Yandex Mail

Design and Implementation of DBMS Asynchronous Client Library

Roman Siromakha

About

What are we doing?

- Mail service
  - Persistent storage (relational database)
  - Consistent data
  - Response time optimization
  - C++ microservices
- Highload
  - ~400TB mailbox metadata at over 100 database shards
  - ~200M users
  - ~10M active users each day

What do we use?

- PostgreSQL
  - Relational DBMS
  - Composite types
  - User defined types
  - Binary protocol

What do we need from client library?

- Minimum runtime cost
- Help to avoid code mistakes
- Simple to use

Why do we make yet another library?

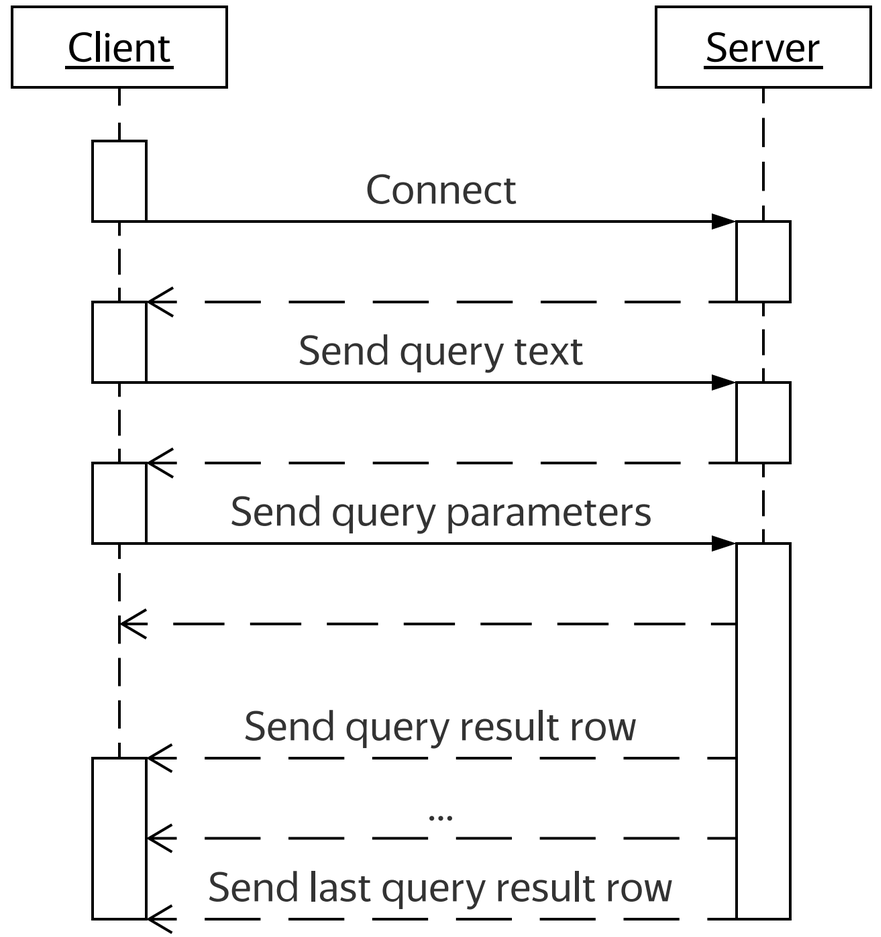
- libpq
  - Not even C++
  - Require boilerplate code
- libpqxx
  - Only text protocol
  - Synchronous I/O

What is our proposal?

- Asynchronous I/O
- Prefer compile time over runtime
- Compile time and runtime checks
- Customizable interface

Ozo

Database communication model



Brief example

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>();
ozo::connection_info<decltype(oid_map)> connection_info(io, "host=localhost");
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool);
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + uid
    + "  AND mid = ANY("_SQL + mids + "::code.mids)"_SQL;
using row_t = std::tuple<int, std::string, std::vector<attach>>;
std::vector<row_t> result;
auto connection = ozo::request(provider, query, std::back_inserter(result),
    asio::use_future);
```

How does it work?

- Boost.Asio for asynchronous network communication
- Boost.Hana for parameters and result introspection
- Boost.Spirit.X3 for query configuration parsing
- Universal asynchronous interface: callbacks, futures, coroutines
- Bind parameters to avoid SQL-injections
- Transfer binary data to minimize traffic and CPU usage for parsing

Connection

Connection abilities

- Provide socket
- Provide bound types
- Provide error context
- Optionally provide statistics storage

Connection provider

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>();
ozo::connection_info<decltype(oid_map)> connection_info(io,
        "host=localhost"); // connection provider

ozo::connection_pool_config config;
config.queue_timeout = 300ms;
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool); // connection provider
```

Connection provider concept

```
template <typename, typename = std::void_t<>>
struct is_connection_provider : std::false_type {};

template <typename T>
struct is_connection_provider<T, std::void_t<decltype(
    async_get_connection(
        std::declval<T&>(),
        std::declval<std::function<void(error_code, connectable_type<T>)>>()
    )
)>> : std::true_type {};

template <typename T>
constexpr auto ConnectionProvider =
    is_connection_provider<std::decay_t<T>>::value;
```

Connection concept

```
template <typename, typename = std::void_t<>>
struct is_connection : std::false_type {};

template <typename T>
struct is_connection<T, std::void_t<
    decltype(get_connection_socket(std::declval<T&>())),
    decltype(get_connection_oid_map(std::declval<T&>())),
    decltype(get_connection_error_context(std::declval<T&>())),
    decltype(get_connection_statistics(std::declval<T&>())),
    // same for const ...
>> : std::true_type {};

template <typename T>
constexpr auto Connection = is_connection<std::decay_t<T>>::value;
```

User connection type

```
struct my_conn {
    // ...
};

auto& get_connection_oid_map(my_conn& conn) noexcept {/* ... */ }
auto& get_connection_socket(my_conn& conn) noexcept { /* ... */ }
auto& get_connection_error_context(my_conn& conn) noexcept { /* ... */ }
auto& get_connection_statistics(my_conn& conn) noexcept { /* ... */ }
```

Connection wrapper concept

```
template <typename T>
struct is_nullable : std::false_type {};
template <typename T>
struct is_nullable<std::shared_ptr<T>> : std::true_type {};
// ... std::unique_ptr ...
// ... std::weak_ptr ...
// ... std::optional ...
// ... boost:: ...
template <typename T>
constexpr auto ConnectionWrapper = is_connection_wrapper<std::decay_t<T>>::value;
```

Unwrap connection

```
template <bool Condition, typename Type = void>
using Require = std::enable_if_t<Condition, Type>;
// ...
template <typename T>
decltype(auto) unwrap_connection(T&& conn,
        Require<!ConnectionWrapper<T>>* = 0) noexcept {
    return std::forward<T>(conn);
}

template <typename T>
decltype(auto) unwrap_connection(T&& conn,
        Require<ConnectionWrapper<T>>* = 0) noexcept {
    return unwrap_connection(*conn);
}
```

Connection and wrapper combination

```
template <typename, typename = std::void_t<>>
struct is_connectable : std::false_type {};

template <typename T>
struct is_connectable <T, std::void_t<
    decltype(unwrap_connection(std::declval<T&>()))
>> : std::integral_constant<bool, Connection<
    decltype(unwrap_connection(std::declval<T&>()))
>> {};

template <typename T>
constexpr auto Connectable = is_connectable<std::decay_t<T>>::value;
```

Why concepts?

- Support custom implementation: good for unit tests
- Minimum runtime overhead: users define only what they need
- Support any kind of wrappers: reference, pointers, optional

Dark side of concepts

- Compilers output for an error in template code
- Only template code

Use static asserts to check your type

```
static_assert(ozo::Connection<my_conn>,
              "my_conn does not meet Connection requirements");
static_assert(ozo::Connectable<my_conn>, /* ... */);
static_assert(ozo::ConnectionWrapper<my_conn_wrapper>, /* ... */);
static_assert(ozo::Connectable<my_conn_wrapper>, /* ... */);
```

Type system

PostgreSQL types

- PostgreSQL has builtin types with predefined OIDs
- PostgreSQL store user types in special table
- Query parameter and result has OID for each value

Back to the request example

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>(); // <-- Look closer here
ozo::connection_info<decltype(oid_map)> connection_info(io, "host=localhost");
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool);
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + uid
    + "  AND mid = ANY("_SQL + mids + "::code.mids)"_SQL;
using row_t = std::tuple<int, std::string, std::vector<attach>>;
std::vector<row_t> result;
auto connection = ozo::request(provider, query, std::back_inserter(result),
    asio::use_future);
```

User defined type

```
CREATE TYPE mail.attach AS (filename text, type text, size bigint);
```

```
struct attach {
    std::string filename;
    std::string type;
    std::int64_t size;
};
BOOST_HANA_ADAPT_STRUCT(attach, filename, type, size);
// or BOOST_FUSION_ADAPT_STRUCT(attach, filename, type, size);
OZO_PG_DEFINE_CUSTOM_TYPE(attach, "mail.attach", dynamic_size);
```

OID map usage

```
auto oid_map = ozo::register_types<attach, /* ... */>();

ozo::set_type_oid<attach>(oid_map, /* ... */);

ozo::type_oid<attach>(oid_map);
```

```
if (ozo::accepts_oid<attach>(oid_map, oid)) {
    // ...
}
```

OID map storage

```
auto impl = hana::make_map(
    hana::make_pair(hana::type_c<T>, null_oid()) ...
);

impl[hana::type_c<T>] = oid;
```

```
ozo::type_oid<attach>(oid_map); // ok

struct recipient {};
ozo::type_oid<recipient>(oid_map); // compile error
```

Query builder

Back to the request example

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>();
ozo::connection_info<decltype(oid_map)> connection_info(io, "host=localhost");
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool);
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + uid
    + "  AND mid = ANY("_SQL + mids + "::code.mids)"_SQL; // <-- Look close here
using row_t = std::tuple<int, std::string, std::vector<attach>>;
std::vector<row_t> result;
auto connection = ozo::request(provider, query, std::back_inserter(result),
    asio::use_future);
```

Ideas for query builder

- Query text is defined as type or value
- Parameters is part of query (both types and values)
- Serialize parameters by using introspection

How to send a query to PostgreSQL?

```
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + 42
    + "  AND mid = ANY("_SQL + {1, 2, 3} + "::code.mids)"_SQL;
```

```
SELECT mid, st_id, attaches
  FROM mail.messages
 WHERE uid = $1
   AND mid = ANY($2::code.mids)
```

Query template

```
std::int64_t uid;
std::string st_id;
std::int64_t mid;
std::vector<attach> attaches;
// ...
const auto query = (
    "INSERT INTO mail.messages (uid, mid, st_id, attaches)"_SQL
    + "VALUES ("_SQL + std::cref(uid)
        + ", "_SQL + std::cref(uid)
        + ", "_SQL + std::cref(st_id)
        + ", "_SQL + std::cref(attaches) + ")"_SQL
).build();
// ...
std::tie(uid, mid, st_id, attaches) = /*...*/;
```

Query builder: compile time tests

```
constexpr auto query = "SELECT "_SQL + 42 + " + "_SQL + 13;

static_assert(query.text() == "SELECT $1 + $2"_s,
              "query_builder should generate text in compile time");

static_assert(query.params() == hana::tuple<int, int>(42, 13),
              "query_builder should generate params in compile time");
```

What does it give us?

- Avoid boilerplate code
- Code type-safety (value provide type)
- Database type-safety (check for database type support)
- Send what you mean (bind type identifier and value)

Query configuration

Why query configuration?

- Change query without rebuild
- Query depends on environment

Query configuration file

```
-- name: GetMessages
-- attributes: ...
SELECT mid, st_id, attaches
  FROM mail.messages
 WHERE uid = :uid
   AND mid = ANY(:mids::code.mids)
```

Query type definition

```
using namespace boost::hana::literals;
struct GetMessages {
    static constexpr auto name = "GetMessages"_s;
    struct parameters_type {
        std::int64_t uid;
        std::vector<std::int64_t> mids;
    };
};
BOOST_HANA_ADAPT_STRUCT(GetMessages::parameters_type, uid, mids);
// or BOOST_FUSION_ADAPT_STRUCT(GetMessages::parameters_type, uid, mids);
```

Query repository

```
std::string_view query_conf_text;
const auto query_repo = ozo::make_query_repository(
        query_conf_text, hana::tuple<GetMessages>());
GetMessages::parameters_type parameters;
parameters.uid = 42;
parameters.mids = std::vector<std::int64_t>({1, 2, 3});
const auto query = query_repo.make_query<GetMessages>(parameters);

auto parameters = std::make_tuple(42, std::vector<std::int64_t>({1, 2, 3}));
query_repo.make_query<GetMessages>(parameters); // compile error
```

Database result

Back to the request example

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>();
ozo::connection_info<decltype(oid_map)> connection_info(io, "host=localhost");
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool);
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + uid
    + "  AND mid = ANY("_SQL + mids + "::code.mids)"_SQL;
using row_t = std::tuple<int, std::string, std::vector<attach>>;
std::vector<row_t> result; // <-- Look closer here
auto connection = ozo::request(provider, query, std::back_inserter(result),
    asio::use_future);
```

Ideas for result

- User allocate place
- Allow to use raw result
- Deserialization by using introspection
- Check types OIDs

Raw database result

```
ozo::result result;
ozo::request(provider, query, result, yield);
std::for_each(result.begin(), result.end(), [&] (const auto& row) {
    std::for_each(row.begin(), row.end(), [&] (const auto& value) {
        std::copy(value.data(), value.data() + value.size(), /*...*/);
    });
});
```

Typed row

```
struct message {
    std::int64_t mid;
    std::string st_id;
    std::vector<attach> attaches;
};

BOOST_HANA_ADAPT_STRUCT(message, mid, st_id, attaches);
// or BOOST_FUSION_ADAPT_STRUCT(message, mid, st_id, attaches);

std::vector<message> rows;
ozo::request(provider, query, std::back_inserter(rows), asio::use_future);
```

Boost.Hana deserialization problem

```
template <class T>
void read(std::istream& stream, T& value) {
    stream >> value;
}

attach a;
std::istringstream in("story.txt text/plain 146");
hana::for_each(hana::members(a), [&] (auto& v) { read(in, v); }); // compile error
fusion::for_each(a, [&] (auto& v) { read(in, v); }); // ok
```

Boost.Hana deserialization problem solution

```
const auto a = hana::unpack(
    hana::fold(hana::members(attach {}), hana::tuple<>(),
        [&] (auto&& r, auto&& v) {
            std::decay_t<decltype(v)> out;
            read(in, out);
            return hana::append(std::move(r), std::move(out));
        }),
    [] (auto&& ... args) { return attach {std::move(args) ...}; }
);
```

Request

Back to the request example

```
asio::io_context io;
const auto oid_map = ozo::register_types<attach>();
ozo::connection_info<decltype(oid_map)> connection_info(io, "host=localhost");
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(io, pool);
const auto query = "SELECT mid, st_id, attaches"_SQL
    + " FROM mail.messages "_SQL
    + "WHERE uid = "_SQL + uid
    + "  AND mid = ANY("_SQL + mids + "::code.mids)"_SQL;
using row_t = std::tuple<int, std::string, std::vector<attach>>;
std::vector<row_t> result;
auto connection = ozo::request(provider, query, std::back_inserter(result),
    asio::use_future); // <-- Look close here
```

Asynchronous completion token

```
ozo::request(provider, query, result, [] (error_code ec, auto conn) {/*...*/});

auto future = ozo::request(provider, query, result, asio::use_future);
auto conn = future.get();

[coro = asio::coroutine {}] (error_code ec, auto conn) {
    reenter (coro) { while (true) {
        yield ozo::request(provider, query, result, *this);
    } } }

asio::spawn(io, [] (asio::yield_context yield) {
    auto conn = ozo::request(provider, query, result, yield);
});
```

Convert completion token into callback

```
template <typename P, typename Q, typename Out,
    typename CompletionToken, typename = Require<ConnectionProvider<P>>>
auto request(P&& provider, Q&& query, Out&& out, CompletionToken&& token) {
    using signature_t = void (error_code, connectable_type<P>);
    async_completion<CompletionToken, signature_t> init(token);

    async_request(std::forward<P>(provider), std::forward<Q>(query),
                  std::forward<Out>(out), init.completion_handler);

    return init.result.get();
}
```

Transactions: naive implementation

```
auto conn = ozo::get_connection(conn_info, yield);
ozo::result result;
ozo::request(conn, "BEGIN"_SQL, result, yield);
// ...
ozo::request(conn, "COMMIT"_SQL, result, yield);
ozo::request(conn, "COMMIT"_SQL, result, yield); // database warning
// ...
ozo::request(conn, "BEGIN"_SQL, result, yield);
ozo::request(conn, "BEGIN"_SQL, result, yield); // database warning
```

Transactions: concept implementation

```
ozo::result result;
auto transaction = ozo::transaction(conn_info, result, yield);
// ...
ozo::request(transaction, "..."_SQL, result, yield);
// ...
auto conn = ozo::commit(transaction, result, yield);
// ...
ozo::rollback(conn, result, yield); // compile error
transaction = ozo::transaction(conn, result, yield); // ok
```

Read database result row by row

- Unknown or expected large result size
- Streaming

Read database result row by row

```
ozo::cursor cursor;
ozo::request(conn, query, cursor, yield);
row_t row;
while (ozo::read_count(ozo::fetch(cursor, row, yield))) {
    process(row);
}
```

Read database result row by row: stackless coroutines

```
const auto cursor = make_shared<ozo::cursor>();
const auto row = make_shared<row_t>();
ozo::request(conn, query, *cursor,
    [cursor, row, coro = coroutine {}] (error_code ec, auto conn) mutable {
        if (!ec) reenter(coro) {
            while (true) {
                yield fetch(*cursor, *row, *this);
                if (!ozo::read_count(conn)) return;
                process(row);
            }
            // ...
```

•

```
asio::io_context io;
asio::io_context::strand strand(io);
asio::spawn(io, [&] (asio::yield_context yield) {
    asio::async_completion<asio::yield_context, void (std::shared_ptr<int>)>
        init(yield);
    asio::post(io, [&io, &strand, h = std::move(init.completion_handler)] {
        asio::post(io, strand.wrap([h = std::move(h),
            ptr = std::make_shared<int>()] () mutable {
                h(std::move(ptr));
            }));
    });
    const auto ptr = init.result.get();
    std::cout << ptr.use_count() << std::endl;
});
io.run();
```

Use latest Boost.Asio API

```
asio::spawn(io, [&] (asio::yield_context yield) {
    asio::async_completion<asio::yield_context, void (std::shared_ptr<int>)>
        init(yield);
    asio::post(io, [&io, &strand, h = std::move(init.completion_handler)] {
        asio::post(io, asio::bind_executor(strand, [h = std::move(h),
            ptr = std::make_shared<int>()] () mutable {
                h(std::move(ptr));
            }));
    });
    const auto ptr = init.result.get();
    std::cout << ptr.use_count() << std::endl;
});
```

•

```
void my_deep_recursion() {
    if (!stop_it) {
        my_deep_recursion();
    }
}
```

```
asio::spawn(io, [] (asio::yield_context yield) {
    // ...
    my_deep_recursion();
    // ...
});
```

Check coroutine stack size

```
void my_deep_recursion() {
    if (!stop_it) {
        my_deep_recursion();
    }
}
```

```
asio::spawn(io, [] (asio::yield_context yield) {
    // ...
    my_deep_recursion();
    // ...
}, coroutine::attributes(stack_size));
```

Tests

Unit tests framework

- GUnit
  - Based on googletest
  - Textual test description
  - Mocks without macro
    - Virtual table patching: `-fdevirtualize` leads to SIGSEGV
    - Runtime errors from Undefined Behavior Sanitizer

Unit tests with mocks

```
GTEST("ozo::async_connect()") {
    SHOULD("start connection, assign socket and wait for write complete") {
        // ...
        connection_mock conn_mock{};

        EXPECT_CALL(conn_mock, (start_connection)("conninfo"))
            .WillOnce(Return(error_code{}));
        EXPECT_CALL(conn_mock, (assign_socket)()).WillOnce(Return(error_code{}));
        EXPECT_CALL(conn_mock, (write_poll)(_));

        ozo::impl::async_connect("conninfo", conn, wrap(cb_mock));
    }
```

Unit tests with mocks

```
struct connection {
    connection_mock* mock_;
    template <typename Handler>
    friend void pq_write_poll(connection& c, Handler&& h) {
        c.mock_->write_poll([h] (auto e) {
            asio_post(ozo::detail::bind(h, e));
        });
    }
};
struct connection_mock {
    virtual void write_poll(handler) const = 0;
};
```

Docker image to build the project

```
FROM ubuntu:17.10
RUN apt-get update && apt-get install -y ccache clang-5.0 ...
RUN wget -qO boost_1_67_0.tar.gz https://sourceforge.net/projects/boost/... && \
    tar xzf boost_1_67_0.tar.gz && \
    cd boost_1_67_0 && \
    ./bootstrap.sh --with-libraries=system,thread,chrono,... && \
    ./b2 ...
VOLUME /ccache
VOLUME /code
WORKDIR /code
ENV CCACHE_DIR=/ccache
```

Docker-compose to organize containers

```
version: '3'
services:
  ozo_build:
    build: docker/build
    image: ozo_build
    volumes: ["~/.ccache:/ccache", ".:/code"]
  ozo_postgres: {"image": "postgres:alpine", "networks": ["ozo"]}
  ozo_build_with_pg_tests:
    build: docker/build
    image: ozo_build
    depends_on: ["ozo_postgres"]
    volumes: ["~/.ccache:/ccache", ".:/code"]
    networks: ["ozo"]
networks: [{"ozo": {}}]
```

Build and run tests

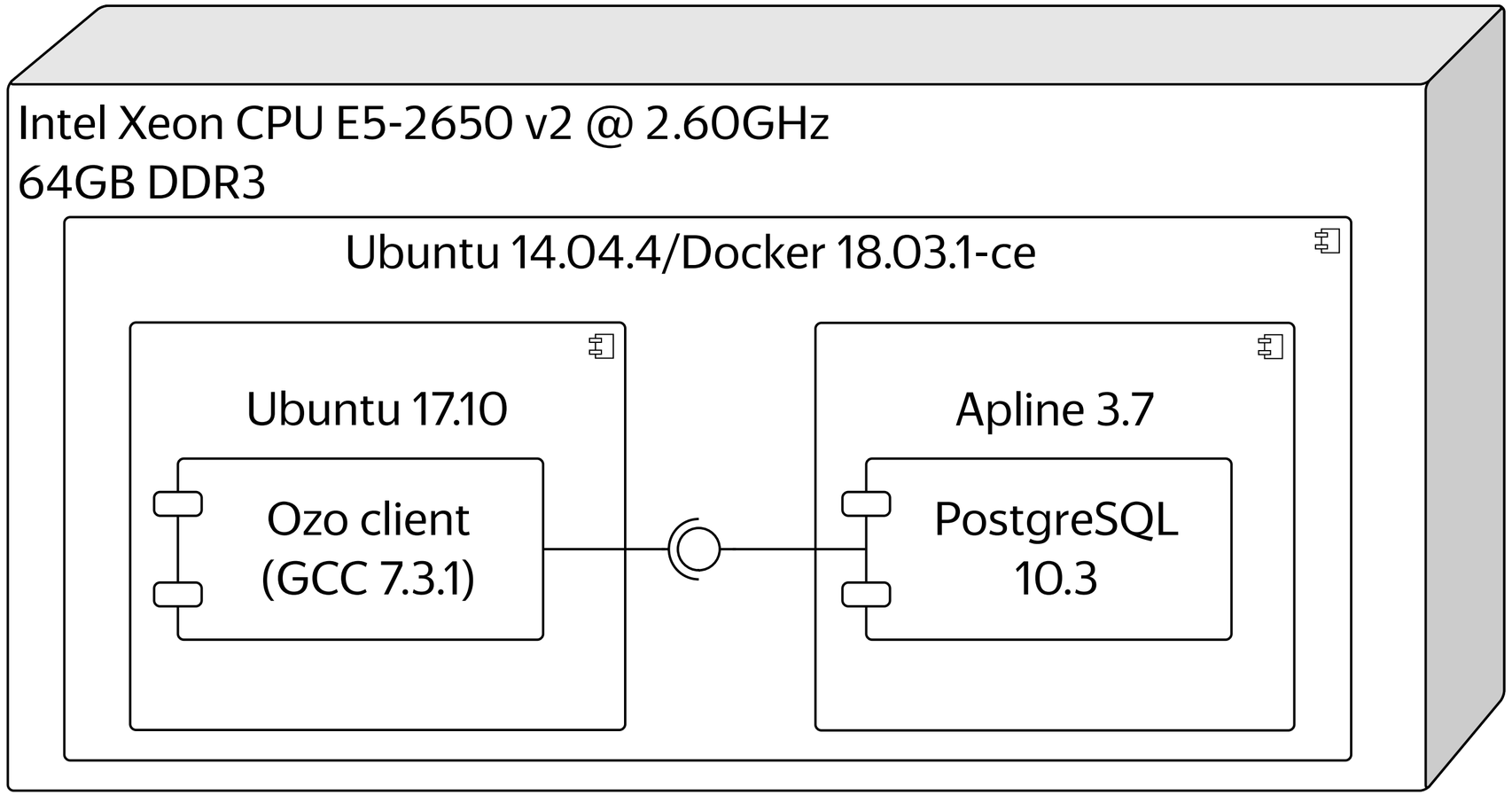
```
# only with unit tests
docker-compose run --rm --user "$(id -u):$(id -g)" ozo_build scripts/build.sh

# with integration tests
docker-compose up -d ozo_postgres
docker-compose run --rm --user "$(id -u):$(id -g)" \
    ozo_build_with_pg_tests scripts/build.sh
docker-compose stop ozo_postgres
docker-compose rm ozo_postgres

#scripts/build.sh
cmake
make
ctest
```

Integration test example

```
GTEST("ozo::request") {
    SHOULD("return selected variable") {
        ozo::io_context io;
        ozo::connection_info<> conn_info(io, OZO_PG_TEST_CONNINFO);
        ozo::result res;
        ozo::request(conn_info, "SELECT "_SQL + std::string("foo"), res,
        [&] (ozo::error_code ec, auto conn) {
            ASSERT_FALSE(ec) << ec.message() << " | " << error_message(conn)
                << " | " << get_error_context(conn);
            EXPECT_EQ(1, res.size());
            EXPECT_EQ(1, res[0].size());
            EXPECT_EQ(std::string("foo"), res[0][0].data());
            EXPECT_FALSE(ozo::connection_bad(conn));
        });
```

Performance

Benchmarks setup



Basic benchmark

```
Benchmark benchmark;
asio::io_context io(1);
ozo::connection_info<> connection_info(io, conn_string);
const auto query = "SELECT 1"_SQL.build();
asio::spawn(io, [&] (auto yield) {
    while (true) {
        ozo::result result;
        ozo::request(connection_info, query, result, yield);
        if (!benchmark.step(result.size())) {
            break;
        }
    }
});
io.run();
```

Basic benchmark

request speed, req/sec	request time (90% quantile), ms
493	2.135

Reuse connection

```
// ...
asio::spawn(io, [&] (auto yield) {
    auto connection = ozo::get_connection(connection_info, yield);
    while (true) {
        ozo::result result;
        ozo::request(connection, query, result, yield);
        if (!benchmark.step(result.size())) {
            break;
        }
    }
});
// ...
```

Reuse connection vs basic benchmark

name	request speed, req/sec	request time, us
basic	493	2135 (2346%)
reuse connection	11353	91 (100%)

Use connection pool

```
// ...
auto pool = ozo::make_connection_pool(connection_info, config);
auto provider = ozo::make_provider(pool, io);
asio::spawn(io, [&] (auto yield) {
    while (true) {
        ozo::result result;
        ozo::request(provider, query, result, yield);
        if (!benchmark.step(result.size())) {
            break;
        }
    }
});
// ...
```

Use connection pool vs reuse connection

name	request speed, req/sec	request time, us
reuse connection	11353	91 (100%)
connection pool	9902	107 (121%)

Use connection pool and parsed result

```
// ...
asio::spawn(io, [&] (auto yield) {
    while (true) {
        std::vector<std::tuple<std::int32_t>> result;
        ozo::request(provider, query, std::back_inserter(result), yield);
        if (!benchmark.step(result.size())) {
            break;
        }
    }
});
// ...
```

Parsed result vs raw result

name	request speed, req/sec	request time, us
raw result	9902	107 (100%)
parsed result	9170	119 (111%)

Complex query

```
// ...
const auto query = ("SELECT typname, typnamespace, typowner, typlen, "_SQL
    + "typbyval, typcategory, typispreferred, typisdefined, typdelim, "_SQL
    + "typrelid, typelem, typarray "_SQL
    + "FROM pg_type WHERE typtypmod = "_SQL + -1 + " AND typisdefined = "_SQL
    + true).build();
asio::spawn(io, [&] (auto yield) {
    while (true) {
        ozo::result result;
        ozo::request(provider, query, result, yield);
        if (!benchmark.step(result.size())) { break; }
    }
});
// ...
```

Complex query vs simple query

name	request speed, req/sec	request time, us	read rows speed, row/sec
simple query (1 row)	9902	107 (100%)	9902 (100%)
complex query (373 rows)	1091	969 (906%)	406843 (4109%)

Complex query: parsed result

```
struct pg_type {
    // ...
};
BOOST_FUSION_ADAPT_STRUCT(pg_type, /* ... */)
// ...
asio::spawn(io, [&] (auto yield) {
    while (true) {
        std::vector<pg_type> result;
        ozo::request(provider, query, std::back_inserter(result), yield);
        if (!benchmark.step(result.size())) {
            break;
        }
    }
});
// ...
```

Parsed result vs raw result (complex query)

name	request speed, req/sec	request time, ms	read rows speed, row/sec
raw result	1091	0.963 (100%)	406843 (432%)
parsed result	253	4.076 (423%)	94221 (100%)

Multiple connections

```
auto pool = ozo::make_connection_pool(connection_info, n + 1, 0);
// ...
for (std::size_t i = 0; i < n; ++i) {
    asio::spawn(io, [i, &] (auto yield) {
        while (true) {
            ozo:result result;
            ozo::request(provider, query, std::back_inserter(result), yield);
            if (!benchmark.step(result.size(), i)) {
                break;
            }
        }
    });
}
// ...
```

Raw result

connections	request speed, req/sec	request time, ms	read rows speed, row/sec
1	1091	0.963 (100%)	406843 (100%)
2	2141	1.035 (107%)	798528 (196%)
4	3507	1.225 (127%)	1308121 (322%)
8	4253	2.064 (214%)	1586623 (390%)
16	5155	3.453 (359%)	1922983 (473%)
32	5707	6.429 (668%)	2128794 (523%)
64	5496	12.1884 (1266%)	2050109 (504%)

Parsed result

connections	request speed, req/sec	request time, ms	read rows speed, row/sec
1	253	4.076 (100%)	94221 (100%)
2	313	6.604 (169%)	116819 (124%)
4	320	12.801 (314%)	119266 (127%)
8	299	27.369 (672%)	111551 (118%)

Questions?

Roman Siromakha


elsid@yandex-team.ru


https://github.com/elsid


https://github.com/YandexMail/ozo
(pull requests are welcome)