Долгий путь к 100k RPS

Как мы хотели обыграть MS SQL на его же поле

Anatoly Zhmur
Senior Technical Architect
Broadridge Financial Solution

О чем мы не будет говорить

- Latency focused soft real-time
- Kernel bypass (DPDK etc)
- Infiniband/RDMA/RoCE
- Забирание ядер (core) у ядра (kernel)
- Отключение GC
- Собственная модель асинхронности

Наши ограничения

- Без нативного кода
- Чуть-чуть unsafe
- Async френдли
- На публичных библиотеках насколько это возможно

Наша прикладная задача

Наши данные

- Справочная информация по ценным бумагам (по-английский Security)
- Долгоживущие и редко меняющиеся объекты (1-3% ежедневно)
- Самый большой поставщик ~10 млн.
- Security (ценная бумага) единственный Aggregation Root
- Между ними много связей
- Десятки селективных индексов и столько же неселективных

Реляционный подход (v1)

- 16 лет продукту
- Объем БД 50-250 GiB с небольшой историей
- Третья нормальная форма
- 450 табличек
- Быстрая выгрузка произвольного набора данных для всех бумаг
- Нормализация изменений справочников и т.п.
- Сходить за одной бумагой целиком ~55 селектов (без учета справочников, они закэшированы)
- MS SQL Server vendor lock-in. Мигрировать на другую БД - нереально.

Документный подход (v2)

- 5 лет продукту
- Быстрая выгрузка информации по любой бумаге целиком (2 селекта)
- Меньше нагрузка на БД (лицензии MS SQL Server это очень дорого)
- Живет так же на бесплатном Postgresql
- Транзакционность на уровне Aggregation Root
- Можно не ограничивать себя так строго схемой
- Денормализованные документы удобны для интеграции с удаленными системами
- Чтобы посмотреть на одно поле бумаги нужно извлечь ее целиком, а так же разжать и десериализовать DOM
- При GET операциях сервер не десериализует содержимое, а просто копирует блоб из БД в Http
- Занимает меньше места на диске (как ни странно!)

Пример денормализованных данных по Security

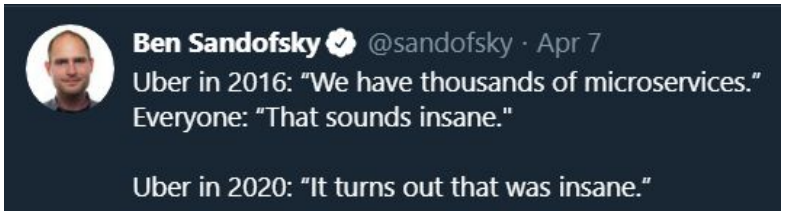
```
  DaqarId: 1487854
Name: TWD10
SecClass: EQ
SecType:
  code: EQTY
  SecMasterID: 50
  Description: Equity
  Multiplier: 1.000
LegalEntity:
  Id:
    SecMasterId: 160225
  Name: MITAC HOLDINGS CORPORATION
  Description: MITAC HOLDINGS CORPORATION
  Country:
    code: TW
    Description: TAIWAN
    CountryIso2Code: TW
    CountryIso3Code: TWN
    Region:
      code: 2
      SecMasterID: 2
      Description: Asia
  IsBankruptcy: !!bool False
  IsAcquired: !!bool False
  LegalEntityID: !!int 160225
  Codes:
  - StartDate: !!timestamp 2019-07-31T00:00:00.0000000Z
    IssuerId: !!long 150010424
    CodeType:
```

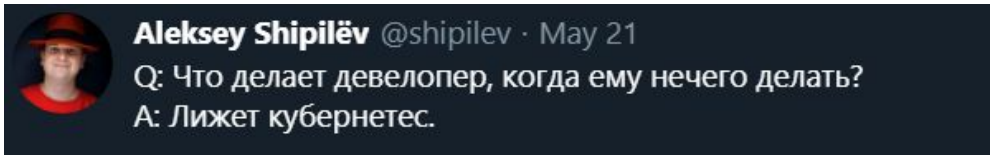
Неожиданная задача (v3)

- Должно быть только дельты каждый день (1-5% бумаг)
- Обычно вендор не может отдать все данные за раз
- Блумберг балк например требует неделю на полную инициализацию, а потом отдает только дельты
- Но у нас хотят все данные каждый день (4 mln / 1k RPS = 1 hour)
- И тут реляционная БД работает сильно лучше

Горизонтальная масштабируемость

- Автоматизация как правило дорогая (и в людях, и в железе) и подходит для больших команд
- Возьмем компанию Uber
- 576 стоек
- По 42 юнита
- На одного IT специалиста 2-4 юнита
- 14 млн поездок в день
- 2.5 минуты 1u сервера на поездку
- Для сравнения 200 секунд компилируется LLVM на 3950x
- https://blog.packagecloud.io/eng/2015/09/15/automacon-infrastructure-as-code-might-be-literally-impossible/
- https://github.com/hjacobs/kubernetes-failure-stories





А сколько вообще можно RPS?

Сколько RPS? (i5-9600K CPU 3.70GHz)

```
[Benchmark(OperationsPerInvoke = Iterations, Baseline = true)]
public void MethodCall()
{
    var k = 1;
    for (int i = 0; i < Iterations; ++i)
    {
        k = Add(k, 1);
    }
}

public int Add(int I1, int I2) => I1 + I2;
```

```
C.Add(Int32, Int32)
    L0000: lea eax, [rdx+r8]
    L0004: ret

C.MethodCall()
    L0000: xor eax, eax
    L0002: xor edx, edx
    L0004: inc eax
    L0006: inc edx
    L0008: cmp edx, 0x64
    L000b: jl short L0004
    L000d: ret
```

А так?

```
[MethodImpl(MethodImplOptions.NoInlining)]
public int AddNoInlining(int I1, int I2) => I1 + I2;
```

```
C.Add(Int32, Int32)
    L0000: lea eax, [rdx+r8]
    L0004: ret

C.MethodCall()
    L0000: push rdi
    L0001: push rsi
    L0002: sub rsp, 0x28
    L0006: mov rsi, rcx
    L0009: xor edx, edx
    L000b: xor edi, edi
    L000d: mov rcx, rsi
    L0010: mov r8d, 1
    L0016: call C.Add(Int32, Int32)
    L001b: mov edx, eax
    L001d: inc edi
    L001f: cmp edi, 0x64
    L0022: jl short L000d
    L0024: add rsp, 0x28
    L0028: pop rsi
    L0029: pop rdi
    L002a: ret
```

Добавим не аллоцирующий ValueTask без авейтера

```
[Benchmark(OperationsPerInvoke = Iterations)]
public void ValueTask()
{
    var k = 1;
    for (int i = 0; i < Iterations; ++i)
    {
        k = AddValueTask(k, 1).GetAwaiter().GetResult();
    }
}
```

```
public ValueTask<int> AddValueTask(int I1, int I2) => new ValueTask<int>(I1 + I2);
```

А теперь с авейтором но без ожидания

```
[Benchmark(OperationsPerInvoke = Iterations)]
public void ValueTaskAsync()
{
    var k = 1;
    for (int i = 0; i < Iterations; ++i)
    {
        k = AddValueTaskAsync(k, 1).GetAwaiter().GetResult();
    }
}
```

```
public async ValueTask<int> AddValueTaskAsync(int I1, int I2) => I1 + I2;
```

А теперь с использованием авейтера

```
[Benchmark(OperationsPerInvoke = Iterations)]
public async Task ValueTaskAsyncAwait()
{
    var k = 1;
    for (int i = 0; i < Iterations; ++i)
    {
        k = await AddValueTaskAsync(k, 1);
    }
}
```

А теперь с yield

```
[Benchmark(OperationsPerInvoke = Iterations)]
public async Task ValueTaskAsyncAwaitYield()
{
    var k = 1;
    for (int i = 0; i < Iterations; ++i)
    {
        k = await AddValueTaskAsyncWithYield(k, 1);
    }
}

public async ValueTask<int> AddValueTaskAsyncWithYield(int I1, int I2)
{
    await Task.Yield();
    return I1 + I2;
}
```

А теперь с yield и Run для клиента и сервера

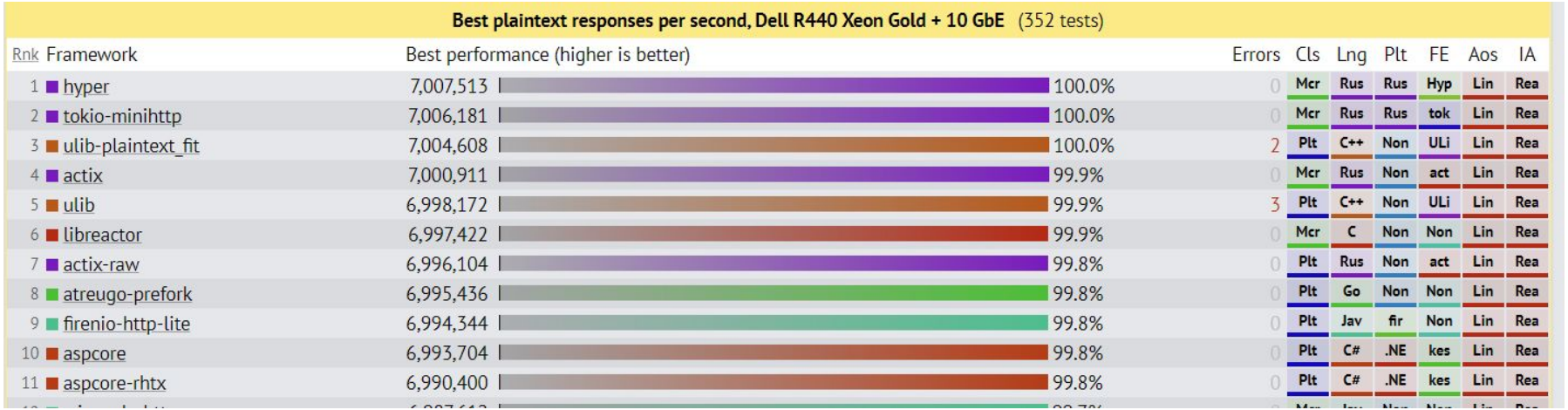
```
public async ValueTask<int> AddValueTaskWithYieldAndRun(int I1, int I2)
{
    await Task.Yield();
    return await Task.Run<int>(() => I1 + I2);
}
```

Результаты

Method	Mean	Error	StdDev	Op/s	Ratio
MethodCall	0.4874 ns	0.0170 ns	0.0159 ns	2,051,524,495.0	1.00
MethodCallNoInlining	1.6204 ns	0.0092 ns	0.0086 ns	617,139,293.3	3.33
ValueTask	6.6717 ns	0.0518 ns	0.0485 ns	149,887,678.0	13.70
ValueTaskAsync	15.4339 ns	0.1037 ns	0.0970 ns	64,792,336.0	31.69
ValueTaskAsyncAwait	15.8982 ns	0.0826 ns	0.0732 ns	62,900,127.0	32.56
ValueTaskAsyncAwaitYield	743.1724 ns	3.6768 ns	3.4393 ns	1,345,582.7	1,526.11
ValueTaskAsyncAwait2Yield	1,393.2782 ns	11.4900 ns	10.7477 ns	717,731.7	2,861.10
SerialDirectConnection	239.9482 ns	2.5524 ns	2.3875 ns	4,167,566.9	492.69

Движок LMDB — особенный чемпион / Леонид Юрьев: https://www.youtube.com/watch?v=5DDEIyfw0RU

Techempower benchmark (Xeon 5120 14/28 cores)

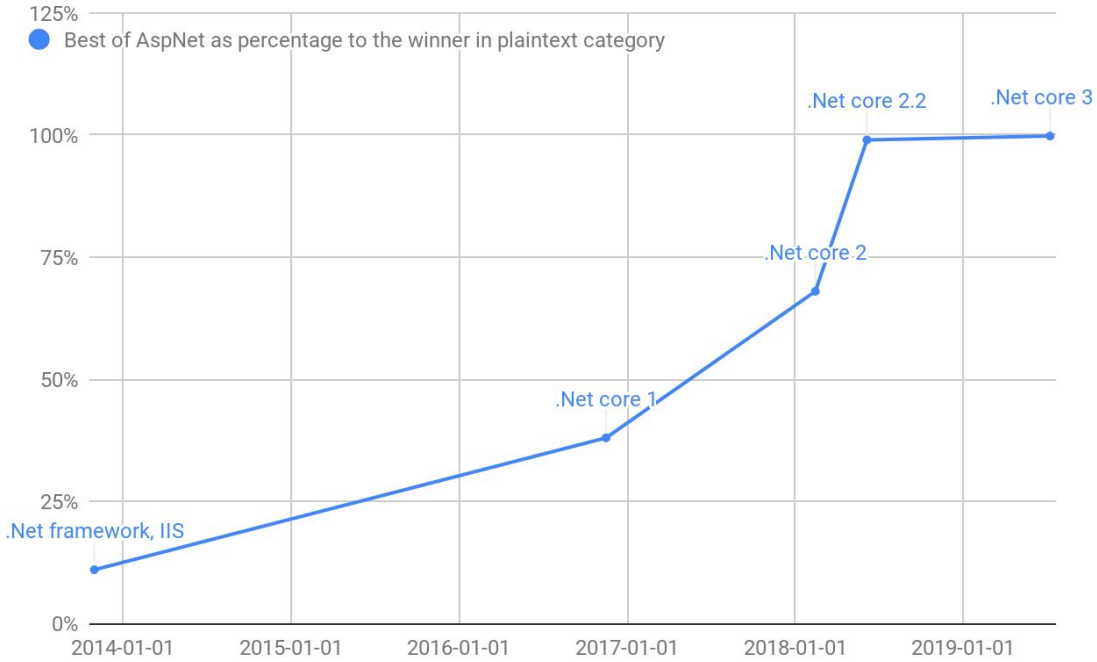


Best plaintext responses per second, Dell R440 Xeon Gold + 10 GbE (352 tests)

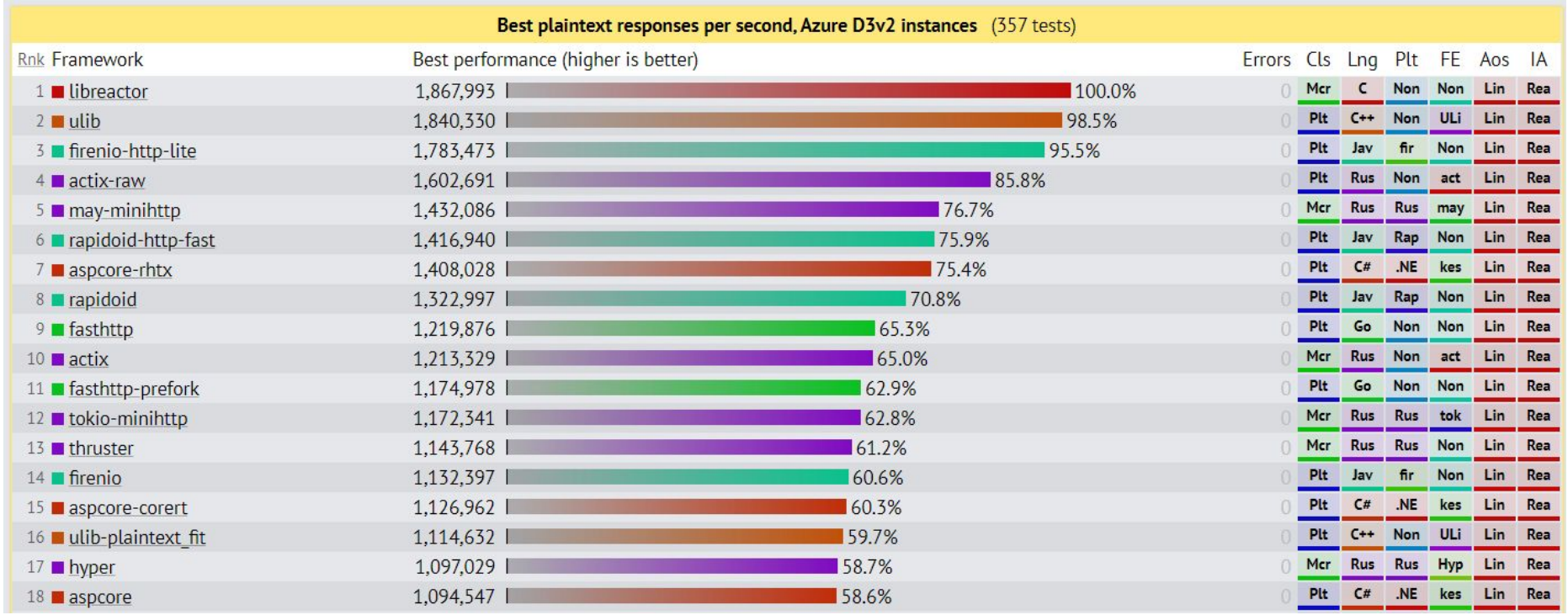
Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	hyper	7,007,513	100.0%	0	Mcr	Rus	Rus	Hyp	Lin	Rea
2	tokio-minihttp	7,006,181	100.0%	0	Mcr	Rus	Rus	tok	Lin	Rea
3	ulib-plaintext_fit	7,004,608	100.0%	2	Plt	C++	Non	ULi	Lin	Rea
4	actix	7,000,911	99.9%	0	Mcr	Rus	Non	act	Lin	Rea
5	ulib	6,998,172	99.9%	3	Plt	C++	Non	ULi	Lin	Rea
6	libreactor	6,997,422	99.9%	0	Mcr	C	Non	Non	Lin	Rea
7	actix-raw	6,996,104	99.8%	0	Plt	Rus	Non	act	Lin	Rea
8	atreugo-prefork	6,995,436	99.8%	0	Plt	Go	Non	Non	Lin	Rea
9	firenio-http-lite	6,994,344	99.8%	0	Plt	Jav	fir	Non	Lin	Rea
10	aspcore	6,993,704	99.8%	0	Plt	C#	.NE	kes	Lin	Rea
11	aspcore-rhtx	6,990,400	99.8%	0	Plt	C#	.NE	kes	Lin	Rea

PS Это 2019 год. В 2020 результаты упали - пелотон опустился до 95%, aspnet core поднялся. Похоже на системные ошибки измерений с мистикой a la microcode update.

История AspNet в Techempower benchmark



In cloud



Best plaintext responses per second, Azure D3v2 instances (357 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	libreactor	1,867,993	100.0%	0	Mcr	C	Non	Non	Lin	Rea
2	ulib	1,840,330	98.5%	0	Plt	C++	Non	ULi	Lin	Rea
3	firenio-http-lite	1,783,473	95.5%	0	Plt	Jav	fir	Non	Lin	Rea
4	actix-raw	1,602,691	85.8%	0	Plt	Rus	Non	act	Lin	Rea
5	may-minihttp	1,432,086	76.7%	0	Mcr	Rus	Rus	may	Lin	Rea
6	rapidoid-http-fast	1,416,940	75.9%	0	Plt	Jav	Rap	Non	Lin	Rea
7	aspcore-rhtx	1,408,028	75.4%	0	Plt	C#	.NE	kes	Lin	Rea
8	rapidoid	1,322,997	70.8%	0	Plt	Jav	Rap	Non	Lin	Rea
9	fasthttp	1,219,876	65.3%	0	Plt	Go	Non	Non	Lin	Rea
10	actix	1,213,329	65.0%	0	Mcr	Rus	Non	act	Lin	Rea
11	fasthttp-prefork	1,174,978	62.9%	0	Plt	Go	Non	Non	Lin	Rea
12	tokio-minihttp	1,172,341	62.8%	0	Mcr	Rus	Rus	tok	Lin	Rea
13	thruster	1,143,768	61.2%	0	Mcr	Rus	Rus	Non	Lin	Rea
14	firenio	1,132,397	60.6%	0	Plt	Jav	fir	Non	Lin	Rea
15	aspcore-corert	1,126,962	60.3%	0	Plt	C#	.NE	kes	Lin	Rea
16	ulib-plaintext_fit	1,114,632	59.7%	0	Plt	C++	Non	ULi	Lin	Rea
17	hyper	1,097,029	58.7%	0	Mcr	Rus	Rus	Hyp	Lin	Rea
18	aspcore	1,094,547	58.6%	0	Plt	C#	.NE	kes	Lin	Rea

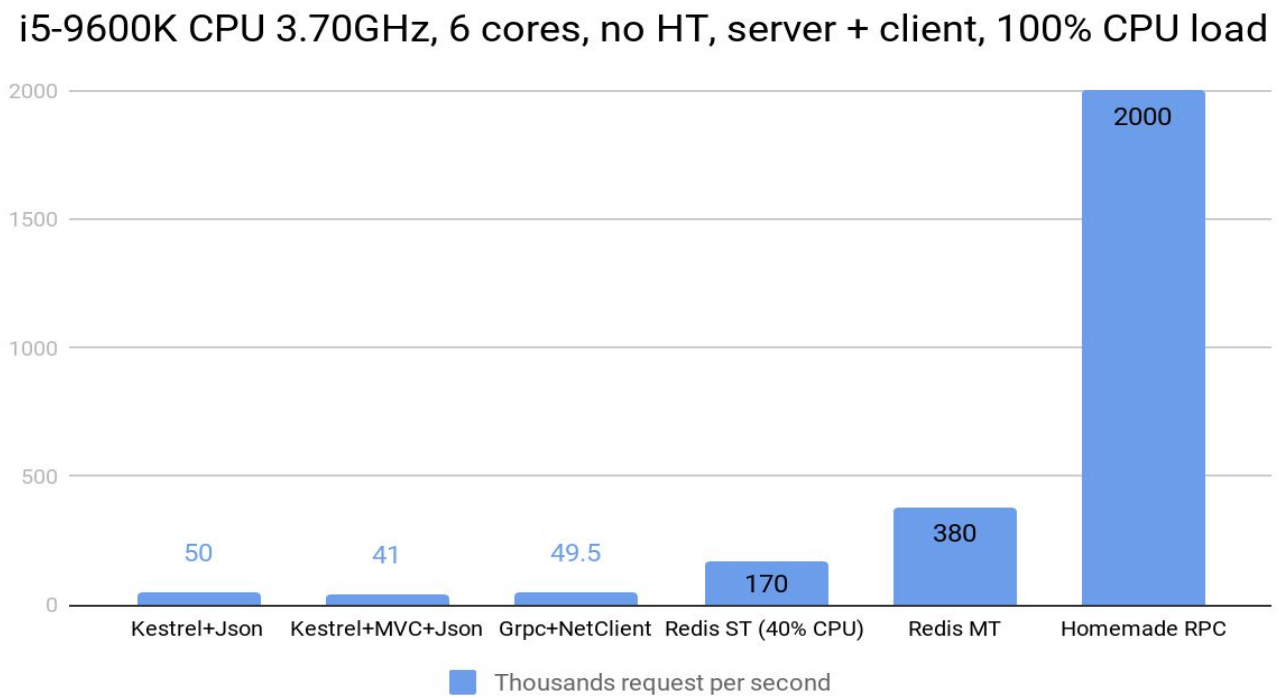
Ах этот пайплайнинг

- Не используется в реальном мире
- head of line blocking
- А так же ломается от reverse proxy
- Заменен мультиплексированием в http2
- Его использовали в данном тесте исключительно для надувательства результатов в буквальном смысле этого слова
- Теперь все упирается в канал 10 Gbit
- https://deavid.wordpress.com/2019/10/06/http-pipelining-is-useless/

Посчитаем что в итоге

- 7mln / 16 = 436k RPS (сетевых)
- 436k/14 = 31k RPS per Core (Server only)
- Клиент ест столько же примерно, получаем 15k RPS per Core
- Но реквесты очень большие

Сравнение RPS на примитивных запросах



PS Мы пожертвовали Latency ради Throughput! Добавляется где-то 3 ms.

1. Сетевой протокол

Http2

- Создатели протокола забыли про реверс прокси
- Серверный пуш в Kestrel и HttpClient так и не запилили
- Хотели сделать включенным по умолчанию в 3.0, но пришел Стивен Тауб и все отменил
- Есть еще много людей в интернетах, которые сомневаются в полезности http2 для RPC сценариев
- Upgrade заголовок для http2 никто не поддержал, в итого он совместим с http1.1 только через ALPN, который часть TLS
- Для нешифрованного http2 (h2c) нет браузерной поддержки
- Наши результаты противоречивы - иногда лучше, иногда хуже
- Techempower тесты на http1.1

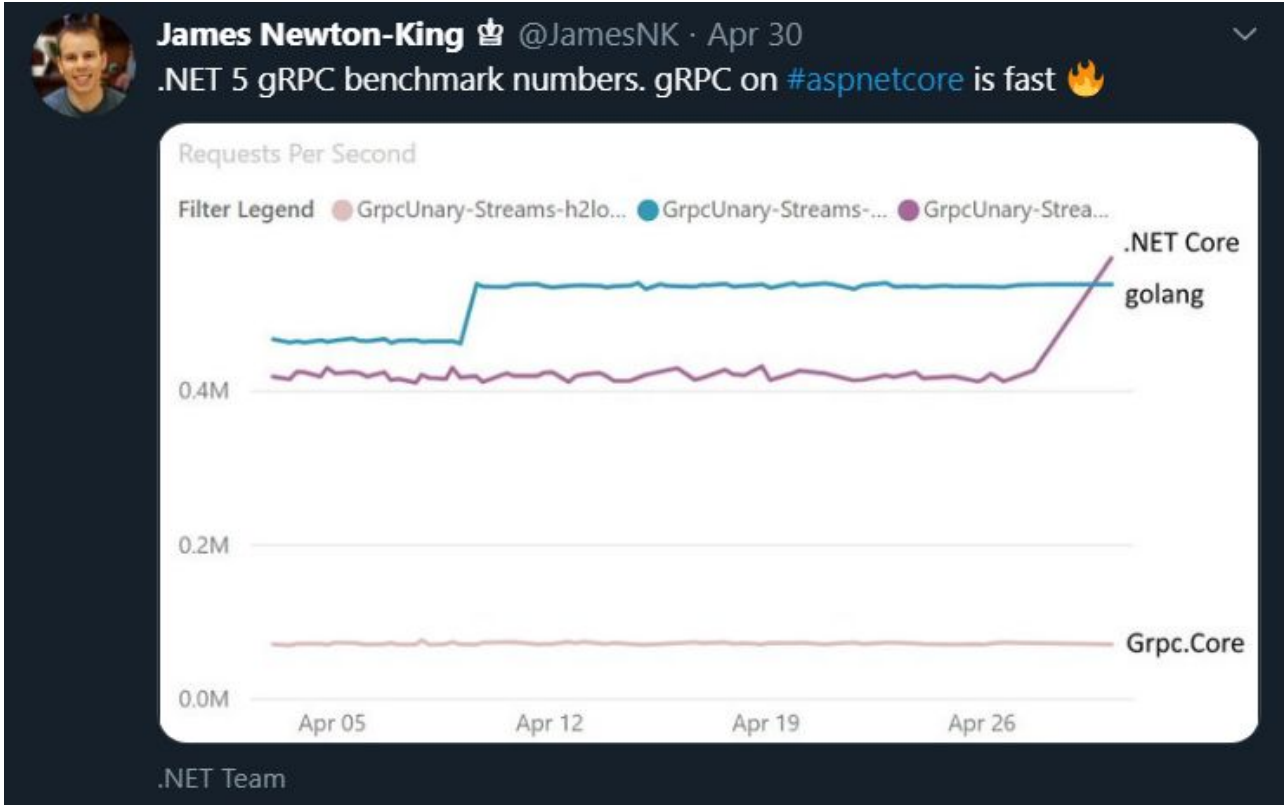
QUIC

- Решает проблему с плохими сетями
- Передача видео поверх UDP от одноклассников: https://www.youtube.com/watch?v=aXYJlizk3CQ
- UDP медленнее в ядре чем TCP (внезапно)
- Для RPC сценария поверх хороший сетей не ждите особых преимуществ
- Про проблемы QUIC: https://www.youtube.com/watch?v=6G2v5ni4VP0
- “китайский” QUIC: https://improbable.io/blog/kcp-a-new-low-latency-secure-network-stack
- https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/

GRPC

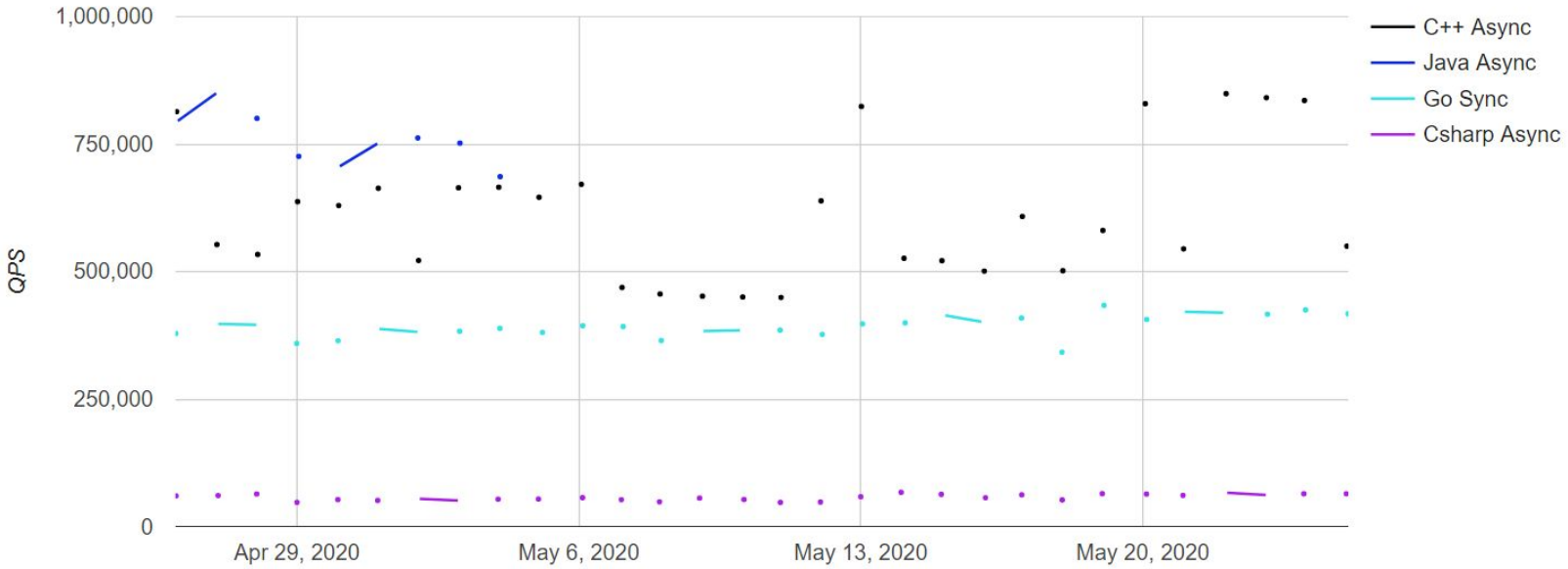
- Protobuf + Http2
- Больше чем RPC (Streaming)
- Поддержан на многих платформах
- GrpcCore - нативная библиотека (своя реализация клиента и сервера http2)
- Гугловая protobuf либа ужасна (кодогенерация и т.п.)
- James Newton-King (да-да, тот самый!) пилит managed реализацию без GrpcCore: https://github.com/grpc/grpc-dotnet
- Marc Gravell пытается переделать с гуглового протобуфа на protobuf-net: https://github.com/protobuf-net/protobuf-net.Grpc
- Есть более дружественная (+ Unity), но ни с кем не совместимая реализации от Yoshifumi Kawai (neucc): https://github.com/Cysharp/MagicOnion

.net Grpc performance hype



Real .net GRPC performance (32 core client to 32 core server)?

Grpc-dotnet/Grpc.Core: 2-4 network syscall на одно сообщение



Назад к основам - сокеты

- Posix/BSD сокеты в unix
- На самом деле все используют расширения, они свои в linux, *bsd, macos, etc
- В Windows NT асинхронные IOCP (1993)
- Формально AIO давно был в Posix, но в линуксе реализовали поделку
- В линуксе совсем недавно (5.4) появился полноценный асинхронный IO
- Уже есть экспериментальный транспорт для Kestrel: https://github.com/tkp1n/IoUring.Transport
- On v5.7, io_uring delivers 23% more throughput and 74% less latency than the current TechEmpower chart-topper “aspcore-rhtx” (это на площадке автора, напоминаю, на площадке TechEmpower все уже уперлось в канал): https://ndportmann.com/io_uring-preview-release/
- Обычно сетевые программисты на юниксе не умеют (и не хотят) правильно использовать Windows сокеты
- Как пример nginx так и не смог отрелизится на Windows
- Возможно это объясняет провал GRPC core в .Net
- Но не объясняет провал grpc-dotnet

Пример с сокетами

```
public static async Task StartClient()
{
    var client = new TcpClient();
    await client.ConnectAsync(IPAddress.Loopback, 2222);

    var watch = Stopwatch.StartNew();

    var buffer = new byte[blockSize];
    for (int i = 0; i < iterationCount; ++i)
    {
        await client.Client.SendAsync(buffer.AsMemory(0, blockSize), SocketFlags.None);
        await client.Client.ReceiveAsync(buffer, SocketFlags.None);
    }
}

public static async Task StartServer()
{
    var listener = new TcpListener(IPAddress.Any, 2222);
    listener.Start();
    var socket = await listener.AcceptSocketAsync();
    byte[] buffer = new byte[blockSize];

    while (true)
    {
        var received = await socket.ReceiveAsync(buffer, SocketFlags.None);
        await socket.SendAsync(new ArraySegment<byte>(buffer, 0, received), SocketFlags.None);
    }
}
```

Meltdown/Spectre/Ghost etc

- Патчи дорогие. Ядро теперь не может мапать свою память в юзер спейс, а должно ее копировать.
- Пострадали в первую очередь клиенты публичных клаудов, так как их гипервизоры пришлось патчить в первую очередь
- Результаты тестирования скачут
- Еще все усложняет то как выходят и доставляются патчи микрокода
- Худшее что видел для RPCClient (48k с митигацией, 86k без)
- Публиковать результаты тестов нельзя - засудят: https://www.imore.com/intel-says-you-cant-publish-benchmarks-after-microcode-update
- Утилита посмотреть/настроить для Windows: https://www.grc.com/inspectre.htm

Еще про микрокод

damageboy @damageboy · Nov 13, 2019

Here's what it looked like Yesterday, vs. Today:

Those of you with keen eye-sight can immediately tell that something is awfully wrong here....
Array.Sort() is 20% (!) slower when sorting 10M numbers for example...

Yesterday

Method	N	Mean [us]
ArraySort	100	1.959 us
Scalar	100	4.264 us
Unmanaged	100	2.134 us
ArraySort	1000	35.455 us
Scalar	1000	52.072 us
Unmanaged	1000	31.780 us
ArraySort	10000	518.356 us
Scalar	10000	696.461 us
Unmanaged	10000	519.058 us
ArraySort	100000	6,269.525 us
Scalar	100000	7,025.668 us
Unmanaged	100000	5,991.170 us
ArraySort	1000000	69,795.523 us
Scalar	1000000	77,646.586 us
Unmanaged	1000000	66,285.149 us

Today

ArraySort	100	2.279 us
Scalar	100	3.767 us
Unmanaged	100	2.152 us
ArraySort	1000	41.409 us
Scalar	1000	61.571 us
Unmanaged	1000	40.223 us
ArraySort	10000	613.608 us
Scalar	10000	792.978 us
Unmanaged	10000	542.217 us
ArraySort	100000	7,609.731 us
Scalar	100000	8,295.480 us
Unmanaged	100000	6,133.407 us
ArraySort	1000000	83,487.392 us
Scalar	1000000	91,233.407 us
Unmanaged	1000000	66,823.423 us
ArraySort	10000000	961,850.814 us
Scalar	10000000	1,015,150.268 us
Unmanaged	10000000	758,285.284 us

2 44 148

Potential Performance Effects of the MCU

The JCC erratum MCU workaround will cause a greater number of misses out of the Decoded ICache and subsequent switches to the legacy decode pipeline. This occurs since branches that overlay or end on a 32-byte boundary are unable to fill into the Decoded ICache.

Intel has observed performance effects associated with the workaround ranging from 0-4% on many industry-standard benchmarks.[1] In subcomponents of these benchmarks, Intel has observed outliers higher than the 0-4% range. Other workloads not observed by Intel may behave differently. Intel has in turn developed software-based tools to minimize the impact on potentially affected applications and workloads.

Vectored IO (Scatter/Gather)

- Stream удобная для использования, но неэффективная, абстракция для описания сетевого и дискового IO
- Как правило использование Stream приводит к лишнему копированию и аллокации (иногда в LOH)
- Использование списка частично заполненных буферов вместо одного непрерывного
- Ядро давно умеет и в Posix и в Windows, вопрос только дотащить до прикладного кода
- На самом деле Socket в .net тоже умеют уже 15 лет через SocketAsyncEventArgs

Новый dotnet API для работы с IO: Pipelines

- Не путать с Windows Pipes (System.IO.Pipes)
- Списки блоков с back pressure
- Сильно асимметрично для чтения и записи SequenceReader/BufferWriter
- Очень сложно писать десериализатор поверх Pipelines
- Единственный способ работы с tls/ssl через SslStream
- Была попытка написать “openSSL“ с нуля на C#, но она скорее мертва: https://github.com/Drawaes/Leto
- К сожалению код сериализаторов и десериализаторов поверх нового API получается медленнее
- Полный процесс: Pipeline (vectorized) -> SslStream (reslice per 19k) -> Socket IOCP (single buffer, not vectorized!)
- Пример интеграции с сокетами: https://github.com/davidfowl/BedrockFramework/blob/master/src/Bedrock.Framework/Transports/Sockets/SocketConnection.cs

  https://github.com/StackExchange/StackExchange.Redis/tree/master/src/StackExchange.Redis

Span и Async не дружат!

- Три варианта span: managed memory, unmanaged memory, stackalloc
- Ref struct (span) следовало бы назвать stackonly struct (идея такая была!)
- Ref struct не может реализовать интерфейсы и быть параметром шаблона
- Ref struct не может попадать в замыкание авейтера
- По факту может участвовать в асинхронном методе если не попадает в переменную
- В итоге десериализатор должен чередовать асинхронное чтение и синхронную работу со спанами
- Sequence/Memory.GetSpan относительно дорогой из-за pinning
- Десериализатор должен уметь остановится, сохранить сохранить и выйти наружу
- Т.е. Десериализатор должен строить свой стек состояний вместо стандартного обеспечиваемого рантаймом

2. База данных

Почему бы не взять магическую мастер-мастер БД?

- Kyle Kingsbury: https://jepsen.io/analyses
- Читать лучше со старых постов подряд
- Продвинуться займет несколько часов
- Доклад к сожалению только по верхам “Anna Concurrenina by Kyle Kingsbury”: https://www.youtube.com/watch?v=eSaFVX4izsQ
- https://en.wikipedia.org/wiki/Amdahl%27s_law
- https://en.wikipedia.org/wiki/CAP_theorem
- https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing
- У Jepsen в тестах нет настоящих коммерческих баз данных, потому что вендоры засудят если что

Jepsen test for mongo 4.2.6

- Очень много ложных маркетинговых утверждений “full ACID” в документации
- Конфигурация по умолчанию склонна к потере данных
- 99.6% инсталляций используют read concern по умолчанию, который небезопасен (~ SQL-92: read uncommitted)
- Snapshot read concern не гарантирует snapshot isolation
- В случае network partitioning жди беды
- “*MongoDB, Fauna, and YugaByte have previously engaged Jepsen for paid analyses.*”

Наш опыт DynamoDB

- “Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second.” © Amazon
- Похоже что хранит строковый формат JSON как есть
- Автоматические шардинг по 10 GiB (добавить и убрать шарды нельзя)
- Максимальный provisioned throughput который можно поставить 80k RPS с одной “таблицы” в кластере
- Реальный throughput ограничен 3-4k RPS с шарда
- Встроенный мониторинг показывает неверный latency
- Eventually consistent вторичные индексы. Нельзя узнать когда документ наконец проиндексировался.
- Требует кеша!? (еще один SaaS elasticache)
- Как же там с согласованностью кэша, данных и вторичных индексов лучше и не спрашивать

Где же хранить

- Поход в out of process БД обычно включают тот же неэффективный метод работы с сокетами, т.е. работает слишком медленно
- Самый быстрый RPC Redis все равно медленнее нашего RPC
- In process БД нам достаточно KV
- In process не скалируется

Rocksdb

- Изначально патченные фейсбуком Leveldb
- Теперь уже настоящий монстр KV баз
- Есть гарантированная записи в WAL как во взрослых базах
- Часто используется как низкоуровневый движок в других базах: MySql/MyRocks
- Есть optimistic/pessimistic transactions и WriteBatch
- Вторичные индексы - на вашей совести
- Про устройство LSM баз популярно “Vinyl: why we wrote our own write-optimized storage engine rather than chose RocksDB”: https://www.youtube.com/watch?v=BlegcIuURv4
- Прототип LSM движка на C#: https://github.com/Drawaes/TrimDB/

3. Сериализация

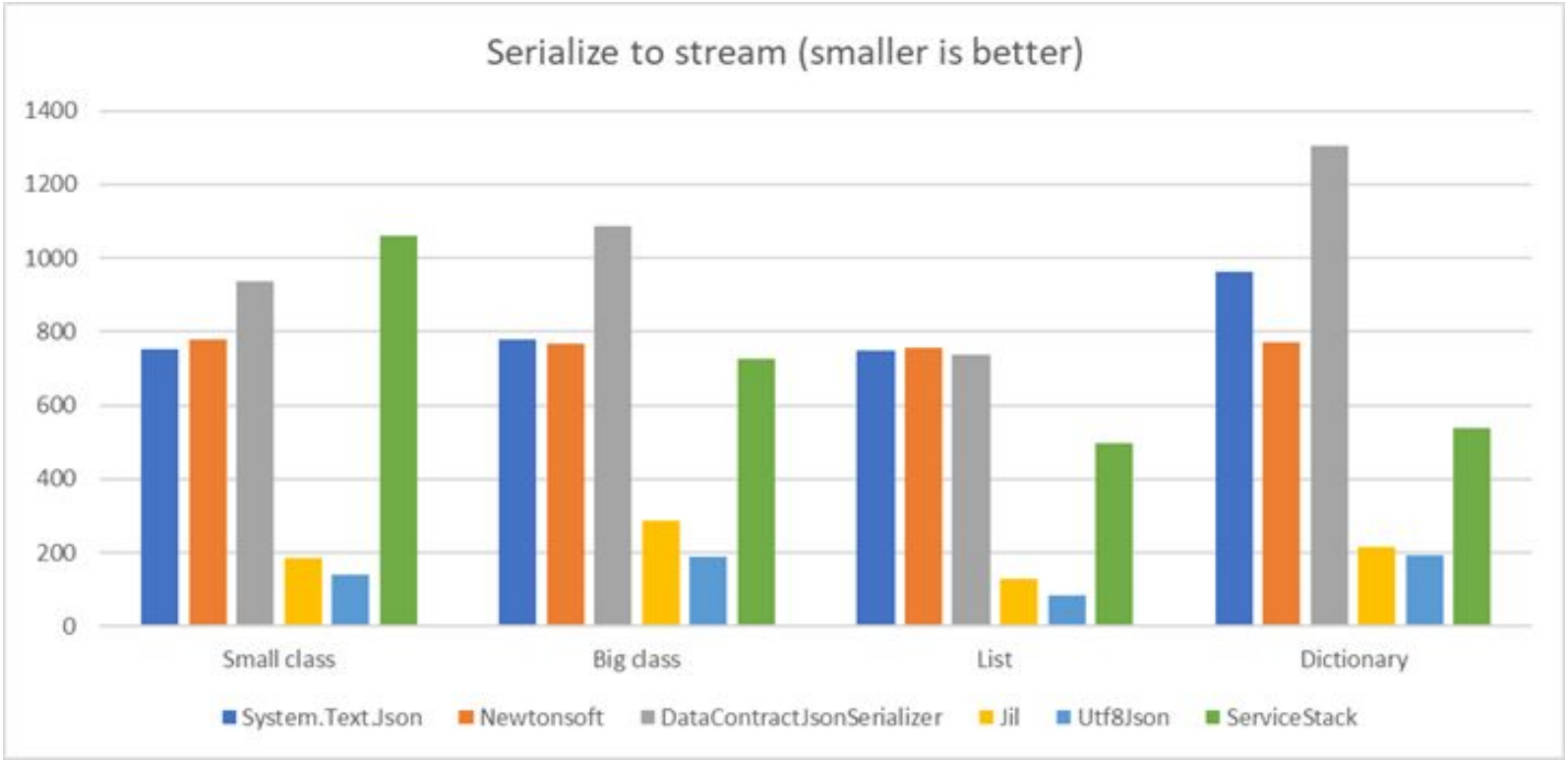
Прошлый и современный Json

- Ajax был до JSON
- JSON по сути появился как суррогатный формат, благодаря тому что парсер JavaScript есть в любом браузере
- Serialize = toString, Deserialize = eval
- Альтернатива в виде сериализации данных прямо в HTML тоже была
- Внезапно очень быстрый (2.2 GiB/s скорость курсора)
- https://arxiv.org/abs/1902.08318
- Порт на C# от Егора Богатова: https://github.com/EgorBo/SimdJsonSharp (курсор, а не десериализатор)
- Arm 8.3-A: FJCVTZS is "Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero".

System.Text.Json (.Net core 3)

- Поддерживает Pipeline модель
- Умеет асинхронно замирать при недостатке данных
- Не такой уж быстрый

Serialize to Stream



https://michaelscodingspot.com/the-battle-of-c-to-json-serializers-in-net-core-3/

Бинарные сериализации Tag vs Offset

- Tag = Schema is optional
- Offset = Schema is required
- Offset быстрее читать если надо небольшой срез данных
- Offset может быть zero copy
- Tag можно читать даже если нет схемы для всех данных
- Offset форматы: FlatBuffers, страницы MS SQL Server для традиционных таблиц

ProtoBuf vs MessagePack

- Удобство для чтения против удобства для записи
- Protobuf пишет физическую длину в байтах
- MessagePack пишет логическую длину в элементах

MessagePack-CSharp

- Yoshifumi Kawai + Andrew Arnott (v2)
- С поддержкой новых API Pipelines/IBufferWriter/ReadOnlySequence
- Без поддержки замирания при недостатки данных, буферизирует весь ReadOnlySequence прежде чем начать десериализацию
- V2 медленнее V1

FrozenObjects

- Размещение immutable графа managed объектов в unmanaged памяти
- https://github.com/microsoft/FrozenObjects

Компрессия

- Любое IO лучше с компрессией!
- Мы бы тоже легко уперлись в 10 Gbit сеть если бы не сжимали сетевой трафик Lz4
- Не используйте gzip/deflate если от вас не требует этого клиент (читай браузер)
- Можете потестить на своих данных разные алгоритмы: https://github.com/inikep/lzbench
- Lz4 (Yann Collet) - очень быстро и практически незаметно
- Zstd (Yann Collet at Facebook) - большой диапазон настроек, поддерживает словарную компрессию, всегда быстрая декомпрессия
- https://facebook.github.io/zstd/#small-data

Наш музей костылей и велосипедов

- Косметически патченный rocksdb (смержено > 6.10.1)
- RocksdbSharp сильно патченый
- Аллокации - свой Unmanaged Buffer Pool (можно перейти на POH в Net 5) со Slice Reference Counting
- RPC - свой целиком поверх NetworkStream/SslStream
- ZSTD, LZ4 - свои обертки для правильной работы со словарями
- MessagePack Reader/Writer свои оптимизированные для Span
- DOM - свой
- System.IO.Pipelines - косметический пуллреквест (смержено)

Наш опыт MongoDb

- BSON, который использует монга, очень похож на MessagePack
- Настроили WiredTiger похоже на наш RocksDb
- Вырезаем из большого документа 15 скалярных проекций
- В монге это можно делать прямо на сервере
- У нас сервер глуповат, клиент забирает документ целиком и разбирается сам в MessagePack
- Пропускная способность 6k RPS на монге против 32k RPS у нас
- Все кластеры монги, про которые я слышал, заканчиваются на 12 машинах
- С учетом Закона Амдала, скорее всего, мы обгоняем любой “разумный” кластер монги

Решающая битва с MS SQL

```
select top (1000000)
    cast(s.MasterSecID as nvarchar(100)) as OwnerId,
    cast(s.PaladyneID as nvarchar(100)) as ExternalId,
    s.Name as Name,
    st.Code as SecTypeCode,
    ex.Code as ExchangeCode,
    cast(s.RowTimestamp as bigint) as ChangedTimestamp,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300001 ORDER BY IdStartDate  DESC) as Cusip,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300002 ORDER BY IdStartDate  DESC) as Isin,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300003 ORDER BY IdStartDate  DESC) as Sedol,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300015 ORDER BY IdStartDate  DESC) as Ticker,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300005 ORDER BY IdStartDate  DESC) as Symbol,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300007 ORDER BY IdStartDate  DESC) as BlmbrgId,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300008 ORDER BY IdStartDate  DESC) as VendorId,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300110 ORDER BY IdStartDate  DESC) as BlmbrgGlobalId,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300004 ORDER BY IdStartDate  DESC) as Ric,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300006 ORDER BY IdStartDate  DESC) as AssetId,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300021 ORDER BY IdStartDate  DESC) as OccCode,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300135 ORDER BY IdStartDate  DESC) as IdcId,
    (SELECT TOP 1 SecCode FROM dbo.SecIdCode WITH(NOLOCK) WHERE SecID = s.SecID AND SecIdCodeTypeId = 300170 ORDER BY IdStartDate  DESC) as FctId,
    cast((case opt.RedemType
        when 'P' then 1
        when 'C' then 0
        else null
    end) as bit) as IsPutOption,
    opt.StrikePrice as StrikePrice,
    opt.ExpDate as ExpirationDate,
    cpn.Rate as CouponRate,
    cnt.EqCode as CountryIso3Code,
    cur.Code as CurrencyIso3Code,
    isnull(ord.IsPrimary,pfd.IsPrimary) as IsPrimary,
    s.MasterSecID
from dbo.Sec s
inner join dbo.SecAuxCustomValues a on a.SecID = s.SecID
left join dbo.RefSecType st on s.RefSecTypeID = st.RefSecTypeID
left join dbo.RefExchange ex on s.RefExchangeID = ex.RefExchangeID
left join dbo.RefCountry cnt on cnt.RefCountryID=s.RefCountryID
left join dbo.RefCurrency cur on cur.RefCurrencyID=s.RefCurrencyID
left join dbo.SecOptions opt on opt.SecID = s.SecID
left join dbo.SecBondCoupon cpn on cpn.SecID = s.SecID
left join dbo.SecEquityORD ord on ord.SecID=s.SecID
left join dbo.SecEquityPFD pfd on pfd.SecID=s.SecID
```

- Все страницы в кеше
- Запрос успешно параллелизуется
- Discard results, shared mem
- 8 Cores
- 27 секунд - лучший результат
- Итого: 37k RPS

Here is why I like duct tape programmers. Sometimes, you're on a team, and you're busy banging out the code, and somebody comes up to your desk, coffee mug in hand, and starts rattling on about how if you use multi-threaded COM apartments, your app will be 34% sparklier, and it's not even that hard, because he's written a bunch of templates, and all you have to do is multiply-inherit from 17 of his templates, each taking an average of 4 arguments, and you barely even have to write the body of the function. It's just a gigantic list of multiple-inheritance from different classes and hey, presto, multi-apartment threaded COM. And your eyes are swimming, and you have no friggin' idea what this frigtard is talking about, but he just won't go away, and even if he does go away, he's just going back into his office to write more of his clever classes constructed entirely from multiple inheritance from templates, without a single implementation body at all, and it's going to crash like crazy and you're going to get paged at night to come in and try to figure it out because he'll be at some goddamn "Design Patterns" meetup.

And the duct-tape programmer is not afraid to say, "multiple inheritance sucks. Stop it. Just stop."

https://www.joelonsoftware.com/2009/09/23/the-duct-tape-programmer/