

# Real-world HTTP performance benchmarking, lessons learned

Julien Viet  
QCon Shanghai

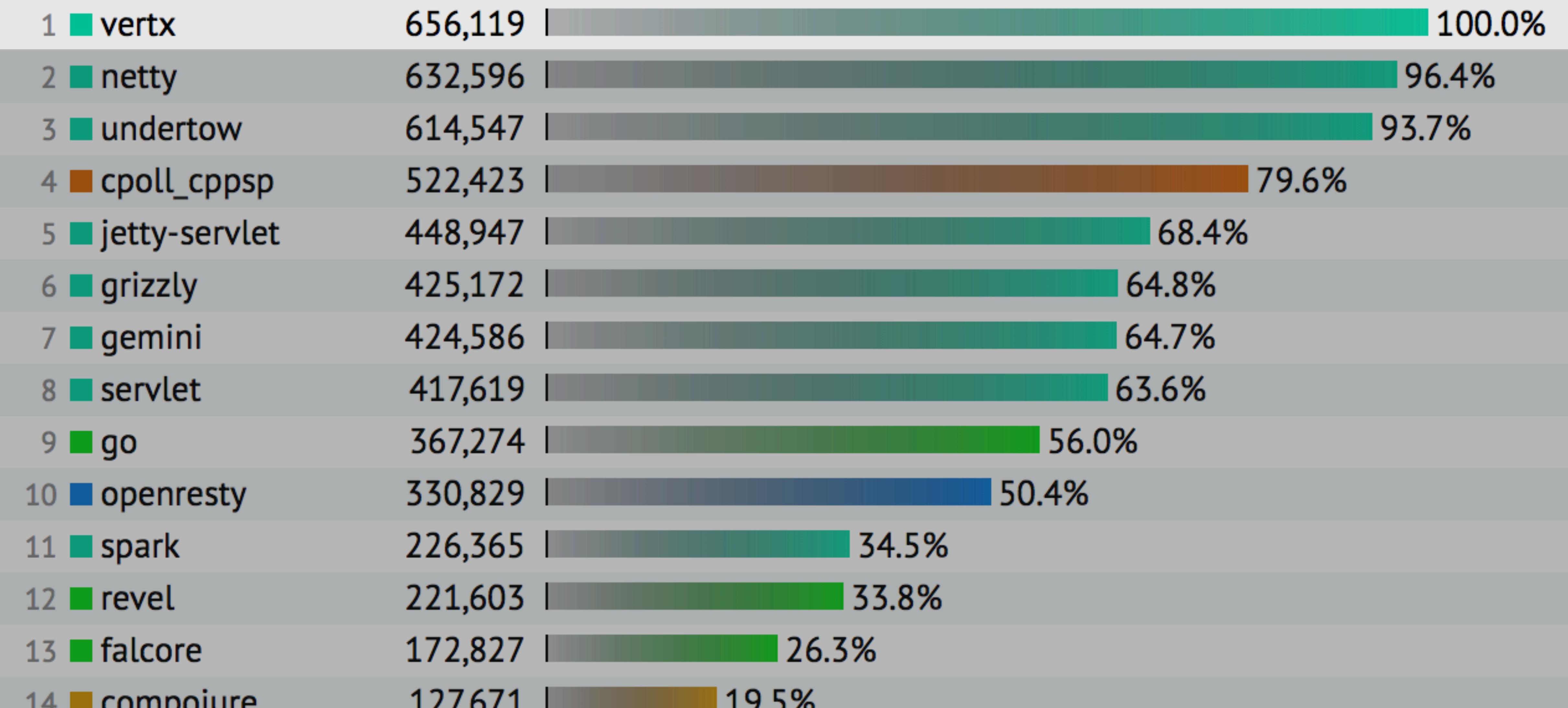


使用折扣码「QCon」优惠报名 咨询电话：13269078023



扫码锁定席位

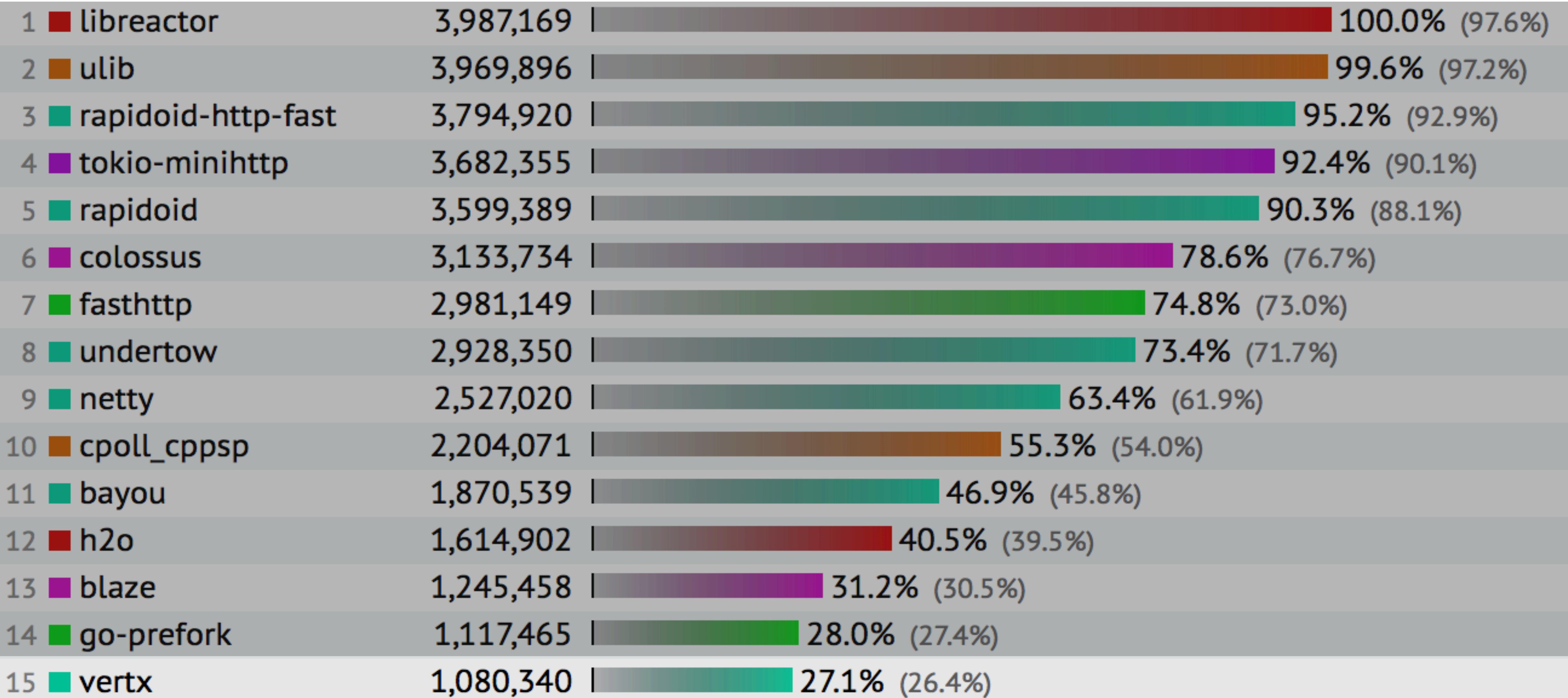
Once upon a time



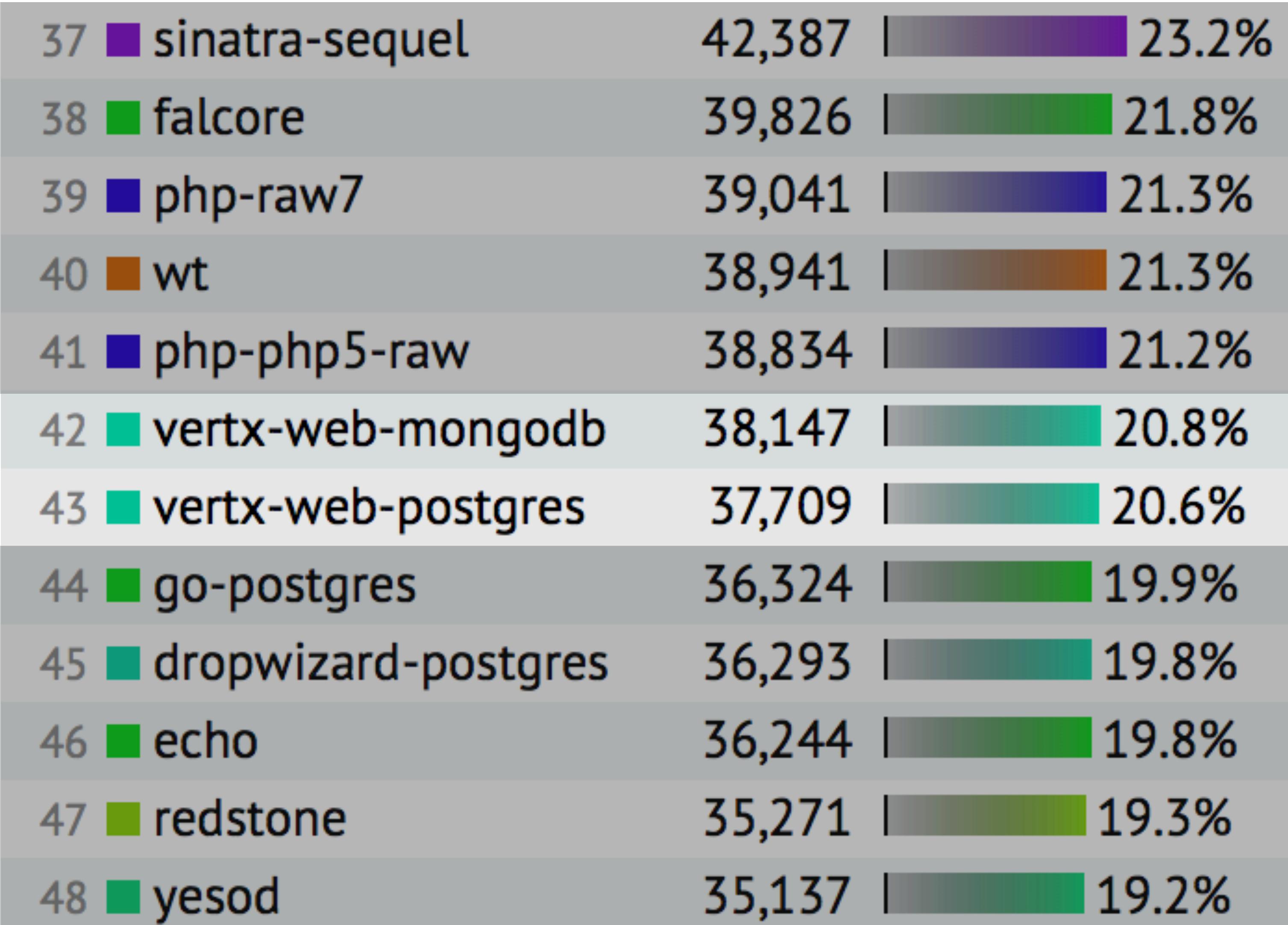
Round #8

*Every one was happy*

But one day...



Round #14



Round #14

|     |  |   |
|-----|--|---|
| 157 |  koa                    | 0   0.0%                                  |
| 158 |  ktor                   | 0   0.0%                                  |
| 159 |  nodejs-mongodb         | 0   0.0%                                  |
| 160 |  nodejs-mongodb-raw     | 0   0.0%                                  |
| 161 |  sailsjs              | 0   0.0%                                  |
| 162 |  play2-scala-reactive | 0   0.0%                                  |
| 163 |  roda-sequel-torquebo | 0   0.0%                                  |
| 164 |  vertx-web-mongodb    | 0   0.0%                                  |
| 165 |  vertx-web-postgres   | 0   0.0%                                  |
| 166 |  yesod                | 0   0.0%                                  |
| 167 |  hexagon              | — <a href="#"><u>Did not complete</u></a> |
| 168 |  jawn                 | — <a href="#"><u>Did not complete</u></a> |

|     |                      |                                    |
|-----|----------------------|------------------------------------|
| 157 | ■ koa                | 0   0.0%                           |
| 158 | ■ ktor               | 0   0.0%                           |
| 159 | ■ nodejs-mongodb     | 0   0.0%                           |
| 160 | ■ nodejs-mongodb-raw | 0   0.0%                           |
| 161 | ■ sailsjs            |                                    |
| 162 | ■ play2-scalajs      |                                    |
| 163 | ■                    |                                    |
| 164 | ■ vertx-mongodb      | 0   0.0%                           |
| 165 | ■ vertx-web-postgres | 0   0.0%                           |
| 166 | ■ yesod              | 0   0.0%                           |
| 167 | ■ hexagon            | – <a href="#">Did not complete</a> |
| 168 | ■ jawn               | – <a href="#">Did not complete</a> |

Hi,

I recently reviewed the latest [techempower results \(#14\)](#) and

Vertx seems to be sliding downward in the list.

Round #14

|     |   |                                    |
|-----|---|------------------------------------|
| 1   | But in the benchmark results, I see that the "Data Table" stats are not really good for Vert.x. Also, nodejs stats are really good in database calls. | 0 10.0%                            |
| 158 | ktor  | 0 10.0%                            |
| 159 | nodejs-mongodb  | 0 10.0%                            |
| 160 | nodejs-mongodb-raw  | 0 10.0%                            |
| 161 | sailsjs   | 0 10.0%                            |
| 162 | play2-scala   | 0 10.0%                            |
| 163 |   | 0 10.0%                            |
| 164 | Hi,<br>I recently reviewed the latest techempower results (#14) and Vertx is sliding downward in the list.  | 0 10.0%                            |
| 165 | vertx-mongodb   | 0 10.0%                            |
| 166 | vertx-web-postgres  | 0 10.0%                            |
| 167 | yesod   | 0 10.0%                            |
| 168 | hexagon   | – <a href="#">Did not complete</a> |
|     | jawn  | – <a href="#">Did not complete</a> |

Real-world HTTP performance  
benchmarking, lessons learned

# Julien Viet

Open source developer for 16+ years

**@vertx\_project** lead

Principal software engineer at  redhat.

Marseille JUG Leader

 <https://www.julienviet.com/>

 <http://github.com/vietj>

 @julienviet

 <https://www.mixcloud.com/cooperdbi/>

# Eclipse Vert.x

Open source project started in 2012

Eclipse / Apache licensing

A **toolkit** for building **reactive** applications for the JVM

8K ★ on 

Built on top of  Netty

 <https://vertx.io>

 @vertx\_project

# TechEmpower Framework Benchmark

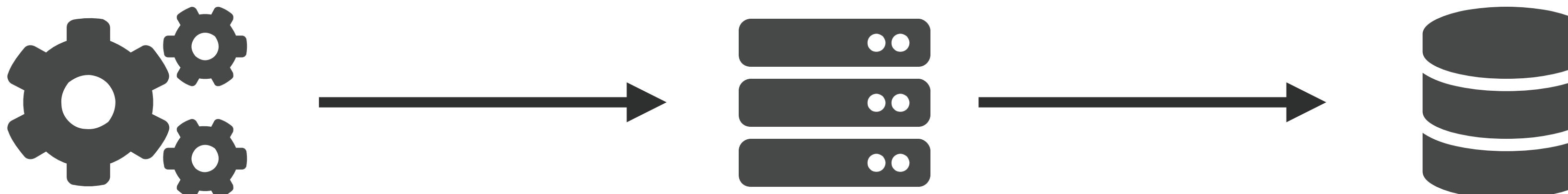
- ✓ Performance of production grade deployments of real-world application frameworks and platforms
- ✓ 464 frameworks - 26 languages
- ✓ Community of contributors on GitHub
- ✓ Physical server or cloud (Azure)

# 6 benchmarks

- ✓ "/plaintext", "/json"



- ✓ "/db", "/queries", "/updates", "/fortunes"



# Things to remember

- ✓ Benchmarking is hard
- ✓ Benchmarking != load testing
- ✓ Measure don't guess
- ✓ Be critic

# The lab



Julien Viet <jviet@redhat.com>  
to Rodney ▾

Hi Rodney,

we inherited from the two boxes that Norman Maurer used for Netty testing (and that Tim also used), one of them is down currently so the single box we have is useless. Besides Jean Marc Larvol from helpdesk said that these machines are unreliable and too old.

We talked a couple of times in the past of having machines for doing performance testing that can be used by the team, pretty much like what Norman and Tim used. The characteristics of these boxes is that they are not virtual (for good reasons) and use private networking.

One use case is for Techempower benchmarking (Tim used them for this scenario), other use case are for clustering performance testing.

Ideally we would need 4 boxes.

Can we get budget for these machines, wether they are hosted by Red Hat or rented on a provider ?



**Rodney Russ <rruss@redhat.com>**

to me ▼

I'll put in for budget for FY18.

Hi Rodney,

We received your request for 4 machines with 10gbe network capability starting 4/10/2017 and ending on 6/16/2017. We will have 4 hosts available with 10gbe during your requested window. A bit closer to the start date we will send you a document with all the ip address space and machine hostnames. In the meantime if you have any questions please reply to this email and one of us will get back with you. If your project specs change please let us know asap.

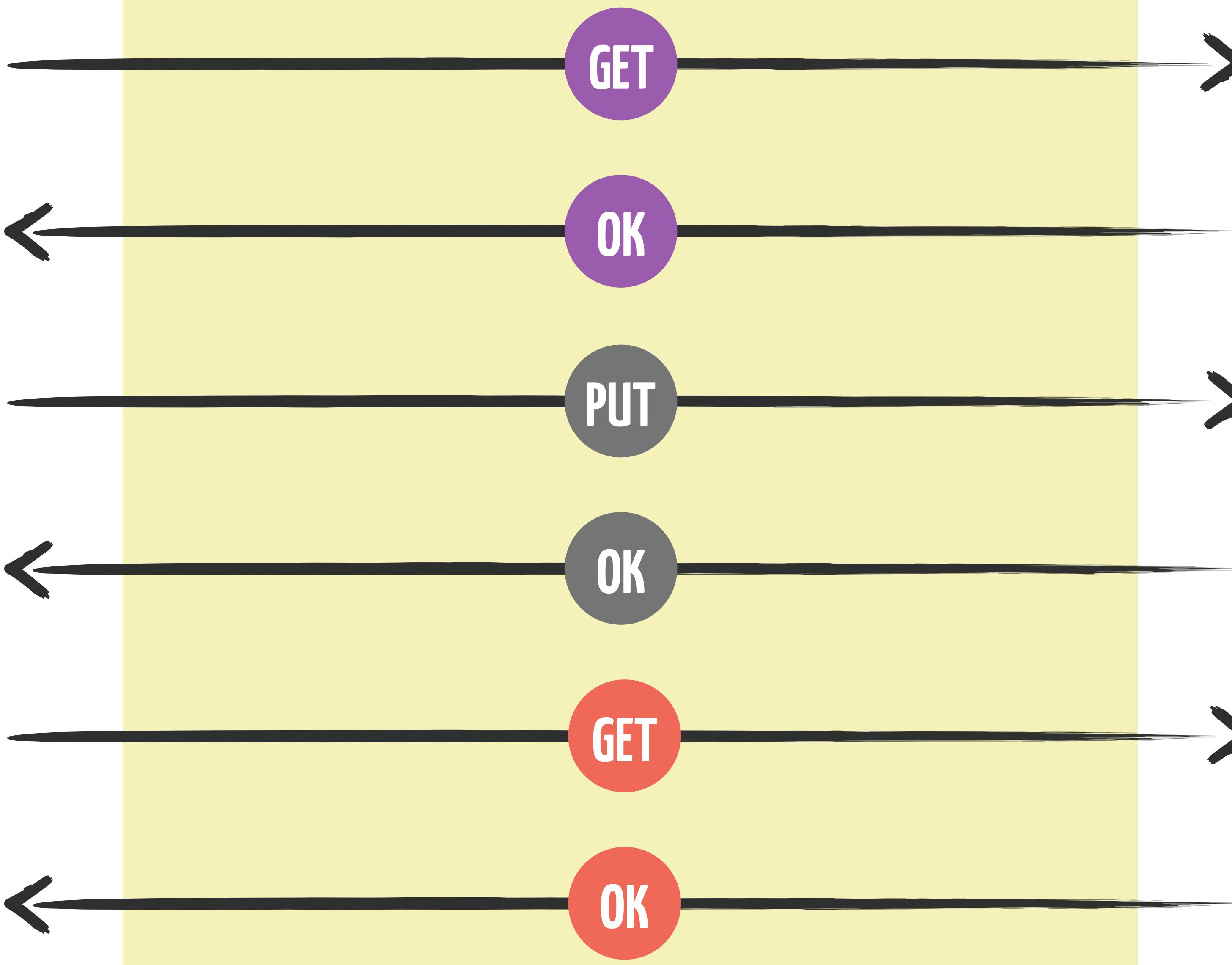
Thanks,  
Jake

/plaintext

# Benchmark

- ✓ Simple Hello World
- ✓ 16,384 concurrent connections
- ✓ HTTP pipelining (16)
- ✓ No back-end
- ✓ Heavily CPU bound

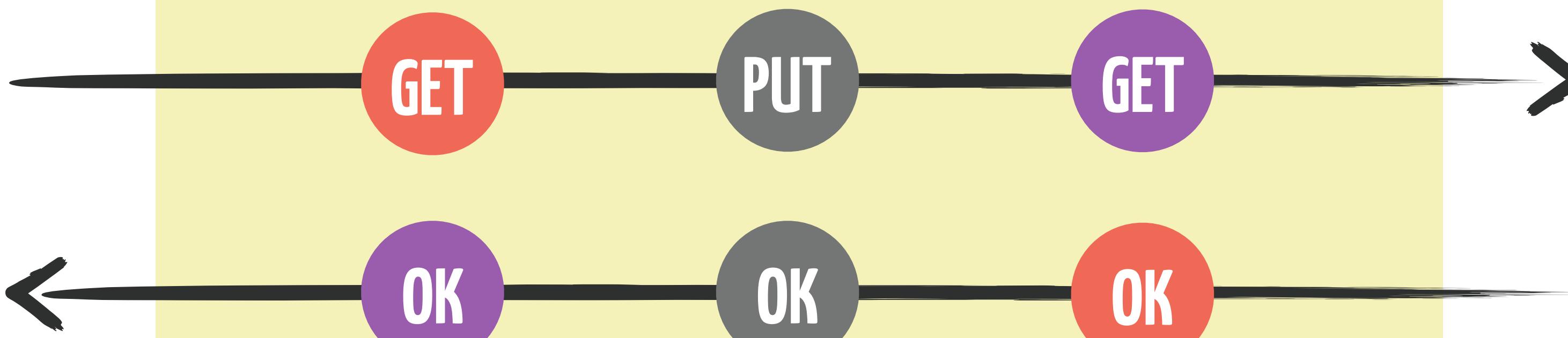
## *Keep-alive*



# Head of line blocking



# *Pipelining*



# Our weapons

- ✓ Async-profiler + Flame graphs
- ✓ Jitwatch
- ✓ Wireshark

# Code inlining

*process  
error*

```
private void processMessage(Object msg) {
    if (msg instanceof HttpObject) {
        HttpObject obj = (HttpObject) msg;
        DecoderResult result = obj.decoderResult();
        if (result.isFailure()) {
            Throwable cause = result.cause();
            if (cause instanceof TooLongFrameException) {
                String causeMsg = cause.getMessage();
                HttpVersion version;
                if (msg instanceof HttpRequest) {
                    version = ((HttpRequest) msg).protocolVersion();
                } else if (currentRequest != null) {
                    version = currentRequest.version() == io.vertx.core.http.HttpVersion.HTTP_1_0 ? HttpVersion.HTTP_1_0 : HttpVersion.HTTP_1_1;
                } else {
                    version = HttpVersion.HTTP_1_1;
                }
                HttpResponseStatus status = causeMsg.startsWith("An HTTP line is larger than") ? HttpResponseStatus.REQUEST_URI_TOO_LONG : HttpResponseStatus.BAD_REQUEST;
                DefaultFullHttpResponse resp = new DefaultFullHttpResponse(version, status);
                writeToChannel(resp);
            }
            channel.pipeline().fireExceptionCaught(result.cause());
            return;
        }
    }
    if (msg instanceof HttpRequest) {
        HttpRequest request = (HttpRequest) msg;
        if (server.options().isHandle100ContinueAutomatically()) {
            if (HttpHeaders.is100ContinueExpected(request)) {
                write100Continue();
            }
        }
        HttpServerResponseImpl resp = new HttpServerResponseImpl(vertx, conn: this, request);
        HttpServerRequestImpl req = new HttpServerRequestImpl(conn: this, request, resp);
        handleRequest(req, resp);
    }
    if (msg instanceof HttpContent) {
        HttpContent chunk = (HttpContent) msg;
        if (chunk.content().isReadable()) {
            Buffer buff = Buffer.buffer(chunk.content());
            handleChunk(buff);
        }
    }
    if (msg instanceof LastHttpContent) {
        if (!paused) {
            handleEnd();
        } else {
            // Requeue
            pending.add(LastHttpContent.EMPTY_LAST_CONTENT);
        }
    }
    } else if (msg instanceof WebSocketFrameInternal) {
        WebSocketFrameInternal frame = (WebSocketFrameInternal) msg;
        handleWsFrame(frame);
    }
    checkNextTick();
}
```

*process  
request*

```
private void processMessage(Object msg) {
    if (msg instanceof HttpObject) {
        HttpObject obj = (HttpObject) msg;
        DecoderResult result = obj.decoderResult();
        if (result.isFailure()) {
            Throwable cause = result.cause();
            if (cause instanceof TooLongFrameException) {
                String causeMsg = cause.getMessage();
                HttpVersion version;
                if (msg instanceof HttpRequest) {
                    version = ((HttpRequest) msg).protocolVersion();
                } else if (currentRequest != null) {
                    version = currentRequest.version() == io.vertx.core.http.HttpVersion.HTTP_1_0 ? HttpVersion.HTTP_1_0 : HttpVersion.HTTP_1_1;
                } else {
                    version = HttpVersion.HTTP_1_1;
                }
                HttpResponseStatus status = causeMsg.startsWith("An HTTP line is larger than") ? HttpResponseStatus.REQUEST_URI_TOO_LONG : HttpResponseStatus.BAD_REQUEST;
                DefaultFullHttpResponse resp = new DefaultFullHttpResponse(version, status);
                writeToChannel(resp);
            }
            channel.pipeline().fireExceptionCaught(result.cause());
            return;
        }
    }
    if (msg instanceof HttpRequest) {
        HttpRequest request = (HttpRequest) msg;
        if (server.options().isHandle100ContinueAutomatically()) {
            if (HttpHeaders.is100ContinueExpected(request)) {
                write100Continue();
            }
        }
        HttpServerResponseImpl resp = new HttpServerResponseImpl(vertx, conn: this, request);
        HttpServerRequestImpl req = new HttpServerRequestImpl(conn: this, request, resp);
        handleRequest(req, resp);
    }
    if (msg instanceof HttpContent) {
        HttpContent chunk = (HttpContent) msg;
        if (chunk.content().isReadable()) {
            Buffer buff = Buffer.buffer(chunk.content());
            handleChunk(buff);
        }
    }
    if (msg instanceof LastHttpContent) {
        if (!paused) {
            handleEnd();
        } else {
            // Requeue
            pending.add(LastHttpContent.EMPTY_LAST_CONTENT);
        }
    }
    } else if (msg instanceof WebSocketFrameInternal) {
        WebSocketFrameInternal frame = (WebSocketFrameInternal) msg;
        handleWsFrame(frame);
    }
    checkNextTick();
}
```

*process  
body*

```
private void processMessage(Object msg) {
    if (msg instanceof HttpRequest) {
        HttpRequest request = (HttpRequest) msg;
        if (request.decoderResult().isFailure()) {
            handleError(request);
            return;
        }
        if (server.options().isHandle100ContinueAutomatically() && HttpUtil.is100ContinueExpected(request))
            write100Continue();
    }
    HttpServerResponseImpl resp = new HttpServerResponseImpl(vertx, conn: this, request);
    HttpServerRequestImpl req = new HttpServerRequestImpl( conn: this, request, resp);
    currentRequest = req;
    pendingResponse = resp;
    if (metrics.isEnabled()) {
        requestMetric = metrics.requestBegin(metric(), req);
    }
    requestHandler.handle(req);
} else if (msg instanceof HttpContent) {
    HttpContent content = (HttpContent) msg;
    handleContent(content);
} else {
    WebSocketFrameInternal frame = (WebSocketFrameInternal) msg;
    handleWsFrame(frame);
}
checkNextTick();
}
```

process  
error

process  
request

process  
body

```
private void processMessage(Object msg) {
    if (msg instanceof HttpRequest) {
        HttpRequest request = (HttpRequest) msg;
        if (request.decoderResult().isFailure()) {
            handleError(request);
            return;
        }
        if (server.options().isHandle100ContinueAutomatically() && HttpUtil.is100ContinueExpected(request))
            write100Continue();
        }
        HttpServerResponseImpl resp = new HttpServerResponseImpl(verticalScope, conn: this, request);
        HttpServerRequestImpl req = new HttpServerRequestImpl( conn: this, request, resp);
        currentRequest = req;
        pendingResponse = resp;
        if (metrics.isEnabled()) {
            requestMetric = metrics.requestBegin(metric(), req);
        }
        requestHandler.handle(req);
    } else if (msg instanceof HttpContent) {
        HttpContent content = (HttpContent) msg;
        handleContent(content);
    } else {
        WebSocketFrameInternal frame = (WebSocketFrameInternal) msg;
        handleWsFrame(frame);
    }
    checkNextTick();
}
```

process  
error

process  
request

process  
body

*reduce method size to favor inlining*

```
private void processMessage(Object msg) {
    if (msg instanceof HttpRequest) {
        HttpRequest request = (HttpRequest) msg;
        if (request.decoderResult().isFailure()) {
            handleError(request);
            return;
        }
        if (server.options().isHandle100ContinueAutomatically() && HttpUtil.is100ContinueExpected(request))
            write100Continue();
    }
    HttpServerResponseImpl resp = new HttpServerResponseImpl(vertx, conn: this, request);
    HttpServerRequestImpl req = new HttpServerRequestImpl( conn: this, request, resp);
    currentRequest = req;
    pendingResponse = resp;
    if (metrics.isEnabled()) {
        requestMetric = metrics.requestBegin(metric(), req);
    }
    requestHandler.handle(req);
} else if (msg instanceof HttpContent) {
    HttpContent content = (HttpContent) msg;
    handleContent(content);
} else {
    WebSocketFrameInternal frame = (WebSocketFrameInternal) msg;
    handleWsFrame(frame);
}
checkNextTick();
}
```

b2073fa091d64a1dfe06699bca1a8befddb5a805

process  
error

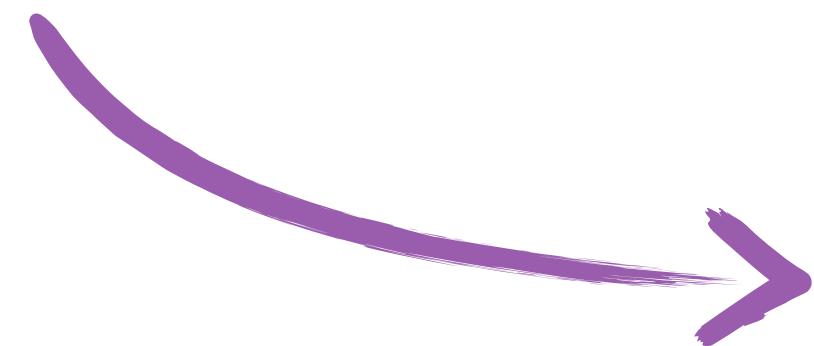
process  
request

process  
body

2. inline by hand

# **Batch to amortise costs**

`chctx.fireChannelRead(msg)`



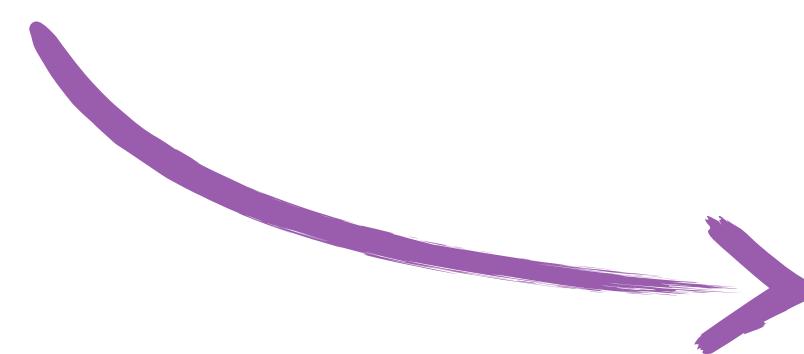
```
// class VertxHandler  
void channelRead(Object msg) {  
    Connection conn = getConnection();  
    Context ctx = conn.getContext();  
    context.executeFromIO(conn::startRead());  
    channelRead(conn, msg);  
}
```

```
// class VertxHttpHandler extends VertxHandler  
void channelRead(Connection conn, Object msg) {  
    conn.handleMessage(msg);  
}
```

`void startRead() { ... }`

`void handleMessage(Object msg) { ... }`

chctx.fireChannelRead(msg)



*Batch here*

```
// class VertxHandler  
public void channelRead(ChannelHandlerContext chctx, Object msg) {  
    Connection conn = getConnection();  
    Context ctx = conn.getContext();  
    context.executeFromIO(() -> {  
        conn.startRead();  
        conn.handleMessage(msg);  
    });  
}
```

```
void startRead() {  
}
```

```
void handleMessage(Object msg) { ... }
```

The fastest code is the code  
that never runs

*Netty*

*Vert.x*

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}
```

*Vert.x*

```
req.response()  
.end("Hello World");
```

*Application*

*Netty*

*Vert.x*

*Application*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}
```

```
void queueForWrite(Object msg) {  
    needsFlush = true;  
    channel.write(encode(obj));  
}
```

```
req.response()  
.end("Hello World");
```

*Netty*

```
ChannelFuture write(Object msg) {  
    return pipeline.write(msg);  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    channel.write(encode(obj));  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
ChannelFuture write(Object msg) {  
    return pipeline.write(msg);  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    channel.write(encode(obj));  
}  
  
// default implementation (inherited)  
void write(ChannelHandlerContext ctx,  
Object msg, ChannelPromise promise) {  
    ctx.write(msg, promise);  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
ChannelFuture write(Object msg) {  
    return pipeline.write(msg);  
}
```

```
void write(Object msg,  
          ChannelPromise promise) {  
    next.invoke(msg, promise)  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    channel.write(encode(obj));  
}  
  
// default implementation (inherited)  
void write(ChannelHandlerContext ctx,  
          Object msg, ChannelPromise promise) {  
    ctx.write(msg, promise);  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
ChannelFuture write(Object msg) {  
    return pipeline.write(msg);  
}  
  
void write(Object msg,  
          ChannelPromise promise) {  
    next.invoke(msg, promise)  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    channel.write(encoded(obj));  
}  
  
// default implementation (inherited)  
void write(ChannelHandlerContext ctx,  
          Object msg, ChannelPromise promise) {  
    ctx.write(msg, promise);  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

# Netty

*the fastest code is the  
code that never runs*

```
void write(Object msg,  
          ChannelPromise promise) {  
    next.invoke(msg, promise)  
}
```

# Vert.x

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    chctx.write(encode(obj));  
}
```

# Application

```
req.response()  
.end("Hello World");
```

217b17c78cd54103ae98557510a7ac431e17c5ea

# Reduce object allocation

*Netty*

```
void write(Object msg) {  
    write(msg, newPromise());  
}  
  
void write(Object msg,  
          ChannelPromise promise) {  
    next.invoke(msg, promise)  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    chctx.write(encode(obj));  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
void write(Object msg, ChannelPromise promise) {  
    next.invoke(msg, promise)  
}  
  
void write(Object msg, ChannelPromise promise) {  
    write(msg, newPromise());  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    chctx.write(encode(obj));  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*Netty*

```
void write(Object msg,  
          ChannelPromise promise) {  
    next.invoke(msg, promise)  
}
```

*Vert.x*

```
void end(Buffer buffer) {  
    FullHttpResponse msg = ...  
    queueForWrite(msg);  
}  
  
void queueForWrite(Object msg) {  
    needsFlush = true;  
    chctx.write(obj,  
                channel.voidPromise());  
}
```

*Application*

```
req.response()  
.end("Hello World");
```

*reduce GC allocation  
by using VoidPromise*

6b9788dec6e1147782a3a7017ead067778095cba

# **Cache expensive operations**

```
void setConnection(Connection conn) {  
    this.conn = conn;  
}
```

```
void channelReadComplete(ChannelHandlerContext ctx) {  
    Runnable task = conn::endReadAndFlush();  
    // Need to use executeFromIO to avoid race conditions  
    context.executeFromIO(task);  
}
```

```
void endReadAndFlush() {  
    if (needFlush) {  
        needFlush = false;  
        channel.flush();  
    }  
}
```



```
void setConnection(Connection conn) {  
    this.conn = conn;  
}
```

```
void channelReadComplete(ChannelHandlerContext ctx) {  
    Runnable task = conn::endReadAndFlush();  
    // Need to use executeFromIO to avoid race conditions  
    context.executeFromIO(task);  
}
```

```
void endReadAndFlush() {  
    if (needFlush) {  
        needFlush = false;  
        channel.flush();  
    }  
}
```

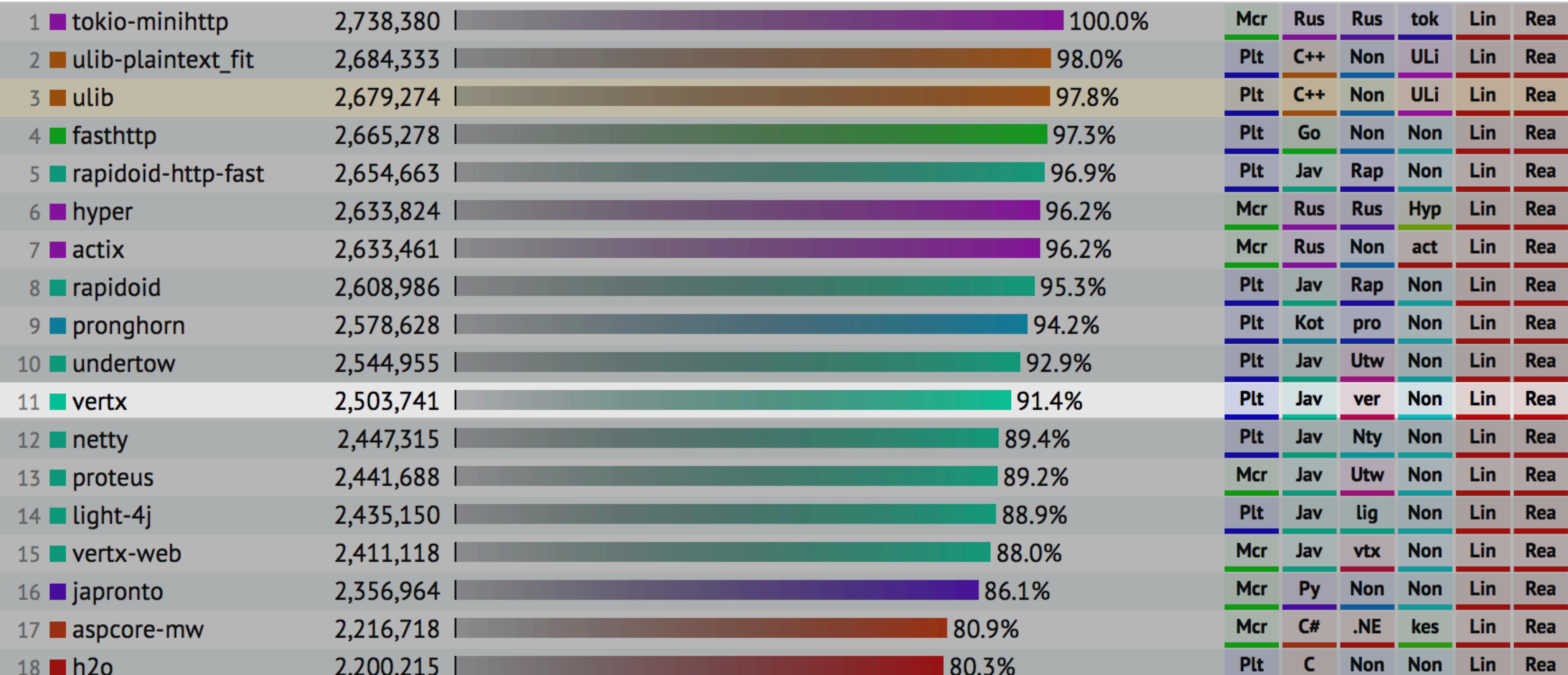
*Called for every  
each flush*

```
void setConnection(Connection conn) {  
    this.conn = conn;  
    this.task = conn::endReadAndFlush();  
}  
  
void channelReadComplete(ChannelHandlerContext ctx) {  
    Runnable task = conn::endReadAndFlush();  
    // Need to use executeFromIO to avoid race conditions  
    context.executeFromIO(task);  
}  
  
void endReadAndFlush() {  
    if (needFlush) {  
        needFlush = false;  
        channel.flush();  
    }  
}
```

*Now called when  
the connection is  
created*

# Extra optimisations

- ✓ Faster HTTP header encoding
- ✓ Cache complex conditions



Round #15

# Plaintext recap

- ✓ A very aggressive benchmark
- ✓ Bottleneck are CPU and networking flushing
- ✓ Less is more

**/db benchmark**

# /db

- ✓ Choice to use PostgreSQL
- ✓ Determine the actual bottleneck: CPU ? Network ? Database ?
- ✓ 256 concurrent connections: non-blocking versus blocking

# First step

- ✓ First improvement: the /updates was actually not using a transaction

# The reactive PostgreSQL client

## ✓ Goals

- Simple, clean and straightforward API
- Performant
- Be a client
- Lightweight

## ✓ Non goals

- Be a driver
- Be an abstraction

```
// Connect directly
PgClient.connect(uri, connection -> {
    // Handle result
});
```

```
// Or create a pool of connections
PgClient pool = PgClient.pool(uri);
pool.getConnection(connection -> {
    // Handle result
});
```

## // Sequential queries

```
connection.query(query1, result1 -> {
    // Got result 1

    connection.query(query2, result2 -> {
        // Got result 2
    });
});
```

```
// What if we do ?  
connection.query(query1, result1 -> {  
    // Got result 1  
});
```

```
connection.query(query2, result2 -> {  
    // Got result 2  
});
```

- the 2 queries executes concurrently ?
- query1 executes then query2 ?
- query1 executes, query2 executes after ?

QUIZ

# Head of line blocking

- ✓ PostgreSQL process one request at a time
- ✓ Send the response after processing
- ✓ Sounds familiar ?

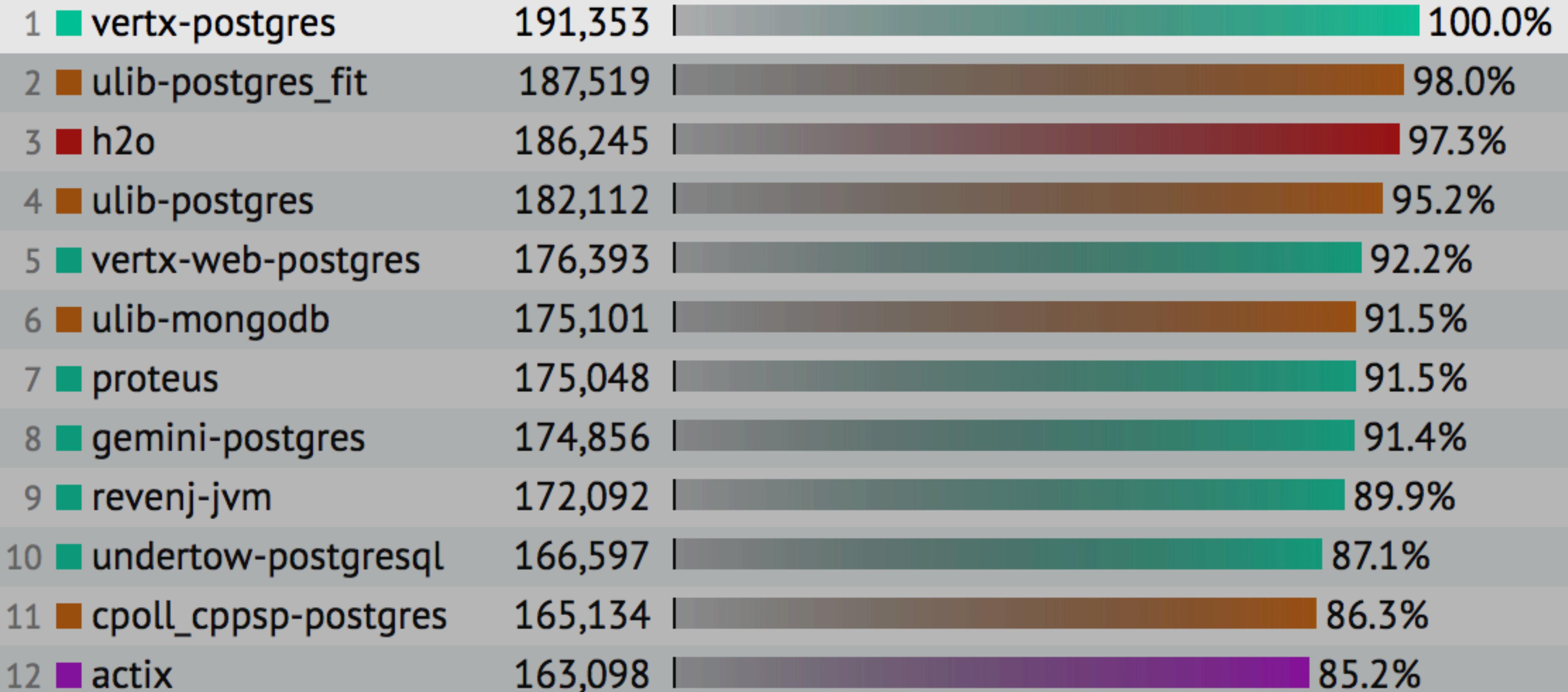
**Let's pipeline it**

# The Reactive Postgres Client

- ✓ A PostgreSQL client
- ✓ Simple and direct API
- ✓ Focusing on performance and low overhead

# Other cool features

- ✓ Direct memory to object without intermediary memory copy
- ✓ Efficient flush to minimise expensive system calls
- ✓ RxJava support
- ✓ Domain sockets support
- ✓ Proxy support



Round #15

**SPEED  
LIMIT  
50**



SPEED  
LIMIT



Hello,

Another interesting bit that I noticed on the dashboard - vertx-postgres and vertx-web postgres perform significantly better than the rest on some of the database benchmarks, and I have been curious to see what techniques allow them to achieve that. Apparently the reactive Postgres client used by them supports command pipelining (up to 256 commands by default, and can be changed by the setPipeliningLimit() method of the PgPoolOptions class), and a quick check with Wireshark shows that the database server may return several results in a single TCP packet. Doesn't that conflict with the requirement that every query results in a full round-trip to the server?



Hello,

Another interesting postgres question... I have

Ah-ha! I was also wondering about vertx-postgres. I tried using the same Postgres client with the Undertow framework and wasn't able to get anything close to the same performance. If it's just a matter of pipelining and that's enabled default, I think I should've gotten similar results. I wonder what I did wrong...

Anyway, thanks for solving that puzzle. You're right, that does sound like a violation of the round-trip requirement. Perhaps we can add a setPipeliningLimit(1) line and call it a day. Now I wonder if other implementations are doing this but they're slow enough (for other reasons) that no one has noticed. It's a cool feature.

... several

every query

★ Hello,  
Another interesting point about vertx-postgres. I tried using the  
framework and wasn't able to  
do just a matter of

Hi,

A  
S  
can you elaborate why it violates the Requirements #6 ?

each query is sent as an individual Postgres command (in the PG protocol) and has its own round trip to the database, besides there is no query aggregation or use of batching.

in asynchronous libraries it is very common to implement this way because it is more efficient, and that's what other DB do as well (like <https://redis.io/topics/pipelining>)

regards

Julien

reasons)

★ Hello,

Another interesting point about vertx-postgres. I tried using the framework and wasn't able to make it work. It's just a matter of configuration.

Hi,

A can you elaborate why it violates the Requirements #6 ?

S each query is sent sequentially

In summary, we are leaning toward prohibiting pipelining to keep consistent with the original intent of this test type. If there are more opinions, we'd like to hear them. We'll aim to have a decision one way or the other by roughly the end of the week. Also, bear in mind that any decision could be later reversed.

efficient, and that's what other DB do as well (like <https://redis.io/topics/pipelining>)

regards

Julien

reasons)

Hi,

A  
can you  
each ou

In summary, we are  
If there are more opinions, we'd like to hear them by  
end of the week. Also, bear in mind that any decision

regards

Julien

reasons)

Hello,

Another interesting

**Pipelining can reduce the number of TCP packets but IMO  
most importantly it can keep the DB server busier and this  
is the reason why I believe this should be allowed.**

I tried using the  
framework and wasn't able to  
just a matter of

intent of this test type.  
the other by roughly the

ral

We (TechEmpower) have decided to allow pipelining between the application and database. We'll clarify the requirements to call this out specifically.

regards

Julien

# Let there be pipelining

7. Except where noted, all database queries should be delivered to the database servers as-is and not coalesced or deduplicated at the database driver. In all cases where a database query is required, it is expected that the query will reach and execute on the database server. However, it is permissible to pipeline traffic between the application and database as a network-level optimization. That is, two or more separate queries may be delivered from the application to the database in a single transmission over the network, and likewise for query results from the database back to the application, as long as the queries themselves are executed separately on the database server and not combined into a single complex query.

THE LEGEND





Brown  
Sugar

# 极客时间 VIP 年卡

每天6元, 365天畅看全部技术实战课程

- 20余类硬技能，培养多岗多能的混合型人才
- 全方位拆解业务实战案例，快速提升开发效率
- 碎片化时间学习，不占用大量工作、培训时间



主办方向 Geekbang & InfoQ

# QCon

全球软件开发大会  
北京·2019

更多技术干货分享，北京站精彩继续  
提前参与，还能享受更多优惠

识别二维码  
查看更多  
[2019.qconbeijing.com](http://2019.qconbeijing.com)



- ✓ <https://github.com/AdoptOpenJDK/jitwatch>
- ✓ <https://github.com/jvm-profiling-tools/async-profiler>
- ✓ <https://reactiverse.io/reactive-pg-client/>