СОФИЙСКИ УНИВЕРСИТЕТ “СВ. КЛИМЕНТ ОХРИДСКИ”
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Катедра “Информационни технологии”

ДИПЛОМНА РАБОТА

за получаване на образователно-квалификационна степен “магистър”

тема:

Уеб базирани системи, представящи големи масиви от данни

Дипломант: Атанас Кънчов Гегов, ф.н. M21704
Магистърска програма: Разпределени системи и мобилни технологии

Научен ръководител: доц. Силвия Илиева,
катедра Информационни технологии, ФМИ, СУ “Св. Кл. Охридски”

София, 2007

Съдържание

Въведение 4
1. Шаблони за проектиране на уеб системи,боравещи с голямо количество от данни 7
1.1. Шаблони за проектиране,подходящи за управление на сложни, многослойни системи 8
1.1.1. Шаблона Model-View-Controller 8
1.1.2. Шаблона Front Controller 9
1.1.3. Шаблона Facade 14
1.2. Шаблони за проектиране за управление комуникацията с носителите на данни 15
1.2.1. Шаблона Data Access Object 16
1.3. Шаблони за проектиране за постигане на висока производителност 18
1.3.1. Шаблона Fast Lane Reader 18
1.3.2. Шаблона Page–by–Page Iterator 19
1.3.3. Шаблона Value Object 20
1.3.4. Шаблона Value Object Factory 21
1.3.5. Шаблона Value List Handler 22
1.3.6. Извод 24
1.4. Заключение 24
2. Техники за обработка на голямо количество данни 26
2.1. Техники за проектиране на потребителският интерфейс 27
2.1.1. Проектиране на филтри 27
2.1.1.1. Drop-down филтър 28
2.1.1.2. Филтер от тип линкове 30
2.1.1.3. Dual Slider филтер 31
2.1.2. Сортиране 32
2.1.3. Заключение 34
2.2. Техники за проектиране на процесите на сървъра..... 35
2.2.1. Техники за проектиране на процесите при изпълнение и подготовката на данните за визуализация 35
2.2.1.1. Техника 1: 36
2.2.1.2. Техника 2: 38
2.2.1.3. Техника 3: 39
2.2.2. Техники за проектиране на процесите при визуализация на резултата..... 40
2.2.2.1. Явно разпределение 42
2.2.2.2. Скритото разпределение 42
2.2.2.3. Комбинирано разпределение 42
2.3. Заключение 42
3. Бързодействието, при системи представящи големи масиви от данни 43
3.1. Стратегии за подобряване на бързодействието 44
3.1.1. Какво означава бързодействие? 44
3.1.1.1. Скорост на изчисление 44
3.1.1.2. Размера на RAM, която се използва по време на изпълнение 44
3.1.1.3. Времето на стартиране на приложението 45

3.1.1.4. Възможността на системата да се адаптира към повишена натовареност 45
3.1.1.5. Времето за отговор на отделната заявка от потребителя (при заявка за търсене, download на файл) 46
3.1.2. Фактори, влияещи върху процеса на бързодействие 46
3.1.2.1. Анализ 47
3.1.2.2. Обектно – ориентиран дизайн 48
3.1.2.3. Кодиране 49
3.1.2.4. Тестване 49
3.1.2.5. Профайлинг (Profiling) 51
3.2. Тактики за постигане високо бързодействието 53
3.2.1. Бързодействие, свързано с I/O 53
3.2.2. Размер на използваната памет 55
3.2.3. Контрол на зареждането на класовете 57
3.2.4. Алгоритми и структури от данни 57
3.2.4.1. Избор на алгоритъм 57
3.2.4.2. Сравнение на алгоритмите 59
3.2.4.3. Елегантни решения 59
3.2.4.4. Разглеждане на проблема за решаване 59
3.2.4.5. Рекурсивни алгоритми 60
3.2.4.6. Избор на структури от данни 60
3.3. Извод 61
4. Демонстрация на подходи на извличане на големи масиви от данни 62
4.1. Представяне на потребителският интерфейс 63
4.1.1. Традиционен подход 63
4.1.2. Page-by-Page Iterator 64
4.1.3. Value List Handler 65
4.2. Представяне принципите на извличане на данни 66
4.2.1 Принципи на извличане на данни според Традиционния подход 66
4.2.2 Принципи на извличане на данни според Page-by-Page Iterator 67
4.2.3 Принципи на извличане на данни според Value List Handler. 68
4.3. Изводи 69
Заключение 71
Терминологичен речник 72
Литературни източници 77

Въведение

При разработката на уеб базирани системи, представящи големи масиви от данни, ориентирани към потребители, които не са софтуерни специалисти, например корпоративния или научен софтуер, съществуват няколко основни проблема:

- намиране на подходящо решение за скелета на софтуер, представящ големи масиви от данни.
- представянето на големи масиви от данни по начин, който е удобен за възприемане от човек.
- бързодействието - представянето на големи масиви от данни пряко влияе върху него.

Цели и задачи

Намирането на актуални решения на тези три основни проблема в контекста на определени условия е основна цел на тази дипломна работа. Поради това, предложените решения са разгледани в дълбочина и освен на многото им преимущества е обърнато внимание и на техните недостатъци. Решенията са насочени предимно към софтуерния архитект и разработчик и отчасти към специалисти, следящи за качеството на софтуера.

Задачите на дипломната работа, които произтичат от поставените цели са:

- Да се разгледа естеството на гореспоменатите проблеми при различни условия и изисквания (като обща натовареност на системата, брой потребители, брой сесии, размер на данните, тип на данните, изисквания от клиентите, време за отговор, хардуерни изисквания и др.).
- Да се направи обзор на шаблони за проектиране, даващи актуални решения за скелета на софтуер, представящ големи масиви от данни и да се изведат препоръки за употребата им.
- Да се направи обзор на техники за проектиране на потребителски интерфейс при визуализацията на големи масиви от данни, включващ в себе си оценка и сравнителен анализ на конкурентни решения.
- Да се направи обзор на техники за проектиране на процесите на сървъра за извличане и подготовка на голямо количество от данни за визуализация, включващ в себе си оценки и сравнителен анализ на конкурентни решения.
- Да се систематизират стратегии за подобряване на бързодействието, ориентирани към дизайна на софтуера на високо ниво.
- Да се систематизират тактики за постигане на високо бързодействие, ориентирани към най-ниско ниво – ниво на писане на програмен код.
- Да се демонстрират конкурентни подходи на представяне на голямо количество от данни и да се обобщат резултатите.

В тази дипломна работа, системите, които стоят на централно място и върху които се фокусира цялото внимание, са уеб базирани. Основни черти, които ги характеризират са големия брой конкурентни потребители, представянето на големи масиви от данни (които могат да се разглеждат като дълги списъци от обекти) и накрая, постоянната натовареност или големия брой активни сесии. Затова навсякъде тук ще се взимат предвид тези фактори. Някои от предложените препоръки най-вероятно няма да

са в сила или отчасти ще бъдат в сила за друг тип системи. Те ще са приложими изцяло, единствено за системите, които отговарят на описанието по-горе.

Полза от дипломната работа

Следването на предложените препоръки в тази дипломна работа ще помогне в изграждането на такъв тип системи. Като се започне от етапа на анализ и се стигне до етапа на тестване. С помощта на тези пропоръки се постига по-бърза разработка и по-високо качеството на софтуера, намаляват се слабостите в дизайна на структурата на системата, дизайна на потребителският интерфейс, в качеството на кода и в процеса на тестване. По този начин разработваната система по-лесно може да изпълни изискванията, които са поставени пред нея, както и да постигне добър успех сред своите потребители.

Структура на дипломната работа

Глава 1 - „*Шаблони за проектиране на уеб системи, боравещи с голямо количество от данни*" разглежда подходящите решения за скелета на софтуер, представящ големи масиви от данни. Представените решения в тази глава са чрез шаблони за проектиране (design patterns), но от най-високо ниво, ниво на комуникация на отделните модули, съставящи приложението.

Шаблоните са групирани в три основни групи:

- Шаблони за проектиране, подходящи за управление на сложни, многослойни системи.
- Шаблони за проектиране, подходящи за управление на комуникацията с носителите на данни (релационни бази от данни, файлови сървъри, XML бази от данни).
- Шаблони за проектиране, подходящи за постигане на висока производителност.

Глава 2 - „*Техники за обработка на голямо количество данни*" разглежда проблемите, свързани с проектирането на визуализацията на данните във вид, който е максимално удобен за масовия потребител. Както и с използване на разнобразни техники на проектиране на процесите, грижещи се за извличането и подготовката на данните за визуализация.

Глава 3 - „*Бързодействието, при системи представящи големи масиви от данни*" - е насочена към постигането на висока производителност на софтуера. Процесът е разделен в две насоки – стратегии и тактики:

- стратегии за постигане на по-висока производителност, ориентирани към дизайна на софтуера на високо ниво.
- тактики за постигане на по-висока производителност, разглеждани като препоръки от ниско ниво – нивото на програмен код.

Глава 4 - „*Демонстрация на подходи на извличане на големи масиви от данни*" - представя уеб базирано приложение, илюстриращо три различни подхода при извличане на големи масиви от данни:

- Традиционнен подход.
- Подход основан на Page-by-Page Iterator.
- Подход основан на Value List Handler.

Глава 1

Шаблони за проектиране на уеб системи, боравещи с голямо количество от данни

При дизайна и разработката на уеб системи, разработчиците се сблъскват със сходни проблеми. Една успешна практика за намаляване на усилията при разработването, е да се черпи опит от вече създадени други такива системи. Задачата може би не изглежда тривиална, едва ли са много разработчиците, които биха си предоставили знанията от натрупан опит и умения или поне не безплатно. На пръв път поглед изглежда, че задачата може да е скъпо струваща, но съвсем не е така. Съществува каталог от шаблони за проектиране, приложими почти за всички обектно–ориентирани езици, а също и за някои не обектно-ориентирани. Достъпни за всеки, създадени от професионалисти и даващи елегантни решения на точно тези проблеми, с които би могъл да се сблъска всеки един разработчик. Тези решения най-общо казано представляват дизайн и взаимодействие на обекти и класове, предоставящи възможно най-добре структурирана схема, позволяващи бърза разработка и лесна модификация (ако се налага, при евентуална промяна в изискванията от клиента или по други причини).

Шаблоните за проектиране предоставят общи решения, които могат да бъдат приложими в най-различни сфери от реалния живот. Ето защо не представлява никаква пречка, приложения с коренно различно предназначение да бъдат изградени въз основата на едни и същи шаблони за проектиране.

Най-популярният каталог от тях е „*Gang-Of-Four*" заради четиримата автори, които са написали книгата „*Design Patterns: Elements of Reusable Object-Oriented Software*" [19] и са въвели това понятие.

Тук обаче предметът на тази глава е малко по-различен, той изхожда от няколко основни ограничения, стоящи пред разработчика – трябва да се разработи система, която ще разполага с огромно количество данни, може да има много конкурентни потребители и ще бъде натоварена през по-голямата част от времето си. Шаблоните, подходящи за тази цел съвсем не са много и може да се използват като комбинация от няколко от тях. Но не подлежи на съмнение, че с тяхна помощ, разработката на една такава система се опростява и ускорява.

Като се има предвид най-общо какви ще са целите на разработваната система, има три основни характерни черти, които тя задължително трябва да притежава, за да може да удовлетвори гореизложените изисквания. Или казано по друг начин, необходими са следните видове шаблони:

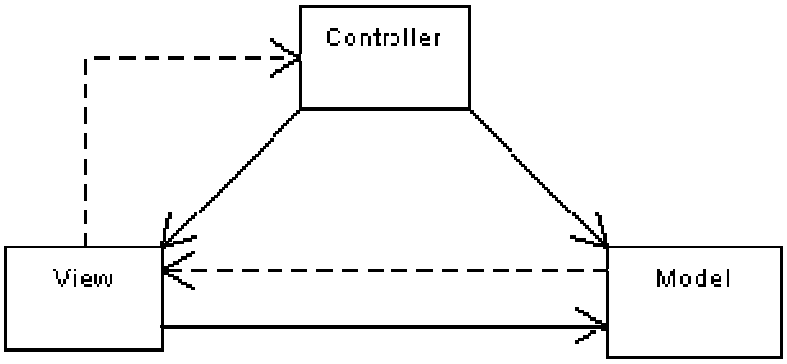
1. Шаблони за проектиране, подходящи за управление на сложни, многослойни системи.
2. Шаблони за проектиране, подходящи за управление на комуникацията с носителите на данни (релационни бази от данни, файлови сървъри, XML бази от данни).
3. Шаблони за проектиране, подходящи за постигане на висока производителност.

1.1. Шаблони за проектиране, подходящи за управление на сложни, многослойни системи.

Уеб система, която ще обработва голямо количество данни, ще подържа голям брой конкурентни потребители и ще бъде натоварена през по-голямата част от времето си, най-вероятно няма да бъде лесна за изграждане. Може да се каже, че характерен белег на този тип системи е тяхната многослойна структура, съставена от поне три слоя – презентационен слой, слой на бизнес логиката и слой за управление на самите данни.

1.1.1. Шаблона Model-View-Controller

Един от шаблоните, който може да задоволи изискванията на този тип системи е Model-View-Controller (MVC), Фиг. 1.



Фиг. 1 - Model-View-Controller

Непрекъснатите линии показват директна връзка, прекъснатите линии показват индиректна връзка между отделните елементи.

Този шаблон е особено подходящ, когато се предоставят голямо количество от данни на потребителите[5]. В практиката често се налага разработчиците да разделят данните (Model) и потребителския интерфейс (View) един от друг. Така, че промяна в потребителския интерфейс да не влияе върху обработката на данните, както и промяна в данните да не налага промяна в интерфейса към потребителите. Mode – View – Controller разрешава този проблем като капсулира достъпа до данните и бизнес логиката от представянето на данните и потребителските действия. Това става чрез добавяне на един междинен компонент - Controller.

Кратка характеристика на всеки един от модулите на този шаблон[11],[12],[13]:

Model

Представлява информацията, данните, с които работи приложението. Системата, боравеща с много данни най-вероятно ще използва база от данни, поради преимуществата, които предоставя тя при организацията на данните. Комуникацията с нея, както и дефинирането, и използването на съхранени функции в базата се намират точно тук.

View

Този елемент както е отбелязано по-горе дефинира потребителския интерфейс.

Controller

Получава и изпраща събития. Най-често, това са потребителски действия. От тук може да се предизвика промяна в Model или казано по друг начин, може да управлява както извличането, така и промяната на данните.

Процесът на взаимодействие между отделните елементи в този шаблон най-често може да се опише така:

1. Потребителят чрез потребителски интерфейс изпълнява заявка, посредством графични контроли (View)
2. Controller (код на някакъв обектно – ориентиран език, който прихваща събития и генерира например динамични HTML страници), прихваща събитието от потребителя.
3. Controller изпраща заявка към базата от данни (Model) (или друг тип носител), заявката е за удоволетворяване искането от потребителя. Може да бъде просто извличане на данни от таблица или от група таблици, може да бъде промяна на данните в таблиците, може да извика процедура от базата данни и да чака резултата от нейното изпълнение и т.н..
4. Controller получава резултата и препраща най-често динамичен HTML към потребителя.
5. Потребителският интерфейс (View) показва резултата и чака последващи събития за да се завърти цикъла отначало.

1.1.2. Шаблона Front Controller

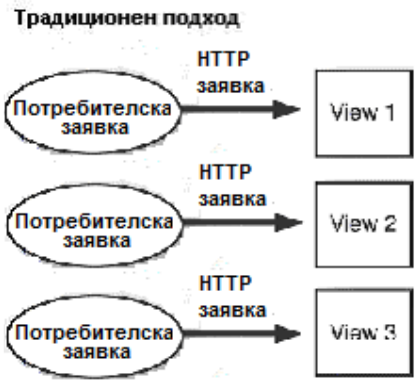
Front Controller наподобява донякъде шаблона Model – View – Controller. Когато Controller и View са свързани с много и сложни взаимодействия, тогава на помощ идва този шаблон. Front Controller служи като единствена отправна точка, през която се достъпват различен набор ресурси. Всичките заявки за информация минават през него, като източникът на данни може да бъде най-различен, а front controller-а ги разпределя по предназначение.

Най-често този шаблон се използва за следните задачи[22]:

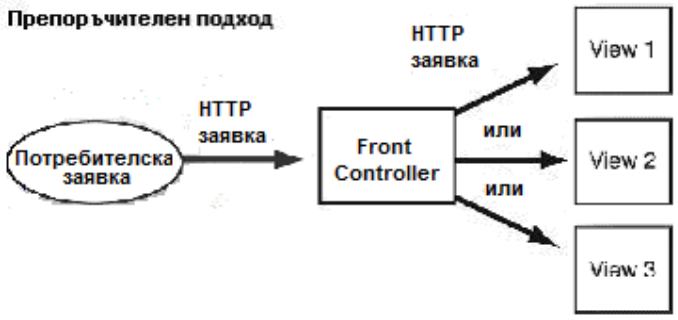
- подпомага навигацията в уеб системите.
- за достъп и управление на някакъв носител на данни, най-често бази от данни, т.к. почти винаги става въпрос за изключително много информация.

- обработка на сложни бизнес процеси.

Този шаблон може да намали дубликацията на код, в случаите когато няколко ресурси изискват един и същи начин на процедиране. Когато става въпрос за огромни и сложни уеб приложения, може да подобри поддръжката и контрола, като централизира всичките потребителски заявки през един Front Controller. Също така значително може да подобри сигурността на едно приложение.



Фиг. 2 – Традиционнен подход.



Фиг. 3 – Препоръчителен подход.

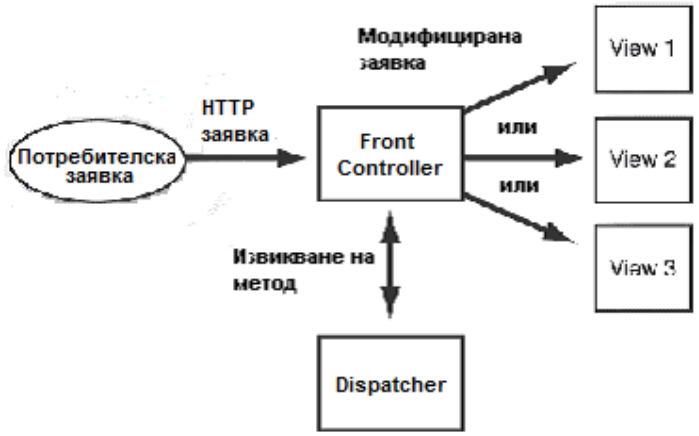
В Фиг. 3 е показаната структура на препоръчителен подход, което е един от вариантите на използване на този шаблон. Лесно се виждат преимуществата по организацията и поддръжката пред традиционния подход (Фиг. 2), при който всяка потребителска заявка трябва да бъде пренасочена към отделен ресурс.

По-трудно забележимо остава неговото преимущество, когато трябва да представи голям списък от обекти на потребителя. Когато всичките тези данни се контролират през един централен компонент, системата може да използва някои похвати, с които да ограничи изразходваемостта на своите ресурсите. Такива похвати са например кеширане, също така представяне на данните по специфични начини (итеративни списъци, с динамично зададени размери, при които не е необходимо извличане на

всички данни и др). При липсата на този компонент, контролът на даните по указаните начини е трудно реализируем.

Dispatcher

Front Controller има и други свои разновидности, като също така се използва и в комбинации с други шаблони за проектиране. Един такъв вариант е когато той координира навигацията на потребителите, с помощта на друг шаблон за проектиране – т.нар. Dispatcher (Фиг. 4).



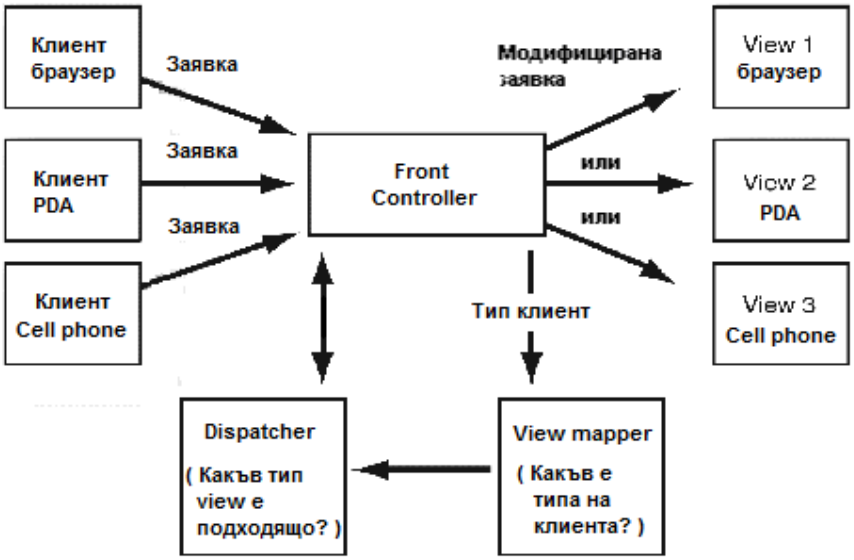
Фиг. 4 - Front Controller с Dispatcher

Dispatcher капсулира пренасочването на потребителските заявки към определени компоненти. Както се спомена по-горе, тези компоненти могат да пестят ресурси на сървъра като предоставят данните към потребителите от кеш или като изпращат дълги списъци от обекти не наведнъж, а раздробени на отделни части.

View Mapper

Когато уеб ресурсите се различават в зависимост от типа потребител, може да се използва View Mapper за да се подпомогне механизма по препращане на потребителя към правилния ресурс. Такива потребители могат да бъдат – уеб браузери, PDA, GSM. Такъв пример е приложение, даващо информация за цените на стоковата борса. В такава ситуация потребителят може да иска да види тази информация от различен източник в зависимост къде се намира – на персоналния си компютър или PDA. Точно тук е мястото на модула View Mapper в шаблона Front Controller, като преценява типа на клиента и му препраща точният вид страница. Защото най-малката разлика между двете страници може да бъде в начина на разположение на данните, например ако на клиента трябва да се върне резултат от 1000 реда, в десктоп приложението може да се разпределят на 20 страници по 50 реда, докато за малкия дисплей на мобилното устройство това не е удобно и по-добрият вариант е да се позиционират данните на 100 страници по 10 реда. Тук в никакъв случай не би трябвало да се изпраща целият списък от 1000 реда наведнъж. Десктоп приложението може и да се справи с визуализацията,

но за мобилното устройство тази задача ще е невъзможна или най-малкото ще отнеме прекалено много време.

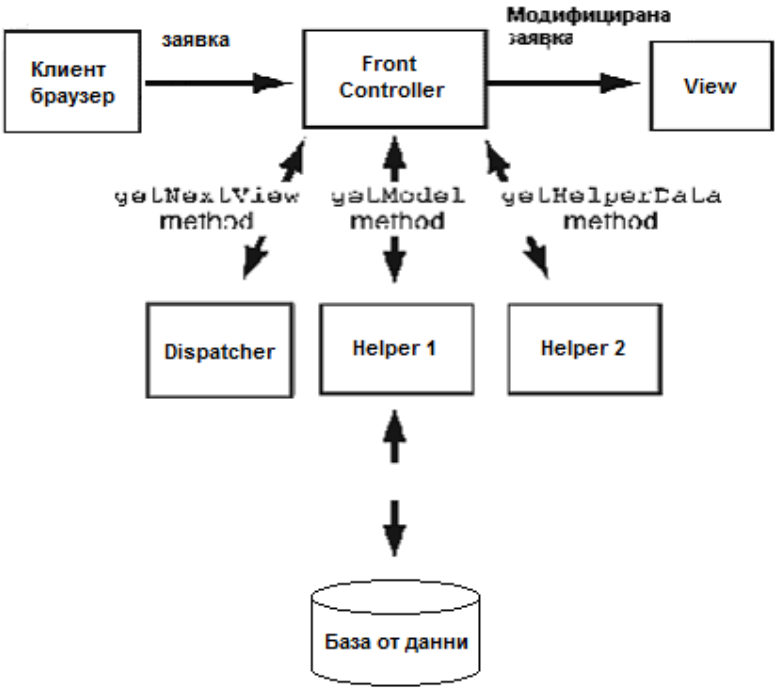


Фиг. 5 - Front Controler с View Mapper

Helper класове

Самият компонент Front Controller в шаблона Front Controller лесно може да нарасне прекалено много на големина и да се усложни. Когато системата притежава сложна бизнес логика, може да се окаже, че в него са капсулирани твърде много задачи. По тази причина е добра идея да се ползват Helper класове за да бъде системата по-проста и по-лесна за поддръжка (Фиг. 6). Ето какви задачи могат да имат Helper класовете:

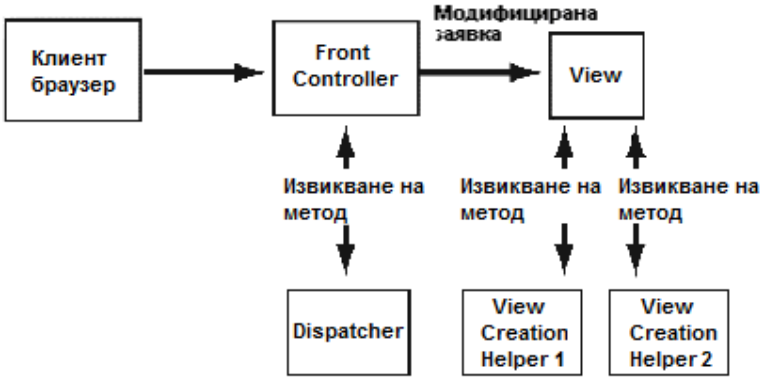
- Получаване на данни от файл, друг уеб сайт и др.
- Валидация на въведената от потребителя информация.
- Управление на бизнес логиката.
- Управление на данните.



Фиг. 6 - Front Controller и Helper класове

View creation Helper класове

Ако се разгледа в детайли шаблона Front Controller и се обърне внимание на примера, даден като препоръчителен във фиг. 3., може да се види как Front Controller избира правилния View клас в зависимост от търсения ресурс. Но това може да се реализира и по друг начин. Възможно е Front Controller да се обръща само към един View модул, който от своя страна да се грижи за извикването на правилния ресурс (Фиг. 7). По този начин се капсулира логиката, свързана с представянето на информацията спрямо различните потребители. Всичко това се реализира с помощта на Helper класове.

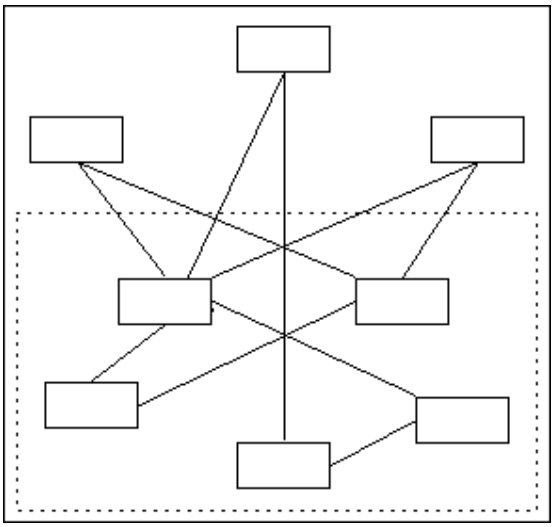


Фиг. 7 - Front Controller с View creation Helper класове

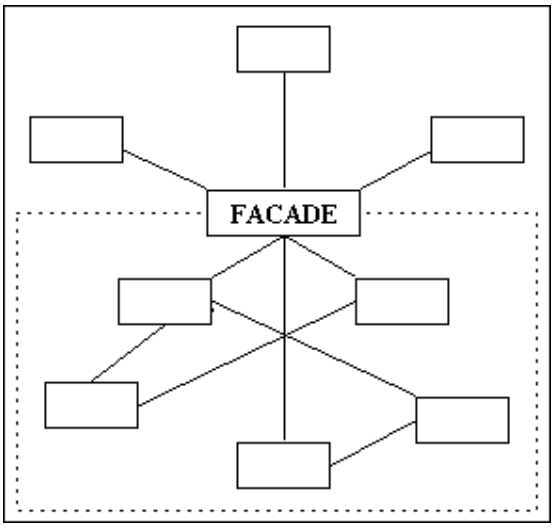
1.1.3. Шаблона Facade

Включването на много бизнес процеси води до сложно управление на класовете. Сложните процеси, които включват множество бизнес класове може да доведе до твърде голяма обвързаност и зависимост между тях, което от своя страна води да слаба гъвкавост и неясен дизайн и дори до слаба производителност (пример за такава система дава Фиг. 8). Ако всеки от тези класове притежава големи структури от данни, хаотичната комуникация между тях може да доведе до създаване на повече инстанции на класовете отколкото е необходимо, водещо до излишно изразходване на ресурси.

Шаблона за проектиране Facade (Фиг. 9) дефинира компонент от високо ниво, който централизира връзката между бизнес класовете. Той премахва голямата зависимост между тях като прави дизайна на системата по-гъвкав и ясен.

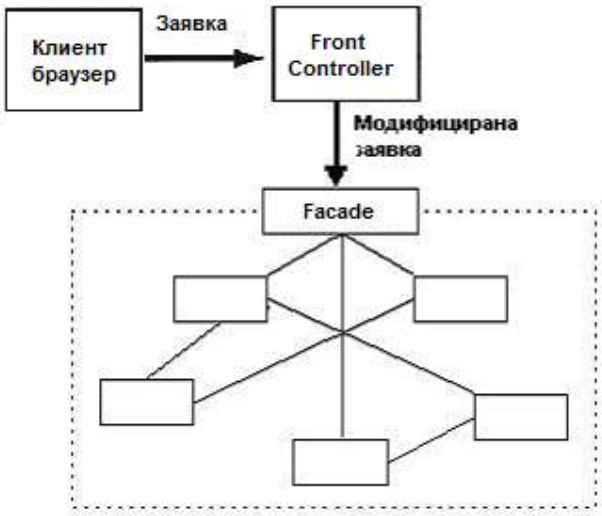


Фиг. 8 - Дизайн преди използването на шаблона Facade



Фиг. 9 - Дизайн след използване на шаблона Facade

Различните шаблони, представени по-горе могат да бъдат елегантно решение за системи, характеризиращи се с висока степен на сложност, боравещи с огромно количество от данни. Поради тази причина горещо се препоръчва на разработчиците да използват и следват идеите и препоръки, представени в тях. Не са редки случаите, когато дизайн решения включват комбинация от няколко шаблона. Често срещан пример е Front Controller да прихваща всички потребителски заявки, да ги обработва и да се обръща към Facade за информация, свързана с бизнес логиката, както е показано на фиг. 10.



Фиг. 10 - Комбинация от Front Controller и Facade

1.2. Шаблони за проектиране за управление комуникацията с носителите на данни

Такъв тип шаблони подпомагат комуникацията със същинските данни, с които разполага системата (релационни бази от данни, XML бази от данни, файлови сървъри и др.). Тъй като тук става въпрос за приложения, притежаващи огромно количество от информация, може да се предположи, че е твърде вероятно да имат и сериозно заложена бизнес логика.

Едно от изискванията за добър дизайн на такива системи са лесната поддръжка. То е необходимо за да е в състояние приложението да удовлетворява бъдещите желания на своите потребители и лесно да се настройва спрямо някои нови функционалности. И за да е по-ясен проблемът, нека се разгледа като пример система, при която източникът на данни е изключително разнообразен, разнороден, в огромни количества и изменящ се във времето. Нека това приложение да има сложен потребителски интерфейс и сложна бизнес логика. Освен това клиентите да могат да бъдат много и от различен тип (браузер, pda и др.). Накрая в изискванията част от информацията да идва от собствена релационна база от данни, а друга част от уеб пространството.

На фона на всичко това, възможно е да се допусне грешка в проектирането, която до доведе например до по-ниска производителност от планираното в началото. Затова всякакви идеи, препоръки, имплементации на подобни решения, са от изключително

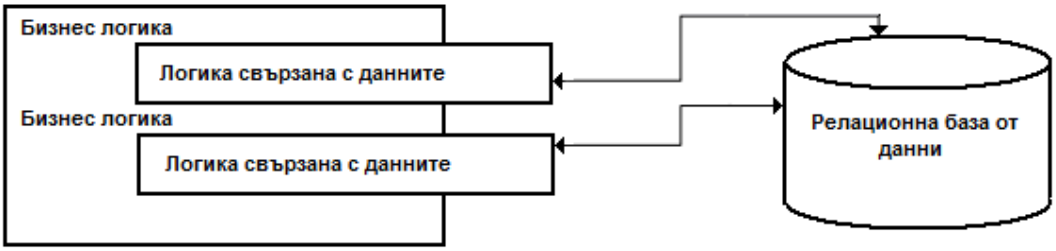
голяма полза. На помощ идва шаблонът за проектиране Data Access Object, който може да послужи за основа при проектирането на една такава система, притежаваща огромни и разнородни източници от данни.

1.2.1. Шаблона Data Access Object

Основната идея, която е залегнала в този шаблон е разделянето на логиката, свързана с данните, от класовете, които ги достъпват. Изразява се в капсулация на достъпа и обработката на данните в отделен слой (например това може да бъде капсулиране на достъпа и обработката на данни от релационна база от данни).

Data Access Object (DAO) внася идеята за абстрактен слой между бизнес слоя и слоя, грижещ се за данните[19]. Бизнес обектите достъпват данните чрез DAO обекти. Така приложението постига гъвкавост, и също така, промени, направени в данните не рефлектират върху бизнес класовете. Например, смяната на типа носител на данни, както и смяна между различни видове релационни бази от данни не изискват промяна в кода, отговарящ за бизнес логиката.

Пример от най-високо ниво на такава система преди прилагането на DAO (фиг 11).



Фиг. 11 – Система преди прилагането на DAO

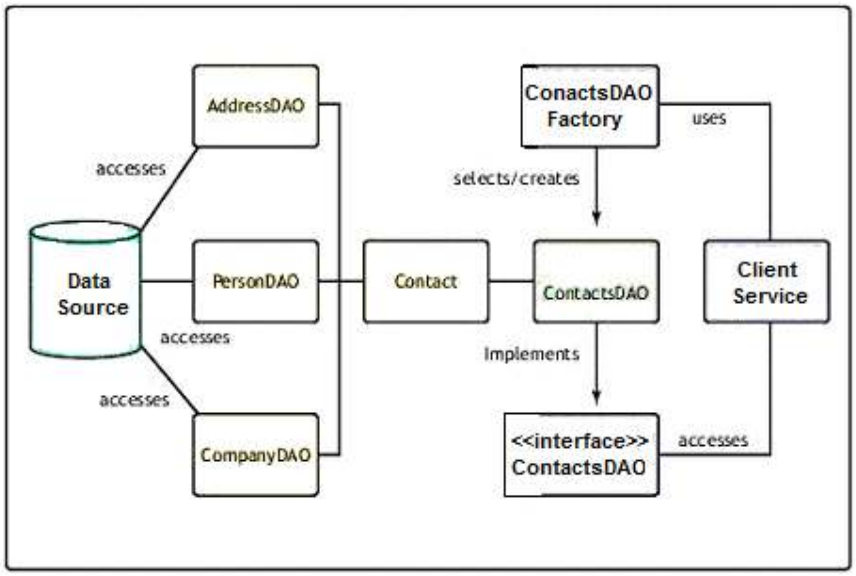
Пример от най-високо ниво на система след използването на DAO (фиг 12).

Фиг. 12 – Система след използването на DAO

DAO има следните преимущества:

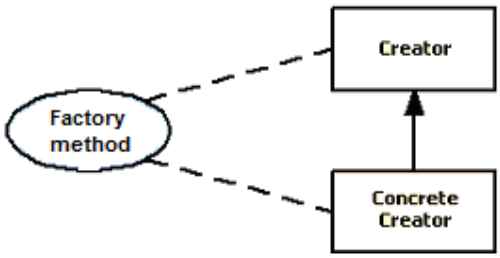
- Отделя данните от бизнес логиката
- Адаптира библиотеките за управление на данните към специфичен интерфейс.
- Позволява имплементацията на достъпа до данните да се настройва самостоятелно от кода, който използва DAO.
- Позволява да се променя механизма на достъп до данните без да се правят промени в програмния код.

Пример на шаблона DAO, достъпващ бизнес контакти (фиг 13):



Фиг. 13 - DAO, достъпващ бизнес контакти

В основата на имплементацията на DAO стои така популярният шаблон за проектиране Factory Method (Фиг. 14):



Фиг. 14 – Factory Method

Едно от преимуществата свързани с шаблона DAO е възможността чрез негова помощ, динамично да се променя начина на извличане на данните. Извличане на данни в зависимост от типа на клиента, от типа на данните, от размера на данните, от местонахождението им и от други фактори, благодарение, на които могат да се пестят важни ресурси на системата. Код от сървъра динамично чрез Factory в комбинация с DAO използвайки един общ интерфейс преценява от къде най-бързо и най-лесно, може да извлекат нужните данни. Като изборът може да се направи из между различни източници - различни релационни бази от данни, различни интернет местонахождения, LDAP сървъри и т.н. Възниква въпросът защо са необходими всичките тези източници, като най-вероятно се знае предварително кой е най-бързият и елегантен подход на извличане на данни (например данните се намират на две места - в релационна база от данни и в някакъв адрес в интернет пространството)? Или ако информацията се съдържа на няколко места как се синхронизират помежду си данните? Затова ще се даде пример, в който да стане напълно ясно какви са преимуществата при използване

на DAO и защо с помощта на него се постига много добър ефект при извличането и предаставянето на големи масиви от данни. Примерът е свързан със система извличаща информация от data warehouse. Такъв тип бази от данни най-общо казано е огромно хранилище от данни, което през определен период от време се захранва от множество източници. Процесът на захранване включва извличане на данните от първичните източници (може да бъдат други релационни бази от данни), почистване и форматиране, проверка за дублиране, проверка за съответствие с ограниченията и т.н. Данните веднъж влезли в Data warehouse пак на определен период от време могат да бъдат разпределени в по-малки логически единици (data marts), които съхраняват информацията в по-обобщен вид, за да може по най-бързият и лесен начин да бъде доставена до потребителя. На фона на целият този сценарии един потребител изисква определен тип информация към сървъра. Сървърът трябва да прецени как най-бързо и лесно да удовлетовори заявката към него. Затова той динамично с помощта на или на DAO за date warehouse (fact tables), или с DAO за data marts или дори с DAO за първоизточниците на data warehouse извлича една и съща информацията необходима на потребителя. В организацията на целият този механизъм стои комбинация на Factory и DAO, като основното е, че съществуват много DAO, но всичките имащи едно и също предназначение от гледна точка на клиента, но реализиращо достъп до различните източници на данни.

1.3. Шаблони за проектиране за постигане на висока производителност

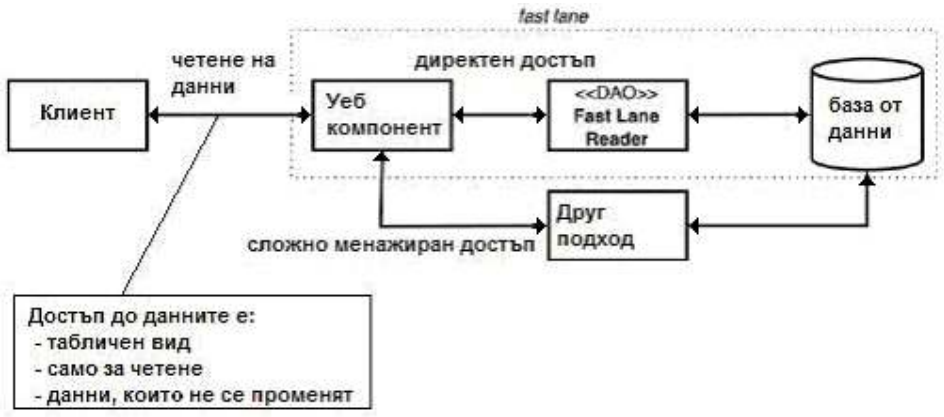
Многослойните уеб приложения се състоят от компоненти, които комуникират помежду си при преноса на данни към потребителя. Това често води до отдалечени извиквания между клиенти, динамични уеб страници (jsp, asp и др.), класове, съхраняващи бизнес логиката, класове, генериращи динамични уеб страници и др. Такава комуникация е често скъпо струваща и засяга бързодействието на приложението като цяло. Несъобразеното увеличение в броя извиквания между отделните компоненти може да увеличи мрежовия трафик. Когато тези извиквания са свързани и с пренос на данни по мрежата, производителността може да се намали до критични нива. Ако размерът на данните е голям, разработчиците може да се замислят за реорганизация дори на ниво дизайн (изключително скъпо струваща операция). Поради тази причина, тук ще се предложат някои препоръчителни шаблони за проектиране, с помощта, на които да се изграждат такива системи. Тези шаблони трябва да се прилагат още във фазата на дизайна, за да се избягват максимално проблеми от такъв характер.

Списъкът от шаблони е както е споменато в [20].

1.3.1. Шаблона Fast Lane Reader

Много уеб приложения имат за цел да визуализират големи списъци от данни в табличен вид. Нека да се вземе за пример система на финансова институция, която предлага услугата онлайн банкиране. Част от функциите и е да визуализира дълъг списък от финансови транзакции за изминал период от време. Обща характеристика на такъв тип функционалност е, че обслужва голям брой конкурентни потребители, с много данни, рядко променящи се във времето.

Едно от препоръчителните решения за дизайн на такава система е шаблона Fast Lane Reader (Фиг. 15) , на който част от идеологията му е заимствана от шаблона Data Access Object[15].



Фиг. 15 - Шаблона Fast Lane Reader.

Основното преимущество на този шаблон е олекотения и по-бърз достъп до данните, алтернативен на типичния и по-сложен достъп.

При протичането на процесите в такава система, се случва да се извличат данни, които имат табличен вид, ще се използват за четене и не се очаква да се променят от момента им на извличане от носителя, където се съхраняват. Добро решение е да се използва метод, който е бърз, лесен и натоварва до минимум системата - Fast Lane Reader[16]. Голямо преимущество на целия този сценарий е, че може да се вземе решение как да се предоставят данните на потребителя динамично, по време на изпълнение на заявката. Ако тя отговаря на гореизложените условия, се използва този подход, който е по-бърз и пестящ ресурси метод вместо стандартния и сложно менажиран.

1.3.2. Шаблона Page–by–Page Iterator

Когато едно приложение има за цел да извлече голямо количество данни наведнъж и колкото и ефективен да е методът на достъп, извикването му прекалено често, може да повлияе много негативно върху неговото бързодействие. Затова е добре да се разгледа вариант, който е по-олекотен и пести максимално ресурсите на системата. Основната идея, която е заложена е, как да не се извлича дългият списък от обекти наведнъж, а само частта, която ще е нужна на потребителя на първо време. Отговорът на този въпрос ще допринесе до значително ускорение на производителността на цялата система. На нея ще са и необходими много по-малко ресурси - заделяне на много по-малко памет и много по-малко натоварване на мрежата.

Съществува шаблон за проектиране, който решава този проблем (Фиг. 16) – Page-by-Page Iterator[18]. Той връща списък от обекти, който има размер колкото е необходим на потребителя или колкото той е поискал от системата. Ако на потребителя са му нужни 10% от данните, му се изпращат само толкова, ако поиска следващите 10% или 20% му се изпращат отново толкова.

Фиг. 16 - Шаблона Page-by-Page Iterator

Ако се наложи, може да се изпрати и целия обем данни, но това дори и да се наложи, може рядко да се случи или пък да се окаже, че списъкът, който очаква потребителят не е толкова голям. Благодарение на този подход обаче се избягва излишното натоварване на системата при прехвърляне на огромни списъци с обекти. Прехвърлят се данни между слоевете и само в количество, което е необходимо на потребителя.

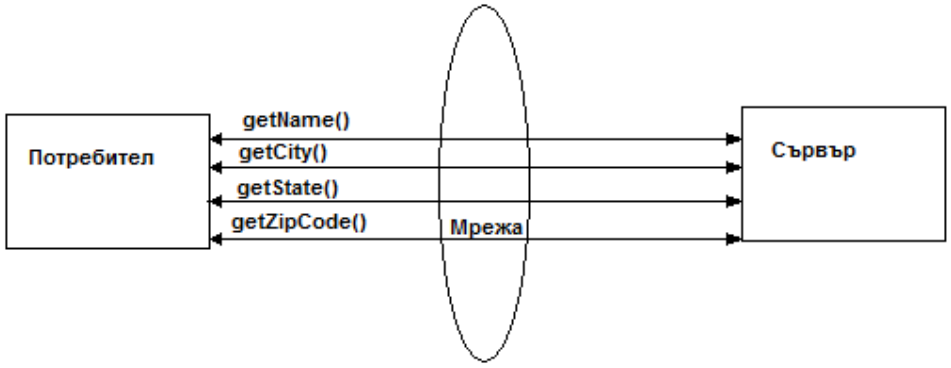
Този подход притежава и друго преимущество - заявките към базата от данни (носителят, съхраняващ данните най-често е релационна база от данни). Те са създадени така, че да не обхождат всичките данни, а само част от тях, която е необходима (ако потребителят е поискал 10%, само тази част), което отново е по-лек вариант от стандартния и пести ресурси на приложението.

Описаният шаблон си има и известни недостатъци. Когато потребител обхожда страниците на някакъв голям списък от данни и между временно данните, претърпят някаква промяна се получава известно изкривяване на информацията. Но ако данните са сравнително статични този проблем спокойно може да се пренебрегне. Друг евентуален недостатък е, че за някои приложения е от важно значение да се знае какъв е общият размер данни или колко евентуално страници може да итерира един потребител. Тук евентуално разрешение се намира като се измени стандартната дефиниция на този шаблон. Изменението се състои като при извикване на първа заявка (или на всяка заявка) за страница към сървъра - да се изчислява и общият размер от страници, които евентуално биха се итерирали. Тази модификация не винаги е универсална, но може да се има предвид при евентуална интерпретация на този шаблон.

1.3.3. Шаблона Value Object

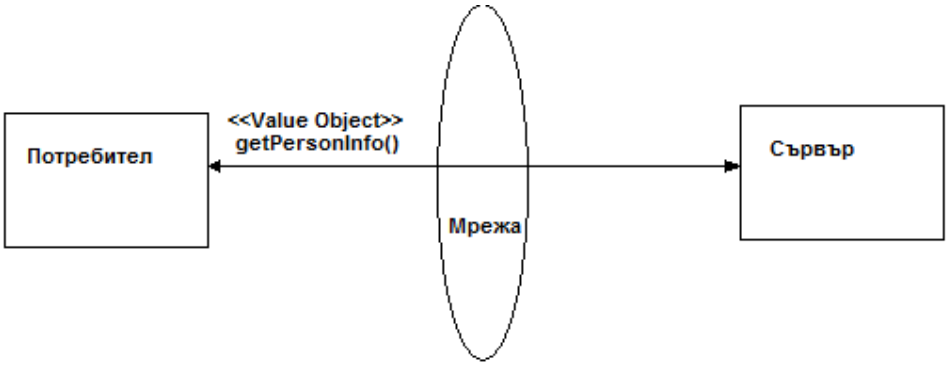
Когато една система обработва големи масиви от данни, разработчиците често се притесняват за нейното бързодействие, процесът на извличане / визуализиране на данните по принцип е тежък и изразходва много ресурси. Поради тази причина те хвърлят много усилия за неговото оптимизиране. Но някои допускат грешката да пропуснат и да не обърнат достатъчно внимание на други процеси, маловажни на пръв поглед, които могат обаче да доведат до ниско бързодействие. Пример за такъв процес е, когато потребител желае да получи информация, която е разбита на много части и изисква множество заявки към сървъра (Фиг. 17). Тези заявки освен, че ненужно натоварват мрежата, изразходват важни ресурси на сървъра.

Традиционен подход при извличане на информация, която е раздробена на много съставни части.



Фиг. 17 - Традиционен подход

Решение на този проблем е да се избегне извикването на всичките тези заявки, като се групират в един обект, който да се извика веднъж, само в една заявка[9]. Комуникацията между потребител и сървър става чрез обект наречен Value Object, който съдържа всички необходими данни[10].



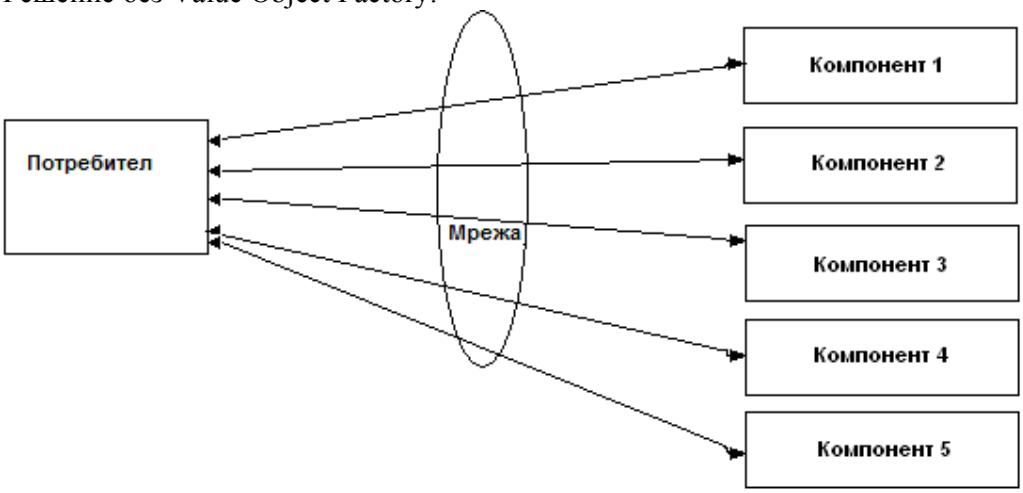
Фиг. 18 - Шаблона Value Object

Този шаблон (Фиг. 18) изглежда прост и тривиален, но има своето важно значение за бързодействието на една система и не трябва да се омаловажава.

1.3.4. Шаблона Value Object Factory

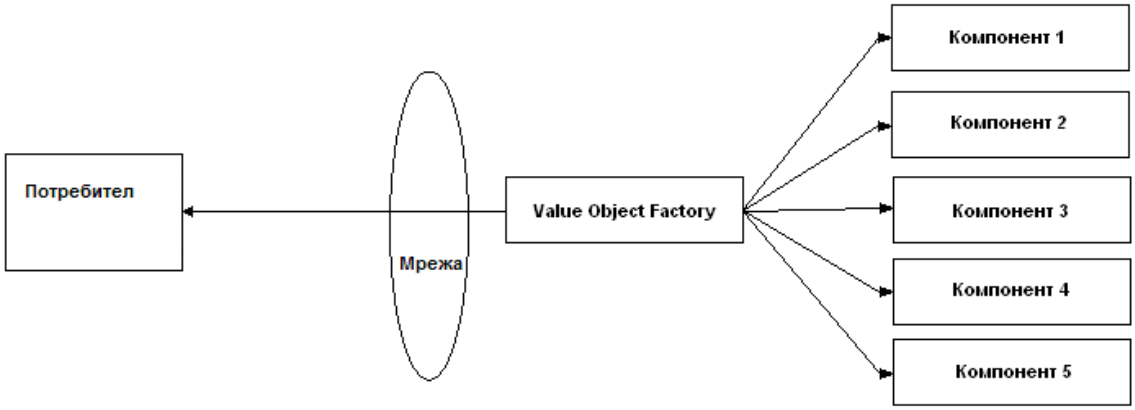
Подобно на горния проблем може да се случи така, че един потребител да желае да получи информация, която е разбита на много части и изисква множество заявки към сървъра. Но сега, всяка една заявка е насочена към различен компонент от уеб системата. За да се намали мрежовия трафик е нужно да се намалят броя потребителски заявки до една. Решението е възможно с реализация на шаблона Value Object Factory фиг. 20 [23]. Той има грижата да създаде нужните Value Object за достъп до съответния уеб компонент.

Решение без Value Object Factory:



Фиг. 19 – Пример без Value Object Factory

Решение с Value Object Factory:

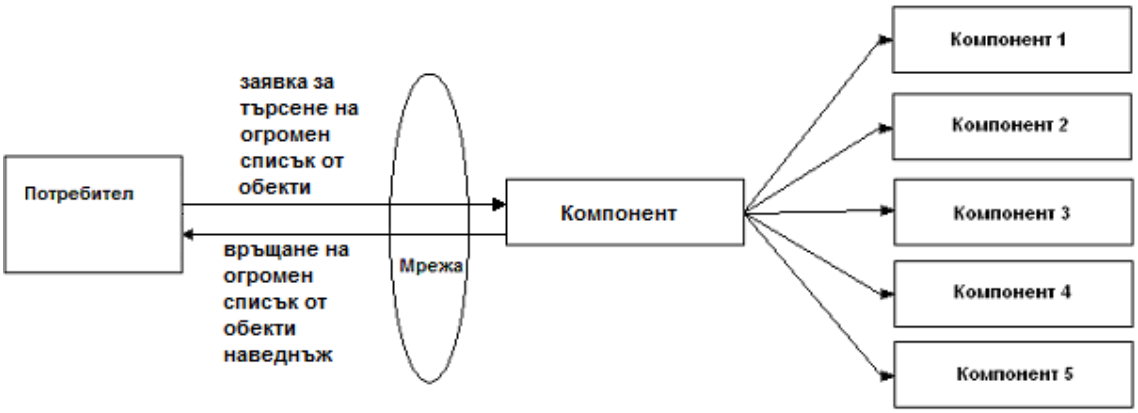


Фиг. 20 – Пример с Value Object Factory

1.3.5. Шаблона Value List Handler

Нека да разгледаме отново горния случай, при който потребителят чака информация, която е раздробена на много съставни части и в общият случай изисква много заявки към сървъра. Решението на проблема - намаляване на броя на извикванията, с което се намалява мрежовото натоварване и се пестят ресурси на сървъра. Постига се като краен резултат повишаване на бързодействието на цялото приложение. Но дали това решение (фиг.21) е приложимо, когато е налице подобен проблем и данните не са в малки размери, а в огромни. На фона на многото клиенти, наред с търсенето на огромни списъци, след това подготвянето им във вид, подходящ за визуализиране и след това изпращането им обратно към потребителя, е тежък процес.

Традиционно решение с намаляване на броя на отделните заявки към сървъра.



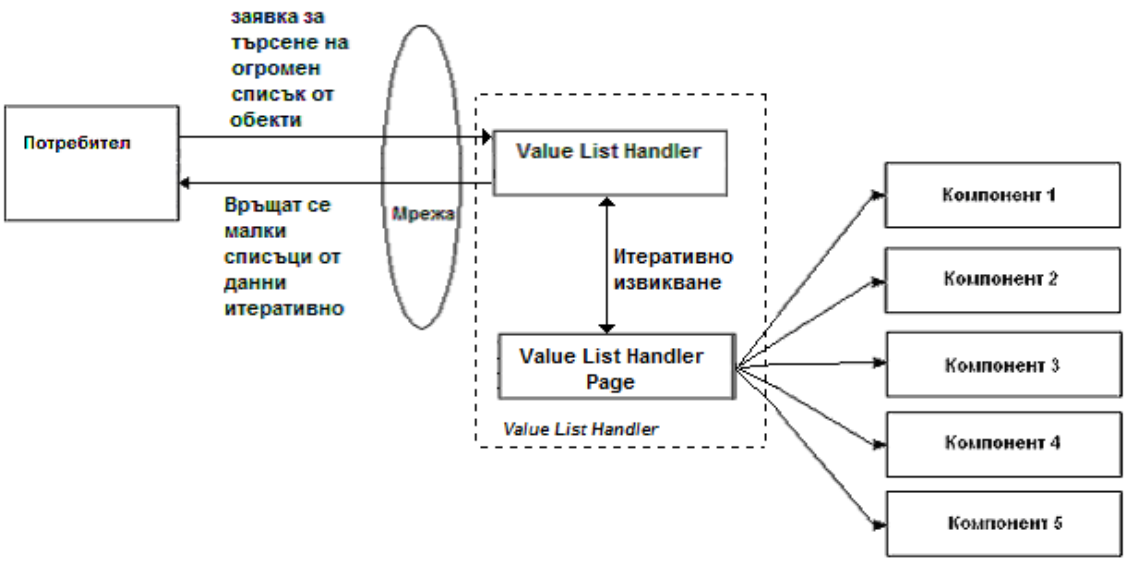
Фиг. 21 - Традиционно решение

Има и още една сериозна слабост в този метод, но сега тя е от страна на клиента. Да се предположи, че заявката е удовлетворена от сървъра и е необходимо клиентът да я визуализира. Най-вероятно браузерът на потребителя ще се затрудни и ще се забави при представянето на огромния списък от обекти наведнъж, при това без да се има впредвид какви са възможностите на клиентската машина и с каква скорост на интернет разполага.

Добро решение на целия този проблем дава шаблона Value List Handler. Компонента към, който клиентът изпраща заявките, кешира част или цялата информация, взета от останалите компоненти на сървъра и я препраща итеративно към потребителя[17]. Механизмът, който решава какво да се кешира и как да се кешира зависи от данните, ресурсите, с които разполага машината / машините на сървъра, както от личните предпочитания на разработчиците и други фактори. Има една друга важна част на този подход - не се препраща цялата информация към потребителят наведнъж, информацията му се препраща итеративно.

При реализация, използваща шаблона Value List Handler (фиг.22), системата ще изразходва много оптимално своите ресури[14]. Една от най-важните характеристики на този подход е, че няма да се налага на клиентската част от системата (браузер, PDA, ...) да генерира огромни html страници и да визуализира огромни списъци от данни.

Разбира се и този шаблон има известни недостатъци. Когато данните са динамични, съществува риск да се кешират остарели данни или да се предоставят на потребителя информация, която не е истинска. Затова е добре да се помисли за надеждна синхронизация с базата от данни при интерпретация на този шаблон. Друг недостатък е, че до известна степен може да припокрие кеширането от страна на базата от данни. “Добрите” бази от данни имат механизъм на кеширане, които евентуално може да се припокрие с кеша на приложението. Но в среда от изключително много и разнобразни потребителски заявки този проблем до известна степен избледнява и може да не се вземе предвид.



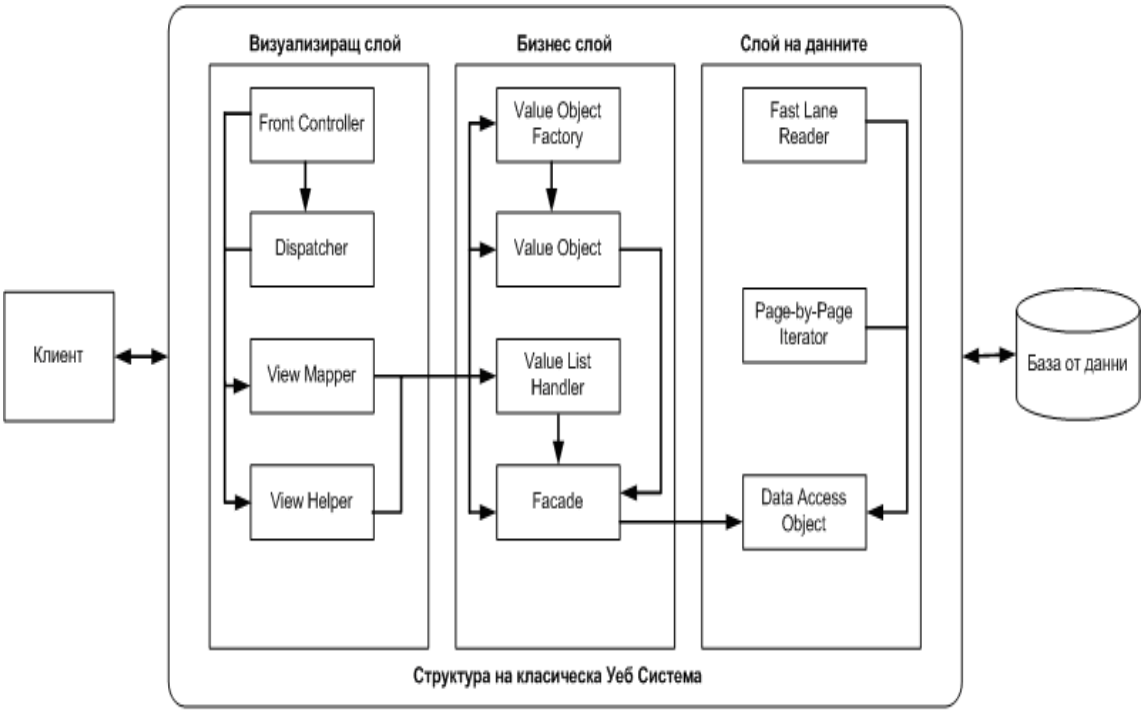
Фиг. 22 - Шаблона Value List Handler

1.3.6. Извод

Представените шаблони за повишаване на бързодействието, предназначени за системи, боравещи с големи масиви от данни по принцип се обединяват от постигането на две основни цели - намаляване на броя заявки към сървъра и намаляване обема, пренасяни данни. Всеки един от тях постига тези цели по различен начин и всеки един от тях в определена ситуация е за предпочитане пред другите. Така, че разработчиците преди да изберат някой шаблон, най-добре е да изследват в детайли изискванията към системата, както и обкръжаващите я фактори (характеристиките на мрежата, на сървърите, на данните, на потребителите и много други).

1.4. Заключение

Ако трябва да се представи глобално къде по принцип се разполага всеки един от представените шаблони в едно уеб приложение, как комуникира с останалите и евентуално дали може да се създаде решение комбинация от няколко шаблона, което е много популярна практика, фиг. 23 ще даде добра представа за това[21]:



Фиг. 23 - Шаблони за проектиране в структурата на класическата уеб система

Шаблоните за проектиране помагат на разработчиците да постигнат функционална и добре структурирана архитектура, максимално приспособима при появата на евентуални нови изисквания към системата[24]. Представените шаблони в тази глава характеризират проблеми, с които може да се сблъска един разработчик при създаването на приложение, боравещо с големи масиви от данни. Някои от решенията биха могли да се реализират при изграждането на скелета на цялата система, други за решаване само на специфични проблеми. Но благодарение на тях системата може по-лесно да се справи с изискванията по извличането, обработката и представянето на данните.

Глава 2

Техники за обработка на голямо количество данни

Една от основните цели на софтуер, боравещ с огромно количество данни е възможността най-ефективно да се трансферират данните от някакъв носител, където се съхраняват (например база от данни) до клиента. Основният проблем изглежда сравнително ясен – трябва да се покажат голямо количество данни без да се изисква потребителя да чака прекалено дълго време за да се появят. Впоследствие се появяват и други като - как да се проектира потребителския интерфейс така, че да бъде удобен, полезен и лесно разбираем. За да се анализира и разреши този проблем, глобално може да се раздели на две – „*Техники за проектиране на потребителският интерфейс на приложения, боравещи с голямо количество данни*" и „*Техники за проектиране на процесите на сървъра*".

Проектирането на потребителския интерфейс на приложение, боравещо с огромен обем от данни е изключително важна задача. Подценяването му може да струва много скъпо. Дори да се предположи, че другите екипи/хора, свързани пряко с разработката на приложението, но в друга сфера (с разработката на базата от данни или разработката на самия application сървър и т.н.) са свършили перфектно своята работа, то слабият и недобре обмислен дизайн на потребителския интерфейс може да провали целия проект.

За по-голяма простота и яснота, повечето примери ще представят с уеб система за продажба на книги (избран е тип приложение с по-опростена бизнес логика, която е по-лесна за разбиране и е подходяща за примери).

Тази система представлява класическо уеб приложение включващо следните слоеве:

- Слой за визуализиране на данните.
- Слой за дефиниране на бизнес логиката.
- Слой за обработка на същинските данни, с които работи системата.

Системата притежава огромна база от данни, с която е препоръчително да се борави „внимателно", за да може информацията най-ефективно да премине през всичките слоеве и оттам да достигне до клиента (заявка от "full scan" на таблица с милиони/милярди редове ще претовари всичките нива на това приложение).

Нека също така приложението да има форма за търсене на книги, която ще бъде обект на анализ в тази глава. Проектантските решения, които ще се предлагат, ще бъдат препоръчителни точно за този модул на системата. Още повече, че слабостите в дизайна на потребителския му интерфейс най-добре биха изпъкнали тук, т.к. това ще е мястото, където основно ще се борави с огромно количество данни. Съдейки от световния опит, свързан с тези системи, именно точно тази част от тях е с най-сериозна посещаемост от потребителите и генерираща най-много заявки към сървъра.

2.1. Техники за проектиране на потребителският интерфейс.

Едни от важните елементи, характеризиращи потребителския интерфейс са различните типове филтри и сортиране на резултата. С тях може да започне този анализ.

2.1.1. Проектиране на филтри.

Поради естеството на приложението, филтър за различните категории книги, за цената, издателството, автора, дата на издаване, биха били на пръв поглед полезни за ограничаване на резултата от търсенето. С тяхна помощ потребителите по-лесно ще се ориентират в резултата. Но филтрите не само ограничават размера на върнатия резултат от търсенето, но и намаляват общият брой на заявки към сървъра. Те спомагат сървърът да изразходва по-малко ресурси за да удоволетвори заявките към него. От една страна това ще ограничи търсенето в базата от данни т.к. заявките ще бъдат ограничени с поне един параметър (най-вероятно тези параметри ще отговарят на колони от таблици и ще са индексирани), а от друга страна ще намали както се спомена по-горе общият брой на заявките. Много от потребители имат ясна цел за това, което търсят и с помощта на филтрите биха ограничили резултата от 10 000 - 20 000 до дори 10 – 20, което най-малко би им спестило много време. Но целта на приложението е не само да достави информацията, а как и по какъв начин да я доставим най-ефективно на клиента така, че той да остане максимално удоволетворен и да не се налага да чака дълго време.

Винаги ли използването на филтри помага на потребителя?

Не винаги. Когато се предостави голям набор от филтри, натрупани из целият екран, това може да доведе до безпорядък и да обърка потребителя. Най-малкото, което може да последва е този сайт да има слаба посещаемост само от факта, че е объркващ за потребителите. Затова добрият дизайн на филтрите е абсолютно задължителен, но за да се постигне трябва добре да се познава в дълбочина психологията на потребителя.

Има различни подходи / методи за изследване на потребителите. Все повече се използват софтуерни продукти, имащи за цел да предоставят детайлен портрет на потребителите на едно приложение – пример за това е идеята за „persona”, въведена от Алън Купър във „Face 2.0”, която е мощен инструмент за моделиране на потребителското поведение, основавайки се на моделирането на въображаеми потребители. Тези виртуални потребители се генерират въз основа на поведението и желанията на истински такива, така че спокойно може да се вземе впредвид тяхното „мнение”. Този модел много помага за разбирането на целите на потребителя и изследване на поведението му в този специфичен контекст (потребителския интерфейс на приложението). По този начин той може да се проектира така, че да е възможно най-полезен за най-важните си потребители. Но не е задължително да се използва винаги софтуер от този тип за да се моделира какви са желанията на потребителя, безценни си остават и традиционните техники с тестовете с истински потребители по време на самият процес на проектиране. Тук може да се използва статичен HTML, като дори и само тестове на лист хартия. Спорен е въпросът, кой подход да се избере при моделиране на желанията на потребителя, дали по-скъпо струващия, основан на някакъв софтуер или по-евтиния, основан на хартиени прототипи. И двата подхода си имат плюсове и минуси, и поради тази причина може би най-добре е да се ползват и

двата в комбинация, за да се осигури най-ефективния начин, така че крайният продукт да бъде успешен и да се хареса на неговите потребители.

Ясно е, че проектирането на филтрите съвсем не е лека задача, но има и добри новини, в повечето случаи те са лесни за създаване и модифициране.

2.1.1.1. Drop-down филтър.

Една популярна имплементация на филтър (нека за простота, яснота, а и поради масовата използваемост – да се разгледа попълнен с категории от книги) е drop-down списъка. Тази имплементация е много удобна когато е необходимо да се покажат много категории на екарана, а мястото е твърде ограничено. Обаче трябва внимателно да се обмислят стойностите, които ще се съдържат в този списък. Много често, такива контроли съдържат страшно много стойности и разучаването на всяка от тях коства малко или много усилия от страна на потребителите. В такъв случай те вместо да разучават отделните категории, предпочитат да browse-ват из приложението докато намерят това, което ги интересува. Което не е за предпочитане т.к. от една страна се губи времето на потребителя в търсене (намалява се неговата удоволетвореност от сайта), а от друга самото browse-ване увеличава броя на заявките, изпратени към application сървъра, както и към базата от данни (най-често такава е структурата на едно уеб приложение клиент – application сървър – базата от данни) и / или към някой друг слой. Все пак, ако данните, с които боравим са с не голям обем, това не би трябвало да е особено притеснително. Но по съвсем различен начин изглежда ситуацията, когато те са много и удоволетворяването на само една от заявките на потребителя не е безобидна операция (може да търси из милиони/миларди редове). Например, потребителят търси книга за подарък и във формата му за търсене има drop-down филтер с 50 категории, и т.к. на него не му се губи време да разучи точно коя е неговата категория (т.к. те са представени прекалено тясно специализирано), той най-вероятно в първата итерация на своето търсене, изпълнява заявка без да ползва този филтър, с което на първо място прави „full scan” на базата от данни (сериозна операция, основна наша задача е да я избягваме). Тази заявка връща огромен резултат, натоварва мрежата или машината за да се предаде резултата към следващия модул/слой в процеса, той от своя страна изразходва ресурси за да подготви резултата в подходящ вид за да го препрати към клиента. Ако имаме няколко слойна архитектура – всеки един от слоевете би се ангажирал по някакъв начин с тези данни. Най-лошото е, че върнатият резултат към клиента, който най-вероятно ще бъде със стотици, хиляди обекти (в този случай обекти книги) ще изглежда като един огромен списък, който ще е труден за обхождане и много е вероятно той да не бъде прегледан въобще или да се прегледа една много малка част от него. Потребителят просто ще смени заявката с някоя друга за да може да намали броя на обектите, които трябва да прегледа.

Ключ към успешен филтър с категории е вместо да се използват тясно специфицирани категории и да се избира измежду дълги набори от стойности, да се използват по-общо генериращи, по-малък списък от стойности, които да не натоварват потребителя.

Обикновено категориите, съдържащи повече обекти съдържат съответно повече от търсените желани резултати за потребителя, в такъв случай е добре те да стоят най-отгоре в drop-down списъка. Ако има възможност е добре след името на всяка категория да се изписва колко обекта притежава тя. Така за потребителя ще е ясно защо

са подредени така обектите в списъка, а и ще е по-ясно за него, избирайки даден филтър какъв обем от данни ще получи впоследствие като резултат.

Пример за попълване на обектите в drop-down списък може да стане чрез изпълнението на една проста SQL заявка. Да приемем, че имената на категориите ни се намират в таблицата „**categories”**, а данните с книгите в таблицата „**books”**. За връзка между двете таблица има друга таблица “**book_category”**(релацията между тях е много към много)**.**

```
select c.name as category_name,
       count(*) as number_of_books
from   books b,
       categories c,
       book_category bc
where  bc.book_id = b.id and
       bc.category_id = c.id
group by c.name
order by 2 desc
```

Важно е, особено когато приложението има възможност да върне огромен резултат от данни, да има достатъчно на брой филтри, с които да се направи нужното сечение на данните. За да може да се търси по-ограничено върху даден обем данни, по-бързо да се върне резултата към потребителя и съответно да се изразходват по-малко машинни ресурси в изпълнението на заявката. Поради това е много важно заявките за търсене да бъдат представени колкото се може по-точно и по-правилно.

Добра методология е, когато имаме филтри, които се попълват само въз основа на действие от друг филтър (избор на произволна стойност от единия филтър попълва стойности в друг филтър). Улеснява се потребителя, интерфейсът става по-ясен и по-приятелски и се намалява възможността да се сгреши при дефинирането на заявки за търсене.

Например, ако потребителят е избрал от филтъра с категории – „Готварски книги”, да му се презареди формата в частта на съседен филтер с данни за диапазон от цени (тези цени ще са пряко свързани в зависимост от избраната стойност или в този случай цени за „Готварски книги”). Реализацията е проста - след като е избран от филтъра с категории стойност „Готварски книги” се изпраща заявка към сървъра с даденото **id,** представящо тази категория – да върне като резултат какъв диапазон от цени притежава.

Този вариант е много по-добър, ако предварително е зареден филтъра с цени, най-малкото, защото има възможност потребителят да избере стойност, за която няма данни. Добре е също вторият филтър по подразбиране да е неактивен и чак когато е избрана стойност за категория тогава да става активен – интерфейса става по-лесен, ясен и се намалява възможността от грешки.

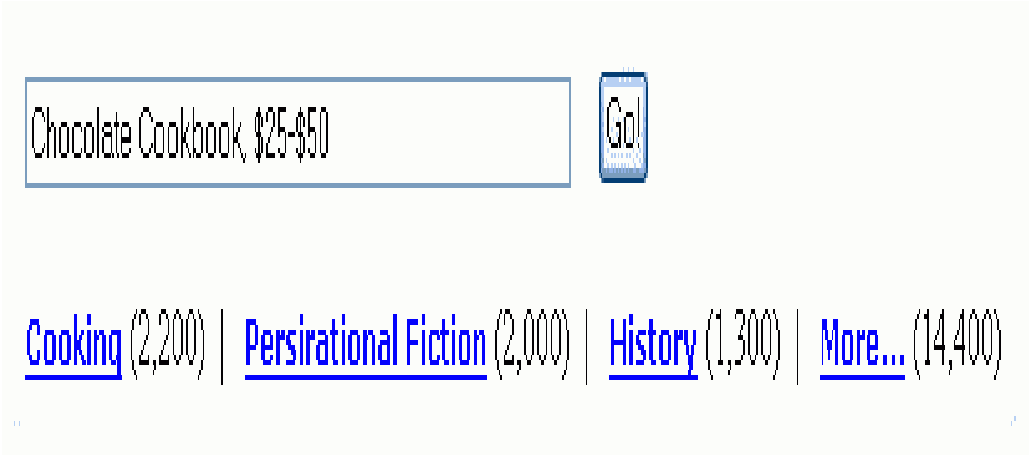
SQL пример за извличането на диапазон от цени на предварително избрана категория:

```
select get_price_range( b.price ) as price
from   books b,
       book_category bc
where  b.id = bc.book_id and
       bc.category_id = 17
```

get_price_range(price number) е функция в базата от данни, чрез която се предифинират диапазоните от цени.
17 – примерно id на категорията „Готварски книги”.

2.1.1.2. Филтер от тип линкове.

Както беше споменато по-горе, концепцията на филтрите е чужда за повечето потребители. Ако има на разположение място, където да се визуализират допълнителни контроли, истинската алтернатива на drop-down контролите е например категория от линкове Фиг. 24 .



Фиг. 24 - Филтер от тип линкове

Кликването върху един от тези линкове, ще има същия ефект, както избирането на дадена категория от drop-down списъка и натискане на бутон Go. Обаче линковете имат много преимущества в сравнение с drop-down контролите[8]:

1. Те са по-лесно забележими и по-интуитивни за ползване.
2. Те са всеобщо прието средство в уеб навигацията.
3. Те не изискват допълнително кликване за да се види и / или избере стойност от drop-down контролата.
4. Потребителите се чувстват по-сигурни, работейки с линкове, защото ако изберат нещо грешно винаги могат да натиснат Back бутона на браузера.
5. Линковете с добър дизайн, улесняват ползването на приложението.

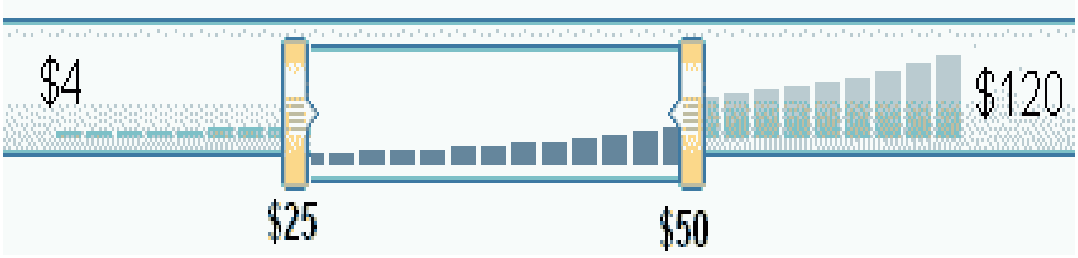
Най-добре е линковете да са разположени в близост до полето за търсене ако има такова или на места, където са по-лесно забележими за потребителя, вместо на някакви панели в крайно дясно или в ляво. Добре проектиран е потребителският интерфейс на фиг. 24, където полето за търсене и филтрите от линковете под него са обособени в една пространствена област. По този начин се подчертава, че имат една обща функционална връзка. Важно е, ако филтрите от линковете са повече от 4-5 да има

допълнителен линк „Още…”/”More…” и с клик върху него да се дава информация за останалите. Ако се наредят всички на екрана е много възможно потребителят да се объркa и най-вероятно ще заобиколи тази функционалност, а целта на приложението е точно обратната - стимулиране на употребата им. Както беше споменато по-горе, добре е до всяка категория да има в скоби информация колко обекта (в този случай книги) съдържа.

В този пример даден по-горе не се забелязва на пръв поглед голямото преимущество, което тези линковете дават, особено когато става въпрос за данни с голям обем. Нека да се разгледа достатъчно сложно приложение с някаква форма за търсене, форма за извличане на справка или нещо подобно, в случая общото е, че заявката ще върне изключително много данни и ще претърсва много данни. Нека генерирането на самата заявка не е много проста задача поради това, че бизнес логиката на сайта не е опростена. Тогава на преден план излиза изключителното преимущество на филтрите с линкове, при които най-популярните, сложни и често желани заявки ще са предварително съхранени / дефинирани, улеснявайки потребителя (ще се спестят кликания и избирания върху различни контроли за да се дефинира нужната заявка). Но може би най-голямото преимущество е за поведението на нашето приложение, т.к. тези заявки ще са оптимизирани докрай, а и с тяхна помощ се намалява общият брой заявки подавани към сървърите, давайки възможност да се ползват рационално машинните му ресурси.

В примера по-горе се вижда, че в полето за търсене, потребителят е задал и обхват от цени, този формат на задаване на заявка е специфичен и за да го използва потребителят трябва да има допълнителни знания. За тази цел е добре приложението да има кратки и прости инструкции, с които да се обучава потребителят как да ползва тези специфични формати (типичен пример е Google). Приложението трябва да има дефинирана функционалност за прихващане на грешки вследствие от неправилно ползване от потребителя. Въпреки това приложението се усложнява и предразполага към грешки. Голям недостатък е, че не може да се укажат допълнителни ограничения към потребителя – например както е в този случай, че цените варират от 0 до 180 $ например и всяка цена извън този обхват е грешно зададен параметър. Поради тази причина на помощ идва един друг тип контрола, която е много полезна в този случай:

2.1.1.3. Dual Slider филтер



Фиг. 25 - Dual Slider филтер

При Dual Slider (Фиг. 25) потребителите нямат нужда да четат инструкции какви формати да ползват за да укажат обхватът от цени (както е в този случай или най-общо казано интервал от стойности с минимална и максимална стойност), които са приемливи за тях. Само с помощта на мишката те лесно могат да укажат желаният от тях обхват[8]. Разбира се, излишно е да се споменава, че тази контрола е пряко свързана

с генерирането на заявката, както останалите филтри. Той може да притежава и друго преимущество – динамично да изчислява броя обекти за интервал от цени, което обаче може да се окаже нож с две остриета. Когато се ползва по този начин неговото динамично презареждане може да коства много машинни ресурси – все пак има голяма вероятност да направи „full scan" на една или повече таблици (както е в нашият случай, т.к. приложението ползва бази от данни) . "Full scan" при таблици, състояща се от милиони / милиарди редове е много тежка операция – както се споменава и по – горе е желателно да се избягва или да се ползва само в извънредни случаи. Още повече, че тази функционалност не дава особено големи преимущества на потребителя, както и не е ясно дали потребителят ще желае да я използва (много потребители не се интересуват от цената) и когато става въпрос за много данни силно се препоръчва да не се ползва този вариант.

Преимуществата на този тип контрола:

- Интуитивна е и е проста за използване.
- Придава добър изглед на приложението.
- Има способност да указва ограничения в произволно избран интервал от стойности.

Недостатъците:

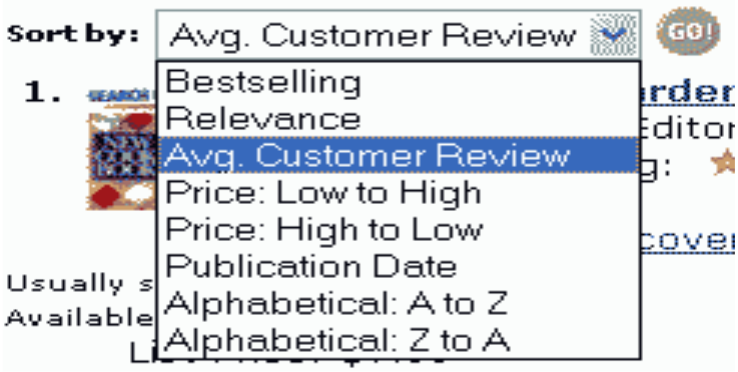
1. Все още не е включена в стандартният HTML, когато става въпрос за Web програмиране (но лесно може да се ползва например с Ajax).
2. Малко по-сложна е за имплементация от останалите типове контроли.

Все още не се среща много често, но ако се включи в стандартния HTML, определено ще придобие масова използваемост, поради изброените и качества по-горе.

2.1.2. Сортиране.

Сортирането е друг съществен инструмент, с който се позволява потребителят да манипулира с резултатните данни. За разлика от филтрирането, сортирането е термин, който повечето потребители разбират интуитивно и спокойно може да бъде надписана контролата с текст „Сортирай по**:".**

Има една добра практика за по-разбираем интерфейс на приложението. Контролите да са организирани като нормално изречение, например Фиг. 26 [8]:



Фиг. 26 - Пример от Amazon.com

Тази тип контроли обаче трябва да бъдат използвани с добре обмислена схема на сортиране, фокусирана върху специфичните цели на потребителя (които остават непроменени) вместо от задачите (които се променят – смяна на технологията, еволюция на потребителския интерфейс, ...). В разглеждания случай ще изглежда например така – търси се „Готварска книга” с най-висок рейтинг, която да струва от 0 до 10 лева. Т.е. заявката, която се подава към сървъра е основана върху категория, цена и „сортировка за най-висок рейтинг”.

Сортирането може да бъде осъществено като „clickable” заглавие на колона в таблицата от резултати. Пести се място на екрана, има добър изглед и е интуитивно за потребителя. Но ползването му особено когато става въпрос за много данни е специфично, опасно и трябва много да се внимава.

В този случай не е необходимо за всички полета да има сортиране. На някои от полетата не може успешно да им се зададе сортировка, на други е възможно, но няма да има голямо значение - по-добре да не се използва, особено когато става въпрос за много данни и всяка една последваща сортировка на вече показан резултат натоварва допълнително сървъра с излишни заявки.

Недостатък на този тип е, че не може да се нареди приоритета на сортировката за разлика от този с drop-down контролата, където по-важния тип сортировка е по-нагоре в списъка. Друг е, че сортировката е винаги свързана с данните в колоната от таблицата и е по-трудна за възприемане от потребителя за разлика от този с drop-down. Например много по-ясна и по-разбираема е сортировката „Най-евтините първо” пред един триъгълник обърнато нагоре или надолу (указващ в намаляващ или във възходящ ред) върху колоната с цена.

Още едно преимущество на drop-down контролата пред „clickable” заглавие на колона при търсене и визуализиране на много данни е в контролирането на сортировката. Изхождайки от психологията на потребителя, когато му се предостави възможността да сортира даден списък от данни в намаляващ или в нарастващ ред – голяма част биха се възползвали от тази възможност дори без да им е необходима (просто от любопитство). Например ако потребител търси книги от категория „Компютърна литература” – тази сфера е бързоразвиваща се и чувствителна по отношение на дата на публикация. Добре ще е да му се предостави само възможност да сортира в намаляващ ред по отношение на този признак. На пръв поглед това изглежда ненужно или поне ограничаващо функционалността на приложението. Но съвсем не е така, когато резултата се състои от няколко десетки, стотици хиляди реда и заявката за тяхното извличане е доста „машинно натоварваща”. Ако за удоволетворяването й трябва да се направи „full scan” или поне да се обходят голямо количество от данни на няколко огромни таблици в базата данни и резултата да премине през многослойната архитектура на нашето приложение за да се върне към потребителя. И ако очакванията са данните да са в голямо количество е по-добре да не се съхраняват в сесията или в паметта на сървъра, а да се извличат всеки път при заявка от клиента. То тогава една ненужна сортировка се явява напълно излишна, та дори вредна за приложението. Важно да се ограничи потребителя или казано по-правилно да му се предостави само възможността да сортира така както би му било полезно или би било полезно на голям процент от тях (например над 90%). Т.к. само заради прищявката да се види „от коя дата най-отдавна има на склад такава литература”, ще коства много ресурси на приложението, както и ще повлияе върху цялостното му поведение. В този случай

горещо се препоръчва да се подходи с drop-down контрола и да се предостави предварително дефинирана сортировка или само в нарастващ или само в намаляващ ред.

Обаче има случаи, в които ситуацията е доста по-сложна – например сортировката в нарастващ или в намаляващ ред определено е важна за приложението, но всяка една допълнителна сортировка изразходва допълнитлно много машинни ресурси. Например ако имаме една такава функционалност на приложението, при която потребителят изпълнява някакво търсене или някъкъв вид справка, която може да натовари машините на сървъра застрашително много и ако броя на потребителите, изпълняващи този вид заявка не е малък, а и времената на извикването им са близки – тогава определено имаме проблем. Този случай трудно може да се разреши с добавянето на допълнителен машинен ресурс. Решението определено е софтуерно и е много строго специфично – зависи от страшно много фактори (честота на извикване на заявката, степен на важност, време за изпълнение на заявката, търпението на потребителя, обем от данни и т.н.). Факт е, че в този случай много често се прибягва до лишаване на потребителя от каквато и да е сортировка, т.к. е много по-важно да има приложение, макар и с някаква по-ограничена функционалност, пред дилемата да няма никакво приложение или много бавно работещо такова.

Добра практика е дори когато не се зададе от потребителя начина на сортиране на данните, приложението да притежава сортиране по подразбиране, съобразен естествено с целите и желанията на потребителите му[29]. Важно е когато има такъв механизъм и след като се приложи да стане ясно, че данните са сортирани по даденият ред по подразбиране. Ще е проява на лош дизайн, ако това не е съобразено. Със сигурност ще се спестят много заявки само за да му стане ясно на потребителя, как са сортирани данните, а от гледна точка на леснота и разбираемост на приложението това е неоспоримо.

2.1.3. Заключение

Един важен аспект на успешния софтуер е способността му да се съобразява с количеството и характеристиките на данните, с начините и времето за обработката им за да се представят във вид най-удобен за потребителя. Тук е направен опит да се опишат трудностите, с които може да се сблъска софтуерният разработчик, когато проектира потребителски интерфейс на приложение с огромна база от данни. Предложени са начини за предпазване от неправилното взаимодействие с нея, което може да обезмисли цялата полза от приложението. И как се проектира потребителският интерфейс така, че да бъде максимално удобен и полезен за потребителя. Поради тази причина простотата и доброто взаимодействие между потребителя и софтуера са едни от най-важните характеристики на приложения от този тип. Нарастващата популярност на сайтове като Google, които имат подходящ набор от прости, но мощни инструменти, които са фокусирани върху желанията и опита на потребителите е ключ към дълги периоди на успех и забележителна преданост на потребителите. Няма смисъл от ефектни технологии, които ще рискуват продукта ако потребителският му интерфейс е труден за използване, претрупан, хаотичен, бавен или лошо проектиран. Например както беше споменато по-горе несъобразеното използване на филтрите, би създало безпорядък в интерфейса и би довело до объркване у потребителя. А когато сървърът трябва да обходи голямо количество от данни (базата му данни е огромна, например в порядъка на десетки / стотици GB) правилната и ясна дефиниция на неговия интерфейс

е от жизнено важно значение. Така се минимизират данните, които се обхождат и се пестят важни машинни ресурси. Ако един софтуерен разработчик не си зададе въпросите „Как ще успеят моите потребители да изпълнят своите целите” или „Как тази функционалност ще помогне на потребителите да постигнат по-бързо и по-лесно своите желания”, както и други от този характер, целящи да се вникне в потребителската психика, то неговия софтуер в много линии ще бъде безполезен, ще работи мудно и най-вероятно ще обърква потребителите му.

2.2. Техники за проектиране на процесите на сървъра.

В подглава 2.1 се разгледаха техники на добър дизайн на потребителския интерфейс. Спазването и следването им ще оптимизира клиентските заявки, изпратени към сървъра. Естествено продължение на тази тема е - *Техники на проектиране на процесите на сървъра при обработката на дълги списъци от данни*. Поради тази причина процесите могат да се разграничат логически на две части:

- Техники за проектиране на процесите при изпълнение и подготовката на данните за визуализация.
- Техники за проектиране на процесите при визуализация на резултата.

Изброените техники ще бъдат изложени по-долу.

2.2.1. Техники за проектиране на процесите при изпълнение и подготовката на данните за визуализация.

Съществуват няколко популярни техники, които се използват за решаване на този проблем:

Техника 1.

Изпълнение само на една заявка към слоя, управляващ данните (към базата от данни), извличане на целият резултат и кеширането му. Последващите заявки от потребителя се взимат от кеша.

Техника 2.

На всяка заявка от потребителя, се прави заявка и към носителя, където се съхраняват данните.

Техника 3.

Комбинация от първите две техники, заявките към слоя управляващ данните не са равни на броя клиентски заявки, частичният резултат пак се кешира. Част от клиентските заявки се удовлетворяват от кеша.

Правилният избор коя техника да се ползва в даден момент е важна задача. Изборът на грешна техника може да доведе до ниско бързодействие на приложението, та дори до неспособност да обработва подадените му заявки.

Всяка една от тях ще се разгледа подробно, ще се отбележат плюсовете и минусите и кога е най-препоръчително да се използва.

Навсякъде в представянето им ще се приема, че носителят за съхранение на данните е релационна база от данни (почти 100% от случайте, когато става въпрос за огромно количество данни, носителят е такъв).

2.2.1.1. Техника 1:

Изпълнение само на една заявка към слоя управляващ данните (към базата от данни), извличане на целият резултат и кеширането му. Последващите заявки от потребителя се взимат от кеша.

Тази техника има ефективна стратегия, в отношение на минимизиране на заявките към базата от данни. На другата крайност е по отношение на използваната оперативна памет. За да е напълно ясен казуса, ще се дадат примери с детайлно изчисление на използваните данни. Примерът ще е различен от досега даваните примери със сайта за книги. Нека се предположи, че потребителите в отговор на своите заявки към сървъра получават списък със средно около 50 000 реда като всеки един ред е с размер от около 1 KB. В такъв случай 50 000 * 1 KB = 50 000 KB ≈ 50 MB. Да вземем впредвид, че конкурентните потребители, боравещи с тази заявка са максимум 20 (смешно малка цифра, но е взета за простота на изчисленията). Тогава размерът на данните, които ще се съхраняват в даден момент в паметта ще достигне 50 MB * 20 ≈ 1 GB. Цифра, която съвсем не е малка. Но една / няколко сървърни машини с достатъчно добри параметри биха се справили без проблем. Стига да няма перспективи да се увеличава средният брой редове или броя потребители. Но в реалността / в практиката процентът на приложенията можещи да гарантират това е малък. За да се представи по-добре техниката, нека да предположим, че размерът данните в паметта много трудно може да превиши 1GB. Какво би накарало екипа, проектиращ приложението да избере точно тази техника, при условие, че изразходва толкова много ресурси? Нека си преставим следната ситуация, имаме една или няколко специфични заявки към базата от данни с изключително много време на изпълнение и машинно натоварване. Дължащо се на това, че данните, които се извличат са разположени в голям брой таблици и всяка една от тях е с огромен размер. В такъв случай една join заявка би била изключително тежка. Тя се изпълнява с цената на много време, солиден I/O, процесорно натоварване и изразходване на много оперативна памет. Въз основа на тези цифри като прибавим и големият брой конкурентни потребители, които я изпълняват, ситуацията става сложна. Естествено в такъв случай тази техника намаляваща до минимум комуникацията с базата от данни и изглежда подходяща на първо четене. Заявката или заявките се изпълняват веднъж, разполагат се в паметта и оттам за кратко време се предоставят на потребителите.

Всичко изглежда добре, но когато се подхожда към тази техника трябва да се отбележи един важен аспект. Честота на опресняване на данните. Колкото по-често данните трябва да бъдат опреснявани в кеша, толкова по-малко подходяща е тази техника. Ако данните са изключително динамични, то тогава може да се каже, че е напълно неподходяща да я използваме поради факта, че във всеки един момент данните в кеша и в базата данни може да са несъгласувани. Но въпреки това, съществуват приложения, за които малка промяна в данните не е фатална. Ако се подходи по този начин, задължително потребителят трябва да бъде предупреден за евентуална несъвместимост и до каква степен може да бъде тя. Добре е предупреждението да бъде разположено така, че да бъде задължително забелязано (разбира се, че може да бъде сложено навсякъде, но е по-добре за потребителя). Например след натискане на бутон „Изпълни / Go” на заявката към сървъра. Тогава може да се появява малко pop-up

прозорче, което да предупреждава или друг вариант - докато тече изпълнението на заявката да се вижда предупреждението за несъгласуваност на мястото, където ще се визуализира резултата.

Към всичко отбелязано досега за тази техника, трябва да се отбележи още обхвата на видимост на кешираните данни. Може да бъде заложен на ниво сесия или на ниво приложение. В първият случай за всяка отделна сесия ще се изпълнява по веднъж заявка към базата от данни и резултата ще се кешира, като видимостта ще бъде само за текущия потребител. Във втория случай заявката към базата от данни отново ще се изпълни веднъж, но този път веднъж за цялото приложение. Резултатът също ще се кешира, но видимостта ще бъде видима за всички потребители на приложението, имащи съответните права, независимо от техните сесии. Както и да се разсъждава не трябва да се забравя факта, че приложението борави с изключително много данни и едва ли всичките те биха могли да се заредят в паметта и оттам да се препращат към клиента. В такъв случай каква би била ползата от базата от данни. Но все пак за да се представи докрай техниката ще се продължи нататък с разсъжденията. И така след като веднъж са заредени данните в кеша, дали със средства, налични от самия Web сървър или по друг начин, е препоръчително да има механизъм, определящ времето им на активност. Например при изтичане на известен период от време - кешът да бъде освободен и да се презареди отново автоматично или да се презареди чак при изпращане на нова заявка от случаен потребител. Ако се използва втория вариант, трябва да се има впредвид, че времето на изпълнението ще трае по-дълго отколкото се очаква, поради факта, че заявката трябва да се изпълни към базата от данни и след това резултатът да се зареди в кеша, за разлика от друг път, когато се взима наготово от кеша.

Важен аспект на тази техника е още - как ще се кешират данните, разпределени по различните типове заявки или по друг начин. Изчисленията в първият параграф от текущата точка бяха направени спрямо една единствена тип заявка т.е. кешираните данни се отнасят само за нея. Ако се подходи по този начин, ако броят на типовете заявки е голям и размерът на данните за всяка една е все така огромен (около 1 GB на заявка), то е ясно, че не препоръчително да се използва тази техника. В този случай/при тези обстоятелства риска машината на сървъра да остане без свободни ресурси е напълно реален.

Окончателното заключение е, че тази техника не е препоръчително да се използва при приложения, боравещи с много данни, имащи много конкурентни потребители, по простата причина, че има голям риск ресурсите бързо да свършат. Направените изчисления, показваха разход на 1GB памет, но те бяха направени върху малък на брой данни и смешно малък брой потребители (20), все пак анализите са насочени към система с голям брой конкурентни потребители, където цифра от 200 или 300 е нищо в сравнение с реалността. Но дори тогава разхода на памет вече става много голямо число 10 GB и то само за потребителски данни, без да се изчислява паметта, заета от класовете и машината, с които тези процеси не биха били възможни.

2.2.1.2. Техника 2:

На всяка заявка от потребителя, се прави заявка и към носителя, където се съхраняват данните.

Тази техника по отношение на обвързаността си към базата данни е коренно различна от предишната. Тук на всяка заявка на потребителя се прави заявка към базата и целият резултат се предава към клиента.

Кога е най-препоръчително да се ползва тази техника?

Изключително ефективна е, когато заявките към базата са бързи и ненатоварващи. Съмнение за приложението й буди обратната ситуация, когато времето за изпълнение е дълго. А ако се вземе в предвид и голям брой едновременни клиентски заявки – определено трябва да се замислим за нейната ефективност. Други въпроси относно вземането на решение дали да се ползва или не са следните:

- **До каква степен са способни потребителите да чакат резултат от своята заявка.**

Колкото по-търпими са потребителите към времето за отговор толкова по-приемлива е тази техника.

- **До каква степен са динамични данните в базата от данни или казано по друг начин колко често се опресняват те.**

Колкото по-динамични са данните, толкова по-подходяща е тази техника. Не може дори да става въпрос за някакъв вид кеширане, ако данните в базата от данни се променят постоянно.

На теория следващите случаи не би трябвало да влияят върху решението за ползването на тази техника, но на практика се случва да се взимат предвид:

- **Способна ли е машината на базата от данни да приеме и обработи голям брой тежки конкурентни заявки.**

Когато натоварването към базата от данни е особено голямо - голям брой тежки заявки - с висок I/O и с голям обем на резултата, и машината (машините) не е способна да ги изпълни в приемлив срок [6]. В този случай за решението за използване на тази техника, определено ще трябва да се върви заедно със задаването и обсъждането и на друг тип въпроси, които не са тема на този труд.

- **Способна ли е самата база от данни да приеме и обработи голям брой тежки конкурентни заявки.**

Тук проблемът отчасти много наподобява горния. Може само да се каже, че когато данните, с които разполагаме са в голямо количество и с голям брой конкурентни заявки е от изключителна важност какъв производител на бази данни ще се избере. Слаба такава реализация (например някоя безплатна версия), наистина може да наклони везните в ползването на друга техника, определено ще намали цената, но и ще повлияе негативно върху приложението.

От подобен род са и разсъжденията и върху останалите два показателя:

- **Способна ли е машината на сървъра на приложението (application server) да обработи голям брой потребителски заявки.**
- **Способна ли е мрежата да поеме голямото натоварване.**

Решението за използването на тази техника, следвайки установените по-горе принципи може да наклони везните в друга посока, което може да се окаже грешно решение. Затова е важно нещата да се погледнат и от друг ъгъл. Да не би някой от въпросите по-горе да са дължат на грешки или недоглеждане от страна на разработчиците на този софтуер. Например когато времето за изпълнение на една заявка е дълго. Могат да се направят редица изследвания особено в частта с базата от данни. Могат да се разгледат самите SQL заявки - как са съставени и до каква степен са оптимизирани. Лошо написана заявка може да доведе до напълно грешно впечатление. Също така, често срещана грешка е по времето на разработка на едно приложение и особено в тази част с SQL заявките - съставянето и тестовете им да се правят върху малко на брой тестови данни, например върху няколко хиляди реда, а реалният размер на данни на приложението например след 1 година експлоатация да бъдат десетки, стотици милиони или милиарди реда. И в крайна сметка заявките да са лошо написани. Друга грешка е тестовете на разработчиците е да се правят върху ненатоварена от към брой активни сесии база от данни или application сървър – техните тестове са спрямо малък брой, та дори често той е равен на броя на разработчиците на това приложение. Ако тогава не се вземе впредвид и не се обмисли как ще изглежда приложението след някакъв по-късен период на неговата експлоатация, когато данните могат да нараснат в голямо количество, а броя на конкурентните заявки да бъде огромен. Грешни изводи могат да произлязат и от лошо направен дизайн на базата от данни, на ключовете и на constraint-ите на таблиците, начините на индексиране. Грешно съставените индекси, наложени върху неправилна колона/и може да коства много от бързодействието. Добре проектираните индекси могат да ускорят в хиляди пъти времето за отговор.
Грешки стават обаче не само в базата от данни, може да ги има и в междинният слой - на application сървъра - там недобре написан код може също да доведе до заблуждение.
Поради тези причини - добре е да се претеглят и тези фактори, за да е сигурно, че е взето решение за правилната техника.
Част от тези въпроси се разглеждат в следващата глава от този труд.

2.2.1.3. Техника 3:

Комбинация от първите две техники, заявките към слоя, управляващ данните, не са равни на броя клиентски заявки и част от резултата е възможно да се кешира.

Тази техника много прилича на **Техника 2,** особено в първата си част - честотата на заявките към базата от данни. Поради това принципите, които се отнасят за нея важат в пълна сила и тук. Разликата между двете техники е в броя заявки към слоя, грижещ се за данните (базата от данни) и в начина на извличане на резултата от заявката.

Ако една част от данните са статични, ползват се често и не в огромни размери, те могат да се извлекат веднъж от базата от данни, да се кешират и след това да се предоставят на потребителя по най-бързият начин - директно от кеша. Това решение би повлияло положително върху бързодействието на приложението, но трябва да се вземе в предвид и разхода на памет.

Втората разлика - начина на извличане на данните (при някои приложения единствена, поради фактите, че данните не са статични или са много или няма свободни ресурси или други). Извлича се не целият резултат, а само определена част, останалите части се връща при евентуално последващо поискване от клиента.

В основата на този подход е залегнал принципа, че не е необходимо да се извлекат всички данни наведнъж, тук просто се предвижда, че данните се състоят от няколко части и се връща само първата/те от тях. Много често това се оказва голямо преимущество т.к. заявката се изпълнява върху много по-малък обем данни - което я прави по-бърза и консумираща по – малко ресурси. Много рядко, когато става въпрос за резултат, съставен от много редове, например 20 000, потребителят желае да разгледа всичките редове веднага. Най-често той се интересува от първите 100 , максимум 1000 (при добре заложена сортировка, обект на предишни точки от главата). А и в много случай човек е неспособен да разгледа повече от 1000 реда (разбира се това е малко субективно). Затова изискването на тези 20 000 на равни порции от базата данни е добър вариант, например на 20 порции по 1000 реда или може би 200 порции по 100, зависи от данните и бизнес логиката на приложението. Типичен пример са Web търсачките, те рядко предоставят наведнъж целият резултат - той дори в някои случаи се състои от милиони редове. При тях потребителят им се интересува предимно от първите десетки реда, останалите рядко представляват интерес за него.

Другите слоеве на приложението също печелят от това, че боравят с по-малък на брой данни. Не е все едно дали обработваш 1000 или 20 000 реда - разликата от 20 пъти е не само като брой редове, но и в разпределена памет, натовареност на процесора и в работата на мрежата.

Не само когато става въпрос за софтуер, манипулиращ с голямо количество от данни, каквато е темата тук, а почти винаги се правят оценки, коя е най-добрата техника за “изпълнение на заявките от потребителя и подготовката на резултата от тях за визуализация”. В повечето случаи голям обем информация изхвърля от употреба **Техника 1** и затова изборът често остава върху **Техника 2** или **Техника 3.**

Пестенето на ресурси и бързодействието на **Техника 3** е голямото и преимущество**.** За **Техника 2** остават случаите, когато на потребителят му трябват или е възможно да му потрябват всички данни на всяка негова заявка. Както и по трудната за реализация и поддръжка **Техника 3**. Но въпреки всичко тя е по-често използваната техника, защото разработчиците предпочитат да отделят малко повече време за разработка и поддръжка пред ситуацията да се окажат с приложение, което работи бавно и неефективно.

2.2.2. Техники за проектиране на процесите при визуализация на резултата.

Проблемът тук е сравнително ясен – “Кой е най-удачният начин да се представят визуално огромен размер данни”[1].

За да се анализира проблема по-добре и за по голяма простота на разсъжденията ще се даде следният пример – потребител изпраща заявка към приложение с огромна база от данни, приложението отговаря със списък от 20 000 реда.

В практиката предимно са известни два варианта за разрешение на този проблем – да се покаже целият резултат в един огромен списък на екрана наведнъж или резултата да се разбие равномерно и да се представи на отделни страници.

Когато се зададе въпроса - колко време и ресурси ще отнеме на едно приложение за да генерира страницата, показваща всичките данни наведнъж, става ясно, че първият вариант е изключително неефективен. Заради това предимно - разработчиците предпочитат варианта, в който резултатът от заявката се разпределя на отделни страници[2]. Разбира се тук разсъжденията изхождат от ситуацията, че данните са в голям размер, ако става въпрос за малко данни, ситуацията ще изглежда по друг начин.

Вариантът със страницирането е изключително подходящ и то не само заради отчитането на фактите за ускореното бързодействие и спестените ресурси. Тази техника е изключително удобна и поради причината, че потребителите лесно се ориентират в многото данни и лесно ги обхождат. Типичен пример за приложения, ползващи тази технология са така популярните интернет търсачки.

Интернет търсачките и не само те - съвсем закономерно ползват такъв тип техника. С нейна помощ се пестят машинни ресурси, данните са представени в по-представителен вид и имат по-приятелски изглед за потребителя. Но има още едно важно преимущество, което натежава везните за този избор - фактът, че потребителите рядко се интересуват от резултатите след 3-тата страница. Направените изследвания в тази насока наистина доказват това, голям процент от потребителите рядко преглеждат данните след 3-тата страница, а процентът на тези, разглеждащи по-висок номер страници е изключително малък. Което естествено навежда на идеята да не се извличат абсолютно всички данни, а само част от тях, което ще се разгледа малко по-надолу.

Какво би станало ако не се използваше техниката на странициране на резултатите, погледнато от гледната точка на обикновения потребител?

Щяха да се получат всички данни на една страница и сред големия списък потребителите да търсят това, което ги интересува, което определено е малко или много объркващо. Това е едната страна, другата страна е, че навярно потребителят ще се наложи да чака доста време преди неговият браузер успее да визуализира html страница съдържаща огромният списък, предаден от сървъра. Също така най-вероятно сървърът ще има доста излишна работа за да извлече и подготви цялото множество от данни към потребителя[27].

Техниката за разпределянето на резултата на страници може да се раздели:

- Явно.
- Скрито.
- Комбинирано.

Характеристиките на тези разпределения малко напомнят на Техники 1,2 и 3 по-горе, но това е нормално, т.к. те са пряко свързани помежду си.

2.2.2.1. Явно разпределение

Явното разпределение е, когато отделната заявка за дадена страница или списък от обекти се явява и отделна заявка към най-ниският слой (слоят, управляващ данните) на приложението. Данните, така както пристигат от носителя, така се препредават към потребителя.

2.2.2.2. Скритото разпределение

Скритото разпределение е, когато данните се извличат наведнъж от базата от данни и след това се обработват от бизнес слоя и слоя, презентиращ данните. Обработката се състои в разпределяне резултата на страници в подходящи размери и чак тогава се изпраща към потребителя.

Естествено тази техника е свързана с изразходване на много ресурси, отнасящо се към междинните слоеве, но се пестят до някаква степен ресурси отнасящи се към слоя, грижещ се за данните (базата от данни), най-малкото се намаляват броя заявки към него.

2.2.2.3 Комбинирано разпределение

Комбинирано разпределение е, както личи от самото му наименование - комбинация от явното и скритото. То е и най-използваният подход от трите. Същността му е, че не всяка заявка отправена от клиентите към сървъра се свързва със заявка към най-ниският слой (базата от данни). Това зависи от данните, някои данни могат да бъдат извлечени, обработени и кеширани, други не – както се извлекат така се предоставят на потребителя.

2.3. Заключение

Целите на гореизложените техники са основават на различни идеи - да се намалят до минимум заявките към базата от данни (едно от най-натоварените места в системата) или да се намали натоварването върху бизнес слоя, или да се избере хибриден вариант. Кой вариант или комбинация от варианти ще се ползва е важно решение и трябва сериозно да се обмисли. Целите са ясни - удовлетвореност на потребителя с постигане на задоволително бързодействие, но с пестене на ресурси. Благодарение това е по-малко вероятно да се достигне до момент, когато натоварването да достигне до такъв връх, че да няма възможност да се обработи нова клиентска заявка, а приложението ще продължи да работи както и преди.

Глава 3

Бързодействието, при системи представящи големи масиви от данни.

Когато едно приложение борави с огромно количество от данни, едни от най-сериозните въпроси / проблеми, върху които се фокусират неговите разработчици е бързодействието му. Нека вземем за пример една интернет търсачка, която връща отговор много бавно при зададено към нея запитване. Всичко друго може да е перфектно проектирано и реализирано – лесен, приятелски, ефектен, приятен и добре проектиран дизайн, но дългото чакане определено ще постави на изпитание търпението на потребителите и. Ниската производителност може да доведе до непопулярност и неизползваемост на приложението. Ясно е, че когато информацията е в огромен размер, процесите на извличане, съхраняване, промяна и представяне на данните на потребителя не са тривиални. Естествено е, че за системи с малък обем данни, фокусирането върху тези процеси въобще не е основна задача и цел – при тях спокойно може да се изберат методи, които се фокусират не върху това да работят бързо, а върху това да са лесни за реализиране и поддръжка. Но ситуацията изглежда по друг начин, когато данните са много, даже много често при тях бързодействието се явява една от най-важните им характеристики. Ако се вземе за пример Google, дизайнът му е прекрасен - лесен, добре обмислен, не е претрупан и доста интуитивен, от потребителя се изискват минимални усилия да напише това, което го интересува и да след това да проследи резултата от неговата заявка. Дотук добре, но как биха изглеждали нещата, ако той трябваше да чака в продължение на минути отговор и много често както се случва по време на търсене, в зависимост от получените данни, да си промени малко шаблона за търсене и да чака още няколко минути. Ясно е, че всичко друго, свързано с този софтуер отива на заден план, потребителят няма да се интересува много от външния изглед на страницата, колкото и добре да е обмислена тя, най-вероятно търпението ще му се изчерпи и ще потърси друга търсачка или друг начин да намери това, което го интересува.

Ето защо да се избегнат проблеми от този род за такъв тип софтуер е важно по време на изграждането му, разработващият го екип да се фокусира върху неговата производителност. Тук за постигането на тази цел, процесът ще се раздели на два дяла (31):

- стратегии за постигане на по-висока производителност, ориентирани към дизайна на софтуера на високо ниво
- тактики за постигане на по-висока производителност, които могат да се разглеждат като препоръки на ниско ниво – ниво на програмен код

3.1. Стратегии за подобряване на бързодействието

Тук в тази подглава ще се разгледат стратегиите за постигане на добро бързодействие по време на разработката на една система. Те ще се концентрират върху най-високите нива на една система. Те няма да касаят само софтуерния инжинер, а също мениджъра на проекта, техническия ръководител или специалистът, следящ за качеството на софтуера. Стратегиите ще помогнат за да се постигнат целите, които касаят производителността на системата.

Преди да се премине към самите стратегии е добре да се изясни въпроса:

3.1.1. Какво означава бързодействие?

Този въпрос звучи прекалено просто, но той притежава много различни аспекти, всеки от които може да бъде важен. Не на всеки е ясно кои са факторите, които влият на приложения, боравещи с много дании. Не е ясно на всеки, какво се има впредвид, когато става въпрос за бързодействие на едно такова приложение.

Най-общо под бързодействие ще се имат предвид следните аспекти:

- Скорост на изпълнение.
- Размера на RAM, която се използва по време на изпълнение.
- Времето за стартиране на приложението.
- Възможността на системата да се адаптира към повишена натовареност.
- Времето за отговор на отделна заявка от потребителя (при заявка за търсене, download на файл).

Всички тези фактори характеризират бързодействието като термин. Разбирането как всеки от тези фактори влияе на системата и прилагането им допринася много за повишаване на нейното по-добро качество и за постигане на успех като работеща система.

3.1.1.1. Скорост на изчисление.

Това е фактора, който най-често се има впредвид от повечето хора, когато стане въпрос за бързодействие. Кой алгоритъм ще се използва и как се имплементира са ключови фактори, които стоят в основата. Но те не са от основно значение за цялостното бързодействие. Много често алгоритмите имат подобна производителност и изборът на един или друг не влияе съществено. За да се постигне добро бързодействие на цялостно ниво, спазването на факторите по-долу играе съществена роля.

3.1.1.2. Размера на RAM, която се използва по време на изпълнение.

Размерът на RAM, която се използва по време на изпълнение може да бъде критично важен за цялостното бързодействие. Всички модерни операционни системи предоставят виртуална памет, където хард дискът може да бъдат използван при недостиг на RAM. Но най-бързият хард диск е далеч по-бавен от най-бавната памет. Приложения, които често изискват да се пише във виртуалната памет като цяло имат ниско бързодействие.

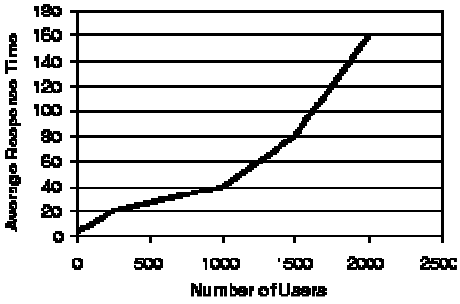
3.1.1.3. Времето на стартиране на приложението.

Времето за стартиране на едно приложение може да бъде критичен фактор. Например, ако една уеб страница първоначално се зарежда за няколко минути, повечето потребители никога няма да я изчакат, те със сигурност ще се насочат към някои други.

Специфична тактика за намаляване на времето за стартиране е контролирането на зареждането на класовете. Ако приложението е уеб базиран клиент-сървър, първото зареждане на една страница се свързва с компилирането и зареждането в паметта, което е бавна операция. Съществуват различни механизми да се избегне това, един от които е да се компилира повече и повече код, с цел намаляване на това време при поискване от клиента.

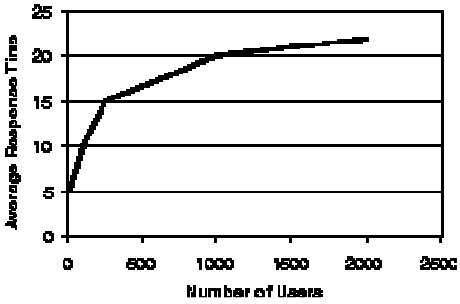
3.1.1.4. Възможността на системата да се адаптира към повишена натовареност.

Системата може да работи добре с 50 конкурентни потребителя, но как ще работи с 1000? Дали времето за отговор ще расте постепенно или ще скочи рязко нагоре след като е достигнат определен праг.



Фиг. 27 - Пример за лошо бързодействие

Графиката във Фиг. 27 дава пример за система с лошо бързодействие. Вижда се, че с увеличаване бройката на потребителите средното време за отговор нараства експоненциално.



Фиг. 28 - Пример за добро бързодействие

Графиката от Фиг. 28 изобразява система с добро бързодействие. И тук с увеличаване на броя потребители нараства времето за отговор, но това време расте постепенно.

Интересен факт е, че и двете системи показват приблизително едно и също бързодействие при малък брой потребители (до 250). Единствено при голям брой потребители те показват истинските си характеристики.

Разработчиците или хората, отговарящи за система, работеща с огромен размер от данни трябва задължително да изследват такива случай. Т.к. при тях само един потребител може да изразходи достатъчно много ресурси. Трябва да се направи анализ какъв е най-големият брой конкурентни заявки към системата и да се проследи държанието в един такъв пик.

3.1.1.5. Времето за отговор на отделната заявка от потребителя (при заявка за търсене, download на файл).

Времето за отговор на отделната заявка от потребителя в много случай е най-важния елемент от бързодействието на една система. Много често крайните потребители не правят точна оценка на този елемент. Например сравнението на това време между различните приложения макар и с една и съща архитектура е много грешен подход. Резултатът често изопачава истината, нещата може да стоят по съвсем друг начин, а системата може да е разработена професионално и да няма слаби страни. Но когато тя борави с терабайти информация, а другата, с която е сравнявана борави с много по-малко, естествено е сравнението да е погрешно. Поради това е много важно да се запознае крайният потребител с тези факти, за да не си прави грешни оценки.

След като се изясни въпроса около термина бързодействие (производителност), фокуса ще се прехвърли върху процеса на създаване на софтуер, преминавайки през всичките му фази и наблягайки на стратегиие за постигане на добро бързодействие.

3.1.2. Фактори влияещи върху процеса на бързодействие

Нека се разгледа системата, дадена за пример по-горе или накратко, опериращa с много данни. Задачата да се настрои една система да работи бързо не е лека. Бързодействието й е от изключително важно значение за успеха й, но не трябва да се гледа на него като на отделна стъпка от плана на разработка, а е нужно да бъде неизменна част от всеки един от етапите.

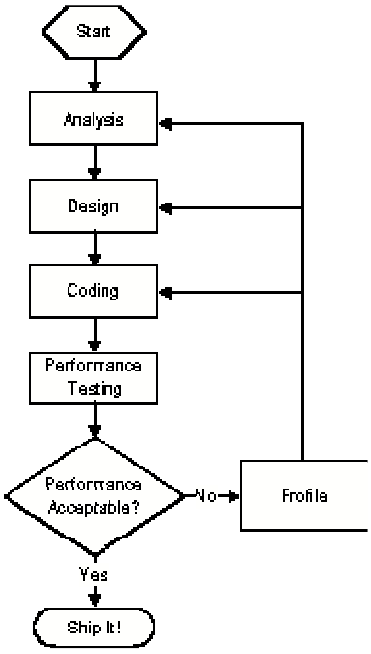
В тази подточка ще се представи процеса за постигане на добро бързодействие.

Постигането на добро бързодействие на такава система е резултат, дължащ се най-вече на солиден анализ и дизайн. Ако не се анализира добре проблема е почти невъзможно да се направи добър софтуерен дизайн. А без добър дизайн не може да се очаква добра производителност, по принцип е невъзможно е едно приложение само да се настрой да работи добре.

В основата на стандартният процес на обектно – ориентиран дизайн стоят следните фази:

- Анализ
- Дизайн
- Кодиране
- Тестване

Когато бързодействието е от важно значение за системата стандартния подход се изменя по следният начин – Фиг. 29 :



Фиг. 29 – Изменение в процеса на стандартният обектно-ориентиран дизайн.

3.1.2.1. Анализ

Анализ е първия етап от процеса на разработка. Както е и първи етап от процеса на разработка на бързоработеща система.

В тази фаза на разработка се определят основните изискванията към системата:

- Изисквания към хардуера (RAM, CPU)
- Изисквания към мрежовата скорост (dial-up, 10-baseT, gigabit, ...)

След определяне на тези изисквания е важно да се определят и изисквания към бързодействието, например:

- Възможността да предоставя средно време за отговор в рамките на 3-5 секунди и дори по-бързо при с работа с 100 конкурентни потребителя.
- Приложението / Страницата да се зарежда за по-малко от 10 секунди.
- Видео връзката да подържа 24 кадъра / секунда при минимални изисквания към хардуера и при мрежова връзка 10 base T.
- И други.

Твърде често проектите преминават от фазата на анализа към дизайн, кодирането и към бета тестване още преди софтуерните разработчици да разберат какви са изискванията към бързодействието на системата. Когато системата работи с огромно количество от данни може да е невъзможно постигането на средно време за отговор от 3-5 секунди, когато продуктът вече почти е завършен. В такъв случай постигането на такъв резултат може да означава започване на проекта отначало. Затова е изключително важно да се определят системните изисквания и изискванията към бързодействието именно в тази фаза. Може да се окаже, че клиентите не са съгласни с такова бързодействие, те могат да искат страницата да се зарежда за 3 секунди, въпреки че данните, които трябва да се покажат са от порядъка на GB. Това води до допълнителни преговори за постигане на консенсус между клиентите и разработчиците, но е важно да се съобрази този факт и да се осъществи още при процеса на анализ. Преместването на тези преговори накрая на процеса на разработка е много опасно за бъдещето на продукта.

3.1.2.2. Обектно – ориентиран дизайн

Обектно – ориентираният дизайн играе основна роля. Докато има много фактори, влияещи върху добрия дизайн, има само една концепция, която е изключително важна за създаването на добра продуктивна система. Концепцията е позната като *каспулиране*. В книгата си Крейг Ларман [3] дава дефиниция на *капсулиране*:

„механизъм, използван за криене на данните, вътрешните структури и детайли по имплементацията на един обект. Всичките взаимодействия с този обект става чрез един публичен интерфейс от операции”.

Капсулирането насърчава да се скрива вътрешната структура на един обект от останалата част от проекта. По принцип капсулирането се използва най-често в контекстта на подръжката на един софтуер, а не в контекст за подобряване на неговото бързодействие. Обаче според някои схващания, капсулирането може дори да навреди на бързодействието, т.к. то изисква по-индиректно да се борави с данните както и извикването на повече методи[4]. Този ред на мисли може и да е правилен, но погледнато глобално е точно обратното, капсулирането е винаги в основата на една добре продуктивна система. Защото то предоставя следните възможности:

- Бързо да се оценяват различните алгоритми и структури от данни кои работят най-бързо.
- Бързо да се адаптира една система към нови променени изисквания (увеличение на размера на данни с няколко пъти, Промяна изискванията от страна на клиента, ...)

Нека се разгледа система с около 100 конкурентни потребителя и данните, с които борави да се съхраняват в обикновени файлове. Но ако тя се предостави за ползване от голяма компания с много потребители или просто потребителите й нараснат над 100 000 например, тя ще е предпоставка за bottleneck. За да се избегне това, най-вероятно системата ще трябва да се настрои да работи с добра база от данни, с която ще е много по-лесно, по-бързо, по-сигурно и т.н. Но ако механизмът или модулът за боравене с данните не е добре капсулиран или ако системата на много места очаква да се оперира с данните посредством файлове - тази настройка би отнела много време за да се промени системата, така че да отговаря на новите изисквания. Резултатът от капсулирането води до пестене на време при промяна на условията, но най-важното за този контекст е да се изпробва с различни софтуерни решения и да се прецени кое е най-бързото, което ще доведе до по-добра производителност.

3.1.2.3. Кодиране

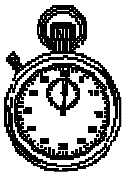
Очевидно начинът, по който е написан кода на системата влияе върху нейната производителност[7]. Ясно е също, че два фрагмента от код, изглеждащи подобно на пръв поглед могат да имат коренно различна производителност. Важно е да се има предвид, че може да се прекара много време в оптимизирането на кода на една система, но най-важните стадии на разработката са анализа, дизайна и профайлинга и ако на тях не е обърнато достатъчно дълго време - оптимизацията на кода няма да има голямо значение.

3.1.2.4. Тестване

Тестването е важен елемент от процеса на разработка на софтуер. Но тук ще се фокусираме върху тестване на бързодействието, а не на качеството. Тестването на бързодействието е всъщност benchmarking. След като системата е стартирана дори и като бета версия е възможно да се конструира benchmarking за измерване на продуктивността. Ако има солиден набор от такива тестове, се позволява да следи и записва бързодействието на една система и най-малкото ще може да се провери дали тя отговаря на изискванията на клиента[25].

Бенчмаркинг е процес на сравняване на различни процеси по начина, по който се извличат количествени данни. Той играе ключова роля в осигуряване на това, че една система работи добре. Процесите, които се сравняват могат да бъдат два различни алгоритъма, които изкарват един и същи резултат или два различни компилатора , изпълняващи един и същи код. Ключов аспект в бенчмаркинга е сравнението. Един единствен бенчмаркинг резултат не е интересен, той е полезен, когато има някаква база за сравнение. Бенчмаркингът обикновено сравнява количеството време за изпълнението на специфична задача, но той може също да бъде използван за измерване и на други променливи като например количеството изразходвана памет[26].

Пример за примитивно средство за бенчмаркинг:



Фиг. 30 - Хронометър

Хронометърът (Фиг. 30) може да бъде много ценно средство, въпреки че изглежда прекалено опростен начин, много често е най-добрия способ за тази цел. С един хронометър може да се измерва[26]:

- Колко време е необходимо да се зареди определено приложение или отделна страница.
- Колко време е необходимо да се отвори огромен документ.
- Колко време е необходимо за да се обходи огромен списък от данни.
- Колко време е необходим за изпълнението на сложна заявка към базата данни, имаща солиден I/O.

Въпреки това, хронометърът не винаги е удобен за тази цел. Друга такава техника, която е полезна в широк спектър е добавянето на времеви изчисления към код, който се наблюдава.Общо взето тази техника е подобна на хронометъра, само че тя дава по-голяма точност, по-лесна за ползване т.к. тя автоматично се стартира и спира и по-лесна за анализ т.к. резултатите може да се записват на определено място и по определен начин – например файл, база от данни и т.н.

Основната цел на бенчмаркинга е:

- Сравнява бързодействието на алтернативни решения.
- Дава детайлен изглед на бързодействието на системата.
- Може да анализира и записва резултатите през целият процес на разработка на едно приложение.

Типове бенчмаркинг:

- Микро бенчмаркинг
- Макро бенчмаркинг

Микро бенчмаркинг

Микро бенчмаркинг може често да бъде написан само с няколко реда код. Той е полезен, когато се сравняват различни решения. Например, когато трябва да се избере най-добър сортиращ алгоритъм на някакъв списък от данни, изборът може да е между няколко алгоритъма върху едно и също множество от данни и да се прецени кой работи най-добре. Може да се избира между два класа за работа с файлове и да се избере този с по-добрата производителност.

Макро бенчмаркинг

Въпреки, че микро бенчмаркинга е полезен в много ситуации, той не е универсален, в такива ситуации може да се ползва макро бенчмаркинга. Истинският тест на макро бенчмаркинг е, когато се тества системата както би работила тя в реалния свят. Но за да се построят такива тестове трябва да се знае как биха постъпили потребителите като използват системата. Най-добрият вариант на такива тестове е да се правят в реална обстановка. Например ако приложението е уеб базирано и искаме да разберем как ще се държи ако голям брой потребители се логнат едновременно и всеки от тях пусне тежка заявка към базата от данни, резултатът от която е милиони редове. Едно от лесните решения е, че почти за всяка технология има готови такива софтуерни средства, които могат да имитират до съвършенство желаната от нас за тест потребителска заявка към дадена уеб система.

Анализ на бенчмаркинга

Когато се използва някакво множество от бенчмаркинги за дадена система е много важно да анализират и представят в подходящ вид техните резултати. Трябва да се има в предвид, че резултатите могат да варират между различните стартирания. Това може да се дължи на много причини, такива като странични процеси на обкръжаващата среда, като например мрежов трафик, което за система, връщаща огромно количество от данни към потребителя, едно такова вариране в производителността на мрежата е сериозен проблем.

Когато резултатите от различните бенчмаркинги са вече записани, ще трябва да им се направи статистически анализ.

Съществуват много типове статистики, които могат да бъдат полезни, но минимума е следният[26]:

- Най-добрия резултат
- Най-лошия резултат
- Среден резултат

Изчислението на тези статистики е лесно - най-лошият е най-дългото време, най-добрият е с най-късото време, средният е средно аритметично на всичките времена.

Важно е да се разбере, че бенчмаркингът е важен процес, чрез него се сравняват два различни хардуера, по-бързият между два процесора, по-бързият хард диск, и т.н. Но може би най-важното му качество е способността му да записва резултатите и да се предоставят за анализ. Когато се поправят бъгове или се добавят нови функционалности системата може да промени своята производителност, но пускането на такива тестове периодически дава възможност да се следи дали софтуера става по-бърз или по-бавен.

3.1.2.5. Профайлинг (Profiling)

Много системи не изпълняват своите изисквания за бързодействие от първия опит. Ако се случи това или ако системата има проблем с бързодействието, е нужно да се започне с профайлинг. Профайлингът определя какви зони от системата консумират

най-много ресурси. Много средства има в помощ на този процес, като те са особено полезни при идентифициране на проблеми с CPU и RAM.

Анализирайки данните от профайлинга, може да се идентифицират частите от системата, които създават проблеми. Тази информация може да бъде използвана за да се определи какви промени са необходими за по-добра производителност. Понякога решенията са лесни - модифициране на един единствен метод, алгоритъм или структура от данни, но с тази информация може да се открият „пукнатини" в обектно-ориентирания дизайн, дори още в процеса на анализ.

Без този процес може само да се гадае къде в системата е необходима оптимизация, както и да се загуби време в оптимизиране на код, който не е толкова важен и проблемен, а истинските причинители да останат скрити и недокоснати.

След изпълнение на този процес трябва да може да се отговори на следните въпроси[26]:

- Кои методи се извикват най-често?
- Кои методи използват най-много време?
- Кои методи извикват най-използваните методи?
- Кои методи заделят най-много памет?

Отговорът на тези въпроси може да даде отговор дали някъде в системата има bottleneck.

Действията за отстраняване на bottleneck са разнообразни, но може да се започне със следните два:

- Правенето на често използваните методи по-бързи.
- Извикването на бавните методи по-рядко

Тези две тактики могат да бъдат много успешни. Но много често разработчиците грешат и са заслепени като използват само първата.

Анализът на данните от профайлинга може да помогне за намиране на дупки в паметта.

Изолирането на такива дупки може да бъде времеемка, досадна и трудна задача.

Добрите средства за профайлинг биха предоставили възможността:

- Запис на броя от живи инстанции на всички класове за желан от нас период на работа на системата.
- Изолиране на специфичен обект и възможността да се видят кои са референциите, които сочат към него.
- Ръчно да се извика garbage collection.

Различните средства за профайлинг имплементират всичките тези функционалности по различен начин, но няма значение кой от тях се използва стига да се достига до същият резултат.

Едно от определенията за стратегия според *Random House Webster's Dictionary:*

"Средство или метод за постигане на цел" [30]

Следването на гореизложените стратегии ще помогне да се избегне един от най-дълбоките проблеми залегнали при разработката на системи, боравещи с огромен размер информация - ниското бързодействие. Следването им ще помогне за постигане на желаната цел - висока производителност.

3.2. Тактики за постигане на високо бързодействие

Тук ще се опишат най-популярните тактики за повишаване на производителността. Гледната точка е представена от страната на програмиста, представена като тактики за писане на ефективен програмен код. Списъкът от тактиките, които ще се разгледат:

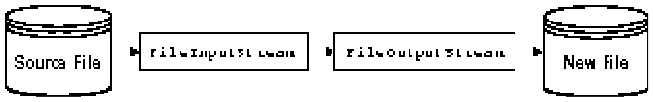
- Бързодействие, свързано с I/O.
- Размера на използваната памет.
- Контрол на зареждането на класовете.
- Алгоритми и структури от данни.

3.2.1. Бързодействие свързано с I/O

Най-популярните обектно-ориентирани езици за програмиране притежават I/O библиотеки. Правилната работа с тях е важна за постигане на добро бързодействие на системата. В основата на всички разсъждения ще бъде работата се тях.

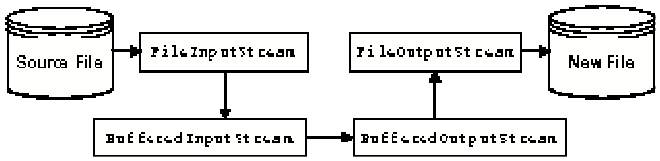
Една от най-често срещаните причини за ниска производителност при I/O операциите е не използването на буфериране при четене и писане на диска. Хард дискът е бърз когато чете и пише от/върху диска, но между различните подходи на използване може да има коренна разлика. Например той е много по-слабо ефективен, когато работи с малки блокове от данни. За максимизиране на работата му е важно да се подават по-големи парчета от информация наведнъж. Такава е имплементацията на буферираните потоци от данни. Ако се представи едно четене/писане от файл и операциите, свързани с тях са разпределени байт по байт, колкото по-малко парчета данни се подават към хард диска толкова по-бавно работи той. Простото обяснение е, че колкото по-голям е броя обръщения към него, толкова повече забавяне се наблюдава. Ако един голям файл се копира от едно място на друго и се чете и пише байт по байт, броя обръщения ще бъде изключително голям, поради това и забавянето може да бъде изключително голямо.

Копиране на файл от едно място на друго на принципа байт по байт:



Фиг. 31 – Копиране байт по байт

Копиране на файл от едно място на друго на принципа на буфериране:



Фиг. 32 – Копиране на принципа на буфериране

От фиг. 31 и фиг. 32 се вижда, че при буферирането се добавя още едно ниво в целия този процес. Примерите отгоре са дадени с I/O библиотеки на езика Java, за други езици за обектно-ориентирано програмиране логиката е подобна.

Има още един метод за подобряване на бързодействието чрез буфериране, което може да доведе до още по-добра производителност – буфериране по схема, определена от самия програмист. В основата на този метод е отново намаляване на обръщенията към хард диска, като се буферират още по-големи масиви от данни. Ако дадената ситуация позволява може да бъде сведена дори до само 1 масив, който да съдържа целия файл. Принципът е следния - данните просто се записват в един байт масив и се подават наведнъж за запис. Но при ползването на тази техника трябва да се вземе предвид и конкретната ситуация. Ясно е, че тук се разглежда система, боравеща с много данни, а при нея има опасност този масив да стане много голям т.к. е твърде вероятно размерът на файловете да бъде голям, може да се изпълнява от голям брой конкурентни потребители. Всичко това трябва да се съобрази от разработчиците за не се стигне до изразходване на наличната памет. Ако е съществува такава опасност – данните могат да се разпределят на няколко порции – на равни по размер структури от байтове.

За доказателство на изложеното по-горе е направен тест с един малък JPEG файл от 370K за копирането му от едно място на друго. Резултатите са следните:

Таблица 1

Метод	**Време**
Основен метод	10800 ms
Буфериране	130 ms
Буфериране2	33 ms

Ускорението на бързодействието при I/O операции може да бъде постигнато и в зависимост от типа приложение. Ако се вземе например FTP сървър или HTTP сървър и се приеме, че тяхна основна задача е копирането на файлове от диск на мрежов сокет. Въпреки, че те най-вероятно боравят с голям брой файлове, много често една малка част от тях се използва изключително често. Също така например при даден уеб сайт, зареждащата страница или някоя друга, винаги се поисква от потребителя. При тези случаи може просто тези най-често използвани файлове да се поставят в байт структури директно в паметта, за да се спестят тези операции. Така няма да е необходимо да се четат от диска всеки път и могат просто бързо и лесно да се копират от паметта в мрежовият сокет.

Сериализацията в обектно-ориентираното програмиране е процес, при който един обект се представя така, че да може директно да бъде четен и писан в поток от данни, да може да бъде записан във файл или изпратен по мрежата. В повечето езици, имплементацията на един такъв обект е проста задача (в Java се имплементира интерфейса Serializable). Тази тактика при някой тип приложения може да се укаже изключително полезна и също да подобри цялостната им производителност.

Голям процент от системите, боравещи с много информация притежават процеси свързани с I/O. Поради тази причина I/O bottleneck е често срещан проблем. Спазването на гореизложените принципи ще помогне максимално да се избегнат тези проблеми, а също така и да се избегнат основни и често срещани заблуждения при програмистите при сблъскване с такива ситуации.

3.2.2. Размер на използваната памет

Размерът на използваната памет може да бъде критично важен за производителността на една система. Приложения, използващи твърде много памет може да стигнат до положение, в което да разчитат на виртуалната памет на операционната система. Т.к. виртуалната памет е много пъти по-бавна от RAM, това може да доведе до ниско бързодействие. При разработката на една система, трябва да се взима в предвид количеството RAM, което ще притежава машината приемник.

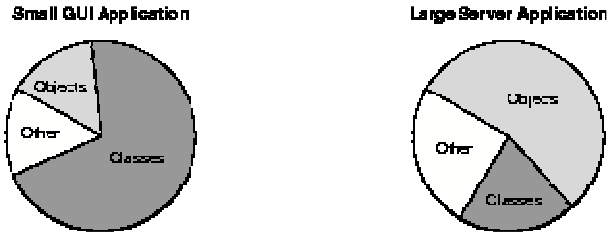
Полезно е да се наблюдава размера на използваната памет от програмните обекти на една система. Разбира се тази статистика въобще не представя реално заделената памет от системата. Но тя дава представа за процентното съотношение между паметта заета от програмните обектите и истинската памет заета от цялото приложение. Тя може да ни предупреди за написан код, който изразходва прекалено много ресурси. Например ако тази статистика се извлича през определен интервал от време и ако размера на използваната памет расте постепенно е ясно, че някъде има проблем.

Един от подходите за измерване на действителната памет, заета от едно приложение е чрез средства на операционната система приемник. При Windows NT нагоре средството се казва Task Manager, при Solaris може да се използва командата pmap чрез командният ред, при Unix / Linux .

Няколко са факторите, които влизат в размера на използваната памет:

- Обектите.
- Класовете.
- Нишките.
- Структури от данни, свързани с операционната система.
- Библиотеки на операционната система.

Обектите и класовете обикновено изразходват най-голямо количество памет. Съотношението между двете обаче варира и зависи от типа на приложението Фиг. 33.



Фиг. 33 - Консумация на RAM на два типа приложения

Тъй като гледната точка, която се разглежда е относно големи сървърни приложения, ясно е, че размерът на инстанциите на обектите в паметта е проблема, върху който трябва да се фокусират разработчиците. Този размер може да се наблюдава, чрез различни профайлинг средства. Резултатите биха ни дали информация, кои са класовете, чиито инстанции най-често и най-много се зареждат в паметта.

Тактиките, които може да се предприемат са:

- Оптимизация на полетата на обектите.
- Ограничение в размера на инстанциите ако е възможно.

Ясно е, че всеки един тип притежава определен размер в байтове. В таблица 2 долу са дадени размерите на примитивните типове в езика Java:

Таблица 2

Примитивен тип	Размер
byte	1 байт
char	2 байта
short	2 байта
int	4 байта
float	4 байта
long	8 байта
double	8 байта
“reference”	4 байта (32 битови системи) / 8 байта (64 битови системи)

По принцип приблизителният размер на обект в Java е:

Сумата от всички полета + допълнително пространство заделено по принцип за всеки обект (около 8 байта).

Разбира се един клас не се състои само от примитивни типове и референции, той често притежава и обекти дефинирани от самият език. Таблица 3 дава малко информация за някои популярни обекти и какъв размер имат те в Java:

Таблица 3

Клас	Java 2
java.lang.Object	8 байта
java.util.Hashtable	96 байта
javax.swing.JtextField	3 109 байта
javax.swing.Jtable	4 086 байта

Показаната статистика по-горе, макар и отнасяща се за езика Java съвсем не налага идеята, че това е валидно само за нея, при другите обектно - ориентирани езици за програмиране логиката е аналогична. Чрез помощта на данни от този тип може да се направи анализ, отнасящ се за най-често заредените обекти в паметта. Всяко поле от тези обекти е препоръчително да се анализира дали правилно му е определен типа. Много често програмистите избират double за тип с плаваща запетая или int за цяло число без много да се замислят, но в действителност float и short да са напълно достатъчни. Оптимизации от този характер биха помогнали да се намали размера на обектите, а оттам и размера на използваната памет.

3.2.3. Контрол на зареждането на класовете

Зареждането на твърде много класове може значително да рефлектира върху размера на използваната памет. Съществуват много обектно-ориентирани техники, които приложени некоректно генерират твърде много класове. Важно е да се отбележи, че създаването на обекти съвсем не е проблем, обектите са ключова част в обектно-ориентираното програмиране и не може да се напише програма без създаването на обект. Но програмистът трябва да е много внимателен за неправилното и ненужното им създаване. Типична и често срещана грешка е създаване им в тялото на цикъл. С помощта на средства за профайлинг, може да покаже на разработчика къде програмата губи най-много време за създаване на специфични обекти и да се отстрани или модифицира с по-ефективно решение.

Минимизирането на временно създадени класове е добра тактика, свързана с контрола по зареждането на класовете. Ключ към тази тактика е използването на ключовата дума const в C++ или подобна концепция в някои други езици за програмиране. С помощта на тази техника всякакъв опит за промяна на даден обект извиква грешка в програмата.

3.2.4. Алгоритми и структури от данни

Разбирането как да се избере най-добър алгоритъм или структура от данни за решението на специфична задача е ключ към високо производителен софтуер.

Умението да се прецени в различните ситуации, с които се сблъсква разработчика кой алгоритъм или структури от данни са най-подходящи е нетривиална и трудна задача.

3.2.4.1. Избор на алгоритъм

Когато се избира алгоритъм и се прави анализ на плюсовете и минусите му, дали е напълно подходящ за прилагане към специфичния проблем - почти винаги се явяват съмнения дали е напълно подходящ за дадената ситуация. Например изборът на даден алгоритъм може да бъде най-бързото решение при един тип от данни, но може да бъде значително бавен при други тип. Също така няма алгоритъм за сортиране, който се държи оптимално във всички ситуации. Поради този факт е важно да се вземе впредвид с какви данни и по какъв начин най-често ще се използва той.

Ако се разгледа например следния проблем – трябва да се избере алгоритъм за изчисление сума от цели числа от специфичен обхват.

Пример 1:

```
public class SimpleSummer {
    public long sum(int start, int stop){
        long acc = 0;
        for(int i=start;i<=stop;i++){
            acc += i;
        return acc;
    }
}
```

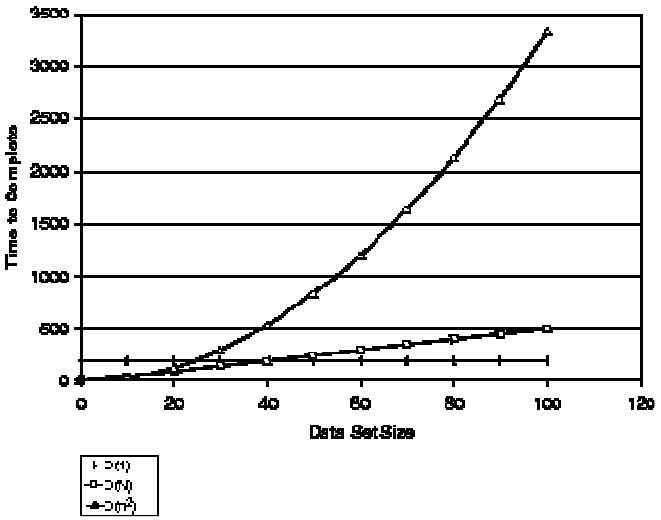
Ако броят от числа, които трябва да се сумират е малък, този прост алгоритъм работи добре. Обаче ако броят им е голям, времето, необходимо за изчисление расте линейно. Ако на приложението се налага често да ползва този алгоритъм и то за сумиране на числа от голям обхват – най-вероятно системата ще има проблеми с бързодействието. Поради този факт е добре разработчиците да помислят за нов по-добър вариант, който да не се влияе до такава степен от големината на обхвата на числата. Ето един по-сложен алгоритъм, който се изпълнява за константно време независимо от броя числа, които се сумират.

```
public class FormulaicSummer {
    public long sum(int start, int stop){
        int bigseries = stop*(stop+1)/2;
        start--; // so result is inclusive of start
        int littleseries = start*(start+1)/2;
        return bigseries-littleseries;
    }
}
```

С прилагането на този алгоритъм – системата ще подобри бързодействието си ако голям брой потребители изискват от нея да сумира числа с голям обхват. Тя няма да се влияе нито от броя потребители, изискващи тази задача от нея, нито големината на обхвата на числата. Но ако се избере първият алгоритъм от примера може дори в определени ситуации да се стигне до bottleneck.

3.2.4.2. Сравнение на алгоритмите

Когато трябва да се избира между два или повече алтернативни алгоритъма, трябва да се сравнят двете решения и да се избере най-подходящото решение за разработвания софтуер. Най-лесният начин да се изпълни това е с бенчмаркинг. Резултатите от него може да се подготвят във вид на графика, с което още по-лесно да се видят характерните особености на всеки от тях.



Фиг. 34 - Сравнение бързодействие на алгоритми

Графиката отгоре Фиг. 34, показва резултатите от три различни алгоритми, които произвеждат едни и същи резултати. Както се вижда алгоритмите с *O(N) O(n2)* имат по-добра производителност при малко данни пред алгоритъма с константно време за изпълнение. Но при увеличение в размера на данните, алгоритъмът с константно време се представя далеч по-добре.

3.2.4.3.Елегантнирешения.

Компютърните специалисти понякога дискутират елегантните решения при писане на програмен код. Обаче елегантният код е като изкуството, изисква в някои случаи далеч повече усилия за реализиране. Но елегантното решение има много преимущества като минимална сложност и максимално подготвеност за решение на даден проблем. Такъв код е почти винаги с добра производителност. За разлика от простият пример по-горе, който е далеч от елегантно решение, въпреки, че си има своите положителни страни.

3.2.4.4. Разглеждане на проблема за решаване.

Когато се преценява какъв алгоритъм да се използва, е много важно да се разгледа в детайл проблема, който трябва да се реши. Защото е по-важно да се избере алгоритъм, който е най-ефективен пред такъв, които е по-обхватен. Или казано по друг начин не важно какво не може да изпълнява, а какво може и колко ефективно се изпълнява.

3.2.4.5. Рекурсивни алгоритми.

Рекурсията е удобно програмно средство, много алгоритми могат да бъдат представени естествено в тази форма. Проблеми, които са сложни и изключително трудни за реализация с линейните алгоритми имат много прости решения чрез рекурсия. Въпреки, че много програмисти и книги съветват да се избягва рекурсията, много често тя предоставя най-доброто решение за дадени проблеми.

Ако рекурсията е толкова полезно нещо, защо много често се препоръчва да се избягва? От гледна точка на бързодействие съществуват две основни причини да се избягва – консумирането на памет и прекалено многото извиквания при изпълнение. Всяко извикване на една функция създава нов стек и ако функцията се извика прекалено много пъти изпълнението на програмата има опасност да предизвика недостиг на памет. Нека се разгледа един достатъчно популярен пример – Ханойските кули и се реализира по рекурсивен и линеен начин. Вижда се, че рекурсивният метод има по-елегантен вид, по-лесен за поддръжка и по-кратък за разлика от линейния, който е съставен от обхождане на цикъл и работи с прекалено много променливи.

Бенчмаркинг отнасящ се за сравнение на двата алгоритъма.

```
public static void main(String args[]) {
   Stopwatch watch = new Stopwatch();
   watch.reset().start();
   Towers(25,'A','B','C');
   watch.stop();
   System.out.println("Time to solve recursively = "+
                         watch.getElapsedTime());
   watch.reset().start();
   LinearTowers(25,'A','B','C');
   watch.stop();
   System.out.println("Time to solve lineraly = "+
                         watch.getElapsedTime());
}
```

След сравнение на бързодействието на двата алгоритъма се наблюдава, че рекурсивният метод е с 15% по-бърз от линейния.

Изводът е, че изборът какъв алгоритъм да се приложи е строго специфична задача и ако изборът падне върху рекурсивен вариант, трябва задълбочено да се анализира евентуалните проблеми, които могат да последват след този избор. Но също така не трябва да се робува на догмата, че рекурсията е лошо нещо и трябва задължително да се избягва, доказателство е примерът с Ханойските кули, както и много други примери от реалния свят.

3.2.4.6. Избор на структури от данни

Често изборът на подходяща структура от данни е толкова важна задача, колкото изборът на правилен алгоритъм. Както при алгоритмите и тук няма такава структура, която да е идеална във всички ситуации.. Но елегантното решение има много преимущества като минимална сложност и максимално подготвеност за решение на даден проблем. Такъв код е почти винаги с добра производителност. За разлика от

простият пример по-горе, който е далеч от елегантно решение, въпреки, че си има своите положителни страни.

3.3. Извод

Когато пред екипа от разработчици стои трудната задача да се създаде система, която ще има предназначението да борави с данни в огромни размери, вниманието на разработчика би трябвало да се насочи към проблема с производителността като се започне от по-високо до по-ниско ниво на анализ и разработка на софтуера. Затова и в тази глава се започна с анализ на стратегиите за постигане на високо бързодействие от най-горното ниво при изграждане на софтуера и се стигна до тактики за писане на програмен код. Подценяването на някой от гореизложените фактори може да изложи на риск цялата система и да доведе до преработка на някои нейни елементи или да се изправи пред проблема да има ниска производителност и да натрупа неудовлетвореност от страна на своите потребители. Още повече, че постигането на задоволително бързодействие съвсем не е лека задача и много често е свързана с дълбоки познания в много области от сферата на информационните технологии.

Глава 4
Демонстрация на подходи на извличане на големи масиви от данни

Демонстрацията е реализирана, чрез уеб базирано приложениие представящо три различни подхода при извличане на големи масиви от данни:

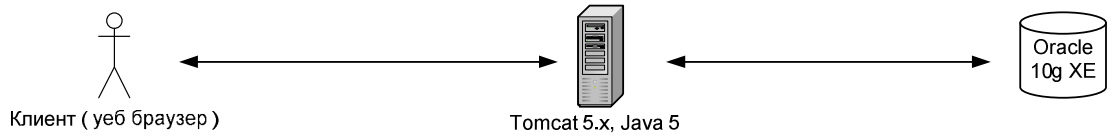
- Традиционнен подход.
- Подход основан на Page-by-Page Iterator.
- Подход основан на Value List Handler.

Бизнес логиката

Бизнес логиката е свързана с представяне на списък от книги. Всяка книга има специфични свойства - Идентификатор, Име, Автор, Издател, Дата на Издаване и Цена.

Технически средства

Техническите средства с помощта, на които функционира приложението са изобразени във Фиг. 35. Клиентската част е уеб браузер, сървърния код се изпълнява на уеб контейнера Apache Tomcat версия 5.x, информацията се съхранява в релационна база от данни - Oracle 10g Express Edition.



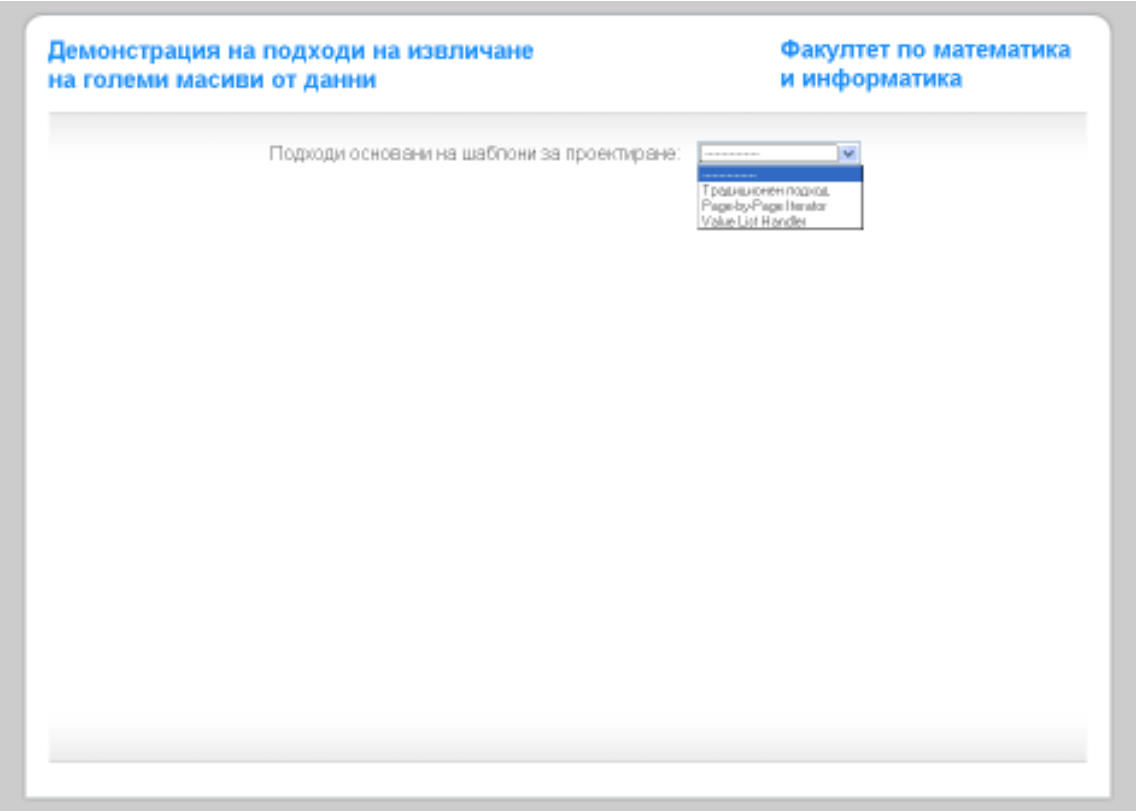
Фиг. 35 – Схема на приложението

Сървърни код е реализиран, чрез Java Server Pages и Java Servlets.

Информацията в Базата от данни се съдържа само в една таблица *books*, състояща се от следните колони – *id*, *name*, *author*, *publisher*, *publication_date* и *price*. Данните в нея са около 1 000 000.

4.1. Представяне на потребителският интерфейс

Приложението се състой от една уеб страница представяща различните подходи при извличане на данни.



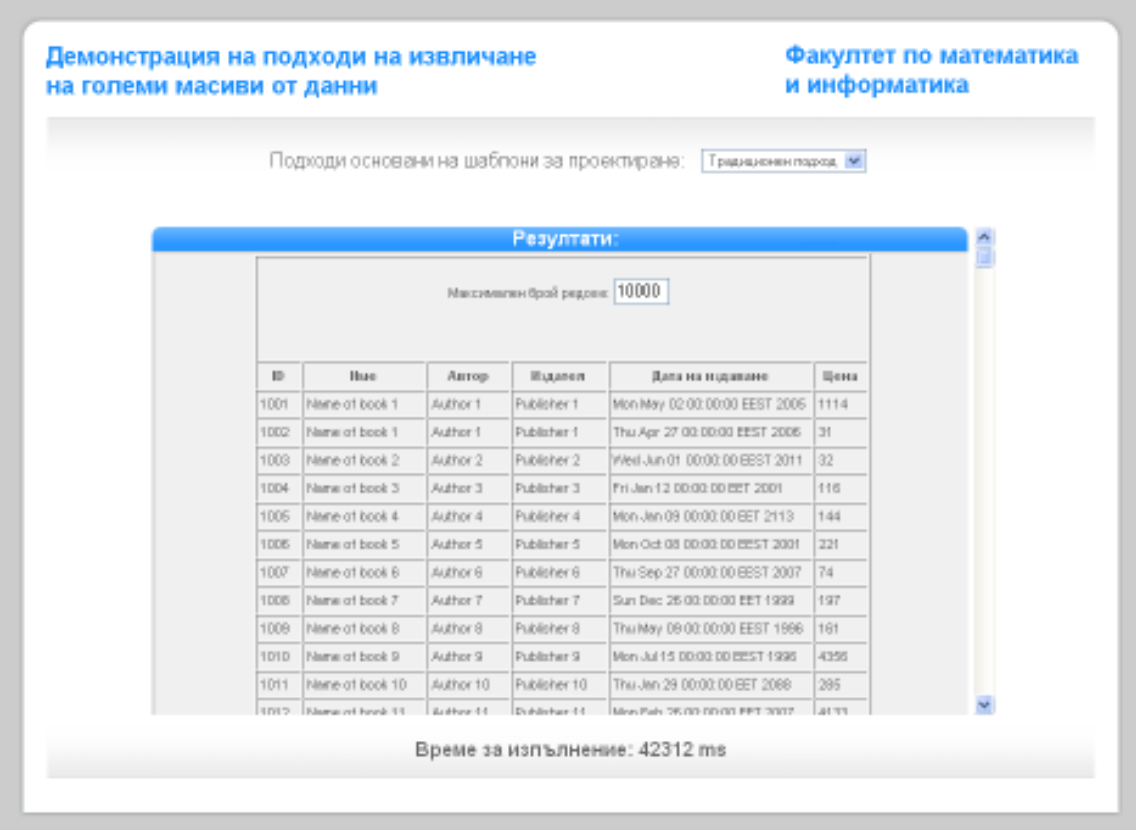
Фиг. 36 – Основна страница от приложението

Дизайна на страницата е създаден да бъде максимално опростен. На потребителят му трябва само един поглед за да разбере какво и е предназначението и как се използва. Основен елемент от уеб страницата е drop-down контрола, която при смяна на текущата си стойност генерира събитие (изпраща се нова заявка към сървърът).

След изпълнието на заявката, независимо според кой подход е реализирана, резултатите винаги се визуализират в табличен вид (таблица разпложена непосредствено под drop-down контролата). Също така винаги след таблицата с данните се изписва съобщение за времето на изпълнение на цялата заявка.

4.1.1. Традиционен подход

Ако потребителят е избрал традиционнен подход на извличане и визуализиране на данни, по подразбиране ще му се визуализира страница подобна на Фиг. 37.



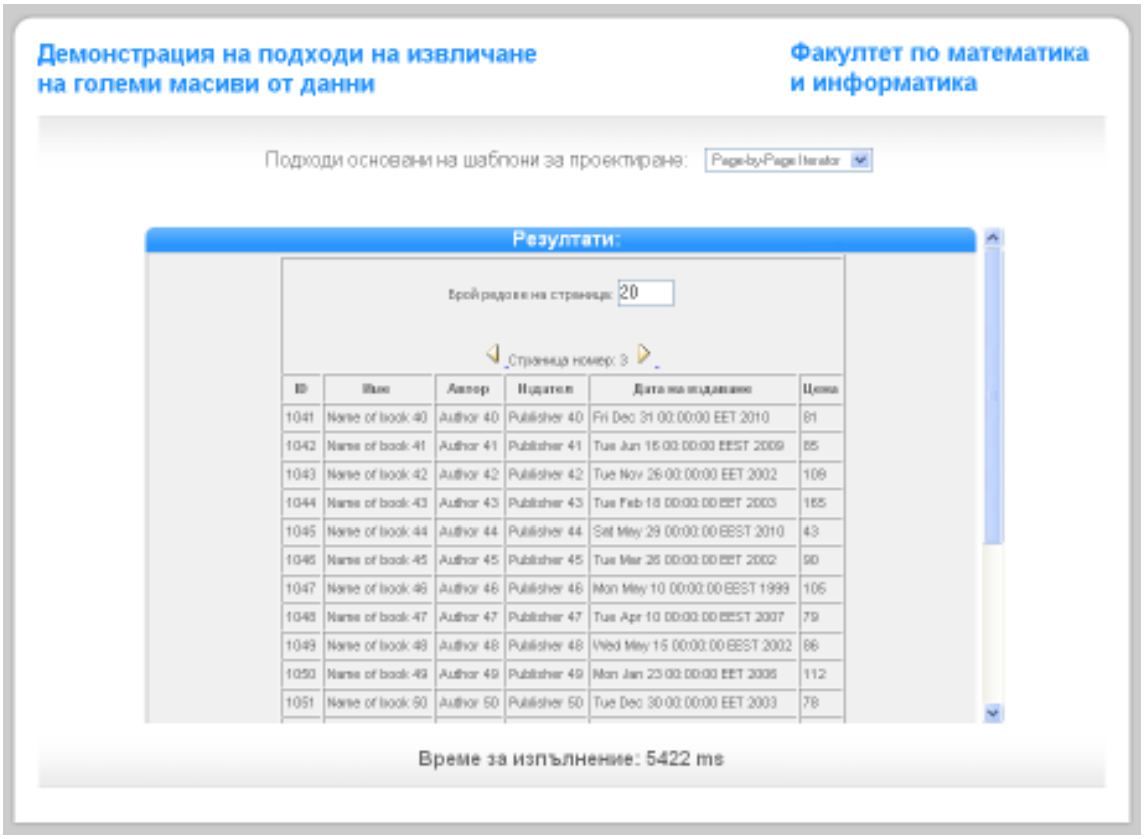
Фиг. 37 – Традиционен подход

Според този подход данните се извличат и визуализират наведнъж в клиентският браузер. За да е по-гъвкъв примера над полето с таблицата има текстово поле, което указва броя редове, които ще бъдат извлечени и визуализирани. Чрез въвеждане на различни стоиности в това поле може да се сравнят отделните времена за изпълнение, с което по-лесно да се видят преимуществата и недостатъците на този подход.

Данните са соритирани по полето идентификатор. С помощта на скролера, потребителят може да обходи целият списък от данни (или преставените 10 000 реда както е показано във Фиг. 37).

4.1.2. Page-by-Page Iterator

Ако потребителят е избрал Page-by-Page Iterator подхода за преставяне на данни, по подразбиране ще му се визуализира страница подобна на Фиг. 38.



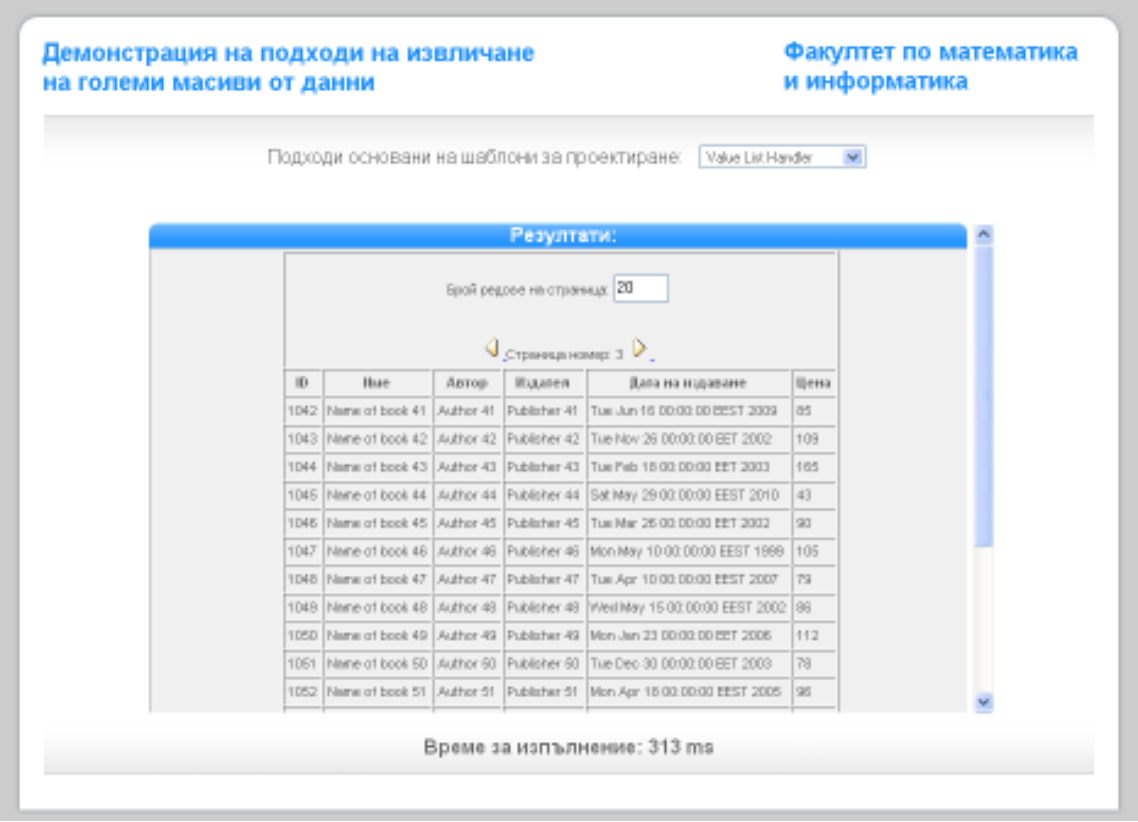
Фиг. 38 – Page-by-Page Iterator

Според този подход данните се извличат и визуализират страници по страници. Броя редове на страница може да се зададе в текстово полето разположено над таблицата с резултатите. В интерфейса на този подход има някои допълнителни елементи – бутон за връщане една страница назад, бутон за преместване на една страница напред и съобщение за номера на текущата страница. И тук след таблицата с резултатите излиза съобщение за времето за изпълнение на текущата заявка.

И тук данните по подразбиране са соритирани по полето идентификатор. С помощта на скролера, потребителят може да обходи целият списък от данни (или 20 реда както е показано във Фиг. 38).

4.1.3. Value List Handler

Ако потребителят е избрал Value List Handler подхода за преставяне на данни, по подразбиране ще му се визуализира страница подобна на Фиг. 39.



Фиг. 39 – Value List Handler

Принципът на визуализиране на данните според този подход по никакъв начин не се различава от този с Page-by-Page Iterator. Всичко казано за потребителският интерфейс за Page-by-Page Iterator важи и за Value List Handler.

4.2. Представяне принципите на извличане на данни.

Основна разлика във всеки един от описаните подходи се състои или в sql заявката към базата от данни или в начина на представяне на данните в слоя грижеш се за бизнес логиката.

4.2.1 Принципи на извличане на данни според Традиционния подход

Традиционния подход извлича цялата информация наведнъж от базата от данни след, което тя се препраща към потребителя. В Опис 1 е показана sql заявка, удовлетворяваща този подход:

```
select id,
       name,
       author,
       publisher,
       pub_date,
       price
from   books
order by id
```

Опис 1 – SQL заявка въз основата на традиционния подход

След като данните се извлекат от базата от данни, в бизнес слоя на приложението java метод (Опис 2) зарежда данните в java.util.List<Book>. От където след това jsp страницата лесно ги визуализира в броузера на потребителя.

```
private List<Book> getBooksByStandardMethod( HttpServletRequest request ) throws SQLException,
ParserConfigurationException, SAXException, SettingException {

       int maxRows = DEF_NUM_ROWS_STAND_APPROACH;
       if( request.getParameter( MainFormHelper.FORM_MAX_ROWS ) != null ) {
              maxRows = Integer.parseInt(
                                          request.getParameter( MainFormHelper.FORM_MAX_ROWS
                                   ) );
       }

       return queryObject.getByStandardApproach( maxRows );
}
```

Опис 2 – Java метод извличащ данни според Традиционния подход

4.2.2 Принципи на извличане на данни според Page-by-Page Iterator.

При Page-by-Page Iterator заявката към базата от данни се различава от традиционният подход. Тук се извлича само толкова информация, колкото е заявена от потребителя, без да се изразходват излишни ресурси на сървъра. В Опис 3 е показа sql заявка, удовлетворяваща този подход:

```
select bb.*
from   (
         select aa.*,
                rownum as row_num
         from   (
                  select id,
                         name,
                         author,
                         publisher,
                         pub_date,
                         price
                  from   books
                  order by id
                ) aa
        ) bb
where   bb.row_num between :1 and :2
```

Опис 3 - SQL заявка извличащ страница по страница.

И тук както в горния подход след като данните се извлекат от базата от данни, в бизнес слоя на приложението - java метод (Опис 4) генерира java.util.List<Book>. От където след това jsp страницата отново лесно ги визуализира в броузера на потребителя.

```
private List<Book> getBooksByPageByPageIterator( HttpServletRequest request ) throws
SQLException, ParserConfigurationException, SAXException, SettingException {

        int numRows = DEF_NUM_ROWS_BY_PAGE;
        if( request.getParameter( MainFormHelper.FORM_NUM_ROWS ) != null ) {
                numRows = Integer.parseInt(
                                        request.getParameter( MainFormHelper.FORM_NUM_ROWS ) );

        }

        int startIndex = ( this.pageNum * numRows ) - ( numRows - 1 );
        int endIndex = ( this.pageNum * numRows );

        return queryObject.getByPageByPageIterator( startIndex, endIndex );
}
```

Опис 4 - Java метод извличащ данни според Page-by-Page Iterator

4.2.3 Принципи на извличане на данни според Value List Handler.

При Value List Handler заявката към базата от данни (Опис 3) е същата като тази в Page-by-Page Iterator. Само че с едно изключение, данните, които се извличат с една заявка имат по-голям обхват. Или за по-голяма яснота, ще се даде следният пример, когато потребител изисква информация за страница, приложението изпълнява заявка не само за страница 1, а и за следващите две страници след нея, след което кешира извлечения резултат. Така, че при следващата заявка на потребителя за страница 2, данните ще се върнат по най-бързия начин от кеша.. В опис 5 е представена реализацията на този подход.

```
private List<Book> getBooksByValueListHandler( HttpServletRequest request ) throws
SQLException, ParserConfigurationException, SAXException, SettingException {

        int numRows = DEF_NUM_ROWS_BY_PAGE;
        if( request.getParameter( MainFormHelper.FORM_NUM_ROWS ) != null ) {
                numRows = Integer.parseInt(
                                        request.getParameter( MainFormHelper.FORM_NUM_ROWS ) );

        }

        List<Book> results = new ArrayList<Book>();
        int startIndex = ( this.pageNum * numRows ) - ( numRows - 1 );;

        if( this.pageNum == 1 || ( this.pageNum-1 )%3 == 0 ) {
                int goNextPage = Integer.parseInt(
                                request.getParameter( MainFormHelper.FORM_GO_NEXT_PAGE ) );

                if( goNextPage > 0 ) {
                        int dbEndIndex = ( this.pageNum * numRows ) + ( 2 * numRows );
                        results = queryObject.getByPageByPageIterator( startIndex, dbEndIndex );
                                this.setBooksInSession( request, results );
                }
        }

        int endIndex = ( this.pageNum * numRows );

        books = this.getBooksFromSession( request );

        return books.subList( startIndex, endIndex );
}
```

Опис 5 - Java метод извличащ данни според Value List Handler

4.3. Изводи

Събрани резултати от примерното уеб приложение относно традиционния подход на извличане и визуализиране на данни:

Брой извлечени редове	Време в милисекунди
20	5 141
100	5 594
1 000	14 128
10 000	115 530
50 000	569 375
100 000	1 073 815

Таблица 1.

От резултатите от Таблица 1 лесно се прави извода, че този подход не е особенно удачен за представяне на голямо количество от данни. Лесно се вижда, че при малък обем данни, информацията бързо се показва на потребителя, но колкото повече расте обема с още по-бързи темпове нараства времето за изпълнение.

Сравнение на резултати между Page-by-Page Iterator и Value List Handler

Шаблон за проектиране	Брой редове на страница	Общо време за обхождане на 1,2 и 3 страница
Page-by-Page Iterator	20	15 583
Value List Handler	20	6 109
Page-by-Page Iterator	100	16 881
Value List Handler	100	7 983
Page-by-Page Iterator	1 000	45 470
Value List Handler	1 000	35 597
Page-by-Page Iterator	10 000	353 453
Value List Handler	10 000	353 540

Таблица 2.

От стоиностите от таблица 2, при малко количество от данни на страница, шаблона Value List Handler дава по-добри резултати от Page-by-Page Iterator. Но с нарастване на обема от данни, преимуществото се обръща на обратно в полза на Page-by-Page Iterator. Макар, че едва ли са много приложенията, които визуализират списък от 10 000 реда наведнъж.

Важна характериситка на извлечените резултати, е че всичките са събрани при един единствен активен потребител. И поради тази причина за да се провери наистина ли Value List Handler е по-препоръчителния избор, както пролича от резултатите от Таблица 2, се направи още един тест. Изпълниха се 50 едновременни произволни заявки към разглежданото уеб приложение, първо спрямо единия подход, след това спрямо другия, с резултат от 1 000 реда на страница. При тестването на Value List

Handler почти всички случаи завършиха със следния exception *java.lang.OutOfMemoryError: Java heap space*. Всички тестовете на Page-by-Page Iterator приключиха успешно. След тези резултати е важно е да се поясни каква е машината, на която се изпълниха всичките тези тестове – Windows XP, CPU 3 GHz, 1 GB RAM. Ясно е, че параметрите дори не могат да се сравняват с тези на един мощен сървър. Но въпреки, това, изводът които произтича е следния - при голям брой потребители и малък рамер свободна памет Value List Handler не е подхода, който трябва да се предпочете при представяне на голямо количество от данни. Ако все пак се предпочете е добре да се комбинира с добър механизъм на попълване и освобождаване на данни от кеша. Най-удачен подход измежду трите представени се оказа Page-by-Page Iterator, показващ най-добри резултати, при голям брой конкурентни потребители и голям обем данни.

Заключение

В дипломната работа вниманието е фокусирано върху системи, представящи големи масиви от данни. Разгледан е дизайна на съставните компоненти, на потребителския им интерфейс, процесите на обработка на данните при подготовката им за представяне, както и са предложени идеи за разрешаване на проблемите,свързани с бързодействието.

Описаните техники имат своите силни и слаби страни, които са изложени нагледно. Решенията им засягат най-често срещаните проблеми и основните принципи и подходи за премахването им. Най-вероятно повечето уеб приложенията, представящи големи масиви от данни биха се сблъскали с точно такива ситуации и по подобен начин биха се справили с тях.

При разработването на такива системи, екипът натоварен със задачите на дизайн на най-ниският слой (слоят грижещ се за данните) поема големи отговорности за постигане на поставените цели. Тук в този труд се фокусира вниманието върху различните методи на извличане на данните от някакъв носител на данни (най-често релационна база от данни). Но не се споменава за методите, техниките или най-добрите практики на дизайн на този важен елемент от една система. Тази дипломна работа може да бъде развита в този аспект за да постигне един по-цялостен и по-завършен вид.

Терминологичен речник

Термин	Тълкуване
10-baseT	Вид мрежова връзка оперираща с 10Mbps
Ajax	Похват в уеб разработките за създаване на интерактивни уеб приложения
Amazon.com	Интернет сайт за продажба на книги
Application сървър	Софтуерен инструмент, чрез които се доставя приложения към клиентски компютри и устройства
asp	Механизъм на основата на скриптове за генериране на динамични html страници
Benchmarking	Процес използван за сравнение и измерване на различните подходи на решение на даден проблем.
Bottleneck	Точка на стеснение на производителността в едно приложение или мрежа.
c++	Обектно – ориентиран език за програмиране
Constraint на базата от данни	Ограничително средство в базата данни
CPU	Вж. Процесор
Data Access Object	Шаблон за проектиране
dial-up	Вид мрежова връзка чрез телефонна линия
Dispatcher	Шаблон за проектиране
download на файл	Копиране на файл на локален компютър от интернет
Drop-down филтър	Падащо меню
Dual Slider филтер	Контрола, при която стойностите се преместват наляво или надясно в предварително дефинирана уразмерителна система
Facade	Шаблон за проектиране
Fast Lane Reader	Шаблон за проектиране
Front Controller	Шаблон за проектиране
Gigabit	Тип мрежова връзка за пренос на голямо количество от данни

Google	Интернет търсачка
GSM	Най-популярното мобилно устройство
Helper класове	Шаблон за проектиране
HTML	Език за програмиране на уеб страници
HTTP	Протокол за комуникация използван за трансфер през интернет
I/O	Входни / Изходни операции твърдият диск на компютър
Java	Обектно-ориентиран език за програмиране
join заявка	Метод на свързване редовете между две таблици
jsp	Технология базирана на езика Java генерираща динамични html страници
Model-View-Controller	Шаблон за проектиране
Page–by–Page Iterator	Шаблон за проектиране
PDA	Електронно устройство, които включва в себе си функционалност присъща за персонален компютър, мобилен телефон, музикален плейър и камера
RAM	Тип на съхранение на данните използвано в компютрите
SQL	Език, чрез които се извличат и менажират данни в релационните бази от данни.
Task Manager	Програма използвана за снабдяване информация за процесите и програмите в компютъра
Value List Handler	Шаблон за проектиране
Value Object	Шаблон за проектиране
Value Object Factory	Шаблон за проектиране
View creation Helper класове	Шаблон за проектиране
View Mapper	Шаблон за проектиране
XML	Език комбиниращ текст и допълнителна инфромация относно текста
Алгоритъм	Упътване за решаване на един проблем или на определен вид проблеми
Анализ	Етап от разработката на софтуер

База от данни	Колекция от логически свързани данни в конкретна предметна област, които са структурирани по определен начин
Байт	Единица за обем на информация, записана в цифров (двоичен)вид
Браузер	Софтуерно приложение даващо възможност на потребителите да боравят с текст, картинки, музика и друга информация обикновено разположена на уеб страница.
Буфер	Памет за междинно съхранение на данни в компютър
Бързодействие	Величина указваща производителността на система, процесор и т.н.
Валидация	Процес които проверява за правилно извършени операции от потребител или компютърана програма
Дизайн	Етап от разработката на софтуер
Инструмент	Средство за специално създадено или пригодено да е в помощ на разработчика при създаване на компютъран програма
Интернет търсачка	Система, предназначена да намира информация, съхранена в Интернет
Капсулация	Вж. Капсулиране
Капсулиране	Наричано също скриване на информация: Прави невъзможно за потребителите на даден обект да променят неговото вътрешно състояние по неочакван начин
Кеш	Спомагателна памет за ускоряване обмена на данни
Клас	Основна единица в обектно-ориентираното програмиране
Кодиране	Етап от разработката на софтуер
Комбинирано разпределение	Вид странициране на данните
Компонент	Съставна част в едно приложение
Мрежа	Компютърна мрежа, обслужваща група персонални компютри
Обекти	Пакетира данни и функционалност заедно в обособени единици в една компютърна програма.

Обектно – ориентирано програмиране	Парадигма в компютърното програмиране, при която една програмна система се моделира като набор от обекти, които взаимодействат помежду си.
Онлайн банкиране	Средство, чрез компютърен софтуер, за осъществяване на разплащания.
Потребител	Човек използващ компютърна програма
Потребителски Интерфейс	Ссредата чрез която потребител се възползва от функционалността на конкретната програма или устройство
Програмен код	Текст, който представлява алгоритъм написан от програмист на даден език за програмиран е
Профайлинг	Изследване на поведението на компютърна програма като се използва информацията която самата програма генерира докато се изпълнява.
Процесор	Основна част от компютъра.
Разработчик	Специалисти, които се занимават с анализиране на изискванията, моделиране на бизнес процесите и проектиране на софтуера, реализация на функционалността, изпитване и поддръжка
Рекурсия	Начин да се определи нещо като част от компютърна програма чрез обръщане към себе си
Система	Множество от обекти и връзки между тях, които се разглеждат като едно цяло
Скритото разпределение	Вид странициране на данните
Сортиране	Подреждане на елементе в определен ред
Структури от данни	Начин на съхранение на данни в програмирането, така че да може да бъде използвано ефективно
Сървър	Компютърна система, която предоставя услуги към други компютърни системи, наречени клиенти
Тестване	Етап от разработката на софтуер
Уеб навигацията	Позволява да се установи положението в уеб страница
Уеб форма	Позволява на уеб потребител да въвежда данни, които обикновено се изпращат към уеб сървър.

Филтър от тип линкове	Набор от уеб линкове, организирани по определен начин
Шаблон за проектиране	Решение на проблем в софтуерният дизайн.
Явно разпределение	Вид странициране на данните

Литературни източници

1. **Googling large results –**
   http://www.preciseja_va.com/book/chapter1/googling_results.html

2. **Large Data Sets -**
   http://weblogs.macromedia.com/mchotin/archives/2004/03/large_data_sets.cfm

3. **Larman, Craig - Applying UML And Patterns -**
   http://www.amazon.com/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0131489062

4. **J2EE design decisions –**
   http://www.javaworld.com/javaworld/jw-01-2006/jw-0130-pojo.html

5. **Design Patterns for Building Flexible and Maintainable J2EE Applications –**
   http://java.sun.com/developer/technicalArticles/J2EE/despat/

6. **A High-Performance Application Data Environment for Large-Scale Scientific Computations -**
   http://www.ece.northwestern.edu/~choudhar/publications/pdf/SheLia03A.pdf

7. **Improve the quality of your J2EE-based projects –**
   http://www.javaworld.com/javaworld/jw-01-2005/jw-0110-quality.html

8. **Improve the usability of search-results pages -**
   http://www.javaworld.com/javaworld/jw-01-2006/jw-0123-usability.html

9. **Concepts -**
   http://www.oracle.com/technology/sample_code/tutorials/vsm1.3/patterns/concepts.htm

10. **Core J2EE Patterns - Transfer Object -**
    http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

11. **Model-view-controller -** http://en.wikipedia.org/wiki/Model-view-controller

12. **Model-View-Controller Pattern -**
    http://www.enode.com/x/markup/tutorial/mvc.html

13. **Model-View-Controller –**
    http://java.sun.com/blueprints/patterns/MVC.html

14. **Value List Handler –**
    http://java.sun.com/blueprints/patterns/ValueListHandler.html

15. **Fast Lane Reader –**
http://java.sun.com/blueprints/patterns/FastLaneReader-detailed.html

16. **Fast Lane Reader –**
http://java.sun.com/blueprints/patterns/FastLaneReader.html

17. **Handling Large Database Result Sets** –
http://wldj.sys-con.com/read/45563.htm

18. **Page-By-Page Iterator Pattern –**
ftp://ftp.wifo.uni-mannheim.de/pub/PEOPLE/korthaus/DP1-02.pdf

19. **J2EE Design Pattern Samples -**
http://www.oracle.com/technology/sample_code/tech/java/j2ee/designpattern/index.html

20. **Design Patterns for Optimizing the Performance of J2EE Applications -**
http://jdj.sys-con.com/read/36683.htm

21. **Creating a Framework - J2EE pattern frameworks provide template for flexible and modular architecture –**
http://websphere.sys-con.com/read/43487.htm

22. **Design Patterns and Frameworks –**
http://docs.sun.com/source/816-4337/03_design_issues.html

23. **Best practices to improve performance using Patterns in J2EE -**
http://www.precisejava.com/javaperf/j2ee/Patterns.htm

24. **Design patterns make for better J2EE apps -**
http://www.javaworld.com/javaworld/jw-06-2002/jw-0607-j2eepattern.html

25. **Performance Planning for Managers -**
http://www.onjava.com/pub/a/onjava/2001/02/22/optimization.html

26. **Java Platform Performance: Strategies and Tactics -**
http://www.amazon.com/Java-Platform-Performance-Strategies-Tactics/dp/0201709694

27. **Search engine optimization -**
http://en.wikipedia.org/wiki/Search_engine_optimization

28. **Best Practices to improve performance -**
http://www.precisejava.com/javaperf/j2ee/Servlets.htm#Servlets105

29. **Simplify & Sort for Better Searches –**
http://www.smartisans.com/articles/web_search.aspx

30. Random House Webster's College Dictionary –
http://www.randomhouse.com/features/rhwebsters/