

Держим 11к рек/сек

Обычно я пишу технические статьи на хабр, но в связи с последними событиями заиграли патриотические нотки, и я решил сделать исключение. На ДОУ частенько возникали ереси споры про унылость проектов в аутсорсе и безысходность бытия. Мне с этим всегда везло, и я попадал в более-менее интересные.

В большинстве случаев бизнес идеи продуктов и архитектура проектов закрыты [NDA](#), поэтому на просторах инета трудно найти что-то интересное и новое, касательно архитектуры каких-то готовых работающих решений. К счастью, мне удалось уговорить нашего заказчика дать добро на разглашение информации о структуре нашего проекта (за исключением имени клиента нашего заказчика =)).

Поэтому представляю Вашему вниманию пост об архитектуре одного рекламного движка.

Около 8 месяцев назад к одному из клиентов [Cogniance](#) обратилась одна довольно большая и известная компания. Назовем ее "X". У компании "X" уже довольно давно существует бесплатное мобильное приложение с огромной пользовательской базой (на текущий момент - 85 млн активных пользователей). Проблема "X" была в том, что они никак не монетизировали приложение. Ну и вполне очевидно, что наступил момент, когда появилась необходимость получения прибыли. Какой самый простой и очевидный способ заработать на приложении? Правильно - баннеры. И, как это часто бывает, "X" захотел **свое решение** со своим блекджеком и ..., ну вы поняли.

Требования

Для нас как исполнителей требование выглядело приблизительно так:

Нужен UI, где можно сконфигурировать рекламные компании. Например, показывать баннер туфель девушкам, которые увлекаются музыкой, старше 14 лет в Калифорнии не чаще чем раз в день. "X" в свою очередь дергает наши сервера с клиентских приложений, запрашивая рекламу для конкретного пользователя, передавая всю имеющуюся информацию для [таргетинга](#). + более-менее real time репорты с инфой о том - кому, сколько и какой рекламы было отдано.

Очевидно, что это десятки страниц спецификации, сжатые в 3 предложения. Но вот отдаленно все выглядело как-то так. Отдельные требования были касательно производительности системы:

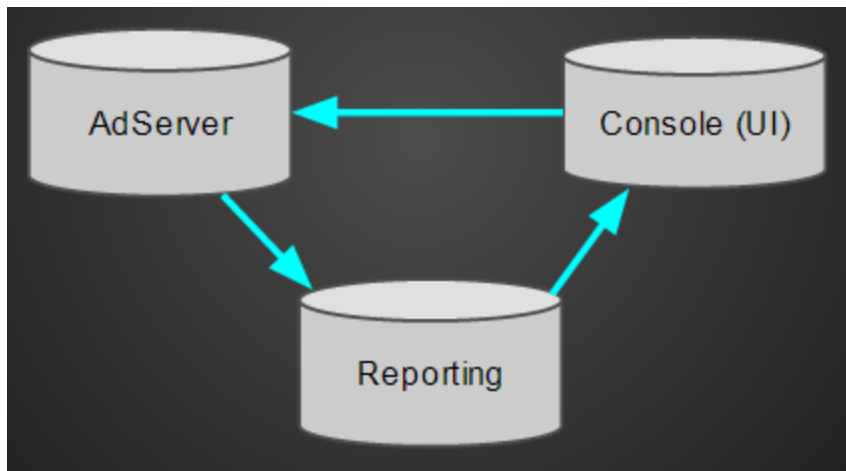
Request rate : 1000 req/sec
Protocol : https

Response time : 99% < 100ms.
No downtime
Hosting: Amazon

После 3-х месяцев разработки требования по нагрузке в 1000 рек/сек изменились (а как же без этого =)), и появилась цифра в 11000 рек/сек. Из-за чего нам в последствии пришлось немного сменить вектор развития продукта.

Архитектура

Из спеки сразу стало ясным, что систему можно условно разделить на 3 подсистемы, что мы собственно и сделали:
UI (CRUD + интеграция с сервисами клиента), AdServer (high-load), Reporting (big data).



Разделение на модули на начальном этапе было необходимо как воздух:

- Во-первых, это позволило сразу вести разработку параллельно;
- Во-вторых, сразу же были максимально снижены зависимости между модулями;
- В-третьих, повышалась отказоустойчивость всей системы в целом. Так как падение какого-то узла никак не влияло на работу других частей системы.

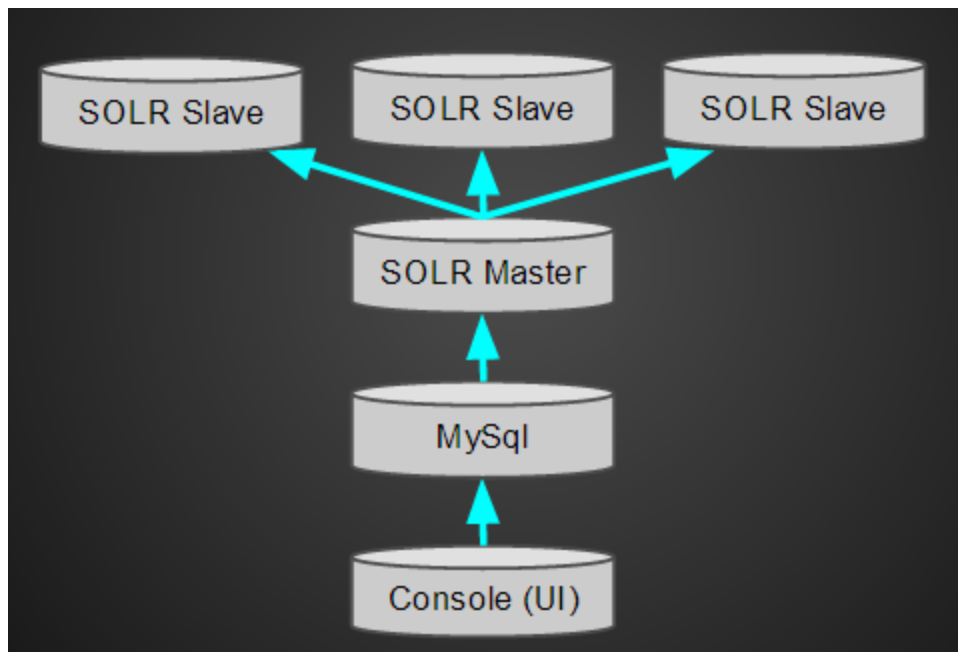
UI

Задача модуля - предоставить удобный, дружелюбный интерфейс для создания рекламной компании.

Техн. стек : js, spring, hibernate, tomcat, mysql.

Детальное описание UI модуля я опущу, так как в нем все довольно банально, и это обычный CRUD модуль + интеграция с сервисами заказчика. Уверен, что 90% проектов делает такое же, только в разных доменных областях.

Одна из важнейших задач, которая решалась на этапе разработки UI модуля, - как соединить UI модуль с AdServer модулем. Дело в том, что возможных решений было очень много. Ну например, это проблему можно было бы решить через RMI, RESTfull API, JMS, обычную master-slave репликацию СУБД, распределенный ehCache, распределенные датагриды и еще +100500 разными способами. Лично я остановил свой выбор на master-slave репликации СУБД, так как это решение не требовало кода и выглядело довольно простым. К сожалению, наш заказчик также обладал технической экспертизой. И после обсуждения предложенных вариантов, имея в багаже схожие запущенные проекты (как основной аргумент - это решение уже есть и оно работает) - клиент настоял на Solr.



Используя Solr, предполагалось убить сразу двух зайцев:

- Во-первых, все нужные для доставки рекламы данные будут локально храниться на каждом из серверов. Таким образом повышаем отказоустойчивость системы и уменьшаем время доступа к данным;
- Во-вторых, всю сложную логику по подбору рекламы для конкретного пользователя можно будет вынести в Solr;

AdServer

Задача модуля - на основе входящего запроса и его параметров подбирать наиболее релевантную рекламу для пользователя в конкретный момент времени.

Техн. стек : spring, tomcat, Solr, Redis.

Solr

После 3-х месяцев разработки, базовый функционал был реализован и мы начали первое нагрузочное тестирование. Результат оказался плачевным. Один [c1.xLarge](#) сервер смог обрабатывать всего 200 реквестов в секунду при времени ответа приближающемся к 100мс.

Быстрый профайлинг сразу позволил обнаружить узкое место - SOLR. Все дело в том, что SOLR - это http сервер, поэтому на каждый реквест от пользователя приходилось делать http запрос на localhost к солру. Что, конечно же, не могло быть дешево. К сожалению, существующий Embedded Solr работал еще хуже. Кеш на уровне приложения помог поднять рейт к 400 рек/сек. Но нас это тоже не устраивало, так как, помимо всего прочего, мы вплотную подошли к требованию времени ответа сервера $99\% < 100\text{мс}$. Это обстоятельство сужало пространство для маневра при добавлении нового функционала, а также накладывало риски в будущем в случае роста индекса:

```
Index size < 1 Gb - response time 20-100 ms  
Index size < 100 Gb - response time 1-10 sec
```

В конце концов было принято решение отказаться от выборки из солра. И вся логика выборки переключалась на уровень приложения. SOLR, тем не менее, остался - исключительно как хранилище delivery индекса. Весь код работы с солром сводился к следующему :

```
volatile DeliveryData cache;  
Cron Job:  
DeliveryData tempCache = loadAllDataFromSolr();  
cache = tempCache;
```

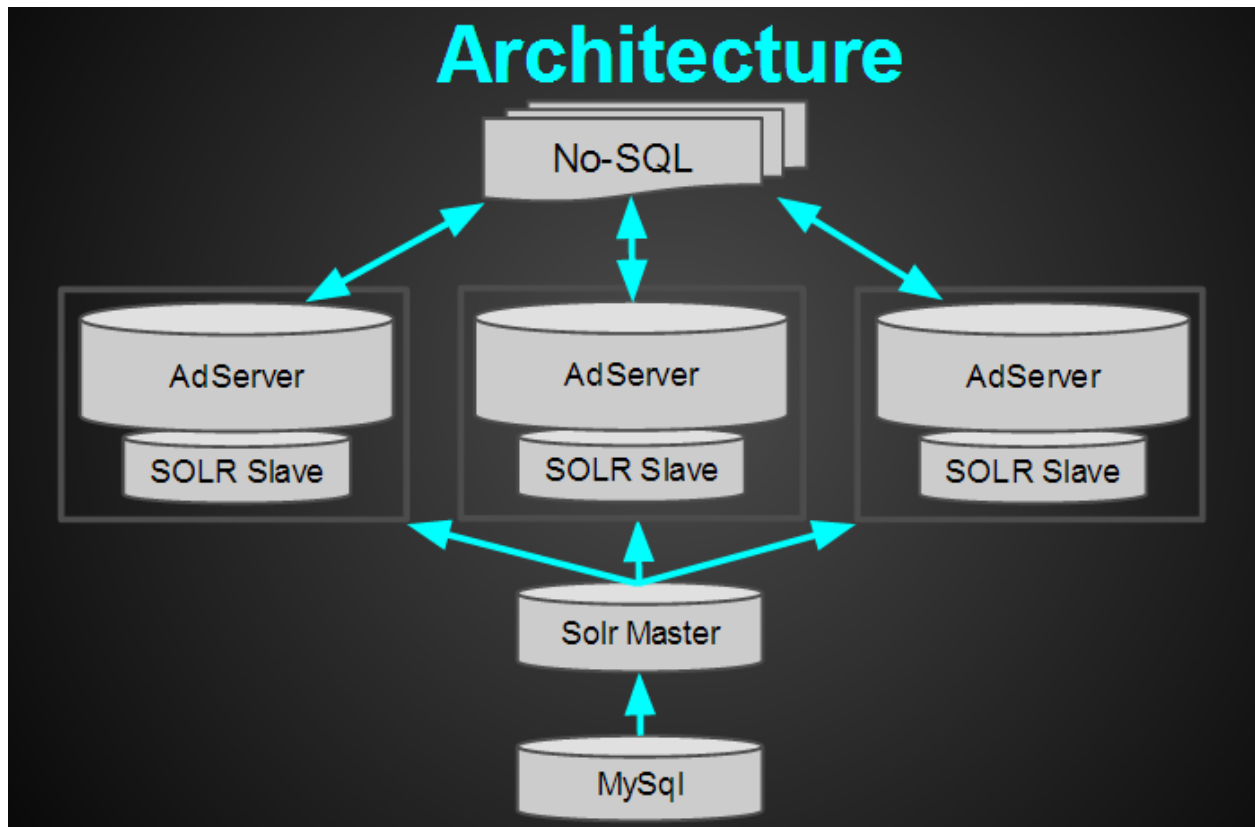
Как результат, скорость работы одного сервера выросла до 600 рек/сек с временем ответа ~15ms при [LA 4](#).

No-sql

В мире веб-рекламы существует такая фишка, как [frequency capping](#). Если коротко - это лимит показов одной и той же рекламы для одного и того же пользователя. Например, если вы не хотите, чтобы пользователь видел ваш баннер чаще, чем раз в день, или если вы хотите, чтобы уже после показанного баннера показывался другой, например, со скидкой, в случае, если после показа первого баннера вы не получили клик. Обычно в веб мире эта проблема решается через куки. И подобная инфа хранится на стороне клиента. "X" почему-то не захотел реализовывать куку на стороне мобильного клиента и переложил эту задачу на нас.

Для нас же это означало хранить фактически всю пользовательскую базу (когда, кому и какая реклама была отдана) и иметь быстрый к ней доступ во время каждого реквеста от пользователя. Нужно было решение которое бы хранило общее состояние

для всех деливери серверов - с максимально коротким временем ответа и максимально возможной производительностью. Идеальная ситуация для no-sql решения.



DynamoDB

Теперь возникла необходимость выбрать нужное решение. Изначально мы решили попробовать DynamoDB. Предполагалось, что мы снизим стоимость AdOps поддержки + по описанию, решение от амазона выглядело очень привлекательно. Первые нагрузочные тесты показали, что DynamoDB очень далек от идеала. Как говорится, я просто оставляю это здесь:

	Price per month	Put, 95%	Get, 95%	Rec/sec
DynamoDB	58\$	300ms	150ms	50
DynamoDB	580\$	60ms	8ms	780
DynamoDB	5800\$	16ms	8ms	1250
Redis	200\$ (c1.medium)	3ms	<1ms	4000
ElastiCache	600\$ (c1.xlarge)	<1ms	<1ms	10000

Помимо большой стоимости и большого SLA, оказалось, что динамо не масштабируется автоматически. Есть свойство, которое нужно задавать при старте кластера, что при резком росте нагрузки чревато отказом в обслуживании части запросов.

Redis

После основательного гуглинга и опыта работы с некоторыми no-sql решениями в прошлом, мы остановили свой выбор на [Redis](#). Редис показывал просто сказочный средний SLA - 0.2ms в приватной сети амазона. При этом он вполне себе держал нагрузку в 50к get рек/сек на одной c3.xLarge нодe. Ну и наконец, у редиса есть пачка вкуснейших фич - [atomic increments](#), [sets](#), [hashes](#). Каждой из которых мы нашли применение.

Конечно, как и у каждого решения у редиса есть своя темная сторона:

- Все хранится в памяти, диск используется только для бекапа и восстановления данных;
- Кластерное решение еще не готово к продакшену, так как вышло несколько месяцев назад;
- Для масштабируемости придется использовать sharding. А значит закладывать сразу в архитектуру при разработке.
- Использование диска влияет на SLA, так же возможен page swapping, который тоже может увеличить время ответа;
- Возможна потеря данных;

Несмотря на недостатки, редис полностью подходил для всех наших бизнес задач.

Оптимизации

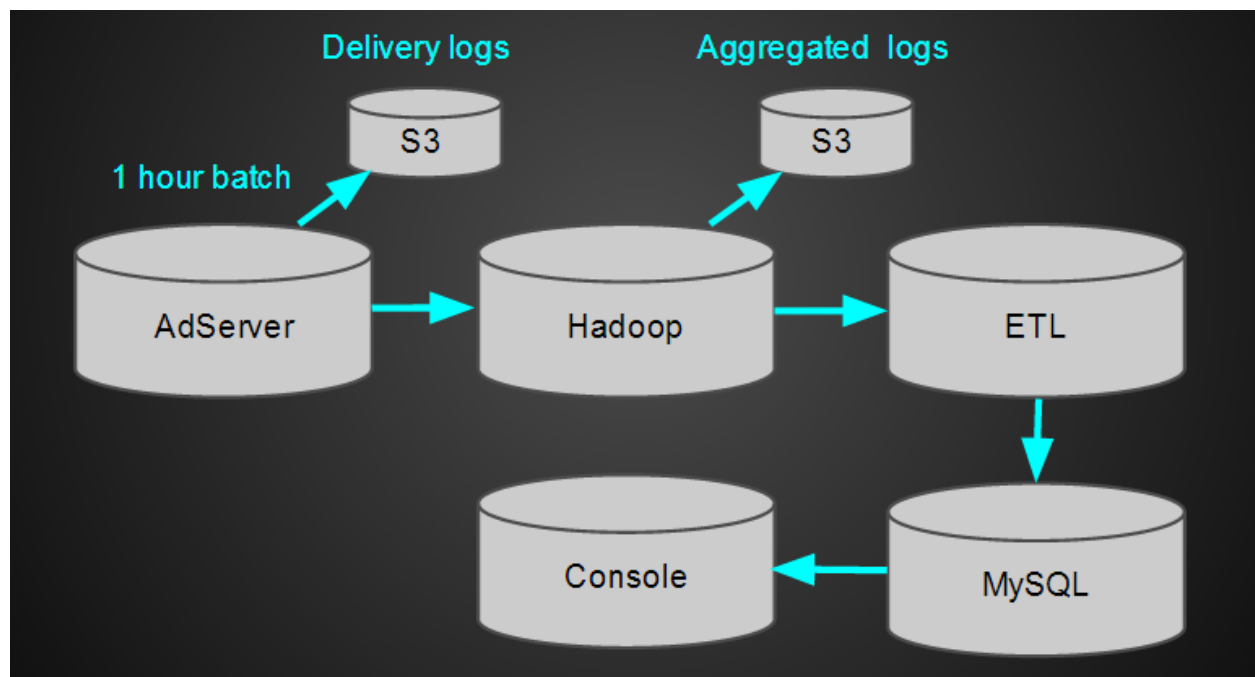
Несмотря на довольно хороший результат в 600 рек/сек и SLA 15ms, было понятно, что можно выжать больше. Оптимизировалось все - начиная от банальных очевидных вещей, о которых стыдно писать, и заканчивая некоторыми алгоритмами. Оптимизация кода в яве - это отдельная тема для поста. Собственно, несколько таких постов я уже написал на хабре. Поэтому не буду повторяться, лишь оставлю на почитать интересующихся - [небольшие трюки](#) и [еще, сколько стоит выделить объект, одна маленькая оптимизация, оптимизируем еще, изменения в String](#).

В результате оптимизаций мы дошли до 1000 рек/сек и SLA 1.2 ms при LA 4 на c3.xLarge. После всех наших стараний узким местом стал редис, который в среднем на 1 пользовательский запрос выполнял 1.5 реквеста по сети, а также использовал синхронизацию при обращении к пулу коннекшенов. На что уходило ~50-60% времени обработки запроса.

Reporting

Понятное дело, что каждый клиент хочет видеть эффективность рекламной компании, видеть свою аудиторию, ее отзывчивость на ту или иную рекламу. Для этого существует reporting модуль. Он раз в час собирает всю информацию с деливери серверов, обрабатывает и отдает в упрощенном виде конечному пользователю.

Техн. стек: hadoop ([cascading](#)), mySql.



Каждый ответ сервера клиенту логируется. Во время первого же тестирования было обнаружено, что генерится довольно большое количество логов. Одна запись в лог

была в среднем ~700 байт. Следовательно, при нагрузке в 11к рек/сек в секунду генерировалось около 7мб логов в секунду или около 25 ГБ в час. Структура лога :

```
{
  "uid": "test",
  "platform": "android",
  "app": "xxx",
  "ts": 1375952275223,
  "adId": 1,
  "education": "Some-Highschool-or-less",
  "type": "new",
  "sh": 1280,
  "appver": "6.4.34",
  "country": "AU",
  "time": "Sat, 03 August 2013 10:30:39 +0200",
  "deviceGroup": 7,
  "rid": "fc389d966438478e9554ed15d27713f51",
  "responseCode": 200,
  "event": "ad",
  "device": "N95",
  "sw": 768,
  "ageGroup": "18-24",
  "preferences": ["beer", "girls"]
}
```

Hadoop

Было ясно, что при генерации 25 ГБ логов в час мы никак не можем напрямую хранить их в базе, так как это стоило бы не дешево. Необходимо было как-то сократить объемы. К счастью, заказчик точно знал, какого рода репорты ему нужны. Поэтому из имеющихся полей лога :

```
device, os, osVer, sreenWidth, screenHeight, country, region,
city, carrier, advertisingId, preferences, gender, age, income,
sector, company, language, ...
```

мы определили набор таблиц необходимых клиенту, например :

Geo table:

```
Country, City, Region, CampaignId, Date, counters;
```

Device table:

```
Device, Carrier, Platform, CampaignId, Date, counters;
```

Uniques table:

```
CampaignId, UID
```

...

Нужно было решение, которое бы легко масштабировалось в случае увеличения трафика или в случае, если кластер хадупа упал и не был доступен несколько часов, чтобы мы могли быстро обработать накопившиеся логи. Мы остановили выбор на Hadoop. По большей части из-за того, что у нас уже был опыт работы с ним, а времени на эксперименты с другими решениями не было.

Да, агрегация входных данных по определенной совокупности полей приводит к потере большей части информации (после агрегации нельзя узнать например, сколько человек с айфоном посмотрели рекламу в Сан-Франциско), но бизнес задачу решение решало, а значит устраивало и нас (хотя кое-какие сырые данные мы все же храним =)). В результате этого подхода объем данных удалось сократить на 3 порядка, а именно к 40ГБ данных в месяц. Которые ну никак не страшны даже самым хиленьким СУБД.

У внимательного читателя наверняка возник вопрос: “А зачем использовать хадуп?”. Вопрос, на самом деле, очень правильный и интересный. Дело в том, что объем в 25 ГБ в час - это совсем не много (для нашей задачи), и даже если надо обработать несколько десятков таких файлов - написанный на коленке агрегатор справится с этой задачей очень быстро. Но в нашем случае хадуп выполняет не только агрегацию, но и определенную валидацию, которая является довольно ресурсоемкой (к сожалению эта часть закрыта NDA). Собственно из-за этой валидации нам и необходимо было решение, которое легко можно было масштабировать в случае необходимости, что на коленке реализовать уже гораздо сложнее.

Заключение

Проект, конечно же, не получился идеальным (хотя очень хотелось), но свою задачу мы выполнили. Уже около 2 месяцев проект успешно запущен в рабочей среде.

Сейчас, оглядываясь назад, если бы я был в тех же условиях сегодня, я бы однозначно настоял на полном избавлении от Solr.. Solr для нас оказался явно overengineering solution. Не имея в проекте солр, мы вполне смогли бы избавиться и от Tomcat, банально заменив его [HttpServer](#), что упростило бы процесс деплоя и написания интеграционных тестов. Ну и касательно репортинга - я бы уже наверное посмотрел в сторону более перспективных технологий, а именно - Spark, Storm, Redshift. У них, конечно, своя специфика, но решить нашу проблему можно и на них. И вполне возможно, что получилось бы дешевле.

Спасибо всем, кто дочитал. Буду рад любой конструктивной критике. Надеюсь пост вам понравился. Piese.