Java & PostgreSQL
Performance,
Features
And The Future

Álvaro Hernández <*aht@8kdata.com*>

About ⟨8K⟩ data

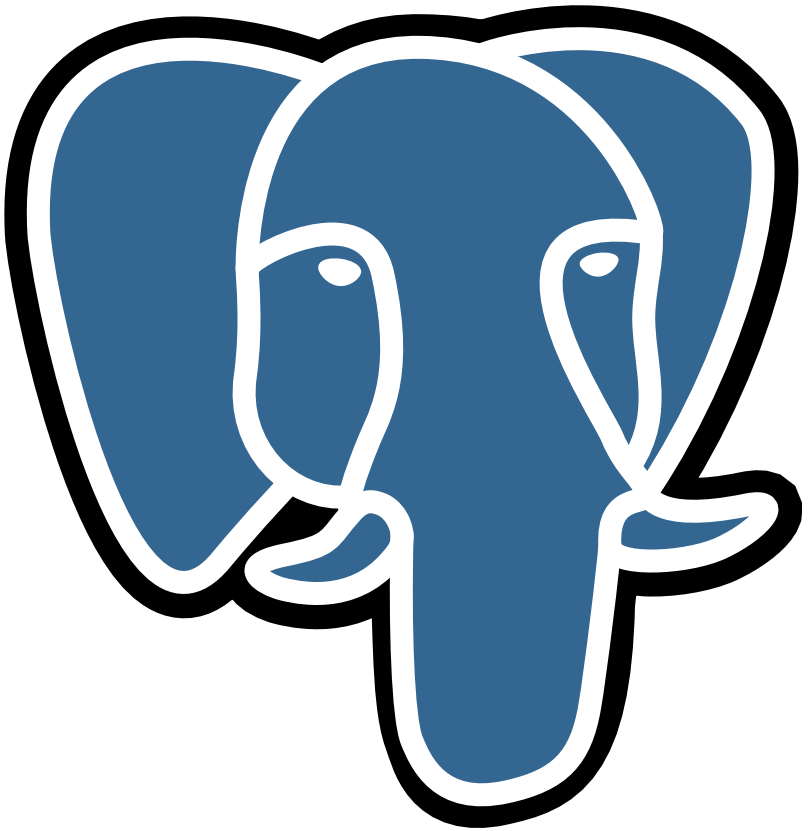
www.8kdata.com

- Research & Development in databases
- Creators of ToroDB.com, NoSQL & SQL database
- Founders of PostgreSQL España, 5th largest PUG in the world (~700 members as of today)
- About myself: CTO at 8Kdata:

@ahachete

http://linkd.in/1jhvzQ3

PostgreSQL & Java



https://www.flickr.com/photos/trevi55/296946221/

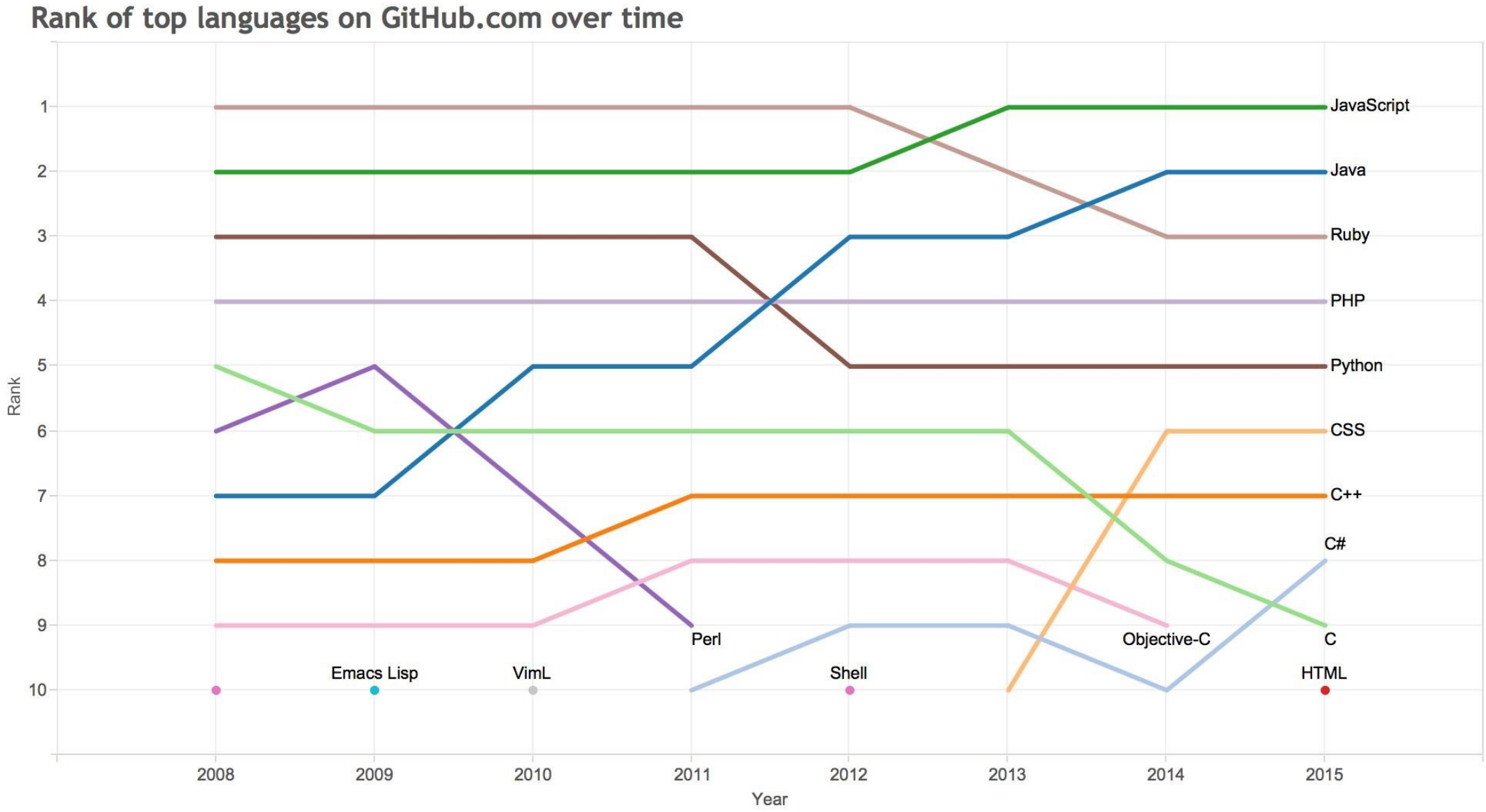
PostgreSQL & Java

- Java IS the enterprise language
- Arguably, there is more Java code accessing PostgreSQL than from any other programming language
- Both Java and PostgreSQL are mature, reliable and trusted

Java: TIOBE index

Feb 2016	Feb 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.145%	+5.80%
2	1	↓	C	15.594%	-0.89%
3	3		C++	6.907%	+0.29%
4	5	↑	C#	4.400%	-1.34%
5	8	↑	Python	4.180%	+1.30%
6	7	↑	PHP	2.770%	-0.40%
7	9	↑	Visual Basic .NET	2.454%	+0.43%
8	12	⇈	Perl	2.251%	+0.86%
9	6	↓	JavaScript	2.201%	-1.31%
10	11	↑	Delphi/Object Pascal	2.163%	+0.59%

Java: GitHub popularity



PYPL: Popularity Programming Languages

Rank	Change	Language	Share	Trend
1		Java	24.2 %	+0.3 %
2	↑	Python	11.9 %	+1.2 %
3	↓	PHP	10.7 %	-0.8 %
4		C#	8.9 %	+0.1 %
5		C++	7.6 %	-0.5 %
6		C	7.5 %	+0.1 %
7		Javascript	7.3 %	+0.3 %
8		Objective-C	5.0 %	-0.9 %
9	↑↑	Swift	3.0 %	+0.4 %
10		R	2.9 %	+0.3 %

PostgreSQL and Java in the past

Java and PostgreSQL haven't mixed well:

- Managed memory vs unmanaged
- Java is (still!) perceived as slow and bloated
- Java requires a runtime (JVM)
- PostgreSQL is ANSI C
- Few Postgres developers like and/or are proficient in Java

How to interact with PostgreSQL from Java

JDBC Driver (pgjdbc)

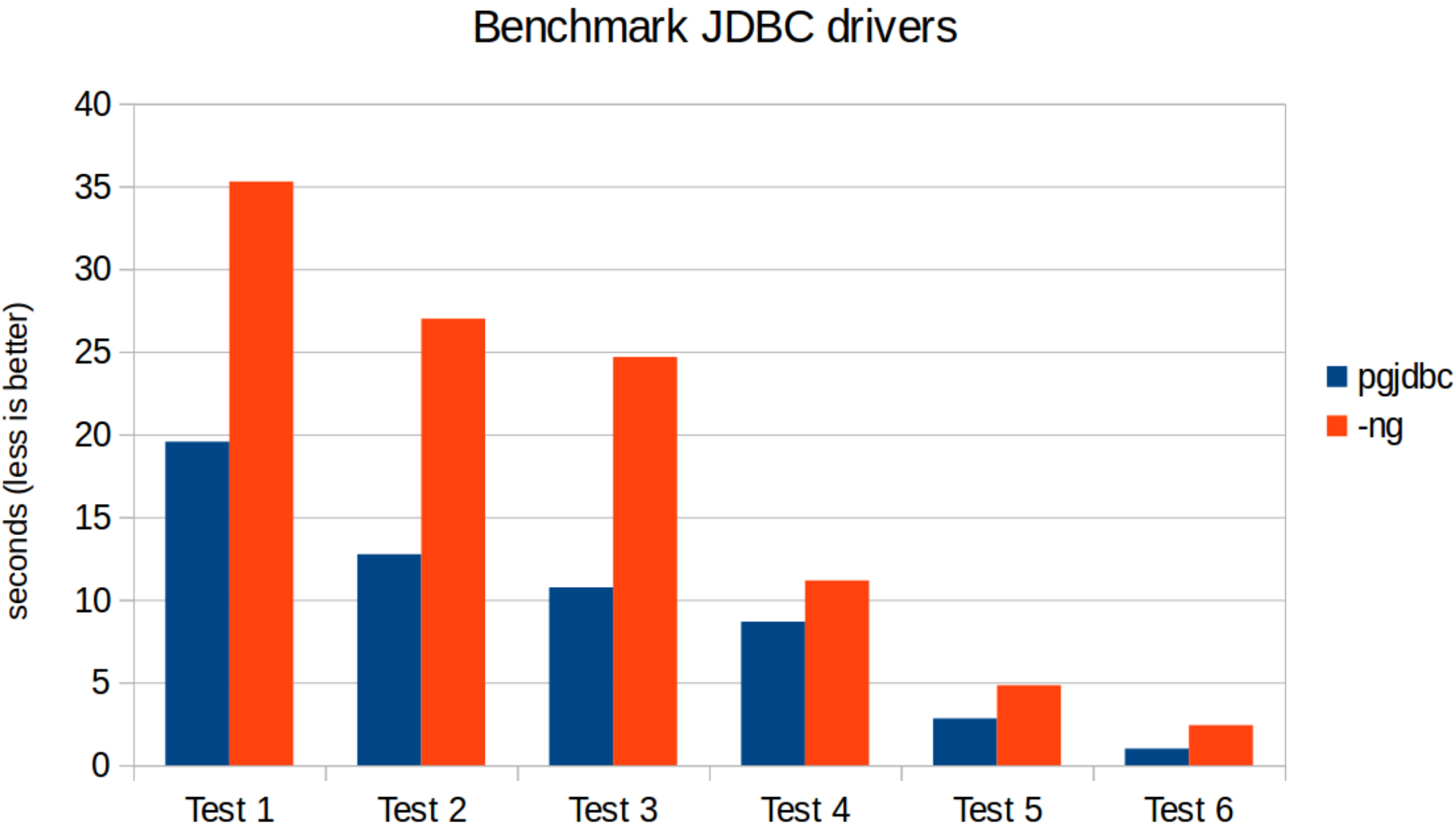
- "Official" driver. Type 4 driver
- Developer base and activity has surged in the last year. Mavenized!
- Latest versions have significantly improved performance
- Solid, reliable choice

Other drivers

pgjdbc-ng

- Modern driver, requieres Java 7
- Uses Netty for network I/O
- Favors binary over text mode
- Goal of being really fast
- Not on par in terms of features with pgjdbc (notably, lacks COPY)
- Latest release: 0.6 (oct 2015)
- https://github.com/impossibl/pgjdbc-ng/releases

Benchmark!


Benchmark JDBC drivers
seconds (less is better)
40
35
30
25
20
15
10
5
0
Test 1
Test 2
Test 3
Test 4
Test 5
Test 6
pgjdbc
-ng

Other drivers

- Progress Type 5 driver
  https://www.progress.com/jdbc/postgresql
  Commercial driver, barely known by community

- PostgreSQL async driver
  https://github.com/mauricio/postgresql-async
  Non-JDBC
  Written in Scala, also supports MySQL
  Netty based
  Active development

Other drivers

- RxJava-jdbc
  https://github.com/davidmoten/rxjava-jdbc
  JDBC generic (not postgres specific)
  All the RxJava goodness!
  Compose queries in serial or parallel
  Map results into tuples or own classes

Non-driver options

- libpq-wrapper
  https://github.com/benfante/libpq-wrapper
  http://benfante.blogspot.ru/2013/02/using-postgresql-in-java-without-jdbc.html

  Uses the SWIG library to wrap PostgreSQL's C library (pq).

  Provides all the functionality of pq (useful for instance for UDS).
  Technique used by EDB for replication.

Server-side Java: pl/java

Recently announced pl/java 1.5

- https://github.com/tada/pljava/releases/tag/V1_5_0
- Coming back! First release since 2011
- Modernized, more active community
- Works with 9.5, Java 6-8 :)

Phoebe (WIP)

- New PostgreSQL driver
- Async & Reactive by design. RxJava based
- Targets clusters, not only individual servers
- Netty-based, async off-heap I/O

Phoebe (WIP)

Expected features:

- Binary mode
- Unix Domain Sockets
- Logical decoding
- Query pipelining
- Fully asynchronous operation
- Execute query on rw or ro nodes
- Fluent-style API
- Compatible with Java >= 6

Phoebe (WIP)

Current API design:

```
RxPostgresClient client = RxPostgresClient
                .create()
                .tcpIp("::1", 5432)
                .tcpIp("localhost", 5433)
                .allHosts()
                .init();
client.onConnectedObservable().subscribe(
    c -> System.out.println(c)
);
```

Measuring the performance

Performance of a query

```
Table "public.number"
```

Column	Type	Modifiers
i	integer	
t	text	
j	json	

```
SELECT * FROM number ORDER BY random() LIMIT 1;
-[ RECORD 1 ]-----------------------------------------------------------
i | 3468053
t | Hello thére 3468053
j | {"i": 3468053, "t": "Hello thére 3468053"}
```

Performance of a query

```java
public class _4_IntStringJson {
    public static final String QUERY = "SELECT i, t, j FROM number";

    private class JsonElements {
        private int i;
        private String t;
    }

    @Benchmark
    public void test(Blackhole blackhole, PgStatStatements pgStatStatements) throws SQLException {
        pgStatStatements.setTestName(QueryBenchmarks.JMHTestNameFromClass(_4_IntStringJson.class));

        Gson gson = new Gson();

        QueryUtil.executeProcessQuery(QUERY, resultSet -> {
            try {
                while (resultSet.next()) {
                    blackhole.consume(resultSet.getInt("i"));
                    blackhole.consume(resultSet.getString("t"));
                    JsonElements jsonElements = gson.fromJson(resultSet.getString("j"), JsonElements.class);
                    blackhole.consume(jsonElements);
                }
            } catch (SQLException e) {}
        });
    }
}
```

Performance of a query

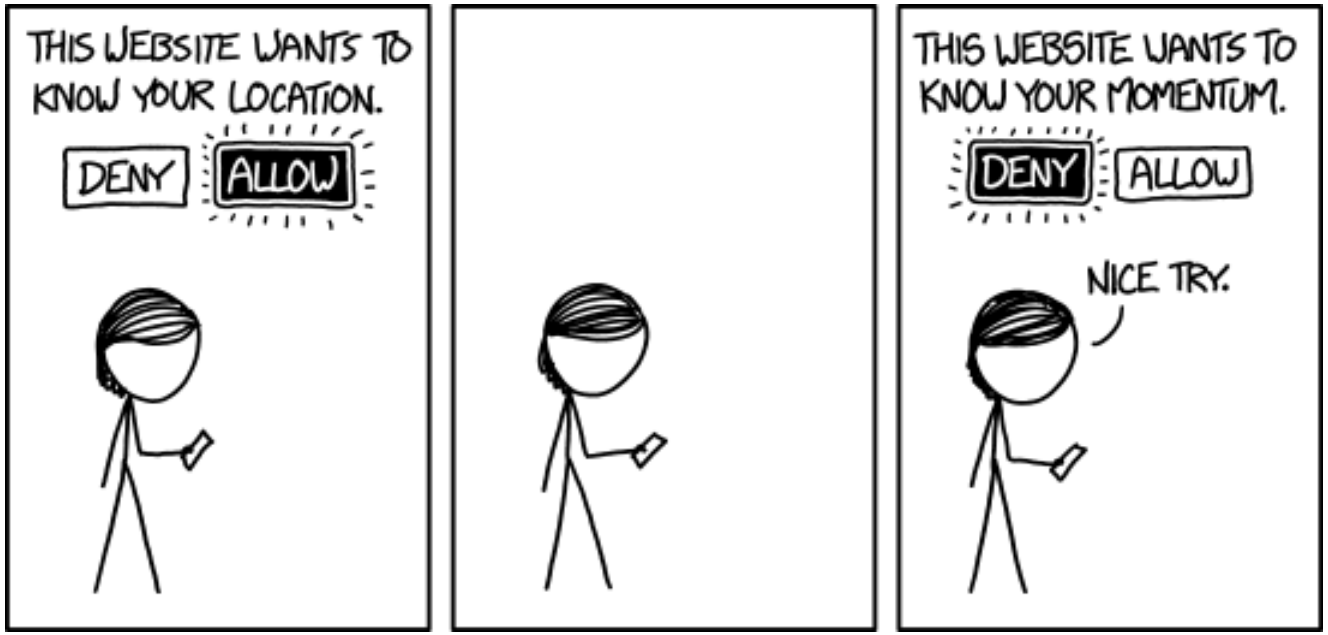
Execution time:

- 17.717ms (avg, 20 runs)

Data facts:

- 10M records
- 1.6Gb table

How do we know how much is spent in the query and how much in Java?



https://xkcd.com/1473/

Let's drill down the execution time

Total Java execution time =

PostgreSQL query execution

+

Network costs (eth + tcp overhed, bw)

+

Java driver overhead

+

Java app processing

Let's drill down the execution time

Using pg_stat_statements we can get the PostgreSQL execution time and infer the rest:

- Java total time = 17.717 ms
- PostgreSQL time = 8.037 ms
- Java overhead is 120%!!!

(run on localhost to minimize network)

Who's the offender? Let's profile!

- Fire up VisualVM's (or your favorite Yourkit, jProfiler, etc) profiler

- And wait
- And make yourself a pizza
- And wait
- And watch a movie

- And then get **very wrong results**!

Sampling profilers to the rescue!

- ➔ Lightweight-java-profiler
- ➔ Honest-profiler
- ➔ Linux perf
- ➔ Oracle Mission Control

- Minimal overhead (0-5%)
- Might not capture everything!
- Not good for low-level performance, but really good for global overview

Using lightweight-java-profiler

```
git clone
https://github.com/dcapwell/lightweight-java-
profiler.git
```

Edit **src/globals.h**:

```
-static const int kNumInterrupts = 100;
+static const int kNumInterrupts = 100000;
-static const int kMaxStackTraces = 3000;
+static const int kMaxStackTraces = 100000;
-static const int kMaxFramesToCapture = 128;
+static const int kMaxFramesToCapture = 1018;
```

```
make
```

(output in build-64/ dir)

Using lightweight-java-profiler

```
java -XX:+PreserveFramePointer
-agentpath:<path>/build-64/liblagent.so
-jar <your.jar>
```

Generates traces.txt

You may need to clean it

```
egrep '^[0-9][0-9]*\ *$' -v traces.txt >
traces2.txt
```

Examine the output!

And now... burn in flames!

And now... burn in flames!

```
Git clone
https://github.com/brendangregg/FlameGraph

<path>/FlameGraph/stackcollapse-ljp.awk
traces.txt | \
<path>/FlameGraph/flamegraph.pl --hash  \
> flameGraph.svg
```

Let's start with simpler examples

```
String QUERY = "SELECT i FROM number";

while (resultSet.next()) {
    blackhole.consume(resultSet.getInt("i"));
}
```

- Java: 3.952 ms
- PG: 3703 ms
- Overhead: 6,7%

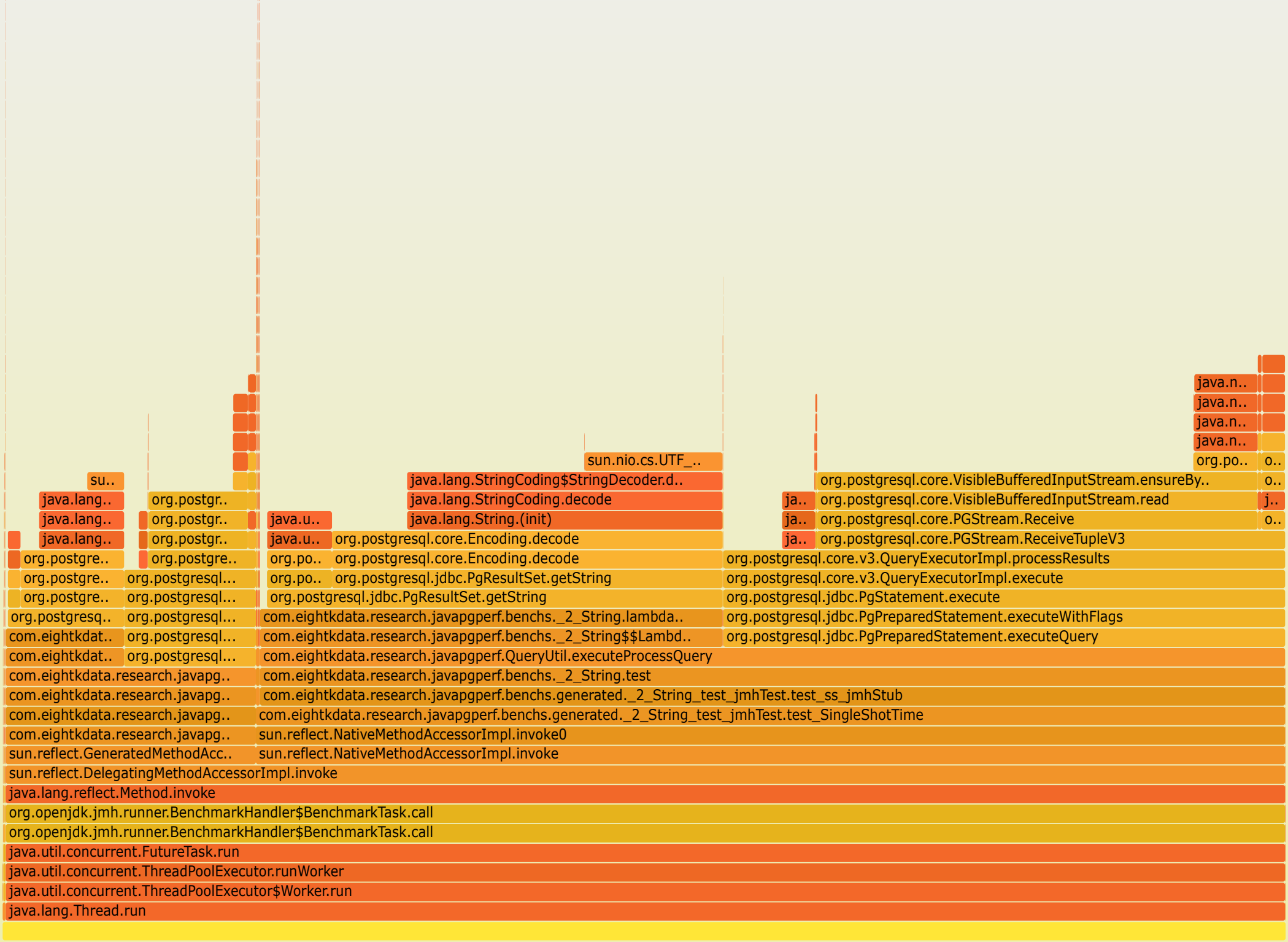

ja..
ja.. jav..
ja.. jav..
ja.. jav..
ja.. org..
or.. org..
or.. org..
java... or.. jav..
java.util.HashMap.g.. java... or.. org..
jav.. java.util.HashMap.get java... org.postgresql.core.PGStream.ReceiveTupleV3
jav.. org.postgresql.cor.. org.postgresql.jdbc.. org.postgresql.core.v3.QueryExecutorImpl.processResults
org.. org.postgresql.core.v3.Q.. org.postgresql.jdbc.. org.. org.postgresql.core.v3.QueryExecutorImpl.execute
org.. org.postgresql.core.v3.Q.. org.postgresql.jdbc.. org.. org.postgresql.jdbc.PgStatement.execute
org.. org.postgresql.jdbc.PgSt.. com.eightkdata.research.ja.. org.postgresql.jdbc.PgPreparedStatement.executeWithFlags
com... org.postgresql.jdbc.PgPr.. com.eightkdata.research.ja.. org.postgresql.jdbc.PgPreparedStatement.executeQuery
com... org.postgresql.jdbc.PgPr.. com.eightkdata.research.javapgperf.QueryUtil.executeProcessQuery
com.eightkdata.research.javapg.. com.eightkdata.research.javapgperf.benchs._1_Int.test
com.eightkdata.research.javapg.. com.eightkdata.research.javapgperf.benchs.generated._1_Int_test_jmhTest.test_ss_jmhStub
com.eightkdata.research.javapg.. com.eightkdata.research.javapgperf.benchs.generated._1_Int_test_jmhTest.test_SingleShotTime
com.eightkdata.research.javapg.. sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.GeneratedMethodAcc.. sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
java.util.concurrent.FutureTask.run
java.util.concurrent.ThreadPoolExecutor.runWorker
java.util.concurrent.ThreadPoolExecutor$Worker.run
java.lang.Thread.run

A String field

```
final String QUERY = "SELECT t FROM number";

while (resultSet.next()) {
    blackhole.consume(resultSet.getString("t"));
}
```

- Java: 3.513 ms
- PG: 6.065 ms
- Overhead: 72%


java.n..
java.n..
java.n..
java.n..
sun.nio.cs.UTF_..
org.po..
o..
su..
java.lang.StringCoding$StringDecoder.d..
org.postgresql.core.VisibleBufferedInputStream.ensureBy..
o..
java.lang..
org.postgr..
java.lang.StringCoding.decode
ja..
org.postgresql.core.VisibleBufferedInputStream.read
j..
java.lang..
org.postgr..
java.u..
java.lang.String.(init)
ja..
org.postgresql.core.PGStream.Receive
o..
java.lang..
org.postgr..
java.u..
org.postgresql.core.Encoding.decode
ja..
org.postgresql.core.PGStream.ReceiveTupleV3
org.postgre..
org.postgre..
org.po..
org.postgresql.core.Encoding.decode
org.postgresql.core.v3.QueryExecutorImpl.processResults
org.postgre..
org.postgresql...
org.po..
org.postgresql.jdbc.PgResultSet.getString
org.postgresql.core.v3.QueryExecutorImpl.execute
org.postgre..
org.postgresql...
org.postgresql.jdbc.PgResultSet.getString
org.postgresql.jdbc.PgStatement.execute
org.postgresq..
org.postgresql...
com.eightkdata.research.javapgperf.benchs._2_String.lambda..
org.postgresql.jdbc.PgPreparedStatement.executeWithFlags
com.eightkdat..
org.postgresql...
com.eightkdata.research.javapgperf.benchs._2_String$$Lambd..
org.postgresql.jdbc.PgPreparedStatement.executeQuery
com.eightkdat..
org.postgresql...
com.eightkdata.research.javapgperf.QueryUtil.executeProcessQuery
com.eightkdata.research.javapg..
com.eightkdata.research.javapgperf.benchs._2_String.test
com.eightkdata.research.javapg..
com.eightkdata.research.javapgperf.benchs.generated._2_String_test_jmhTest.test_ss_jmhStub
com.eightkdata.research.javapg..
com.eightkdata.research.javapgperf.benchs.generated._2_String_test_jmhTest.test_SingleShotTime
com.eightkdata.research.javapg..
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.GeneratedMethodAcc..
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
java.util.concurrent.FutureTask.run
java.util.concurrent.ThreadPoolExecutor.runWorker
java.util.concurrent.ThreadPoolExecutor$Worker.run
java.lang.Thread.run

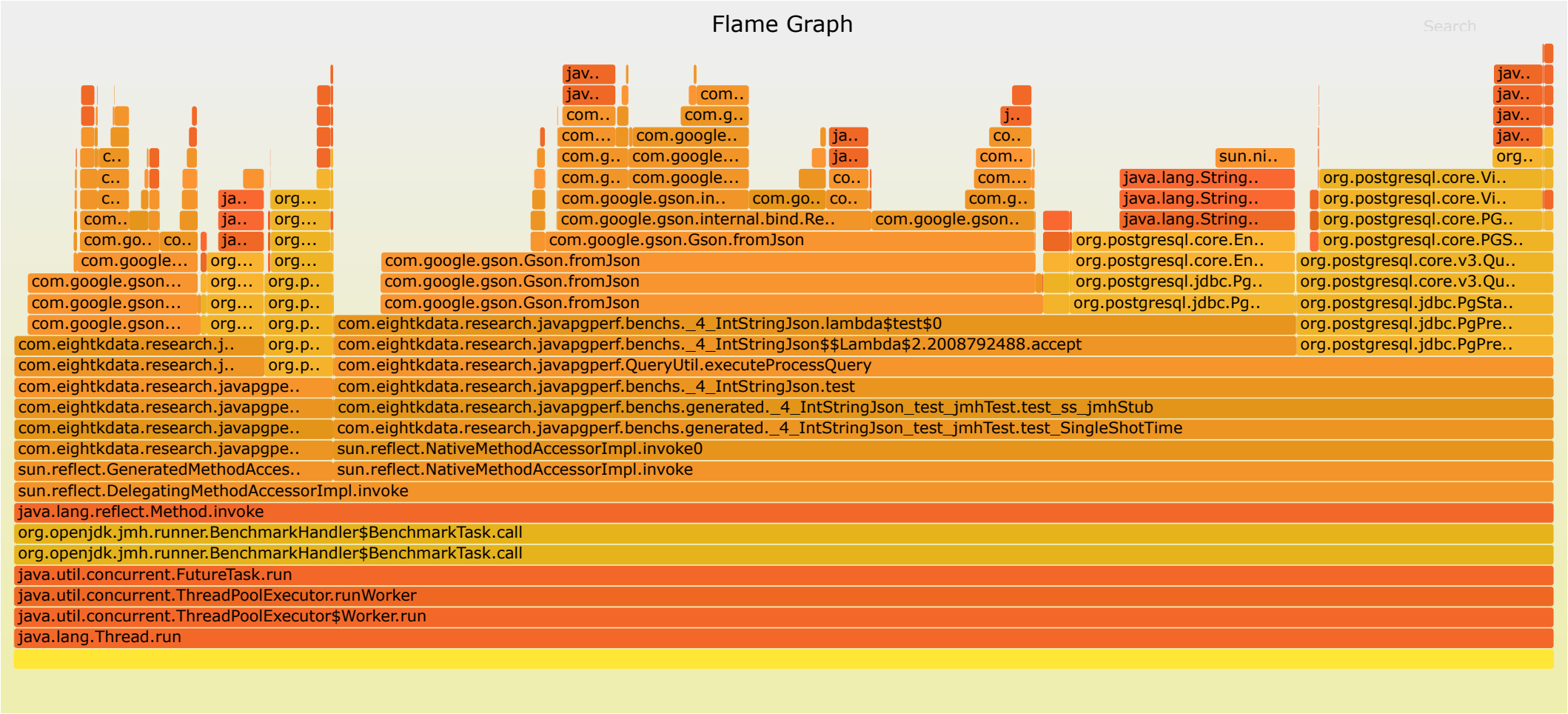
Int, String and JSON fields

```
final String QUERY = "SELECT i, t, j FROM number";

while (resultSet.next()) {
    blackhole.consume(resultSet.getInt("i"));
    blackhole.consume(resultSet.getString("t"));
    JsonElements jsonElements = gson.fromJson(resultSet.getString("j"), JsonElements.class);
    blackhole.consume(jsonElements);
}
```

- Java: 8.037 ms
- PG: 17.017 ms
- Overhead: 120%

Int, String and JSON fields

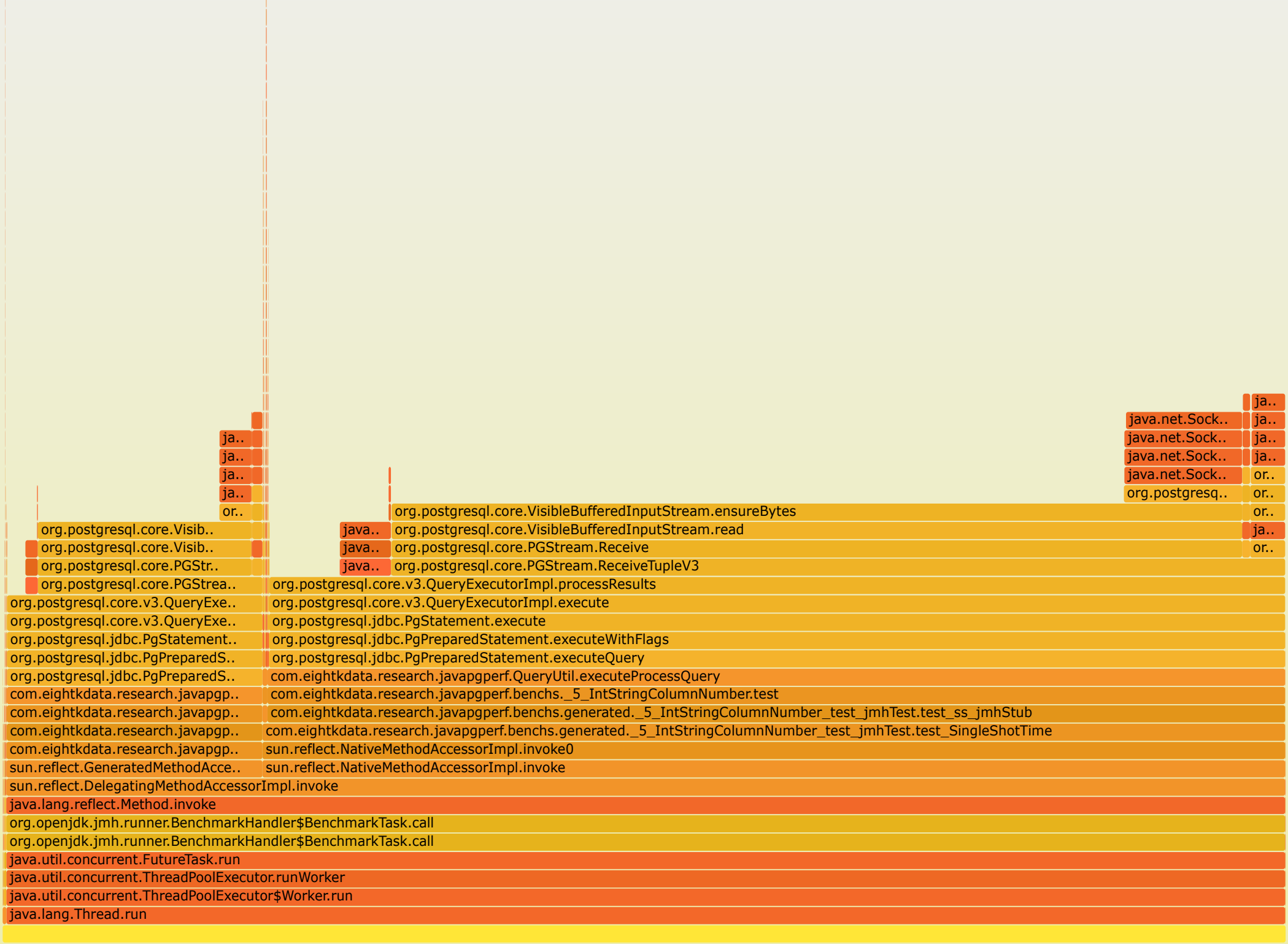


An easy, partial improvement

```
final String QUERY = "SELECT i, t FROM number";

while (resultSet.next()) {
    blackhole.consume(resultSet.getInt(0));
    blackhole.consume(resultSet.getString(1));
}
```

- Java: 5.835 ms
- PG: 6.031 ms
- Overhead: 3,3%


ja..
java.net.Sock..
ja..
ja..
java.net.Sock..
ja..
ja..
java.net.Sock..
ja..
ja..
java.net.Sock..
or..
ja..
org.postgresq..
or..
or..
org.postgresql.core.VisibleBufferedInputStream.ensureBytes
or..
org.postgresql.core.Visib..
java..
org.postgresql.core.VisibleBufferedInputStream.read
ja..
org.postgresql.core.Visib..
java..
org.postgresql.core.PGStream.Receive
or..
org.postgresql.core.PGStr..
java..
org.postgresql.core.PGStream.ReceiveTupleV3
org.postgresql.core.PGStrea..
org.postgresql.core.v3.QueryExecutorImpl.processResults
org.postgresql.core.v3.QueryExe..
org.postgresql.core.v3.QueryExecutorImpl.execute
org.postgresql.core.v3.QueryExe..
org.postgresql.jdbc.PgStatement.execute
org.postgresql.jdbc.PgStatement..
org.postgresql.jdbc.PgPreparedStatement.executeWithFlags
org.postgresql.jdbc.PgPreparedS..
org.postgresql.jdbc.PgPreparedStatement.executeQuery
org.postgresql.jdbc.PgPreparedS..
com.eightkdata.research.javapgperf.QueryUtil.executeProcessQuery
com.eightkdata.research.javapgp..
com.eightkdata.research.javapgperf.benchs._5_IntStringColumnNumber.test
com.eightkdata.research.javapgp..
com.eightkdata.research.javapgperf.benchs.generated._5_IntStringColumnNumber_test_jmhTest.test_ss_jmhStub
com.eightkdata.research.javapgp..
com.eightkdata.research.javapgperf.benchs.generated._5_IntStringColumnNumber_test_jmhTest.test_SingleShotTime
com.eightkdata.research.javapgp..
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.GeneratedMethodAcce..
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
org.openjdk.jmh.runner.BenchmarkHandler$BenchmarkTask.call
java.util.concurrent.FutureTask.run
java.util.concurrent.ThreadPoolExecutor.runWorker
java.util.concurrent.ThreadPoolExecutor$Worker.run
java.lang.Thread.run

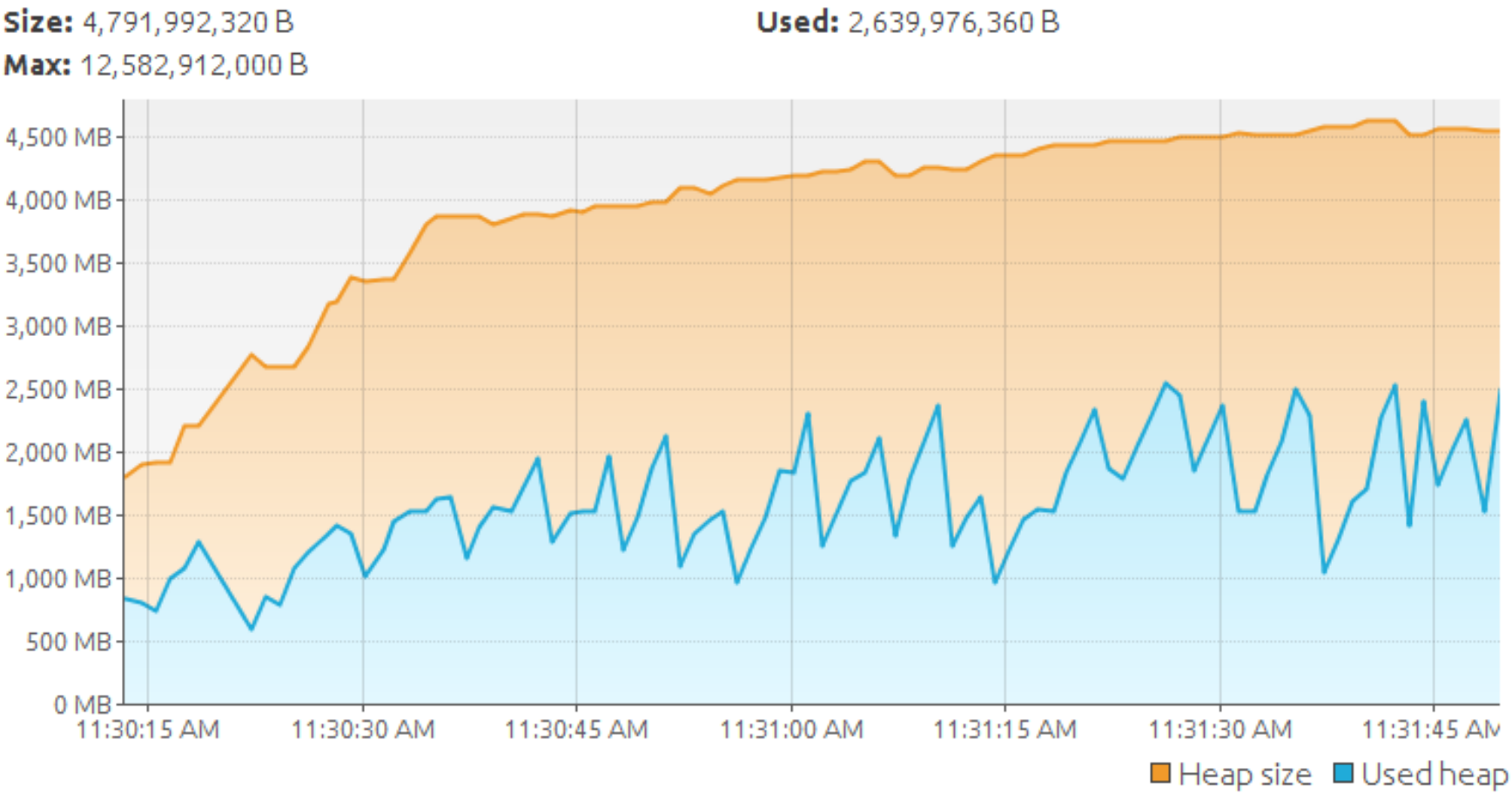
Another easy fix, courtesy of Vladimir

```
Properties properties = new Properties();
PGProperty.DEFAULT_ROW_FETCH_SIZE.set(properties, 1000);
PGProperty.USER.set(properties, "aht");
PGProperty.PASSWORD.set(properties, "aht");
try(Connection connection = DriverManager.getConnection("jdbc:postgresql://localhost:5432/test", properties)) {
    connection.setAutoCommit(false);
    try(PreparedStatement preparedStatement = connection.prepareStatement(query)) {
        ResultSet resultSet = preparedStatement.executeQuery();
        action.accept(resultSet);
    }
    connection.commit();
}
```

- Java: 2.818 ms
- PG: 4.041 ms
- Overhead: 43,3%

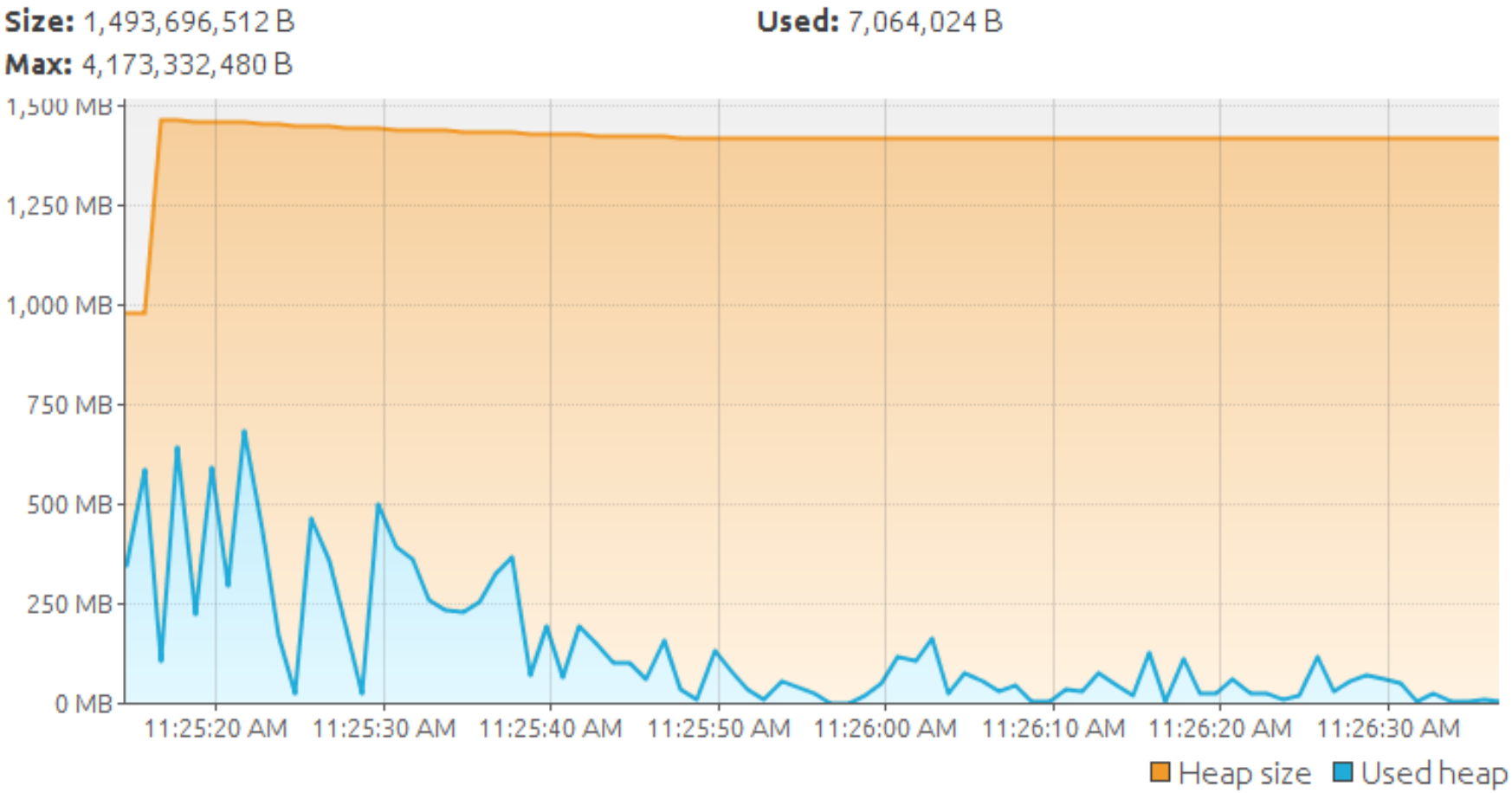
(String test; was 72% overhead)

Another easy fix, courtesy of Vladimir



➔ Before, without fetch size being used

Another easy fix, courtesy of Vladimir



- After, fetch size = 1000

Some conclusions

- Identify your hot spots on Flame Graphs
- The profile + jmh methods to find out problems at micro level
- Lots of searches on HashMap may eat your CPU, because of equals. Pool!
- All the results would be worse over LAN

What else? Introducing pgj

- Marshalling/unmarshalling accounts for most of the driver overhead
- Copy from your data format to wire format. Object creation, GC… even if strings are sent in UTF-8, they are parsed
- What if we could avoid… the network?

What else? Introducing pgj

- Run Java code **inside** PostgreSQL
- pl/java is cool but not enough: it has to be like a *server*, run independently of the user
- Background workers provide the base
- Wrap SPI calls with JNI and profit!

Accessing PostgreSQL's functionality from Java

Features not available for Java users

- json and jsonb types are serialized as text over the network!
- Indeed, most custom or "complex" types are also sent in text format
- Logical decoding requires postgres protocol support. Coming to pgjdbc! https://github.com/pgjdbc/pgjdbc/pull/550

Features not available for Java users

- No support for async queries (they are possible with the current protocol)
- No built-in support for UNIX Domain Sockets (but available with minor work)
- Limited multi-server support (just round-robin servers, not cluster-aware, read-only and read-write replica operation)

How to solve these limitations?

- Submit PRs to pgjdbc
- Join the Phoebe project, which aims to address some (most) of the previous limitations
- Join the pgj project and run your Java code inside PostgreSQL with minimal overhead.


8K
data