LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

DOKUMENTU MEKLĒŠANAS SISTĒMA

BAKALAURA DARBS

Autors: **Arnolds Sarmulis**
Studenta apliecības Nr.: as18284
Darba vadītāja: Dr.sc.comp. Lelde Lāce

RĪGA 2022

ANOTĀCIJA

Bakalaura darbs ir veltīts dokumentu meklēšanas problēmai, kas tiek risināta izmantojot dokumentu indeksa struktūru. Darba ietvaros ir izpētītas un izanalizētas dažādas datu struktūras, kas sniedz iespēju indeksēt dokumentus. Aprakstītās datu struktūras tiek definētas izmantojot klašu diagrammas un to darbības principi tiek attēloti uz konkrētiem piemēriem izmantojot instanču diagrammas. Ir aprakstītas, notestētas un optimizētas autora realizētas programmas datu struktūru izveidošanai un arī dokumentu meklēšanai. Darbā ir pamatoti un ar testēšanas rezultātiem pierādīti ātrdarbības ziņā efektīvākie varianti indeksa struktūras izveidošanai un meklēšanai tajā. Tika aprakstīti programmu realizācijas ietvaros izmantotie tehnoloģiskie risinājumi.

Atslēgvārdi: dokumentu meklēšana, datu struktūra, datu uzglabāšana un apstrāde, datu bāze

ABSTRACT

DOCUMENT SEARCH SYSTEM

The bachelor's thesis is dedicated to the problem of document search, which is solved using the structure of the document index. Within the framework of the work, various data structures that provide an opportunity to index documents have been researched and analyzed. The described data structures are defined using class diagrams and their operating principles are depicted on specific examples using instance diagrams. The programs implemented by the author for creating data structures and also for searching for documents have been described, tested and optimized. The most effective options for creating and searching the structure of the index in terms of speed are substantiated and proved by the test results. The technological solutions used in the implementation of the programs were described.

Keywords: document search, data structure, data storage and processing, database

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

Boolean – Datu tips kurš satur tikai divas vērtības vai nu patiess vai nepatiess.
Brakepoint – Programmkoda rinda kurā programmas izpilde tiek apstādināta, izmanto dažādiem atkļūdošanas uzdevumiem.
C# – Dažādu paradigmu, bet pamatā objektorientēta programmēšanas valoda. Sniedz iespēju izstrādāt dažāda tipa programmatūru.
CSV (*angl.* Comma Separated Values) – Strukturētu datu pieraksta veids, kurā ieraksti tiek atdalīti ar noteiktu atdalītājsimbolu, kas parasti ir komats.
DBPS – Datu bāzu pārvaldības sistēma ir programmatūra, kas nodrošina dažādus ar datu uzglabāšanu saistītus servisus.
Dokumentu indeksa struktūra – Datu struktūra kurā tiek glabātas dokumentu kolekcijas katra dokumenta visas vārdu pozīcijas sakartētas ar attiecīgo vārdu saturu, to strukturālajām kategorijām un citiem elementiem.
Entity Framework – Atvērtā pirmkoda ORM ietvars priekš .NET programmām.
Id – Tabulas vai datu vienības unikālais un primārais identifikators.
Int Identity – SQL datubāzes serverī nodrošina, ka ievietojot jaunu ierakstu tam automātiski piešķirs "Id" vērtību, kas par noteiktu skaitli ir lielāka par esošo lielāko.
IT – Informācijas tehnoloģijas.
Lambda – Dod iespēju C# programmēšanas valodā rakstītas anonīmas funkcijas pēc funkcionālās programmēšanas principiem.
LINQ (*angl.* Language Integrated Query) – Unificēta sintakse, kas tiek izmantota C# programmēšanas valodā un tiek lietota priekš operācijām ar datu kolekcijām.
.NET – Platforma, kas ietver sevī dažādas programmēšanas valodas un bibliotēkas dažādu tipu programmu izstrādei un darbināšanai.
Null – Reference, kas nenorāda uz kādu atmiņas apgabalu, tukša vērtība.
OOP (*angl.* Object Oriented Programming) – Objektorientētā programmēšanas paradigma.
ORM (*angl.* Object Relation Mapping) – Objektu relāciju kartējuma ietvars, nodrošina iespēju veikt dažādus uzdevumus ar relāciju datu bāzes objektiem izmantojot tās objektorientētu reprezentāciju, kā arī citas līdzīga rakstura funkcijas.

OS (*angl.* Operating System) – Operētājsistēma ir sistēmas līmeņa programmatūra, kas nodrošina programmu komunikāciju ar aparatūru un vēl citas nozīmīgas funkcijas.

Pointeris – Objekts, kas satur digitāla datu nesēja atmiņas adresi.

Pozīciju saraksts – Dokumentu indeksa struktūrā katram vārdam lemmas formā atbilst visas tā pozīcijas visos dokumentos un katra vārda pozīciju saraksts satur visas šīs pozīcijas. Pozīciju saraksts tiek veidots izmantojot "Nākamā pozīcija" asociāciju.

SmallInt, Int, BigInt – Relāciju datu bāzēs lietoti datu tipi, katrs no tiem apzīmē veselu skaitli ar noteiktu izmēru 2, 4 un 8 baiti.

SQL (*angl.* Structured Query Language) – Strukturēta vaicājumu valoda, kas sniedz iespēju definēt dažādas operācijas ar relāciju datu bāzes objektiem.

SQL Server Management Studio – Integrēta vide SQL un relāciju datu bāzu infrastruktūras pārvaldīšanai.

Vārds lemmas formā – Valodas vārds tā pamatformā, lietvārdam tas ir vienskaitļa nominatīvs, bet darbības vārdam nenoteiksme.

Visual Studio – Integrēta izstrādes vide, ko izstrādājis ir Microsoft. Nodrošina dažādas ar izstrādi saistītas funkcijas, dodot iespēju izstrādāt dažāda tipa programmatūru.

Windows – Microsoft izstrādāta operētājsistēma.

XML (*angl.* Extensible Markup Language) – Marķējuma valoda, ko var izmantot strukturētu datu aprakstīšanai.

IEVADS

Mūsdienās digitalizācijas procesa ietekmē daudzi arī ar IT tieši nesaistīti uzņēmumi ir pārorientējuši savu darbību tā, lai tās veikšanā tiktu vairāk izmantotas digitālas sistēmas, un līdz ar to arī biežāk dokumenti tiek glabāti digitālā formā. Lai pilnvērtīgi varētu izmantot dokumentu digitālās uzglabāšanas sniegtās iespējas ir nepieciešams nodrošināt efektīvus servisus visām problēmām, kas lietotājiem varētu rasties šajā kontekstā. Dokumentu meklēšanas problēma ir viena no būtiskajām problēmām digitālajā vidē. Darbā pētītās meklēšanas problēmas fokuss ir uz situāciju, kurā lietotājam ir jāatrod noteikta informācija lielā dokumentu kolekcijā, kas glabājas uz datu nesēja un viņam precīzi nav zināms kurā dokumentā tieši atrodas viņam nepieciešamā informācija, bet lietotājam ir zināms kaut kas par šo informāciju, kas var būt atslēgvārdi, frāze, dokumenta metadati vai tamlīdzīga rakstura informācija, attiecīgi lietotājs izmanto viņam zināmo informāciju, lai dokumentu meklēšanas programmā atrastu oriģinālo dokumentu uz datu nesēja.

Šīs problēmas manuāla risināšana var paņemt ļoti daudz laika un it īpaši biznesa vidē, kur uzņēmuma kopējā datu nesējā tiek glabāti milzīgi daudzu lietotāju veidoti dokumentu apjomi un kur laikam ir liela vērtība, ir nepieciešamība spēt automatizēti atrisināt šāda tipa problēmas izmantojot programmu. Spēt automatizēti veikt dokumentu meklēšanu ir būtiski ne tikai biznesa vidē, bet arī ikdienišķos apstākļos, kur lietotājam ir vajadzība atrast informāciju pašam savos uzglabātajos dokumentos, jo ja tie ir daudz un veidoti laika gaitā, vai daļa no tiem nav paša veidota, tad manuāla meklēšana var paņemt daudz laika, un tas ir tāds uzdevums, ko programma spēj veikt būtiski efektīvāk un ātrāk par pašu lietotāju.

Šajā darbā tiek pētītas metodes un tehnoloģijas, kuras ir pamatā programmai, kas risina dokumentu meklēšanas problēmu. Darbā praktiski izpētītie tehnoloģiskie risinājumi, kas ir pierādīti kā efektīvi ir derīgi ne tikai tieši šādai dokumentu meklēšanas programmai, bet arī citiem līdzīgiem tāda rakstura uzdevumiem, kur ir svarīgi pēc iespējas ātrāk ievietot, atjaunot vai pārlūkot liela apjoma datu bāzē glabātus datus. Darba pamatmērķis ir izpētīt un noteikt efektīvākos risinājumus dokumentu meklēšanas programmas realizēšanai, kur risinājumu efektivitāte primāri tiek mērīta pēc to ātrdarbības.

Darbā tiek risināti sekojoši uzdevumi:

- Izpētīt un izanalizēt datu struktūras, kas sniedz iespēju indeksēt dokumentus;
- Sniegt ieskatu datu struktūru apstrādes algoritmos;

- Sniegt ieskatu pamatfunkcijās, kuras ir iespējams realizēt aprakstītajās datu struktūrās un salīdzināt tās ar citu autoru meklēšanas programmu funkcijām;
- Aprakstīt teorētiskos pamatus tehnoloģiskajiem risinājumiem, kurus var izmantot, lai uzglabātu un uzturētu dokumentu indeksa datu struktūru;
- Aprakstīt dažādas uzprogrammētās dokumentu indeksa datu struktūras izveidošanas funkcijas realizācijas, kā arī realizēt dažādas ātrdarbības optimizācijas šajās funkcijās;
- Notestēt izpildes laiku indeksa datu struktūras izveidošanas funkcijām, to dažādajām optimizācijām un balstoties uz testēšanas rezultātiem noteikt variantu ar viss ātrāko izpildes laiku;
- Noprogrammēt, aprakstīt un izanalizēt dažādas dokumentu meklēšanas pēc frāzes realizācijas aprakstītajās datu struktūrās;
- Notestēt dažādo dokumentu meklēšanas realizāciju variantu ātrdarbību, balstoties uz testēšanas rezultātiem noteikt viss ātrāko variantu;
- Izskaidrot un pamatot dažādās programmu izpildes ātrdarbības optimizācijas, kas tiek realizētas gan dokumentu indeksa struktūras izveidošanā, gan dokumentu meklēšanā pēc frāzes.

Darba izstrādē tiek izmantotas sekojošas pētniecības metodes:

- Aprakstošā metode - tiek izpētītas, izprastas un aprakstītas tehnoloģijas un metodes, kas var kalpot par daļu no problēmas risinājuma;
- Salīdzinošā metode - aprakstītās metodes tiek savstarpēji salīdzinātas, tiek noteiktas to pozitīvās un negatīvās īpašības;
- Analītiskā metode - tiek izpētīti un analizēti dažādi literatūras avoti.

Darba izstrādē tiek lietoti tiešsaistes avoti - citu autoru veikti pētījumi, tehnoloģiju ražojošo uzņēmumu veidoti formāli apraksti un dokumentācijas.

Darba pirmajā nodaļā tiek modelētas, analizētas, pētītas un aprakstītas datu struktūras kuras tiek veidotas tā, lai tajās var veikt meklēšanu pēc indeksēšanas principa. Datu struktūru modelēšanā un attēlošanā tiek izmantotas klašu un instanču diagrammas, kur klašu diagrammas definē datu struktūras entītijas, to asociācijas, lomas un kardinalitātes, bet instanču diagrammās ar konkrētiem piemēriem tiek demonstrēti datu struktūru darbības būtiskākie principi.

Otrajā nodaļā ir aprakstīti tehnoloģisko risinājumu teorētiskie pamati, kuri tālāk attiecīgi arī tiek izmantoti praktiskajā daļā uzprogrammējot divas atsevišķas programmas, kuras izveido dokumentu indeksa struktūru datu bāzē, kā arī ir realizētas, aprakstītas un notestētas dažādas

ātrdarbības optimizācijas šīm programmām. Ir aprakstīti un izskaidroti tehnoloģiskie aspekti, kas ietekmē realizēto programmu ātrdarbību, kā arī balstoties uz testēšanas rezultātiem ir noteikts ātrākais risinājums.

Trešajā nodaļā ir aprakstīti uzprogrammētie algoritmi kuri veic dokumentu meklēšanu pēc frāzes pirmās nodaļas aprakstītajās datu struktūrās. Ir izanalizēti un izskaidroti algoritmu darbības principi izmantojot algoritmu augsta līmeņa aprakstus, kā arī instanču un aktivitāšu diagrammas. Realizētie meklēšanas varianti tiek notestēti uz dažāda apjoma testa datiem, līdz ar to arī tiek noteikta to ātrdarbības efektivitāte un tiek noteikts viss ātrākais variants.

Visās trijās nodaļās kopumā ir iekļauti 29 attēli un 14 tabulas, kā arī izmantoti 17 literatūras avoti.

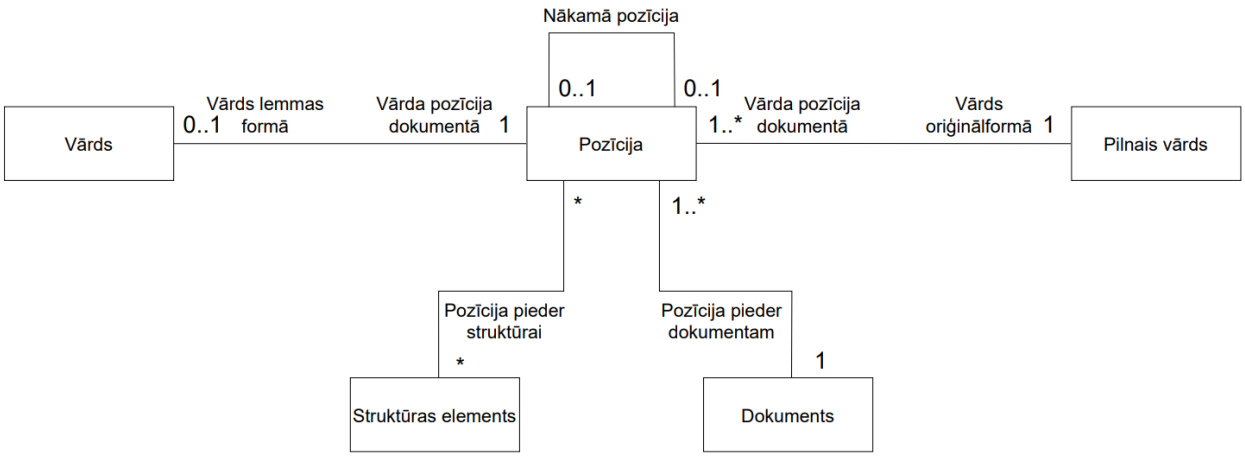
1. DATU STRUKTŪRAS DOKUMENTU MEKLĒŠANAI

Praksē ir izpētīts un pierādīts, ka ātru meklēšanu lielā dokumentu kopā ir iespējams veikt, ja tā tiek izpildīta nevis apstaigājot visus dokumentu oriģinālus failu sistēmā, kas paņemtu ļoti daudz laika, bet meklēšanu veicot tikai vienā speciālā datu struktūrā – dokumentu indeksā, kurā noteiktā strukturālā formā tiek glabāts saturs no daudziem dokumentiem [12, 13]. Šīs struktūras pamatā ir dokumentos esošie vārdi, kas ir sakartēti ar to attiecīgajām pozīcijām atbilstoši dokumentu oriģinālajam saturam un papildus tam ir iespējams glabāt arī cita rakstura informāciju kā dokumenta metadatus, dokumenta oriģinālās struktūras raksturojošus elementus u.tml. Tādējādi atrast dokumentus pēc to satura ir iespējams nevis apstaigājot pašus dokumentus, bet dokumentu indeksa struktūru, kas reprezentē dokumentu saturu tādā formā, ka tajā ir iespējams veikt efektīvu meklēšanu un implementēt arī dažādas papildus funkcijas.

1.1. Dokumentu indeksēšanas datu struktūras pamatvariants

Lai varētu veikt izpildes laika ziņā efektīvu meklēšanu lielā vārdu kopā ir nepieciešams to uzglabāt tā, lai atrast meklēto varētu apstaigājot mazu daļu no visas lielās vārdu kopas. Ja glabāta tiek liela dokumentu kolekcija vienā valodā, tad viennozīmīgi skaidrs ir tas, ka daudzi tās vārdi vairākas reizes atkārtosies tajos pašos vai citos locījumos. Tāpēc apstaigājamo vārdu kopu var samazināt vārdus tajā uzglabājot lemmas formā tādējādi vārdu kopā tiks uzglabāti tikai unikāli vārdi vienā pamata locījumā. Ja šie unikālie vārdi lemmas formā uzglabāti tiek sakārtoti, tad tiem var veikt bināro meklēšanu, vai datu bāzē meklēšanu B plus kokā izmantojot indeksu un tas dod iespēju ļoti ātri atrast jebkuru vārdu dokumentu kolekcijā.

Katrs unikālais vārds lemmas formā ir sasaistīts ar pozīciju sarakstu kurā atrodas tā katra pozīcija katrā dokumentā, līdz ar to arī atrodot vārdu var uzreiz piekļūt visām tā indeksa struktūras instancēm visos dokumentos. Katrs pozīciju saraksta elements ir sasaistīts ar pilno vārdu, tā strukturālo kategoriju, un pašu dokumentu ar tā metadatiem. Visa šī papildus informācija sniedz iespēju realizēt plašāku meklēšanas funkcionalitāti.



1.1. att. **Dokumentu indeksēšanas datu struktūras pamatvarianta klašu diagramma**

Attēla 1.1. datu struktūra sastāv no 5 tabulām, kur katrai no tām ir sava loma:

- Tabula "Vārds" glabā dokumentu kolekcijā esošos vārdus lemmas formā bez dublikātiem. Vārdu un frāžu meklēšana dokumentos primāri notiek pēc šīs tabulas izmantojot "Id" lauku kurš satur ar jaucējfunkciju no vārda satura iegūtu veselu skaitli. Ieraksti tiek glabāti sakārtoti pēc "Id" lauka un līdz ar to izmantojot bināro meklēšanu, vai meklēšanu B plus kokā jebkuru vārdu dokumentu kolekcijā var atrast O(log n) laikā;
- Tabulā "Pozīcija" tiek glabātas katra dokumentu kolekcijā esošā dokumenta visas vārdu pozīcijas. Vārdu pozīcijas tabulā tiek glabātas secīgi tāpat kā tās ir dokumentā, tā lai ērti varētu iegūt teksta fragmentu no dokumenta satura. Vārdu pozīcijas atbilstoši vārdam lemmas formā ir sasaistītas izmantojot "nākamā pozīcija" referenci tā lai atrodot vārdu lemmas formā, tam var ātri iegūt sarakstu ar visām pozīcijām visos dokumentos kuros tas atrodas, turpmāk "Pozīciju saraksts";
- Tabulā "Pilnais vārds" tiek glabāti vārdu pozīcijām atbilstošie pilnie vārdi, tādā pašā formā, kā tie atrodas dokumentā. Pilnos vārdus var izmantot dažādām papildus funkcijām kā piemēram meklēšanas rezultātu sakārtošana pēc atbilstības ar lietotāja meklēto teksta fragmentu, vai dokumenta oriģinālam atbilstoša teksta fragmenta uzkonstruēšanai;
- Tabula "Dokuments" satur dokumenta metadatu informāciju, kas ļauj meklēšanā iesaistīt papildus informāciju par dokumentu, kā arī pie atgrieztajiem meklēšanas rezultātiem lietotājam sniegt vairāk informāciju par atrastajiem dokumentiem;
- Tabulā "Struktūras elements" tiek glabāta informācija par pozīcijas vārda piederību noteiktam struktūras tipam – teikuma beigas vai sākums, u.tml. Šo un līdzīgu informāciju

var izmantot dažādu funkciju veidošanai, piemēram lai varētu pie atgrieztajiem rezultātiem parādīt teksta priekšskatījumu, kurš sevī ietver atrasto frāzi un tās kontekstu.

1.1.1. Datu struktūras izveidošana

Datu struktūrai ir jābūt izveidotai pirms tajā tiek veikta meklēšana. Lai varētu veikt sekmīgu meklēšanu izveidošanas procesam jābūt veiksmīgi pabeigtam visa meklējamā apgabala ietvaros. Veidojot datu struktūru lietotājam ir jāapzinās kādu failu sistēmas daļu viņš vēlēsies iekļaut indeksa struktūrā un attiecīgi ir jāsniedz šī informācija programmai pirms tiek uzsākts veidošanas process. Tāpat lietotājam jāapzinās kādus dokumentu failu tipus programma spēj apstrādāt un attiecīgi jāsniedz informācija par failu tipiem kādus lietotājs vēlēsies iekļaut indeksā. Lai īstenotu sekmīgu struktūras izveidošanas procesu jāveic noteikti soļi:

1) Jāapstaigā katras mapes un tās apakšmapes dokuments kurš atbilst kādam no lietotāja noteiktajiem failu tipiem;
2) Dokumentam jāizveido un jāizpilda ieraksts tabulā “Dokuments”, kurā tiek glabāta informācija par dokumenta metadatiem;
3) Jāapstaigā katrs dokumenta vārds nosakot tā robežas;
4) Vārds jāpārveido lemmas formā, tad izmantojot jaucējfunkciju no tā iegūst skaitli, ko ievieto “Id” laukā, ja tāds vārds lemmas formā jau neeksistē tabulā “Vārds”, tad tas ir tur jāievieto un tam attiecīgi arī jāizveido tā pirmā pozīcija tabulā “Pozīcija”, ja vārds jau eksistē tabulā “Vārds”, tad jāpievieno jauna pozīcija tabulā “Pozīcija”, tad ir jāiegūst šī vārda pozīciju saraksta pēdējā pozīcija un tad to savieno ar jauno izveidoto pozīciju izmantojot “nākamā pozīcija” referenci (atbilstoši vienvirziena saistītā saraksta uzturēšanas principiem);
5) Vārda oriģinālā forma jāievieto tabulā “Pilnais vārds” un jāsasaista ar jauno pozīcijas ierakstu, ja vārds tādā formā jau tur nav, bet ja ir tad jāveic tikai datu sasaiste ar jau esošo ierakstu;
6) Jānosaka kādam struktūras tipam vai tipiem vārds pieder un tā pozīcijas instance ir jāsasaista ar attiecīgajiem ierakstiem tabulā “Struktūras elements”;

7) Pozīcija tiek sasaistīta ar dokumenta metadatiem sasaistot izveidoto pozīcijas ierakstu ar attiecīgā dokumenta ierakstu tabulā “Dokuments”.

Secinājumi

Struktūru veidojot būs nepieciešams apstaigāt visu to failu sistēmas daļu, kas tiek iekļauta dokumentu indeksā, kā arī katra dokumenta katru vārdu, tam veikt analīzi/apstrādi, un arī katram vārdam izveidot vienu jaunu un potenciāli vairākus jaunus ierakstus datu struktūrā. Tātad datu struktūras pirmreizējā izveidošana ir laikietilpīga procedūra, kas var aizņemt no dažām minūtēm līdz pat daudzām stundām. Programmu jāveido tā, lai veidojot datu struktūru tā tikai daļēji noslogotu datora resursus un lietotājs indeksa struktūras veidošanas laikā varētu datorā lietot tādas funkcijas kā interneta pārlūkprogrammas, failu pārlūkošanu un rediģēšanu bez būtiskiem darbības palēninājumiem, kur būtisks darbības palēninājums būtu +1 sekunde tām pašām funkcijām zemas noslodzes apstākļos.

1.1.2. Datu struktūras atjaunošana

Sākotnēji izveidojot dokumentu indeksa struktūru tā atbildīs aktuālajai situācijai, bet līdzko dokumentos tiks veiktas kādas izmaiņas, tā izveidotā indeksa struktūra vairs potenciāli neatbildīs reālajai situācijai un potenciāli sniegs kļūdainus rezultātus. Lai novērstu šādu situāciju programmā ir jābūt iespējai atjaunot indeksa struktūru.

Katram dokumentam tabulā “Dokuments” ir jāglabā pēdējās atjauninašanas datums, tā lai to salīdzinot ar OS reģistrēto pēdējās modifikācijas datumu varētu noteikt vai dokumentu indeksa struktūra priekš attiecīgā dokumenta ir jāatjauno.

Lai veiktu datu struktūras atjaunošanu jāveic sekojoši soļi:

1) Jāapstaigā katrs dokumentu kolekcijas dokuments kura faila paplašinājums atbilst kādam no dokumentu indeksa struktūrā iekļautajiem dokumentu tipiem;

2) Izmantojot dokumenta pilno ceļu uz failu nesēja, kā identificējošu raksturlielumu, tas ir jāatrod tabulā "Dokuments" un tad jāatjauno dokumenta metadati tabulā "Dokuments". Ja dokuments veiksmīgi tika atrasts, tad tas jāuzglabā atrasto dokumentu masīvā, lai to izmantojot pēc tam var noteikt kuri dokumenti netika atrasti;
3) Ja izmantojot dokumenta pilno ceļu to neizdevās atrast tabulā "Dokuments", tad tiek pieņemts, ka šis ir jauns dokuments dokumentu kolekcijā un attiecīgi tas tiek pievienots tāpat kā tas tiek darīts datu struktūras pirmreizējā izveidošanā;
4) Dokumentam jāatrod indeksa datu struktūras tabulā "Dokuments" uzglabātais tā pēdējās atjaunināšanas datums, tas ir jāsalīdzina ar OS pēdējās modifikācijas datumu, un ja OS datums ir lielāks, tad ir jāveic indeksa struktūras atjaunināšana šim dokumentam. Ja dokumentam nav jāveic atjaunināšana, tad nākamos soļus jāizlaiž un jāpāriet pie nākamā dokumenta;

4.1.) (Turpmāk "šis dokuments" ir dokuments kurš attiecīgajā cikla iterācijā tiek atjaunots). Jānosaka vai dokumenta izmērs vārdu skaitā ir palielinājies (Salīdzinājumā ar pirms pēdējās rediģēšanas stāvokli);

4.1.1.) Ja vārdu skaits nav palielinājies, tad turpina ar 4.2 soli;

4.1.2.) Ja vārdu skaits ir palielinājies, tad tabulā "Pozīcija", visiem tiem elementiem, kuriem "nākamā pozīcija" referencē esošais elements ir elements, kas pieder šim dokumentam, ir references laukā "nākamā pozīcija" jāievieto "nākamā pozīcija" referencē esošā elementa lauka "nākamā pozīcija" vērtība, tādējādi izmetot šī dokumenta pozīciju no pozīciju saraksta, bet ja "nākamā pozīcija" referencē esošajam elementam laukā vērtība ir null, tad "nākamā pozīcija" laukā jāievieto vērtība null, un tādējādi tiek izmests pēdējais elements no pozīciju saraksta. Bet ja tabulā "Vārds" ir reference uz pozīciju saraksta pirmo elementu, un šim tabulas "Pozīcija" elementam "nākamā pozīcija" laukā vērtība ir null un tas ir šī dokumenta elements, tad ir jādzēš šis ieraksts no tabulas "Vārds" un tad no tabulas "Pozīcija";

4.2.) Jāizdzēš visi tie n pret n tabulas "StruktūraPozīcija" elementi kuri ir sasaistīti ar pozīcijām, kas pieder šim dokumentam, tad no tabulas "Pilnais vārds" jāizdzēš tikai tās vērtības uz kurām ir references tikai no šī dokumenta;

4.3.)

4.3.1.) Ja dokumenta izmērs vārdu skaitā ir palielinājies, tad ir jādzēš tabulas “Pozīcija” ieraksti, kas pieder šim dokumentam;

4.3.2.) Ja dokumenta izmērs vārdu skaitā nav palielinājies, ir jādzēš tabulas “Pozīcija” ieraksti, kas pieder šim dokumentam, bet katrs ieraksts pirms izdzēšanas ir jāievieto pozīciju masīvā, tā lai kad ierakstus atkal ievietos atpakaļ, var izmantot šo masīvu un tos ierakstus kuriem vārda pozīcija pieder vēljoprojām tam pašam lemmas vārdam, varēs ātri ievietot atpakaļ šos vārdus;

4.4.)

4.4.1.) Ja dokumenta izmērs vārdu skaitā ir palielinājies, tad šī dokumenta saturs ir jāievieto dokumentu indeksā tāpat kā tas tiek darīts pirmreizējā indeksa struktūras izveidošanā;

4.4.2.) Ja dokumenta izmērs vārdu skaitā nav palielinājies, tad šī dokumenta saturs ir jāievieto dokumentu indeksā tāpat kā tas tiek darīts pirmreizējā indeksa struktūras izveidošanā, bet ar izņēmumu, ka tabulai “Pozīcija” tiem vārdiem, kas atbilst vēljoprojām tam pašam lemmas vārdam pozīciju var ievietot no pieglabātā pozīciju masīva, bet tiem vārdiem kas neatbilst, šis vārds ir jāatrod tabulā “Vārds” un tad jāievieto atpakaļ pozīciju sarakstā kā tas bija pirms tam;

4.5.) Tabulā “Dokuments” jāatjauno dokumenta pēdējās atjaunināšanas datums;

5) Kad visi dokumentu kolekcijas dokumenti ir apstrādāti, tad ir jāapstaigā tabula “Dokuments” un katram tajā esošajam dokumentam jāveic pārbaude uz to vai tas atrodas atrastajā dokumentu masīvā, kas tiek veidots 2. solī, ja dokuments neatrodas šajā masīvā, tad visi šim dokumentam piederošie ieraksti ir jāizdzēš.

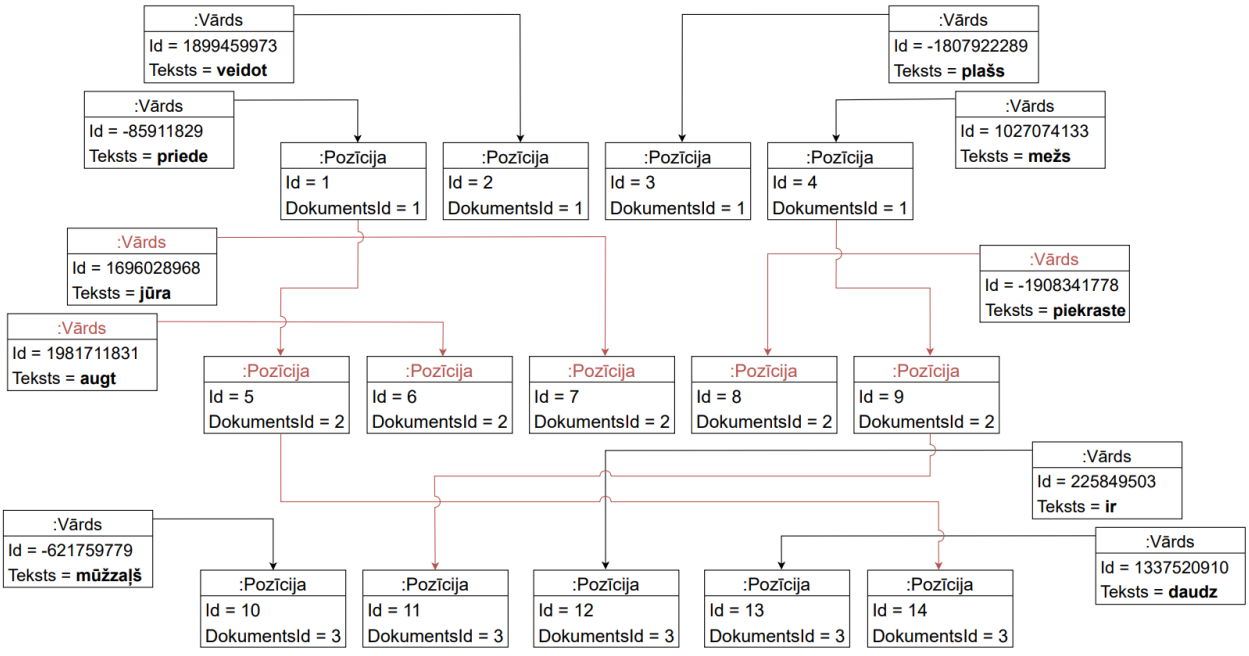
Piemērs

Piemērā tiek uzkonstruēta situācija kurā dokumenta izmērs vārdu skaitā ir palielinājies. Dokumenta atjaunošana tiek attēlota atbilstoši augstāk aprakstītajam algoritmam. Tiek dots sākotnējais dokumentu satura stāvoklis, un tiek pieņemts, ka mainījies ir dokuments 2, tā saturs ir mainījies uz “Priedes aug Baltijas jūras piekrastes mežos”. Attēlos tās bultas, kas iziet no tabulas “Pozīcija” un savieno to ar citu tabulas “Pozīcija” instanci ir “nākamā pozīcija” lauka references,

bet tās bultas, ka iziet no tabulas “Vārds” ir pozīciju saraksta sākums. Attēlos 1.2, 1.3 tiek iekļautas instances tikai no tabulām “Vārds” un “Pozīcija”, lai demonstrētu kā notiek atjaunināšana šajā indeksa struktūras daļā, citas tabulas netiek iekļautas.

Piemērā tiek doti 3 dokumenti un to sākotnējais stāvoklis:
Dokuments 1: Priedes veido plašus mežus.
Dokuments 2: Priedes aug jūras piekrastes mežos.
Dokuments 3: Mūžzaļos mežos ir daudz priedes.

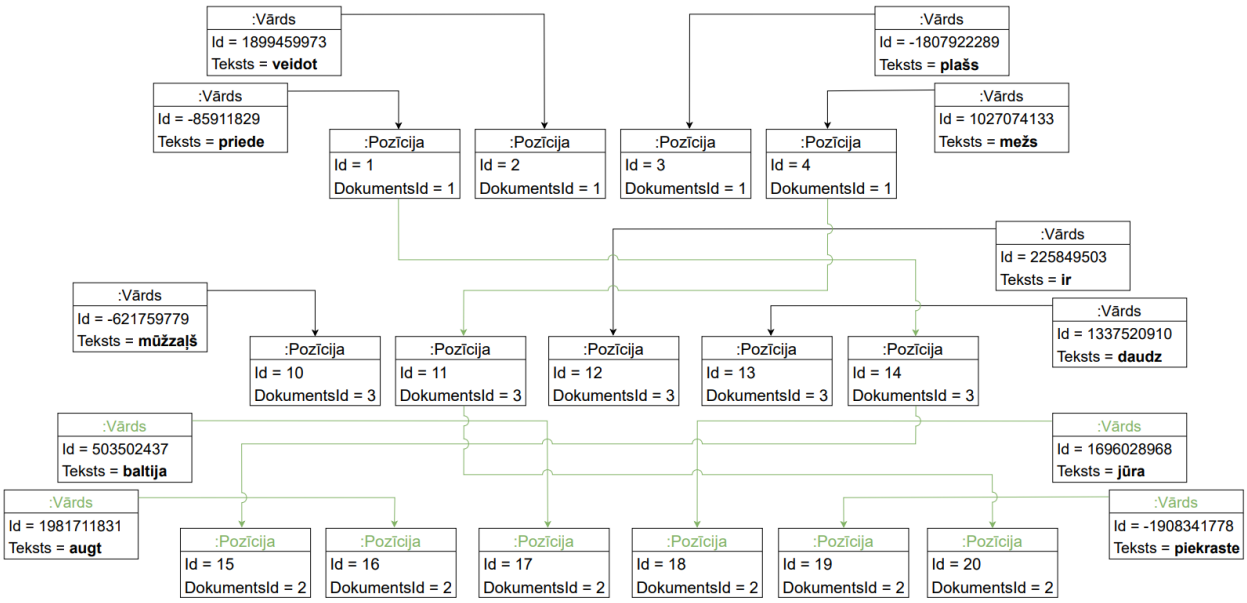
Attēlā 1.2. ir attēlots dokumentu indeksa struktūras sākumstāvoklis un izmantojot gaiši sarkano krāsu apzīmēti tiek objekti, kas tiks izmesti no indeksa datu struktūras. Tiek izmestas visas tabulas “Pozīcija” instances, kas pieder dokumentam 2, tiek izdzēstas tās tabulas “Pozīcija” lauka “nākamā pozīcija” references, kas norāda uz pozīcijām, kas tiks izdzēstas. Tāpat arī no tabulas “Vārds”, tiek izdzēsti tie vārdi kuriem pa pozīciju sarakstiem izriet tikai viena pozīcija, ja būtu vairākas, tad šie vārdi netiktu dzēsti.



1.2. att. **Datu struktūras sākumstāvokļa instanču diagramma**

Attēlā 1.3. ir attēlots dokumentu indeksa struktūras stāvoklis pēc atjaunošanas un izmantojot gaiši zaļo krāsu tiek apzīmēti objekti, kas tiek pievienoti indeksa datu struktūrā.

Dokumenta 2 iepriekšējās un jaunās pozīcijas tiek pievienotas ar “Id” lauku vērtībām, kas lielākas par iepriekšējo lielāko, tādējādi dokuments ieņem pēdējo sektoru pēc “Id” lauku vērtībām. Tā kā dokuments 3 ir ienācis dokumenta 2 vietā, tad attiecīgi arī pozīciju saraksti tagad savieno dokumenta 1 un dokumenta 3 vārdus, kas lemmas formā ir vienādi.



1.3. att. **Datu struktūras pēc atjaunošanas stāvokļa instanču diagramma**

Secinājumi

Kā alternatīva ierakstu dzēšanai un atpakaļ ievietošanai varētu būt pārbaude uz to vai dokumenta vārds atrodas indeksa datu struktūrā tajā pašā pozīcijā, tādā pašā formā. Šāds variants prasīs vairāk darbības un tā kopējais izpildes laiks būs ilgāks jo:

- Apstaigājot dokumenta saturu katram vārdam papildus būs jāveic sākumā meklēšana, tas būs jāatrod lemmas formā tabulā “Vārds”, tad apstaigājot pozīciju sarakstu jānonāk līdz attiecīgajai instancei tabulā “Pozīcija” un tad tikai varēs atjaunot datus;
- Ja dokumenta izmērs vārdu skaitā ir palielinājies, tad katram papildus jaunajam vārdam lai uzturētu to ka katra dokumenta visas pozīcijas pēc “Id” laukiem ir secīgas un blakus, ievietojot šo vārdu savā vietā pēc pozīcijas, vajadzētu visas pozīcijas kas lielākas par šo vārdu gan šajā pašā dokumentā, gan visos pārējos palielināt par 1, attiecīgi vajadzētu

dzēst un atkal ievietot visas references uz šiem ierakstiem, kas būtu ļoti laikietilpīga procedūra;

- Kā arī ja dokumenta izmērs vārdu skaitā ir palielinājies, tad pēc pirmā klāt nākušā vārda visiem pārējiem vārdiem kas pēc tā nav mainījušies pozīcijas dokumentā būs citas, kas radīs papildus problēmas identificēt vārdus ja tie tiek identificēti pēc pozīcijas numura un vārda oriģinālā pilnā satura.

Sliktākajā gadījumā dokumentu indeksa struktūras atjaunināšana prasītu pat vairāk laika nekā pirmreizējā indeksa izveidošanā, ja pilnīgi visi dokumenti būtu jāatjauno, bet visbiežāk tas būs daudz ātrāk, ja atjaunināšanu veiks pietiekami regulāri, un nebūs jāatjauno visi dokumenti.

Pie dokumentu meklēšanas rezultātu atgriešanas jāparedz tāda funkcija, ka dokumentiem tiek salīdzināti tabulā "Dokuments" glabātie pēdējās atjaunināšanas datumi ar OS sistēmā reģistrētajiem modifikācijas datumiem, un ja kāds OS datums ir jaunāks, tad pie meklēšanas rezultātiem lietotāja saskarnē ir jābūt paziņojumam, ka dokumentos ir bijušas izmaiņas, kas nav iekļautas dokumentu indeksa struktūrā un rezultāti var būt neprecīzi, jāparedz arī iespēja lietotājam uzreiz uzspiest pogu un veikt atjaunināšanu.

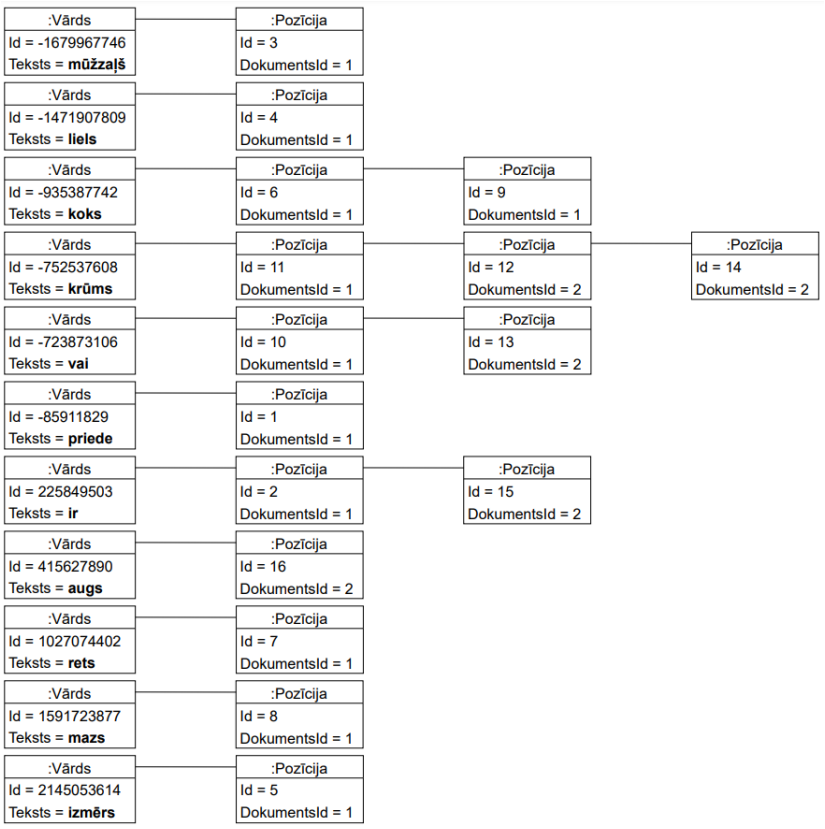
1.1.3. Datu struktūras piemērs

Piemērā tiek izmantoti 2 dokumenti, uz kuriem tiek nodemonstrēts, kā datu struktūra reprezentē oriģinālās dokumentu kopas saturu, un kā tabulu objektu instances ir saistītas un veido dokumentu indeksa struktūras pamatu. Piemērs ir daudz mazāks nekā tas būtu realitātē, bet principi paliek tie paši neatkarīgi no dokumentu daudzuma un izmēra.

Piemērā izmantotie dokumenti:
Dokuments 1= Priedes ir mūžzaļi, liela izmēra koki, retāk mazi koki vai krūmāji.
Dokuments 2= Krūms vai krūmājs ir augs.

Piemēra attēlojums (attēls 1.4.) instanču diagrammas formā veidots tā lai attēlotu primāri tabulu "Vārds" un sekundāri tabulu "Pozīcija" un tās pozīciju sarakstus, kur pozīcijas ir saistītas ar nākamo tā paša vārda instanci izmantojot referenci "nākamā pozīcija". Šis attēlojums skaidri

demonstrē to, ka šajā struktūrā var veikt efektīvu meklēšanu atrodot tabulā “Vārds” jebkuru vārdu O(log n) laikā izmantojot “Id” lauku, un tad atrastajam vārdam uzreiz tiek iegūts saraksts ar visām tā vārda instancēm visos dokumentos.

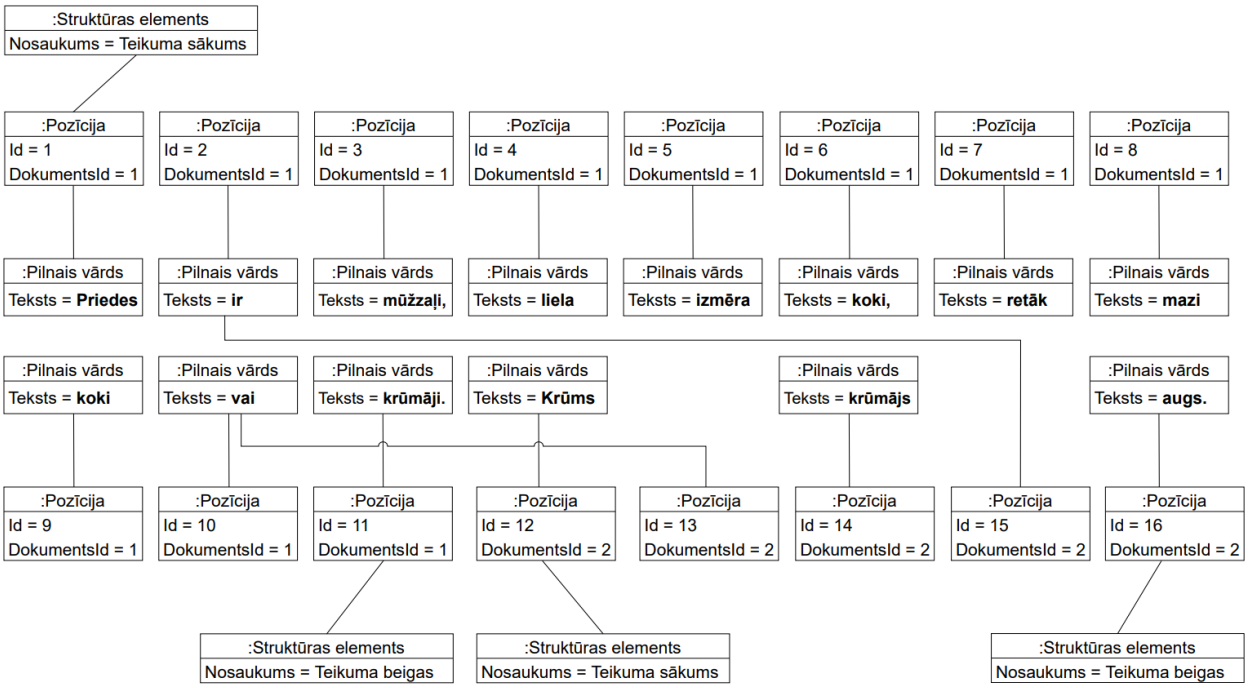


1.4. att. **Dokumentu indeksēšanas datu struktūras tabulu “Vārds”, “Pozīcija” attēlojums**

Piemēra nākamais attēlojums (attēls 1.5.) instanču diagrammas formā veidots tā lai attēlotu kā tabulas “Pozīcija” ieraksti ir sakārtoti pēc “Id” lauka vērtībām atbilstoši dokumenta oriģinālajam saturam, kur “Id” lauka vērtība nosaka pozīciju dokumentā. Tāpat arī visi dokumenti tiek indeksēti secīgi un viena dokumenta ietvaros izņemot robežas visas tā pozīcijas būs blakus tikai tā paša dokumenta pozīcijām, bet pozīciju saraksti var saistīt jebkuru dokumentu ar ikvienu citu dokumentu kolekcijā.

Attēlā 1.5. skaidri redzams, ka izmantojot tabulu “Pozīcija” un tabulu “Pilnais vārds” iespējams iegūt dokumenta oriģinālo vārdu saturu, kā arī tabula “Struktūras elements” dod iespējas noteikt teikuma robežas. Šajā piemēra gadījumā ir attēlotas tikai teikuma robežas, bet varētu būt arī teksta paragrāfa robežas, virsraksti, vai kāda cita tamlīdzīga rakstura informācija,

kas var būt nepieciešama. Katrs tabulas “Pozīcija” ieraksts ir arī saistīts ar dokumentu un tā metadatiem, kas piemēra attēlojumā nav iekļauts, lai samazinātu tā kopējo elementu skaitu.



1.5. att. **Dokumentu indeksēšanas datu struktūras tabulu “Pilnais vārs”, “Pozīcija”, “Struktūras elements” attēlojums**

1.1.4. Datu struktūru sniegtās funkcionālās iespējas

Tabulā 1.1. aprakstītas funkcijas, ko spēj realizēt šajā nodaļā aprakstītās datu struktūras “Dokumentu indeksēšanas datu struktūras pamatvariants”, “Alternatīvais variants dokumentu glabāšanai indeksa struktūrā”, “Alternatīvais variants dokumenta vārdu pozīcijām”.

1.1. tabula

Funkcijas ko kā minimums aprakstītās datu struktūras spēj realizēt

Funkcija	Ievaddati	Izvaddati
Meklēt dokumentus pēc noteiktas frāzes, vai vārdu kopas. Atgriezīs visus tos un	Frāze vai vārdu kopa teksta formā. Piemēram frāze: “mežs sastāv no”, vai vārdu kopa: “priedes”, “koki”.	Programmā tiek atgriezta dokumentus identificējoša rakstura informācija –

tikai tos dokumentus, kas satur noteiktu frāzi, vai vārdu kopu.		nosaukums, autors, ceļš uz dokumentu failu sistēmā, iespēja atvērt dokumentu ar peles klikšķi.
Meklēt dokumentus pēc vārdu kopas, vai frāzes, kas ietilpst teikuma, vai paragrāfa ietvaros. Atgriezīs visus tos un tikai tos dokumentus, kuros ir teikums, vai paragrāfs, kas satur noteiktus vārdus, vai noteiktu frāzi.	Frāze vai vārdu kopa teksta formā un pazīme par to ka ir jābūt teikuma vai paragrāfa ietvaros. Piemēram frāze: “koki vai krūmāji pazīstami” teikuma ietvarā, vai vārdu kopa “krūmājs”, “krūms”, “priede” paragrāfa ietvarā.	
Meklēt dokumentus pēc vārdu kopas, vai frāzes noteiktā attālumā (Attālumu mērot pēc vārdu skaita). Atgriezīs visus tos un tikai tos dokumentus, kuri satur frāzi vai vārdu kopu noteiktā attālumā.	Frāze vai vārdu kopa teksta formā un informācija par to starp cik vārdiem to ir jāvar atrast. Piemēram vārdu kopa: “priedes”, “sastāv” atrodas iekš 7 vārdiem, vai frāze un vārds: “koks ir”, “derīgs” iekš 6 vārdiem.	
Meklēt un sakārtot dokumentus izmantojot dokumenta metadatus. Eksistē arī iespēja kombinēt šo funkciju ar visām pārējām.	Frāze vai vārdu kopa teksta formā un dokumenta metadati. Piemēram frāze: “priedes ir koki” un metadatu lauks dokumenta autors: “Jānis”.	Programmā tiek atgriezta sakārtota pēc līdzības ar meklēto frāzi, vai metadatiem, dokumentus identificējoša rakstura informācija – nosaukums, autors, ceļš uz dokumentu failu sistēmā, iespēja atvērt dokumentu ar peles klikšķi.
Meklējot dokumentus pēc frāzes sakārtot atgrieztos rezultātus pēc atrasto frāžu līdzības pakāpes ar meklēto frāzi. (Simbolu virknēm	Frāze teksta formā. Piemēram frāze: “ir zināms, ka mežs”.	Programmā tiek atgriezta sakārtota pēc līdzības ar meklēto frāzi, dokumentus identificējoša rakstura informācija – nosaukums,

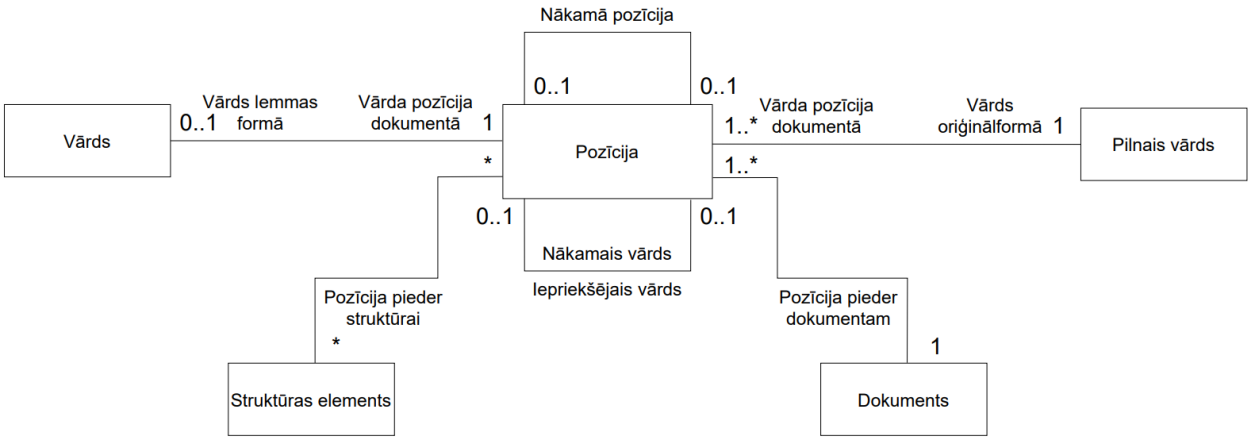
līdzības pakāpi aprēķina izmantojot levenšteina algoritmu [17])		autors, ceļš uz dokumentu failu sistēmā, iespēja atvērt dokumentu ar peles klikšķi.
Iegūt noteiktu teksta fragmentu no oriģinālā dokumenta satura. Sniedz iespēju lietotājam atgriezt dokumenta vai tā daļas priekšskatījumu (ātrā izpildes laikā nelasot dokumenta oriģinālu failu sistēmā). Šī funkcija sniedz iespēju iegūt dokumenta priekšskatījumu arī ja programmai nav tiešas piekļuves dokumenta oriģinālam failu sistēmā.	Informācija par to kāda teksta fragmenta daļa ir jāatgriež un kādu pozīciju ietvaros. Var būt paragrāfs, vai teikums, lappuse, vai pat viss dokuments.	Dokumenta priekšskatījuma fragments, lappuse vai pilnīgs dokumenta priekšskatījums.

1.2. Alternatīvais variants dokumentu glabāšanai indeksa struktūrā

Dokumentu veidošanas vai papildināšanas procesā to izmērs vārdu skaitā visbiežāk palielinās un līdz ar to veicot indeksa struktūras atjauninašanu būs jāveic darbietilpīgākais un ilgākais atjaunošanas variants. Dokumentu atjauninašanas ātrumam var būt liela nozīme gadījumos, kur lietotājs nav atradis meklēto datu struktūrā neesošu izmaiņu dēļ un šeit būtu svarīgi, ka lietotājam programmā ir paziņojums, ka sistēmā ir dokumenti, kas dokumentu indeksā nav atjaunoti un lai lietotājam būtu iespēja ātri atjaunot dokumentu indeksu un atrast meklēto ir svarīgi paātrināt dokumentu atjauninašanas procesu.

Dokumentu saturs, kas indeksa datu struktūrā tiek glabāts kā dokumenta vārdu pozīcijas tabulā “Pozīcija” ar to saistītajiem elementiem ir jāglabā tādā formā, ka vārdu pozīcijas ir izvietotas blakus viena otrai tāpat kā tas ir oriģinālajā dokumenta saturā. Indeksa struktūras aprakstītajā pamatvariantā šī īpašība tiek realizēta tā, ka dokumentu pozīcijas tabulā “Pozīcija” tiek izvietotas secīgi pēc “Id” lauku skaitliskajām vērtībām attiecīgi dokumenta izvietojumam. Šī īpašība nodrošina dažādu efektīvu funkciju realizēšanu, bet tās iepriekš aprakstītā indeksēšanas

struktūras pamatvarianta realizācija laika un operāciju ziņā sadārdzina datu struktūras atjaunošanu situācijā kur dokumenta izmērs vārdu skaitā ir palielinājies.



1.6. att. **Dokumentu indeksēšanas datu struktūras alternatīvā varianta klašu diagramma**

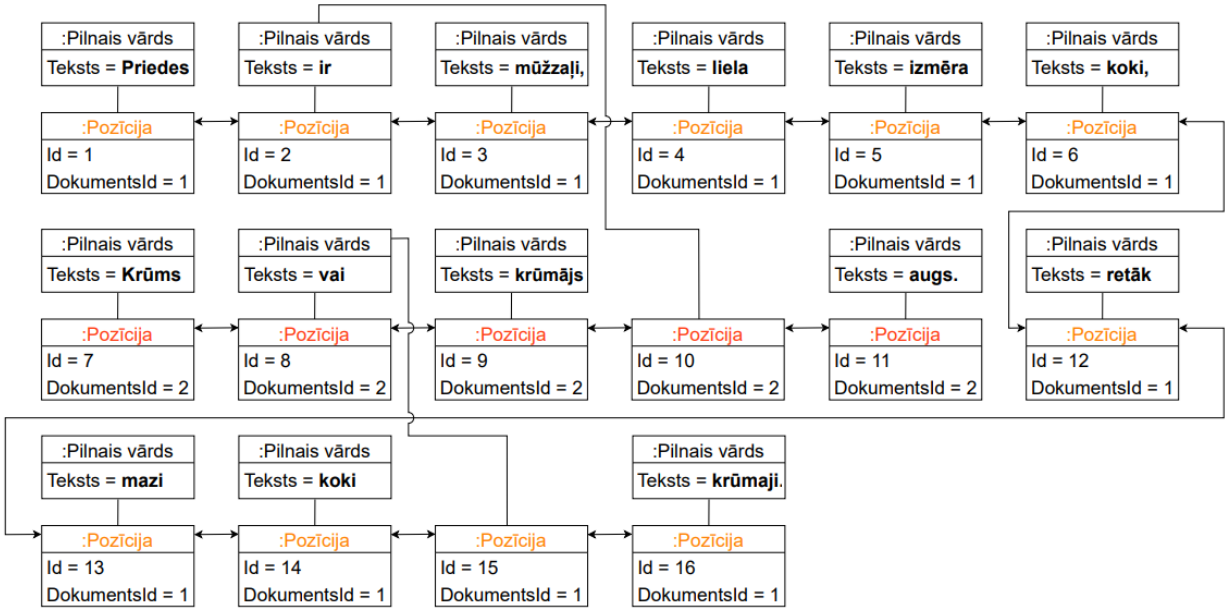
Attēlā 1.6. visas tabulas ir tādas pašas kā pamatvariantā, bet izmaiņas ir tabulā "Pozīcija". Šajā tabulā tiek pievienotas divas papildus references, kas katru pozīciju saistīs ar iepriekšējo un nākamo pozīciju atbilstoši tam kā tas ir dokumentā. Ja blakus esošās pozīcijas tiek saistītas ar referencēm, tad vairs nav vajadzība tām fiziski atrasties blakus pēc "Id" lauku vērtībām, lai varētu efektīvi iegūt blakus esošās pozīcijas, oriģinālu reprezentējošu teksta fragmentu.

Piemērs

Piemērā izmantotie dokumenti:
Dokuments 1= Priedes ir mūžzaļi, liela izmēra koki, retāk mazi koki vai krūmāji.
Dokuments 2= Krūms vai krūmājs ir augs.

Attēlojumā (attēls 1.7.) dokumenta 1 pozīcijas apzīmētas izmantojot oranžu krāsu, bet dokumenta 2 pozīcijas apzīmētas izmantojot sarkanu krāsu. Attēlotajā situācijā skatoties pēc "Id" lauku vērtībām dokuments 2 ir iekš dokumenta 1, bet to robežas ir noteiktas vietās kur "iepriekšējais vārds", vai "nākamais vārds" references lauki ir null. Šāda situācija varētu rasties kad sākotnēji tika ievietoti no pirmā dokumenta pirmie seši vārdi un viss otrais dokuments, bet

pēc tam dokuments 1 ir papildināts ar atlikušajiem 5 vārdiem. Visas attēlā 1.7 attēlotās saites starp tabulas "Pozīcija" instancēm pieder klašu diagrammas (attēls 1.6.) asociācijām "Nākamais vārds", "Iepriekšējais vārds".



1.7. att. **Dokumentu indeksēšanas datu struktūras alternatīvā varianta instanču diagramma**

1.2.1. Ieguvums atjaunināšanai

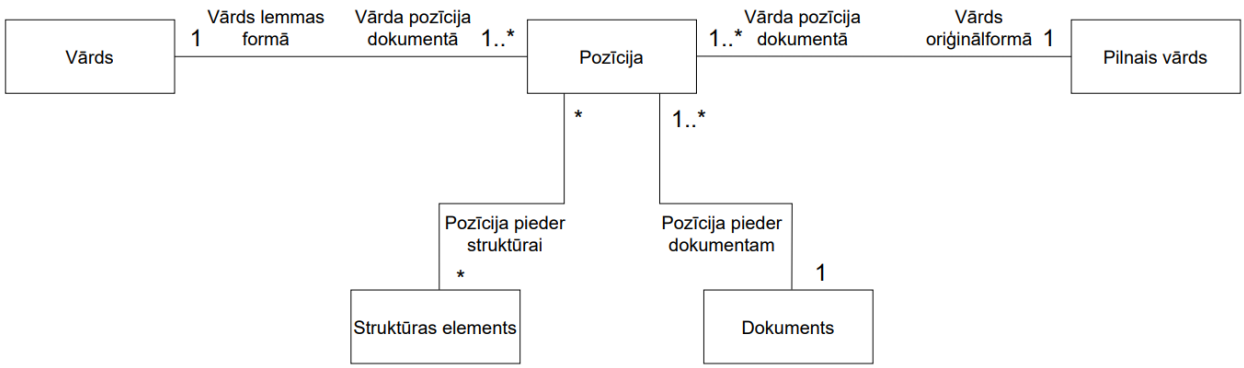
Veicot dokumenta atjaunināšanu ja dokumenta izmērs vārdu skaitā ir palielinājies atbilstoši pamatvarianta aprakstītajam atjaunošanas algoritmam, tad tabulā "Pozīcija" tas tiek ievietots beigās, un katrai pozīcijai kas tiek ievietota ir jāatrod vārds lemmas formā, ir jāapstaigā pozīciju saraksts līdz beigām, un tad iepriekšējo pozīciju saraksta pozīciju var savienot ar jauno izmantojot "Nākamā pozīcija" references lauku tādējādi atjaunojot pozīciju sarakstu. Šādi tas ir jāveic visiem dokumenta vārdiem.

Savukārt šajā variantā kur blakus esošās pozīcijas ir sasaistītas ar referencēm tās var glabāt arī tā, ka tās fiziski pēc "Id" lauku vērtībām nav blakus, un līdz ar to arī ja dokumenta izmērs vārdu skaitā ir palielinājies (salīdzinot ar pirms pēdējās modificēšanas stāvokli), tad pievienot pozīciju saraksta beigās vajadzēs tikai to daļu, kas ir tas papildinājums dokumentam un tad tikai

šai daļai vajadzēs veikt jaunas pozīcijas pievienošanas procedūru. Līdz ar to arī ievērojami samazinot skaitļošanas apjomu.

1.3. Alternatīvais variants dokumenta vārdu pozīcijām

Dokumentu meklēšanas procesā būtiska loma ir frāzēm, jo visbiežāk dokumentu meklēšanā tiek iesaistīts teksta fragments, kas ir no vairākiem vārdiem sastāvoša frāze. Kad tiek veikta dokumentu meklēšana pēc frāzes, tad katram frāzē esošajam vārdam tiek iegūts masīvs ar tā visām pozīcijām visos dokumentos, lai kad katram vārdam šāds masīvs ir iegūts tos varētu salīdzināt un atrast vietas kur pozīcijas atrodas blakus veidojot meklēto frāzi. Tātad dokumentu meklēšanas ātruma uzlabošanai būtiski ir pēc iespējas ātrāk no datu bāzes iegūt šos vārdu pozīciju masīvus.



1.8. att. **Dokumentu indeksēšanas datu struktūras alternatīvā vārdu pozīciju varianta klašu diagramma**

Šis variants ir veidots uz pamatvarianta bāzes, bet "Alternatīvais variants dokumentu glabāšanai" uzlabojumi šeit netiek demonstrēti, jo šī varianta fokuss ir uz meklēšanu nevis atjaunināšanu. Programmas realizācijā ir iespējams sakombinēt principus pēc vajadzības no aprakstītajām variantu alternatīvām.

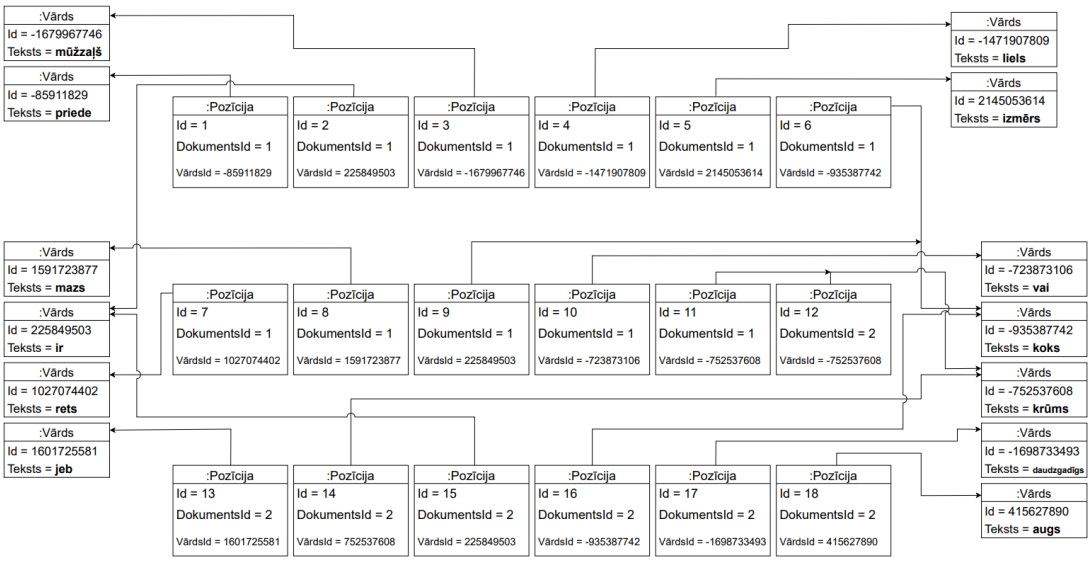
Šajā variantā tabulas "Pozīcija" references lauks "Nākamā pozīcija" ir aizvietots ar kardinalitāti "1..*", kas nozīmē to, ka tagad katra pozīcija ar vārdu lemmas formā būs saistīta nevis pozīciju sarakstā kā iepriekšējos variantos, bet pa tiešo ar tabulas "Vārds" ierakstu izmantojot referenci.

Piemērs

Piemērā izmantotie dokumenti:

Dokuments 1= Priedes ir mūžzaļi, liela izmēra koki, retāk mazi koki vai krūmāji.

Dokuments 2= Krūms jeb krūmājs ir kokains, daudzgadīgs augs.



1.9. att. **Dokumentu indeksēšanas datu struktūras alternatīvā vārdu pozīciju varianta instanču diagramma**

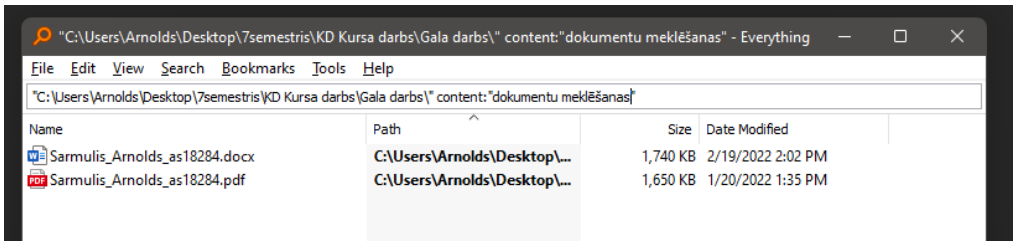
1.3.1. Ieguvums frāžu meklēšanai

Katrai pozīcijai ir lauks "VārdsId" šajā laukā tiek glabāta tabulas "Vārds" primārā atslēga, kas šajā gadījumā ir arī no vārda lemmas formas teksta ar jaucējfunkciju iegūts vesels skaitlis. Meklētās frāzes vārdiem no to lemmas formām ar to pašu jaucējfunkciju tiek iegūti veseli skaitļi un tad atkarībā no realizācijas atlasot pēc šī skaitļa ir iespējams ar vienu vai vairākām tabulas "Pozīcija" apstaigāšanām iegūt visiem vārdiem visas to pozīcijas visos dokumentos. Tātad katrai vienai pozīcijai var noteikt vai tā pieder attiecīgajai lemmas formai neveicot nekādas papildus datu sasaistes darbības un līdz ar to kopumā ar mazāk darbībām varēs iegūt visas nepieciešamās pozīcijas.

1.4. Citu autoru veidoti meklēšanas rīki

1.4.1. Rīks “Everything”

Failu meklēšanas rīks priekš windows operētājsistēmas, kas ir spējīgs ļoti ātri atrast direktorijas un dažādu tipu failus pēc to nosaukuma. Lietotāja saskarne ir ērta un vienkārša tāda, ka pirmo reizi atverot programmu uzreiz ir skaidrs ko un kā var tajā paveikt, kā arī veicot dubultklikšķi uz meklēšanas rezultāta ir iespējams to atvērt. Rakstot meklēšanas vaicājumu uzreiz parādās rezultāti, nav jāveic nekādi papildus klikšķi. Sniedz dažādas iespējas konfigurēt meklēšanu, izvēlēties failu tipu, meklēšanā lietot regulārās izteiksmes un vēl citas iespējas. Kad programma tiek pirmo reizi palaista tā izveido indeksa struktūru kurā tiek iekļauti vārdi jebkurai datnei. Šis rīks ir pieejams bezmaksas komerciālai lietošanai. [14]



1.10. att. **Lietotāja saskarne ar redzamiem meklēšanas rezultātiem rīkā “Everything”**

Pozitīvie aspekti [14]:

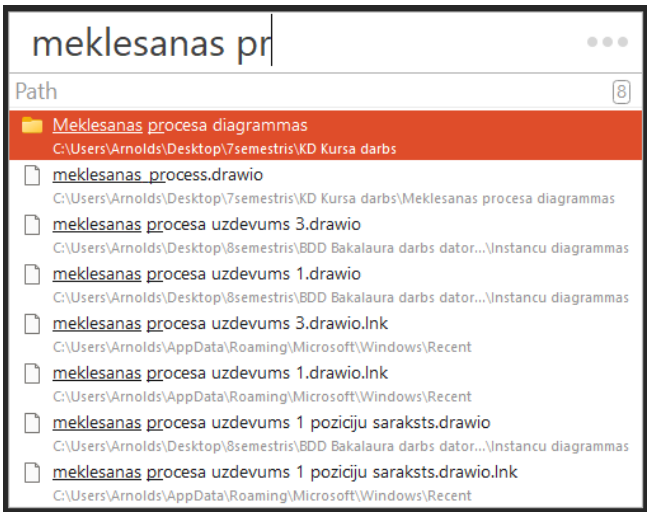
- Meklēšanu veic iepriekš izveidotā indeksā;
- Sniedz iespēju veikt meklēšanu uzreiz daudzos datortīklā pieslēgtos datoros, ja arī tajos ir ieinstalēta šī programma;
- Vienkārša, labi strukturēta un ērti lietojama grafiskā lietotāja saskarne.

Negatīvie aspekti:

- Meklējot datnes pēc frāzes spēj atgriezt rezultātos tikai datnes, kas satur meklēto frāzi, neatgriež atrastās instances datņu saturā, kā arī nekādu citu informāciju par tām;
- Nav spējīgs meklēt datnes pēc frāzēm vārdu atbilstības noteikšanā izmantojot vārda lemmas formu.

1.4.2. Rīks “Listary”

Meklēšanas rīks, kas ne tikai spēj meklēt un atvērt datnes, bet ir spējīgs arī kopēt un pārvietot failus lietotājam ērtā veidā. Rīkā ir iespēja izvēlēties failu pārvaldnieka, vai arī failu atvēršanas režīmu, kur pirmais piedāvā izvērstākas failu pārvaldīšanas iespējas, bet otrs meklēšanas. Atrodoties jebkurā vietā datorā iespējams divreiz uzspiest taustiņu “ctrl” un rīks uzreiz atvērs meklēšanas logu, kā arī atrodoties jebkurā direktorijā iespējams sākt rakstīt tekstu, kas uzreiz atvērs rīka meklēšanas logu [15].



1.11. att. **Lietotāja saskarne ar redzamiem meklēšanas rezultātiem rīkā “Listary”**

Pozitīvie aspekti [15]:

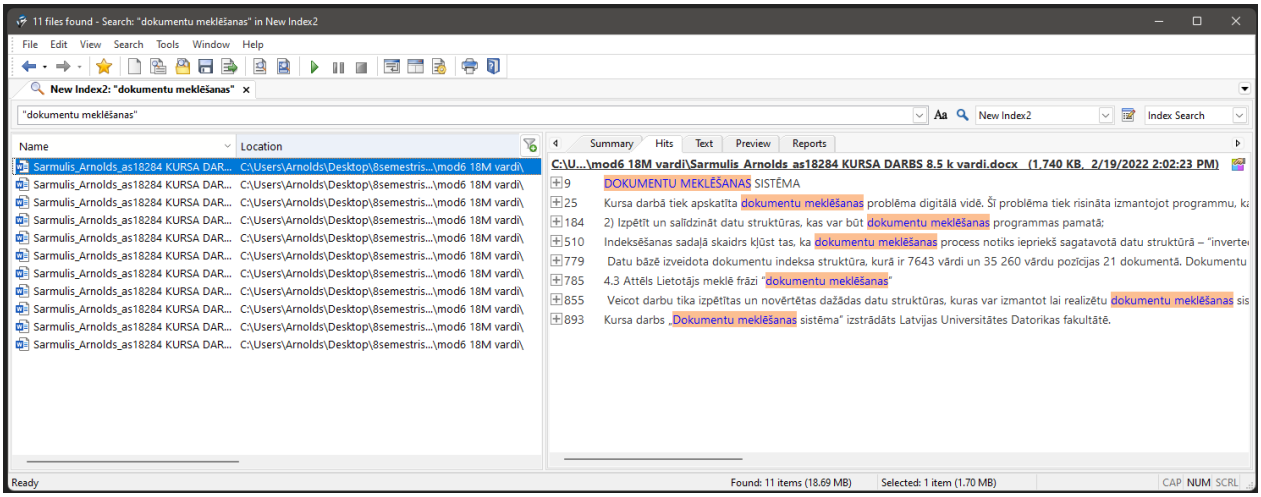
- Ērta programmas lietojamība, programmu var palaist no jebkuras vietas datorā uzspiežot divreiz tastatūras taustiņu, tāpat arī citas funkcijas var ērti lietot ar dažādām taustiņu kombinācijām;
- Sniedz iespēju lietot simbolu “*”, kas meklēšanas tekstā nozīmē jebkādi simboli un jebkādā daudzumā, kā arī citus līdzīgus simbolus ar šāda rakstura funkcijām;
- Meklēšanu veic iepriekš indeksētās datu struktūras;
- Uzspiežot taustiņu “Tab” tiek veikta automātiskās teksta pabeigšanas funkcija.

Negatīvie aspekti:

- Nav iespējas mainīt meklēšanas rezultātu sakārtojumu;
- Nav iespējas meklēt datnes pēc to satura.

1.4.3. Rīks “Agent Ransack”

Rīka izstrādātāju kompānija specializējas tieši uz augstas veiktspējas meklēšanas risinājumiem. Šis rīks ir integrēta meklēšanas platforma, kas apvieno sevī plašu meklēšanas funkcionalitāti, tai pat laikā nodrošinot ātru meklēšanas funkciju izpildi. Lietotājam ir arī iespēja manuāli izveidot indeksa struktūru un veikt meklēšanu tajā, bet meklēšanu var veikt arī neizmantojot indeksu. Ietver arī meklēšanas rezultātu eksportēšanas funkcionalitāti. Rīks ir bezmaksas gan personālai, gan komerciālai lietošanai [16].



1.12. att. **Veikta meklēšana izmantojot indeksu rīkā “Agent Ransack”**

Pozitīvie aspekti:

- Ļoti plašas meklēšanas iespējas, kuras ir iespējams dažādi kombinēt, dažas no tām ir: meklēšana ar boolean tipa operatoriem; regulārās izteiksmes; “Fuzzy” meklēšana; meklēšana izmantot datņu kategorijas un metadatus; meklēšana indeksā;
- Labi strukturēti meklēšanas rezultāti, tiek attēlots atrastās frāzes konteksts teksta priekšskatījuma formā, turpat lietotāja grafiskajā saskarnē pie meklēšanas rezultātiem ir iespējams arī apskatīt visa dokumenta priekšskatījumu;
- Efektīva meklēšanas funkcionalitātes realizācija izmantojot daudz pavedienu pieeju (Veicot meklēšanu tika izmantoti visi procesora 16 izpildes pavedieni).

Negatīvie aspekti:

- Ja meklēšana tiek veikta neizmantojot indeksu, tad tā strādā būtiski lēnāk un uz lieliem apjomiem strādā lēni, kā arī uz apjomīgiem dokumentiem potenciāli vispār neatrod rezultātus. (Tajā pašā dokumentu kopā veicot meklēšanu ar indeksu mazāk kā sekundes laikā atrada visas frāzes visos dokumentos, bet veicot meklēšanu neizmantojot indeksu bija jāgaida 20 sekundes, un atgrieza tikai kļūdas paziņojumus);
- Izmantojot meklēšanu ar indeksu nav spējīgs noteikt vārdu atbilstību izmantojot vārdu lemmas formu.

1.4.4. Autora aprakstītā rīka salīdzinājums ar citu autoru rīkiem

1.2. tabula

Meklēšanas rīku būtiskāko kopīgo un atšķirīgo īpašību apkopojums

Citu autoru rīks	Kopīgais ar autora aprakstīto rīku	Atšķirīgais ar autora aprakstīto rīku
"Everything"	Izmanto indeksēšanas pieeju meklēšanai, sniedz iespēju sakārtot atgrieztos rezultātus, meklēšanas rezultātos attēlo dokumenta metadatus.	Rīks "Everything" ir spējīgs tikai noteikt vai dokuments satur meklēto frāzi vai nesatur, bet autora aprakstītais risinājums atrod un atgriež rezultātos visas frāzes visos dokumentos, kas ietilpst meklēšanas apgabalā.
"Listary"	Izmanto indeksēšanu meklēšanā.	Rīks "Listary" nav spējīgs meklēt dokumentus pēc to satura, rezultātos neatgriež dokumenta metadatu informāciju. Autora aprakstītais risinājums sniedz dažādas funkcijas meklējot dokumentus pēc to satura, atgriež rezultātos dokumenta metadatus.
"Agent Ransack"	Abi rīki ir spējīgi veikt efektīvu meklēšanu indeksa struktūrā.	Rīks "Agent Ransack" ir platforma ar ļoti plašu meklēšanas funkcionalitāti un

	Atgriež saturīgus rezultātus visām atrastajām frāzēm ar teksta priekšskatījumu un tajā iezīmētu atrasto frāzi. Sniedz iespēju iegūt visa dokumenta priekšskatījumu. Rezultātos atgriež dokumentu metadatu informāciju.	piedāvā iespēju arī meklēt dokumentus neizmantojot indeksu. Veicot meklēšanu indeksā nav spējīgs meklēt frāzes nosakot atbilstību izmantojot lemmas formu, bet autora aprakstītais risinājums sniedz šādu iespēju. Autora aprakstītais risinājums sniedz iespēju atrastās frāzes sakārtot pēc līdzības pakāpes ar meklēto frāzi, vai arī pēc dokumenta metadatiem.

1.5. Nodaļas nobeiguma secinājumi

Visas nodaļā aprakstītās un izanalizētās datu struktūras ir veidotas uz dokumentu indeksēšanas pamatprincipa bāzes, kas nozīmē veidot vārdu-pozīciju kartējumu, ko var izmantot lai atrastu jebkura noindeksētā dokumenta jebkuru vārdu. Tātad dokumentu indeksu var veidot izmantojot jebkuru šajā nodaļā aprakstīto datu struktūru. Katrai no datu struktūrām ir savi pozitīvie un negatīvie aspekti un lai patiesi varētu noteikt kura struktūra ir efektīvākā ir jāveic praktisks pētījums, kas sevī ietver dažādas dokumentu meklēšanas funkcionalitātes realizācijas šajās struktūrās, kas tiek izpildīts 3. nodaļā.

2. DATU STRUKTŪRU UZGLABĀŠANAS UN UZTURĒŠANAS RISINĀJUMI

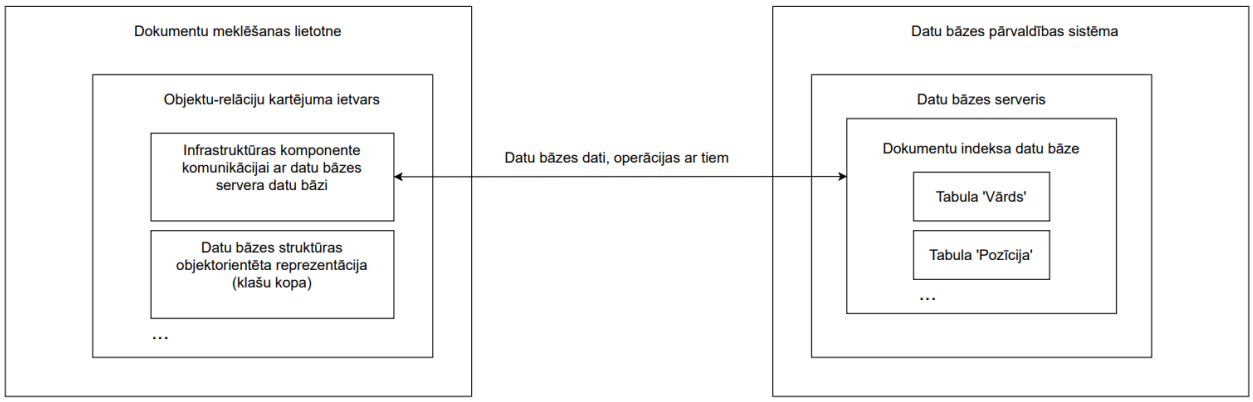
Efektīvas meklēšanas funkcionalitātes realizēšanai liela nozīme ir tam kādas datu struktūras ir tās pamatā, kā operācijas ar tām tiek implementētas un arī kā un kur šis datu struktūras tiek uzglabātas. Datu struktūras ir iespējams tieši glabāt failos izmantojot XML, CSV vai tam līdzīgu metodiku. Šāda metodika ir mazāk efektīva gan datu uzturēšanā, gan pārlūkošanā un labāk ir piemērota datu transportēšanai. Funkcijām kurās kritiski svarīgs ir datu apstrādes ātrums un ir jāveic ar datu uzturēšanu saistītas netriviālas operācijas, daudz labākas un plašākas iespējas sniedz relāciju datu bāzu pārvaldības sistēmas.

Datu bāzu pārvaldības sistēmas nodrošina dažādas ar datu uzturēšanu saistītas funkcijas kā datu ievietošanu, dzēšanu, pārlūkošanu, kārtošanu, indeksēšanu u.tml., kas ir implementētas izmantojot ļoti efektīvas metodes. Relāciju datu bāzes, kas tiek glabātas DBPS var savienot ar izstrādes vidi/programmu izmantojot objekta-relācijas kartējumu, kas ļauj izstrādes vidē operēt ar datu bāzes tabulām kā objektiem, kuru pamatā ir klases.

2.1. Objektu-relāciju kartējuma ietvars

2.1.1. Sistēmas arhitektūra ORM lietojuma kontekstā

Diagramma (attēls 2.1.) attēlo izvietojumu tikai tām sistēmas būtiskākajām komponentēm, kas ir saistītas ar datu struktūru uzglabāšanu izmantojot DBPS un ORM. Izvietojums tiek attēlots hierarhiski izejot no programmas un DBPS.



2.1. att. **Sistēmas arhitektūra datu glabāšanas kontekstā**

Objektorientētā programmēšanas pieeja un relāciju datu bāzes tiek veidotas pēc dažādām paradigmām, un viens no galvenajiem ORM uzdevumiem ir izveidot objektorientētu datu bāzes reprezentāciju tā, ka ar to var operēt objektorientētā programmēšanas valodā. Katrai datu bāzei tiek izveidots tai atbilstošs ORM modelis, kurš sastāv no klasēm, kas reprezentē datu bāzes tabulas un to struktūru, kā arī tiek definēts datu bāzes konteksts kuru izmantojot var piekļūt datiem un veikt dažādas operācijas ar tiem. Līdz ar to izmantojot ORM modeli ir iespējams veikt dažādas operācijas ar datu bāzi neizmantojot SQL vaicājumus, bet vaicājumus pierakstot programmēšanas valodā un tie tiek automātiski konvertēti uz SQL vaicājumiem un izpildīti datu bāzē. Būtisks ieguvums no šī ir, ka to pašu var paveikt ar īsāku pierakstu un tajā pašā programmēšanas valodā, kas ir programmas pamatā, kas savukārt programmkodu padara vieglāk uztveramu.

Veidojot objektorientētu relāciju datu bāzes reprezentāciju katrai tabulai tiek izveidota atbilstoša klase ar tādu pašu nosaukumu, un katram tabulas laukam tiek izveidots lauks ar tādu pašu nosaukumu un lauka datu tips tiek konvertēts tā lai tas pēc iespējas labāk atbilstu datu bāzes tipam. Datu tipu konvertēšanai ir jāpievērš īpaša uzmanība, jo starp datu bāzes un OOP programmēšanas valodu tipiem pastāv daudz neatbilstības, kuru nepareiza apstrāde var novest pie kļūdām programmas darbībā.

2.1.2. Sintakses piemēri

ORM programmētāja lietotajā programmēšanas valodā pierakstīto operāciju konvertē uz SQL, tā var būt jebkurā no programmēšanas valodām, kurās ir ORM atbalsts. No sintakses viedokļa tāpat kā programmēšanas valodā tiek operēts ar dažādām datu kolekcijām, kā masīvi vai iebūvētie saistītie saraksti var operēt ar datu bāzes objektiem. Līdz ar to no sintaktiski tiek panākta vienota pieeja dažādām programmēšanas valodām.

Tiek izvērtētas ORM sniegtās iespējas salīdzinot sintaksi starp vienādām operācijām ko var veikt SQL un programmēšanas valodā. ORM pusei tiek lietots C# Lambda LINQ, kas programmēšanas valodā dod iespēju izmantojot to pašu sintaksi operēt ar dažādiem objektiem. Tabulā 2.1. "SQL" kolonnā tiek parādīts programmkods, kas automātiski tiek uzģenerēts no kolonnas "C# Lambda LINQ" atbilstošā programmkoda un tiek izpildīts datu bāzē priekš programmētāja.

2.1. tabula

Sintakses piemēri datu bāzes operācijām C# Lambda LINQ un SQL [1]

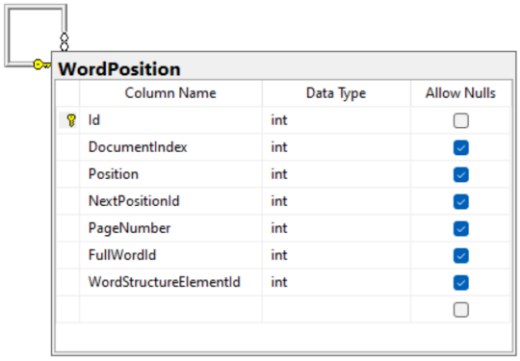
Nr	C# Lambda LINQ	SQL
1	context.PilnaisVards.Add(new PilnaisVards { Id = 55, VardaSaturs = "mežs" });	INSERT INTO [dbo].[PilnaisVards] ([Id] ,[VardaSaturs]) VALUES (55 ,'mežs')
2	var pv = context. PilnaisVards.FirstOrDefault(x => "mežs".Equals(x.VardaSaturs));	SELECT TOP (1) [Extent1].[Id] AS Id, [Extent1].[VardaSaturs] AS VardaSaturs FROM [dbo].[PilnaisVards] AS [Extent1] WHERE 'mežs' = [Extent1].[VardaSaturs]
3	var pv = context.PilnaisVards.Where(x => "mežs".Equals(x.VardaSaturs));	SELECT [Extent1].[Id] AS Id, [Extent1].[VardaSaturs] AS VardaSaturs FROM [dbo].[PilnaisVards] AS [Extent1]

		WHERE 'mežs' = [Extent1].[VardaSaturs]

2.1.3. Tabulas kartējuma piemērs

Lai iegūtu ORM ietvara ģenerētu datu bāzes tabulas reprezentāciju ir izveidota dokumentu indeksa datu bāze iekš "Microsoft SQL Server". Tālāk tiek izmantota "Visual studio" iebūvētā funkcija ORM modeļa izveidošanai, kurai tiek norādīti serveris, datu bāze un tad attiecīgi šai datu bāzei tiek uzģenerēts ORM modelis.

Attēls 2.2. ir iegūts izmantojot "SQL Server Management Studio" ir uzģenerēta datu bāzes tabulas "WordPosition" vizuālā reprezentācija, kas raksturo datu bāzē eksistējošo tabulu. Attēls 2.3. ir iegūts no "Visual Studio" projekta no kura tika ģenerēts ORM modelis, un tiek parādīta tā pati tabula objektorientētā reprezentācijā C# valodā.



2.2. att. **Datu bāzes tabula "WordPosition"**

```
public partial class WordPosition
{
    1 reference
    public int Id { get; set; }
    1 reference
    public Nullable<int> DocumentIndex { get; set; }
    1 reference
    public Nullable<int> Position { get; set; }
    0 references
    public Nullable<int> NextPositionId { get; set; }
    1 reference
    public Nullable<int> PageNumber { get; set; }
    1 reference
    public Nullable<int> FullWordId { get; set; }
    0 references
    public Nullable<int> WordStructureElementId { get; set; }
    0 references
    public virtual FullWord FullWord { get; set; }
    1 reference
    public virtual TextStructureElement WordStructureElement { get; set; }
    1 reference
    public virtual WordPosition NextPosition { get; set; }
}
```

2.3. att. **Datu bāzes tabulas "WordPosition" reprezentācija C# programmēšanas valodā**

ORM ietvaru pozitīvās īpašības [11]:

- Sniedz iespēju ērti un īsi pierakstīt vienkāršas operācijas, nav nepieciešamība rakstīt SQL kodu, kas bieži atkārtojas;
- Programmētājam ir iespēja veikt SQL vaicājumus un operācijas izmantojot projektā lietoto programmēšanas valodu, nav nepieciešamība pašam rakstīt SQL;
- ORM modeli un savienojumu ar datu bāzi var automātiski uzģenerēt un atjaunot atbilstoši datu bāzes struktūrai;
- Pareizi lietojot samazina programmatūras izstrādes laiku un izmaksas.

ORM ietvaru negatīvās īpašības:

- Lai gan strādāt ar ORM lielākoties ir viegli, programmētājam ir jāzina tā darbības pamatprincipi;
- Specifiskām un apjomīgām operācijām ar datiem var tikt samazināta izpildes efektivitāte.

Secinājumi

Uzģenerēt ORM modeli dokumenta indeksa datu bāzei izmantojot "Visual Studio" var diezgan ērti un ātri. Ņemot vērā to, ka pāris minūšu laikā automatizēti ir iespējams iegūt darbam gatavu ORM ietvara komponenti, kura atvieglo darbu ar datu bāzi ir nopietni vērts apsvērt šī rīka izmantošanu dokumentu meklēšanas programmas izstrādē.

2.2. Datu bāzu pārvaldības sistēmas

DBPS nodrošina plašu datu bāzu apstrādes funkcionalitāti un daļa no šīs funkcionalitātes var būt ļoti būtiska datu apstrādes laika uzlabošanā, kā piemēram indeksi. Lietojot integrēto datu bāzu izstrādes vidi, kā piemēram "SQL Server Management Studio", ir plašas iespējas ne tikai izveidot, pārvaldīt un uzturēt datu bāzes, bet arī iegūt analītiska rakstura informāciju par dažādiem datu bāzēm piederošiem objektiem.

2.2.1. Tabulu indeksēšana

Veicot vienkāršu lineāru meklēšanu paredzamais sliktākais meklēšanas laiks ir O(n), bet ja meklēšana tiek veikta izmantojot indeksu, tad sliktākais meklēšanas laiks ir O(log n), kur n ir datu kolekcijas elementu skaits. Izmantojot lineārās meklēšanas metodi katrā iterācijā apgabals kurā elements vēl ir jāmeklē tiek samazināts par vienu vienību, bet izmantojot B plus koku, ko lieto indeksi katrā iterācijā apgabals kurā vēl būs jāmeklē tiek samazināts aptuveni vidēji uz pusi.

Tātad indeksu lietošana dokumentu meklēšanas sistēmas datu struktūrām ir fundamentāli svarīga, jo ir paredzēts, ka šīs datu struktūras potenciāli būs ļoti lietas, kas var būt arī vairāki miljoni ieraksti un jāņem vērā arī tas, ka ierakstus vajadzēs ne tikai atrast, bet arī apstrādāt un sagatavot atgriešanai, kas nozīmē ka šajā situācijā indeksu lietošana ir obligāti nepieciešama.

Ieteikumi indeksu veidošanai [2, 3]:

- Izvēloties indeksējamo kolonnu ņemt vērā kolonnas datu tipu, indeksi strādās efektīvāk ja kolonnas datu tips būs vesels skaitlis un ja vērtības šajā kolonnā būs tikai unikālas un nebūs null. Arī tad ja indekss jāveido kolonai kurai nav skaitļa datu tips, tad var atmaksāties izveidot papildus kolonu kurā izveido šīs vērtības skaitļa reprezentāciju, piemēram reālo skaitli var pārveidot uz veselo, vai arī simbolu virknes ar jaucējfunkciju var pārveidot uz skaitli un tad attiecīgi indeksu veido kolonnai, kas reprezentē oriģinālās kolonnas vērtību;
- Veidot indeksus tā lai tajos būtu iekļautas pēc iespējas mazāk kolonnas un izmērā mazāki datu tipi, jo mazāki būs indeksi, jo ātrāk tie strādās. Vispiemērotākie SQL datu tipi indeksiem ir: “smallint”, “int” un “bigint”. Tāpat arī vienai tabulai veidot pēc iespējas mazāk indeksus, jo katrs papildus indekss tabulai paildzina datu atjaunošanas laiku;
- Izvēloties indeksējamo kolonnu pēc iespējas izvēlēties kolonnas kuru vērtības tiks retāk mainītas, jo ja vērtība tiek mainīta kolonnai kurai nav indekss, tad atjaunošanas ātrdarbību tas neietekmēs, bet mainot vērtību kolonnai kurai ir indekss ātrdarbība tiks negatīvi ietekmēta;
- Ja ir zināms ka kolonnas vērtības ir unikālas, tad veidojot indeksu deklarēt to kā unikālu, tad DBPS to attiecīgi apstrādās un tas strādās ātrāk. Bet ja ir zināms, ka kolonnā ir ļoti

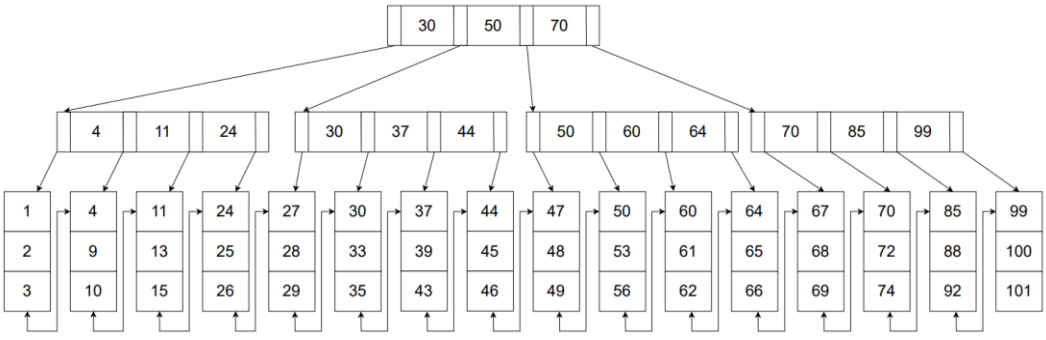
daudz vērtību dublikāti, tad izvairīties indeksēt šādu kolonnu, jo tas ievērojami samazinās indeksa ātrdarbību;

- Izvairīties indeksēt tabulas par kurām ir zināms, ka tās izmērā būs mazas, jo mazām tabulām vienkārša meklēšana bez indeksa var strādāt pat nedaudz ātrāk un atjaunot tabulu indekss tāpat būs jāuztur paildzinot tabulas datu atjaunošanas laiku;
- Lietot indeksus ar filtru kolonnām kurām ir labi identificējamas vērtību apakškopas, tas var uzlabot indeksa ātrdarbību un samazināt tā uzturēšanas izmaksas;
- Netriviālu SQL skatu, kas satur tabulu datu sasaistes un filtrēšanas, indeksēšana var ievērojami uzlabot ātrdarbību;
- Indeksus veidot kolonnām, kas tiek visbiežāk lietotas meklēšanai un programmu strukturēt tā lai tā veicot meklēšanas izmantotu tikai indeksētas kolonnas.

B plus koks

Binārās meklēšanas koks, kas pēc principa ir līdzīgs B plus kokam, tiek uzglabāts datora primārajā atmiņā, savukārt B plus koks tiek uzglabāts datora sekundārajā atmiņā un tas tiek veidots tā lai varētu efektīvāk izgūt atmiņu. Datu bāzu tabulu indeksēšanai pamatā tiek lietoti B plus koki, kas ir balancēts meklēšanas koks, kas nozīmē ka visām koka lapām ir vienāds dziļums.

Koka sakne un mezgli satur pointerus uz to bērniem, un katrs mezgls satur identifikatoru, kuri ir sakārtoti un tādējādi var orientēties kokā, bet lapās tiek glabāti pointeri, kas norāda uz noteiktu atmiņas apgabalu un dod iespēju piekļūt meklētajai informācijai, kas var būt noteiktas tabulas ieraksti.



2.4 att. **B plus koka attēlojums**

2.2.2. Statiskās procedūras

SQL statiskās procedūras ir izpildāmi SQL koda fragmenti, kas tiek noglabāti konkrētā datu bāzē tā, ka tos ir iespējams izpildīt, padot ieejas parametrus, un arī saņemt atgriezto vērtību. Katra procedūra veic kādu noteiktu uzdevumu vai uzdevumus ar datu bāzes objektiem, tā var modificēt, ievietot vai dzēst datu bāzes objektus, vai izgūt to vērtības. Procedūras atgriežamā vērtība vienmēr ir vesels skaitlis un pēc noklusējuma tā atgriež pazīmi par to vai procedūras izpildes laikā bija kļūda vai nē, kur 0 nozīmē, ka kļūda nav konstatēta. Atgriežamās vērtības vietā var tikt definēti arī izejas parametri kurus var izmantot tālāk programmā vai SQL skriptā. SQL statiskās procedūras var tikt lietotas kā alternatīva ORM ietvara operācijām ar datu bāzi. [4, 5]

Statisko SQL procedūru lietošanas ieguvumi [4, 5]:

- Sniedz iespēju atkalizmantot SQL kodu, procedūras ietvaros ir iespējams arī izsaukt citas procedūras, kas dod iespēju kodu veidot vieglāk uztveramu un labāk strukturētu, veidojot modulāru koda struktūru;
- Nodrošina ātrāku SQL koda izpildi, jo atkārtoti izpildot procedūru tiek izmantots uzglabāts vaicājuma izpildes plāns un tas nav atkārtoti jāveido;
- Procedūras tiek glabātas tieši uz SQL servera izpildāmā formā (ir nokompilēts), un tās var tikt izpildītas ātrāk nekā SQL kods kurš tiek sūtīts uz serveri, procedūras gadījumā ir jāsūta tikai procedūras izsaukums ar ieejas parametriem, kas arī samazina sūtīto datu apjomu;
- Nodrošina labāku drošību un aizsargā pret SQL injekcijām, ja tās tiek lietotas parametrizēti.

Secinājumi

Indeksu lietošana var ļoti būtiski ietekmēt meklēšanas laiku datu bāzes ierakstos, bet lai varētu pilnīgi izmantot indeksēšanas sniegto potenciālu ir nepieciešams tos lietot pareizi

programmas vajadzību kontekstā tā lai realizētie indeksi pēc iespējas vairāk atbilstu to lietošanas ieteikumiem. Datu bāzes operācijām, kas tiek bieži atkārtotas ir vērtīgi lietot statiskās procedūras, jo to lietošana ne tikai uzlabo koda atkalizmantojamību, bet arī operāciju izpildes laiku.

2.3. Dokumentu indeksa struktūras izveidošanas praktiskais pētījums

Dokumentu meklēšanas programmai ļoti būtisks ir ne tikai dokumentu meklēšanas ātrums, bet arī indeksa datu struktūras izveidošanas un atjaunošanas ātrums. Tas ir nozīmīgs situācijās kur lietotājs nav atradis meklēto tāpēc, ka datu struktūra nav atjaunota un neatbilst aktuālajam dokumentu stāvoklim, un šādā situācijā ir kritiski svarīgi spēt ātri atjaunot datu struktūru, lai lietotājs spētu ātri atrast meklēto. Tāpat svarīgi ir pēc iespējas ātrāk arī izveidot dokumentu indeksa struktūru, lai izveidošanas process pēc iespējas īsāku brīdi noslogotu datora resursus. Svarīgi ātri izveidot struktūru var būt arī situācijās, kur lietotājs vēlas izveidot struktūru no jauna un uzreiz viņā meklēt, kas nozīmē, ka viņam nāksies gaidīt kamēr tā tiks izveidota līdz varēs veikt meklēšanu.

2.3.1. Uzdevums

Lai varētu noteikt kā var ātrāk veikt operācijas ar datu bāzes datiem tieši dokumentu indeksa datu struktūrai ir nepieciešams norealizēt, notestēt un salīdzināt dažādus variantus, kuros tiek veikts kāds dokumentu datu struktūras atjaunošanas uzdevums. Galvenais uzdevums ir noteikt ātrāko variantu dokumentu indeksa struktūras izveidošanai.

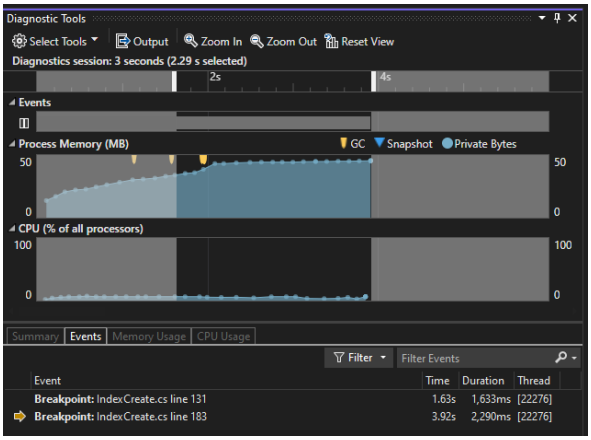
Tiks salīdzināti divi veidi kā var izveidot dokumentu indeksa struktūru. Pirmajā variantā tiks uzprogrammēta dokumentu indeksa struktūras izveidošana izmantojot ORM pieeju, bet otrajā variantā tas pats tiks realizēts izmantojot SQL “Stored Procedures”. Abiem variantiem tiks apskatītas dažādas modifikācijas, lai varētu noteikt kura modifikācija ir efektīvākā katrā variantā, un tad attiecīgi izsecināt kurš ir viss ātrākais veids kā izveidot dokumentu indeksa struktūru. Pētījums tiek realizēts tieši uz dokumentu izveidošanas uzdevuma pamata, bet tos pašus principus var pielietot arī citiem līdzīgiem uzdevumiem.

2.3.2. Izmantotie rīki un metodes

Praktiskais pētījums programmas daļā tiek realizēts izmantojot C# programmēšanas valodu un integrēto izstrādes rīku “Visual Studio”, datu bāzei tiek izmantots SQL un “SQL Server Management Studio”. Uzsākot praktisko pētījumu dokumentu indeksa datu bāze ir realizēta iekš “Microsoft SQL Server”. ORM ietvara veiktspējas testēšanai tiek lietots “Entity Framework”.

Funkciju izpildes laiks tiks mērīts izmantojot “Visual Studio” rīka atkļūdošanas funkcionalitāti, uzstādot pirmo “brakepoint” izveidošanas funkcijas sākumā, pirms tiek ielasīts pirmais vārds no pirmā dokumenta, un otrais “brakepoint” tiek uzstādīts uz to rindu, kur funkcija ir pabeigusi savu darbu, un tajā brīdī datu bāzē ir pilnīgi izveidota dokumentu indeksa datu struktūra, attiecīgi “Visual Studio” ar milisekunžu precizitāti izmēra laiku starp abiem “brakepoint”.

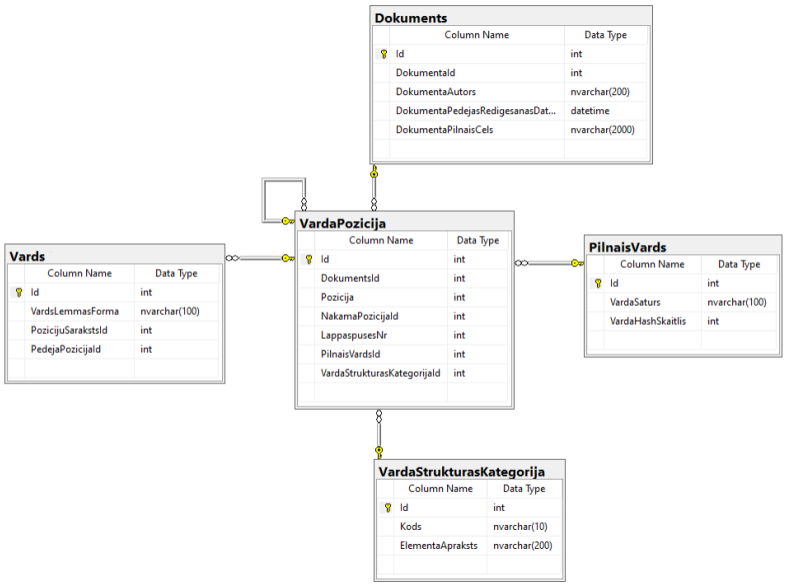
Alternatīvo realizāciju testēšana tiek veikta izpildot izveidošanas algoritmu uz šī darba autora kursa darba “Dokumentu meklēšanas sistēma”, kas ir dokuments, kas sastāv no 46 lappusēm un aptuveni 8000 vārdiem. Visi testpiemēri tiek izpildīti uz tā paša datora un tādas pašas noslodzes apstākļos.



2.5. att. **Piemērs laika mērīšanai no “Visual studio” noteikts laiks starp “brakepoint”**

2.3.3. Datu bāzes struktūra

Attēls 2.6. satur relāciju datu bāzes fizisko modeli un visi tālāk aprakstītie dokumentu indeksa struktūras izveidošanas varianti aizpilda datu bāzi ar tieši šādu struktūru. Šīs nodaļas ietvaros indeksa struktūras izveidošanas variantu aprakstos, komentāros un secinājumos tiek lietoti tabulu un lauku nosaukumi, kas ir atbilstoši šī attēla datu bāzes modelī esošajiem. Vienkāršības pēc tekstā tabulu vai lauku nosaukumi var tik vienkāršoti, piemēram tabula "VardaPozicija" var tikt saukta kā "Vārda pozīcija".



2.6. att. **Dokumentu indeksa struktūras datu bāzes fiziskais modelis**

2.3.4. Dokumentu indeksa struktūras izveidošanas algoritms

Dažādos variantos tiks testēts pamatā tas pats algoritms, lai varētu izsecināt kāda ir tā visefektīvākā implementācija. Izveidošanas algoritms apstaigā katru indeksējamā dokumenta vārdu, un katram vārdam tiek potenciāli izveidoti ieraksti trijās tabulās – "Vārds", "Vārda pozīcija" un "Pilnais vārds". Tālāk testētais algoritms izveido pilnvērtīgu dokumentu indeksa struktūru, kas ir atbilstoša "Datu struktūras dokumentu meklēšanai" sadaļā aprakstītajai struktūrai "Dokumentu indeksēšanas datu struktūras pamatvariants". Ja testēšanas kopsavilkumā

attiecīgajai modifikācijai nav pierakstītas optimizācijas, tad tā tiek pildīta atbilstoši aprakstītajam izveidošanas algoritmam.

Katram no dokumenta ielasītajam vārdam:

1) Tabulā "Pilnais vārds" atrast ielasītajam vārdam atbilstošu vārdu pēc ielasītā vārda pilnā satura.
    1.1) Ja atbilstošs vārds nav atrasts, tad ielasītais vārds tiek ievietots tabulā "Pilnais vārds".
2) Tabulā "Vārda pozīcija" tiek izveidots jauns ieraksts, kurš satur informāciju par ielasīto vārdu un referenci uz atbilstošā pilnā vārda ierakstu.
3) Tabulā "Vārds" atrast ielasītajam vārdam no tā lemmas formas ar jaucējfunkciju iegūtam skaitlim atbilstošu skaitli.
    3.1) Ja atbilstošs vārds ir atrasts, tad tiek iegūts pozīciju saraksta pēdējais elements (tabulā "Vārda pozīcija" ejot pa lauka "Nākamā pozīcija" vērtībām uz priekšu līdz lauka "Nākamā pozīcija" vērtība ir null). Tad tabulā "Vārda pozīcija" šim atrastajam ierakstam laukā "Nākamā pozīcija" ievieto 2 solī jaunās izveidotās pozīcijas identifikatoru.
    3.2) Ja atbilstošs vārds netika atrasts, tad tabulā "Vārds" tiek ievietots jauns ieraksts, kuram references laukā "Pozīciju saraksts" ievieto identifikatora vērtību no 2 solī izveidotās jaunās pozīcijas.

2.3.5. ORM ietvara variants

Indeksa struktūras izveidošanas algoritms ir norealizēts C# programmēšanas valodā, visas operācijas ar datu bāzi tiek veiktas ORM ietvara "Entity Framework" pakļautībā izmantojot LINQ un lambda izteiksmes. Katrā testēšanas izpildes reizē ievietoti tika 12 247 ieraksti. Veicot testēšanu datu bāzē tabulām ir izveidots "Int Identity". Lai noteiktu ātrāko modifikāciju tiek veiktas izmaiņas gan datu bāzes konteksta konfigurācijā, gan datu bāzes tabulā "Pilnais vārds" un katrai modifikācijai tiek izpildīta programma fiksējot izpildes laiku. Laiks, kas nepieciešams lai apstaigātu dokumentu, izgūtu visus vārdus ir aptuveni 1 800ms.

2.2. tabula

Dažādo ORM ietvara varianta modifikāciju testēšanas kopsavilkums

Nr	Datu bāzes konteksta konfigurācija	Vidējais Izpildes laiks (ms)	Mēģinā-jumu skaits	Optimizācijas
1	Noklusējuma	580 585	3	Nav
2	"AutoDetectChangesEnabled" iestatīts false	17 979	5	Nav
3	"AutoDetectChangesEnabled", "ValidateOnSaveEnabled" iestatīti false	12 212	5	Nav
4	"AutoDetectChangesEnabled", "ValidateOnSaveEnabled" iestatīti false	12 164	5	Izveidots neklasterēts indekss tabulai "Pilnais vārds", pilnā teksta laukam. Algoritmā tiek lietots solī 1.
5	"AutoDetectChangesEnabled", "ValidateOnSaveEnabled" iestatīti false	13 869	5	"Pilnais vārds", izveidots jauns lauks – "Vārda hash skaitlis", kuram izveido neklasterētu indeksu kurā glabā no vārda teksta ar jaucējfunkciju iegūtu skaitli, un tad attiecīgi solī 1, dokumenta ielasīto vārdu konvertē ar to pašu jaucējfunkciju uz skaitli, un meklē pēc skaitļa.
6	"AutoDetectChangesEnabled", "ValidateOnSaveEnabled" iestatīti false	12 947	5	Tabulai "Vārds", izveidots jauns lauks "Pēdējā pozīcija Id", kurā tiek glabāta pozīciju saraksta pēdējā identifikatora "Id" lauka vērtība. Attiecīgi tiek veiktas izmaiņas algoritma solī 3.1. pēdējā saraksta elementa vērtība tiek iegūta no "Pēdējā pozīcija Id" laukā pieglabātās un tā vairs nav jāmeklē

				apstaigājot pozīciju sarakstu līdz tā beigām.

2.3. tabula

Novērojumi un komentāri par ORM ietvara varianta modifikācijām un testēšanas rezultātiem

Atbilst Nr	Apraksts
1, 2	Noklusējuma variantā visas datubāzes konteksta papildiespējas bija iestatītas uz "true". "AutoDetectChangesEnabled" ļoti ievērojami paildzināja izpildes laiku – aptuveni 30 reizes, salīdzinājumā ar variantu kur tas tika atspējots, šī funkcija automātiski nosaka kuriem objektiem ir veiktas izmaiņas, lai zinātu kuri objekti ir jāatjauno datu bāzē. Izmaiņu noteikšanas algoritms tiek izsaukts, katru reizi, kad tiek izsaukta funkcija "SaveChanges", un šajā realizācijā ir nepieciešams pēc katras vienas indeksa struktūras izveidošanas algoritma izpildes izsaukt funkciju "SaveChanges", kas nozīmē ka izmaiņu noteikšanas algoritms tiks izsaukts tik reizes cik dokumentā kopumā ir vārdi, kas izskaidro tik ievērojamu kopējā izpildes laika palielināšanos. Atspējojot šo funkciju vajadzēja pievienot papildus vienu rindiņu, lai "Entity framework" zinātu, ka algoritma 3.1 solī ir bijušas izmaiņas un ka tās vajag atjaunot arī datubāzē. Savukārt pārējās datu bāzes konteksta konfigurācijas izmaiņas būtiskas atšķirības izpildes laikā neizraisīja un tabulā netiek reģistrētas. [6]
3	"ValidateOnSaveEnabled" iestatīšana uz "false" neradīja nekādas papildus problēmas un neprasīja arī nekādas papildus modifikācijas dokumentu izveidošanas programmai, bet ietaupīja laiku, kas nozīmē, ka šajā gadījumā to var lietot, lai uzlabotu kopējo izpildes laiku.
4, 5	Šajā situācijā indeksu lietošana neuzlaboja indeksa struktūras izveidošanas ātrumu, jāņem vērā, ka indeksi var ievērojami uzlabot meklēšanas laiku, bet katru reizi, kad tiek veiktas izmaiņas attiecīgajā tabulā ir jāpārbūvē arī indekss, kas paņem papildus laiku. Līdz ar to šajā situācijā izdevīgāk ir indeksus izveidot tikai tad, kad dokumentu indeksa struktūras izveidošana ir pilnīgi pabeigta. [7]

3, 6	Modifikācijā nr 3 pēdējās pozīciju saraksta vērtības iegūšanai tiek lietota algoritmā aprakstītā metode solī 3.1, ka tiek iets pa nākamā pozīcija referencēm uz priekšu līdz saraksta galam. Modifikācijā nr 6 šī darbība tiek aizvietota ar to, ka vērtība tiek pieglabāta tabulas laukā, pēc katras pozīcijas ievietošanas ir jāveic tās atjaunošana, kas prasa papildus laiku. Testējot atklājās, ka izmantojot ORM ietvaru tomēr ātrāk būs apstaigāt pa nākamā reference laukiem uz priekšu nekā atjaunot pieglabāto vērtību.

Secinājumi

ORM ietvara "Entity Framework" funkcijas, kas apstrādā datu bāzes objektus, kā piemēram "AutoDetectChangesEnabled" var ļoti negatīvi ietekmēt programmas darbības laiku. Dokumentu indeksa struktūras izveidošanā šī funkcija rada graujošu efektu notestētajā situācijā paildzinot programmas darbības laiku aptuveni 30 reizes.

Salīdzinot modifikācijas nr 3 un 6 tiek secināts, ka ORM ietvars nodrošina efektīvu piekļuvi references objektiem, kur references objekts ir lauks kura datu tips ir klase, kas atbilst datu bāzes tabulai.

ORM ietvara lietošana var atvieglot programmētāja darbu daudzās situācijās, bet tas arī prasa papildus zināšanas par šī ietvara īpašībām un nepareiza ietvara lietošana var ļoti negatīvi ietekmēt programmas izpildes laiku.

2.3.6. SQL procedūras variants

Tas pats iepriekš aprakstītais dokumentu indeksa struktūras izveidošanas algoritms tiek realizēts izmantojot SQL programmēšanas valodu un datu bāzē tas tiek saglabāts kā statiskā procedūra. Attiecīgi programma padod parametrus un aizsūta datu bāzei procedūras izsaukumu izpildei un tādējādi tiek veidota dokumentu indeksa struktūra. Katrā testēšanas izpildes reizē tika izveidoti 11 763 ieraksti. Tiek realizētas dažādas optimizācijas gan SQL procedūrā, gan datu bāzes tabulās, gan izsaucošajā programmā ar mērķi noteikt visātrāko variantu. Laiks, kas nepieciešams lai apstaigātu dokumentu, izgūtu visus vārdus ir aptuveni 1 800ms.

2.4. tabula

Dažādo SQL procedūras varianta modifikāciju testēšanas kopsavilkums

Nr	"Int Identity"	Vidējais Izpildes laiks (ms)	Mēģinā-jumu skaits	Optimizācijas
7	Nē	14 675	5	Nav
8	Jā	7 695	5	Nav
9	Jā	6 860	5	Visās vietās kur notiek atlasīšana, tai skaitā ieraksta atjaunošanai algoritma 3.1 solī, kopumā 4 vietas procedūrā, tiek pievienots TOP(1)
10	Jā	7 546	5	Procedūras sākumā tiek pievienota sql koda rinda: "SET NOCOUNT ON"
11	Jā	8 921	5	Tabulā "Pilnais vārds" izveidots neklasterēts indekss laukam "Vārda saturs", tiek lietots algoritma 1 solī.
12	Jā	5 677	5	Tabulai "Pilnais vārds", izveidots jauns lauks – "Vārda hash skaitlis", kuram izveido neklasterētu indeksu kurā glabā no vārda teksta ar jaucējfunkciju iegūtu skaitli, un tad attiecīgi solī 1, dokumenta ielasīto vārdu konvertē ar to pašu jaucējfunkciju uz skaitli, un meklē pēc šo skaitļu atbilstības.

13	Jā	7 038	5	Tabulai "Vārds", izveidots jauns lauks "Pēdējā pozīcija Id", kurā tiek glabāts pozīciju saraksta pēdējā identifikatora Id. Attiecīgi tiek veiktas izmaiņas algoritma solī 3.1. pēdējā saraksta elementa vērtība tiek iegūta no "Pēdējā pozīcija Id" laukā pieglabātās un tā vairs nav jāmeklē apstaigājot pozīciju sarakstu līdz tā beigām.
14	Jā	5 133	5	Programma tiek pārveidota tā lai uz datu bāzi sūtītu nevis 1 procedūras izpildi, bet tiek uzkrātas 100 procedūras izpildes inicializācijas un tiek sūtītas pa 100 uz datubāzi.
15	Jā	5 056	5	Tas pats, kas 14, tikai tiek sūtīts pa 1000 procedūras izpildēm.
16	Jā	5 007	5	Tas pats, kas 14, tikai tiek sūtīts pa 2000 procedūras izpildēm.
17	Jā	2 283	5	Apvienotas optimizācijas no 9, 10, 12, 13, 15.

2.5. tabula

Novērojumi un komentāri par SQL procedūras varianta modifikācijām un testēšanas rezultātiem

Atbilst Nr	Apraksts
7, 8	Modifikācijā nr 7 ievietojot ierakstus tabulās "Vārda pozīcija" un "Pilnais vārds" vertības laukos "Id" tika noteiktas izmantojot SQL vaicājumu un atlasot lielāko attiecīgajā tabulā tajā brīdī esošo "Id" lauka vērtību un pieskaitot tai 1. Savukārt modifikācijā nr 7 šie vaicājumi nebija nepieciešami, jo tika izmantota SQL iebūvētā funkcija "Int Identity", kas pie ieraksta ievietošanas pati nosaka nākamo lauka

	vērtību uzzinot iepriekšējo lielāko, tādējādi sanāca ievērojami uzlabot darbības laiku, gandrīz divas reizes.
10	"NoCount" funkcija SQL serverī pēc noklusējuma ir izslēgta, kas nozīmē, ka kad tiek veiktas kādas darbības ar datu bāzes ierakstiem, tad konsolē tiek izvadīts paziņojums par to cik ieraksti ir modificēti. Dokumentu izveidošanas programmai nav vajadzības pēc šādas funkcijas, tāpēc to var atslēgt un tādējādi iegūstot pavisam nelielu ātrdarbības uzlabojumu. [8]
11, 12	Modifikācijā nr 11 izveidotais indekss sastāv no vārda oriģinālās formas, un šī lauka datu tips ir "nvarchar(100)", bet modifikācijā nr 12 izveidotais indekss sastāv no 4 baitīga skaitļa, kas iegūts ar jaucējfunkciju pārveidojot vārda saturu uz skaitli un testējot tiek pierādīts, ka indekss modifikācijā nr 12 ir būtiski efektīvāks.
13	SQL procedūra sākotnēji tika izveidota tā, ka atbilstoši aprakstītā algoritma 3.1 solim tiek atrasta katra nākamā pozīcija, kas prasa vairākas meklēšanas atkarībā no pozīciju saraksta garuma, katram "Nākamā pozīcija Id" references laukā esošajam Id ir jāveic atlase "Vārda pozīcija" tabulā lai atrastu ierakstu un varētu iegūt no tā nākamo referenci. Garu pozīciju sarakstu gadījumā šāda realizācija ļoti degradētu kopējo izpildes laiku, jo "Vārda pozīcija" tabula satur visu dokumentu visas pozīcijas. Veicot optimizāciju SQL procedūra pārveidota uz tādu, kurai tabulā "Vārds" tiek pieglabāta saraksta pēdējās vērtības Id un tad tas nemaz nav jāmeklē, tikai pēc jaunās pozīcijas pievienošanas ir jāatjauno.
14-16	Lai programma varētu datu bāzē izpildīt SQL procedūru tā pieprasījuma formā ir jāaizsūta datu bāzes serverim, kurš tad attiecīgi tālāk to izpildīs un vēl nosūtīs atpakaļ atbildi. Šis process paņem kādu laiku atkarībā no serveru noslodzes, infrastruktūras un dažādām lietām, normālos apstākļos tas varētu aizņemt 5-50 ms. Tātad sūtot SQL serverim porcijas ar komandām var ietaupīt sūtīšanas laiku, kas tiek pierādīts modifikācijās nr 14-16.

Secinājumi

"Int Identity" lietošana būtiski paātrināja dokumentu indeksa izveidošanas programmu, neviena cita optimizācija neuzlaboja tik ļoti kopējo izpildes ātrumu. Šo optimizāciju ir ļoti ieteicams lietot, bet ir jāņem vērā, ka lietojot šo funkciju ir iespējams nonākt situācijā kur vairs jaunus ierakstus nevarēs ievietot, jo būs sasniegta maksimālā vērtība attiecīgajam datu tipam, kas ir laukam kuram rēķina nākamo vērtību. Lai izvairītos no šādas situācijas ir vai nu uzreiz laukam jāizveido tāds datu tips kurš netiks pārsniegts, vai jāparedz pāriešana uz lielāku datu tipu situācijā kur ir sasniegta attiecīgā tipa maksimālā vērtība. [9]

Otra nozīmīgākā optimizācija ir procedūru sūtīšana uz serveri pa lielām porcijām. Salīdzinājumā ar realizāciju kur uz serveri sūta vienu procedūras izpildi šāda pieeja kur sūta uzreiz lielākas porcijas var uzlabot kopējo izpildes ātrumu par aptuveni 35%, kas ir diezgan nozīmīgs ātruma uzlabojumus. Ne visām programmām un funkcijām šī optimizācija dos tādu efektu, bet šāds efekts tiek sasniegts tieši indeksa struktūras izveidošanas programmai, kas nozīmē ka šajā un līdzīgās situācijās kur jāveic lieli datu modificēšanas darbi šī optimizācija ir ļoti vērtīga.

Liela nozīme ir datu bāzes indeksiem, labākajā gadījumā visas meklēšanas tiek veiktas izmantojot indeksu, tādējādi jūtami tiek uzlabots meklēšanas ātrums tabulās, kas savukārt uzlabo kopējo indeksa struktūras izveidošanas algoritmu, jo vairākos tā soļos pirms ievietošanas ir jāveic meklēšana noskaidrojot vai tāds vārds jau nav tabulā. Jāņem vērā, ka indeksi, kas veidoti no skaitļiem ir būtiski efektīvāki nekā indeksi kas veidoti no garām simbolu virknēm un ir ieteicams simbolu virknes pārveidot ar jaucējfunkciju uz skaitļiem, tādējādi lietojot efektīvāku indeksēšanas variantu. Indeksi būtiski uzlabo meklēšanas ātrumu, bet arī samazina ierakstu izveidošanas, atjaunošanas ātrumu, tāpēc ir jāseko līdzi, lai nebūtu lieki indeksi, kā arī atsevišķus indeksus var noņemt pirms atjaunošanas operācijām un atkal izveidot pēc.

2.3.7. Gala secinājumi par abiem variantiem

Dokumentu indeksa struktūras izveidošanas realizācijai daudz piemērotāks ir variants izmantojot SQL procedūru, jo tas strādā būtiski ātrāk. Labākā SQL varianta modifikācija datu

bāzes darbības paveica aptuveni 450ms, kas sevī ietver 11 763 ierakstu izveidošanu, bet labākajā ORM ietvara varianta modifikācijā datu bāzes darbības paveica aptuveni 10 300ms, kas sevī ietvēra 12 247 ierakstu izveidošanu. Tātad labākā SQL varianta modifikācija ir aptuveni 22 reizes ātrāka par labāko ORM ietvara modifikāciju. Kā datu bāzes darbības šeit tika mērītas ierakstu izveidošanas, modificēšanas darbības, kā arī meklēšanas esošajos datu bāzes ierakstos.

Dokumentu indeksa struktūras izveidošanai SQL procedūras variants ir būtiski ātrāks par ORM ietvara variantu vairāku iemeslu dēļ:

- ORM ietvara variantā aprakstītā dokumentu izveidošanas algoritma soļos 1 un 3 tiek iegūti objekti no datu bāzes, lai tālāk programmā varētu izmantot šos objektus, kas nozīmē ka katram solim notika datu apmaiņa ar datu bāzes serveri, kas atkarībā no situācijas var paņemt dažādu laiku, bet SQL procedūras variantā tas pats tiek darīts serverī uz vietas un līdz ar to netiek tērēts laiks uz datu apmaiņu starp serveriem;
- ORM ietvara variantā nav iespējams sūtīt lielas komandu izpildes porcijas uz datu bāzes serveri, kā tas ir iespējams SQL procedūras variantā. Pēc katras viena vārda apstrādes ir nepieciešams izsaukt saglabāšanas funkciju, jo aprakstītā algoritma soļos 1 un 3 ir nepieciešams noteikt vai datu bāzē eksistē tāds vārds, un tas nozīmē ka visiem iepriekšējiem vārdiem jau ir jābūt saglabātiem datu bāzē;
- ORM ietvars visām manipulācijām ar datu bāzi veic dažādas papildus operācijas, kas SQL procedūras variantā vispār nav jāveic, kā piemēram - vaicājuma sagatavošana, kas sevī ietver SQL vaicājuma izveidošanu, modelim un tā metadatiem atbilstoša komandu koka ģenerēšanu, atgriežamo datu, struktūras noteikšana, atgriezto objektu materializēšanas process [10];
- ORM ietvara variants nespēja meklēšanā izmantot neklasterēto tabulas indeksu, līdz ar to tika zaudēts ātrdarbības ieguvums tabulai “Pilnais vārds”.

2.3.8. Testēšana uz apjomu

Lai labāk varētu saprast kā indeksa struktūras izveidošanas laika patēriņš mainās attiecībā pret indeksējamo dokumentu izmēriem un variantiem tiek veikta katra varianta labākās modifikācijas testēšana uz dažādiem indeksējamo apgabalu izmēriem. Indeksējamais apgabals ir dokuments vai to kopa, kas atrodas vienā mapē.

2.6. tabula

Apjoma testēšanas rezultātu kopsavilkums

Variants	Ievieto ierakstu skaits	Vidējais laiks ar datu bāzes operācijām (ms)	Vidējais laiks bez datu bāzes operācijām (ms)	Mēģinājumu skaits
ORM ietvara variants	28 471	35 930	6 933	3
SQL variants	28 660	8 278	6 956	3
ORM ietvara variants	72 909	102 644	15 106	3
SQL variants	73 200	18 708	15 046	3
ORM ietvara variants	221 684	356 285	56 690	3
SQL variants	222 674	67 515	56 245	3

2.3.9. Nodaļas nobeiguma secinājumi

Ņemot vērā datu bāzes operācijām tērēto laiku, ko var iegūt no kolonnas "Vidējais laiks ar datu bāzes operācijām" atņemot "Vidējais laiks bez datu bāzes operācijām" ORM ietvara variants ir būtiski lēnāks gan pie mazāka ierakstu skaita, gan arī pie lielāka. Pie 28 tūkstošiem ievietotu ierakstu ORM ietvara variants ir aptuveni 20 reizes lēnāks par SQL variantu, un pie 222 tūkstošiem 26 reizes lēnāks, kas nozīmē, ka pie vēl vairāk ierakstiem šī starpība par labu SQL variantam tikai vēl var pieaugt.

Visi varianti ar to modifikācijām tika testēti izpildot dokumentu izveidošanas algoritmu programmā, bet ņemot vērā to, ka dokumentu atjaunināšana tiek veikta pēc tāda principa, ka ieraksti tiek izdzēsti un atkal ievietoti, kas sevī ietver izveidošanas operācijas, tad šīs izpētītās un notestētās optimizācijas būs derīgas arī atjaunināšanas funkcionalitātei un ne tikai. Izpētītās optimizācijas ir derīgas arī citiem līdzīga tipa uzdevumiem, kur datu bāzē ir jāveic lieli datu ievietošanas/atjaunošanas uzdevumi.

Izvirzītais uzdevums ir izpildīts pēc testēšanas rezultātiem nosakot ātrāko un efektīvāko variantu dokumentu indeksa izveidošanai, kas ir SQL varianta modifikācija nr 17.

3. DOKUMENTU MEKLĒŠANAS RISINĀJUMI

3.1. Asinhronā un paralēlā programmēšana

Sinhronā programmas izpildē uzdevumi vai funkcijas tiek izpildīti secīgi pēc kārtas, katra nākamā funkcija tiek izpildīta, kad iepriekšējā ir pabeigusi savu darbu, kur uzdevums vai funkcija ir noteikts programmkoda fragments, ko izpilda procesors. Sinhronās programmēšanas modelis ir visizplatītākais un visvienkāršākais, bet mūsdienās kur lielākā daļa procesori ir daudzkodolu un ir spējīgi vienlaicīgi strādāt ar daudziem pavedieniem, absolūti sinhronas programmas nespēj izmantot šo sniegto potenciālu.

Asinhronā programmas izpildē procesora izpildes pavediens var sākt izpildīt funkciju kamēr iepriekšējā funkcija programmā vēl nav pabeigusi savu darbu. Šāda programmu izpilde uzlabo kopējo programmas spēju vienlaicīgi izpildīt vairākus lietotāja inicializētus atsevišķus uzdevumus, jo kamēr tiek izpildīts uzdevums, kas prasa vairāk laiku piemēram datora sekundārās atmiņas lasīšana var tikt apstrādāts jau nākamais uzdevums, ko iespējams lietotājs ir uzsācis lietotāja saskarnē. Izmantojot šādu programmēšanas modeli lietotājam ir iespēja veikt lietotnē darbības, kamēr fonā tiek izpildīti citi uzdevumi, kas nav iespējams pilnīgi sinhronā modelī. Šāda pieeja arī var uzlabot, bet ne būtiski mainīt kopējo izpildes laiku funkcijām kuras tiek izpildītas asinhroni.

Paralēlā programmēšana dod iespēju izpildīt funkcijas pilnīgi paralēli izmantojot vairākus procesora izpildes pavedienus, un tas sniedz iespēju būtiski uzlabot programmas kopējo izpildes laiku, programmas izpildi sadalot pa apakšfunkcijām kuras tiek pildītas paralēli. Šis modelis pēc savas būtības neuzlabo programmas spēju vienlaicīgi izpildīt vairākus lietotāja inicializētus uzdevumus, bet to var realizēt ja paralēlo izpildi implementē asinhroni.

3.2. Dokumentu meklēšana pēc frāzes

Lietotājs dokumentu meklēšanas procesu veic tā, ka grafiskajā lietotāja saskarnē ir ievadījis frāzi, kas ir teksta fragments kurš sastāv no vārdiem, kur vārds ir simbolu virkne kas atdalīta ar tukšuma simboliem. Lietotājs izvaddatos atpakaļ saņem meklēšanas rezultātu kopu, kas sastāv

no: teksta fragmenta priekšskatījuma kurš sevī ietver atrasto frāzi un tās kontekstu, kur konteksta robežas nosaka teikuma sākums vai beigas, vai arī paragrāfa sākums vai beigas; dokumenta metadatu informācija; lapaspuse kurā atrasta frāze. Meklēšanas rezultātu kopā instance tiek ievietota ja viena dokumenta ietvaros ir lietotāja ievadītajai frāzei atrasta atbilstoša frāze, kur vārdu atbilstība tiek noteikta vārda lemmas formas līmenī, kas nozīmē ka vārda locījumi var atšķirties.

Dokumenta saturs indeksa datu struktūrā tiek reprezentēts izejot no vārdu pozīcijām, katrai pozīcijai ir savs kārtas numurs un attiecīgi frāzi atrast var ja atbilstošajiem vārdiem lemmas formā kārtas numuri ir secīgi blakus. Priekšskatījuma veidošanā tiek izmantota vārda oriģinālā forma, un vārda strukturālās piederības kategorija.

3.3. Piemērs meklēšanas rezultātiem

Tabulā tiek demonstrēts, kā tiek attēloti un strukturēti meklēšanas rezultāti. Reālā programmā papildus varētu būt iespēja atvērt attiecīgo dokumentu ar klikšķi, vai dokumenta pilnais ceļš uz datu nesēja, kā arī reālā situācijā būtu vairāk rezultātu katrai meklētajai frāzei. Piemērā tiek pieņemts, ka dokumentu indeksa struktūrā ir noindeksēti sekojoši dokumenti:
Dokuments 1: Parastais ozols izaug liels koks ar vērtīgu koksni, augļi — rieksti, kurus sauc arī par zīlēm jeb ozolzīlēm. Zīles senāk cilvēki izmantoja pārtikā (gatavoja "ozolzīļu kafiju"), kā arī tās ir nozīmīgs barības avots dažādiem dzīvniekiem.
Dokuments 2: Bērzi aug tikai Ziemeļu puslodē, tas ir, Eiropā, Āzijā līdz Himalaju dienvidiem un Ziemeļamerikā.

3.1. tabula

Meklēšanas rezultātu attēlošanas piemērs

Meklētā frāze	Atrastās frāzes teksta priekšskatījums	Lapas Nr	Dokumenta nosaukums	Dokumenta autors
Parastais ozols	**Parastais ozols** izaug liels koks ar vērtīgu koksni, augļi — rieksti, kurus sauc arī par zīlēm jeb ozolzīlēm.	1	Dokuments 1	Jānis

cilvēks izmanto	Zīles senāk **cilvēki izmantoja** pārtikā (gatavoja "ozolzīļu kafiju"), kā arī tās ir nozīmīgs barības avots dažādiem dzīvniekiem.	1	Dokuments 1	Jānis
Eiropa, āzija	Bērzi aug tikai Ziemeļu puslodē, tas ir, **Eiropā, Āzijā** līdz Himalaju dienvidiem un Ziemeļamerikā.	1	Dokuments 2	Līga

3.4. Realizēšanas iespēju praktiskais pētījums

Datu struktūras, kas veidotas pēc dokumentu indeksēšanas principa sniedz iespēju veikt ātru meklēšanu lielā dokumentu kolekcijā. Tomēr meklēšanas ātrums ir ļoti atkarīgs no veicamajiem meklēšanas uzdevumiem un veida kādā meklēšana tiek realizēta, un lai varētu izmantot pilnu dokumentu indeksa struktūras potenciālu ir nepieciešams implementēt efektīvu meklēšanu. Meklēšanas funkcionalitātes ātrumam svarīgs ir ne tikai izveidotais meklēšanas algoritms, bet arī izmantotās tehnoloģijas un infrastruktūras lietojums, tāpēc lai varētu noteikt efektīvāko realizācijas variantu tiek veikts praktisks pētījums.

3.4.1. Uzdevums

Eksistē dažādi veidi kā var realizēt meklēšanu “Datu struktūras dokumentu meklēšanai” nodaļā aprakstītajām datu struktūrām. Šīs praktiskās daļas ietvaros tiks izstrādes vidē realizēti un notestēti dažādi algoritmi dokumentu meklēšanai pēc frāzes. Galvenie šī praktiskā pētījuma mērķi ir noteikt ātrāko meklēšanas algoritma realizāciju, identificēt algoritma izpildes laika efektīvākās optimizācijas, kā arī noteikt cik ātri meklēšanas algoritmi strādā pie dažādiem meklējamo dokumentu apjomiem.

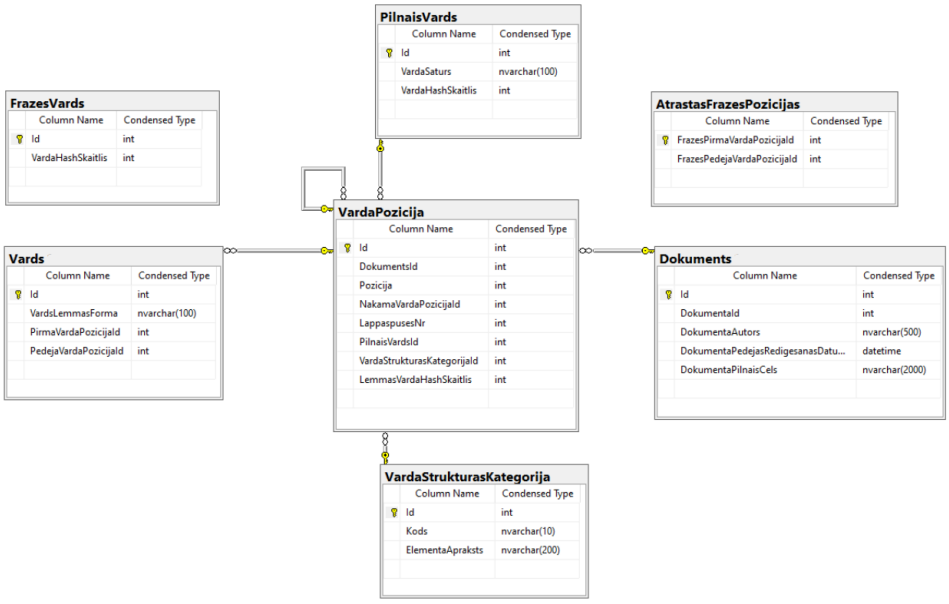
3.4.2. Izmantotie rīki un metodes

Meklēšanas algoritmi tiek realizēti programmēšanas valodā C# izmantojot integrēto izstrādes vidi “Microsoft Visual Studio”. Datu bāze ir izveidota un tiek pārvaldīta izmantojot “SQL Server Management Studio”. Interakcijā ar datu bāzi tiek lietots ORM ietvars “Entity Framework”, vai arī “SQL”. Paralēlo uzdevumu izpildei tiek lietota C# klase “Task”. Laiks tiek mērīts izmantojot “brakepoint” metodi, kas ir aprakstīta apakšnodaļā “2.3.2.”.
Testēšana tiek veikta izmantojot datoru ar sekojošām specifikācijām:

- Procesors: Intel(R) Core(TM) i7-11800H, 2.30 GHz, 8 kodoli, 16 loģiskie procesori;
- Operatīvā atmiņa: 16.0 GB, DDR4 3200 MHz;
- Sekundārā atmiņa: Samsumg mzvlq1t0halb, lasīšana 2300 MB/s, rakstīšana 1350 MB/s;
- Operētājsistēma: Microsoft Windows 11 Home.

3.4.3. Datu bāzes struktūra

Attēlā 3.1. redzamais relāciju datu bāzes fiziskais modelis reprezentē datu bāzes tabulas kurās tiek uzglabāti dokumentu indeksa dati un attiecīgi lietojot šo datu bāzi tiek realizēti visi tālāk aprakstītie dokumentu meklēšanas varianti. Tālāk šajā nodaļā meklēšanas variantu aprakstos tiek lietoti tabulu un lauku nosaukumi no attēlā redzamā datu bāzes modeļa, tie tekstā var tikt lietoti vienkāršoti, piemēram tabula “VardaPozicija” var tikt saukta kā “Vārda pozīcija”. Šī datu bāzes fiziskā realizācija sevī apvieno īpašības no 1. nodaļā aprakstītajām datu struktūrām “Dokumentu indeksēšanas datu struktūras pamatvariants” un “Alternatīvais variants dokumenta vārdu pozīcijām”.



3.1. att. **Datu bāzes fiziskais modelis**

3.4.4. Testpiemēru dati

Lai varētu savā starpā salīdzināt dažādo meklēšanas variantu izpildes ātrumus, katrs no variantiem tiek testēts uz tiem pašiem testa datiem. Kā testa dati tiek izmantota datu bāzē veiksmīgi izveidota dokumentu indeksa struktūra noteiktai dokumentu kopai.

3.2. tabula

Priekš testēšanas noindeksēto testpiemēru datu bāzes saturs

Nr	Dokumentu skaits	Kopējais noindeksētais vārdu skaits visos dokumentos (Ieraksti tabulā "Vārda pozīcija")	Kopējais unikālo pilno vārdu skaits visos dokumentos	Datu bāzes kopējais rezervētais izmērs (GB)
1	16	1 177 270	85 019	1,04
2	260	18 354 893	116 482	2,08
3	1 240	300 032 685	128 978	31,45

3.4.5. Kopīgie meklēšanas procesa veicamie uzdevumi

Tālāk ir aprakstīti dažādi varianti kā realizēt meklēšanas algoritmu, tāpēc šeit tiek dots augsta līmeņa ieskats kopīgajos uzdevumos kuri ir jāpaveic dokumentu meklēšanas funkcionalitātes realizēšanas ietvaros. Attiecīgi tālāk aprakstītie algoritmi tiek strukturēti atbilstoši šiem uzdevumiem.

Meklēšanas procesa veicamie uzdevumi:

1. **Pozīciju saraksta iegūšana**. No datu bāzes tabulas "Pozīcija" ir jāiegūst saraksts ar visām katra meklētās frāzes vārda pozīcijām visos dokumentos. Visu vārdu pozīcijas ir apvienotas vienā sarakstā un katra vārda pozīcijām šajā sarakstā ir savs intervāls;
2. **Pozīciju saraksta sakārtošana**. Iegūtais saraksts ir jāsakārto tā, lai tas primāri pēc secības atbilstu meklētās frāzes vārdu secībai, sekundāri pēc lauka "dokumentaId" vērtības, un tikai tad pēc vārdu pozīcijām;
3. **Frāžu meklēšana pozīciju sarakstā un atgriežamo datu sagatavošana**. Apstaigājot iegūto sarakstu ir jāatrod visas frāzes. Frāzes noteikšana tiek veikta tā, ka ir atrasta tāda saraksta elementu kombinācija, kurai katrā vārda intervālā sākot no pirmā intervāla ir pozīcija kura pēc kārtas numura ir lielāka par vienu vienību salīdzinot ar iepriekšējo intervālu. Atrastajām frāzēm ir jāizgūst to oriģinālā teksta satura konteksts, lai varētu izveidot teksta priekšskatījumu.

Piemērs

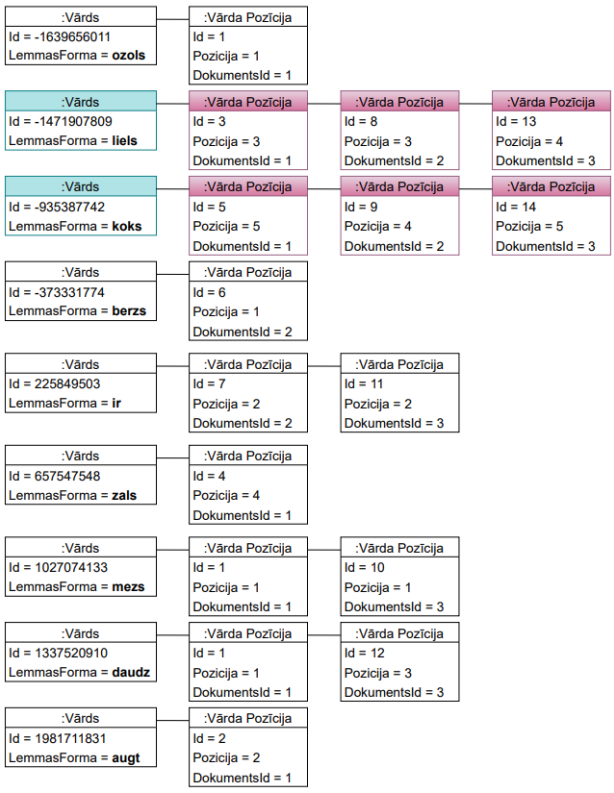
Piemērā izmantojot instanču diagrammas tiek nodemonstrēts, kas notiek katrā meklēšanas procesa uzdevuma solī. Piemēra demonstrējumā netiek attēlotas datu bāzes tabulas ar to visiem laukiem un to pilnajiem nosaukumiem, jo tad diagrammas būtu pārāk lielas un grūti uztveramas, piemēra mērķis ir attēlot būtiskākos principus atbilstoši meklēšanas procesa veicamajiem uzdevumiem. Tiek pieņemts, ka lietotājs meklē frāzi "liels koks" un dokumentu indeksa struktūrā ir noindeksēti sekojoši dokumenti:

Dokuments 1: Ozols izaug liels zaļš koks.

Dokuments 2: Bērzs ir liels koks.

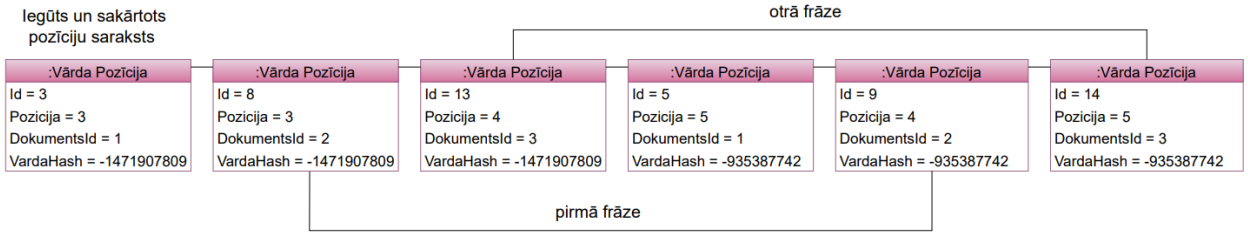
Dokuments 3: Mežā ir daudz lieli koki.

Attēlā 3.2. redzama daļa no piemēra dokumentu indeksa struktūras. Iekļautas ir visas tabulu "Vārds" un "Vārda Pozīcija" instances, bet ne visi to lauki, tiek parādīti tikai lauki, kas ir nepieciešami frāžu meklēšanā. Ar tumši rozā krāsu ir iezīmētas instances, kas soļa 1 izpildes ietvaros tiek pievienotas pozīciju sarakstā.



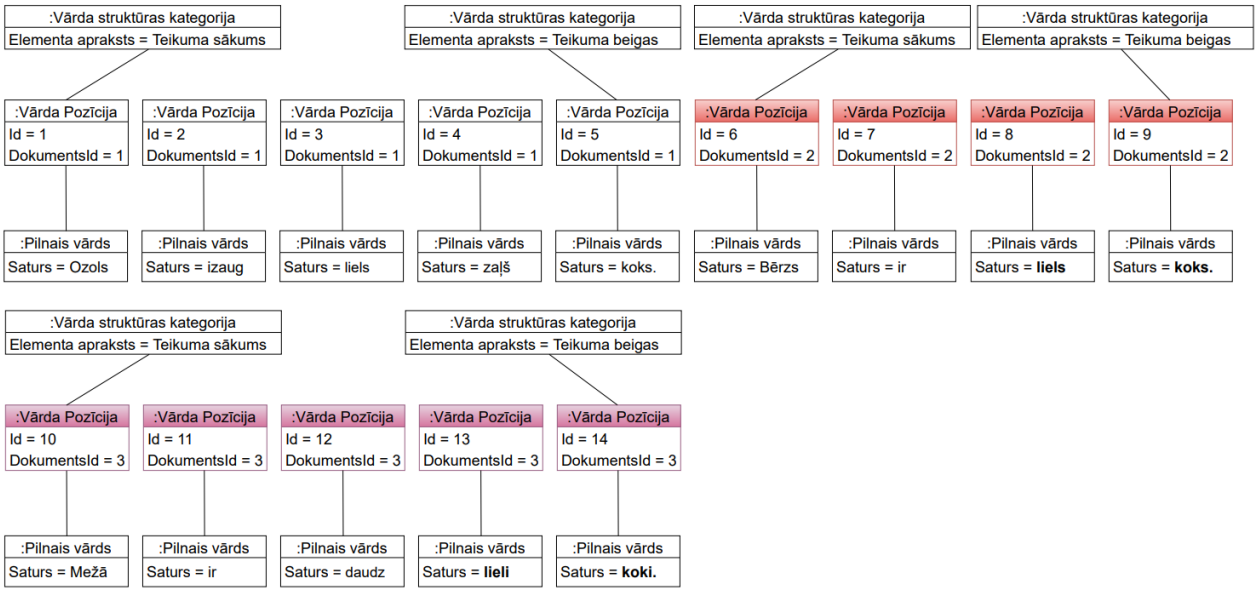
3.2. att. **Piemēra dokumentu indeksa struktūras instanču diagramma**

Attēlota (attēls 3.3.) tiek instanču diagramma, kas reprezentē iegūtu pozīciju sarakstu kurā tiek meklētas frāzes aprakstīto meklēšanas uzdevumu solī 3.



3.3. att. **Sakārtots pozīciju saraksts instanču diagramma**

Attēlā 3.4. treknrakstā ir iezīmēti vārdi, kas pieder pie abām atrastajām frāzēm. Izmantojot oranžo un violeto krāsu ir iezīmēti katras atrastās frāzes konteksti, kuri tiek atgriezti veiksmīgi pabeidzot 3 soli, lai no tiem varētu izveidot teksta priekšskatījumu.



3.4. att. **Frāzes konteksta piemēra instanču diagramma**

3.4.6. Frāžu meklēšanas pamatvariants

Meklēšanas algoritms ir realizēts tā lai tas varētu strādāt ar aprakstīto datu struktūru "Dokumentu indeksēšanas datu struktūras pamatvariants". Šim algoritmam ir divas modifikācijas, kur viena ir pilnīgi sinhrona, bet otrā atsevišķas algoritma daļas tiek izpildītas paralēli. Variants tiek realizēts izmantojot tikai ORM ietvara kontrolētās datu apmaiņas ar datu bāzi.

Algoritms

Algoritms tiek aprakstīts saskaņā ar iepriekš definētajiem meklēšanas procesa veicamajiem uzdevumiem. Šeit tiek aprakstīts precīzāk kas notiek katrā solī konkrēti šim variantam, solis 2 ir tieši tāds pats kā iepriekš definētajā aprakstā, tāpēc šeit tas netiek specificēts.

1.1. Katram no ievadītās simbolu virknes izgūtajam vārdam:

1.1.1. Vārds tiek pārveidots lemmas formā, un tad ar jaucējfunkciju no tā iegūst veselu skaitli.

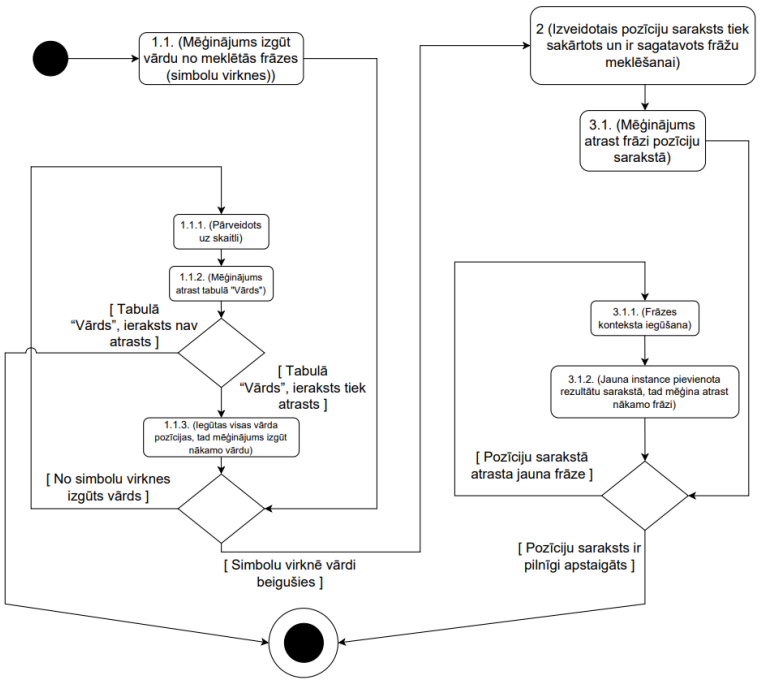
1.1.2. Izmantojot iegūtu veselo skaitli tiek atrasts ieraksts datu bāzes tabulā "Vārds", ieraksts tiek atrasts izmantojot "Id" lauku. Bet ja ieraksts netiek atrasts, tad beidz darbu un atgriež paziņojumu par to, ka frāze neeksistē dokumentu indeksā.

1.1.3. Izmantojot atrastā ieraksta lauku "PozīcijaId" tiek iegūta instance tabulā "Vārda Pozīcija", tālāk izmantojot šīs instances lauku "Nākamā pozīcija" tiek iets līdz saraksta beigām līdz "Nākamā pozīcija" vērtība ir null, un visi šie elementi tiek ievietoti pozīciju sarakstā.

3.1. Katrai pozīciju sarakstā atrastajai frāzei:

3.1.1. Tabulā "Vārda pozīcija" tiek atlasītas pozīcijas sākot no atrastās frāzes pirmās pozīcijas. No atrastās frāzes pirmās pozīcijas uz kreiso pusi atlasa pozīcijas līdz pirmajam struktūras elementam, kas ir vainu paragrāfa, vai teikuma sākums. Tad uz labo pusi atlasa līdz pirmajam struktūras elementam, kas ir vainu paragrāfa, vai teikuma beigas. No iegūtajām pozīcijām tiek iegūti pilnie vārdi no tabulas "Pilnais vārds" un attiecīgi tiek uzkonstruēts teksta priekšskatījums.

3.1.2. Rezultātu sarakstā tiek pievienots ieraksts, kurš satur uzkonstruēto teksta priekšskatījumu, atrasto frāzi un dokumenta metadatus.



3.5 att. **Meklēšanas programmas soļu sinhronās izpildes plūsmas diagramma**

Algoritma asinhronais variants

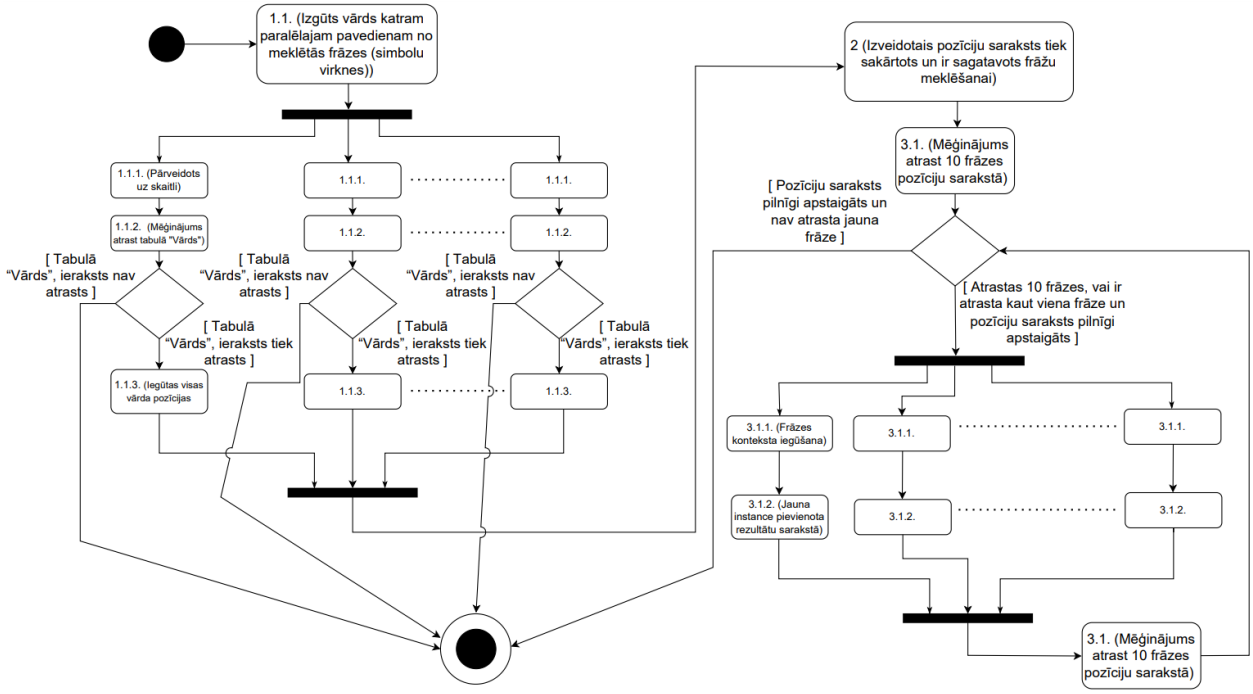
Asinhronā varianta mērķis ir izmantojot vairākus procesora pavedienus pildīt paralēli tās algoritma daļas, kas aizņem viss vairāk laika, lai kopumā meklēšanu varētu paveikt ātrāk. Algoritma asinhronais variants tiek veidots uz sinhronā varianta bāzes un tiek veiktas tās pašas darbības, tikai tās programmkodā tiek izveidotas tā, lai tās varētu noteiktu brīdi notikt paralēli, un ārpus šiem paralēlajiem izpildes apgabaliem programma veic sinhronu darbības plūsmu. Katrs programmas izpildes pavediens veido savu savienojumu ar datu bāzi un atsevišķi veic datu apmaiņu ar to.

Algoritma papildinājumi asinhronajai izpildei:

- Solis 1.1 un visi tā apakš soļi tiek veikti sinhroni viena vārda ietvaros, bet paralēli katram frāzes vārdam, un ne vairāk par 10 pavedieniem vienlaicīgi. Attiecīgi lai uzsāktu soli 2 ir jāsagaida visi soļa 1.1 paralēlie pavedieni, un solis 2 tiek uzsākts tikai tad, kad šie sagaidītie pavedieni ir pabeiguši darbu. Katrs soļa 1.1 pavediens paralēli aizpilda pozīciju sarakstu kurš tiek lietots tālāk soļos 2 un 3;

- Solis 3.1 tiek pārveidots tā, ka tas uzkrāj sarakstā atrastās frāzes un tad, kad ir uzkrātas 10 frāzes vai arī ja pozīciju saraksts ir pilnīgi apstaigāts, tad visas uzkrātās frāzes tiek apstrādātas paralēli, un programma turpina meklēt nākamās frāzes tikai tad, kad visu 10 frāžu paralēlā apstrāde ir pabeigusi savu darbu.

Attēlā 3.6. tiek attēlota meklēšanas algoritma asinhronā soļu izpildes plūsma pieņemot, ka meklētā frāze nesatur vairāk par 10 vārdiem, ja tā satur vairāk, tad solī 1.1. kamēr no frāzes var izgūt 10 jaunus vārdus tiek veiktas iterācijas pa 10 izpildes pavedieniem vienlaicīgi, un pēdējā iterācijā attiecīgi tik cik palika pāri vārdi. Tātad soļa 1.1. izveidotais izpildes pavedienu skaits ir tāds pats kā frāzes vārdu skaits, ja vien tas nav lielāks par 10. Soļa 3.1 izveidotais izpildes pavedienu skaits ir vienāds ar atrasto frāžu skaitu, ja tas ir mazāks par 10, bet ja ir atrastas 10 frāzes, tad izpilda 10 pavedienus vienlaicīgi.



3.6. att. **Meklēšanas programmas soļu asinhronās izpildes plūsmas diagramma**

Testēšanas kopsavilkums

Atsevišķi ir realizēti gan sinhronais, gan asinhronais variants un tie tiek testēti uz tieši tādiem pašiem testa datiem, lai noteiktu kurš un cik lielā mērā būs efektīvāks. Realizētās

meklēšanas programmas tiek testētas atkļūdošanas vidē un to izpildes laiks tiek mērīts pa soļiem 1-2, un 3, attiecīgi kopējais to izpildes laiks ir vienāds ar šo soļu summu. Katram meklētās frāzes tekstam katrā variantā vidējais laiks pa soļiem tiek iegūts veicot 5 testēšanas piegājienus.

3.3. tabula

Sinhronā un Ahinhronā variantu testēšanas rezultātu kopsavilkums

Meklētās frāzes teksts	Variants	Vidējais laiks soļiem 1-2 (ms)	Vidējais laiks solim 3 (ms)	Atrasto frāžu skaits dokumentu kolekcijā	Apakšnodaļas "Testpiemēru dati" lietotais testpiemērs
dokumentu meklēšanas	Sinhronais	110	237	13	1
dokumentu meklēšanas	Asinhronais	86	124	13	1
datu struktūra	Sinhronais	333	1 226	65	1
datu struktūra	Asinhronais	244	684	65	1
dokumentu meklēšanas sistēma	Sinhronais	361	52	3	1
dokumentu meklēšanas sistēma	Asinhronais	206	39	3	1
sanāk glabāt diezgan daudz nulles kas	Sinhronais	1 175	33	1	1
sanāk glabāt diezgan daudz nulles kas	Asinhronais	852	31	1	1

Secinājumi

No testēšanas rezultātu kopsavilkuma skaidri secināms, ka jo garāka ir meklētā frāze, jo ilgāk tiks izpildīti soļi 1-2, un jo vairāk tiks atrastas frāzes, jo ilgāk izpildīsies solis 3, kas nozīmē, ka ir vienlīdz svarīgi optimizēt abus visus šos soļus. Asinhronā uzdevumu izpilde visos testēšanas scenārijos uzlabo izpildes ātrumu un jo vairāk paralēli uzdevumi tiek pildīti noteikto robežu ietvaros, jo ievērojamāks laika ieguvums.

3.4.7. Alternatīvais variants frāžu meklēšanai

Pamatvarianta realizācijā tiek noteikts, ka asinhronā programmēšana būtiski uzlabo meklēšanas programmas izpildes laiku un līdz ar to šajā variantā netiks izšķirti sinhronais no asinhronā kā apakšvarianti, bet uzreiz tiek aprakstīts efektīvākais asinhronais variants. Šis variants tiek veidots tā, lai meklēšanu veiktu atbilstoši tam kā ir aprakstīts apakšnodaļā 1.3. "Alternatīvais variants dokumenta vārdu pozīcijām". Datu bāzes datu iegūšanai tiek lietots ORM ietvars un atsevišķas SQL operācijas netiek lietotas.

Algoritms

Algoritms tiek aprakstīts saskaņā ar apakšnodaļā "Kopīgie meklēšanas procesa veicamie uzdevumi" definētajiem uzdevumiem un uzdevums 2. atsevišķi vēlreiz šeit netiek aprakstīts. Solis 2 veic pozīciju saraksta sakārtošanu tikai tad, kad visi 1.1 izpildes pavedieni ir pabeiguši darbu. Attiecīgi solis 3.1. uzsāk savu darbību, kad solis 2 ir pabeidzis darbu.

1.1. Katram no ievadītās simbolu virknes izgūtajam vārdam tiek izveidots un inicializēts asinhrons izpildes pavediens, bet ne vairāk par 10 vienlaicīgi, tālāk katrā pavedienā tiek veiktas sekojošas darbības:

1.1.1. Vārds tiek pārveidots lemmas formā, un tad ar jaucējfunkciju no tā iegūst veselu skaitli;

1.1.2. Tiek apstaigāta datu bāzes tabula “Vārda pozīcija” un visi tie ieraksti kuriem lauka “LemmasVardaHashSkaitlis” vērtība sakrīt ar solī 1.1.1. iegūto veselo skaitli, tiek pievienoti pozīciju sarakstā. Tātad vienā datu bāzes tabulas “Vārda pozīcija” apstaigāšanā pozīciju saraksts iegūst visas šī vārda pozīcijas visos dokumentos. Ja pozīciju saraksts nav ieguvis papildus pozīcijas, tad ir zināms ka vārds neeksistē dokumentu indeksā un darbs tiek pabeigts atgriežot paziņojumu par to ka frāze nav atrasta;

3.1. Katrai pozīciju sarakstā atrastajai frāzei:

3.1.1. Pievieno atrasto frāzi atrasto frāžu sarakstā;

3.2. Kad atrasto frāžu saraksta izmērs ir 10, vai ja viss pozīciju saraksts ir apstaigāts, tad katrai saraksta frāzei uzsāk paralēlās izpildes pavedienu kurā tiek veikti sekojoši soļi:

3.2.1. No datu bāzes tabulas “Vārda pozīcija” tiek atlasītas pozīcijas sākot atrastās frāzes pirmās pozīcijas uz kreiso pusi atlasa pozīcijas līdz pirmajam struktūras elementam, kas ir vainu paragrāfa, vai teikuma sākums. Tad uz labo pusi atlasa līdz pirmajam struktūras elementam, kas ir vainu paragrāfa, vai teikuma beigas. No iegūtajām pozīcijām tiek iegūti pilnie vārdi no tabulas “Pilnais vārds” un attiecīgi tiek uzkonstruēts teksta priekšskatījums;

3.2.2. Rezultātu sarakstā tiek pievienots ieraksts, kurš satur uzkonstruēto teksta priekšskatījumu, atrasto frāzi un dokumenta metadatus.

Testēšanas kopsavilkums

Aprakstītais variants ir realizēts un tiek notestēts izmantojot atkļūdošanas vidi, laiks tiek mērīts atsevišķi soļiem 1-2 un 3. Katram meklētās frāzes tekstam katrā solī vidējais laiks tiek iegūts veicot 5 testēšanas piegājienus. Katras meklēšanas izpildes kopējais laiks iegūstams saskaitot laiku no abiem izmērītajiem soļiem.

3.4. tabula

Alternatīvā frāžu meklēšanas varianta testēšanas rezultātu kopsavilkums

Meklētās frāzes teksts	Vidējais laiks soļiem 1-2 (ms)	Vidējais laiks solim 3 (ms)	Atrasto frāžu skaits dokumentu kolekcijā	Apakšnodaļas "Testpiemēru dati" lietotais testpiemērs
dokumentu meklēšanas	18	144	13	1
datu struktūra	46	676	65	1
dokumentu meklēšanas sistēma	49	45	3	1
sanāk glabāt diezgan daudz nulles kas	174	37	1	1

Secinājumi

Salīdzinājumā ar pamatvariantu šīs variants ir spējīgs ātrāk veikt meklēšanu tieši soļu 1-2 ietvaros, jo tas izdara to pašu ar mazāk darbībām. Tomēr solī 3 laiks ievērojami pieaug palielinoties atrasto frāžu skaitam, un tā ir potenciāla problēma, tāpēc šo programmas daļu ir nepieciešams realizēt efektīvāk.

3.4.8. Alternatīvais variants ar SQL skatiem

Iepriekšējiem variantiem datu atlasīšana no datu bāzes tika realizēta izmantojot ORM ietvara sniegtās iespējas un datu atlasīšana datu bāzē programmas izpildes laikā notika daudzas reizes. Savukārt šajā variantā lai uzlabotu meklēšanas programmas soļu izpildes laiku tiek samazināts datu pārsūtīšanu skaits starp programmu un datu bāzes serveri un programma tiek strukturēta tā, lai vienā datu bāzes datu atlasīšanā varētu paveikt vairāk datu bāzes servera vidē un iegūt uzreiz visus nepieciešamos rezultātus, nevis tikai daļu no rezultātiem kā iepriekšējos variantos.

Šajā variantā programma ir strukturēta tā, lai datu atlasīšanā tā izmantotu SQL skatus "Frāzes pozīcijas" un "Frāzes konteksts", kuri izmanto pagaidu datus, kas tiek ierakstīti divās speciāli tam veidotās tabulās un līdz ar to tiek iegūts konstants datu bāzes datu atlasīšanas skaits, kas vienmēr būs divas reizes. Papildus divām atlasīšanas reizēm šajā variantā ir vēl nepieciešams ievietot un pēc tam izdzēst pagaidu datus, kas kopumā nozīmē vēl 3 programmas interakcijas ar datu bāzi. Līdz ar to arī asinhronā programmēšana šeit netiek lietota.

Pagaidu dati, kas datu bāzē eksistē tikai programmas izpildes brīdī tabulā "Frāzes vārds" ir atsevišķs ieraksts katram lietotāja meklētās frāzes vārdam, kurš satur no vārda lemmas formas ar jaucējfunkciju iegūtu skaitli, un tabulā "Atrastās frāzes pozīcijas" katrai solī 3. atrastajai frāzei ir izveidots ieraksts kurš satur frāzes pirmā vārda tabulas "Vārda pozīcija" lauka "Id" vērtību un arī frāzes pēdējā vārda tabulas "Vārda pozīcija" lauka "Id" vērtību. Attiecīgi izmantojot šo tabulu vērtības ir katrai tabulai izveidots savs SQL skats, kurš atgriež katrā solī nepieciešamos rezultātus veicot vienu atlasīšanu datu bāzē.

Izmantotie SQL skati

Skats (attēls 3.7.) "Frāzes pozīcijas" atlasa visas pozīcijas visiem meklētās frāzes vārdiem un tas tiek panākts izmantojot tabulas "Frāzes vārds" esošās pagaidu vērtības. "LemmasVardaHashSkaitlis" un "VardaHashSkaitlis" ir lauki kurā pēc viena un tā paša algoritma no vārdiem ir iegūti skaitļi un šeit tie tiek lietoti vārdu identificēšanai. Tātad tiek atlasītas visas tās un tikai tās pozīcijas kuru lauka "LemmasVardaHashSkaitlis" vērtība ir vienāda ar kādu no tabulas "Frāzes vārds" lauka "VardaHashSkaitlis" vērtībām. Kad programmā skata atgrieztās vērtības ir iegūtas, tad tās meklēšanas procesa veicamo uzdevumu solī 2 attiecīgi tiek sakārtotas operatīvajā atmiņā programmas darbības vidē.

```
CREATE VIEW [dbo].[FrazesPozicijas] AS
SELECT vp.Id, vp.Pozicija, vp.LemmasVardaHashSkaitlis, vp.DokumentsId
FROM VardaPozicija vp
WHERE vp.LemmasVardaHashSkaitlis IN (SELECT VardaHashSkaitlis FROM FrazesVards)
```

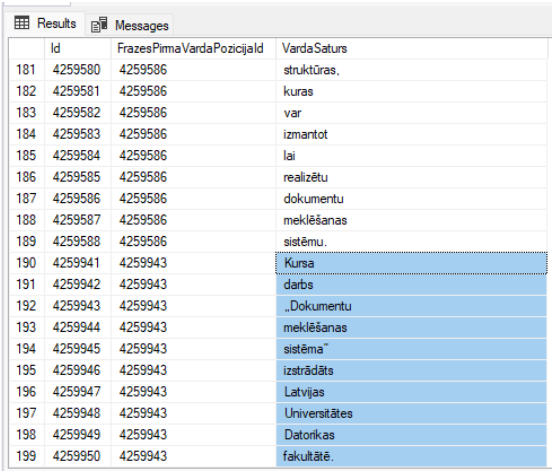
3.7. att. **"Frāzes pozīcijas" skata SQL skripts**

Kad pozīciju saraksts ir pilnīgi apstrādāts un visas tajā esošās frāzes ir atrastas un tās ir ierakstītas tabulā "Atrastās frāzes pozīcijas", tad skats "Frāzes konteksts" (attēls 3.8.) izmantojot šīs tabulas vērtības katrai frāzei atlasa tabulā "Vārda pozīcija" pozīcijas līdz pirmajai netukšajai lauka "VardaStrukturasKategorija" vērtībai un ja vērtība nav null, tad tas nozīmē ka tas ir vai nu paragrāfa vai teikuma sākums, vai arī paragrāfa vai teikuma beigas. Līdz ar to arī 3 soļa laikietilpīgākā procedūra, kas ir atrasto frāžu priekšskatījuma izveidošana šeit ir optimizēta tā, ka to paveic visām atrastajām frāzēm uzreiz SQL serverī ar vienu vaicājumu.

```
CREATE VIEW [dbo].[FrazesKonteksts] AS
    SELECT vp.Id, pp.FrazesPirmaVardaPozicijaId, pv.VardaSaturs
    FROM (SELECT FrazesPirmaVardaPozicijaId
        ,(SELECT MAX(Id) FROM VardaPozicija vp
        WHERE vp.Id <= afp.FrazesPirmaVardaPozicijaId AND
            vp.VardaStrukturasKategorija IS NOT NULL) AS FrazesPirmaVardaId
        ,(SELECT MIN(Id) FROM VardaPozicija vp
        WHERE vp.Id >= afp.FrazesPirmaVardaPozicijaId AND
            vp.VardaStrukturasKategorija IS NOT NULL) AS FrazesPedejaVardaId
        FROM AtrastasFrazesPozicijas afp
    ) AS pp
    INNER JOIN VardaPozicija vp ON
        vp.Id >= FrazesPirmaVardaId AND vp.Id <= FrazesPedejaVardaId
    INNER JOIN PilnaisVards pv ON pv.Id = vp.PilnaisVardsId
    ORDER BY pp.FrazesPirmaVardaPozicijaId, vp.Id
    OFFSET 0 ROWS
```

3.8. att. **"Frāzes konteksts" skata SQL skripts**

Attēlā 3.9 ir parādīts, kā atrastais frāzes konteksts tiek atgriezts no skata "Frāzes konteksts", situācijā kur lietotājs meklē frāzi "dokumentu meklēšanas", iezīmētā atgriezto rezultātu daļa ir frāzes konteksts vienai atrastai frāzei, šajā gadījumā tas ir teikums frāzei "dokumentu meklēšanas". Attiecīgi no šiem atgrieztajiem datiem var ātri uzkonstruēt teksta priekšskatījumu.



Results | Messages

	Id	FrazesPirmaVardaPozicijaId	VardaSaturs
181	4259580	4259586	struktūras,
182	4259581	4259586	kuras
183	4259582	4259586	var
184	4259583	4259586	izmantot
185	4259584	4259586	lai
186	4259585	4259586	realizētu
187	4259586	4259586	dokumentu
188	4259587	4259586	meklēšanas
189	4259588	4259586	sistēmu.
190	4259941	4259943	Kursa
191	4259942	4259943	darbs
192	4259943	4259943	„Dokumentu
193	4259944	4259943	meklēšanas
194	4259945	4259943	sistēma"
195	4259946	4259943	izstrādāts
196	4259947	4259943	Latvijas
197	4259948	4259943	Universitātes
198	4259949	4259943	Datorikas
199	4259950	4259943	fakultātē.

3.9. att. **Atlasītas vērtības no "Frāzes konteksts" skata**

"Frāzes pozīcijas" skata optimizēšana

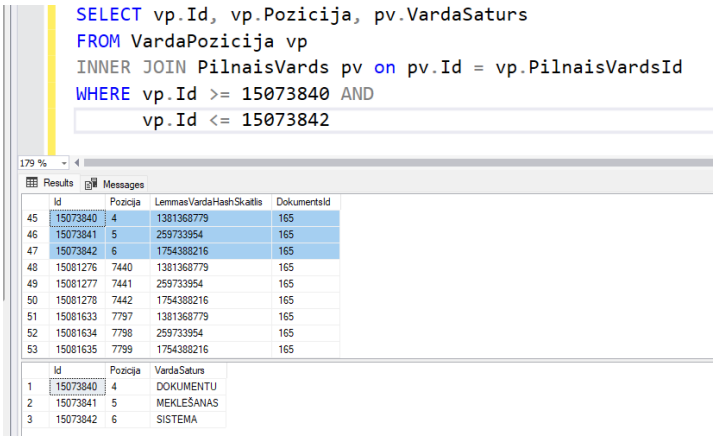
Iepriekš aprakstītā "Frāzes pozīcijas" skata versija atgriež visas pozīcijas, kuru vārdi atbilst kādam no meklētās frāzes vārdiem, un tad šīs pozīcijas tiek ielasītas operatīvajā atmiņā, kur tajās programma solī 3. atrod visas frāzes, tas ir pietiekami efektīvs risinājums situācijas, kur atgrieztās pozīcijas ir daži simti līdz daži tūkstoši. Situācijās kur atgrieztās pozīcijas būs ap simts tūkstošiem un vairāk jau būs manāma ātrdarbības degradēšanās. Tātad risinājums ir visu frāzei piederošo pozīciju atlasīšanu jau veikt servera pusē izmantojot to pašu skatu, līdz ar to arī situācijā kur iepriekšējais skats atgrieztu simts tūkstošus ierakstus, šis variants atgrieztu aptuveni dažus simtus ierakstu būtiski uzlabojot kopējo programmas izpildes laiku.

Liela nozīme ir tabulai "Frāzes vārds", jo tajā uz meklēšanas brīdi atrodas visi meklētās frāzes vārdi ar jaucējfunkciju iegūta skaitļa formā. Skriptā otrais un trešais šķēlums kopumā atšķel nost tās pozīcijas, kas nav katra meklētās frāzes pirmā vārda pozīcija plus meklētās frāzes vārdu skaits mīnus viens. Tātad tas, kas paliek pāri izpildot abus šķēlumus ir tikai tās pozīcijas, kas sākas no pirmā frāzes vārda un beidzas ar pēdējo frāzes vārdu. Šo var realizēt izmantojot dokumentu indeksa struktūras īpašību, ka visas tabulas "Vārda pozīcija" lauka "Id" vērtības ir pēc kārtas pieaugošā secībā, tieši tā pat kā oriģināla dokumenta vārdi.

```
CREATE VIEW [dbo].[FrazesPozicijas] AS
SELECT vp.Id, vp.Pozicija, vp.LemmasVardaHashSkaitlis, vp.DokumentsId
FROM VardaPozicija vp
INNER JOIN (
    SELECT COUNT (VardaHashSkaitlis) - 1 AS skaits FROM FrazesVards
) fsk ON fsk.skaits > 0
INNER JOIN (
    SELECT vp.Id FROM VardaPozicija vp
    WHERE vp.LemmasVardaHashSkaitlis = (SELECT TOP(1) fv.VardaHashSkaitlis FROM FrazesVards fv)
) firstw ON vp.Id >= firstw.Id
INNER JOIN (
    SELECT vp.Id FROM VardaPozicija vp
    WHERE vp.LemmasVardaHashSkaitlis = (SELECT TOP(1) fv.VardaHashSkaitlis FROM FrazesVards fv ORDER BY Id DESC)
) lastw ON firstw.Id = lastw.Id - fsk.skaits AND
           vp.Id <= lastw.Id + fsk.skaits
WHERE
    vp.LemmasVardaHashSkaitlis IN (SELECT VardaHashSkaitlis FROM FrazesVards)
```

3.10. att. **“Frāzes pozīcijas” skata uzlabotais SQL skripts**

Attēlā 3.11. pirmajā rezultātu tabulā redzama daļa no optimizētā “Frāzes pozīcijas” skata atgrieztajām vērtībām, kas iegūtas meklējot frāzi “Dokumentu meklēšanas sistēma”, iezīmētā daļa ir viena no atrastajām frāzēm. Nākamajā rezultātu tabulā šīs “Id” lauka vērtības izmantotas, lai atlasītu tabulu “Vārda pozīcija” un “Pilnais vārds” ierakstus, lai varētu attēlot atrastās frāzes oriģinālo saturu.



3.11. att. **“Frāzes pozīcijas” uzlabotā skata atgrieztās vērtības**

Algoritms

Arī šis algoritms tiek aprakstīts saskaņā ar apakšnodaļā “Kopīgie meklēšanas procesa veicamie uzdevumi” definētajiem meklēšanas uzdevumiem.

1.1. Katram no ievadītās simbolu virknes izgūtajam vārdam:
    1.1.1. Vārds tiek pārveidots lemmas formā, un tad ar jaucējfunkciju no tā iegūst veselu skaitli;
    1.1.2. Vārda iegūtais veselais skaitlis tiek pievienots vārdu sarakstā;
1.2. Vienā ievietošanas operācijā visas vārdu veselo skaitļu saraksta vērtības tiek ievietotas datu bāzes tabulā "Frāzes vārds";
1.3. Pozīciju saraksta vērtības tiek iegūtas no SQL skata "Frāzes pozīcijas"
3.1. Katrai pozīciju sarakstā atrastajai frāzei:
    3.1.1. Pievieno frāzi atrasto frāžu sarakstā;
3.2. Vienā ievietošanas operācijā visas atrasto frāžu saraksta vērtības (Frāzes pirmā un pēdējā vārda tabulas "Vārda pozīcija" lauka "Id" vērtības) tiek ievietotas datu bāzes tabulā "Atrastās frāzes pozīcijas";
3.3. Rezultātu saraksts tiek iegūts no SQL skata "Frāzes konteksts";
3.4. Datu bāzē tiek izpildīta operācija ierakstu izdzēšanai no tabulām "Frāzes vārds" un "Atrastās frāzes pozīcijas".

Testēšanas kopsavilkums

Katrs testēšanas rezultāts katram solim tiek iegūts 5 reizes izpildot programmu. Lai labāk demonstrētu atšķirību izpildes laikā, sākumā tiek notestēts laiks bez "Frāzes pozīcijas" skata optimizācijas, un tad uz lielāka apjoma testa datiem tiek notestēts optimizētais variants.

3.5. tabula

Alternatīvā SQL skatu varianta testēšanas rezultātu kopsavilkums

Meklētās frāzes teksts	Vidējais laiks soļiem 1-2 (ms)	Vidējais laiks solim 3 (ms)	Atrasto frāžu skaits dokumentu kolekcijā	Apakšnodaļas "Testpiemēru dati" lietotais testpiemērs	"Frāzes pozīcijas" skata optimizācija

dokumentu meklēšanas	14	12	13	1	Nē
datu struktūra	17	19	65	1	Nē
dokumentu meklēšanas sistēma	16	13	3	1	Nē
sanāk glabāt diezgan daudz nulles kas	36	14	1	1	Nē
dokumentu meklēšanas sistēma	171	21	33	2	Nē
dokumentu meklēšanas sistēma	10	18	33	2	Jā
sanāk glabāt diezgan daudz nulles kas	658	34	1	2	Nē
sanāk glabāt diezgan daudz nulles kas	15	16	1	2	Jā

Secinājumi

Šis variants ir ievērojami ātrāks situācijās kur ir vai nu garāka meklētā frāze, vai arī vairāk atrastās frāzes, kas ir diezgan būtisks aspekts dokumentu meklēšanas sistēmas ātrdarbībai.

Testēšanas rezultāti labi demonstrē, ka pie mazāka indeksa struktūras apjoma testpiemērā 2, kur ir tikai ap miljonu noindeksēti vārdi ātrdarbības problēma skatam "Frāzes pozīcijas" nebija tik jūtama, kā testpiemērā 3, kur ir noindeksēti 18 miljoni vārdi un aprakstītā algoritma solī 1.3. pozīciju sarakstā sanāca ielasīt desmitus tūkstošus un vairāk pozīciju. Testpiemērā kur kolonnā

“Vidējais laiks soļiem 1-2” izpildes laiks ir 658ms, solī 1.3. tika ielasītas simts tūkstoš pozīcijas, bet optimizētajā variantā tikai 77 pozīcijas.

Salīdzinājumā ar iepriekšējajiem variantiem šis variants ir mazāk atkarīgs no ievades datiem, jo iepriekšējos variantos meklējot frāzi “datu struktūra”, solis 3. bija 4 un pat 5 reizes lēnāks nekā meklējot frāzi “dokumentu meklēšanas”, bet šajā variantā tas ir tikai uz pusi lēnāks, kas ir ievērojama atšķirība.

3.4.9. Testēšana uz dažādiem apjomiem

Testēti tiek divi labākie varianti “Alternatīvais variants ar SQL skatiem” izmantojot skata “Frāzes pozīcijas” optimizāciju un “Alternatīvais variants frāžu meklēšanai”, lai saprastu kurš no tiem būs efektīvāks pie dažāda apjoma dokumentu indeksa datiem. Laiks tiek mērīts visu soļu izpildei kopā sākot no meklēšanas funkcijas sākuma līdz pat rezultātu atgriešanai. Katrs vidējais laiks tiek iegūts veicot 5 testēšanas piegājienus. Kolonnā “Variants” ar “Alt” tiek apzīmēts “Alternatīvais variants frāžu meklēšanai” un ar “SQL” tiek apzīmēts “Alternatīvais variants ar SQL skatiem”.

3.6. tabula

Apjoma testēšanas rezultātu kopsavilkums

Meklētās frāzes teksts	Variants	Vidējais meklēšanas izpildes laiks (ms)	Atrasto frāžu skaits dokumentu kolekcijā	Apakšnodaļas “Testpiemēru dati” lietotais testpiemērs
dokumentu meklēšanas	Alt	18 738	154	2
dokumentu meklēšanas	SQL	34	154	2
sanāk glabāt diezgan daudz nulles kas	Alt	4 592	10	2
sanāk glabāt diezgan daudz nulles kas	SQL	31	10	2
neeksistējoša frāze	Alt	25	0	2
neeksistējoša frāze	SQL	5	0	2

dokumentu meklēšanas	Alt	102 856	41	3
dokumentu meklēšanas	SQL	40	41	3
dokumentu meklēšanas sistēma	SQL	66	8	3
sanāk glabāt diezgan daudz nulles kas	SQL	219	3	3

Secinājumi

Jāņem vērā, ka lietotājs visticamāk nevēlēsies apskatīt 100 un vairāk meklēšanas rezultātus, bet tā vietā pārformulēs meklēšanas ievadi tā lai atrastu vēlamo pārskatot mazāku rezultātu kopu. Tāpēc ir jāparedz programmā iespēja lietotājam noteikt cik pirmās frāzes viņš vēlas atrast un programmu strukturēt tā, lai tā izdarītu darbu tikai tik frāzēm cik lietotājs noteicis, ka viņš vēlas atrast. Tādējādi var izsargāties no situācijas, kur meklēšanas programma nostrādā ilgāk un atgriež lietotājam vairākus simtus frāžu kuras viņš nemaz nepārbaudīs.

3.4.10. Dokumentu meklēšanas nobeiguma secinājumi

Datu struktūras nosaka datu uzglabāšanas formu un tā var ierobežot vai arī paplašināt vērtīgas iespējas algoritmiem kuri tās apstrādā. Līdz ar to arī datu struktūrām ir vērā ņemama ietekme uz dokumentu meklēšanas programmas izpildes ātrumu, tas tiek apliecināts salīdzinot variantus “Pamatvariants” un “Alternatīvais variants frāžu meklēšanai”. Alternatīvais variants frāžu meklēšanai sniedz iespēju paveikt to pašu uzdevumu 1, kas pie kopīgajiem meklēšanas uzdevumiem ir Pozīciju saraksta iegūšana, veicot mazāk operācijas un līdz ar to arī tiek uzlabots šī uzdevuma paveikšanas kopējais laiks.

Neapšaubāmi ir skaidrs arī tas, ka liela ietekme uz datu struktūru apstrādes ātrumu ir programmai un algoritmiem, kas to organizē. Algoritmiem ir jābūt realizētiem tā, lai tie pēc

iespējas efektīvāk spētu izmantot datu struktūru sniegtās vērtīgās iespējas pie iespējami mazāku soļu skaita, tādējādi izvairoties no liekām darbībām.

Veidojot asinhronas programmas daudzkodolu procesoru arhitektūrā ir iespējams būtiski uzlabot programmas darbības laiku un ja visas pārējās iespējas jau ir izmantotas, tad šis ir labs risinājums, kā vēl var padarīt programmu efektīvāku. Tas labi tiek nodemonstrēts uzlabojot darbā aprakstīto “Pamatvariants” algoritma realizāciju.

Liela nozīme ir infrastruktūras videi un tās lietojumam. Ignorējot infrastruktūras iespējas un ierobežojumus izveidojot citādi efektīvus algoritmus un datu struktūras to vēlamais efekts var netikt sasniegts, ja kādi infrastruktūras ierobežojumi tam traucē.

REZULTĀTI

Darbā ir uzmodelēti, aprakstīti un izanalizēti 3 varianti datu struktūrām “Dokumentu indeksēšanas datu struktūras pamatvariants”, “Alternatīvais variants dokumentu glabāšanai indeksa struktūrā”, “Alternatīvais variants dokumenta vārdu pozīcijām”, kuras sniedz iespēju indeksēt dokumentus un to tekstuālo saturu un veikt efektīvu dokumentu meklēšanu. Vienam variantam ir aprakstītas struktūras izveidošanas un atjaunināšanas operācijas, kā arī funkcijas kuras ir iespējams realizēt izmantojot attiecīgo struktūru, bet tā kā visas darbā aprakstītās datu struktūras ir veidotas pēc līdzīgiem principiem, tad šīs lietas ar minimālām izmaiņām ir attiecināmas uz visām trim aprakstītajām datu struktūrām. Katrai datu struktūrai ir attēlota un definēta tās struktūra izmantojot klašu diagrammu un tās darbības principi ir nodemonstrēti uz konkrētiem piemēriem izmantojot instanču diagrammas. Dokumentu meklēšanas risinājumu nodaļā pamatojoties uz testēšanas rezultātiem tiek noteikts, ka dokumentu meklēšanai pēc frāzes visspiemērotākā datu struktūra ir “Alternatīvais variants dokumenta vārdu pozīcijām”.

Tika izmēģinātas un izpētītas citu autoru izveidotas meklēšanas programmas, kas sniedz līdzīgas meklēšanas iespējas ar bakalaura darbā risināto meklēšanas problēmu. Līdz ar to tiek sniegts apkopojums autora piedāvāto datu struktūru sniegto iespēju salīdzinājumam ar iespējām ko sniedz citu autoru meklēšanas programmas.

Autors ir apkopojis un aprakstījis būtiskākos teorētiskos pamatus tehnoloģijām, kas tiek izmantotas datu struktūru uzglabāšanas un atjaunināšanas praktiskajās realizācijās. Aprakstīti ir praktiski realizēti divi dažādi varianti indeksa struktūras izveidošanai, kur viens no tiem datu bāzes operācijas veic izmantojot ORM ietvaru “Entity framework”, bet otrs tās veic izmantojot SQL statiskās procedūras. Kopumā abiem variantiem ir aprakstītās, realizētas un notestētas 17 dažādas modifikācijas. Pamatojoties uz testēšanas rezultātiem tiek izskaidrotas un pamatotas realizētās programmu optimizācijas, kas uzlabo izpildes laika ātrdarbību, kā arī ir noteikts ātrākais variants indeksa struktūras izveidošanai. Izpētītās un notestētās optimizācijas ir derīgas ne tikai tieši dokumentu meklēšanas problēmai, bet arī citiem līdzīga rakstura uzdevumiem, kur jāveic lieli datu ievietošanas uzdevumi datu bāzē.

Meklēšanas procesa veicamie uzdevumi un to soļi ir aprakstīti un arī attēloti izmantojot instanču diagrammas. Izmantojot aktivitāšu diagrammas ir attēlota programmas sinhrona un arī asinhrona izpildes plūsma. Kopumā ir realizēti un uz ātrdarbību notestēti 3 dažādi dokumentu

meklēšanas pēc frāzes varianti. Pamatojoties uz testēšanas rezultātiem tiek noteikta pēc programmas izpildes laika ātrākā realizācija, kas ir aprakstītais "Alternatīvais variants ar SQL skatiem" variants. Testēšanas rezultāti liecina, ka šis variants ir spējīgs atrast frāzes un uzkonstruēt to priekšskatījumu dokumentos, kuru vārdu kopskaits ir trīs simts miljoni vidēja garuma meklēšanas frāzēm ap 3-6 vārdi 40-250 milisekunžu laikā, bet patērētais laiks ir atkarīgs no frāzes garuma un atgriezto rezultātu skaita.

SECINĀJUMI

Indeksa struktūru, kas sniedz iespēju indeksēt dokumentus un to saturu priekš dokumentu meklēšanas programmas var īstenot izmantojot dažādas datu struktūras, un to strukturālās īpašības sniedz noteiktas priekšrocības vai trūkumus. Tāpēc lai varētu noteikt efektīvāko datu struktūru priekš dokumentu meklēšanas programmas tika veikts praktisks pētījums. Darba teorētiskās un praktiskās daļas rezultātā tiek veikti attiecīgi secinājumi:

- Darbā aprakstītais dokumentu indeksēšanas datu struktūras variants "Alternatīvais variants dokumentu glabāšanai" sniedz iespējas veikt ātrdarbības ziņā efektīvāku struktūras atjaunošanu, bet lietojot šo variantu tiek zaudēta īpašība, kur viena dokumenta visas pozīcijas ir secīgas un blakus, līdz ar to arī būtu zaudējumu dokumentu meklēšanas ātrumam, un tā kā dokumentu meklēšanas ātrums ir prioritāte un struktūras atjaunošanu var veikt automatizēti, kā fona procesu lietotājam par to nemaz nezinot, nav ieteicams lietot šo datu struktūru.
- Pamatojoties uz dokumentu meklēšanas testēšanas rezultātiem tiek pierādīts, ka datu struktūras variants "Alternatīvais variants dokumenta vārdu pozīcijām" tiešām ir ātrākais variants un līdz ar to arī ir vispiemērotākā datu struktūru dokumentu meklēšanas programmai.
- Datu struktūrai "Alternatīvais variants dokumenta vārdu pozīcijām" dokumentu meklēšanas sadaļā tiek realizēti divi dažādi varianti un vienam no tiem vēl būtiska optimizācija un visiem šiem variantiem ir atšķirīgi dokumentu meklēšanas ātrdarbības rezultāti, kas nozīmē to, ka svarīgi ir ne tikai lietot datu struktūru, kas sniedz labākas iespējas, bet arī spēt efektīvi tās izmantot.
- Dokumentu meklēšanas realizācija "Alternatīvais variants ar SQL skatiem", kur tika pielietoti SQL skati un programma tika pārstrukturēta tā, lai visas datu bāzes datu atlasīšanas notiktu konstantā reižu skaitā un katrā atlasīšanā atlasītu pēc iespējas mazāk liekus ierakstus, tādējādi samazinot datu apmaiņas satura apjomu un arī reižu skaitu ar datu bāzes serveri tika būtiski uzlabots meklēšanas programmas izpildes laiks. Tātad lai dokumentu indeksā varētu implementēt patiesi efektīvu dokumentu meklēšanu, tai ir jābūt kvalitatīvi un pareizi realizētai visās dimensijās: datu struktūrās, programmas realizācijā, infrastruktūras lietojumā.

- Izmantojot asinhrono programmēšanu dokumentu meklēšanas realizācijā “Frāžu meklēšanas pamatvariants” tika būtiski uzlabots programmas izpildes laiks, taču variantā “Alternatīvais variants ar SQL skatiem” kur programma tika pārstrukturēta un optimizēta, lai efektīvi lietotu SQL servera iespējas asinhronā programmēšana nemaz nebija nepieciešama un šis variants strādāja būtiski ātrāk arī bez tās.
- Izpētot līdzīgas citu autoru veidotas meklēšanas programmas tiek secināts, ka neeksistē programma, kas sniedz tieši tādas pašas funkcijas, kā bakalaura darba autora piedāvātais variants.
- ORM ietvaru nav ieteicams lietot dokumentu indeksa struktūras izveidošanā, jo tas vairāku iemeslu dēļ strādā būtiski lēnāk, nekā alternatīvs SQL variants, kurš lieto SQL statiskās procedūras, kā arī sniedz iespēju realizēt vairākas citas ātrdarbības optimizācijas.
- SQL servera indeksiem ir ļoti liela ietekme uz ātrdarbību un pareiza to lietošana apvienojumā ar citām optimizācijām ir izšķiroša loma meklēšanas programmas ātrdarbībai.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] Entity Framework, “Linq-to-Entities Query” [tiešsaiste] [skatīts 01.05.2022] Pieejams: https://www.entityframeworktutorial.net/querying-entity-graph-in-entity-framework.aspx
[2] Microsoft, “SQL Server and Azure SQL index architecture and design guide” [tiešsaiste] [skatīts 01.05.2022] Pieejams: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver15
[3] SQLShack, “Designing effective SQL Server clustered indexes” [tiešsaiste] [skatīts 02.05.2022] Pieejams: https://www.sqlshack.com/designing-effective-sql-server-clustered-indexes/
[4] Esat Erkec, “Using Stored Procedures with Return Values” [tiešsaiste] [skatīts 04.05.2022] Pieejams: https://www.sqlshack.com/using-stored-procedures-with-return-values/#:~:text=Return%20Value%20in%20SQL%20Server,zero%20values%20indicate%20an%20error
[5] Jason Beres, “The Power and Potential of Stored Procedures” [tiešsaiste] [skatīts 04.05.2022] Pieejams: https://devops.com/the-power-and-potential-of-stored-procedures/#:~:text=Stored%20procedures%20are%20among%20the,servers%20and%20clients%2C%20for%20example
[6] Microsoft, “Automatic detect changes” [tiešsaiste] [skatīts 10.04.2022] Pieejams: https://docs.microsoft.com/en-us/ef/ef6/saving/change-tracking/auto-detect-changes
[7] Microsoft, “Efficient Querying” [tiešsaiste] [skatīts 11.04.2022] Pieejams: https://docs.microsoft.com/en-us/ef/core/performance/efficient-querying#use-indexes-properly
[8] Rajendra Gupta, “SET NOCOUNT ON statement usage and performance benefits in SQL Server” [tiešsaiste] [skatīts 11.04.2022] Pieejams: https://www.sqlshack.com/set-nocount-on-statement-usage-and-performance-benefits-in-sql-server/
[9] Ed Pollack, “How to solve the SQL Identity Crisis in SQL Server” [tiešsaiste] [skatīts 12.04.2022] Pieejams: https://www.sqlshack.com/solve-identity-crisis-sql-server/
[10] Microsoft, “Performance Considerations (Entity Framework)” [tiešsaiste] [skatīts 12.04.2022] Pieejams: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/performance-considerations\

[11] Priya Pedamkar, “What is ORM?” [tiešsaiste] [skatīts 09.04.2022] Pieejams: https://www.educba.com/what-is-orm/
[12] Stanford Natural Language Processing Group, “Building an inverted index” [tiešsaiste] [skatīts 20.05.2022] Pieejams: https://nlp.stanford.edu/IR-book/html/htmledition/a-first-take-at-building-an-inverted-index-1.html
[13] Sodel Vázquez-Reyes, María de León-Sigg, Perla Velasco-Elizondo, Juan Villa-Cisneros, Sandra Briceño-Muro, “The Use of Inverted Index to Information Retrieval” [tiešsaiste] [skatīts 20.05.2022] Pieejams: https://ingsoftware.reduaz.mx/~pvelasco/files/2016-TheUseofInvertedIndex.pdf
[14] Stacy Fisher “8 Best Free File Search Tools” [tiešsaiste] [skatīts 21.05.2022] Pieejams: https://www.lifewire.com/11-free-file-search-tools-1356644
[15] Cometdocs blog “Best Free Software for Searching Through Multiple Word, Excel, PDF and Other Textual Files” [tiešsaiste] [skatīts 20.05.2022] Pieejams: https://blog.cometdocs.com/best-free-software-for-searching-through-multiple-word-excel-pdf-and-other-textual-files
[16] Mythicsoft, “Agent Ransack” [tiešsaiste] [skatīts 21.05.2022] Pieejams: https://www.mythicsoft.com/agentransack/
[17] Arnolds Sarmulis (2022). Kursa darbs “Dokumentu meklēšanas sistēma”