Ковалевский Владимир Александрович
Руководитель отдела разработки
компании «Миланор» (WMJ.RU)
email:ikik1@yandex.ru

Abstract

In the modern world of internet and ***web-applications*** *there are a lot of typical solutions for your brand new web-site, but still if you want to have a serious internet business with a great traffic load and huge page views counters, you have to collaborate with a software development team, if it is so and you've made the decision, text below is for you and for your team.*

It is very important to get together three parts in order to draw a good and fast web-application. Here they are: simple and flexible architecture design, interactive cache system, high-performed database with not complicated queries. My idea is not of any original kind, these three points are spread into a complex project and experience of creating fast and interactive web-application (talking about 100.000 – 1.000.000 unique users generating up to 700.000 – 6.000.000 views, according to our stress tests[1]).

So first of all we've hit the design and choosing right level of abstraction was very important, because at this stage we have to decide how deep our database structure will be "denormalized ", and how much CPU time will be lost forming and counting on an inheritance, object mapping, abstraction and other "perfect" object oriented design features.

Next we have to draw a simple cache system with objects and queries support. It is very important to determine how to cache queries, how to store there an objects and of course how to organize links between objects and queries. OR mapping principals helped us to clarify the situation and had connected our "perfect" object model to a data storage.

The last step was to make fast database architecture. It was simple: well-organized automated archive and consolidated tables (that include data from many others) solved the problem.

The experience of our team gives a key how build a high-performed web system; design production, cache structure and data storage concepts are shown after this small abstract part.

Keywords*: Web-application; Cache system; Design concept; High performance; Software development.*

[1] Web server – HP Proliant 3.2 Ghz (Dual Core), 4GB RAM; SQL Server HP Proliant 2.7 Ghz (Dual Core), 8 GB RAM.

Архитектура высокопроизводительных web-приложений.

Часть 1. Дизайн.

В данной части мы не рассматриваем конкретные примеры дизайна, а даём общие рекомендации и строим концепцию того, как проектировать систему так, чтобы она была быстрой, расширяемой и немного поддерживаемой.

Уровни абстракции и основные шаблоны проектирования для производительных web - приложений.

Очевидно, что любое web-приложение в конечном итоге, независимо от клиентской и серверной частей является парой запрос-ответ, обмен которыми происходит по протоколу HTTP, в заголовке которого передаются некоторые переменные, а в теле данные. Т.е. в данном случае целесообразно ввести некоторое абстрактное понятие Контент (слово пришло из сферы деятельности контент-менеджеров) – то с чем работает наше приложение, отправляя запрос или получая ответ от сервера.

Два других понятия, которые мы введём, будут характеризовать позицию или координаты нашего Контента относительно всего объёма данных, с которым работает наше приложение – Рубрикатор и Группа. Эти понятия представляют собой две древовидные структуры данных. Таким образом, Контент всегда «располагается» в некотором Рубрикаторе и входит в абстрактную Группу (рисунок 1).

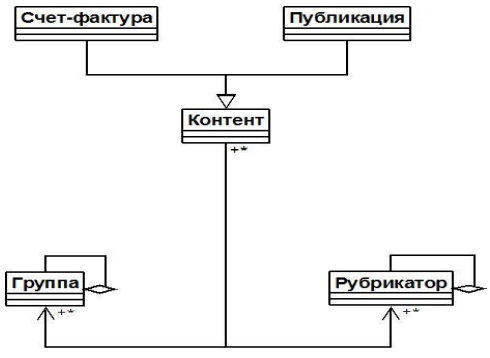

Рисунок 1 (Положение Контента)

Эти на первый взгляд общие или размытые понятия могут иметь явное практическое применение, например с их помощью мы можем осуществлять простую навигацию в нашем приложении:

http://домен.ru/рубрикатор1/рубрикатор2/ключевоесловo/группа1/группа2/идентификатор_контента/параметры.

Необходимость введения двух древовидных структур объясняется просто, на примерах:

- Фотография (Контент), находится в разделе Автомобили (Рубрикатор) и входит в альбом Мой Ламборджини (Группа).
- Платёжное поручение (Контент), находится в Бухгалтерии(Рубрикатор) в офисе Тверская(Группа).

Можно обойтись одной структурой, но работать с одной структурой будет сложнее и медленней, чем с двумя, так как увеличится количество уровней вложенности, а следовательно, процессорное время и не будет явного разделения сущностей (всё будет в одной куче). Можно вводить более двух структур, но поддержка и построение такой системы будет более громоздким и сложным.

Ключевым признаком производительности в данном подходе является определение типа связей: Контент-Группа, Контент-Рубрикатор, если связи определены, как многие-ко-многим – мы теряем в производительности из-за неоднозначности положения Контента, но получаем гибкость при определении различных типов Контента, например: публикация, платёжное поручение, видео, могут находиться в нескольких рубрикаторах, и возникает проблема выбора из какого именно Рубрикатора получать один и тот же Контент и скорее всего, как следствие, будет написан лишний код и станет ощутима потеря производительности системы в целом. Ограничивая себя связями один-ко-многим (Рубрикатор-Контент, Группа-Контент) мы выигрываем в производительности, но теряем гибкость системы в целом, вводя понятие уникальности Контента. Какой тип связей использовать выбор за вами.

Кроме определения координат Контента, нам хотелось бы его описать – введём Типизацию и будем использовать данное понятие, выражаясь языком разработчика, как мета информацию для наших объектов, наследников Контента.

Введём в нашу систему ещё одно понятие, важное для производительности – Статистика Контента. Часто для простых страниц и даже для сложных, нам не надо иметь дела с самим Контентом или его массивом, сам объект в том или ином случае использования, содержит массу ненужной в данный момент информации, требующей время со стороны процессоров наших серверов. Будем собирать необходимые нам данные (количество правильных ответов, анонсы, итог по расчетному периоду, количество комментариев, число сотрудников в компании) в некоторые статистические сущности, которые будут располагаться в системе по тем же принципам и законам, что и сам Контент. Таким образом, нам достаточно обратиться к Статистике Контента один раз и получить необходимый ответ, чем опрашивать всех участников нашего запроса (весь массив Контента).

Подводя итог, мы можем сказать, что построили для нашей «абстрактной команды» некую среду, в которой есть все необходимые для построения и развития архитектуры термины: Контент, координаты Контента (Рубрикатор, Группа), Типизация, Статистика Контента.

Хотелось бы отметить, что данная, достаточно общая концепция имеет конкретное практическое применение:

- Системы публикаций сайта.
- CMS системы.
- Платёжные сервисы.
- Интерактивные сервисы (игры, форумы, блоги, соц. сети) и т.д.

А также является залогом производительности веб-приложения, исключая большинство ошибок проектирования, которые могут сказаться на его быстродействии.

Проблемы безопасности.

Всем прекрасно известны два метода: аутентификация и авторизация, для защиты от несанкционированного доступа, проверки подлинности и назначения ролей. Существует огромное количество модулей, встроенных в веб-серверы, обеспечивающих данный механизм. Мы сознательно не будем рассматривать данный вопрос с этой точки зрения, а обратимся к построению политик безопасности и разграничению доступа к Контенту с точки зрения сценариев его использования.

При проектировании систем мы часто сталкиваемся с проблемой повторяющихся типовых операций, накладывания одних и тех же фильтров, определения правил взаимодействия между объектами. В общем случае все эти действия можно определить, как политики, например:

- Показывать только актуальный Контент.
- Показывать скрытый, удалённый Контент.
- Показывать Контент, только его владельца.

В последнем примере мы видим взаимодействие политики и роли. Таким образом, говоря языком разработчика, мы можем сказать, что каждый объект наследник от Контента обладает теми или иными политиками безопасности и в общем случае данные политики зависят от Роли пользователя в системе.

На практике, такой подход отражается в создании определённого модуля, «Менеджера» - его основной функцией будет являться регистрация новых политик и применение к запросам уже зарегистрированных.

Системная архитектура глазами администратора

Содержание данного раздела, будет носить скорее практический оттенок, с целью обосновать и подчеркнуть важность Администратора приложения в рамках проблем отказоустойчивости и быстродействия.

Для обеспечения быстродействия и стабильности нашего приложения на нескольких серверах нам понадобится Администратор, но не системный, а Администратор приложения.

Простой системный Администратор, мало, что может сделать при нестабильной работе приложения на больших нагрузках. Максимум, чего он может добиться – это логирование ошибок и наращивание мощности. В случае если такой специалист обладает знаниями об архитектуре приложения и, которому предоставлена возможность управлять его настройками, то он может обеспечить требуемые параметры простой оптимизацией архитектуры, настройками распределения памяти и процессорного времени. Например:

- Оптимизировать структуру рубрикатора.
- Организовать перенос неактуальных данных в архив.
- Настроить распределение памяти для разных модулей приложения.
- Управлять политиками безопасности.
- Определять уязвимые места и определять ошибки разработчиков и т.д.

Именно расширение круга интересов Администратора, даёт ему больше возможностей для того, чтобы обеспечивать отказоустойчивость и производительность системы.

Сформулируем для примера несколько общих, но крайне важных для нашей системы требований, которые могут быть предоставлены Администратором и включены в проект:

- Модульность – система (приложение) должно состоять из подсистем (модулей), таким образом, чтобы выход из строя одного из них не привёл к отказу системы в целом и нарушениям в работе ядра.
- Утечка ресурсов – при стабильной нагрузке и постоянном среднесуточном трафике, система (приложение) должна занимать неизменяемый или меньший объём оперативной памяти и процессорного времени.
- Диагностика – система (приложение) должна предоставлять набор средств, для диагностики и управления.

Часть 2. Кэширование.

В данной части мы затронем вопрос о буферизации данных, рассмотрим различные уровни кэширования от хранилища данных до html вёрстки.

Три уровня кэширования.

Представим, что мы построили некоторый образец нашей системы. Пусть это будет простая система публикаций – некий информационный портал. Отдадим наше приложение Администратору и получим его обратно с диагнозом – неудовлетворительные стресс тесты. Действительно, разбирая графики, мы видим полную загрузку CPU хранилища данных и «простой», практически не загруженный процессор серверов приложения.

Решение очевидно – используйте кэширование, но какое, как, и когда оно уместно, а когда нет?

Для того чтобы понять и разобраться в данной проблеме, а главное предложить разработчикам адекватное решение, давайте мысленно разделим данные, которыми оперирует наше приложение, на две части: статичные (редко обновляемые) и интерактивные (требующие обновления в течение времени – «интерактива»). А с другой стороны введём категорию их представлений: близко к интерфейсу (часто запрашиваемые) и далеко от интерфейса (редко запрашиваемые).

Рассмотрим пересечение этих двух, вымышленных категорий:

- Статичные данные, близкие к интерфейсу (Новости на сайте).
- Статичные данные, далекие от интерфейса (Архив данных).
- Интерактив близкий к интерфейсу (Сервисы, форум, блог).
- Интерактив далёкий от интерфейса (Статистика, рейтинги).

Из представленного пересечения видно, что в некоторых случаях кэширование должно выполняться максимально близко к веб-серверу – кэшируется сама вёрстка (HTML код страницы), например Новости. В другом случае, кэшировать лучше объекты, там же изменять их. Или вовсе кэшировать только результаты (ключи запроса), а объекты каждый раз брать из хранилища. Мы видим, что сценариев может быть огромное количество, но в каждом случае мы используем тот или иной кэш: Кэш с вёрсткой, кэш объектов или кэш ключей.

Объект, говоря метафорами, совершает путешествие (Рисунок 2) в нашей системе из хранилища данных, через уровни кэша, к пользовательскому интерфейсу и обратно, тем самым распределяя нагрузку между серверами приложения и хранилищем данных.

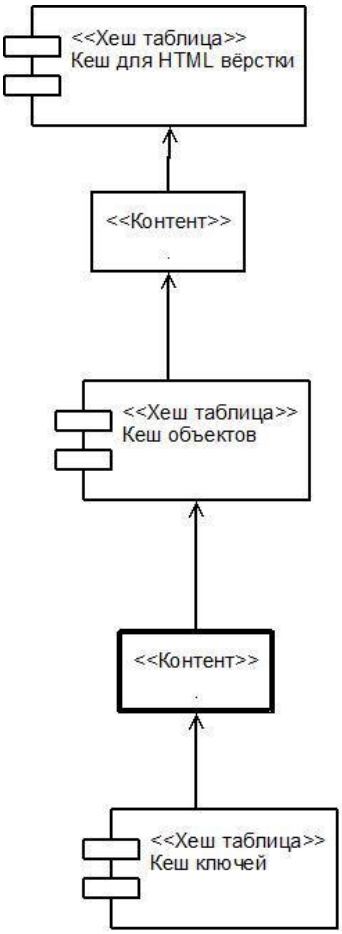

Рисунок 2 (Объект в кэше)

Представление объекта в кэше.

Предположим, что основная нагрузка нашего приложения придётся на кэш объектов, которые мы будем получать из хранилища по ключам запроса, который в свою очередь будет находиться в кэше ключей.

В общем случае кэш объектов может содержать любые объекты, любого типа и тогда вычислений, связанных с определением типа и приведением к типу не избежать, а также использование объекта внутри ядра системы будет крайне затруднительным, мы попросту не знаем, с чем работаем.

Пусть кэш будет однородным, т.е. все объекты в кэше имплементируют некий общий интерфейс, тогда многих проблем, возможно, мы избежим, так как мы заранее договорились с определением взаимодействия объектов с кэшем и ядром.

Это простое, на мой взгляд, крайне важное правило существенно упрощает понимание системы и её проектирование, а также положительно влияет на наши стресс тесты.

Кэшируем запрос.

Итак, как мы выяснили, все объекты находятся в кэше, и все они имплементируют некий интерфейс. Основная задача этого интерфейса – предоставить уникальный идентификатор объекта для однозначного определения его положения в кэше, таким образом, избегая избыточного сканирование всех объектов.

Пусть результатом нашего произвольного запроса к данным всегда будет некоторый набор ключей, по которому мы сможем получить наши объекты. Тогда, создавая кэш ключей, мы освобождаем систему от обращения к данным в том случае, если запросы одинаковые.

Кэшируем вёрстку.

В некоторых случаях данные расположены настолько близко к интерфейсу и достаточно статичны, что у нас отпадает необходимость в наличии объекта. В таких случаях мы будем кэшировать результат http запроса или какую-то его часть.

Это реализовано на многих web-серверах, но если мы введём зависимость кэш вёрстки от кэша объектов, то теоретически мы сможем не контролировать более этот уровень кэширования. Объекты сами будут решать, кэшировать ответ сервера или нет.

Хотелось бы отметить, что синхронизация может происходить не только по вертикали кэша, т.е. от уровня к уровню, но и по горизонтали:

Допустим, около 100.000 пользователей работают с неким сервисом, который предоставляет им личный кабинет с фотоальбомом. Каждый пользователь решает показывать ему фото остальным, или каким пользователям показывать, а каким нет. В нашем случае такая задача сведётся к синхронизации кэшей для разных пользователей, что и будет в наших терминах синхронизацией по горизонтали.

OR mapping.

Для того чтобы перейти от данных к объектам и обратно будем использовать подход OR mapping. Пусть в нашей системе будет некий реестр, связующее звено между уровнями кэша, архитектурной концепцией и данными. Добавим в этот реестр связи между объектами и описание политик безопасности. В результате мы получим мощный инструмент для администрирования системы, управления дизайном системы и «переходник» между запросами к объекту и запросами к данным.

Таким подходом мы обеспечиваем прозрачность приложения для разработчиков, простоту понимания для администратора системы, уменьшаем время разработки и проектирования. Дополнив данный реестр визуальным редактором и генератором кода, мы успешно сможем имплементировать MDA подход к созданию приложения

Важным результатом проделанной нами работы является обобщение всей системы без потери производительности, что, на мой взгляд, ведёт к успеху проекта в целом и нашей команды в частности.

Часть 3. Хранилище данных.

Организуя хранилище данных, в нашем случае следует отходить от нормальных форм, не пренебрегать избыточностью данных и строить максимально простые запросы. В этой части мы сделаем акцент на общих рекомендациях и будем максимально кратки. Итак:

- Если нам надо использовать сложные структуры, агрегированные данные – используем представления и определяем для них класс в нашем реестре.
- Используем запросы к объектам, вместо запроса к данным (SQL92) – они короче, понятнее и занимают меньше места в кэше ключей.
- Используем промежуточные таблицы для агрегации статистики и определяем для них класс в реестре. Обновления данных в таких таблицах делаем асинхронными сервисами.
- Убираем не актуальные данные в архивное хранилище. Чем меньше актуальная база данных, тем система быстрее будет работать.
- Если мы доверяем нашему приложению и файлу реестра, а главное гарантируем целостность данных, то можно отказаться от работы с внешними ключами, они понижают производительность базы.

Используя наш подход и концепцию много времени тратить на организацию данных, не имеет смысла. Основная функция базы – хранилище данных. А всю логику мы уже имплементировали. Однако крайне важно следить за корректностью запросов, при ошибочной и слишком громоздкой архитектуре система может очень сильно понизить производительность хранилища. Перечислим узкие места, на которые стоит обратить внимание:

- Связи между объектами.
- Рекурсия.
- Вычисляемые свойства.
- Низкие уровни кэширования.
- Открытые запросы к объектам (запросы без условия и политик).

Вывод

Итак, мы ввели такие термины как: Контент, Рубрикатор, Группа, связи между этими понятиями, типы Контента, статистика Контента, политики безопасности, Администратор приложения.

В дополнение к вышеперечисленному мы знаем структуру кэша и правила организации данных.

Несомненно, концепция, изложенная в данном докладе, может явиться некоторым единым документом, определением для нашей системы и, опираясь на это определение (конечно, расширенное и дополненное), мы можем приступать к разработке.