LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FALULTĀTE
DATORIKAS NODAĻA

UZ MS SQL BĀZĒTU SISTĒMU VEIKTSPĒJAS PROBLĒMU MEKLĒŠANA UN RISINĀŠANA

BAKALAURA DARBS

Autors:

Ģirts Upenieks

Stud. apl. Mate980015

Darba vadītājs:

LU FMF Datorikas nodaļa asoc.prof.

Dr.sc.comp. Guntis Arnicāns

RĪGA 2008

ANOTĀCIJA

Šī darba mērķis ir izpētīt veidus kā uzlabot uz Microsoft.NET platformas un Microsoft SQL servera būvētu biznesa sistēmu veiktspēju un mērogojamību. Šajā darbā tiek veikta sistēmas ātrdarbības problēmu izpēte, analīze un ieteikumu izveide. Par analīzes pamatu ir izvēlēta esoša apdrošināšanas un finansu sistēma. Rezultātā ir iegūti sistēmas ātrdarbības uzlabojumu ieteikumi un to lietderīguma pamatojumi, piedāvāta arhitektūras izmaiņa apskatītajai sistēmai un izstrādāts un pārbaudīts tās prototips, kā arī ieteikumi tālākai sistēmas attīstībai, lai palielinātu tās ātrdarbību un mērogojamību.

Atslēgvārdi: sql, linq, wcf, msmq, rindas, ātrdarbība, mērogojamība

ABSTRACT

The purpose of this diploma work is to research opportunities and ways to improve productivity and scalability of business systems built on Microsoft.NET platform and Microsoft SQL server. In this work performance problem research and analysis is done and proposals submitted. Existing insurance and finance system is used for analysis. As a result system performance suggestions are verified and proposed; system architecture's change; verified architecture prototype and offered proposals for future system's development to increase its performance and scalability.

Keywords: sql, linq, wcf, msmq, rows, performance, scalability

SATURS

APZĪMĒJUMU SARAKSTS

Apzīmējums	Skaidrojums
MS SQL	Microsoft izstrādāta datu bāzu pārvaldības sistēma.
LINQ	Language INtegrated Query – Programmēšanas valodas sintaksē iestrādāta iespēja ērti apstrādāt jebkāda veida datus.
WCF	Microsoft piedāvāts risinājums ziņojumu apmaiņai un attālinātai servisu izsaukšanai.
MSMQ	Microsoft piedāvāts risinājums ziņojumu pārsūtīšanai starp aplikācijām izmantojot rindas.
Deadlock	Strupsaķere - Situācija datu apstrādes sistēmā, kad divi procesi, kas pieprasa vienus un tos pašus resursus, bloķē viens otru.
Lazy-Load	Ielāde pēc nepieciešamības. Apzīmē paņēmienu ko izmanto lai ielādētu datus, vai inicializētu objektu tikai tad kad tas nepieciešams. [1]

IEVADS

Šajā darbā tiek risināta uz Microsoft tehnoloģijām būvētu biznesa aplikāciju ātrdarbības problēma. Šīs problēmas rodas vairāku iemeslu dēļ – nepietiekoši izplānota sistēmas arhitektūra, slikti realizēta arhitektūra, nepareizi izvēlētas tehnoloģijas, u.c.. Kā vienu no netiešiem iemesliem var minēt to, ka aplikācijas izstrādes procesā visi nepieciešamie resursi ir ierobežoti, un tāpēc vienmēr paliek kaut kas ne līdz galam izplānots un realizēts.

Praktiski visas biznesa aplikācijas, kas tiek būvētas uz Microsoft tehnoloģijām izmanto .NET platformu un datu bāzi, tāpēc arī šajā darbā tiek apskatītas uz .NET platformas būvētas aplikācijas, kas strādā ar Microsoft SQL datu bāzi. Microsoft SQL Servera gadījumā netiek piedāvāts universāls risinājums, kā sadalīt datubāzi uz vairākiem serveriem (klasteriem) un tādejādi palielināt tās kopējo produktivitāti bez papildus izmaiņām sistēmā. Pašas datu bāzes ātrdarbības uzlabošanas izpēte tika veikta Kursa darbā [7], tāpēc šajā darbā uzmanība tiek pievērsta tieši aplikācijai un tās darbībai ar datu bāzi.

Visas aplikācijas ātrdarbību var uzlabot gan veicot izmaiņas programmas kodā, gan pārplānojot un uzlabojot vidi, kurā aplikācija tiek darbināta. Aplikācijas darba vides uzlabošanu var veikt gan palielinot esošo serveru jaudu, gan pievienojot papildus serverus. Otrajā gadījumā aplikācijas arhitektūrā ir jābūt iespējai visu aplikāciju, vai atsevišķus tās moduļus darbināt vienlaicīgi uz vairākiem serveriem.

Kā viens no paša Microsoft ieteiktajiem risinājumiem ir jaunāko tehnoloģiju ieviešana[16], tādejādi iegūstot papildus iespējas aplikāciju izstrādē un ātrdarbībā. Viena no lietām uz ko orientējas jaunās tehnoloģijas ir ātrdarbības uzlabošana, piemēram, .NET 2 salīdzinājumā ar .NET 1 ir ātrāks, kā arī .NET 3 salīdzinājumā ar .NET 2 ir ātrāks. Bet ne vienmēr ieviešot jaunās tehnoloģijas uzreiz tiek iegūts ātrdarbības uzlabojums, jo parasti jaunās tehnoloģijas tiek orientētas uz kādas šaurākas problēmas risināšanu un tās ātrdarbības uzlabošanu.

Kā piemērs šajā darbā tiek apskatīta strādājoša *apdrošināšanas un finansu sistēma*. Sistēma tika būvēta 6 gadu garumā. Sākotnēji izstrādājot sistēmu tā tika plānota kā salīdzinoši neliela aplikācija. Laika gaitā tā pārauga par lielu daudzlietotāju sistēmu, bet sistēmas arhitektūra palika gandrīz nemainīga. Sistēmā tika veikti datu bāzes ātrdarbības uzlabošanas darbi, kas deva jūtamu rezultātu. Bet lai turpinātu palielināt sistēmas ātrdarbību bija nepieciešams veikt lielākas izmaiņas sistēmā. Kā viena no galvenajām izmaiņām, kas tika konstatēta kā nepieciešana ir sistēmas arhitektūras izmaiņas lai to varētu darbināt uz vairākiem aplikāciju serveriem.

Pārbaudot jaunās tehnoloģijas un to iespējamo pielietojumu esošajā apdrošināšana un finansu sistēmā tika konstatēts, ka tās ne vienmēr dod garantētu ātrdarbības uzlabojumu. Tāpēc ir nepieciešams katru tehnoloģiju izpētīt sīkāk un atrast tai labāko pielietojumu sistēmas arhitektūrā.

Šī darba 1. nodaļā tiek apskatīti pasaulē pazīstamākie paņēmieni, kā uzlabot aplikāciju ātrdarbību, gan tās projektēšanas un izstrādes laikā, gan pēc tam, kad notiek saskaršanās ar konkrētām problēmām.

2. nodaļā tiek apskatīta viena konkrēta uz Microsoft tehnoloģijām būvēta apdrošināšana un finansu sistēma, un izanalizēta kādas ātrdarbības problēmas tajā ir balstoties uz 1. nodaļā minētajiem paņēmieniem.

3. nodaļā tiek izpētīti vairāki paņēmieni sistēmu ātrdarbības uzlabošanai. Tāpat tiek praktiski pārbaudīti izskatītie risinājumi, un piemēroti 2. nodaļa apskatītajai sistēmai un tās ātrdarbības problēmām. Tiek izveidota iespējamā sistēmas arhitektūra un pēc tā uzbūvēta konceptuāla sistēma, kas atbilst apskatāmās sistēmas vienam scenārijam un tiek salīdzināta tā ātrdarbība ar esošās arhitektūras risinājumu. Iegūtie rezultāti apliecina, ka izmainītā arhitektūra un izmantotās tehnoloģijas ir piemērotas esošās sistēmas ātrdarbības uzlabošanai.

1. PAŅĒMIENI SISTĒMU ĀTRDARBĪBAS UZLABOŠANAI

1.1. Datubāzes noslodzes izpēte

Lai meklētu datubāzes noslodzes problēmas var izmantot gan Windows piedāvātos sistēmas skaitītājus (performance counters), gan SQL piedāvāto servera pārskata instrumentu (SQL Server Profiler). Lai varētu noteikt datubāzes problēmas ir iespējams apskatīt sekojošus aspektus:

- Pieslēgumu pūls (connection pool) – nepieciešams apskatīt pieslēgumu izlietošanas un pūļa lietošanas efektivitāti.
- SQL pieprasījumi – nepieciešams apskatīt pieprasījumu izpildes laikus un to efektivitāti
- Indeksi – indeksu efektivitātes pārbaude meklēšanas gadījumos
- Keša izmantošanu – Keša izmantošanas efektivitāti
- Transakcijas – transakciju skaitu sekundē, konkurējošās transakcijas
- Datubāzes objektu slēgšana (locks) – pārbaudīt tabulu slēgšanas un rindu slēgšanas ietekmi uz ātrdarbību, ka arī vidējo laiku kas patērēts uz gaidot uz noslēgtu objektu, kā arī strupsaķeru (deadlock) skaitu.

Izmantojot Windows piedāvātos sistēmas skaitītājus, ir iespējams analizēt sekojošus skaitītājus, kas ir saistīti ar datubāzes servera izmantošanu:

Pieslēgumu pūļa pārraudzīšanai var izmantot sekojošus skaitītājus:

.*NET CLR Data\ SqlClient: Current # connection pools*

- procesam piesaistīto pūļu skaits

.*NET CLR Data\ SqlClient: Current # pooled connections*

- Procesam piesaistīto pūļu pieslēgumu kopējais skaits

.*NET CLR Data\SqlClient: Peak # pooled connections*

- Maksimālais Procesam piesaistīto pūļu pieslēgumu kopējais skaits kopš procesa sākuma

.*NET CLR Data\SqlClient Total # failed connects*

- Kopējais neizdevušos pieslēgumu atvēršanas mēģinājumu skaits

Indeksu efektivitātes pārbaudei var izmantot sekojošus skaitītājus:

SQL Server: Access Methods\Index Searches/sec

- Indeksu meklēšanas izmantošanas skaits sekundē. Indeksu meklēšana tiek izmantota lai uzsāktu apgabala skenēšanu (range scan), viena ieraksta atrašanai pēc indeksa, nopozicionēšanās pašā indeksā.

SQL Server: Access Methods\Full Scans/sec

- Pilno tabulas skenēšanas vai pilno indeksa skenēšanas skaits sekundē. Jo mazāk ir pilno skenēšanu, jo labāk, jo tā ir viena no vislēnākajām operācijām datubāzē.

Lai izmērītu keša izmantošanu var izmantot sekojošus skaitītājus:

SQL Server: Cache Manager\Cache Hit Ratio

- Attiecība starp keša izmantošanu un datu izmantošanu kas neatrodas kešā

SQL Server: Cache Manager\Cache Use Counts/sec

- Katra keša objekta tipa izmantošanas skaits sekundē.

SQL Server: Memory Manager\SQL Cache Memory (KB)

- Kopējais dinamiskās atmiņas daudzums, ko datubāzes serveris izmanto dinamiskajam kešam.

Memory\Cache Faults/sec

- Cik bieži operētājsistēma meklē failu sistēmas datus kešā un neatrod un rezultātā meklē tos failu sistēmā. Reižu skaits sekundē. Šim skaitītājam jābūt pēc iespējas mazākam, jo griešanās pie failu sistēmas un datu ielāde ir salīdzinoši lēns process.

Lai izmērītu un analizētu transakcijas var izmantot sekojošus skaitītājus:

SQL Server: Databases\Transactions/sec

- Uzsākto transakciju skaits datubāzes serverī sekundē. Šis ir viens no galvenajiem rādītājiem datubāzes veiktspējā.

SQL Server: Databases\Active Transactions

- Aktīvo transakciju skaits datubāzē dotajā brīdī.

Lai izmērītu slēgto datubāzes objektu ietekmi var izmantot sekojošus skaitītājus:

SQL Server: Locks\ Lock Requests/sec

- Jaunu slēgumu izveides un esošo slēgumu veida maiņu skaits sekundē

SQL Server: Locks\ Lock Timeouts/sec

- Slēgumu skaits, kam iestājās noildze sekundē. Te tiek iekļauti arī NOWAIT slēguma pieprasījumi.

SQL Server: Locks\Lock Waits/sec

- Pieprasīto slēgumu skaits sekundē, kurus nevar izpildīt uzreiz, jo to bloķē kāds cits slēgums, un izsaucējam ir jāgaida līdz tas būs iespējams.

SQL Server: Locks\ Number of Deadlocks/sec

- Slēgumu skaits, kas nokļuva strupsaķerē (deadlock). Tipiska situācija ir ilgi izpildoša pieprasījuma vienlaicīga izpilde ar vairāku rindu izmaiņām. Šim skaitītājam ir jābūt pēc iespējas mazākam, vai ideālā gadījumā jābūt 0. Jo pretējā gadījumā ir nepieciešams aplikācijai šos gadījumus īpaši apstrādāt. Lai izvairītos no strupsaķerēm ir nepieciešamas izmaiņas transakciju izolācijas līmeņu dizainā. Tas parasti ir sarežģīts process.

SQL Server: Locks\Average Wait Time (ms)

- Vidējais slēguma gaidīšanas laiks tiem slēgumiem, kurus nevar izpildīt uzreiz.

SQL Server: Latches\Average Latch Wait Time (ms)

- Vidējais laiks, kas nepieciešams aiztures pieprasījumam (request for a latch) datubāzē. Aiztures pieprasījumi ir īstermiņa rindu slēgumi.

Tāpat datubāzes ātrdarbības uzlabošanai var izmantot Microsoft piedāvātās programmas Database Engine Tuning Advisor un SQL Server Profiler, kas tiek piedāvāts kopā ar SQL Server 2005. Ar Database Engine Tuning Advisor programmu var analizēt konkrētu datubāzes darbības scenāriju žurnālu un, baltoties uz to, iegūt ieteikumus gan datubāzes indeksu izmaiņām, gan citiem datubāzes parametriem. Ar Microsoft piedāvāto SQL Server Profiler var veikt SQL pieprasījumu analīzi pēc to izpildes laika, lasīšanas / rakstīšanas skaita, utt.

Analizējot datubāzes žurnālus, iespējams noteikt lēnākos procesora laiku / diska operāciju visvairāk izmantojošos pieprasījumus. Nemainot aplikācijas uzbūvi (pieprasījuma izsaukumu un atgriežamo datu kopu) ir ļoti ierobežotas iespējas uzlabot ātrdarbību. Ir iespējams mainīt pieprasījuma uzbūvi un datu bāzes indeksus.

1.2. Datubāzes mērogojamība

Viens no vienkāršākajiem veidiem, kā palielināt aplikācijas ātrdarbību ir palielināt esošo serveru jaudu.[6] Bet to nevar darīt bezgalīgi, jo serveru cena palielinās nevis proporcionāli jaudai, bet gan daudz straujāk. Tāpēc sistēmas arhitektūrai ir svarīgi atbilst plānotajam piegājienam, kā sistēmu plānots mērogot. Sistēmu iespējams mērogot divos veidos:

- „Uz augšu" (scale up) – palielinot servera jaudu
- „Uz āru" (scale out) – pieslēdzot papildus serverus

„Uz augšu" – nozīmē, ka tiek palielināta serveru jauda un tādā veidā iegūta papildus ātrdarbība. „Uz āru" – nozīmē, ka aplikācija tiek sadalīta pa vairākiem mazākas jaudas serveriem, tādejādi iegūstot lielāku summāro serveru jaudu un palielinātu aplikācijas ātrdarbību.

Apskatot Microsoft SQL Server salīdzinājuma ar Oracle, pirmajam netiek piedāvāta iespēja kā vienkāršā veidā vienu datubāzi sadalīt pa vairākiem serveriem, lai iegūtu lielāku kopējo ātrdarbību. Microsoft piedāvātais risinājums Microsoft Cluster Services, neskatoties uz tā nosaukumu piedāvā tikai iespēju automātiski pārslēgties no viena datubāzu servera uz citu gadījumā ja pirmais serveris dažādu iemeslu dēļ vairs nestrādā.

Microsoft piedāvā dažādus risinājumus datubāzu servera mērogojamībai:[14]

- Mērogojama koplietošanas datubāze;
- Vienādranga (peet-to-peer) replikācija;
- Saistītie serveri un dalītie pieprasījumi;
- Dalītie pieprasījumi;
- Dalīšana balstoties uz datiem;
- Servisu orientēta datu arhitektūra.

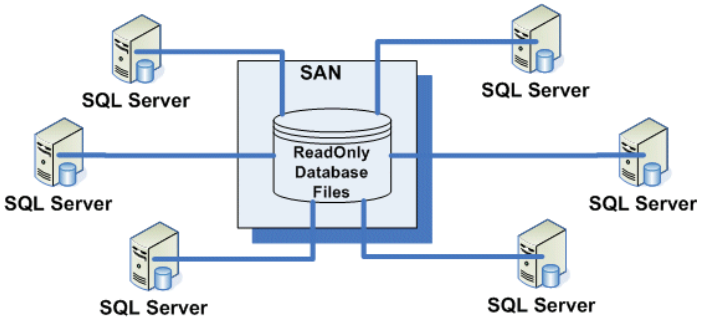
Lai izvēlētos iespējamo mērogojamības risinājumu, ir vairāki faktori, kuri ir jāņem vērā, jo katram mērogojamības risinājumam ir savi ierobežojumi. Šie faktori ir:

- Esošo ierakstu mainīšanas biežums. Ierakstus, kas tiek bieži mainīti ir grūti efektīvi replicēt, jo replicēšanai patērētais laiks kopsummā tikai samazinās kopējo sistēmas ātrdarbību.
- Iespēja veikt izmaiņas aplikācijā. Dažiem mērogošanas risinājumiem ir nepieciešamas lielas izmaiņas aplikācijā, citiem mazākas izmaiņas.
- Datu sadalīšana. Viens no efektīvākajiem veidiem kā mērogot datus ir to sadalīšana starp vairākiem serveriem tādā veidā, ka katrs serveris strādā ar saviem datiem.
- Datu saistība. Ja datubāzē ir dažādi dati ar kuriem strādā dažādas aplikācijas, tad tos var sadalīt starp diviem dažādiem datubāzu serveriem.

Mērogojama koplietošanas datubāze.

Vienkāršākais mērogojamības paņēmiens ir izveidot mērogojamas koplietošanas datubāzi. Tiek izveidota viena datubāze iekš SAN (Storage Area Network) un līdz pat 8 atsevišķi SQL serveri strādā ar šo datubāzi. Šim risinājumam ir viens liels ierobežojums – visiem datubāzu serveriem ir jāstrādā tikai lasīšanas režīmā.

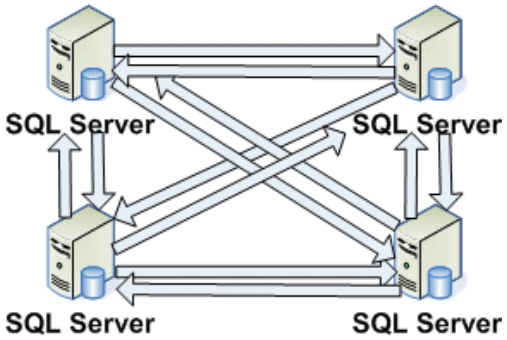
Šāds risinājums ir piemērots datu noliktavām vai arī vēsturisko datu analīzes sistēmām (Attēls 1.2.1.).



1.2.1. att. **Mērogojama koplietošanas datubāze**

Vienādranga (peet-to-peer) replikācija

Šis risinājums ir piemērojams gadījumos, ja ir nepieciešama datu mainīšana, bet tā notiek salīdzinoši reti. Šajā risinājuma katrs datubāzu serveris strādā ar saviem datiem, bet tie savā starpā tiek replicēti. Šis risinājums nenodrošina konfliktsituāciju risināšanu, tāpēc ir piemērojams tikai tad, ja viena datu elementa kopija tiek mainīta (Attēls 1.2.2.).



1.2.2. att. **Vienādranga (peer-to-peer) replikācija**

Saistītie serveri un dalītie pieprasījumi

SQL serverim ir iespēja veikt pieprasījumus pret citas datubāzes objektiem tā, it kā tie atrastos uz tā paša servera. Lai to izdarītu, SQL serverim ir jāpievieno saistītais serveris un SQL pieprasījumos ir objektam papildus jānorāda saistītais serveris, uz kura tas atrodas. Šāds risinājums nav piemērojams, ja no saistītā servera ir jāizmanto dati sarežģītos pieprasījumos, kur var būt nepieciešama visu datu saistīšana ar JOIN (Attēls 1.2.3.).

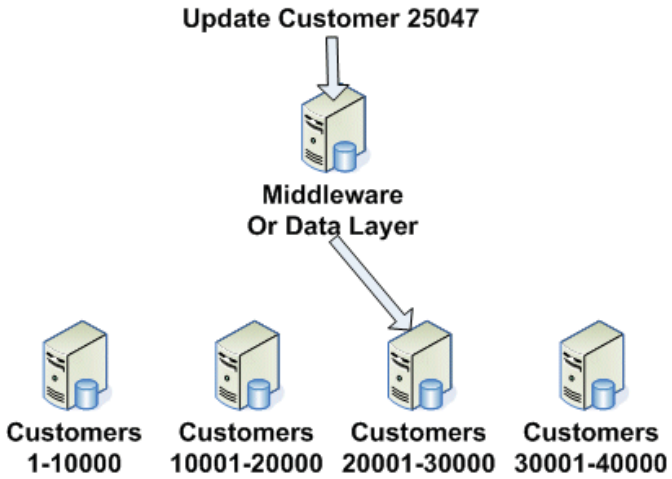
1.2.3. att. **Saistītie serveri un dalītie pieprasījumi**

Dalītie pieprasījumu skati (views)

Dalītie pieprasījumu skati ir pieejami lai aplikācijai padarītu neredzamus dalītos datus. Piemēram, Klientu tabula var tikt sadalīta pa vairākiem serveriem, pirmie 1000 klienti uz viena servera, otrie 1000 klienti uz cita servera.

Dalīšana balstoties uz datiem

Ar dalīto pieprasījumu skatiem datubāze zina, kur meklēt datus, kad tie ir nepieciešami kādam pieprasījumam. Ar Dalīšanu Balstoties uz Datiem (Data-Dependent Routing) dati tiek sadalīti starp datubāzēm un vai nu aplikācija vai kāds starpposma serviss nosūta pieprasījumu uz nepieciešamo datubāzi (Attēls 1.2.4.).



1.2.4. att. **Dalīšana balstoties uz datiem**

Servisu orientēta datu arhitektūra

Apvienojot Servisu Orientētu Arhitektūru un jaunās SQL 2005 servera iespējas ir izveidota Servisu Orientēta Datu Arhitektūra. Ar šo arhitektūru ir iespējams izvietot servisus uz dažādiem serveriem, kā arī datus sadalīt starp serveriem, tā, lai katram servisam būtu pieejami tikai viņam nepieciešamie dati.

Apkopojot visus piedāvātos risinājumus var izveidot pārskatāmu tabulu (Tabula 1.2.1.).

1.2.1. Tabula

Datubāzes mērogojamības risinājumi un to nosacījumi

	Izmaiņu biežums	***Izmaiņas aplikācijā***	***Datu sadalīšanas iespēja***	***Datu saistība***
Mērogojama koplietošanas datubāze	*Tikai lasīšana*	*Nav vai nelielas*	*Nav nepieciešama*	*Nav nepieciešama*
Vienādranga (peet-to-peer) replikācija	*Pamatā lasīšana, nav konfliktu*	*Nav vai nelielas*	*Nav nepieciešama*	*Nav nepieciešama*
Saistītie serveri un dalītie pieprasījumi	*Minimālas starp-datubāzu izmaiņas*	*Nelielas*	*Pamatā nav nepieciešama*	*Svarīgi lai ir maza datu saistība*
Dalītie pieprasījumi	*Biežas izmaiņas*	*Vidējas izmaiņas*	*Ļoti svarīga*	*Maza ietekme*
Dalīšana balstoties uz datiem (DBD)	*Biežas izmaiņas*	*Lielas izmaiņas*	*Ļoti svarīga*	*Maza saistība var palīdzēt*
Servisu orientēta datu arhitektūra	*Biežas izmaiņas*	*Ļoti lielas izmaiņas*	*Pamatā nav nepieciešama, izņemot saistībā ar DBD*	*Nepieciešama maza saistība starp servisiem*

1.3. Jaunu tehnoloģiju ieviešana

Viens no veidiem kā uzlabot sistēmas ātrdarbību ir ieviest jaunākās tehnoloģijas. Kā piemērs tiek apskatīta viena no jaunākajām Microsoft piedāvātajām tehnoloģijām – LINQ.

LINQ apskats

[3] LINQ (Language INtegrated Query) var uzskatīt par metodoloģiju, kas vienkāršo un unificē piekļūšanu un apstrādi jebkāda veida datiem. Ar LINQ nav nepieciešams izmantot kādu speciālu datu piekļuves arhitektūru. LINQ ietver sevī vairākas jau esošas datu piekļuves arhitektūras. Mūsdienās ir nepieciešamība apstrādāt dažāda veida datus – masīvi, objektu

grafi, XML dokumenti, datubāze, teksta fails, Windows reģistrs, e-pasts, SOAP (Simple Object Access Protocol) sūtījums, Microsoft Excel dokuments, utt.

Katram datu veidam ir savs datu piekļuves modelis. Piemēram, lai iegūtu datus no datubāzes tiek izmantoti SQL pieprasījumi. XML tiek apstrādāts ar DOM (Document Object Model) vai XQuery. Objektu kopa tiek apstrādāta ar speciāli rakstītām algoritmu konstrukcijām. Rezultātā lai apstrādātu dažādus datus ir nepieciešami daudz dažādi programmēšanas modeļi.

Jau vairākus gadus ir bijuši mēģinājumi unificēt datu piekļuves mehānismus. Piemēram, ir ODBC (Open Database Connectivity), kas ļauj veikt pieprasījumus vienlīdz gan no Microsoft Excel dokumenta gan teksta faila. Šajā gadījumā ar SQL-tipa valodu notiek pie visiem datiem kas ir attēloti relāciju datu modelī. Bet dažkārt datus ir daudz vienkāršāk uzturēt un hierarhiskā vai grafu modelī. Šādu neatbilstību dēļ parasti nākas veidot datu pārveidojumus. LINQ ir veidots lai atrisinātu šīs problēmas un unificētā veidā varētu apstrādāt dažāda tipa datus. LINQ izmanto unificētas operācijas ar datiem tā vieta lai unificētu pašas datu struktūras.

[11] LINQ ir programmēšanas modelis, kas piedāvā pieprasījumus veidot pa tiešo .NET programmēšanas valodā. [11] Piemēram, LINQ pieprasījums, kas atgriež klientu vārdus, kam uzvārdi ir „Bērziņš”:

```
var query =
from k in Klienti
where c.Uzvards == „Bērziņš”
select c.Vards
```

Visu rezultātu var ērti apskatīt ar FOREACH konstrukciju C# valodā:

```
foreach (string vards in query)
{
Console.WriteLine(vards);
}
```

Klienti var būt objektu kopa:

```
Customer[] Customers;
```

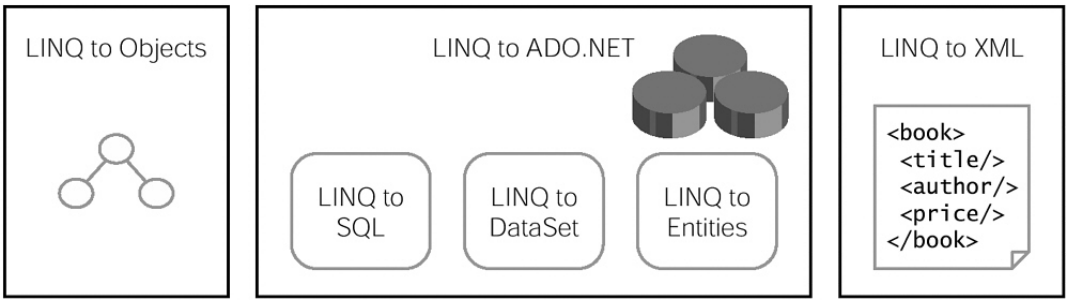
Klienti var būt DataTable no DataSet:

```
DataSet ds = GetDataSet();
DataTable Customers = ds.Tables["Customers"];
```

Klienti var būt klase, kas apraksta fizisko tabulu relāciju datu bāzē:

```
DataContext db = new DataContext(ConnectionString);
Table<Customer> Customers = db.GetTable<Customer>();
```

LINQ ir tehnoloģija, kas var strādāt ar dažādiem datu domēniem. Microsoft .NET 3.5 piedāvā sekojošas implementācijas (Attēls 1.3.1.).



1.3.1. att. **Microsoft .NET 3.5 piedāvātās LINQ implementācijas**

LINQ to Objects

LINQ to Objects mērķis ir strādāt ar kolekcijām no objektiem, kas savā starpā ir saistīti un veido hierarhiju vai grafu. Linq to Objects var tikt izmantots ne tikai lai apstrādātu programmētāja veidotus sarakstus, piemēram:

```
string tempPath = Path.GetTempPath();
DirectoryInfo dirInfo = new DirectoryInfo(tempPath);
var query =
from f in dirInfo.GetFiles()
where f.Length > 10000
orderby f.Length descending
select f;
```

LINQ to ADO.NET

LINQ to ADO.NET iekļauj sevī vairākas LINQ realizācijas, kas var apstrādāt relacionālus datus:

- LINQ to SQL – uztur saiti starp C# datu tipiem un fiziskajām tabulām
- LINQ to Entities – ir ļoti līdzīgs LINQ to SQL. Fizisko tabulu vietā izmanto konceptuālo entītiju datu modeli EDM (Entity Data Model). Rezultātā tiek izveidots abstrakcijas līmenis, kas ir neatkarīgs no fiziskā datu līmeņa.
- LINQ to DataSet – Strādā ar DataSet un ļauj veikt pieprasījumus pret tiem.

Piemērs:

```
DataContext db = new DataContext(ConnectionString);
Table<Customer> Customers = db.GetTable<Customer>();

var query =
from c in Customers
where c.Country == "USA"
&& c.State == "WA"
select new { c.CustomerID, c.CompanyName, c.City };

foreach (var row in query)
{
```

```
Console.WriteLine(row);
}
```

LINQ to XML

LINQ to XML piedāvā nedaudz savādāku sintaksi kas strādā ar XML datiem. LINQ to XML piedāvā DOM un XPath un XQuery apvienojumu.

```
XDocument customer =
    new XDocument(
        new XDeclaration("1.0", "UTF-16", "yes"),
        new XElement("customer",
            new XAttribute("id", "C01"),
            new XElement("firstName", "Paolo"),
            new XElement("lastName", "Pialorsi"),
            new XElement("addresses",
                new XElement("address",
                    new XAttribute("type", "email"),
                        "paolo@devleap.it"),
                new XElement("address",
                    new XAttribute("type", "url"),
                        "http://www.devleap.it/"),
                new XElement("address",
                    new XAttribute("type", "home"),
                        "Brescia - Italy"))));

foreach (XElement a in customer.Descendants("addresses").Elements())
{
    Console.WriteLine(a);
}
```

1.4. Arhitektūras izmaiņas

[1] Viens no lielu aplikāciju pamatprincipiem lai apstrādātu resursu konkurenci ir transakcijas. [1] Transakcija ir konkrētu darbību secība ar strikti definētu sākuma un beigu stāvokļiem, kā arī gan sākuma, gan beigu stāvokļos visi iesaistītie resursi ir strādājošā stāvoklī. Programmatūras izstrādē transakcijām piemīt sekojošas īpašības – ACID (angl. Atomicity, Consistency, Isolation, Durability):

- Atomicity – Visi soļi kas tiek veikti transakcijas ietvaros ir jāveic pilnībā un veiksmīgi. Pretējā gadījumā visa transakcija ir jāpārtrauc un visi resursi jāatgriež to sākuma stāvoklī, kāds tas bija uzsākot transakciju.
- Consistency – sistēmas resursiem jābūt strādājošā stāvoklī gan pirms gan pēc transakcijas.
- Isolation – kāda transakcijas soļa rezultāts nedrīkst būt redzams ārpus transakcijas, tikmēr, kamēr visa transakcija nav veiksmīgi pabeigta.
- Durability – veiksmīgas transakcijas rezultātam ir jābūt drošam – jāspēj izturēt jebkāda veida traucējumi.

Transakcijas attiecas ne tikai uz datu bāzēm, bet arī, piemēram, ziņojumu rindas, izdrukas, utt. jebkuri resursi, kas ir transakcionāli – izmanto transakcijas, lai apstrādātu konkurenci.

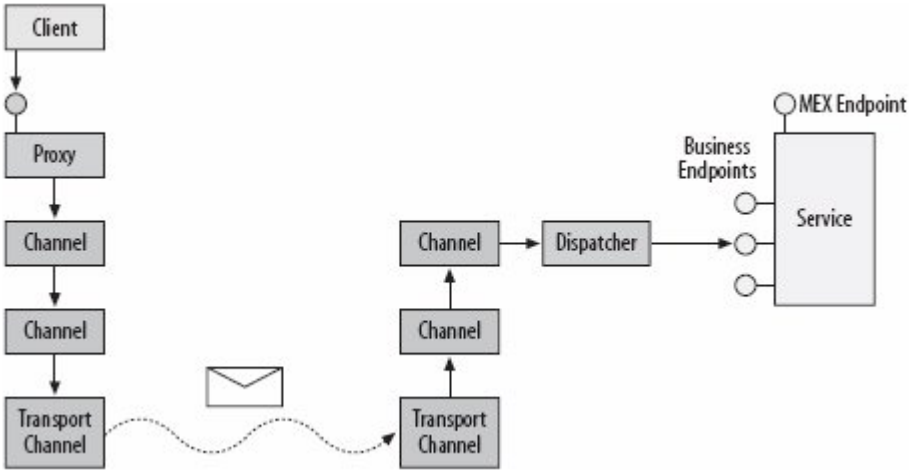
Viens no veidiem kā uzlabot ātrdarbību ir sniegt lietotājam atbildi par procesa rezultātu pēc iespējas ātrāk, pat ja process vēl nav beidzies. Vienlaicīgi ar šo atbildi pabeigt procesu. Tas rada lietotājam sajūtu, ka sistēma strādā ātrāk. Šī metode ir īpaši aktuāla Interneta aplikācijām. Šo metodi var realizēt arī sadalot kādu no procesiem divos vai vairākos paralēlos procesos. [2]

[6] Lai nodrošinātu lielāku datu apstrādi un ātrdarbību, sistēmas nepieciešams izstrādāt ar pēc iespējas īsākām transakcijām. Tas ir nepieciešams, jo transakcijas laikā pieeja pie iesaistītajiem resursiem ir ierobežota. [6] Kā viena no iespējām ir sadalīt esošo transakciju loģiskās biznesa procesa daļās un apstrādāt katru daļu atsevišķi. Bet, lai uzlabotu kopējo sistēmas ātrdarbību un mērogojamību, kādu no šīm biznesa procesa daļām var apstrādāt asinhroni un uz atsevišķa servera. [9] Lai realizētu šādu risinājumu Microsoft piedāvā WCF (Windows Communication Foundation) attālinātu procesu izsaukšanai un MSMQ (Microsoft Message Queue).

WCF

WCF nodrošina ziņojumu apmaiņas tehnoloģiju starp dažādiem aplikācijas servisiem. Protams, servisus var izstrādāt arī bez WCF, bet WCF piedāvā daudz ērtāku un universālāku veidu kā to izdarīt. WCF piedāvā vairākus ērtus risinājumus, piemēram, servisu darbināšanu, servisu administrēšanu, asinhronus izsaukumus, transakcijas, izsaukumus ar rindām, drošību. Tāpat WCF piedāvā paplašināt esošo funkcionalitāti, piemēram, pievienojot paštaisītus ziņojumu transporta kanālus. [5] [4]

Izmantojot WCF Klients nekad nesazinās ar servisu pa tiešo, pat ja tas notiek uz lokālā datora. Klients vienmēr izmanto proxy, lai nodotu izsaukumu servisam. Proxy piedāvā tās pašas metodes, kas ir izsaucamajam servisam, kā arī dažas vadības metodes (Attēls 1.4.1.).



1.4.1. att. **WCF Arhitektūra**

Salīdzinājumā ar DCOM vai .NET Remoting, WCF piedāvā identisku programmēšanas modeli servisiem neatkarīgi no tā vai serviss atrodas uz tā paša datora tanī pašā procesā, vai uz pilnīgi cita datora. Aplikāciju ar šādā veidā izstrādātiem servisiem nākotnē ir daudz ērtāk mērogot pārvietojot servisus starp dažādām atrašanās vietām.

WCF var izmantot dažāda veida transporta protokolus, tai skaitā paštaisītos. Microsoft piedāvātais WCF risinājums piedāvā sekojošus transporta protokolus:

- HTTP
- TCP
- Peer networking
- IPC
- MSMQ [12]

WCF servisi uztur līgumus (contract). Līgums ir no platformas neatkarīgs standartizēts veids kā aprakstīt servisa darbību. WCF piedāvā 4 veidu līgumus:

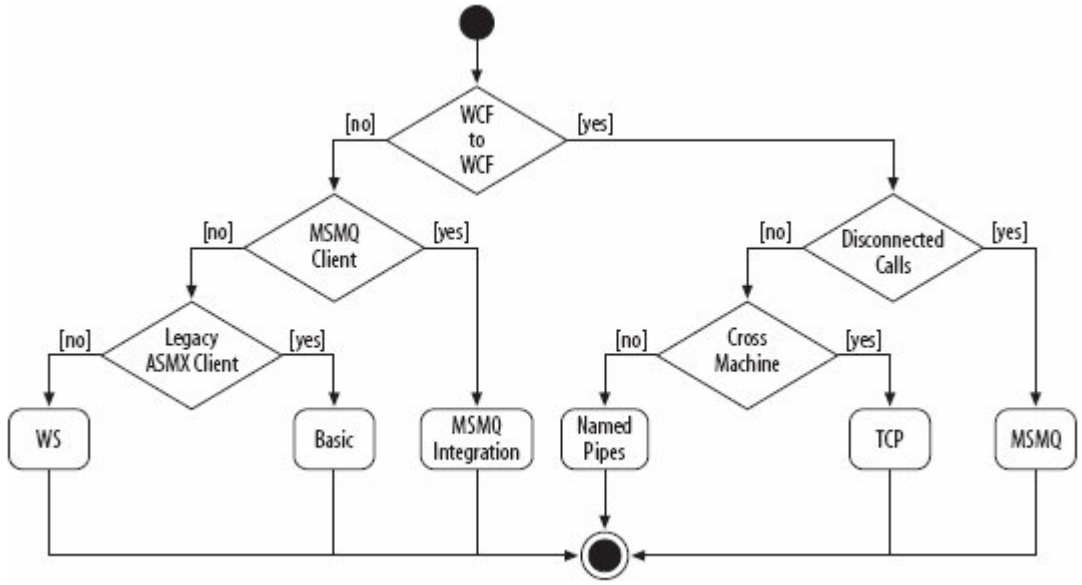
- Servisa līgums – apraksta kādas operācijas klients var veikt ar servisu.
- Datu līgums – apraksta kādu datu tipi tiek nodoti servisam un kādi no tā saņemti. WCF jau standartā nodrošina pamata datu tipus, piemētam, INT un STRING. Bet var tikt definēti arī paštaisītie tipi.
- Kļūdu līgums – apraksta kādas kļūdas serviss var izsaukt un kā tās tiks nodotas klientam.
- Ziņojumu līgums – ļauj servisam sadarboties ar ziņojumiem. Tas ir nepieciešams gadījumos, kad, piemēram, servisam ir viena veida parametri, bet klients nodod cita veida datus.

WCF serviss nevar eksistēt pats no sevis. Katrs WCF serviss ir jāuztur kādā no Windows procesiem – host process. Viens host process var uzturēt vairākus servisus, kā arī viens serviss var tikt uzturēts vairākos dažādos host procesos. Host process var arī būt tas pats klienta process no kura tiek izsaukts serviss. Kā host process var būt gan IIS (Internet Information Services) serveris, gan jebkurš cits process pēc izstrādātāja izvēles.

Kā viens no variantiem host procesam ir IIS. Galvenā tā priekšrocība ir tā, ka host process tiek startēts automātiski pie pirmā klienta pieprasījuma, kā arī par procesa dzīves ciklu var pilnībā paļauties uz IIS. Kā trūkumu IIS var uzskatīt, ka šajā gadījumā var izmantot tikai HTTP transportu. Izmantojot IIS kā host procesu servisa uzturēšana tajā līdzinās parastā Web Servisa uzturēšanai. Ir nepieciešams izveidot virtuālo direktoriju un izveidot .svc failu, kas pēc satura ļoti līdzinās .asmx failam.

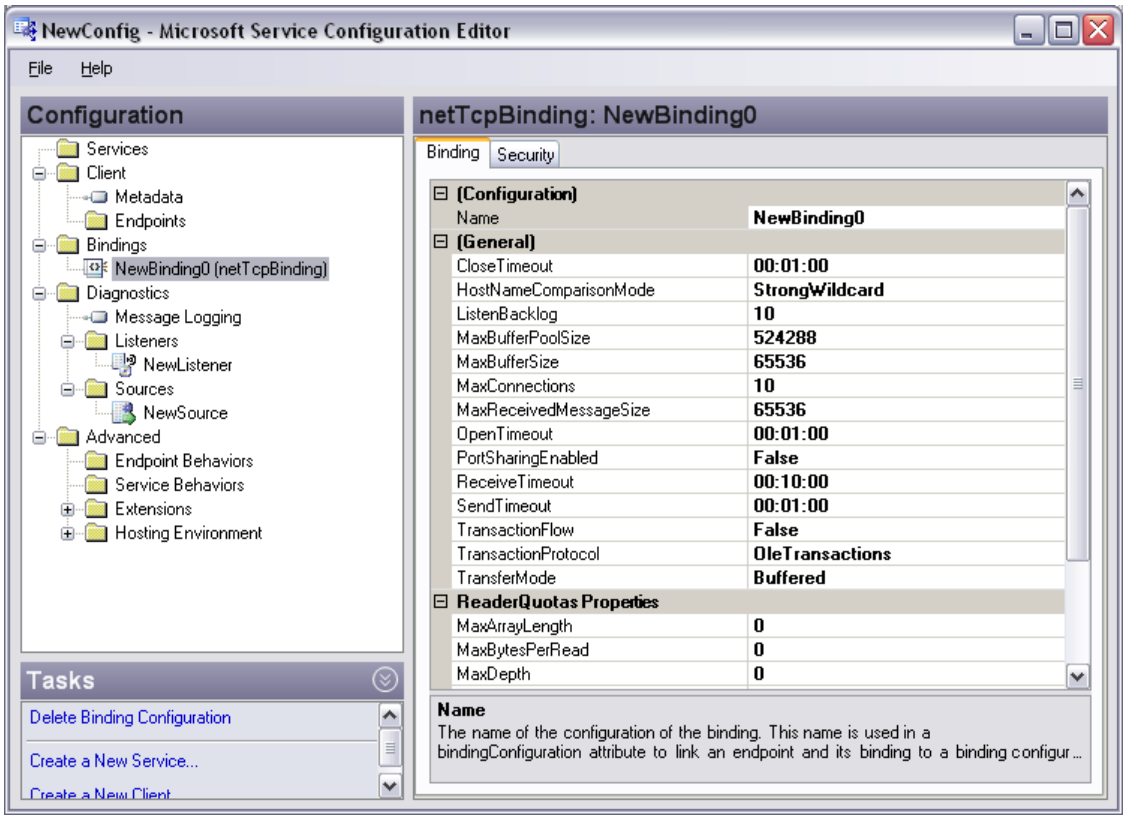
Windows Vista piedāvā WAS (Windows Activation Service) servisu, kurā arī var uzturēt servisus. Servisa uzturēšana iekš WAS ir līdzīga uzturēšanai iekš IIS, ar izņēmumu, ka nav ierobežojuma uz transportu – HTTP.

Lai izvēlētos atbilstošo transportu un saiti starp servisu un klientu Microsoft piedāvā sekojošu lēmuma pieņemšanas shēmu (Attēls 1.4.2.).



1.4.2. att. **Microsoft piedāvātā WCF transporta un saites tipa izvēles shēma**

WCF klienta programmēšanai var izmantot Visual Studio piedāvāto proxy ģenerēšanas iespēju no servisa meta datiem. Servisa konfigurācija tiek norādīta klienta konfigurācijas failā. Klientu var konfigurēt gan no XML konfigurācijas faila, gan pašā programmas kodā. Tāpat WCF piedāvā ērtu konfigurācijas faila mainīšanas programmu (Qttēls 1.4.3.).

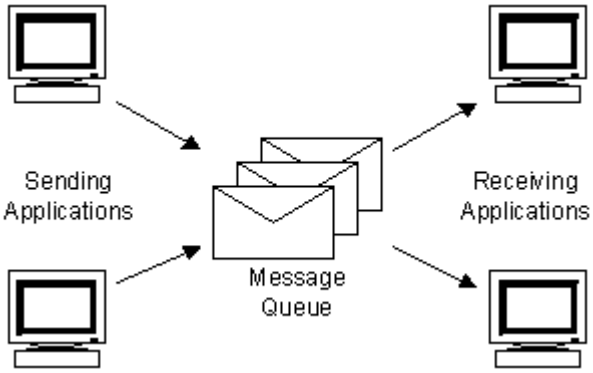


1.4.3. att. **Microsoft piedāvātā WCF klienta un servisa konfigurācijas mainīšanas programma**

Atsevišķos savienošanās veidos WCF nodrošina drošus sakarus, garantējot to, ka servisu izsaukumi nonāks līdz servisiem un tādā kārtībā, kā tie tika izsaukti. Šis izsaukumu drošības nodrošinājums balstās uz industrijas standartiem par drošu ziņojumu pārsūtīšanu. Tādejādi WCF spēj nodrošināt gan atkārtotus izsaukumu mēģinājumus, ja, piemēram, uz laiku pārtrūkst bezvadu savienojums, gan ziņojumu uzturēšanu buferī, gan paša savienojuma izveidošanu, pārbaudi un aizvēršanu, kad tas vairs nav nepieciešams.

MSMQ

MSMQ ir ziņojumu apmaiņas protokols, kas ļauj aplikācijām, kas atrodas uz attālinātiem serveriem sazināties drošā veidā. MSMQ galvenā ideja ir visu sūtījumu saglabāšana rindā, un izņemšana no tās, kad ziņojums tiek veiksmīgi nosūtīts saņēmējam, tādā veidā garantējot, ka jebkuros apstākļos neviens ziņojums nepazudīs, ka arī ziņojumi var tikt izsūtīti saņēmējam, kamēr tas nav pieejams (Attēls 1.4.5.). Turpretī citi tīkla protokoli spēj strādāt tikai ja starp sūtītāju un saņēmēju eksistē strādājošs savienojums. [10]



1.4.5. att. **MSMQ rindas attēlojums starp sūtītājiem un saņēmējiem**

Aplikācijas, kas izmanto MSMQ ir iedalāmas divās kategorijās – ziņojumu izsūtīšanas un ziņojumu saņemšanas aplikācijas. Izsūtīšanas aplikācija sūta ziņojumus nevis saņemšanas aplikācijai, bet uz MSMQ rindu. Tas ļauj abām aplikācijām strādāt neatkarīgie vienai no otras. Izstrādājot izsūtīšanas mehānismu ir jāņem vērā vairāki apstākļi:

- Transakcionāli / netransakcionāli ziņojumi – transakcionālie ziņojumi garantē to, ka ziņojums tiks nogādāts tieši vienu reizi un nepieciešamajā secībā, turpretim netransakcionālie ziņojumi to nenodrošina.
- Ātrais sūtījums / drošais sūtījums – ātrā sūtījuma gadījumā sūtījums tiek saglabāts datora atmiņā un pārsūtot uz nākamo serveri tiek dzēsts. Drošā sūtījuma gadījumā sūtījums tiek saglabāts uz visu serveru diskiem tik ilgi, kamēr tas netiek veiksmīgi nodots saņēmējam.
- Tiešais sūtījums / pārsūtīšana – Tiešā sūtījuma gadījumā sūtījums tiek uzreiz nosūtīts uz nepieciešamo serveri. Pārsūtīšanas gadījumā var tikt nokonfigurēts ceļš, caur kādiem serveriem ir jāsūta dotais ziņojums.
- Viens / vairāki saņēmēji – ir iespējams norādīt vai ziņojums jānosūta tikai vienam vai vairākiem saņēmējiem.
- Ziņojums ar / bez autentifikācijas – autentificētus ziņojumus MSMQ nogādā saņēmējam ar garantiju, ka neviens doto ziņojumu tā sūtīšanas brīdī nav izmainījis.
- Nokodēts / nenokodēts ziņojums – ziņojumus ir iespējams nokodēt, lai neviens cits tos nespētu pārķert un atkodēt.
- Ar / bez tiešsaistes operācijām – sūtot ar tiešsaistes pieeju, sūtīšanas aplikācijai ir pieeja pie servisa, kas kontrolē ziņojumu pārsūtīšanu.

Izstrādājot saņemšanas mehānismu ir jāņem vērā šādi apstākļi:

- Tieša / attālināta ziņojumu lasīšana – saņēmēja aplikācijai ir iespēja lasīt ziņojumus gan no tā paša servera, gan attālināta servera.

- Transakcionāli / netransakcionāli ziņojumi – darbība ir analoģiska ar izsūtīšanas aplikāciju.
- Sinhronā / asinhronā ziņojumu lasīšana – asinhroni lasot ziņojumus, aplikācija var turpināt darbu līdz kamēr ziņojumi tiks nolasīti.
- Apskate / saņemšana – saņemšanas aplikācijai ir iespējams saņemt ziņojumu vai tikai pārbaudīt vai ir pienācis atbilstošs ziņojums.
- Ar / bez tiešsaistes operācijām – darbība ir analoģiska ar izsūtīšanas aplikāciju.

Rindas ir loģiski konteineri ziņojumiem, kuros MSMQ tos saglabā un vēlāk pārsūta saņēmējam, tādā veidā dodot iespēju apmainīties ar ziņojumiem reti savienotām aplikācijām. Tādejādi ziņojumu izsūtošā aplikācija nemaz nezin, kad ziņojums tiks nosūtīts vai kad saņēmējs to no rinda apstrādās. Sūtīšanas un saņemšanas procesi ir pilnībā neatkarīgi. Aplikācijas var veidot jaunas rindas, atrast esošas rindas, atvērt rindas, nosūtīt ziņojumu uz rindu, lasīt ziņojumu no rindas un mainīt rindas uzstādījumus.

Pavisam eksistē 5 veidu rindas:

Galamērķa rindas – tās ir rindas, kurās glabā ziņojumus ko izsūta aplikācija. Parasti galamērķa rindas glabājas uz servera, kas rindu apstrādās.

Administratīvās rindas izmanto sistēmas ģenerētiem ziņojumiem par veiksmīgu vai neveiksmīgu aplikācijas ziņojuma nosūtīšanu.

Atbildes rindas – ir rinda, kurā izsūtītā ziņojuma saņēmējs ieliek atbildes ziņojumu.

Atskaišu rindas – šajā rindā tiek ielikti ziņojumi par to kā aplikācijas ziņojumi tika nosūtīti uz rindu saņēmējam.

Apakšrindas – ir kādas citas fiziskas rindas loģiska daļa, kurā tiek glabāti ziņojumi, kas atbilst kādiem īpašiem nosacījumiem.

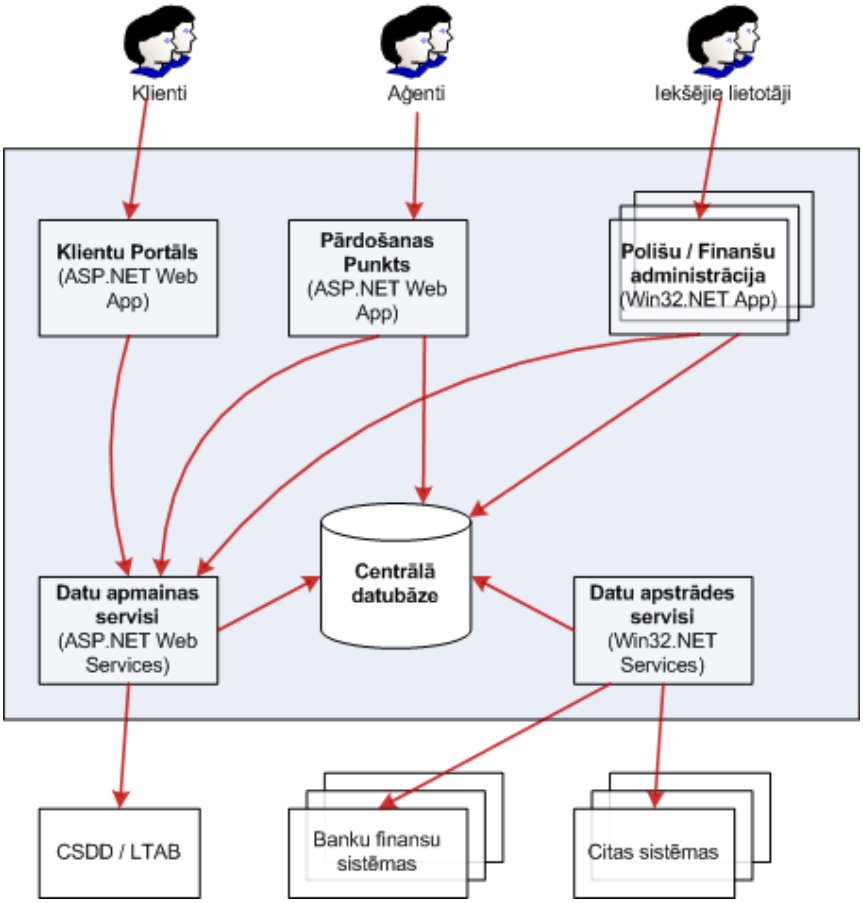
Ziņojumi ir informācijas kopums, kas tiek pārsūtīts ar MSMQ palīdzību. Katrs ziņojums sastāv no vairākiem parametriem un satura. Parametrus izmanto MSMQ lai pārsūtītu ziņojumus, bet saturā ir informācija, ko sūtītājs nodod saņēmējam. Maksimālais ziņojuma izmērs ir 4MB. Šis ierobežojums ir saistīts ar to, ka visi ziņojumi ir rindās, kas ir fiziski faili uz servera.

2. REĀLAS SISTĒMAS APSKATS

2.1. Sistēmas arhitektūra

Ka piemērs tika izvēlēta viena apdrošināšanas / finansu sistēma, kurai jau iepriekš tikai veikti veiksmīgi ātrdarbības uzlabojumi, bet laika gaitā radās nepieciešamība pēc papildus ātrdarbības uzlabojumiem.

Sistēma sastāv no vairākiem moduļiem, kas visi izmanto vienu centrālo datu bāzi (Attēls 2.1.1.). Visi procesi, izņemot datu apstrādes / apmaiņas servisus ir asinhroni. Piemēram Polises reģistrācija notiek kopā ar finansu datu reģistrāciju.



2.1.1. att. **Sistēmas uzbūve**

Centrālā datubāze

Centrālā datubāze ir viena no galvenajām sistēmas sastāvdaļām. Svarīgākie dati, kas tiek uzglabāti datubāzē ir:

- Dati par klientiem;
- Polišu dati;
- Atlīdzības;
- Finansu un grāmatvedības dati.

Polišu / Finanšu administrācija

- Izdod polises;
- Administrē atlīdzības;
- Uztur klientu informāciju;
- Administrē finansu un grāmatvedības uzskaiti.

Klientu portāls

- Klienti iegādājas polises.

Pārdošanas Punkts

- Partneri reģistrē polises;
- Partneri iegūst atskaites.

Datu apmaiņas servisi

- Piekļuve CSDD un LTAB;
- Atbalsta funkcijas Klientu portālam.

Datu apstrādes servisi

- Sadarbība ar ārējām sistēmām;
- Iekšējo procesu uzturēšana.

2.2. Sistēmas ātrdarbība

Sistēmas un datubāzes arhitektūra

Sistēmas arhitektūra ir viens no galvenajiem sistēmas ātrdarbības faktoriem. Arhitekturālie risinājumi ietekmē sistēmas ātrdarbību pat vairāk nekā reālā aplikācijas implementācija.

Sistēma tika būvēta 6 gadu garumā. Sākotnēji izstrādājot sistēmu tā tika plānota kā salīdzinoši neliela aplikācija. Laika gaitā tā pārauga par lielu daudzlietotāju sistēmu, bet sistēmas arhitektūra palika gandrīz nemainīga. Sistēmā tika veikti Microsoft SQL servera ātrdarbības uzlabošanas darbi, kas deva jūtamu rezultātu. Izmantojot sekojošus paņēmienus tika iegūts nepieciešamais sistēmas ātrdarbības uzlabojums:

- SQL Server Profiler rezultātu analīze – biežāk izsauktie pieprasījumi / pieprasījumu skaits vienā biznesa procesā
- Execution Plan / Client Statistics analīze – SQL pieprasījumu uzbūves analīze
- Indeksu izmantošanas veidu pārbaude

- Neizmantoto / trūkstošie indeksu pievien ošana / dzēšana
- Indeksu fragmentācijas / klāsterēto indeksu lietojuma pārbaude

Datu bāzē tika izņemti neizmantotie indeksi, salikti trūkstošie indeksi / klasterētie indeksi, nokonfigurēta indeksu fragmentācija un optimizēti SQL pieprasījumi.

Bet sistēmai tālāk attīstoties un patstāvīgi augot lietotāju skaitam, tā nespēj nodrošināt pietiekošu ātrdarbību. Biežākās sūdzības no lietotājiem ir:

- Izdodot polisi jāgaida ļoti ilgi
- Sistēma paziņo par neparedzētu kļūdu un aizveras
- Sistēmā jāspiež poga atkārtoti lai izdarītu nepieciešamo darbību, jo iepriekšēja reizē bija kļūda.

Lai atrastu sistēmas ātrdarbības problēmas, tika gan izmantots SQL Server Profiler, lai analizētu datubāzes pieprasījumus, gan sistēmā iestrādātais notikumu žurnāls, gan programmas koda caurskatīšana.

Viena no vislielākajām problēmām, kas tika konstatēta bija strupsaķerju skaits sistēmā. Vidēji viena dienā notiek 21 strupsaķeres. Caurskatot programmas kodu, tika konstatēts, ka strupsaķeres gadījums sistēmā netiek apstrādāts.

Apskatot ilgākās SQL transakcijas, tika konstatēts, ka datubāzes objektu bloķējumu dēļ atsevišķas transakcijas ilgst līdz pat 10 sekundēm, kas absolūti nav pieņemams ne no sistēmas uzbūves viedokļa, ne no sistēmas lietotāja skata punkta.

Izpētot sistēmas arhitektūru tika konstatēts, ka atsevišķi sarežģītu aprēķinu algoritmi (finansu aprēķini) tiek izpildīti uz klienta datora, ka rezultātā datora resursu nepietiekamības dēļ tie notiek salīdzinoši lēnu.

Tāpat sistēmas arhitektūra nepiedāvā nekādu slodzes sadalīšanas risinājumu un WEB aplikācijas gadījumā pilnīgi viss notiek uz 1 WEB servera.

Pārskatot SQL Profiler žurnālu tika konstatēts, ka atsevišķi identiski pieprasījumi tiek izpildīti vairākas reizes (piemēram, klienta datu ielāde).

Tāpat tika konstatēts, ka lietotājam uzsākot biznesa procesu ir ilgi jāgaida, jo notiek visu šim procesam iespējami nepieciešamo klasifikatoru un datu ielāde, pat ja lietotājam tie tobrīd nav nepieciešami.

Ātrdarbības problēmas

Apkopojot izanalizētās sistēmas rezultātu tika secināts, ka sistēmai ir nepieciešami arhitektūras uzlabojumi un tika sastādīts šāds nepieciešamo uzlabojumu saraksts:

- Garo SQL transakciju sadalīšana

- Atsevišķu sistēmas sadaļu (finansu) izdalīšana un novietošana uz aplikāciju servera
- Datu ielāde izmantojot Lazy-Load (ielāde pēc nepieciešamības)
- Izvairīšanās no atkārtotas vienu un to pašu datu vairākkārtējas ielādes
- Izdalīt sistēmas procesus ko var izpildīt asinhroni

3. PIELIETOTĀS METODES

3.1. Nepieciešamie uzlabojumi

Balstoties uz konstatētajām problēmām un nepieciešamajiem risinājumiem tika izvelēts izpētīt un pārbaudīt Microsoft LINQ tehnoloģiju, Windows Communication Foundation un MSMQ tehnoloģijas lai risinātu šīs sistēmas problēmas.

3.2. Jaunāko tehnoloģiju pārbaude

Tika veikti vairāki dažāda veida eksperimenti, lai iepazītos un salīdzinātu LINQ ātrdarbību ar jau esošajām tehnoloģijām.

LINQ to Objects (LINQ cikls)

Tika izstrādāta C# programma, lai novērtētu LINQ to Objects ātrdarbību. Tika izvēlēts vienkāršākais gadījums – datu meklēšana dažāda izmēra nesakārtotā masīvā. Tika salīdzinātas FOR, FOREACH un LINQ piedāvāto iespēju konstrukcijas.

Mērījumiem tika izmantotas šādas konstrukcijas:

LINQ konstrukcijai

```
int oddNumbers = array.Count(n => n % 2 == 1);
return oddNumbers;
```

FOREACH konstrukcijai

```
int counter = 0;
foreach (int n in array)
{
      if (n % 2 == 1)
      {
            counter++;
      }
}
return counter;
```

FOR konstrukcijai

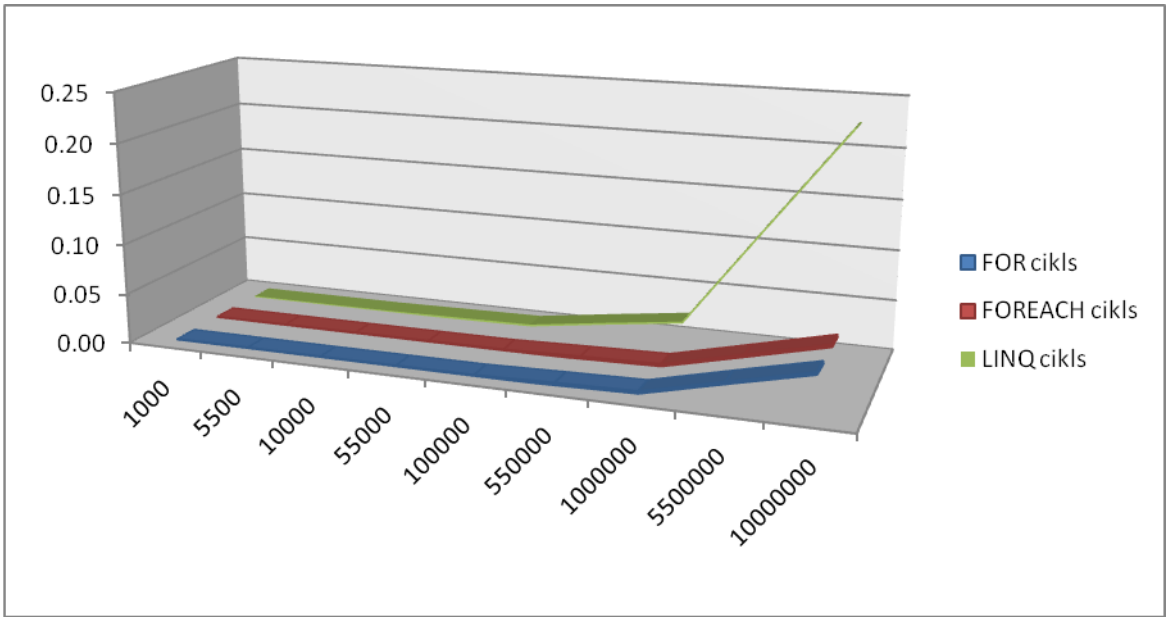
```
int counter = 0;
for (int n = 0; n < array.Length; n++)
{
      if (array[n] % 2 == 1)
      {
            counter++;
      }
}
return counter;
```

Katrs mēģinājums tika veikts 100 reizes lai iegūtu precīzāku rezultātu. Rezultāts attēlots tabulas (Tabula 3.2.1.) un grafika (Attēls 3.2.1.) veidā.

3.2.1. Tabula

Mērījumu dati LINQ to Objects ātrdarbības novērtēšanai

Masīva elementu skaits	*FOR cikls (sek.)*	*FOREACH cikls (sek.)*	*LINQ cikls (sek.)*
1000	*0.0000067327*	*0.0000064170*	*0.0000489140*
5500	*0.0000216257*	*0.0000211340*	*0.0001249684*
10000	*0.0000392201*	*0.0000455589*	*0.0002000813*
55000	*0.0002073141*	*0.0002180389*	*0.0012004598*
100000	*0.0004050571*	*0.0003853674*	*0.0021453676*
550000	*0.0023320925*	*0.0023709131*	*0.0125735952*
1000000	*0.0040519396*	*0.0040776300*	*0.0224127759*
5500000	*0.0227801718*	*0.0225631274*	*0.1250334386*
10000000	*0.0409896090*	*0.0411158010*	*0.2263997324*



3.2.1. att. **Mērījumu līknes ar masīva apstrādei nepieciešamo laiku (sek.) atkarībā no masīva elementu skaita**

No tabulas var redzēt, ka LINQ šādā situācijā strādā aptuveni 5 reizes lēnāk nekā FOR vai FOREACH konstrukcijas.

LINQ to SQL (LINQ datu ielāde)

Tika izstrādāta C# programma, lai novērtētu LINQ to SQL ātrdarbību. Tika izvēlēts vienkāršākais gadījums – datu atlase no datubāzes bez papildus nosacījumiem. Tika salīdzinātas ADO.NET, ADO.NET ar DataSet un LINQ piedāvāto iespēju konstrukcijas.

Mērījumiem tika izmantotas šādas konstrukcijas:

LINQ konstrukcijai

```
using (DataClasses1DataContext dc = new DataClasses1DataContext())
{
    var contactNames = from c in dc.Customers
                       select new { c.CompanyName, c.ContactName };
    foreach (var contactName in contactNames)
    {
        string fullName = contactName.CompanyName + contactName.ContactName;
    }
}
```

ADO.NET konstrukcijai

```
string cnctString = ConsoleApplication1.Properties.Settings.Default.MyConnection;
using (SqlConnection cnct = new SqlConnection(cnctString))
{
    SqlCommand cmd = new SqlCommand("Select CompanyName, ContactName From
dbo.Customers",  cnct);
    cnct.Open();
    using(SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection))
    {
        while (rdr.Read())
        {
            string fullName = rdr.GetString(0) + rdr.GetString(1);
        }
        rdr.Close();
    }
    cnct.Close();
}
```

ADO.NET ar DataSet konstrukcijai

```
string cnctString = ConsoleApplication1.Properties.Settings.Default.MyConnection;
using (SqlConnection cnct = new SqlConnection(cnctString))
{
    SqlCommand cmd = new SqlCommand("Select CompanyName, ContactName From
dbo.Customers", cnct);
    cnct.Open();
    DataSet ds = new DataSet();
    SqlDataAdapter adp = new SqlDataAdapter(cmd);
    adp.Fill(ds);
    foreach (DataRow dr in ds.Tables[0].Rows)
    {
        string fullName = dr[0].ToString() + dr[1].ToString();
    }
    cnct.Close();
}
```

Kopa tika veiktas 9 iterācijas un katrā iterācijā 100 reizes izpildīts minētais programmas kods, lai iegūtu precīzāku rezultātu. Rezultāti ir attēloti tabulas veidā (Tabula 3.2.2.).

3.2.2. Tabula

Mērījumu dati LINQ to SQL ātrdarbības novērtēšanai

Iterācija	*ADO.NET datu ielāde (sek.)*	*ADO.NET DataSet datu ielāde (sek.)*	*LINQ datu ielāde (sek.)*
1	*0.000928386*	*0.001296363*	*0.003229137*
2	*0.000999110*	*0.001196211*	*0.003621882*
3	*0.000908669*	*0.001177558*	*0.002910216*
4	*0.000908711*	*0.001272955*	*0.002797411*
5	*0.000977194*	*0.001134734*	*0.002904998*
6	*0.000982538*	*0.001145034*	*0.002835212*
7	*0.000914189*	*0.001178429*	*0.002809072*
8	*0.000909822*	*0.001162818*	*0.003057945*
9	*0.001539891*	*0.001124878*	*0.002944011*

No tabulas var redzēt, ka LINQ šādā situācijā strādā aptuveni 2 reizes lēnāk nekā ADO.NET konstrukcijas. Bet LINQ datu atlases programmas kods ir daudz īsāks un pārskatāmāks. Rezultātā var secināt, ka atsevišķos gadījumos LINQ to SQL var izmantot, jo tas palīdz programmas kodam būt pārskatāmākam.

LINQ to XML (LINQ izmantošana XML ielādē)

Tika izstrādāta C# programma, lai novērtētu LINQ to XML ātrdarbību. Tika izvēlēts vienkāršākais gadījums – datu atlase no XML faila. Tika salīdzinātas XPath un LINQ piedāvāto iespēju konstrukcijas. [8]

Mērījumiem tika izmantotas šādas konstrukcijas:

LINQ konstrukcijai

```
var elements = (from region in xDoc.Descendants(ns + "ShipRegion")
  join customer in xDoc.Descendants(ns + "Customer")
  on region.Ancestors(ns + "Order").Elements(ns + "CustomerID").First().Value
  equals customer.Attribute("CustomerID").Value
  where region.Value == "OR"
  select customer.Elements(ns + "CompanyName").First().Value).Distinct();
```

XPath konstrukcijai

```
string xPath = "//co:Customer[@CustomerID = //co:ShipRegion[. = '" +
  state + "']//ancestor::co:Order/co:CustomerID]/co:CompanyName";
IEnumerable<XElement> elements = xDoc.XPathSelectElements(xPath, nsmgr);
```

Kopa tika veiktas 9 iterācijas un katrā iterācijā 100 reizes izpildīts minētais programmas kods, lai iegūtu precīzāku rezultātu. Rezultāti ir attēloti tabulas veidā (Tabula 3.2.2.).

3.2.2. Tabula

Mērījumu dati LINQ to XML ātrdarbības novērtēšanai

Iterācija	*XPath (sek./100)*	*LINQ XML apstrāde (sek./100)*
1	*1.5422176*	*0.2203168*
2	*1.6223328*	*0.2103024*
3	*1.5522320*	*0.2303312*
4	*1.5522320*	*0.2203168*
5	*1.5522320*	*0.2403456*
6	*1.5622464*	*0.2303312*
7	*1.5522320*	*0.2203168*
8	*1.5322032*	*0.2203168*
9	*1.5622464*	*0.2303312*

No tabulas var redzēt, ka LINQ šādā situācijā strādā aptuveni 7 reizes ātrāk nekā XPath konstrukcijas. LINQ datu atlases programmas kods ir nedaudz garāks nekā XPath, bet toties daudz pārskatāmāks. Rezultātā var secināt, ka LINQ to XML var izmantot, jo tas gan uzlabo ātrdarbību, gan palīdz programmas kodam būt pārskatāmākam.

LINQ lietošanas ērtums

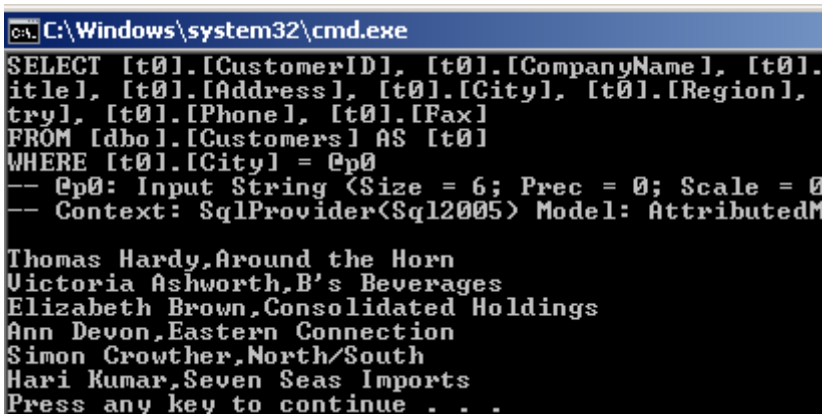
Veidojot LINQ konstrukcijas tika konstatēts, ka tās ir daudz ērtāk un saprotamāk izmantot, lai realizētu to pašu funkcionalitāti, ko bez LINQ.

Ar LINQ var ērtāk manipulēt ar datu ielādi un izvairīties no atkārtotas vienu un to pašu datu ielādes, kas aktīvas aplikācijas izmantošanas gadījumā ļauj ietaupīt resursus.

Izveidojam vienkāršu datu ielādes pieprasījumu:

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
      db.Log = Console.Out;
      var q = from c in db.Customers
            where c.City == "London"
            select c;
      foreach (Customer c in q)
      {
            Console.WriteLine("{0},{1}", c.ContactName, c.CompanyName);
      }
}
```

Izpildot šo programmas kodu iegūstam rezultātu, var redzēt, ka sākumā tiek veikts SQL pieprasījums uz datubāzi lai iegūtu nepieciešamos datus (Attēls 3.2.2.).

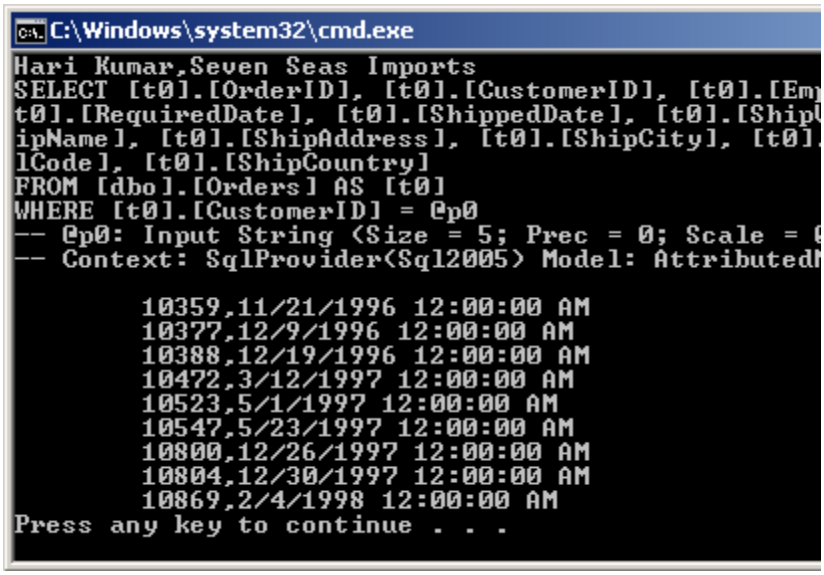


3.2.2. att. **Programmas izpildes rezultāts pēc vienkārša pieprasījuma**

Papildinām programmas kodu ar:

```
foreach (Order o in c.Orders)
{
    Console.WriteLine("\t{0},{1}", o.OrderID, o.OrderDate);
}
```

Izpildot programmas kodu, var redzēt, ka bez papildus koda rakstīšanas LINQ automātiski uzģenerē atbilstošu SQL pieprasījumu. Pieprasītie klienta pasūtījumu dati tiek ielādēti tikai tad, kad tie nepieciešami (Lazy-Load) (Attēls 3.2.3.).

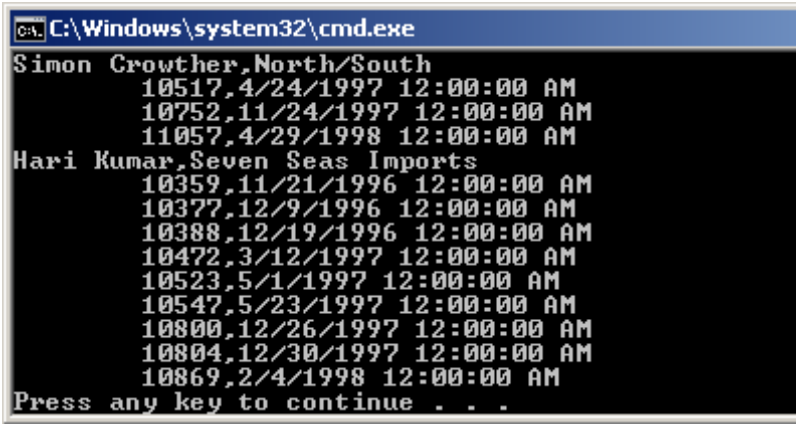


3.2.3. att. **Programmas izpildes rezultāts pēc lazy-load pieprasījuma**

Papildinām programmas kodu ar:

```
DataLoadOptions options = new DataLoadOptions();
options.LoadWith<Customer>(c => c.Orders);
db.LoadOptions = options;
```

Izpildot programmas kodu, var redzēt, ka tagad LINQ visus nepieciešamos klienta pasūtījuma datus ielādē uzreiz. Tas ir ērti gadījumos, ja ir zināms, ka visi pasūtījumu dati būs nepieciešami (Attēls 3.2.4.).



3.2.4. att. **Programmas izpildes rezultāts pēc saistītās datu ielādes**

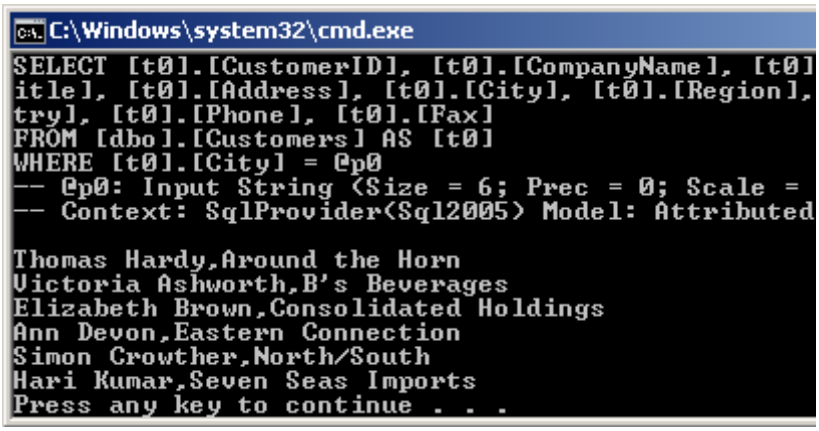
Papildinām programmas kodu ar:

```
db.DeferredLoadingEnabled = false;
```

un atkomentējam rindu:

```
//db.LoadOptions = options;
```

Izpildot programmas kodu var redzēt, ka klientu pasūtījumu dati netiek ielādēti ne sākumā, ne pēc pieprasījuma, ko ir ērti izmantot atsevišķās situācijās (Attēls 3.2.5.).



3.2.5. att. **Programmas izpildes rezultāts pēc aizliegtas saistīto datu ielādes**

Iepriekš apskatītais piemērs ir izmantojams apskatītajā apdrošināšanas sistēmā, lai nodrošinātu Lazy-Load. Kā arī citās situācijās, kad saistītie dati ir jāielādē uzreiz, vai arī to ielāde ir jāaizliedz.

3.3. Arhitektūras izmaiņas

WCF ātrdarbības pārbaude

Lai novērtētu WCF ātrdarbību tikai izstrādāta C# aplikācija, kurā tika salīdzināts WCF TCP savienojuma ātrums ar labi pazīstamajiem Web Servisiem. Gan WCF klients gan serviss

tika darbināts viena procesā, bet savienojuma veids starp viņiem veidots tāds, kas pieļauj tos novietot uz dažādiem datoriem. Rezultāts attēlots tabulas veidā (Tabula 3.3.1.).

3.3.1. Tabula

Mērījumu dati WCF ātrdarbības novērtēšanai

Masīva elementu skaits	*WS izsaukums (sek.)*	*WCF TCP izsaukums (sek.)*
1	*0.00581444*	*0.001526809*
5	*0.004853787*	*0.001171132*
10	*0.005156809*	*0.001261482*
55	*0.005797583*	*0.001640337*
100	*0.005796918*	*0.002550436*
550	*0.005692338*	*0.001432841*
1000	*0.005391962*	*0.001623055*
5500	*0.006659587*	*0.001981361*
8000	*0.006830863*	*0.001921767*

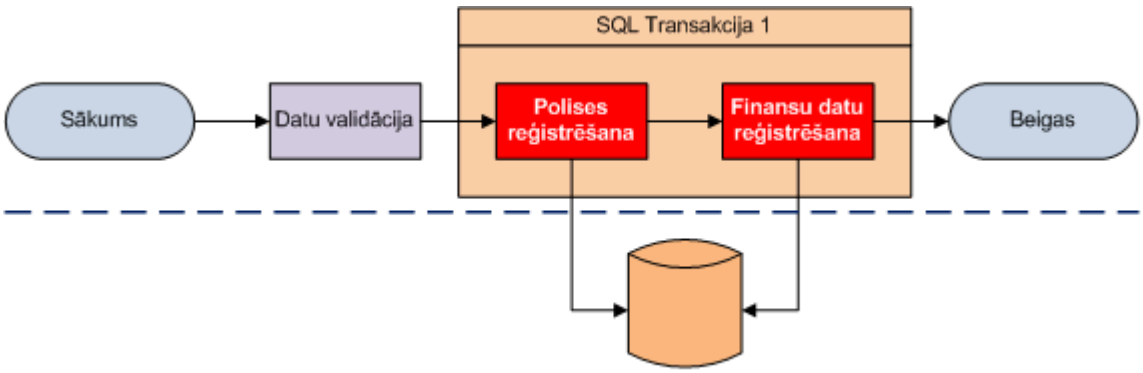
No rezultātiem var secināt, ka uz katru izsaukumu WCF patērē aptuveni 0.02 sekundes, kas salīdzinot ar WS izsaukumu ir jūtami ātrāk.

Esošās aplikācijas apskats

Izmantojot SQL Profiler, tika apzināti sistēmas apgabali, kas izmanto „garas" transakcijas. Viena no tām ir polises izdošanas transakcija. Transakcijā ir aptuveni 250 līdz 400 SQL pieprasījumi, atkarībā no izvēlētajiem polises parametriem. Izpētot sistēmas arhitektūru, polises izdošanas process sastāv no loģiskam daļām:

- Datu validācija – nav SQL pieprasījumu, jo visi dati ir jau ielādēti klasifikatoros
- Polises reģistrēšana sistēmā – aptuveni 50 – 200 SQL pieprasījumi
- Finansu datu reģistrēšana sistēmā – aptuveni 200 SQL pieprasījumi

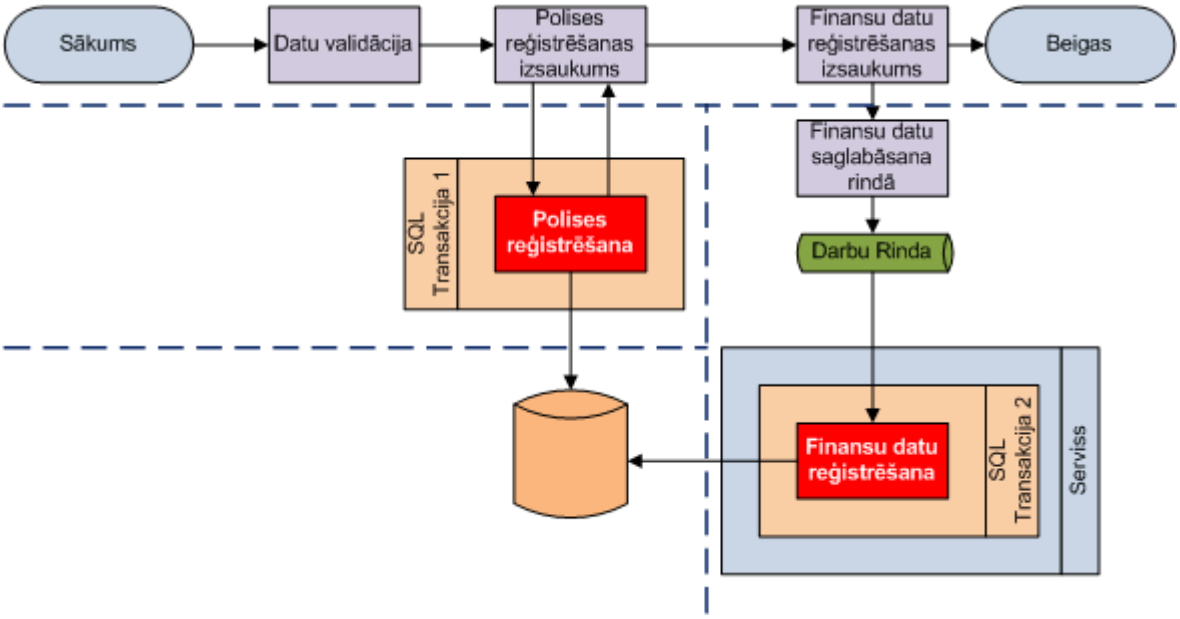
Šādu procesu var attēlot sekojošā diagrammā (Attēls 3.3.1.).



3.3.1. att. **Polises izdošanas process**

Apskatot šo procesu var secināt. Datu validācija notiek neatkarīgi no pārējā procesa un aizņem salīdzinoši maz resursu. Polises reģistrēšana un Finansu datu reģistrēšana atrodas vienā SQL transakcijā. Balstoties uz procesa shēmu, ir nepieciešams sadalīt šo transakciju divās daļās: Polises izdošana un Finansu datu reģistrēšana.

Tika konstatēts, ka finansu datu reģistrāciju ir iespējams veikt asinhroni. Tādejādi ir nepieciešama liela izmaiņa sistēmas arhitektūrā, kas ļauj veikt asinhronus procesus un noslodzi sadalīt starp serveriem. Finansu daļa tiek iznesta atsevišķā sadaļā, kas strādā asinhroni. Iespējamais izmainītais process attēlots sekojošā diagrammā (Attēls 3.3.2.).



3.3.2. att. **Uzlabotais polises izdošanas process**

Lai realizētu šādu arhitekturālu izmaiņu, vairākos sistēmas apgabalos ir nepieciešamas izmaiņas:

- Polises izdošanas funkcionalitātē
- Finansu moduļa funkcionalitātē

- Jauns serviss – finansu datu reģistrēšana
- DB izmaiņas – polišu / finansu arhitektūrā

Izpētot MSMQ iespējas, kā viena no negatīvajām pusēm tika konstatēts, ka MSMQ kā rindu var izmantot tikai uz lokālā servera saglabātu fizisku failu. Tas ir diezgan liels trūkums, jo salīdzinot ar Web servera sesijām, kuras var saglabāt gan SQL datubāzē, gan jebkurā citā nepieciešamā veidā, turpretim MSMQ spēj saglabāt tikai failu sistēmā. Tāpēc izvēlētajā arhitektūrā MSMQ rindas tika realizētas tādā veidā, lai tās varētu viegli nomainīt ar jebkādu paštaisītu risinājumu.

Lai pārliecinātos, ka izplānotais risinājums dos pietiekamu ātrdarbības uzlabojumu, tika izstrādāts atsevišķs aplikācijas piemērs lai pārbaudītu doto risinājumu.

Izstrādātās aplikācijas apraksts

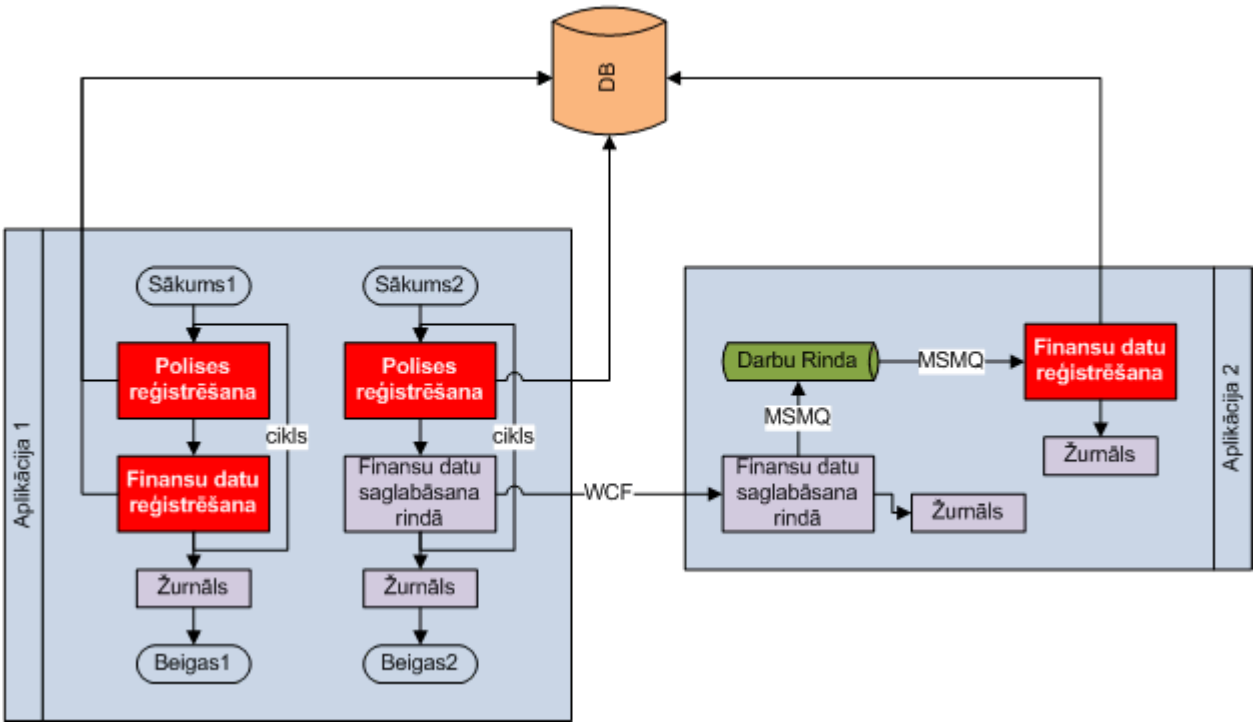
Lai pārbaudītu praktiski iepazītās tehnoloģijas un to iespējas tika izveidotas 2 aplikācijas. Pirmā aplikācija realizē Klienta aplikāciju gan esošajai gan plānotajai arhitektūrai. Otra aplikācija realizē papildus aplikāciju serveri, kurš uztur rindu un no tās apstrādā finansu datus. Viss risinājums tika darbināts uz 3 datoriem. Uz viena datora atrodas Pirmā aplikācija, uz otra datora Otrā aplikācija un uz trešā datora atrodas datubāze. Datoru konfigurācija attēlota tabulā (Tabula 3.3.2.).

3.3.2. Tabula

Mērījumos izmantoto datoru konfigurācijas

Komponente	***Datubāzes serveris***	***1. aplikācijas dators***	***2. aplikācijas dators***
Procesors	*P4 1800 MHz*	*P4 1800 MHz*	*P4 1800 MHz*
Atmiņa	*2GB*	*2GB*	*2GB*
Tīkla savienojums	*100Mbit*	*100Mbit*	*100Mbit*
Operētājsistēma	*Windows 2003 server*	*Windows XP*	*Windows XP*
.NET	*.NET Framework 3.5*	*.NET Framework 3.5*	*.NET Framework 3.5*
Datubāze	*MSSQL 2005*		

Pirmā aplikācija sastāv no 2 daļām, kur katra daļa realizē savu arhitektūru polises izdošanas procesam. Pēc tam katra aplikācijas daļa tiek darbināta atsevišķi un iegūtie ātrdarbības rezultāti tiek reģistrēti un salīdzināti. Izstrādāto aplikāciju arhitektūra attēlota diagrammā (Attēls 3.3.3.).



3.3.3. att. **Testa aplikāciju uzbūves arhitektūra**

Esošās situācijas simulēšana

Pirmā aplikācijas daļa realizē patlaban sistēmā esošo arhitektūru. Visu procesu simulējošie dati tika iegūti, balstoties uz reālās sistēmas izpēti un SQL transakciju žurnāla analīzi (Attēls 3.3.4.).

Procesa sākumā tiek veikti dažādi aprēķini, kas praktiski nepatērē nepieciešamo laiku. Pēc tam tiek atvērta SQL transakcija un tiek veikti polises reģistrēšanu simulējoši procesi:

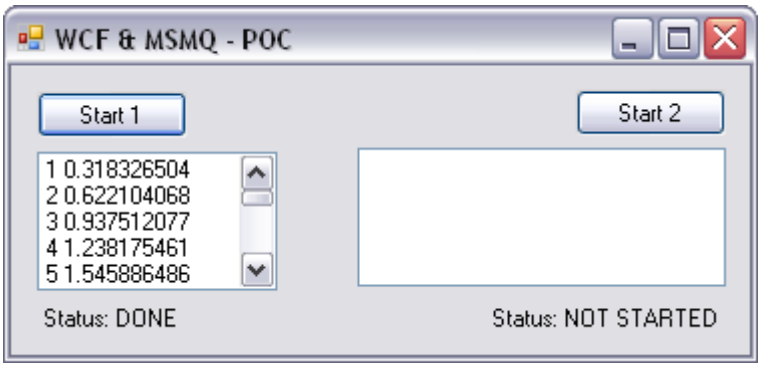
- Polises papildus datu saglabāšana
- Polises minimālo finansu datu pievienošana
- Klienta datu ielāde
- Polises statusa nomaiņa

Pēc polises datu reģistrēšanas tiek veikti finansu datu reģistrēšanas simulējoši procesi:

- Iesaistīto klientu datu ielāde
- Finansu klasifikatoru ielāde
- Finansu aprēķini
- Finansu datu saglabāšana

Pēc finansu datu reģistrēšanas SQL transakcija tiek aizvērta.

Viss process tiek atkārtots 100 reizes.



3.3.4. att. **1. aplikācijas 1. varianta izpildes rezultāts**

Plānotās situācijas simulēšana

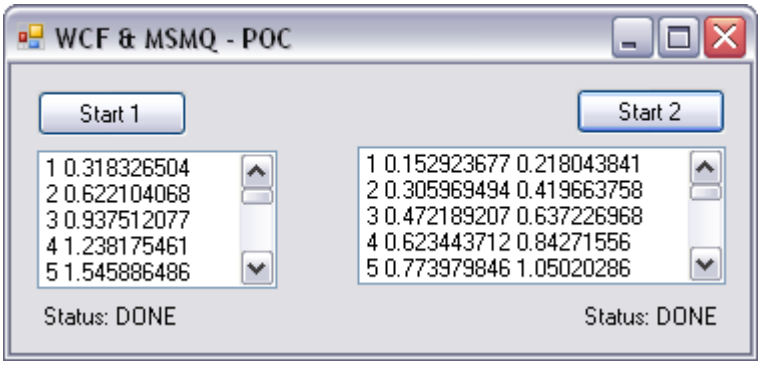
Otrā aplikācijas daļa realizē plānoto sistēmas arhitektūru. Visu procesu simulējošie dati tika iegūti balstoties uz reālās sistēmas izpēti un SQL transakciju žurnāla analīzi. (Attēls 3.3.5. un 3.3.6.)

Procesa sākumā tiek veikti dažādi aprēķini, kas praktiski nepatērē nepieciešamo laiku. Pēc tam tiek atvērta SQL transakcija un tiek veikti polises reģistrēšanu simulējoši procesi:

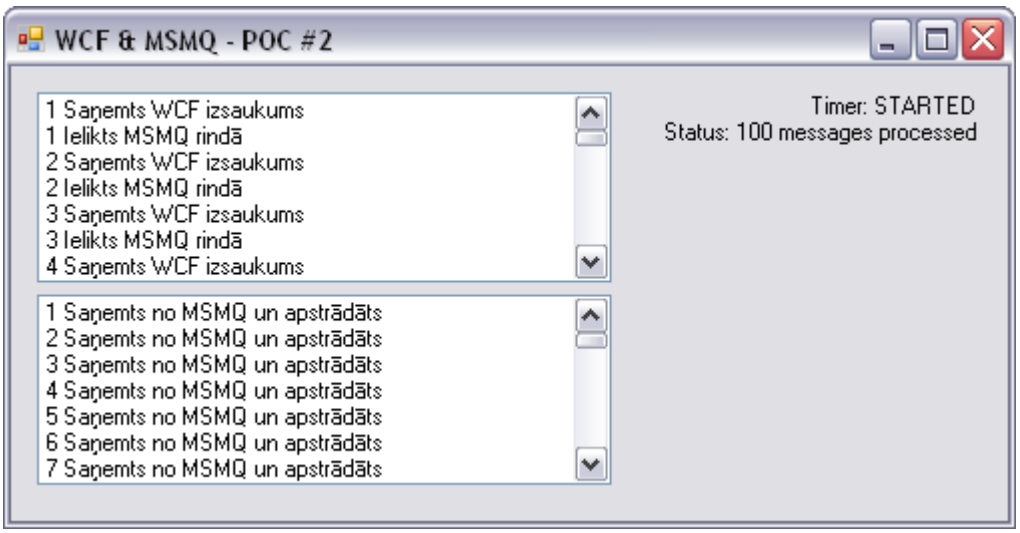
- Polises papildus datu saglabāšana
- Polises minimālo finansu datu pievienošana
- Klienta datu ielāde
- Polises statusa nomaiņa

Pēc polises datu reģistrēšanas SQL transakcija tiek aizvērta. Pēc tam ar WCF starpniecību tiek izsaukts finansu datu reģistrēšanas process no otrās aplikācijas, kas atrodas uz cita datora. Saņemot atbildi par veiksmīgu darba reģistrēšanu rindā process tiek uzskatīts par pabeigtu.

Viss process tiek atkārtots 100 reizes.



3.3.5. att. **1. aplikācijas 2. varianta izpildes rezultāts**



3.3.6. att. **2. aplikācijas izpildes rezultāts**

Paralēli uz otra datora, kur tiek darbināta otra aplikācija, aplikacija ar WCF starpniecību saņem darba uzdevumus no pirmās aplikācijas un reģistrē tos MSMQ rindā un atgriež atbildi par veiksmīgu datu reģistrēšanu rindā. Paralēls process saņemot no rindas finansu datu reģistrēšanas uzdevumu atver SQL transakciju un veic pilnvērtīgu finansu datu reģistrēšanu simulējošus procesus:

- Iesaistīto klientu datu ielāde
- Finansu klasifikatoru ielāde
- Finansu aprēķini
- Finansu datu saglabāšana

Pēc finansu datu reģistrēšanas SQL transakcija tiek aizvērta.

Pēc 100 ziņojumu apstrādes tiek izdots paziņojums, par veiksmīgu visu finansu datu reģistrēšanu.

Rezultātu salīdzināšana

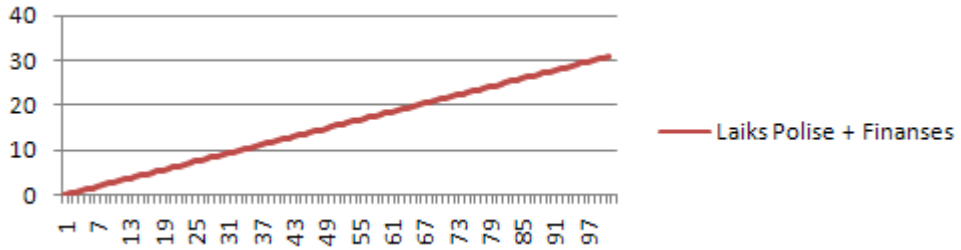
Iegūtie rezultāti tika apkopoti to analīzei (Tabula 3.3.3.), kā arī grafiski attēloti ērtākai pārskatīšanai (Attēls 3.3.7. un 3.3.8.).

3.3.3. Tabula

Mērījumu dati apskatītās arhitektūras salīdzināšanai ar patreizējo

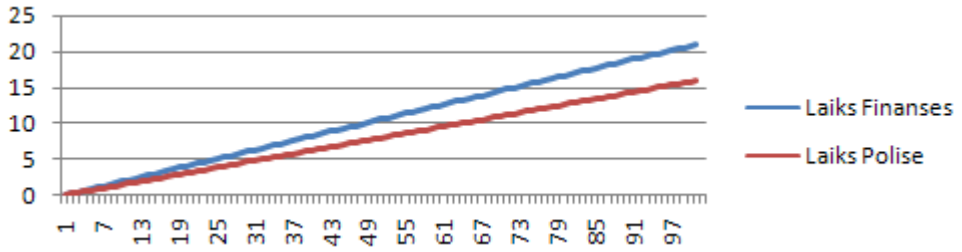
Iterācija	*Laiks Polise + Finanses (sek.)*	*Laiks Polise (sek.)*	*Laiks Finanses (sek.)*
1	*0.318326504*	*0.152923677*	*0.218043841*
2	*0.622104068*	*0.305969494*	*0.419663758*
3	*0.937512077*	*0.472189207*	*0.637226968*
4	*1.238175461*	*0.623443712*	*0.84271556*
5	*1.545886486*	*0.773979846*	*1.05020286*
6	*1.863485538*	*0.925149975*	*1.250586906*
7	*2.166078358*	*1.091101044*	*1.469342399*
...			
94	*29.2054697*	*15.00592744*	*19.76613207*
95	*29.51782233*	*15.16497302*	*19.97500314*
96	*29.83009358*	*15.31975178*	*20.19433616*
97	*30.14244511*	*15.47847005*	*20.39514828*
98	*30.44689071*	*15.64192518*	*20.6124825*
99	*30.75327632*	*15.80544696*	*20.81877415*
100	*31.06651739*	*15.97369822*	*21.03433007*

Grafiski apskatot pirmā varianta rezultātu ir redzams, ka viss process ilgst nedaudz virs 31 sekundēm, katras polises un finansu datu saglabāšanas vidējais ilgums ir 0.31 sekundes.



3.3.7. att. **Testa aplikāciju esošā varianta rezultāts**

Grafiski apskatot otrā varianta rezultātu, kur polises un finansu datu reģistrēšana ir salikta vienā grafikā ir redzams, ka kopējais procesa ilgums ir nedaudz virs 20 sekundēm, kas ir jūtami ātrāk nekā pirmajā variantā. Bet saskaitot kopā patērēto laiku polises un finansu datu reģistrēšanai ir nedaudz virs 37 sekundēm. Tas izskaidrojams ar to, ka aptuveni 0.06 sekundes tiek tērētas datu pārsūtīšanai starp serveriem un datu apstrādi rindā.



3.3.8. att. **Testa aplikāciju plānotā varianta rezultāts**

Par cik finansu datu reģistrēšana aizņem vairāk laika, nekā polises datu reģistrēšanai, tad MSMQ rindā jauni ziņojumi tika ielikti ātrāk, nekā ņemti no tās ārā.

No dotā eksperimenta rezultātiem var redzēt, ka risinājums, kas balstīts uz izmainīto arhitektūru, strādā par 30% ātrāk nekā patreizējais risinājums.

NOBEIGUMS

Iepazīstoties un pārbaudot jaunākās Microsoft tehnoloģijas var secināt, ka ne vienmēr tās uzreiz dod gaidīto ātrdarbības uzlabojumu. Piemēram, LINQ to Objects vienkāršākajā piemērā strādā 5 reizes lēnāk kā jau esošas C# valodas konstrukcijas.

Jaunās tehnoloģijas dažkārt mēdz ne tikai uzlabot ātrdarbību, bet arī ļaut izstrādāt aplikācijas daudz ērtāk nesamazinot to ātrdarbību, kā tas tika iepriekš apskatīts pielietojot LINQ Lazy-Load tehniku.

Balstoties uz iegūtajiem rezultātiem par esošās sistēmas uzbūvi un ātrdarbību, kā arī uz praktiskiem eksperimentiem par izmainītās arhitektūras ātrdarbību, var dot sekojošus ieteikumus sistēmas ātrdarbības uzlabošanai:

Ir nepieciešams izmantot LINQ lai programmas kods kļūtu pārskatāmāks, kā rezultātā tas veicinās koda kvalitāti. Kvalitatīvā kodā ir vieglāk pamanīt dažādas nepareizas vai neoptimālas konstrukcijas, kuru dēļ var stipri kristies aplikācijas ātrdarbība.

LINQ ir piemērots variants Lazy-Load tehnikas izmantošanai, jo nav nepieciešams rakstīt papildus pārbaudes un apstrādes moduļus.

Sistēmā nepieciešams veikt arhitektūras izmaiņas atbilstoši šajā darbā praktiski pārbaudītajam risinājumam. Izstrādātā sistēmas arhitektūra pārbaudītajā piemērā strādā atbilstoši plānotajam un dod 30% ātrdarbības uzlabojumu. Kā arī sistēma var tikt sadalīta uz vairākiem serveriem, kas dod papildus resursus sistēmas ātrdarbībai. No tā var secināt, ka realizējot šo arhitektūru esošajā sistēmā, arī būs līdzīgs ātrdarbības uzlabojums dotajam procesam. Esošajā sistēmā ir jāpārveido polises reģistrēšanas process atbilstoši aprakstītajai arhitektūrai, kā arī jāpārbauda pārējie sistēmas procesi un jāpielieto līdzīga metode.

Būvējot sistēmas arhitektūru, ir nepieciešams apzināt izvēlētās arhitektūras iespējas. Tas palīdzēs novērtēt sistēmas veiktspēju pie plānotās noslodzes, kā arī vienkāršāk un lētāk uzlabot sistēmas veiktspēju nākotnē, ja sistēmas noslodze palielināsies.

Lai turpinātu sistēmas veiktspējas uzlabošanu ir iespējams:

Veikt testus sistēmas maksimālās veiktspējas noteikšanai. Tāpat var eksperimentāli noteikt un grafiski attēlot veiktspējas pieauguma attiecību pret datorsistēmu izmaksu un skaita pieaugumu. Kā arī šajā darbā apskatītā veidā uzlabot sistēmas arhitektūru citos sistēmas procesos.

IZMANTOTĀ LITERATŪRA UN AVOTI

Drukātie izdevumi

1. Fowler M. Patterns of Enterprise Application Architecture, Boston: Addison Wesley, 2002. – 560 lpp. ISBN 0-321-12742-0

2. Davidson B.D., Liberatore V. Pushing Politely: Improving Web Responsiveness One Packet at a Time, Performance Evaluation Review, 2000, Nr 28-2, lapas 43-49.

3. Pialorsi P., Russo M. Introducing Microsoft LINQ, Redmond: Microsoft Press, 2007. – 240 lpp. ISBN 9780735623910

4. Lowy J. Programming WCF Services, Sebastopol: O'Reilly, 2007. – 634 lpp. ISBN 978-0-596-52699-3

5. Service Oriented Architecture for dummies / Aut. kol. J. Hurwitz, R. Bloor, C. Baroudi u.c. – Hoboken: Wiley Publishing, Inc., 2007. – 359 lpp. ISBN 978-0-470-05435-2

6. Improving .NET Application Performance and Scalability / Aut. kol. J.D. Meier, S. Vasireddy, A. Babbar, Microsoft, 2004. – 1152 lpp.

7.Upenieks Ģ. Uz MS SQL bāzētas finansu sistēmas veiktspējas uzlabošana, LU FMF fakultāte, Rīga, 2007. – 38 lpp.

Konferences

8. VS2008 un LINQ, HEROES happen {here}, V.Iļjučonoks, 2008. gada 3. aprīlis

9. Optimizing Performance and Scalability of Distributed .NET Applications, TechEd, Rammer I. (thinktecture), 2006

Elektroniskie avoti

10. Microsoft, Message Queuing (MSMQ). Pieejams Internetā: http://msdn.microsoft.com/en-us/library/ms711472%28VS.85%29.aspx

11. Rico Mariani, Microsoft, *Rico Mariani's Performance Tidbits*. Pieejams Internetā: http://blogs.msdn.com/ricom/archive/tags/databases/performance/default.aspx

12. Microsoft, Queues in Windows Communication Foundation. Pieejams Internetā: http://msdn.microsoft.com/en-us/library/ms731089(VS.85).aspx

13. Termini, Pieejams Internetā: http://termini.laka.lv/

14. Scaling Out SQL Server 2005

Pieejams Internetā: http://msdn.microsoft.com/en-us/library/aa479364.aspx

15. MSDN, Pieejams Internetā: http://msdn.microsoft.com/

16. Microsoft.NET, Pieejams Internetā: http://www.microsoft.com/net/default.aspx

Bakalaura darbs „Uz MS SQL bāzētu sistēmu veiktspējas problēmu meklēšana un risināšana” izstrādāts LU Fizikas un matemātikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Ģirts Upenieks

Rekomendēju darbu aizstāvēšanai

Vadītājs: Dr.sc.comp. Guntis Arnicāns

Recenzents: Mg.sc.comp. Aivars Niedrītis

Darbs iesniegts Datorikas nodaļā 27.05.2008.

Metodiķe: Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

10.06.2008. prot. Nr._____, vērtējums:

Komisijas sekretārs: Mg.sc.comp. Uldis Straujums