Elias Keränen

Tiedostojen satunnainen lukeminen Javalla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

12.4.2025

Tiivistelmä

Tekijä:	Elias Keränen
Otsikko:	Tiedostojen satunnainen lukeminen Javalla
Sivumäärä:	34 sivua
Aika:	5.4.2025
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaaja:	Jorma Räty, Lehtori

Tässä opinnäytetyössä tarkastellaan Java-pohjaisen pelipalvelinsovelluksen ajonaikaista muistinkäyttöä sekä sen optimointimahdollisuuksia. Opinnäytetyössä käsitellään tietokoneen, Javan ja Java Virtual Machinen (JVM) tapoja käsitellä tiedostoja sekä vertaillaan eri menetelmiä suurten tiedostojen lukemiseen muistinkäytön ja suorituskyvyn kannalta.

Erityisesti keskitytään tiedostojen satunnaislukuun, jota analysoidaan perinteisemmän I/O- ja uudemman NIO-pakettien tarjoamien luokkien kannalta. Tarkastelussa käytetään Kotlinilla toteutettuja koodiesimerkkejä, joilla havainnoidaan eri lukutapojen vaikutusta suorituskykyyn ja muistinkäyttöön.

Ratkaisujen resurssienkäyttöä arvioidaan tutkimalla JVM:n muistinkäyttöä ja analysoidaan heap dump -tiedostoja. Suorituskykyä vertaillaan testitapauksilla ja Java Microbenchmark Harness (JMH) -suorituskykytestillä.

Avainsanat: Java, Kotlin, JVM, Benchmark, JMH, Heap Dump

Abstract

Author:	Elias Keränen
Title:	Random Access File Reading in Java
Number of Pages:	34 pages
Date:	12 April 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisor:	Jorma Räty, Senior Lecturer

This thesis examines the runtime memory usage of a Java-based game server application and explores potential optimization strategies. It discusses how computers, Java, and the Java Virtual Machine (JVM) handle files and compares different approaches to reading large files in terms of memory usage and performance.

A particular focus is placed on random access files, which are analyzed using both traditional I/O and the more modern solutions provided by the NIO package. The study utilizes Kotlin-based code examples to demonstrate the of impact different reading methods on performance and memory usage.

The resource usage of the solutions is assessed by examining JVM memory consumption and analyzing heap dump files. The performance is compared using test cases and Java Microbenchmark Harness (JMH) tests.

Keywords: Java, Kotlin, JVM, Benchmark, JMH, Heap Dump

Sisällys

Lyhenteet

JVM: *Java Virtual Machine*. Java-koodia suorittava virtuaalikone.

WAR: *Web Application Archive*. Pakattu tiedosto, joka sisältää sovelluksen tarvitsemat resurssit, kirjastot ja käännetyt Java-luokat.

JIT: *Just In Time.* JIT compiler, ajonaikainen kääntäjä, joka muuntaa lähdekoodin suoritettavan ohjelman ajon aikana konekieleksi.

API: *Application Programming Interface.* Ohjelmointirajapinta.

NIO: *New Input/Output*. Javan paketti, joka sisältää mm. buffereihin, merkistöihin ja kanaviin liittyvät luokat ja rajapinnat.

CPU: *Central Processing Unit*. Tietokoneen suoritin.

PID: *Process Identifier*. Jokaisella käynnissä olevalla prosessilla Linux -käyttöjärjestelmässä on oma uniikki tunnistenumero.

LIFO: *Last In, First Out*. Tietorakenne, jossa viimeisimpänä lisätty tietue poistetaan tai käsitellään ensimmäisenä.

JMH: *Java Microbenchmark Harness*. Ohjelmistokehys Java-ohjelmien suorituskyvyn mittaamiseen.

JSON: *JavaScript Object Notation.* Tiedostotyyppi, jossa tieto on tallennettuna avain-arvo -pareina.

1 Johdanto

Tässä opinnäytetyössä esitellään Air Dicelle kehittämääni ratkaisua Javalla kirjoitettujen pelipalvelinsovellusten ajon aikaisen muistinkäytön vähentämiseen.

Kehittäessämme uuden tyyppisen pelin prototyyppiä havaittiin seuraavanlainen ongelma: ajon aikana palvelin tarvitsi kookkaita tiedostoja toimiakseen ja alkuperäisessä versiossa nämä tiedostot ladattiin suoraan muistiin. Koska yksittäisellä palvelinkoneella on samanaikaisesti käynnissä usean pelin palvelinsovelluksia, todettiin, ettei haluta turhaan lisätä ylimääräistä kuormaa palvelinympäristöön.

Tässä opinnäytetyössä käydään läpi tapaa, jolla ongelma ratkaistiin sekä vertaillaan vaihtoehtoisia tapoja lukea tietoa Java-pohjaisten ohjelmien muistiin. Oleellinen osa muistinkäytön vähentämisessä oli siirtyä käyttämään Javan RandomAccessFile-luokkaa, jonka avulla on mahdollista lukea haluttu määrä tietoa tiedostosta mielivaltaisista kohdista. Ratkaisutapoja vertaillaan suorituskyvyn ja muistinkäytön kannalta testitapauksilla ja profilointituloksia hyödyntäen.

Alkuperäisestä prototyypistä on kulunut jo pari vuotta aikaa, ja sitä käyttäviä pelejä on aktiivisesti tuotannossa. Ratkaisua on sittemmin jatkokehitetty, ja sitä käyttäviä pelejä on julkaistu useita kappaleita.

2 Ongelman esittely

2.1 Skenaarioista

Käytetään esimerkkinä korttipeliä, jossa jokaisen pelikierroksen alussa sekoitetaan korttipakka. Uuden pelin kehitysprosessin alkuvaiheessa käytettiin täysin satunnaista arvontaa. Hyvin pian huomattiin, että täysin sekoitettujen pakkojen käyttö aiheutti haasteita pelimekaniikkojen tasapainottamisen suhteen. Tietynlaisten tilanteiden esiintyvyyttä oli mahdotonta kontrolloida. Osa pelikierroksista saattoi olla huomattavasti helpompia tai vaikeampia kuin toiset.

Tämän vuoksi päädyttiin tekemään pelistä skenaariopohjainen, eli normaalin sekoittamisen sijaan peli käyttäisi joukkoa etukäteen valittuja korttipakan järjestyksiä, joista jokaisen pelikierroksen alussa arvottaisiin yksi. Skenaarioiden avulla pelissä esiintyviä tilanteita pystyttiin säätämään hyvin tarkasti. Liian helpot tai vaikeat kierrokset pystyttiin karsimaan.

Yksittäinen skenaario sisältää kaikki korttipakan 52 korttia jossakin hyväksi todetussa järjestyksessä. Skenaariossa jokainen kortti koostuu sen maan ensimmäisestä kirjaimesta (H = Hearts, S = Spades, D = Diamonds ja C = Clubs) ja kortin arvosta (A = Ace, K = King, Q = Queen, J = Jack sekä numerot 2-10). Esimerkiksi ”5H” tarkoittaa herttavitosta ja ”KD” ruutukuningasta.

Kotlinilla kirjoitettu koodiesimerkki, jossa luodaan pakka kaikkine kortteineen muodossa AH, KH, QH, JH, 10H, …, 2C.

```
class Deck {
    // H hearts, S spades, D diamonds, C clubs
    private val suits = listOf("H", "S", "D", "C")
    private val ranks = listOf("A", "K", "Q", "J", "10", "9", "8", "7", "6", "5", "4", "3", "2")
    private val cards = suits.flatMap { suit -> ranks.map { rank -> "$rank$suit"
} }.toMutableList()

    override fun toString(): String {
        return cards.joinToString(",")
    }
```

```
    fun shuffle() {
        cards.shuffle()
    }
}
```

Esimerkkikoodi 1: Yksinkertainen Kotlin-luokka, joka luo merkkijonon pakan 52 kortista muotoon AH,KH,QH,JH,10H,9H, …, 2H.

Päätettiin, että skenaarioiden luonti ja valinta suoritetaan pelin palvelinsovelluksen ulkopuolella. Käynnistyessään palvelinsovellus lukee etukäteen valmistellun skenaariotiedoston.

Pelin suunnitteluvaiheessa arvioitiin, että riittävään lopputulosten varianssiin ja monipuoliseen pelikokemukseen tarvitaan satojatuhansia skenaarioita. Seuraavassa koodinäytteessä luodaan opinnäytetyössä käytettävä skenaariotiedosto.

```
fun generateTestScenarios(): MutableList<String> {
    val decks = mutableListOf<String>()
    repeat(250_000) {
        val d = Deck()
        d.shuffle()
        decks.add(d.toString())
    }
    return decks
}
```

Esimerkkikoodi 2: Luodaan 250 000 pakkaa ja sekoitetaan ne Kotlinin Collections -paketin shuffle-metodilla.

Kirjoitetaan jokainen generoitu pakka tiedostoon yksi skenaario per rivi, jotta voimme tarkistella sitä.

```
fun writeToFile(filename: String, contents: List<String>) {
    val resourceDir = File(getResourcePath())
    val file = File(resourceDir, filename)
    // Unix new lines
    file.printWriter().use { out -> contents.forEach { line -> out.write("$line\n") } }
}
```

Esimerkkikoodi 3: Pakat kirjoitetaan tiedostoon.

Käyttämällä Linuxin wc-ohjelmaa (word count) -l (rivien määrä) ja -c (tavujen määrä argumenteilla voimme todentaa, että tiedostossa on 250 000 riviä ja se koostuu 40 000 000 tavusta [1]. Tiedoston koko on siis noin 39 MB.

Peli tarvitsee yhden, satunnaisesti arvotun, korttipakan jokaisen pelikierroksen aluksi, joten pelin palvelinsovelluksella pitää olla pääsy mihin tahansa skenaarioon. Helpoin ratkaisu on lukea tiedosto suoraan ohjelman muistiin listaan merkkijonoja. Muistiin luettuna tiedoston koko on kuitenkin huomattavasti enemmän kuin 39 MB.

2.2 Palvelinympäristöstä

Palvelinympäristö koostuu seuraavista komponenteista:

- Palvelinkoneen käyttöjärjestelmänä on Linux, tarkemmin Ubuntu, joka on yksi laajimmin käytetyistä Linux-jakeluista.

- Apache HTTP Server, joka toimii käänteisenä välityspalvelimena (reverse proxy). Sen tehtävänä on vastaanottaa ja käsitellä HTTP-pyyntöjä sekä ohjata ne edelleen Apache Tomcat -palvelimelle, jonka tehtävä on hallinnoida varsinaista sovelluslogiikkaa.

- Apache Tomcat on erityisesti Java Servlet -ohjelmia suorittava kontti. Pelien palvelinohjelmien koodi on kirjoitettu Javalla ja koottu WAR -tiedostoiksi, jotka Tomcat purkaa ja käynnistää. Tomcat suorittaa usean pelin palvelimia, ja Apache ohjaa palvelinkoneelle saapuvan pyynnön halutun pelin palvelinsovellukselle.

- Tietokantajärjestelmänä toimii PostgreSQL, joka on avoimena lähdekoodina jaettava relaatiotietokanta.

Pelien palvelinsovellukset ovat Javalla kirjoitettuja servlettejä. Servlet on Java-luokka, joka yleisesti ottaen käsittelee sille ohjattuja HTTP-pyyntöjä ja lähettää vastauksen asiakassovellukselle.

Palvelinsovellukset kootaan WAR-tiedostoiksi ja siirretään palvelinkoneelle, jossa Tomcat-palvelin käyttöönoton jälkeen ajaa niitä. Yhdellä palvelinkoneella on siis samanaikaisesti ajossa useamman pelin palvelinohjelmat Tomcatin yhteisessä JVM:ssä.

Tuotanto-, kehitys- ja testiympäristöt muodostuvat useiden palvelinkoneiden verkosta. Tuotantokoneita sijaitsee fyysisesti eri puolilla maailmaa, ja pelaajan sessio ohjataan automaattisesti sopivalle palvelimelle maantieteellisen sijainnin tai viiveen perusteella.

Yksittäisen pelin kohdalla skenaariotiedostosta koituva muistin käytön kasvu ei olisi ongelma, mutta mikäli samankaltaisia pelejä tehtäisiin tulevaisuudessa lisää, olisi muistinkäyttöä optimoitava. Koska pelit käyttävät samaa JVM:ää, haluttiin välttää tilanne, jossa pelien palvelinsovellukset joutuisivat kilpailemaan järjestelmän resursseista. Pahimmassa tapauksessa tämä voisi aiheuttaa virhetilan muistin loppumisen vuoksi, mikä toki huomattaisiin jo kehitysympäristössä.

3 Teoria

3.1 Tiedostoista

Tiedostot ovat peräkkäinen tavujoukko [2]. Yhdessä tavussa on kahdeksan bittiä. Käytännössä tämä tarkoittaa, että kaikki tiedostot tallennetaan levylle yhtenäisenä bittivirtana, oli kyseessä teksti, kuva tai äänitiedosto. Tavujen merkitys vaihtelee eri tiedostotyyppien välillä, esimerkiksi tekstitiedostoissa tavu voi edustaa kirjainta.

Aiemmassa pakanluomisesimerkissä teksti tallennettiin tiedostoon käyttäen Kotlinin File-luokan printWriter-oliota, jonka oletusmerkistö on UTF-8 [3]. UTF-8 on

muuttuvanpituinen merkistökoodaus, joka pystyy esittämään kaikki Unicode-merkit samalla säilyttäen samalla yhteensopivuuden vanhojen ASCII-pohjaisten järjestelmien kanssa. Ensimmäiset 128 ASCII-merkkiä (arvot 0-127) vievät yhden tavun verran tilaa levyltä, mutta monimutkaisemmat merkit, kuten ä ja ö, vievät enemmän [4]. Esimerkiksi iso H-kirjain on binäärimuodossa 01001000 ja numero kahdeksan on 00111000.

Koko opinnäytetyön aikana käytetään esimerkkinä aiemmin luotua 250 000 rivin skenaariotiedostoa, jossa jokainen skenaario koostuu yhdestä 160 merkin rivistä. Jokainen 52 kortin pakka on tallennettu tekstimuodossa, jossa kortit on erotettu toisistaan pilkuilla. Pakat on erotettu rivinvaihdolla. Jokaisella kortilla on vähintään kolme ja enintään neljä merkkiä: 1 maalle, 1-2 arvolle sekä yksi erotinmerkille. Numerossa 10 on kaksi merkkiä, ja se esiintyy pakassa neljä kertaa.

Tavuissa tiedoston koon voi laskea kertomalla rivien määrän pakan pituudella, eli 250 000 × 160 tavua, joka on 40 miljoonaa tavua. Tiedoston koon voi tarkistaa Linuxilla komennolla du (disk usage) -h (human readable) BM (block-size=M), joka ilmoittaa koon megatavuina. Tiedoston koko on ilmoitettu 1024 tavun potensseina, jolloin 1 MiB = 1024 KiB. [5.]

3.2 Java ja JVM

Java on yleiskäyttöinen, staattisesti tyypitetty ja korkean tason olio-ohjelmointikieli. Java on alustariippumaton kieli, joka on mahdollista siksi, että tyypillisesti Javalla kirjoitetut ohjelmat käännetään tavukoodiksi (bytecode). Tavukoodin suorittaa Java Virtual Machine (JVM), joka toimii välikerroksena ohjelman ja käyttöjärjestelmän välillä. JVM ei pelkästään suorita ohjelmakoodia, vaan se myös hallinnoi muistia, suorittaa roskienkeruuta (Garbage Collection) ja mahdollistaa ohjelman monisäikeisyyden. [6.]

Opinnäytetyön koodiesimerkit on kirjoitettu Kotlinilla, joka on Java-yhteensopiva ohjelmointikieli ja toimii samassa ekosysteemissä. Myös Kotlin käyttää oletuksena JVM:ää. Kotlin tarjoaa tiiviimpää ja ilmaisukykyisempää syntaksia Javaan

verrattuna, mutta käyttää samoja kirjastoja, koodirakenteita ja JVM:n ominaisuuksia, kuten JIT-kääntäjä, roskien keruuta ja heap-muistia. [7.]

Käynnistyessään JVM varaa tietyn määrän muistia ajettavan ohjelman käyttöön, jonka suuruus voi vaihdella JVM:n asetuksista ja ohjelman vaatimuksista riippuen. Ajonaikainen muisti JVM:ssä on jaettu kahteen pääalueeseen: heap (keko) ja stack (pino).

Heap on JVM:n muistialue, jota käytetään dynaamiseen muistiallokointiin, erityisesti olioiden luomiseen. Kun olio luodaan new-avainsanalla, se varataan heapissa. Jos sovellus ei enää tarvitse oliota, poistaa JVM:n roskienkerääjä sen virtuaalikoneen muistista. Heap on jaettu alue, joka on Java-sovelluksen kaikkien säikeiden käytettävissä. Heap voi kasvaa tai kutistua dynaamisesti sovelluksen muistivaatimusten mukaan. [8; 9; 10.]

JVM:n stack taas on pienempi muistialue, jota käytetään metodikutsujen, paikallisten muuttujien ja viitteiden hallintaan. Jokaisella säikeellä on oma stack, jonka muisti on organisoitu kehyksiksi (frame), joista kukin vastaa metodikutsua. Stack noudattaa LIFO (Last In, First Out) -tietorakennetta, eli viimeisenä lisätty metodi käsitellään ensin. Kun metodin suorittaminen päättyy, kehys poistetaan ja sen varaama muisti vapautuu. Stackin koko on rajoitettu, mikä voi rekursiivisten metodien tai liiallisten kutsujen vuoksi johtaa StackOverflowError-virheeseen. [8; 9; 10.]

Pelin palvelinsovellus tarvitsee satunnaisesti pääsyn mihin tahansa skenaarioon, mikä alkuperäisessä prototyypissä aiheutti sen, että ajon aikana skenaarioiden varaama muisti ei missään vaiheessa voi vapautua.

3.3 Olioiden muistinkäyttö

Aiemmin todettiin skenaariotiedoston koon olevan 39 MB levyllä. Alkuperäisessä versiossa tiedoston lukeminen suoraan Javan muistiin listaan merkkijonoja

käyttää kuitenkin huomattavasti enemmän muistia verrattuna tiedoston kokoon levyllä. Tämä johtuu tavasta, jolla Java pitää merkkijonoja muistissa.

Merkkijonon toteutus, eli String-luokka, käytti Java 8:aan saakka listaa merkkejä, char-listaa, jossa kukin merkki vei kaksi tavua muistia, koska Java käytti UTF-16-koodaustapaa [11]. Javan versiosta 9 eteenpäin on otettu käyttöön compact strings -tekniikka, jossa merkkien sijaan käytetään listaa tavuja. String-luokka päättelee ajon aikana tarvitsevatko merkkijonon merkit yhden tavun vai kaksi tavua riippuen siitä, koostuuko merkkijono täysin Latin-1-standardiin (ISO 8859-1) kuuluvista merkeistä vai ei. [12; 13.]

Tuotantokäytössä oleva pelipalvelinten koodi on kirjoitettu Java 8:lla, jossa ohjelman muistiin luettu tieto vie yli kaksi kertaa enemmän muistia kuin levyllä, mikä johtuu Javan String-luokan kenttien viemistä tavuista [11].

3.4 Tiedostojen lukemisesta

Kun Javalla avataan tiedosto, ohjelma pyytää käyttöjärjestelmää avaamaan tiedoston määritetyn polun perusteella. Tämä tapahtuu käyttämällä java.io.File- tai java.nio.file.Path-olioita. Käyttöjärjestelmä tarkistaa, onko prosessille riittävät käyttöoikeudet tiedostoon. Mikäli on, käyttöjärjestelmä palauttaa prosessille tiedostokahvan (file descriptor), joka toimii viittauksena avattuun tiedostoon ja mahdollistaa sen käsittelyn ohjelmallisesti. [14.]

Java tarjoaa useita mekanismeja tiedoston lukemiseen. Perinteisesti java.io-paketin luokat, kuten FileInputStream, BufferedReader tai Scanner, tarjoavat mahdollisuuden lukea tiedostoa merkki- tai tavuvirtoina. Nämä luokat hyödyntävät käyttöjärjestelmän tiedostonkäsittelyrajapintoja, jotka voivat kutsua matalan tason järjestelmäkutsuja, kuten read(), tietojen hakemiseksi levyltä.

Moderneissa POSIX-yhteensopivissa käyttöjärjestelmissä read()-järjestelmäkutsu kopioi tiedostosta kutsujan spesifioiman määrän tavuja levyltä argumenttina annettuun, prosessin hallinnoimaan, puskuriin (buffer) RAM -muistissa. [15.]

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Esimerkkikoodi 4: Systeemikutsun argumentit: File descriptor, Pointer to buffer, Number of bytes

Puskuri on yleisesti ottaen tilapäinen varasto, jossa yksi komponentti syöttää dataa toiseen. Tiedoston lukemisessa puskurointi vähentää järjestelmäkutsujen määrää ja parantaa suorituskykyä lataamalla suurempia tietolohkoja kerralla. Kun Java lukee puskurista tavuja tai merkkejä, data puretaan tiedostokoodauksen, esimerkiksi UTF-8, mukaan String- tai char[]-muotoon.

Kun suorittava ohjelma on lopettanut tiedoston käyttämisen, tiedostokahva tulee vapauttaa kutsumalla close()-metodia, joka informoi käyttöjärjestelmää siitä, ettei prosessi enää tarvitse kyseistä tiedostoa, jolloin käyttöjärjestelmä voi vapauttaa resurssin. [16.]

Java-ohjelman suorituksen aikana luetut tiedot säilyvät JVM:n hallinnoimassa muistissa olioina tai muina datatyyppeinä, kunnes ne eivät enää ole viittauksien kohteena, jolloin JVM:n roskienkeruu tunnistaa käyttämättömät objektit ja vapauttaa niihin liittyvät resurssit automaattisesti. [10.]

3.5 Tiedostojen lukeminen satunnaisesti

Kuten aiemmin todettiin, ovat tiedostot pohjimmiltaan lineaarinen joukko tavuja. Jos ohjelma tarvitsee tiedostosta sen hetkiseen käyttöön vain pienen osan kerrallaan, voidaan siirtyä suoraan haluttuun kohtaan tavulistaa ja lukea siitä alkaen tarvittava määrä tietoa ohjelman käyttöön. Tässä tapauksessa muistin käytön kannalta optimaalisempaa on se, että pelikierroksen alussa ohjelma lukee tiedostosta vain yhden rivin, eli yhden skenaarion, sen sijaan, että kaikki skenaariot ovat samanaikaisesti ohjelman muistissa.

Javan IO -paketin RandomAccessFile-luokkaa käyttämällä voidaan lukea ja kirjoittaa mielivaltaisiin kohtiin tiedostoa ilman, että ohjelman tarvitsee käydä koko tiedostoa ensin läpi. Tämä eroaa perättäiseen tiedostojen käsittelyyn tarkoite-

tuista luokista, kuten FileInputStream tai FileReader, jotka etenevät tiedostossa alusta loppuun.

RandomAccessFile sisältään sisäisen osoittimen (file pointer), joka pitää kirjaa luokan sen hetkisestä luku- tai kirjoituspaikasta tiedoston tavulistassa. Osoitinta voidaan siirtää tiedostossa suoraan haluttuun kohtaan seek(long pos) -metodilla, jolloin lukiessaan luokka aloittaa valitusta kohdasta. [17.]

RandomAccessFile soveltuu opinnäytetyössä kuvailtuun käyttötarkoitukseen hyvin, sillä tiedostossa esiintyvien skenaarioiden pituus on aina sama. Täten kun halutaan lukea yksittäinen skenaario tiedostosta, esimerkiksi viides pakka, joka sijaitsee rivillä viisi, voidaan laskea kohta, johon osoitin tulee siirtää. Osoittimen uusi positio on tällöin yksittäisen skenaarion koko kerrottuna edeltävien rivien määrällä.

$$offset = rivin\ pituus \times rivin\ indeksi = 160 \times 4 = 640$$

Tämä tarkoittaa sitä, että kutsumalla seek(640) ohjelma voi hypätä suoraan viidennen skenaarion alkuun ja aloittaa lukemisen siitä.

RandomAccessFile-luokkaa käyttäessä on huomioitava, että mikäli useampi säie käsittelee samaa instanssia oliosta, ei osoittimen siirto ole säieturvallista. Mikäli toinen säie kutsuu seek(long pos) -metodia, samalla kun toinen suorittaa lukutoimenpiteen, voi osoittimen sijainti muuttua arvaamattomasti, joka voi johtaa virheellisiin lukutuloksiin. Tästä syystä lukuoperaatio pitää tehdä synkronoidusti tai säikeen tulee käyttää omaa RandomAccessFile-instanssiaan.

RandomAccessFile:llä voi vahingossa lukea tiedoston ulkopuolelle, mikä Javassa aiheuttaa EOFException-poikkeuksen (End Of File). Tämä voidaan välttää pitämällä huolta siitä, ettei osoittimen positio ja luettavan alueen koko mene yli tiedoston tavujen määrästä.

Huonona puolena RandomAccessFilessä on tehtyjen systeemikutsujen määrä. Yksi lukuoperaatio vaatii vähintään kahta käyttöjärjestelmän metodia: seek ja read. Lisäksi metodi readFully(buffer) aiheuttaa datan siirtämisen kahteen kertaan. Ensimmäinen tapahtuu kun käyttöjärjestelmän kernel lukee tiedoston sisällön tilapäiseen puskuriin. Toinen vaihe tapahtuu, kun tiedot siirretään Javan heap-muistiin. [18.]

Opinnäytetyön esimerkissä tiedoston skenaariot ovat kaikki saman kokoisia. Mikäli ne olisivat eripituisia, tulee kunkin tietueen alkupositiot tietää etukäteen. Tällöin positiot voi vaihtoehtoisesti pitää muistissa tai tallennettuna erilliseen indeksitiedostoon.

4 Ratkaisuvaihtoehdot

4.1 Yleistä

Määritellään pelin käyttöön skenaarioita hallinnoiva rajapinta, josta peli hakee yhden skenaarion jokaisen kierroksen alussa. Skenaarioita voidaan hakea satunnaisesti tai skenaarion järjestysluvulla alkuperäisessä tiedostossa. Näiden lisäksi lukijalla on metodi, jolla voidaan vapauttaa lukijan käyttämät resurssit.

```
interface ScenarioReader {
    fun getRandomScenario(): String

    fun getScenarioById(index: Int): String

    fun close()
}
```

Esimerkkikoodi 5: Rajapinta tiedostolukijalle joka sisältää kaksi vaihtoehtoista metodia skenaarioiden lukuun.

Kaikki testattavat ratkaisuvaihtoehdot sisältävät siis samat metodit, mutta eri toteutuksilla. Tämä mahdollistaa sen, että testikoodi pysyy identtisenä, mutta testattava asia, eli ScenarioReader -rajapintaa toteuttava luokka, vaihtuu.

Eri implementaatioista tullaan vertailemaan suorituskykyä ja muistinkäyttöä.

4.2 Alkuperäinen toteutus

Ensimmäinen versio oli kaikessa yksinkertaisuudessaan luokka, joka luki tiedoston kaikki rivit kerralla muistiin listaan merkkijonoja. Tämän version suurin hyöty on nopeus. Satunnaisesti arvotun skenaarion hakuoperaatio listasta on O(1) [19]. Tämän ansiosta tiedostoa tarvitsee lukea vain kerran.

Kuten aiemmin todettiin, ohjelman muistiin ladatut merkkijonot vievät huomattavasti enemmän muistia kuin tiedoston koko levyllä. Eikä tulevaisuutta ajatellen

haluttu lisätä pelejä, jotka tuovat tämänkaltaista turhaa kuormaa palvelinympäristöön.

4.3 Välimuisti

Välimuisti (cache) olisi tilapäinen säiliö skenaarioiden pitämiseksi muistissa. Ajatuksena olisi välttää suhteellisen kallis tiedostosta lukuoperaatio pitämällä osaa skenaarioista muistissa [20]. Ongelma tässä lähestymistavassa on se, että tiedostosta halutaan yksittäinen skenaario kierroksen alussa satunnaisesti tietoa, joten todennäköisyys sille, että välimuisti sisältää tarvittavan tiedon, on pieni.

4.4 Tietokanta

Tiedoston sijaan peli voisi käyttää tietokantaa skenaarioille. Tietokannassa olisi pelikohtainen taulu, jossa tässä tapauksessa olisi kaksi saraketta: skenaarion indeksit ja niiden sisällöt. Indeksi olisi automaattisesti numeroitu taulun pääavain (primary key). Skenaarion sisältö olisi tekstikenttä pakan korteille. PostgreSQL-tietokannan alustus voidaan tehdä seuraavalla tavalla:

```
CREATE TABLE scenarios
(
  id   SERIAL PRIMARY KEY,
  deck TEXT
);
```

Esimerkkikoodi 6: Luodaan tietokantaan taulu ”scenarios”, jossa on kaksi saraketta: automaattisesti numeroitu taulun pääavain (primary key) ”id” ja sitä vastaava tekstisarake ”deck”, johon skenaarioiden sisältö asetetaan.

Tiedoston sisältö voidaan siirtää tietokantaan komennolla ”COPY scenarios (deck) FROM ’polku_tiedostoon’;”. Komento kopioi annetussa polussa sijaitsevan skenaariotiedoston sisällön scenarios-taulun deck-sarakkeeseen [21]. Taulun id-sarake on määritelty kasvavaksi pääavaimeksi, joka alkaa oletuksena numerosta yksi ja kasvaa jokaisella uudella rivillä yhdellä. [22.]

Yksittäisen skenaarion voi hakea tietokannasta komennolla ”SELECT deck FROM scenarios WHERE id = 1;”. Lisäämällä komennon alkuun ”EXPLAIN ANALYZE” voidaan tarkistaa tietokantakyselyyn kuluva aika tietokannan sisällä.

```
Index Scan using scenarios_pkey on scenarios
(cost=0.42..8.44 rows=1 width=163) (actual
time=0.067..0.074 rows=1 loops=1)

Index Cond: (id = 250000)

 Planning Time: 0.348 ms

 Execution Time: 0.116 ms
```

Esimerkkikoodi 7: Tietokantakyselyn ”EXPLAIN ANALYZE SELECT deck FROM scenarios WHERE id = 1;” antama tulos.

EXPLAIN ANALYZE -komennoilla saadaan suuntaa-antavat luvut ”Planning Time” ja ”Execution Time”. Planning Time on aika, joka kuluu kyselyn käsittelyyn ennen sen suorittamista. Siihen kuluu tietokantakyselyn jäsentämistä, syntaksin analysointi ja kyselyn suunnittelu. Execution Time on aika, joka kuluu itse kyselyn suorittamiseen ja tulosten palauttamiseen. Näiden kahden luvun summa on se aika, joka kuluu kyselyn suorittamiseen. [23.]

Koska skenaarioita haetaan taulusta pääavaimella, voi tietokanta hyödyntää tehokasta indeksihakua, jolloin tietokantakysely on äärimmäisen nopea [24]. Esimerkkihaussa kesti alle yksi millisekunti.

Peliohjelman puolella tietokantaa käyttävä ScenarioReader-rajapintaa toteuttava luokka tekee tietokantakyselyjä ohjelmallisesti suorittaen valmisteltuja kyselyitä (prepared statement). Tietokantakyselyjä tehtäessä hyödynnetään HikariCP-kirjastoa, joka on kevyt ja suorituskykyinen kirjasto tietokantayhteyksien hallintaan. Kirjaston avulla ohjelma voi ylläpitää joukkoa valmiiksi avattuja yhteyksiä, mikä vähentää tarvetta avata ja sulkea yhteyksiä jokaisen kyselyn yh-

teydessä. Yhteyspoolin merkitys korostuu, kun tietokantaa käyttää samanaikaisesti usea säie. [25.]

4.5 RandomAccessFile

Satunnaisesti tiedostoa lukeva luokka toimii laskemalla, arvotun indeksin perusteella kohdan, josta pakka alkaa tiedostossa, lukee siitä vakiomäärän tavuja puskuriin ja muuttaa ne merkkijonoksi. Tämä on mahdollista, koska jokainen skenaario on yhtä pitkä.

```
open class RandomAccessReader(filename: String) : ScenarioReader {
   private val scenarioFile: RandomAccessFile =
RandomAccessFile(FileUtils.getResourceFile(filename), "r")
   protected val lineSize = 160
   private val scenarioCount = scenarioFile.length().toInt() / lineSize

   override fun getScenarioById(index: Int): String {
      synchronized(this) {
         val position = index.toLong() * lineSize
         scenarioFile.seek(position)
         val buffer = ByteArray(lineSize)
         scenarioFile.readFully(buffer)
         return String(buffer, Charsets.UTF_8).trim()
      }
   }
}
```

Esimerkkikoodi 8: Valitaan satunnainen rivinnumero. Lasketaan rivin alkupositio ja siirretään RandomAccessFilen osoitin sinne. Alustetaan yhden skenaarion kokoinen tyhjä lista tavuja ja luetaan se täyteen.

RandomAccessFile ei ole säieturvallinen, joten osoittimen siirto ja tiedostosta luku on pidettävä synkronoituna. Synkronointi varmistaa, että vain yksi säie kerrallaan voi suorittaa kyseistä kohtaa ohjelmasta. [26.]

Synkronointi kuitenkin hidastaa ohjelman suoritusta estämällä muita säkeistä lukemasta tiedostoa. Tämä voidaan ratkaista käyttämällä tiedostopoolia, jota hallitaan BlockingQueue-tietorakenteella. Tiedostopoolin avulla säikeet voivat käyttää erillisiä tiedostoinstansseja skenaarioiden lukemiseen.

```
class PooledRandomAccessReader(filename: String) : RandomAccessReader(filename) {
    private val pool: BlockingQueue<RandomAccessFile>

    init {
        val poolSize = Runtime.getRuntime().availableProcessors()
        pool = ArrayBlockingQueue(poolSize)
        repeat(poolSize) {
            pool.put(RandomAccessFile(FileUtils.getResourceFile(filename), "r"))
        }
    }

    override fun getScenarioById(index: Int): String {
        val raf = pool.take()
        try {
            val buffer = ByteArray(lineSize)
            val position = index.toLong() * lineSize
            raf.seek(position)
            raf.read(buffer)
            return String(buffer, Charsets.UTF_8).trim()
        } finally {
            pool.put(raf)
        }
    }

    override fun close() {
        super.close()
        pool.forEach { it.close() }
    }

    // getRandomScenario
}
```

Esimerkkikoodi 9: RandomAccessReader -luokkaa laajentava ja BlockingQueue -tekniikkaa hyödyntävä lukija.

Kun säie tarvitsee skenaariota, ottaa lukija yhden tiedostoinstanssin poolista säikeen käyttöön. Lukuoperaation päätteeksi lukija palauttaa tiedostoinstanssin takaisin pooliin seuraavaa säiettä varten.

4.6 FileChannel

FileChannel on osa Javan NIO-pakettia (New Input/Output), joka on suunniteltu tehokkaaseen ja suorituskykyiseen tiedostojen käsittelyyn. Yksi FileChanneliin merkittävimmistä eduista perinteiseen I/O-malliin verrattuna on sen kyky käyttää

puskureita JVM:n heap:n ulkopuolella. Perinteisessä Java I/O:ssa data luetaan ensin levyltä käyttöjärjestelmän ytimeen (kernel) kuuluvaan puskuriin ja sitten käyttäjätilan puskuriin (user-space buffer), jota Java-sovellus käsittelee. Tämä kaksivaiheinen prosessi aiheuttaa tarpeettomia muistinsiirtoja ja ylimääräistä kuormitusta. FileChannel sen sijaan voi lukea suoraan käyttöjärjestelmän varaamasta puskurista. [18; 27.]

FileChannelia hyödyntävä lukija toimii hyvin samankaltaisesti, mitä aiemmin esitelty RandomAccessReader, mutta suoriutuu tehokkaammin lukuoperaatioista.

```
class FileChannelReader(filename: String) : ScenarioReader {
    private val lineSize = 160
    private val channel = FileChannel.open(FileUtils.getResourceFilePath(filename),
StandardOpenOption.READ)
    private val scenarioCount = channel.size().toInt() / lineSize

    private val threadLocalBuffer =
        ThreadLocal.withInitial { ByteBuffer.allocateDirect(lineSize) }

    override fun getScenarioById(index: Int): String {
        val buffer = threadLocalBuffer.get()
        buffer.clear()
        val position = index * lineSize.toLong()
        channel.read(buffer, position)
        buffer.flip()
        return StandardCharsets.UTF_8.decode(buffer).toString().trim()
    }
    // close
    // getRandomScenario
}
```

Esimerkkikoodi 10: FileChannel-luokkaa käyttävä lukija.

Luokka käyttää ThreadLocal-luokkaa instantioimaan jokaiselle säikeelle omat puskurit, mikä estää tietojen ylikirjoittamisen sekä vähentää tarvetta synkronoinnille.

Skenaariota lukiessa puskuri nollataan ja täytetään datalla. Koska puskuri on suora puskuri (DirectBuffer), tapahtuu luku suoraan ytimen muistista. Jotta puskurin tavut voidaan dekoodata merkkijonoksi, kutsutaan sen flip-metodia.

4.7 Memory-Mapped Files

FileChannel-luokan map-metodilla tiedosto voidaan kartoittaa suoraan MappedByteBuffer-puskuriin, joka sijaitsee JVM:n ulkopuolella. Tämä tarkoittaa, että käyttöjärjestelmä käsittelee tiedostoa muistissa olevana alueena, ja ohjelma voi lukea ja kirjoittaa siihen aivan kuin se olisi tavallinen RAM-muistissa sijaitseva tietorakenne. Suurin etu tässä menetelmässä on se, että perinteiset tiedoston lukemiseen liittyvät systeemikutsut voidaan ohittaa lähes kokonaan. Tällöin kontekstinvaihdot vähenevät ja suorituskyky paranee merkittävästi. [18.]

Käyttöjärjestelmä luo muistiin kartoitetulle tiedostolle taulun, joka sisältää tiedoston virtuaalimuistiin sijoitetut osoitteet. Linuxilla muistiin kartoitetut alueet voi tarkistaa komennolla ”cat /proc/<pid>/maps”.

Menetelmä ei kuitenkaan ole täysin riskitön. Koska käyttöjärjestelmä hallitsee sen lataamista RAM-muistiin, josta saattaa aiheutua viiveitä, jos järjestelmä alkaa aktiivisesti sivuttaa dataa levyltä takaisin muistiin.

Koska MappedByteBuffer sijaitsee off-heap-muistissa, JVM:n roskienkeruu ei hallitse sen käyttöä suoraan, eikä ohjelma välttämättä vapauta sitä tehokkaasti. Liian suuret kartoitukset voivat johtaa suorituskykyongelmiin tai jopa kaataa prosessin, jos järjestelmä ei pysty allokoimaan tarvittavia muistialueita. Kuten aiemmin todettiin, yhdellä palvelinkoneella on käynnissä useamman pelin palvelinsovellukset, eikä ylimääräistä kuormaa haluttu lisätä. MappedByteBuffer ei siis välttämättä ole paras ratkaisu.

5 Vertailu

5.1 Suorituskyvyn analysoinnista

Vaihtoehtoja vertaillaan muistinkäytöstä ja lukunopeudesta suhteen. Tätä varten luodaan testejä lukijoiden suorituskyvyn mittaamiseen. Kuten aiemmin todettiin, kaikki lukijat toteuttavat ScenarioReader-rajapintaa, joten testit pysyvät identtisinä per lukija. Käyttämällä Junit-kirjaston parametrisoituja testejä voidaan yksittäinen testi ajaa joukolle annettuja argumentteja [28], tässä tapauksessa ScenarioReader-luokkien nimiä.

```
class FileReaderTest {
    private val FILENAME = "decks.csv"

    @ParameterizedTest
    @ValueSource(strings = ["InMemory", "Database", "FileChannel",  "MemoryMapped",
"PooledRandomAccess", "RandomAccess"])
    fun `generic reader test case`(readerType: String) {
        val reader = getReader(readerType)
        // Test code
        reader.close()
    }

    private fun getReader(readerType: String): ScenarioReader {
        return when (readerType) {
            "InMemory" -> InMemoryReader(FILENAME)
            "Database" -> DatabaseReader()
            "FileChannel" -> FileChannelReader(FILENAME)
            "MemoryMapped" -> MemoryMappedReader(FILENAME)
            "PooledRandomAccess" -> PooledRandomAccessReader(FILENAME)
            "RandomAccess" -> RandomAccessReader(FILENAME)
            else -> throw IllegalArgumentException("Unknown reader: $readerType")
        }
    }
}
```

Esimerkkikoodi 11: Testi, joka suoritetaan jokaiselle annetulle lukijalle.

@ParameterizedTest-annotaation avulla on siis mahdollista luoda lukijaolio jokaisen testin alussa. Testin lopuksi lukija suljetaan, resurssit vapautetaan ja mittaustulokset tallennetaan lukijan nimen mukaisesti.

5.2 Suorituskyky

Testataan eri SkenarioReader-implementaatioiden suorituskykyä tekemällä yhteensä miljoona satunnaista skenaarionlukuoperaatioita rinnakkaisesti säiettä käyttäen. Testi on parametrisoitu siten, että se voidaan toistaa jokaisella annetulla lukijatyypillä. Suorituksen jälkeen mittaustulokset tallennetaan ja lukija suljetaan.

```
@ParameterizedTest
@ValueSource(strings = ["InMemory", "Database", "RandomAccess",
"PooledRandomAccess", "FileChannel", "MemoryMapped"])
fun `test threaded random reads`(readerType: String) {
    val header = "reader,ms"
    val reader = getReader(readerType)
    val executor = Executors.newFixedThreadPool(threadCount)
    val tasks = List(threadCount) {
        Callable {
            repeat(times / threadCount) {
                val scenario = reader.getRandomScenario()
                scenario.hashCode()
            }
        }
    }
    val elapsedTime = measureTimeMillis {
        executor.invokeAll(tasks)
    }
    executor.shutdown()
    executor.awaitTermination(1, TimeUnit.MINUTES)
    testResults.computeIfAbsent(header) { mutableListOf() }.add(readerType to elapsedTime)
    reader.close()
}
```

Esimerkkikoodi 12: Testi, joka mittaa lukijoiden suorituskykyä monisäikeisesti.

Testin alussa alustetaan säiepooli Executors-luokan newFixedThreadPool(threadCount)-metodia. Kullekin säikeelle luodaan lista Callable-tehtäviä, joissa suoritetaan osa miljoonasta lukuoperaatiosta.

Suoritusajan mittaamiseen käytetään Kotlinin measureTimeMillis-funktiota, joka laskee ajan hetkestä, jolloin kaikki tehtävät käynnistetään invokeAll-metodilla,

siihen hetkeen, jolloin ne kaikki ovat valmistuneet [29]. Metodi invokeAll() käynnistää annetut tehtävät rinnakkain ja odottaa niiden valmistumista ennen ohjelman suorituksen jatkamista. Taulukko 1 näyttää, kauanko aikaa eri lukijtatyypeillä kului miljoonan skenaarion satunnaiseen lukemiseen usealla säikeellä.

Taulukko 1: Miljoonaan skenaarion satunnaiseen monisäikeiseen lukemiseen kulunut aika

Lukija	**Kulunut aika (ms)**
Database	23222
FileChannel	784
InMemory	144
MemoryMapped	250
PooledRandomAccess	1132
RandomAccess	3899

JUnit-testin avulla saadaan selville, että kaikki skenaariot muistissa pitävä lukija on kaikista nopein. Tämä johtuu siitä, että lukemisen sijaan voidaan skenaario hakea suoraan ohjelman muistista ilman erillisiä I/O-operaatioita. MemoryMappedReader pärjää myös todella hyvin, koska se hyödyntää käyttöjärjestelmän muistiinkartoitettua tiedostoa, jolloin tiedostoa voidaan lukea lähes yhtä nopeasti kuin tavallista muistia. FileChannel pärjää vertailussa myös hyvin, mutta se jää jälkeen suuremmasta systeemikutsujen määrästä johtuen.

RandomAccessReader tuo esiin synkronoinnin vaikutuksen suorituskykyyn, sen perusversio on selvästi hitaampi, kun taas tiedostopoolia käyttävä versio on yli kolme kertaa nopeampi.

Tietokantaa käyttävä lukija taas on ylivoimaisesti hitain. Tämä johtuu siitä, että skenaarion haku vaatii erillisen tietokantakyselyn, mikä on huomattavasti raskaampi operaatio verrattuna tiedoston lukemiseen.

Edellä kuvatun JUnit-testin lisäksi tehdään suorituskykytesti JMH (Java Mirco-benchmark Harness) -kehyksellä. JMH ottaa huomioon JVM:n optimoinnit, kuten JIT-kääntämisen, roskienkeruun ja suorittimen optimoinnit, jotka muuten voisivat vääristää mittaustuloksia. [30.]

Ennen varsinaista mittausta JMH suorittaa lämmittelyiteraatiota, joita tehdään oletuksena kymmenen. Lämmittelyvaiheen tarkoituksena on varmistaa, että JIT-kääntäjä ehtii optimoida koodin suorituksen ennen kuin mittausten tuloksia aletaan kerätä. JMH tarjoaa useita asetuksia testien ajamiseen.

```
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Benchmark)
@Fork(2)
@Warmup(iterations = 3, time = 10, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)
@Threads(8)
open class PerformanceTest {
    // Test code
}
```

Esimerkkikoodi 13: Testiluokan asetukset.

Alla selitys kunkin käytetyn asetuksen toiminnasta [31]

- @BenchmarkMode(Mode.Throughput) määrittelee mittaustavaksi suoritusnopeuden, eli montako operaatiota voidaan suorittaa tietyssä ajassa.
- @OutputTimeUnit(TimeUnit.MILLISECONDS) on mittauksessa käytettävä yksikkö, tässä tapauksessa millisekunti.
- @State(Scope.Benchmark) säätää testien tilan säilytyksestä. Scope.Benchmark tarkoittaa, että tilan elinikä on koko benchmarkin kesto.
- @Fork(2) määrittää, että testit suoritetaan kahdessa erillisessä JVM-instanssissa.
- @Warmup-annotaatiolla säädetään, montako lämmittelykierrosta tehdään ennen varsinaista testaamista ja kauanko yksi kierros kestää.

- @Measurement-annotaatiolla asetetaan mittausten määrä ja kauanko kuhunkin mittaukseen käytetään aikaa.
- @Threads(8) on käytettävien säikeiden määrä.

Kuten aiemmassa yksikkötestissä, myös JMH-testit voidaan kirjoittaa parametrisoituna, jolloin testiajot suoritetaan jokaiselle parametrina annetulle lukijalle.

```
// Annotations...
open class PerformanceTest {
    @Param
    lateinit var readerType: ReaderType
    private lateinit var reader: ScenarioReader

    companion object {
        private const val FILENAME = "decks.csv"
        private const val SCENARIO_COUNT = 250_000
    }

    @Setup(Level.Iteration)
    fun setup() {
        reader = readerType.createReader()
    }

    @TearDown(Level.Iteration)
    fun tearDown() {
        reader.close()
    }

    @Benchmark
    fun randomRead(blackhole: Blackhole) {
        val id = ThreadLocalRandom.current().nextInt(SCENARIO_COUNT)
        blackhole.consume(reader.getScenarioById(id))
    }

    enum class ReaderType {
        IN_MEMORY {
            override fun createReader(): ScenarioReader = InMemoryReader(FILENAME)
        },
        // Rest of the readers...
        abstract fun createReader(): ScenarioReader
    }
}
```

Esimerkkikoodi 14: Parametrisoitu JMH-testi.

@Param-annotaatio mahdollistaa sen, että jokainen ReaderType-enumin sisältämä lukija testataan. Jokaisen testitapauksen aluksi suoritetaan @Setup-metodi, joka alustaa sillä hetkellä testattavan lukijan. Kun lukijan testaus on päättynyt, suljetaan se @TearDown-metodissa. Alustus- ja lopetusmetodeille on määritetty suorittamaan per iteraatio.

Itse testattava asia, tässä tapauksessa yksittäisen skenaariohaku, on merkitty @Benchmark-annotaatiolla, jota toistetaan koko lämmittelyyn ja testitapauksien kestoon varatun ajan verran.

Kun halutaan mitata pelkästään skenaarion lukemiseen kuluvaa aikaa, eikä varsinaisesti tehdä itse skenaariolla mitään, saattaa JVM:n JIT-kääntäjä havaita tämän ja optimoida koko operaation pois. Ongelma voidaan välttää käyttämällä JMH:n Blackhole-luokkaa, joka mustanaukon kaltaisesti nielaisee sille annetut arvot ja estää JIT-kääntäjää optimoimasta niitä turhina. [32.]

Testiajon jälkeen JMH kirjoittaa tulokset tiedostoon sekä tulostaa ne konsoliin (kuva 1).

Benchmark	(readerType)	Mode	Cnt	Score	Error	Units
PerformanceTest.randomRead	IN_MEMORY	thrpt	10	63573.604 ±	7206.121	ops/ms
PerformanceTest.randomRead	DATABASE	thrpt	10	51.688 ±	5.671	ops/ms
PerformanceTest.randomRead	RANDOM_ACCESS	thrpt	10	226.770 ±	13.814	ops/ms
PerformanceTest.randomRead	POOLED_RANDOM_ACCESS	thrpt	10	805.929 ±	43.444	ops/ms
PerformanceTest.randomRead	FILE_CHANNEL	thrpt	10	1731.325 ±	787.486	ops/ms
PerformanceTest.randomRead	MEMORY_MAPPED	thrpt	10	8669.568 ±	298.796	ops/ms

Kuva 1: JMH-testin mittaustulokset.

Kuvan 1 sisältämät testitulokset ovat linjassa aiemman yksikkötestin kanssa ja voidaan todeta samat, merkittävät erot eri lukijoiden rinnakkaisessa lukunopeudessa. Mittaukset ovat muodossa operaatiota per millisekunti. Tulokset ovat samansuuntaisia kuin aiemassa yksikkötestissä, mutta luotettavampia.

5.3 Muistinkäytön profilointi

Yksinkertainen tapa seurata ohjelman käyttämää muistia on käyttää MemoryMXBean-luokkaa, joka on osa Java Management Extensions (JMX) -pakettia ja tarjoaa tavan tarkastella JVM:n heap-muistin käyttöä. [33.]

```
object MemoryUtils {
    fun getHeapUsageMB(): Long {
        val memoryMxBean = ManagementFactory.getMemoryMXBean()
        val usedHeap = memoryMxBean.heapMemoryUsage.used
        return usedHeap / (1024 * 1024)
    }
}
```

Esimerkkikoodi 15: Yksinkertainen apumetodi, joka palauttaa JVM:n varaaman heap-muistin megatavuissa.

Käyttämällä aiemmin kuvailtuja parametrisoituja testejä voidaan kätevästi verrata heap-muistin käyttöä ennen ja jälkeen erityyppisten lukijainstanssien luomista. Testin tulokset on listattu taulukossa 2. Tulokset kuitenkin sisältävät hieman hajontaa, koska ne toimivat käynnissä olevan JVM:n sisällä, jolloin kunkin hetkiseen varatun muistin määrään vaikuttaa usea tekijä. Roskienkeruu saattaa vapauttaa muistia arvaamattomasti, muistialueita voidaan käyttää uudelleen ja JIT-kääntäjän optimoinnit voivat vaikuttaa olioiden allokointiin.

Taulukko 2: Eri lukijoiden keskimääräinen heap-muistin käyttö

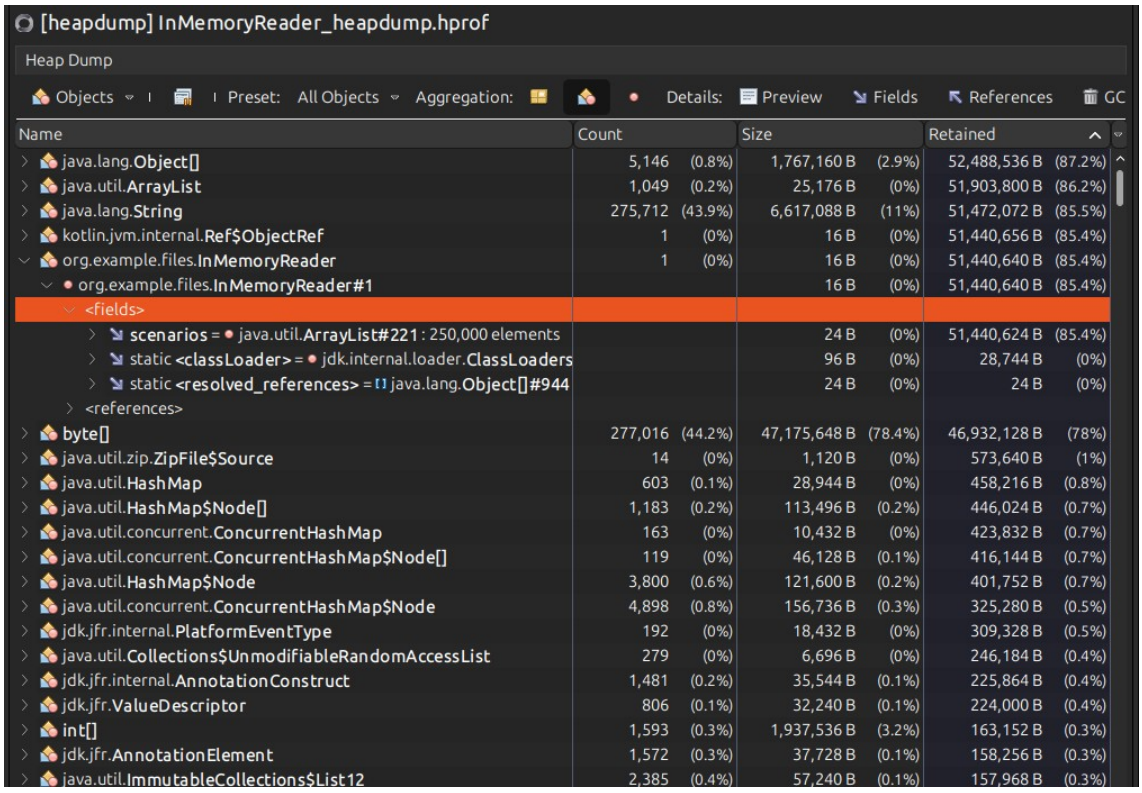
Lukija	**Heap muistin käyttö (MB)**
Database	8.28
FileChannel	7.63
InMemory	65.83
MemoryMapped	7.63
PooledRandomAccess	7.63
RandomAccess	7.63

Yksityiskohtaisemman tiedon saamiseksi voidaan ottaa tilannekuva Java-prosessin muistista (heap dump). Se sisältää kaikki ottamishetkellä olemassa ole-

vat oliot, niiden kentät, primitiiviarvot ja viittaukset. Tämän lisäksi siitä löytyy kaikki luokat staattisine kenttineen. Heap dumpin avulla voidaan analysoida sovelluksen muistinkäyttöä, tunnistaa mahdollisia muistivuotoja ja ymmärtää ohjelman rakennetta. [34.]

Linuxissa aktiiviset JVM-prosessit voidaan listata komennolla jps -l. Komento näyttää käynnissä olevien Java-sovellusten PID:t ja pääluokkien pakettinimet [35]. Ajossa olevalle JVM:lle voidaan lähettää jcmd-komentorivityökalulla diagnostiikkapyyntöjä. Komennolla jcmd <PID> GC.heap_dump <tiedostopolku.hprof> työkalu luo HPROF-muotoisen heap dump -tiedoston. [36.]

VisualVM on työkalu Java-pohjaisten ohjelmien monitorointiin ja profilointiin. Ohjelmalla avulla voidaan seurata muun muassa prosessorin käyttöä, roskien keruuta, muistinkäyttöä ja säikeiden toimintaa reaaliajassa. Kuva 2 sisältää VisualVM-ohjelman heap dump -tilannekuvan ohjelman muistinkäytöstä. [37.]

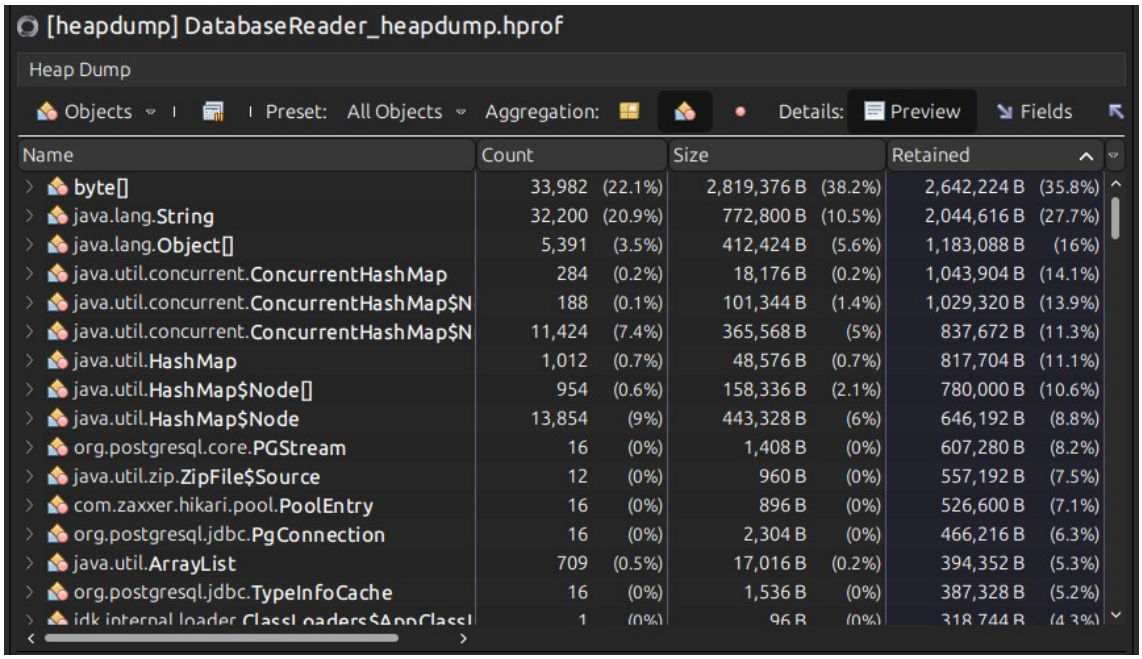


Name	Count		Size		Retained	
java.lang.Object[]	5,146	(0.8%)	1,767,160 B	(2.9%)	52,488,536 B	(87.2%)
java.util.ArrayList	1,049	(0.2%)	25,176 B	(0%)	51,903,800 B	(86.2%)
java.lang.String	275,712	(43.9%)	6,617,088 B	(11%)	51,472,072 B	(85.5%)
kotlin.jvm.internal.Ref$ObjectRef	1	(0%)	16 B	(0%)	51,440,656 B	(85.4%)
org.example.files.InMemoryReader	1	(0%)	16 B	(0%)	51,440,640 B	(85.4%)
org.example.files.InMemoryReader#1			16 B	(0%)	51,440,640 B	(85.4%)
<fields>						
scenarios = java.util.ArrayList#221 : 250,000 elements			24 B	(0%)	51,440,624 B	(85.4%)
static <classLoader> = jdk.internal.loader.ClassLoaders			96 B	(0%)	28,744 B	(0%)
static <resolved_references> = java.lang.Object[]#944			24 B	(0%)	24 B	(0%)
<references>						
byte[]	277,016	(44.2%)	47,175,648 B	(78.4%)	46,932,128 B	(78%)
java.util.zip.ZipFile$Source	14	(0%)	1,120 B	(0%)	573,640 B	(1%)
java.util.HashMap	603	(0.1%)	28,944 B	(0%)	458,216 B	(0.8%)
java.util.HashMap$Node[]	1,183	(0.2%)	113,496 B	(0.2%)	446,024 B	(0.7%)
java.util.concurrent.ConcurrentHashMap	163	(0%)	10,432 B	(0%)	423,832 B	(0.7%)
java.util.concurrent.ConcurrentHashMap$Node[]	119	(0%)	46,128 B	(0.1%)	416,144 B	(0.7%)
java.util.HashMap$Node	3,800	(0.6%)	121,600 B	(0.2%)	401,752 B	(0.7%)
java.util.concurrent.ConcurrentHashMap$Node	4,898	(0.8%)	156,736 B	(0.3%)	325,280 B	(0.5%)
jdk.jfr.internal.PlatformEventType	192	(0%)	18,432 B	(0%)	309,328 B	(0.5%)
java.util.Collections$UnmodifiableRandomAccessList	279	(0%)	6,696 B	(0%)	246,184 B	(0.4%)
jdk.jfr.internal.AnnotationConstruct	1,481	(0.2%)	35,544 B	(0.1%)	225,864 B	(0.4%)
jdk.jfr.ValueDescriptor	806	(0.1%)	32,240 B	(0.1%)	224,000 B	(0.4%)
int[]	1,593	(0.3%)	1,937,536 B	(3.2%)	163,152 B	(0.3%)
jdk.jfr.AnnotationElement	1,572	(0.3%)	37,728 B	(0.1%)	158,256 B	(0.3%)
java.util.ImmutableCollections$List12	2,385	(0.4%)	57,240 B	(0.1%)	157,968 B	(0.3%)

Kuva 2: InMemoryReader luokkaa käyttävän ohjelman heap dump VisualVM:ssä.

Kuvan 2 Heap Dump -näkymän Count-sarake näyttää olioiden tai tyyppien määrän heapissa. Size-sarake sisältää olioiden yhteenlasketun koon. Retained size -sarake on arvio muistista, joka vapautuisi kun kyseinset oliot viittauksineen poistettaisiin [38]. Kuvasta voidaan havaita InMemoryReaderin toimivan nimensä mukaisesti. Kaikki 250000 skenaariota ovat listassa String-olioita, ladattuna suoraan muistiin ja varaavat noin 49 MB JVM:n heap-muistia.

Kun ohjelman lukija implementaatio vaihdetaan PostgreSQL-tietokantaa käyttävään DatabaseReader-luokkaan, huomataan, että muistinkäyttö on huomattavasti vähäisempää, kuten havaitaan kuvassa 3.



Kuva 3: DatabaseReader-luokkaa käyttävän ohjelman heap dump VisualVM:ssä

Kuvan 3 Name-kolumni paljastaa, että DatabaseReader luokkaa itsessään sisältää vain metodit, jotka kutsuvat DatabaseManager-oliota, jonka tehtävänä on huolehtia tietokantayhteyspoolista ja suorittaa tietokantakyselyjä.

6 Lopputulos

6.1 Yhteenveto

Alkuperäisessä pelissä päädyttiin käyttämään RandomAccessFile-pohjaista ratkaisua, ja ensimmäinen skenaariopohjainen peli julkaistiin. Tuolloin todettiin, että muistinkäytön ongelma saatiin ratkaistua siirtymällä tiedostojen satunnaiseen lukemiseen.

Vaikka suorituskyvyn kannalta lukuoperaatio on suhteellisesti huomattavasti hitaampi, kyseinen operaatio on loppujen lopuksi vain pieni osa pelipalvelimella kierroksen käsittelyyn kuluvasta ajasta. Tästä syystä lukuoperaation suhteellinen hitaus ei muodosta merkittävää pullonkaulaa kokonaisuuden kannalta.

Tietokantaratkaisua olisi voinut vielä jatkokehittää hyödyntämään jonoa esihaettuja skenaarioita. Jonoa täytettäisiin automaattisesti aina, kun se täyttöaste laskisi tarpeeksi matalaksi, hakemalla useita skenaarioita tietokannasta yhdellä kyselyllä.

Opinnäytetyössä käytetty korttipakka esimerkki oli hieman yksinkertaisempi verrattuna todelliseen, jossa skenaariot olivat eri pituisia JSON-olioita. Eri kokoisten skenaarioiden käsittely tapahtuu kuitenkin hyvin samankaltaisesti. Sovelluksen käynnistyessä tarvitsee tiedosto skannata kerran ja jokaisen skenaarion aloituspositio tiedoston tavulistassa tulee tallentaa erilliseen indeksiin joko muistissa tai tiedostossa.

6.2 Jatkokehitys

Skenaarioihin liittyviä ratkaisuja on ehditty jo jatkokehittämään. Ensimmäinen merkittävä askel oli jakaa skenaariotiedosto useampaan, joilla on omat painot. Tämä mahdollistaa sen, ettei tietyntyyppisten skenaarioiden todennäköisyys ole enää sidonnainen skenaariotiedoston kokoon.

Tämänhetkistä tilannetta ajatellen seuraava askel systeemin jatkokehityksessä jatkokehitystä olisi siis siirtyä käyttämään Java NIO -paketin FileChannel-luokkaa. Kun lukemista tekevä luokka olisi päivitetty käyttämään puskuroitua I/O:ta, voitaisiin ryhtyä tekemään lisätutkimusta MappedByteBufferin käytettävyyden suhteen.

Lähteet

1 wc(1) - Linux man page. Verkkoaineisto. <https://linux.die.net/man/1/wc>. Luettu 19.02.2025.

2 Operating Systems: Three Easy Pieces, 39.1 Files And Directories (Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau). Verkkoaineisto. <https://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf> Luettu 19.02.2025.

3 Kotlin File.printWriter. Verkkoaineisto. <https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.io/print-writer.html>. Luettu 19.02.2025.

4 Glossary of Unicode Terms. Verkkoaineisto. <https://www.unicode.org/glossary/>. Luettu 26.03.2025.

5 du(1) - Linux man page. Verkkoaineisto. <https://linux.die.net/man/1/du>. Luettu 19.02.2025.

6 Oracle – About the Java Technology. Verkkoaineisto. <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>. Luettu 26.03.2025.

7 Kotlin Language Reference. Verkkoaineisto. <https://kotlinlang.org/docs/kotlin-reference.pdf>. Luettu 19.02.2025.

8 Memory Footprint of the JVM. Verkkoaineisto. <https://spring.io/blog/2019/03/11/memory-footprint-of-the-jvm>. Luettu 26.02.2025.

9 Baeldung: Stack Memory and Heap Space in Java. Verkkoaineisto. <https://www.baeldung.com/java-stack-heap>. Luettu 26.02.2025.

10 The Structure of the Java Virtual Machine. Verkkoaineisto. <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html>. Luettu 26.02.2025.

11 Memory usage of Java Strings and string-related objects. Verkkoaineisto. <https://www.javamex.com/tutorials/memory/string_memory_usage.shtml>. Luettu 19.02.2025.

12 DZone Compact Strings. Verkkoaineisto. <https://dzone.com/articles/going-beyond-java-8-compact-strings>. Luettu 19.02.2025.

13 Baeldung Compact String. Verkkoaineisto. <https://www.baeldung.com/java-9-compact-string>. Luettu 19.02.2025.

14 Java File. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html>. Luettu 19.03.2025.

15 POSIX read. Verkkoaineisto. <https://pubs.opengroup.org/onlinepubs/9799919799/functions/read.html>. Luettu 26.02.2025.

16 POSIX close. Verkkoaineisto. <https://pubs.opengroup.org/onlinepubs/9799919799/functions/close.html>. Luettu 26.02.2025.

17 Java RandomAccessFile. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/RandomAccessFile.html>. Luettu 18.03.2025.

18 JAVA NIO. Verkkoaineisto. <https://www.oreilly.com/library/view/java-nio/0596002882/ch01.html>. Luettu 19.03.2025.

19 Time complexity of Java Collections. Verkkoaineisto. <https://www.baeldung.com/java-collections-complexity>. Luettu 18.03.2025.

20 What is Cache? Verkkoaineisto. <https://www.spiceworks.com/tech/tech-101/articles/what-is-cache/>. Luettu 19.03.2025.

21 PostgreSQL COPY. Verkkoaineisto. <https://www.postgresql.org/docs/current/sql-copy.html>. Luettu 19.03.2025.

22 PostgreSQL Serial Types. Verkkoaineisto. <https://www.postgresql.org/docs/current/datatype-numeric.html#DATATYPE-SERIAL>. Luettu 18.03.2025.

23 PostgreSQL EXPLAIN ANALYZE. Verkkoaineisto. <https://www.postgresql.org/docs/current/sql-explain.html>. Luettu 18.03.2025.

24 PostgreSQL PRIMARY KEY. Verkkoaineisto. <https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-PRIMARY-KEYS>. Luettu 18.03.2025.

25 HikariCP. Verkkoaineisto. <https://github.com/brettwooldridge/HikariCP>. Luettu 18.03.2025.

26 Java Synchronized block. Verkkoaineisto. <https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html>. Luettu 19.03.2025.

27 Java FileChannel. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/channels/FileChannel.html>. Luettu 22.03.2025.

28 JUnit Parameterized Tests. Verkkoaineisto. <https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests>. Luettu 21.03.2025.

29 Java Executors. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html>. Luettu 22.03.2025.

30 JMH. Verkkoaineisto. <https://github.com/openjdk/jmh>. Luettu 21.03.2025.

31 JMH Cheatsheet. Verkkoaineisto. <https://leogomes.github.io/assets/JMH_cheatsheet.pdf>. Luettu 21.03.2025.

32 JMH Blackhole. Verkkoaineisto. <https://javadoc.io/doc/org.openjdk.jmh/jmh-core/latest/org/openjdk/jmh/infra/Blackhole.html>. Luettu 21.03.2025.

33 Java MemoryMXBean. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/api/java.management/java/lang/management/MemoryMXBean.html>. Luettu 22.03.2025.

34 Eclipse Memory Analyzer - Heap Dump. Verkkoaineisto. <https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.mat.ui.help/welcome.html>. Luettu 24.03.2025.

35 The jps Command. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/specs/man/jps.html>. Luettu 24.03.2025.

36 The jcmd command. Verkkoaineisto. <https://docs.oracle.com/en/java/javase/17/docs/specs/man/jcmd.html>. Luettu 24.03.2025.

37 VisualVM Features. Verkkoaineisto. <https://visualvm.github.io/features.html>. Luettu 21.03.2025.

38 Eclipse Memory Analyzer - Shallow vs Retained Heap. Verkkoaineisto. <https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.mat.ui.help/welcome.html>. Luettu 22.03.2025.