valamit, akkor a szervereken addig nem jelenik meg a változtatás, amíg a saját gyorsítótáraikból törlésre nem kerülnek az eredeti állományok. azt, hogy mikor történik az adatok törlése, azt egy előre meghatározott törlési „politika” adja meg. Jelen szoftver esetén *sliding expiration* módszerrel megy végbe a törlés, ami azt jelenti, hogy, ha egy meghatározott időn belül újra elkérik a kérdéses adatállományt, akkor a törlésig visszamaradó idő visszaáll egy kezdőértékre. Amennyiben a meghatározott időablakon belül nem érkezik kérés az adott információra, akkor eltávolításra kerül a gyorsítótárból.

A *Unipoll* alkalmazás esetében párhuzamosan több különböző alkalmazás szerver használatának igénye sok esetben felmerült már a vizsgáztatás során a szimultán történő kitöltések számának növelése érdekében. Több adatbázis szerver használata indokolatlanul nagy többletköltséget jelentene egy felsőoktatási intézmény számára, hiszen az adatbázis szoftvereket szolgáltató cégek futtatott példányok számától teszik függővé az ellenszolgáltatás értékét. Az tehát egyértelművé vált, hogy a skálázhatóság nem igazán kivitelezhető *In-memory cache* használata mellett. Felmerülhet az az alternatíva is, hogy a vizsgáztató rendszerek alkalmazása esetén egyáltalán ne kerüljön gyorsítótár megoldás használatra. Így több alkalmazás szerver párhuzamosan való használata is biztonságos működést biztosíthat a felhasználni kívánt adatok naprakész

állapotának szempontjából, hiszen nem kell a különböző szerverek *cache* megoldásainak memóriájában lévő, esetenként elavuló adatok problémáját megoldani. Ezáltal a rendszer kapacitásának skálázhatósága megoldható lenne, mert ezen a módon bármennyi alkalmazás szervert lehet a hálózatba bekapcsolni. Azonban az *In-memory* gyorsítótár szolgáltatás nélkül az adatbázis szerverről érkező konkrét adatcsomagok több, azonos időben történő kérés esetén számottevően lassabban érkeznek.

3.3 Tesztprogram elemzése a gyorsítótár hatékonyságának igazolására

Az előző felvetés alátámasztásának érdekében készítettem egy izolált módon működő *lightweight* (kisméretű) szoftvert, mely az *Unipoll* alkalmazás egyik adatbázisának másolatát felhasználva adatokat kér le egy *natív SQL szkript* által. A teszt programban három különböző *api* függvényt futtattam, amelyek egy darab azonos lekérdezést tartalmaznak, ami – a szemléltetés érdekében – az adatbázis számára egy számításigényes, több tábla összekapcsolása révén létrejövő halmazt készít válasz gyanánt.

A vizsgálat során az adatbázis szerver felé indított kérésekre érkezett válaszadás azon paraméterit elemzem, amelyek a lehető legjobban alkalmasak az eltérő implementációk közötti teljesítmény különbségek reprezentálására. Ezek többek között az átlagos válaszidő, a maximális válaszidő, valamint az egy másodperc alatt beérkezett adatok mértéke *kilobyte*-ban. Az általam vizsgált három tényező értéke erőteljesen összefügg, ám ezek közül talán a legfontosabb, ami legobjektívebben érzékelteti az adatbázissal való kommunikációt, az az átlagos válaszidő. A válaszidő – ebben az esetben – lényegében az az időintervallum, ami függvényhívástól/kérés elküldésététől az adatbázis szerver által összeállított válaszérték, vagy adatállomány kérőhöz való visszaérkezéséig terjed. A válaszidő nagyban befolyásolja egy rétegzett alkalmazás működésének hatékonyságát, gyorsaságának érzetét, hiszen többé-kevésbé ez az idő az, amit egy végfelhasználó a számítógép monitorja előtt várakozással eltölt, amíg egy alkalmazás visszatér a számára összeállított tartalommal. A kísérlet során eltérő számú, egyidőben indított *szkript* futtatását hajtottam végre. Az *api* függvények hívása aránylag alaposan megfeleltethető egy kérdőív kitöltésének folyamatával az adatbázis szerver terhelése szempontjából. Nyilvánvalóan egy vizsga lebonyolításakor lényegesen több és bonyolultabb lekérdezés futhat, melyeket az adattárház eltérő „részeiről” kell felolvasni.

Ennek ellenére az elkészült *pilot* program véleményem szerint jól reprezentálja a kérdőívek letárolt állományainak adatbázisból való kiolvasását.

A teszt alkalmazás esetében két különféle gyorsítótár implementációt, valamint egy „kontroll” változatnak nevezhető egyszerű adatbázis lekérdezést működtető *api* függvény hívása történik. Mind a három megoldás futtatása során az első lépés egy úgynevezett „*SQL* kontextus” létrehozása, amely lényegében megteremti a kapcsolatot az alkalmazás és az adatbázis között. A kapcsolat létrejötte után lehet az adatbázis szerver felé lekérdezéseket indítani. A *cache* megvalósítással nem rendelkező változat lényegében ennyiből áll. A további két változat közül a könnyebben elkészíthető, és működtethető a *.NET. Framework* beépített *MemoryCache* implementációja. Ezen megvalósítás a függvény hívás alkalmával ellenőrzi, hogy az ideiglenes memóriában szerepel-e már a kérésre vonatkozó adathalmaz. Abban az esetben, ha a megfelelő „kulccsal” rendelkező állomány megtalálható itt, akkor a gyorsítótár memóriájából másolja ki az adatokat. Ha ezek nem találhatóak itt, akkor a megnyitott adatbázis kapcsolaton keresztül szerzi be őket a megfelelő adattáblákból, és mielőtt visszaküldené a kérőnek, bemásolja a gyorsítótár memóriájába, – ami jelen esetben az alkalmazást futtató eszköz gyors elérésű *RAM*-ja – az előre megadott *expiration policy* rögzítésével, mely meghatározza, hogy mennyi ideig tárolja itt az adatokat.

A másik *cache* megvalósítás, egy *third party*, azaz másik fél által készített gyorsítótár szoftver, mely a *couchbase* névre hallgat. Ez a megoldás már alkalmas osztott *cache* működtetésére is. Futása során egy gyorsítótár szerver igénybevételével üzemel, mely úgynevezett *cluster* és *bucket* virtuális adattároló egységekre osztja fel tárhelyét. Ez a megvalósítás aszinkron módon működik, ami azt jelenti, hogy az alkalmazás eltérő szálakon, párhuzamosan képes több, különböző programkódot futtatni. Ennek előnye, hogy több, egymástól független kérést tud kezelni a szoftver, amennyiben ez a lehetőség jól van kezelve az implementáció során. A *couchebase* használata rengeteg új, a nagy webalkalmazások futásához elengedhetetlen funkciókkal rendelkezik a keretrendszer beépített gyorsítótár megoldásához képest, viszont ennek meg van az a hátulütője, hogy használatához lényegesen nagyobb hardver erőforrással rendelkező szervergépek szükségesek, valamint számottevően bonyolultabb és specifikusabb szoftverkód készítését igényli. A szoftver használata korlátozott tárhellyel, *developer*, azaz fejlesztői verzióban ingyenes, azonban, ha egy webalkalmazás működéséhez megfelelő

kapacitással rendelkező változatot szeretnék igénybe venni, ahhoz meg kell vásárolni a különféle szükséges licenszeket. Működése során a megoldás egy külön szervert futtat, melyhez egy *conneciton string*[1] révén csatlakoznia kell az alkalmazás szervereknek, ilyen módon ezek elérik a szerveren lévő átmeneti adatbázist, amin a *cache*-elt adatok kerülnek eltárolásra. Ennek a megvalósításnak az az előnye, hogy az összes, az alkalmazást futtató eszköz hozzáfér a gyorsítótár tárhelyéhez, ily módon elosztott rendszerű *cache* megoldást eredményezve.

Az általam készített szoftver működése során lekérdezések indításakor az első néhány az adatbázis által küldött válasz – mind a *cache* megoldással, mind annak használata nélkül – nagyjából azonos idő alatt érkezett meg, sőt az is előfordult, hogy gyorsítótár igénybevételével üzemelő változat esetében a teszt futásának első másodperceiben még lassabban is értek vissza az állományok az adatbázisról. Ez azért van, mert az általam készített alkalmazásban – és általában a *cache* implementációkban is – az első, az adatbázis szerver révén visszaküldött eredmény a kérőhöz való visszajuttatása során a gyorsítótár memóriájába is kiírásra kerül. Amíg ez a folyamat végbe nem megy, addig mind a két verzió működése során közvetlenül az adatbázis szerver küldi vissza az igényelt információkat, illetve némi számítási kapacitást igényel az ideiglenes tárhelyre való írás, emellett az adatok ezen a helyen való felesleges keresése is. Miután megtörtént a *cache* memóriájának feltöltése a kérdéses adatokkal, az alkalmazás először innen próbálja „kinyerni” a szükséges állományokat. A tesztprogramban lévő függvények futását egy *JMeter* névre hallgató szoftver segítségével vizsgáltam meg.

A *JMeter* tesztelő alkalmazás különböző szolgáltatások (*service)* performanciájának mérésére, és analizálására alkalmas. Felhasználása főként a webalkalmazások témakörben népszerű, ugyanis a program lehetővé teszi a felhasználói interakciókhoz hasonló kérések indítását egy webalkalmazás felé. Az alkalmazás felé közölt kérések válaszairól részletes statisztikai jelentéseket készít, melyek ezután könnyen vizualizálhatóak. A tesztelés során beállítható, hogy mekkora számosságú szimultán függvényhívást intézzen a tesztkörnyezet egyszerre (Hamilton, 2023).

[1] Hálózati kapcsolatok kialakításához szükséges szöveges elérési cím, mely egy bizonyos eszközre utal

A program esetében, melyet készítettem, tíz, száz, kétszázötven, valamint ötszáz párhuzamos függvényhívás segítségével állítottam elő teszteredményeket. Fontos befolyásoló tényező, hogy maga az adatbázis szerver is használ gyorsítótár megoldást. Ez abból látszik, hogy ha egymás után többször elindítok például kétszázötven párhuzamos *api* függvényhívást, – amely csak ez egyszerű, *cache* implementációt mellőző *SQL* szkript hívása – az első próbálkozásra nagyságrendekkel magasabb az egyes kérések válaszideje. Ez a hatás nyilvánvalóan a *.Net Framework* keretrendszerbe beépített gyorsítótár implementáció, illetve a *Couchbase cache* megoldásának használata esetében rövid távon nem befolyásolja ezek működését. A tesztelés során annak érdekében, hogy ezt a hatást ki tudjam iktatni, és ezáltal lényegesen objektívabb eredményeket tudjak rögzíteni, a különböző variációk használata között, meg kellett volna várnom, amíg az adatbázis szerver gyorsítótárának memóriájából törlésre kerülnek azt általam legkérdezett adatok. Sajnálatos módon az említett *cache* megvalósítás törlési módjának módszerét, illetve az ehhez paraméterként megadott időintervallum pontos értékét teljes magabiztossággal nem tudtam megállapítani. A fentiekben taglaltak miatt az első függvényhívást által produkált teszteredményeket kihagytam az általam készített statisztikából, a teszt lefolytatásának megkönnyítésére, illetve torzító hatásuk kizárása érdekében.

# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
250	1958	0	2303	226.40	0.00%	75.7/sec	1037.39

6. ábra Első futatás cache nélkül

Forrás: Saját készítés

# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
250	818	0	1475	465.69	0.00%	103.8/sec	1422.38

7. ábra Második futtatás cache nélkül.

Forrás: Saját készítés

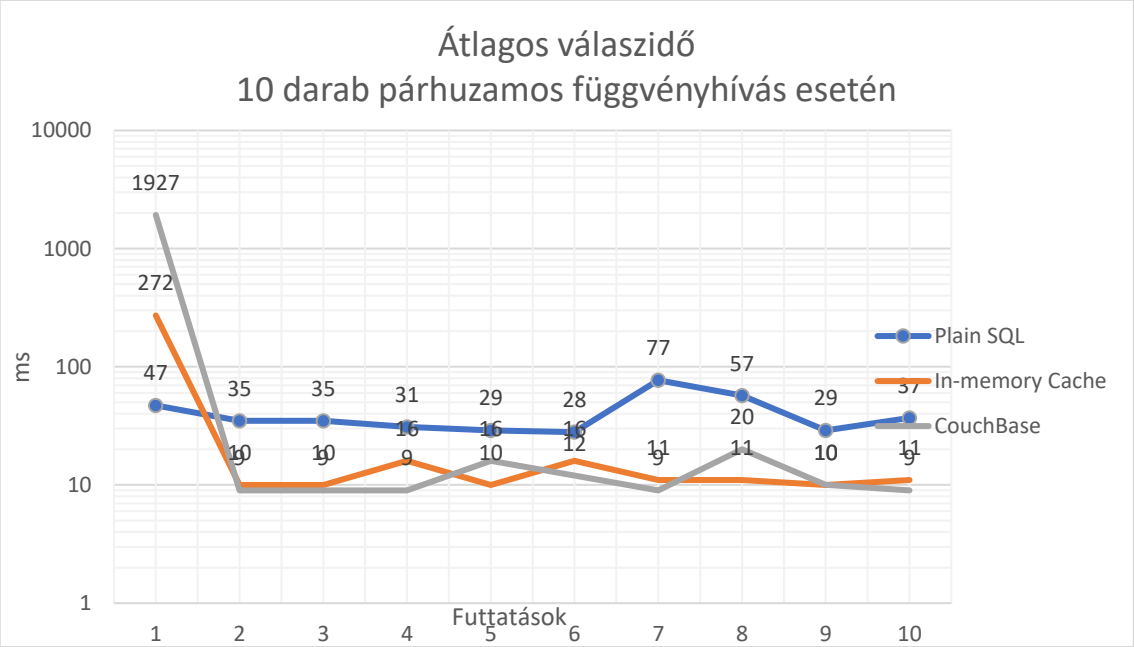
A vizsgálat során használt személyi számítógép kapacitása nem feltétlenül elengedő nagyobb számú párhuzamosan indított függvényhívás, valamint maga a futtató környezet egy időben történő működtetésére. Ezért az *api* függvényeket futtató

alkalmazást „kihelyeztem” egy másik jóval nagyobb teljesítményű, performancia tesztek futtatására tervezett és összeállított hardverre. Ennek a módszernek hála a tesztkörnyezetet futtató számítógép kapacitása bőségesen elegendő volt a kétszázötven párhuzamos függvényhíváshoz. Abban a tudatban kezdtem el tesztelni a három különböző implementációt, hogy így fogom a legobjektívebb és leginkább felhasználható adatokat kinyerni a tesztprogramból. A valóságban azonban a vizsgáztató rendszerek realisztikus, mindennapi működését érintő további vizsgálat során kiderült, hogy az általam fölállított kísérlet sokkal inkább idealisztikus, nem a valóságos működésnek megfelelő. A felsőoktatási intézmények egyrészt a limitált költségvetésük miatt ritkán rendelkeznek ilyen kapacitású hardvereszközökkel, amelyeket a tesztelés alkalmával igénybe vettem, másrészt az említett intézmények szervergépjein számos másik, akár jóval nagyobb erőforrás igényű szoftver fut egyidejűleg, illetve az adatbázis szervereik egyszerre több alkalmazás adatkérelmeit szolgálják ki. A kísérletet ebből a megfontolásából inkább a munkahelyi személyi számítógépemen folytattam, mely elgondolás ellenére olyan nagyságrendű szimultán függvényhívásokkal teszteltem az általam készített programot, hogy lehetőleg a személyi számítógépem erőforrásai ne befolyásolják túl nagy mértékben, ne jelentsenek szűk keresztmetszetet a kísérlet során.

Az első próbálkozások során nem hozta a teszt program az elvárt eredményeket. Az egyszerű adatbázis hívás közel teljesen azonos eredményeket produkált, mint az *In-memory* gyorsítótár megoldással támogatott függvényhívás, valamint a *Couchbase cache* sokkal lassabb volt az előbb említett mindkét megoldásnál. Ezután gyorsítótár implementációk esetében hosszadalmas kutatómunka és számos próbálkozás után megvalósítottam a *Singleton pattern*[2]-t, mely hatására a *Couchbase* segítségével létrehozott gyorsítótár lényegesen gyorsabb lett, de még mindig nem produkált semelyik *cache* megoldás rövidebb válaszidőket, mint az egyszerű adatbázis kommunikáción alapuló elképzelés. Ekkor arra a feltételezésre jutottam, hogy valószínűleg az adatbázis szervert egyáltalán nem terheli meg az adatok lekérése. Először egy kollégám segítségével megpróbáltuk „másik oldalról” hosszabb időintervallumot lefedő lekérdezéseket indítani a kérdéses adatbázis felé, hátha emellett a terhelés mellett már

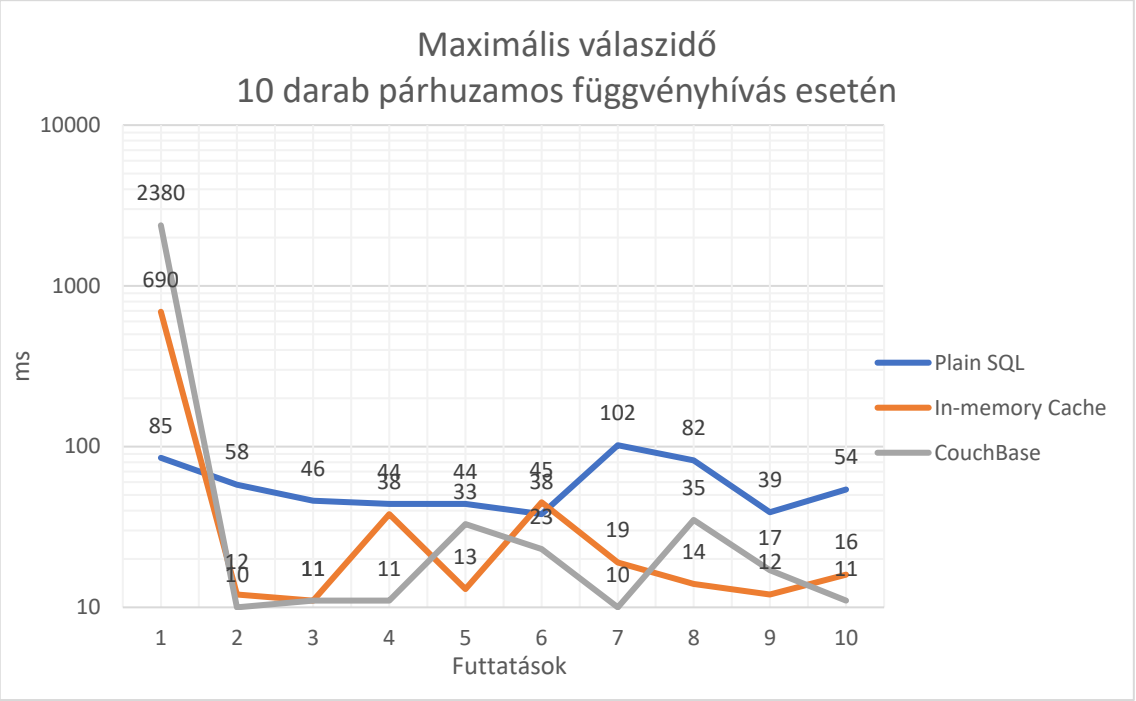
[2] Egy programozási megoldás, mely kieszközöli, hogy egy bizonyos osztályból, csak egy példány létezhessen egyidőben.

kimutathatók lesznek az általam készített gyorsítótár implementációk, és az ezek gyorsítása nélkül használt függvényhívások közötti válaszidők közötti különbségek. Sajnos, a kísérlet nem hozta a várt eredményeket: nem lehetett szignifikáns eltérést tapasztalni a három megvalósítás között. Ezek után a problémát kiküszöbölendő az *api* függvények által indított adatbázis lekérdezést alakítottam át úgy, hogy jóval lassabban összeállítható adatállományokból, több tábla összekapcsolásával, összetettebb számítások révén képződjön az adatbázis szerver válasza. Ezúttal sikerült megoldást találni arra, hogy a gyorsítótár implementációk kifejtsék hatásukat: lényegesen lecsökkentették a kérések válaszidejét.



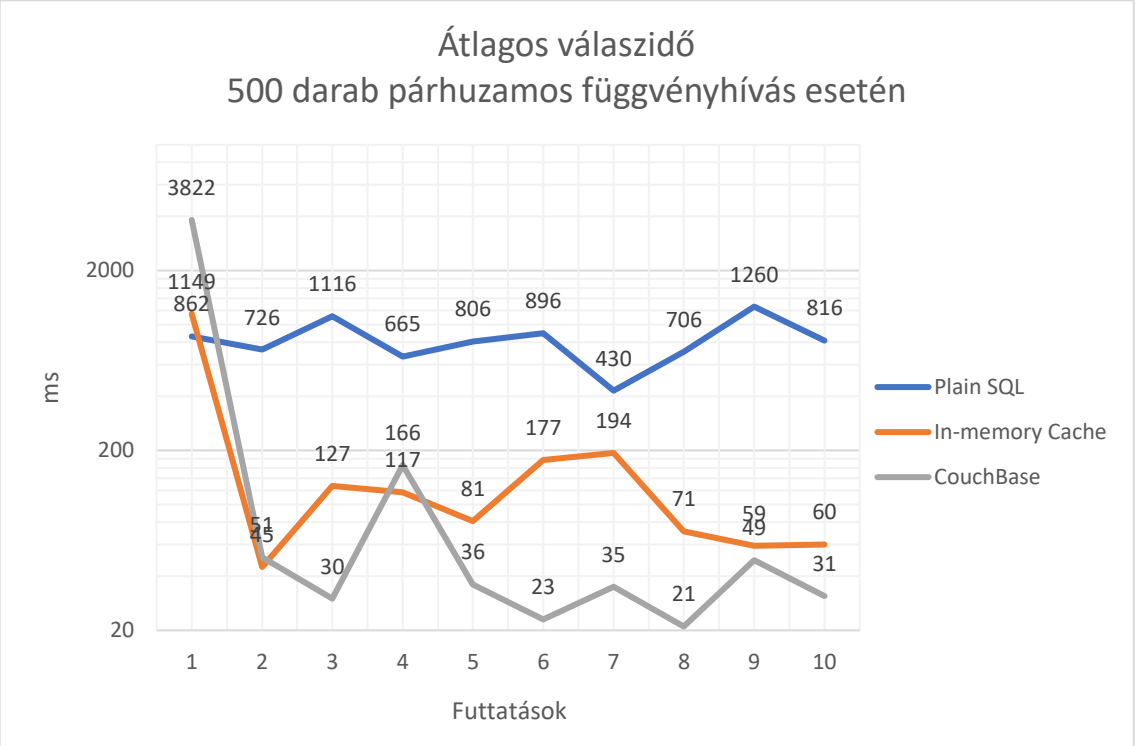
8. ábra Átlagos válaszidők alakulása 10 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés



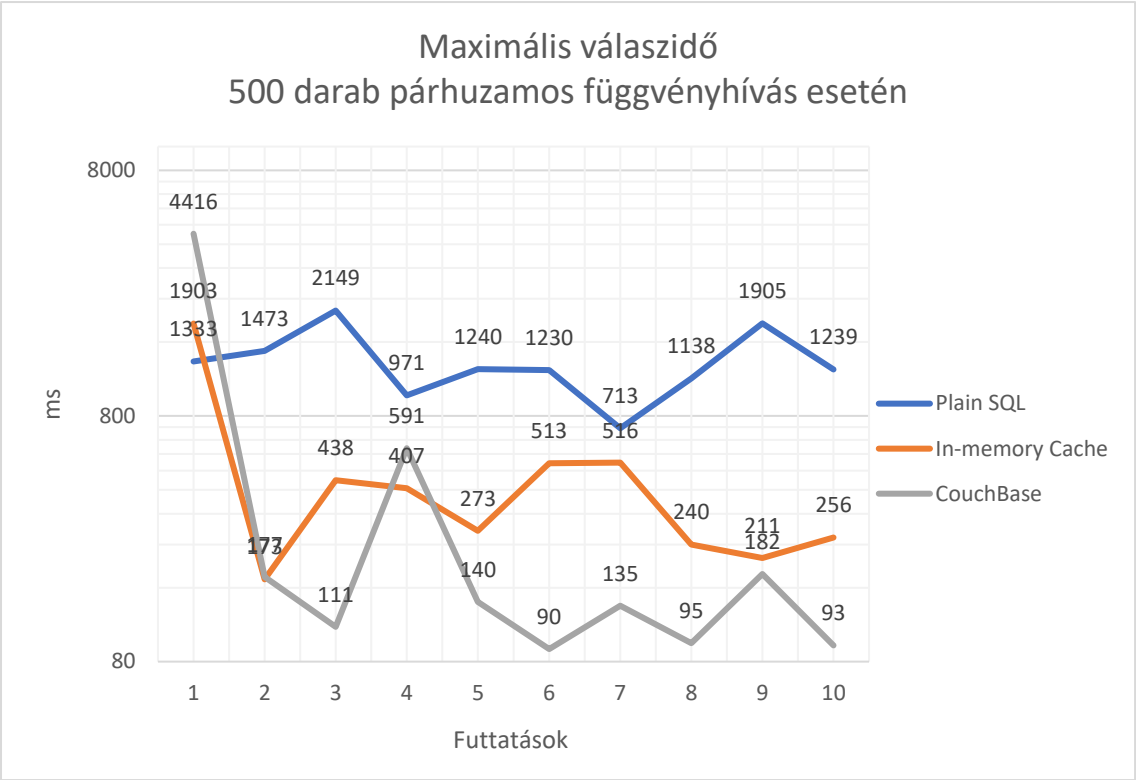
9. ábra Maximális válaszidők alakulása 10 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés



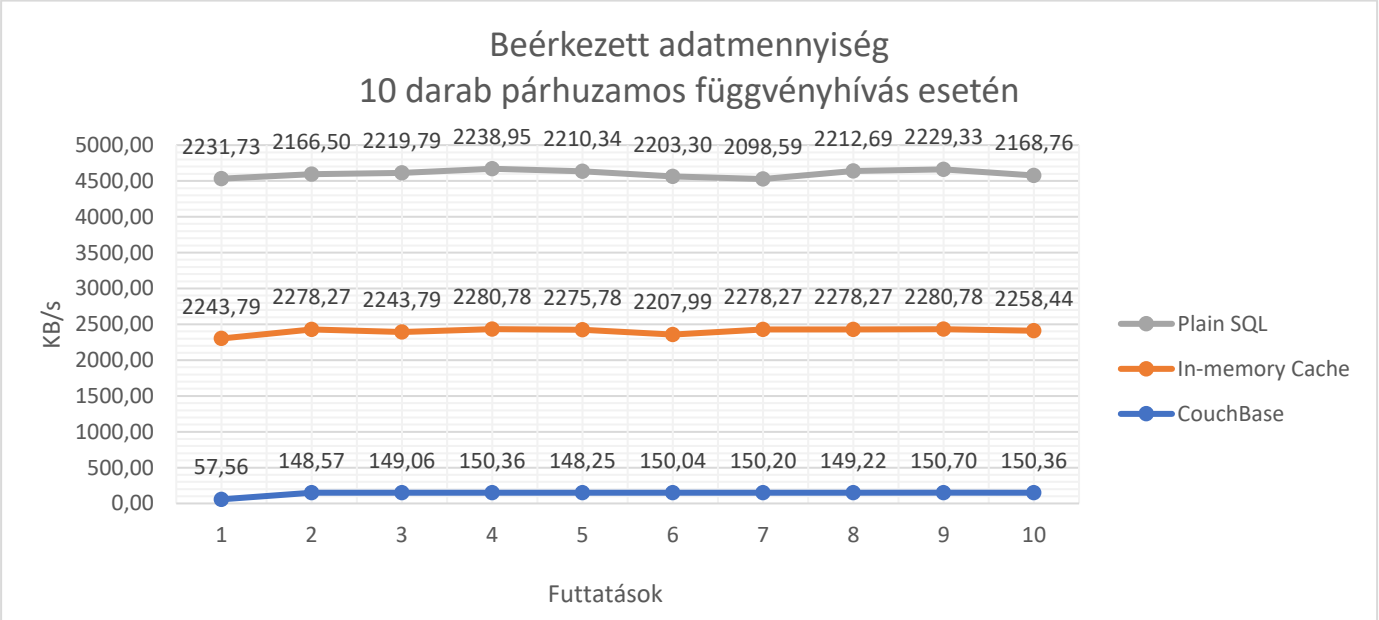
10. ábra Átlagos válaszidő 500 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés



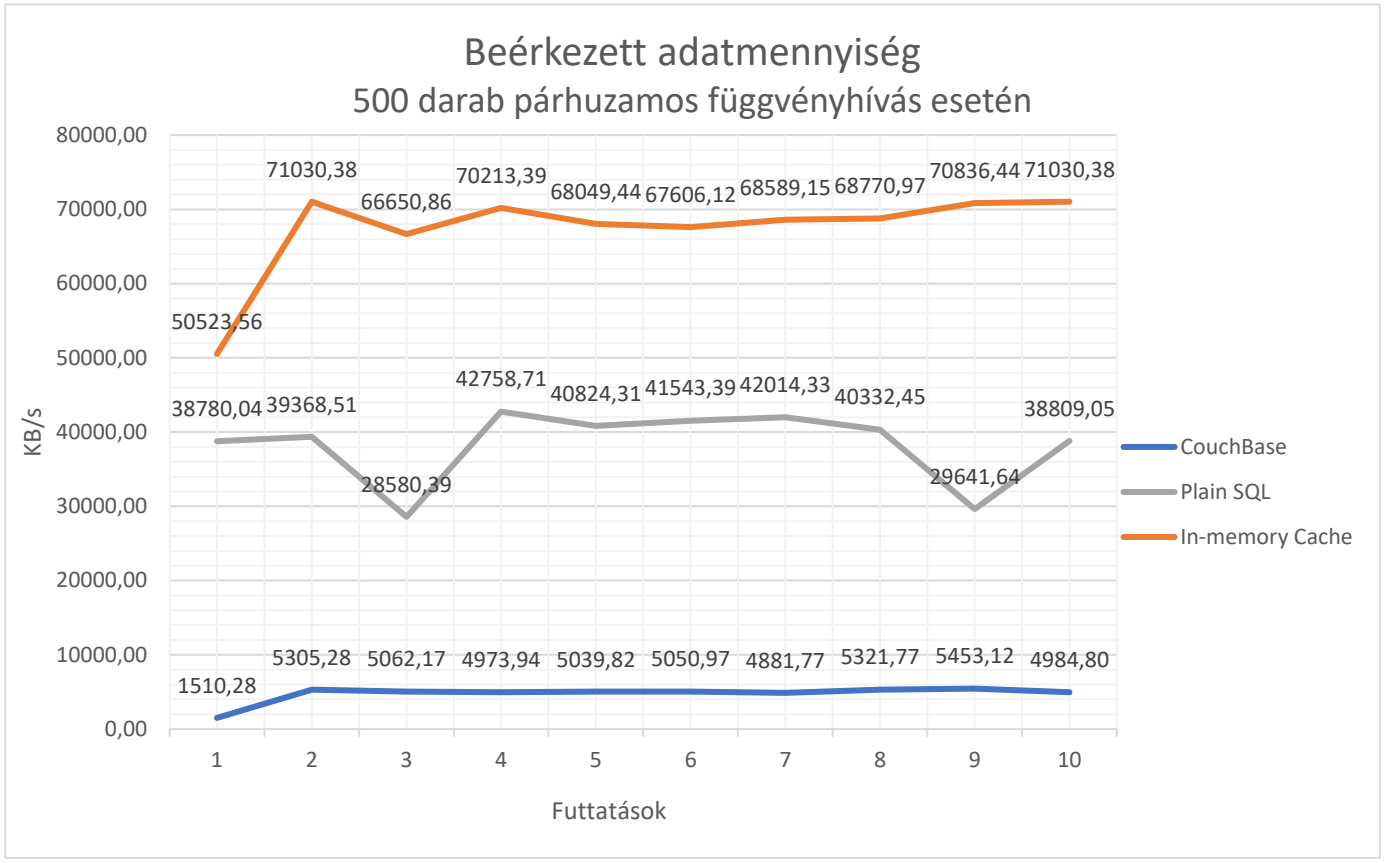
10. ábra Maximális válaszidők alakulása 500 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés



11. ábra Beérkezett adatmennyiség 10 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés



12. ábra Beérkezett adatmennyiség 500 darab párhuzamos függvényhívás esetén

Forrás: Saját készítés

A kinyert adatok alapján megállapítható, hogy a mért értékek számottevően alacsonyabb mértékűek a gyorsítótár implementációk használatával, mint anélkül, tehát a kérdéses megvalósítások az adatbázis kommunikáció során lényeges hatékonyság, illetve kapacitás növekedéshez vezetnek. Természetesen bizonyos hátrányai ezeknek az eljárásoknak is vannak. A leginkább szembetűnő probléma a *cache* használatával a kísérletben alkalmazott program alapján, hogy az első függvényhívások során, amíg az átmeneti tárhelyre nem kerülnek kiírásra a kérdéses állományok, addig az adatbázis szerver felé indított kérelmek válaszideje jóval hosszabb. Természetesen több függvényhívás esetén, melyek a valóságban általában nem pontosan egy időben történnek, a rendszer kezdeti hosszabb válasz idejét ellensúlyozza a folyamat során történő kapacitásnövekedés. Megfigyelhető továbbá, hogy az első, az adatbázis felé kommunikált kérések alkalmával a *Couchbase* szoftvert használó megoldás használata eredményezi a leghosszabb válaszidőket, hiszen ennek a megvalósításnak a működése során egy külön szerverre – ha csak virtuális szerverként tekintünk is rá – történik az adatok kiírása, illetve sajátos adatstruktúrában történik az eltárolás, ennek ellenére, ahogy az *In-memory* változat esetében itt is „megtérül” az eljárás korai szakaszában

történő lassabb működés a későbbi gyorsulás által. A tesztelés eredményeiből jól látszik, hogy végsősoron a tárgyalt eljárás a leghatékonyabb a három különböző megoldás közül.

4. ÉRTELMEZZE AZ ELÉRT EREDMÉNYEKET, ADJON MEG JAVASLATOKAT AZOK ALAPJÁN!

4.1 Az Unipoll vizsgáztató szoftver számára megfelelő gyorsítótár megvalósításának problémái

Az általam megvalósított és tesztelt kisméretű alkalmazás fejlesztése során számos problémába ütköztem. Az első ilyen nehézség annak az eldöntése volt, hogy saját fejlesztésű gyorsítótár megoldást fejlesszek-e, vagy az interneten elérhető, már meglévő, alaposan tesztelt, dokumentációval ellátott eszközök használatával készítsek tesztprogramot a *cache* technológia létjogosultságának igazolására. Számos forrásból igyekeztem tájékozódni, hogy erre a kérdésre megalapozott választ tatáljak. Arra, hogy konkrétan melyik opció a jobb megoldás, a szakirodalom nem tér ki egyértelműen, ezért az interneten megtalálható szoftverfejlesztői közösségek portáljain keresztül, a saját munkakörnyezetemben lévő programozókon át próbálkoztam a szóban forgó tapasztalatok, ajánlások begyűjtésével. Az említett közösségek által megosztott ismeretek alapján és a szoftverfejlesztő kollégák egybehangzó véleménye alapján egyértelművé vált számomra, hogy nagyon idő- és pénzigényes folyamat volna egy saját gyorsítótár változat készítése, akár hónapokig is eltarthat, valamint nagy valószínűséggel a közelébe sem érne hatékonyságban és felhasználhatóságban a kapott eredmény a már meglévő megoldásokhoz képest. A piacon számos, a konkrét alkalmazásra való implementációra kész *cache*-t megvalósító könyvtárat lehet beszerezni, sokakat akár – ha valamilyen limitációval is ugyan, de – ingyen is. A gyorsítótárak alapvető funkciója a webalkalmazások témakörében szinte minden esetben azonos, nagyon újító jellegű ötletet manapság nehéz lenne megalkotni a kérdéskörében. A fentiek alapján tehát egy meglévő *cache* megvalósítás használata mellett döntöttem, így „csupán” az implementációt kell megvalósítani.

Az implementáció elkészítésének nehézsége, valamint ennek időigénye nagyban függ attól, hogy milyen komplexitású az alkalmazás, amellyel kapcsolatban fel szeretnénk felhasználni a szóban forgó megvalósítást. A konkrét implementáció létrehozásakor figyelembe kell venni a szoftvercsomag adatainak struktúráját, hiszen az entitás példányok felolvasásakor, valamint az állományok módosításakor a hatékony és konzisztens állapotot megőrző megoldás osztályszinten továbbítani az információkat.

Ezek alapján tehát a bizonyos felépítésű adatokat újra megfelelő típusú példányokká való alakítását a gyorsítótárba helyezésük, valamint az innen való kiolvasásuk során kezelni kell. Ehhez az összes különböző entitás esetében külön-külön implementációt kell készíteni. Ez a feladat abban az esetben, hogy ha az adott program nagy számú eltérő osztályú példánnyal[3] operál, akkor ezzel egyenesen arányosan növekszik az implementálás ideje, ezen felül ennek hosszát az egyes osztályok összetettsége is erősen befolyásolhatja.

A gyorsítótár megvalósításhoz, – melyet az *Unipoll* vizsgáztató rendszer megfelelően elosztott rendszerű módozatban tudna felhasználni – olyan tesztelési környezetre van szükség, mely több alkalmazás szerverből, - valamint az általam optimálisnak tartott megoldás alapján – külön a gyorsítótár futására fenntartott szerver is szükséges. Ezeknek kialakítása a szoftvert fejlesztő vállalat esetén nem lehetetlen kihívás, azonban időigényes hiszen az informatikai rendszerek üzemeltetésével foglalkozó részlegnek össze kell állítania ezt az igényelt installációt, valamint értelemszerűen ezt megelőzően a megfelelő konzultációknak és engedélyezési folyamatoknak le kell zajlania. A szituációt befolyásoló tényező továbbá, hogy a *cache* implementáció korai állapotában ez a folyamat el sem kezdődhet, mindaddig amíg a konkrét eredményeket nem tud a tesztprogram szolgáltatni.

4.2 Tesztprogram alapján levonható konklúziók

A harmadik fejezetben tárgyalt és kielemzett tesztprogram alapján a vizsgált szoftveres gyorsítótár megvalósítások segítségével bizonyítottan kapacitás növekedés érhető el, valamint csökkenthető a hálózati sávszélesség felhasználása, aminek segítségével a középpontba helyezett mérőszám a szimultán kitöltések száma fokozható, így az egész szoftver nagyobb számú végfelhasználó párhuzamosan történő alkalmazása esetén is fenntartható és biztonságos módon tud zajlani a vizsgáztató rendszer. Az elvégzett kísérlet nyomán megalapozott elképzelés, hogy a harmadik féltől származó (*third party*), tehát az alkalmazástól független külső szervezetek által készített *cache* megoldásokkal közreműködésével elérhető az *Unipoll* szoftver működése során célként

[3] Az objektum orientált programozás alapvető elemét osztálynak nevezzük, mely egy konkrét dolog, fogalom vagy eljárás reprezentált modellje

kitűzött teljesítmény növekedés. A teszt az elosztott rendszerű *third party* gyorsítótár megvalósítások közül a már korábban említett *Couchbase* névre hallgató *cache* eljárás segítségével produkált eredményeket mutatta be, amely tényállástól függetlenül számos különböző szoftverkomponens található meg a tárgyalt probléma megoldására, melyekről az alábbi fejezetben még kitérek.

Az elkészített tesztalkalmazás által fókuszált problémakörbe tartozik, – a gyorsítótár mint olyan használatának létjogosultsága mellett – hogy a szakdolgozatban tárgyalt vizsgáztató szoftver esetében aktuális *cache* implementációja, mely a *.NET* keretrendszerbe integrált *runtime caching* megoldásnál kívánatosabb hatást lehet-e elérni teljes mértékben azonos körülmények fentállása esetén a *Couchbase* gyorsítótár megvalósítás segítségével. A kísérlet alapján levonható következtetés, hogy a két *cache* megoldás megfelelő alapos implementációja esetén a *Couchbase* közreműködésével operáló változat az korai függvényhívások során valamivel hosszabb válaszidőt produkáltak, illetve a többszöri ismétlés során némileg gyorsabb működést mutat, mely említett két tényező hozzávetőlegesen kiegyenlíti egymást. Ezek alapján nem jelenthető ki egyértelműen semelyik esetben sem, hogy az egyik megvalósítás magasabb szintű hatékonyságot biztosít a vizsgáztató rendszer futása során. A lényeges különbség azonban, hogy a *Couchbase* gyorsítótár elosztott rendszerű működést is lehetővé tesz, mely eljárás során megvalósíthatóvá válik a közel korlátlan teljesítménybeli skálázhatóság az adatok konzisztens állapotának megtartása mellett. Ezen tényezők az *Unipoll* alkalmazás számára szükséges funkciók, amelyek megléte egyértelműen implikálja, hogy a szoftver használta szempontjából a jövőben kifejezetten előnyös lenne a *Couchbase*, illetve az egyéb elosztott működést lehetővé tevő gyorsítótár megvalósítások.

4.2 A vizsgáztató rendszer számára ideális gyorsítótár tervezése

Az előző három fejezetben végzett kutatómunka, valamint levezetett tesztszoftver alapján megtervezhető a *Unipoll* vizsgáztató és kérdőív készítő alkalmazás működése szempontjából a lehető legmegfelelőbb gyorsítótár megoldás, valamint a példaalkalmazás során feltárt gyakorlati problémák, összefüggések segítségével levezethető, hogy az ehhez hasonló implementációk kivitelezése esetén milyen nehézségekbe ütközhet egy fejlesztőcsapat.

Az ideális gyorsítótár megvalósítás egy vizsgáztató rendszer számára, egy olyan szoftver megoldás, amely lehetővé teszi az alkalmazás kapacitásának skálázható módon való növelését, tehát, hogy mintegy építőkockánként lehessen a program futását támogató informatikai infrastruktúrát bővíteni anélkül, hogy ez bármilyen működésbeli zavart okozzon. Ezen felül egy hatékony *cache* implementáció lényegesen csökkenti az adatbázis egy kérésre való válaszidejét, mely mérőszám mértéke létfontosságú egy valós futási környezetben például egy egyetem webszerverein. Az optimális gyorsítótár a lehető legköltséghatékonyabbá teszi az adatbázis szerverekkel való kommunikációt, valamint minimalizálja a lokális és globális hálózatok sávszélességének felhasználását.

4.2.1 Hálózati topológia

Egy vizsgáztató rendszer esetében, amely nagy számú végfelhasználót szolgál ki, elosztott rendszerű gyorsítótár megoldást ideális alkalmaznia, hiszen ez teszi lehetővé a korlátlan skálázhatóságot és az adatok konzisztens állapotának fenntartását (Chengjian et al., 2015), ugyan az ilyen típusú *cache* megvalósítás hátránya, hogy magasabb költséget jelent az bevezetése, valamint folytonos üzemelése. Az elosztott gyorsítótárak problémakörében marginális kérdésként fölvetődik továbbá, hogy fizikailag, illetve topológiai szempontból hol helyezkedjen el a szoftverkomponens, ezen felül mekkora erőforrással rendelkező eszközön fusson. Számos megvalósításra érdemes variáció áll rendelkezésre a kérdés megválaszolására, amelyek nagyban függenek az informatikai infrastruktúrától, valamint a rendelkezésre álló technológiai és anyagi keretektől. Ha megfelelő anyagi erőforrások állnak rendelkezésre, akkor érdemes egy saját, külön erre a célra kijelölt szervergéppel támogatni a gyorsítótárat, így a lehető legegyszerűbb szoftvertechnológiai megoldással vihető véghez az implementáció. Ebben az esetben az összes alkalmazás vagy webszerver mindig ezen az eszközön keresi először az szükséges adatokat a *cache*-ből való olvasás során, nincs szükség kommunikációra az egyes alkalmazás implementációk között. Ilyenkor ideális esetben közös, helyi hálózatban vannak az alkalmazást futtató szerverek, így nem fogyasztják az internet hálózat sávszélességét, valamint ezen szolgáltatás kimaradásából származó nehézségeket is ki lehet küszöbölni.

Eltérő variációkban a gyorsítótár valamelyik vagy az összes szoftverpéldányt üzemeltető berendezésen helyezkednek el a *cache* megoldások. Ekkor nincs szükség

egy elszeparált szervergép működtetésére, mely jelentős költséget ölelhet fel, viszont minden egyes adatállomány változáskor meg kell bizonyosodni róla, hogy az összes gyorsítótár példány esetében a megfelelő naprakész információk kerülnek eltárolásra, illetve frissítve legyenek ezek. Ezt olyan módszerrel lehet abszolválni, ami az adatok módosítása esetén egy tranzakció során az összes alkalmazást futtató eszköz gyorsítótárának memóriájában felülírja az adott, elavult állományokat a legújabbra. Ilyenkor a tranzakció csak akkor teljesül, ha az összes átmeneti tárhelyen egyaránt frissül a kérdéses adathalmaz, ha valamiért bármelyik eszközön meghiúsul ez a folyamat, akkor a változás semelyik helyen nem megy végbe. Ezen a megoldáson egyértelműen érezhető, hogy az adatok módosítása számottevően lassabban megy végbe, bár ugye ez az alkalmazás szerverek számától is függ, de nagyobb rendszerek esetén nem igazán optimális, hiszen minél több van a tárgyalt eszközből, annál több helyen kell módosítani egyidejűleg. Hátránya továbbá még ennek a megoldásnak, hogy lényegesen nagyobb mennyiségű hálózati adatforgalmat generálhat. Olyan esetekben amikor kizárólag kevésbé frekventált időpontokban történik az adatok módosítása, működés képes lehet ez a kivitelezés, bár szoftveres megoldása is jóval bonyolultabb, mint az erre a célra fenntartott szerver használatának segítségével.

4.2.2 Caching stratégia

Az eltérő gyorsítótár megvalósítások között különböző lehet még, hogy az átmeneti memória, illetve adatbázis elérése milyen sorrendben történik, azaz milyen *caching* stratégia szerint zajlik. Ezen témakörben is különféle megoldások léteznek, melyeket sok esetben nem lehet jobbnak vagy rosszabbnak titulálni egymásnál, hiszen ez a bizonyos értékítélet számos változótól lehet függővé tenni. Többek között a *cache-aside* nevezetű stratégia felhasználható, hiszen ez akkor hatékony a több végfelhasználó azonos adatokat kér el az adatbázistól, például egy vizsga kitöltésénél ez többé-kevésbé így működik. Ennek a változatnak az igénybevétele akkor lehet igazán biztonságos, ha külön szerveren fut a gyorsítótár, mert a stratégiát használva az állományok futás közbeni változtatása esetén az összes gyorsítótár példányban végre kell hajtani a módosításokat, és ez értelemszerűen akkor a legkézenfekvőbb, ha csak egy ilyen átmeneti tár alkalmazásával üzemel az alkalmazás. A *read-through* és a *write-through* megoldások is használhatóak egy számonkérést támogató szoftver esetén, mely

használata nagyban elősegíti az adatok konzisztenciájának fenntartását. Hátránya viszont, hogy minden egyes adathívás a gyorsítótár implementációján keresztül történik, mely folyamat zavartalan működése nagyobb sávszélesség felhasználás mellett, valamint nagyobb hardver erőforrás kihasználtság révén tartható fent.

4.4 Kész alternatívák bemutatása

A különböző készen beszerezhető megoldások esetében a gyorsítótár hálózatban elfoglalt helyét, cache stratégiáját, törlési politikát sokszor a készítő fél határozza meg, tehát a gyorsítótár működésének módját, stratégiáját eleve meghatározza a konkrét szoftvertermék. Némely „polcos” *cache* megvalósítás alkalmas ezen a tényezők utólagos megváltoztatására, ennek ellenére érdemes az ilyen jellegű a szoftverkomponenseket igénybe vevő magánszemélyeknek vagy társaságoknak a beszerzés előtt informálódni a megoldás működési elvével kapcsolatban, amennyiben ezek a tényezők befolyásolhatják az általuk üzemeltetett szoftver hatékonyságát. Ezen tényezőkről a saját alkalmazásaik szükségletei alapján kell dönteniük annak érdekében, hogy a számukra legmegfelelőbb gyorsítótár megvalósítás szerezzék be.

13. ábra Redis embléma

Forrás: https://redis.io/

4.4.1 Redis

A *Redis* névre hallgató *cache* megoldást fejlesztő projekt 2009 elején indult egy olasz szoftverfejlesztő Salvatore Sanfilippo által. A megoldás egy nyílt forráskódú szoftverterméke, melynek elsődleges célja egy belső vállalti rendszer teljesítményének javítása volt. 2009 júniusára a gyorsítótár megvalósítást már éles környezetben való felhasználása is elindult. A szoftver előnyei közé tartozik, hogy más kulcs-érték pár tárolására alkalmas programmal ellentétben számos adattípus használatát támogatja, valamint az, hogy a működése során az ideiglenes tárhely gyors elérésű memóriában kerül létrehozásra ebből kifolyólag. Ezen típusú gyorsítótár működtetése azoknál az alkalmazásoknál a legelőnyösebb melyeknél az adatok beérkezésének sebessége a

legfontosabb faktor. A *Redis* egy egyszerűen felhasználható, számos programozási nyelvet támogató megoldás, melynek incializálása percek alatt véghezvihető. Hátránya, hogy az által tárolt adatokat nem lehet elérni adatbázis lekérdezések formájában, kizárólag szoftverkód futtatása révén manipulálhatóak az eltárolt állományok. Elosztott működéséhez több párhuzamosan üzemelő példányt kell létrehozni, melyek megfelelő kezelése komplikált lehet. Ezen felül működése során a futtató eszközök *RAM*-jának számottevő részét lefoglalják. Ingyenes változatok használata esetén a kezelő személyek számára csak egy nagyon kezdetleges szöveges (*Command Line Interface*) használata elérhető. A megoldás kizárólag *cache-aside* módon, azaz sosem az alkalmazás szerver és az adatbázis szerver közé ékelődve üzemel (Deepti Mittal, 2022).

14. ábra Memcached embléma

Forrás: https://medium.com/swlh/what-is-memcached-d1498623db3b

4.4.2 Memcached

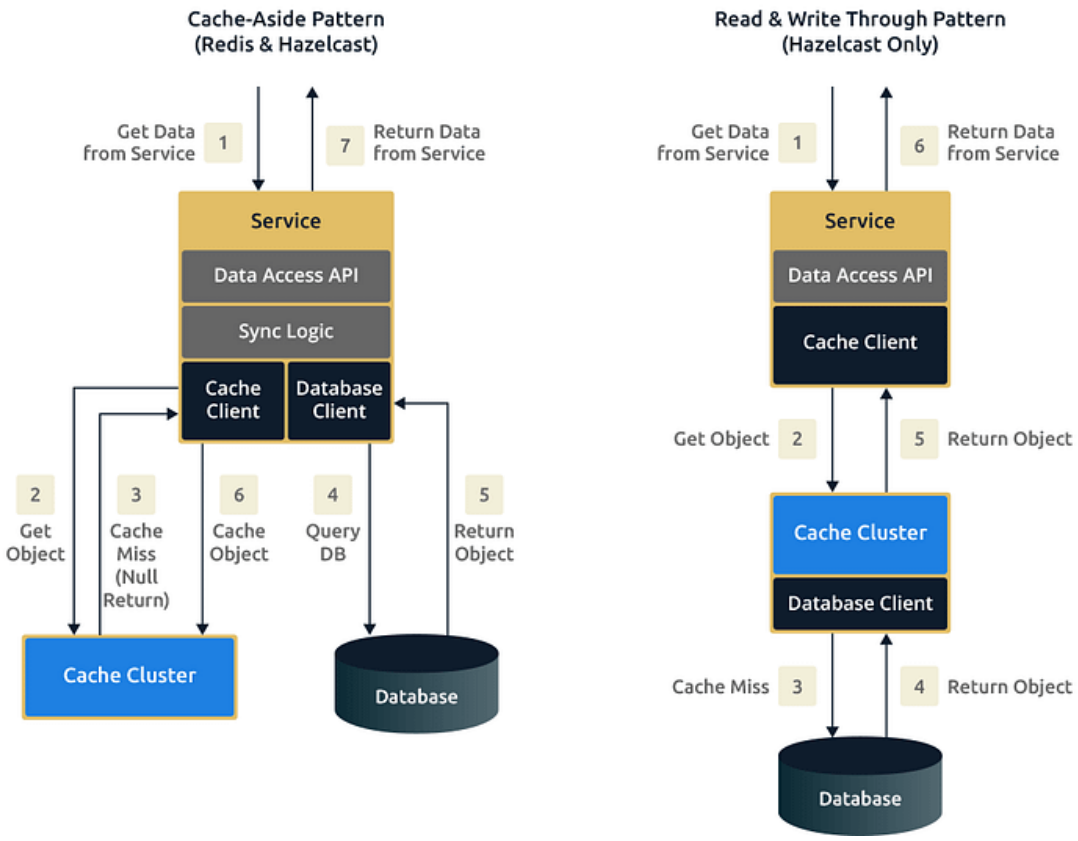
A *Memcached* gyorsítótár megvalósítását 2003-ban készítette Brad Fitzpatrick, aki a saját weboldalának a *LiveJournal*-nak gyorsítása érdekében a fejlesztette le ezt a megoldást. A szoftver eredetileg *Perl* nyelven íródott, majd később Anatoly Vorobey írta át *C*-re a programkódot. A *Redis*-hez hasonlóan a *Memcached* is nyílt forráskódú szoftver, mely szintén kulcs-érték párok útján tárolja ideiglenes az információkat, ily módon téve lehetővé a gyors adatáramlást. A *Redis*-el ellentétben mely „egyszálas” futtatás tesz csak lehetővé a szoftver támogatja a „többszálú” kódvégrehajtást, ezúton használva ki a több processzor maggal rendelkező processzorokat. A tárgyalt *cache* – az előzővel ellentétben – kizárólag szöveges (*string*) formátumban tárolja le az érték párokat, ebből kifolyólag az adatok feldolgozásához, illetve módosításához az egész állomány betöltése szükséges. A megvalósítás működése során szintén csak a memóriában tárol adatokat, ám a *Redis* esetében készülnek biztonsági mentések, a *Memcached* alapértelmezetten nem készít ilyen jellegű mentéseket, ezért az alkalmazása esetén ezt a problémát megoldandó addicionális fejlesztésre van szükség (Deepti Mittal 2022).

15. ábra Hazelcast embléma

Forrás: https://github.com/hazelcast

4.4.3 Hazelcast

Az elosztott rendszerű gyorsítótár szoftver fejlesztését 2008-ban kezdte meg Talip Ozturk és Fuad Malikov, mely nyíltforráskód alkalmazás használata előnyös komplex adatstruktúrájú állományok átmeneti tárolására. A *cache* megoldás *Java* programozási nyelven íródott, és szintén alkalmas többszálon futó programok gyorsítására, ezen felül támogatja az elosztott rendszerben használatos csomópontok (*node)* automatikus megtalálását. A gyorsítótár megoldás használata során az biztonsági mentések és az ideiglenes adattárolás helye ugyanazon a fizikai eszközön elérhető, ennek köszönhetően kevesebb eszközt használata is elegendő a stabil működéshez. A szoftverkomponens hátránya, hogy kevés programozási nyelve felhasználást támogatja, valamint egyszerre alkalmas kulcs-érték párok, illetve összetett objektum példányok eltárolására. A *Hazelcast cache* továbbá támogatja az adatbázis szerverek által feldolgozható lekérdezéshez hasonló (*SQL like*) kéréseket, melynek köszönhetően csak a szükséges, szűrt adatok utaznak az eszközök közötti hálózaton, ami erősen csökkentheti a sávszélesség felhasználását az alkalmazás és a gyorsítótár közötti kommunikáció során. A megvalósítás ezen felül alkalmas a rendszer csomópontjain való különböző részszámítások elvégzésére. Különféle *cache* stratégiákat egyszerre támogat a *Hazelcast*, így a *cache-aside,* valamint *read-through, write-through* mintázatok közül választhat az ezen megvalósítást alkalmazó szervezet. Ellenben hátránya a tárgyalt *cache* megoldásnak, hogy kevés dokumentáció lelhető föl hozzá, valamint a fejlesztői közösség által még kevéssé ismert (Deepti Mittal 2022).



16. ábra Redis és Hazelcast cache stratégiája

Forrás: https://deeptimittalblogger.medium.com/redis-and-its-alternatives-important-points-to-decide-distributed-caches-7ecc8515a37a

4.4.4 Couchbase

17. ábra Couchbase embléma

Forrás: https://en.m.wikipedia.org/wiki/File:Couchbase,_Inc._official_logo.png

A *Couchbase* egy általános felhasználású (*general-purpose*) *NoSQL*, dokumentum orientált[4] adatbázis típus, melynek készítői részt vettek a *Memcached* fejlesztésében. A *Couchbase* első stabil változata 2010 augusztusában került kiadásra. A megoldás hatékony, kifejezetten jól skálázható *cache* implementációja miatt vonzó alternatívájává vált a *Redis* -nek, valamint a *Memcached*-nek. Az alkalmazás elosztott rendszerek esetében is jól használható, folyamatai során a szerver gép(ek) memóriájában tárolja el az egymás után többször is kért adatokat. A megvalósítás kulcs-érték párok segítségével tárolja el a kérdéses állományokat, valamint jól kezeli a nagyszámú párhuzamosan indított adatkérelmet. Több hálózati csomópont esetén magas hibatűrő képességgel bír, mert ebben az esetben redundáns adattárolásra is alkalmas. Ezen felül jól skálázható kapacitású rendszer kialakítására alkalmas. A szoftvert úgy tervezeték, hogy egy csomóponttól akár több száz csomópontig is problémamentesen működjön. A gyorsítótár megvalósítás képes a többszálon futó rendszerek kiszolgálására (Tiesinga 2023; https://info.couchbase.com 2023).

4.4.5 Elosztott rendszerű gyorsítótár alkalmazások összehasonlítása

Az alábbi alfejezetben röviden bemutatott elosztott rendszerű gyorsítótár variációk összehasonlítására készítettem egy táblázatot, amelyben az egyes megoldások által igénybe vehető funkcionalitások, az általuk használt technológiák, valamint *cache* stratégiák szerepelnek. A táblázat végén feltüntettet a szolgáltatás használatért fizetendő költségek jellemzően nem a konkrét szoftverelemek, hiszen ezek közül majdnem mindegyik teljese mértékben nyílt forráskódú, hanem jellemzően a hozzájuk társítható

[4] Olyan félig strukturált adattárolási forma, mely használata során a strukturáltságnak köszönhetően úgynevezett *metaadatok* nyerhetőek ki a dokumentumból

adatbázis és csomópont jellegű termékek használatáért cserébe megtérítendő összeget jelölik. Ezeket a fizetős szolgáltatásokat nem is minden esetben a szoftver készítője nyújtja, hanem fejlesztőkkel szerződésben más üzleti társaságok (például az *azure*). Természetesen számos más hasonló termékeket vagy szolgáltatások nyújtó megoldások, valamint cégek is elérhetőek a piacon, melyek közül csak a legelterjedtebbeket, a legtöbbek által igénybe vett megoldások működését vizsgáltam meg.

		Couchbase	Redis	Memcached	Hazelcast
Multithread		yes	no	yes	yes
Strategy		yes			
	Cache-aside	yes	yes	yes	yes
	Cache-through	yes	no	no	yes
Backup		yes	yes	no	yes
Node architecture		yes	master-slave	no replication	peer-to-peer
Price		$0.28-$0.56/hour/node (azure cache)	$0.55-$0.45/hour/2nodes	no server offer	unique offer

1. táblázat Elosztott rendszerű gyorsítótár alkalmazások összehasonlítása

Forrás: Saját készítés

ÖSSZEFOGLALÁS

A távoktatás növekvő térhódítása miatt egyre nagyobb mértékben szorulnak kapacitásnövelésre, hatékonyságuk javítására a számonkérést lebonyolító webalkalmazások, így az *Unipoll* szoftver is. A jövőben várhatóan ez a hatás még inkább fel fog erősödni, ebből kifolyólag a tárgyalt típusú alkalmazások fejlesztői csapatának a folyamatos teljesítmény javításon kell dolgozniuk, hiszen előreláthatólag nagy időtávban a különböző, új technológiák megjelenésétől függetlenül frekventáltan igénybe vett programok lesznek a vizsgáztató rendszerek.

Jelen dolgozat célja az volt, hogy a vizsgáztató alkalmazások – különösen a konkrétan tárgyalt program – működése során a szűk keresztmetszetként megjelenő párhuzamos kitöltések számának skálázható növelésére lehetséges megoldási javaslatot nyújtson. A dolgozat ezen megoldások közül kifejezetten a gyorsítótárak témakörében keresi a probléma kiküszöbölésének eszközét. A *cache* típusokat feltérképezve több lehetséges válfaj is jól felhasználható megvalósításnak tűnik, melyek különböző tulajdonságait eltérő módon kombinálva a különféle működésű alkalmazások igényeihez jól igazíthatók ezek a gyorsítótár implementációk.

A szakirodalom alapos átvizsgálása révén elméleti síkon elmondható, hogy a vizsgáztató szoftverek kapacitásának közel korlátlan módon való növeléséhez, a legjobb megoldás az elosztott működést támogató gyorsítótár megvalósítások használata. Az ilyen típusú *cache* szoftverek igénybevételének előnye, hogy az adatok – jelen esetben főként kérdőívek, vizsgák – konzisztens állapotának fenntarthatóságát lényegesen egyszerűbb garantálni. Ezen felül a webalkalmazások által használt, több csomópont támogatásával üzemelő adatréteg számottevő módon mérsékeli az adatbázis szerverek által földolgozandó, valamint kiszámítandó, összeállítandó adatok mennyiségét, ezáltal nagy mértékben csökkentve az adatbázis leterheltségét, valamint a hálózati sávszélesség felhasználását. Ezek a tényezők könnyen befolyásolhatják egy webalkalmazás működésének zavartalanságát, illetve a végfelhasználók által érzékelt gyorsaságot.

Az általam végzett kutatás után megpróbáltam a gyakorlatba átvinni a szakirodalomból megszerzett információt, tudásanyagot. Készítettem egy tesztprogramot, melynek célja, hogy igazolja a gyorsítótár megoldások pozitív hatását a webalkalmazásokra nézve. A kérdéses vizsgáztató rendszer működésének javulását, hatékonyságának növekedését a

kísérlet folyamán megvalósított kis méretű alkalmazás segítségével mért adatokkal terveztem alátámasztani. A konkrét mérőszámok monitorozását egy nyílt forráskódú, bárki által elérhető szoftverrel hajtottam vége. A vizsgált értékek az *api* függvényhívások során az adatbázis szerver felé közvetített adatkérelmek átlagos és maximális válaszideje, valamint az adatbázissal való kommunikáció folytán egy másodperc alatt beérkező adatok mennyisége voltak. A teszt során három, egymástól eltérő függvényhívás segítségével produkáltam adatokat a három különböző implementáció működéséről.

Az első adatbázissal való kommunikációt megvalósító megoldás egyszerű *SQL* lekérdező nyelv használatának hatására gyűjtötte be a függvényben meghatározott adatokat. A másik kettő gyorsítótár megvalósítást igénybe vevő funkció ugyanezt az előbb említett adatbázison történő lekérdezést hajtotta végre azzal a különbséggel, hogy ezek futásuk során eltérő eljárással ideiglenes másolatot készítettek az igényelt adatokról. Az így létrejött átmeneti tárhelyekre kiírt adatok révén az újbóli adatkérelmek már számottevően gyorsabban érkeztek meg. Az adatok rövidebb idő alatt történő beszerzése azért volt lehetséges a két gyorsítótárat használó függvény esetében, mert nem volt arra szükségük, hogy az adatbázistól kérjék el a kívánt információkat, hiszen ezeket már a saját futtató környezetük gyors elérésű memóriájából ki tudták olvasni.

Az egyik gyorsítótár megoldás a *.NET* fejlesztői keretrendszerbe épített *runtime* névtér alatt megtalálható megvalósítás volt. A másik változat egy külső cég által fejlesztett implementáció, mely a *Couchbase* névre hallgat. A két megoldás közötti különbség, hogy az utóbbi képes elosztott rendszerek üzemeltetésére az által, hogy létrehoz magának egy szervert, melyen tárolja, valamint elérhetővé teszi a hálózatban lévő többi eszköz számára ezeket. Erre azért van szükség, mert a vizsgáztató rendszereket egyre többen használják, így egyre nagyobb kapacitást kell biztosítaniuk, melyre megfelelő megoldást nyújt az elosztott rendszerű gyorsítótárak alkalmazása.

A szakdolgozatom célja abban az értelemben megvalósult, hogy be tudtam bizonyítani, hogy a gyorsítótárak használata hatékonyabbá teszi a vizsgáztató rendszerek működését. A konkrét vizsgáztató rendszer, az *Unipoll* számára készítendő gyorsítótár megoldás elkészítése azonban lényegesen nagyobb volumenű feladatnak bizonyult annál,

minthogy a szakdolgozatom kereti között ezt megvalósítsam, a tervezési folyamatot és a gyorsítótár implementáció megvalósításának kezdeti fázisait azonban elvégeztem, a kutatás során bemutattam, hogy több, jól működő megoldást be lehet szerezni, melyek alkalmasak a szoftver hatékonyságának javítására.

IRODALOMJEGYZÉK

1. Monolescu D. – Schifter C. (2003) Distance Education Evolution - Issues and Case Studies, Hershey: Information Science Publishing
2. Borje Holmberg (1995): Theory and Practice of Distance Education (Routledge Studies in Distance Education) 2.o., London: Routledge
3. A W Bates (2005): Technology, Distributed Learning and Distance Education, London: Routledge
4. Som Naidua (2006): E-Learning: Guidebook of Principles, Procedures and Practices. New Delhi, India: Commonwealth Educational Media Center for Asia
5. Michael Grahame Moore - William G. Anderson (2003): Handbook of Distance Education, London: Lawrence Erlbaum Associates,
6. Lengyel Zsuzsanna Mária (2007): E-learning: tanulás a világhálón keresztül 8-21.o. Debrecen
7. Lannert Judith (2022): A pandémia miatti iskolabezárások és a digitális oktatás hatása a tanulók felkészültségére a közoktatásban és a felsőoktatásban Társadalmi Riport. Budapest: TÁRKI
8. Saritha Reddy (2023. 03.31). Online Exam Software – Types, Features, Benefits, Advantages & Disadvantages. etutor.co https://etutor.co/blog/online-exam-software/ (Utolsó letöltés: 2023. 11. 15.)
9. Josheena Jose (2021): Study On Effect Of E-Learning Process During Covid 19 Pandemic. Irinjalakuda: Christ College
10. Alfred Yang. (2022.01.12.). Common performance bottlenecks in a web application server. medium.com. https://medium.com/finnovate-io/common-performance-bottlenecks-in-a-web-application-server-674954b0b05c (Utolsó letöltés 2023.11.19.)
11. Sven Hammar (2017.03.30.) The 5 Most Common Application Bottlenecks. APMDigest.com. https://www.apmdigest.com/the-5-most-common-application-bottlenecks. (Utolsó letöltés 2023.11.19.)
12. Martin Fowler – David Rice – Matthew Foemmel – Edward Hieatt – Robert Mee – Randy Stafford (2002): Patterns of Enterprise Application - Architecture Signature Series, 1, 25.o. Addison Wesley
13. Adel F (2019.03.28) Architecture of complex web applications. Leanbpub

14. Larson, P.A. – Goldstein – Zhou, J. (2004): IEEE Comput. Soc Proceedings. 20th International Conference 177-188.o.
15. Ali M. Alakeel (2010): A Guide to Dynamic Load Balancing in Distributed Computer Systems - IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.6, 153.o. ,Tabuk: College of Computing and Information Technology University of Tabuk
16. Kanika Modi. (2021.09.11). System Design - Distributed Database vs Cache. medium.com. https://medium.com/codex/system-design-distributed-database-vs-cache-eab8e067bd15 (Utolsó letöltés 2023.11.21.)
17. Manish Kumar Paneri (2023.06.13.). How Caching Helps In Improving Performance Of Application. medium.com. https://medium.com/@manishkp220/how-caching-helps-in-improving-performance-of-application-a2c4ca0cfcc2 (Utolsó letöltés 2023.11.28.)
18. Podlipnig Stefan – Böszörmenyi, Laszlo (2003): A survey of Web cache replacement strategies ACM Computing Surveys, 374.o
19. Zhong Liang – Zheng, Xueqian – Liu, Yong – Wang, Mengting (2020): Cache hit ratio maximization in device-to-device - China Communications 232.o.
20. Chiradeep BasuMallick (2022.12.12.). What Is Cache? Definition, Working, Types, and Importance. spiceworks.com. https://www.spiceworks.com/tech/tech-101/articles/what-is-cache/ (Utolsó letöltés: 2023.11.23)
21. Gabriel Torres (2007.09.12.). How The Memory Cache Works. hardwaresecrets.com. http://gec.di.uminho.pt/Discip/MInf/cpd0708/SCD/MemoryCache_HwSecrets.pdf (Utolsó letöltés: 2023.11.23)
22. Cisco Network Caching, (2000) https://www.cisco.com/c/dam/global/de_at/assets/docs/Net_Caching.pdf (Utolsó letöltés: 2023.11.13)
23. Thomas Shinder (2008.09.02.). Understanding Web Caching Concepts for the ISA Firewall. techgenix.com. https://techgenix.com/understanding-web-caching-concepts-isa-firewall/ (Utolsó letöltés: 2023.11.16.)

24. Yorgos Fountis (2023.10.11.). How does the browser cache work? pressidium.com. https://pressidium.com/blog/browser-cache-work/. (Utolsó letöltés 2023.11.16.)
25. S. Messaoud – H. Youssef International Journal of Communication Systems (2009): An analytical model for the performance evaluation of stack-based Web cache replacement algorithms 1-4.o. Tunézia: Sousse-i Egyetem
26. Love, Robert (2011): Linux Kernel Development
27. Erik Nygren – Ramesh K. Sitaraman – Jennifer Sun (2012): The Akamai Network: A Platform for High-Performance Internet Applications, Cambridge Akamai Technologies
28. John Dilley – Bruce Maggs – Jay Parikh – Harald Prokop – Ramesh Sitaraman – Bill Weihl (2002): Globally Distributed Content Delivery, Akamai Technologies; IEEE Internet Computing
29. Peter Norvig (1991): Techniques for Automatic Memoization with Applications to Context-Free Parsing - Computational Linguistics Vol.17 91-98.o.
30. Stasoz (2021.12.10.). In-Memory and Distributed cache (.NET Core). medium.com. https://blog.devgenius.io/in-memory-and-distributed-cache-net-core-9be16bec34d7 (Utolsó letöltés: 2023.11.29)
31. Sudheer Sandu (2022.07.08.) Distributed Caching — The Only Guide You'll Ever Need. medium.com. https://medium.com/@sudheer.sandu/distributed-caching-the-only-guide-youll-ever-need-fe152357f912. (Utolsó letöltés: 2023.11.29)
32. In-Memory Cache. Gridgain. https://www.gridgain.com/resources/glossary/in-memory-computing-platform/in-memory-cache (2023.11.25.)
33. Liu Chengjian – Ouyang Kai – - Chu Xiaowen – Liu, Hai – Leung (2015): R-memcached A reliable in-memory cache for big key-value - Tsinghua Science
34. Umer Mansoor (2017.08.11). Caching Strategies and How to Choose the Right One. codeahoy.com. https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/. (Utolsó letöltés: 2023.11.29)
35. Rahul (2022.12.15). Caching, caching, caching everywhere. medium.hu https://medium.com/@SaiRahulAkarapu/caching-caching-caching-everywhere-8855ebe56f57. (Utolsó letöltés: 2023.11.29)

36. Alex DeBrie (2023): 6 common caching design patterns to execute your caching strategy, www.gomomento.com/blog
37. Moshe Binieli (2023): Cache Strategies – www.medium.com/@mmoshikoo/cache-strategies-996e91c80303
38. Thomas Hamilton (2023.11.17.) What is JMeter? Introduction & Uses. guru99.com. https://www.guru99.com/introduction-to-jmeter.html. (Utolsó letöltés: 2023.11.17.)
39. Pasan Williams (2019.03.21.) Distributed caching in Redis https://medium.com/@123williams93/distributed-caching-in-redis-8ff882bf79ac (Utolsó letöltés 2023.12.07.)
40. Deepti Mittal (Jan 17, 2022) Redis: Something to think of while choosing data store next time https://deeptimittalblogger.medium.com/redis-something-to-think-of-while-choosing-data-store-next-time-377727039ff8 (Utolsó letöltés: 2023.12.01.)
41. Deepti Mittal: (Jan 23, 2022) Redis and its alternatives: Important points to decide distributed caches https://deeptimittalblogger.medium.com/redis-and-its-alternatives-important-points-to-decide-distributed-caches-7ecc8515a37a (Utolsó letöltés: 2023.12.03.)
42. Johan Tiesinga (2023): Improving Performance Using Distributed Cache with Couchbase - https://medium.com/agoda-engineering/improving-performance-using-distributed-cache-with-couchbase-aa7ebf3a12c9 (Utolsó letöltés: 2023.12.04.)
43. High-Performance Applications with Distributed Caching (2023) https://info.couchbase.com/rs/302-GJY-034/images/High_Performance_With_Distributed_Caching_Couchbase.pdf (Utolsó leöltés:2023.12.04)