YDB C++ Russia

КоРутинная практика: пишем свой TPC-C на корутинах

Евгений Иванов,
Ведущий разработчик YDB, Яндекс

Евгений Иванов

Ведущий разработчик YDB,
Яндекс

Александр Крюков

Старший разработчик YDB,
Яндекс

YDB — отказоустойчивая распределенная платформа, реализованная на C++

1

Изначально OLTP

2

YDB Topic Service (аналог kafka)

Транзакции между топиками и таблицами

3

OLAP

Векторный поиск

4

И не только

Любишь СУБД писать — люби и бенчмарки запускать

1. Нагрузочное тестирование
2. Предотвращение performance-регрессий
3. Проверка оптимизаций и фич
4. Сравнение с другими СУБД


Нагружать распределённую СУБД непросто
кластер
→ 100+ нод
клиент
Мы хотели нагрузить кластер .... Но сначала нагрузили клиент

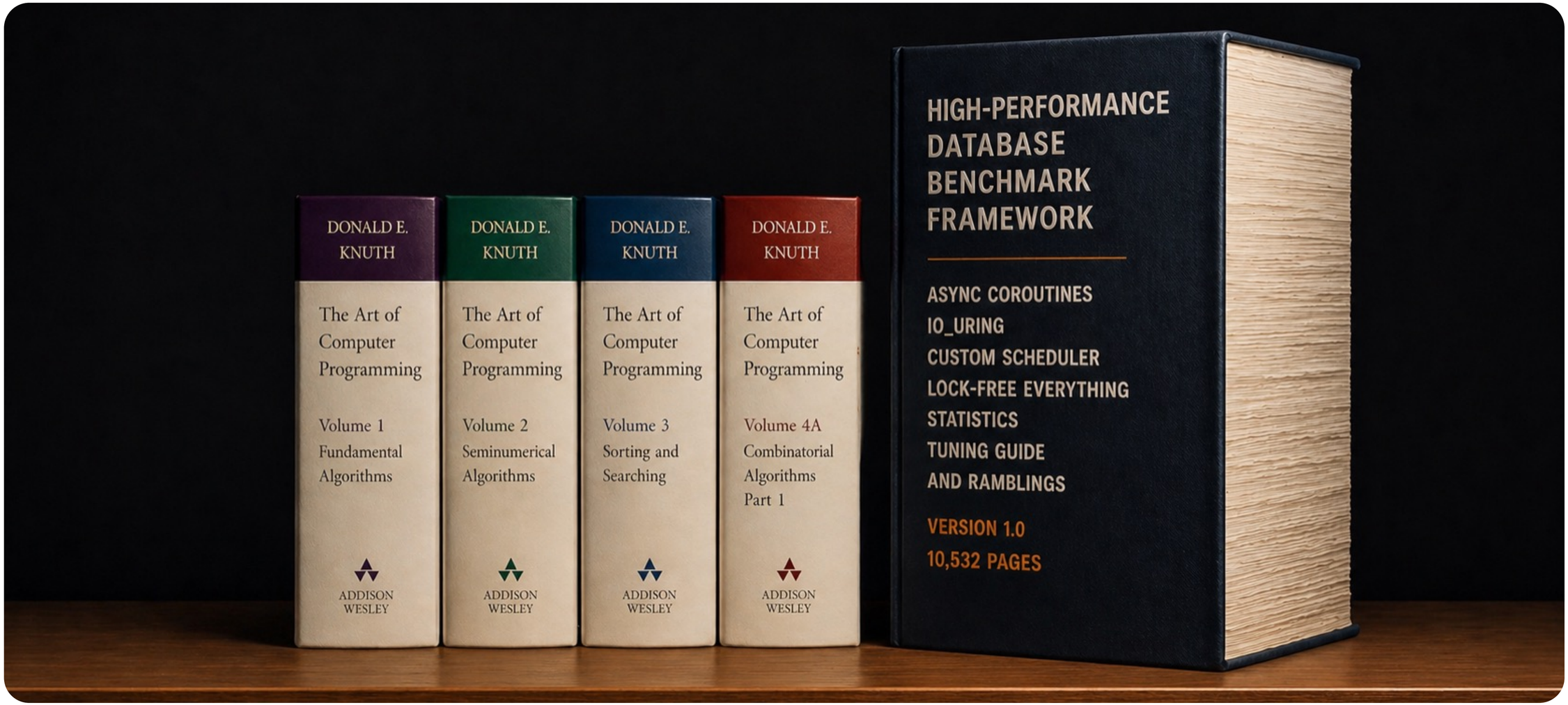
Доклад о том, как эффективно реализовать бенчмарк СУБД

1. TPC-C — самый популярный бенчмарк OLTP.
Когда СУБД распределенная, крайне важна эффективность его реализации

2. Мы попробовали популярную многопоточную реализацию на Java — нам не понравилось

3. Корутины — идеальный инструмент для таких задач.
В C++ он сложный, но стоит научиться его применять

Не только эффективно, но и читаемо

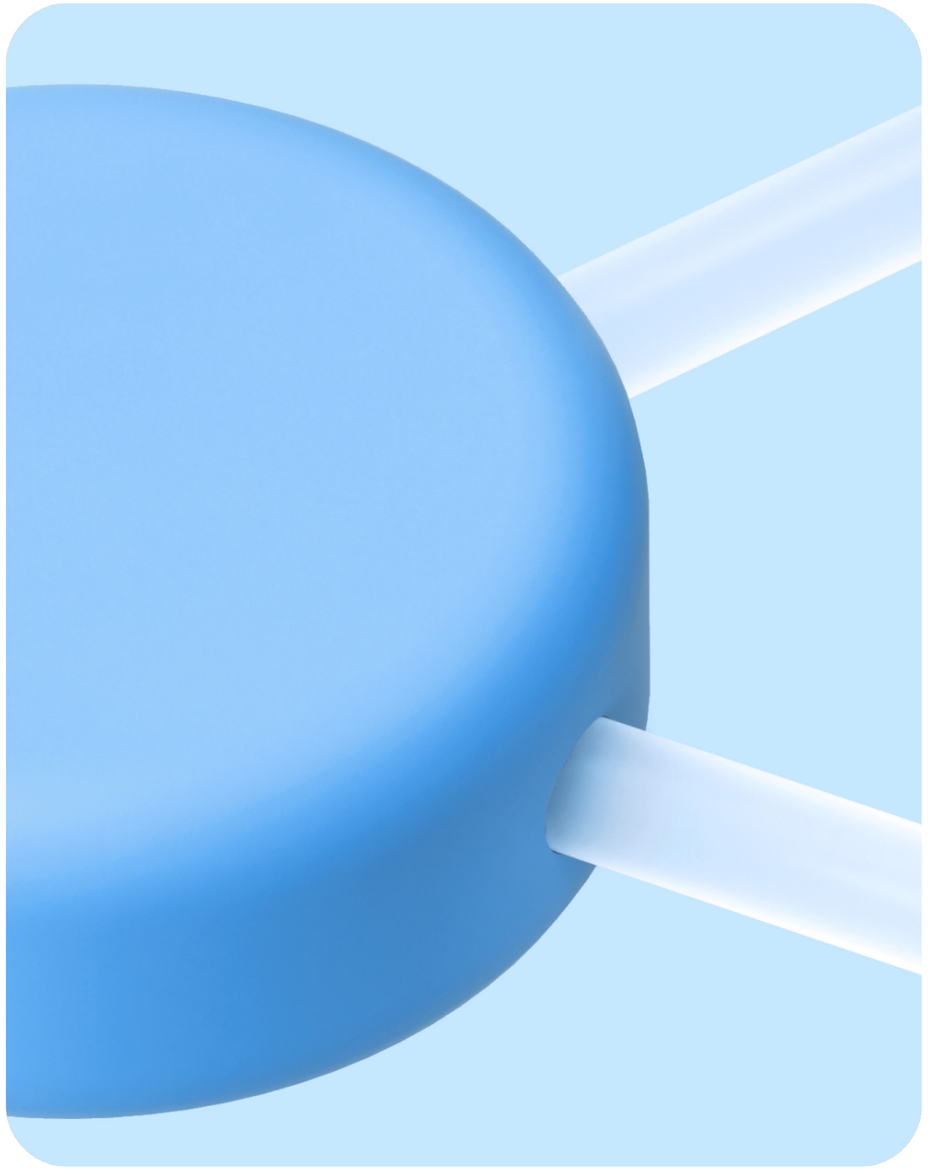


Код в презентации

1. На основе нашего TPC-C и сопутствующих проектов.
   Адаптирован для презентации

2. В конце презентации есть ссылка на реальный код,
   в т.ч. вне YDB, с которым легко поэкспериментировать

Знакомимся с ТРС-С



01

TPC-C

Стандарт появился в 1992 году

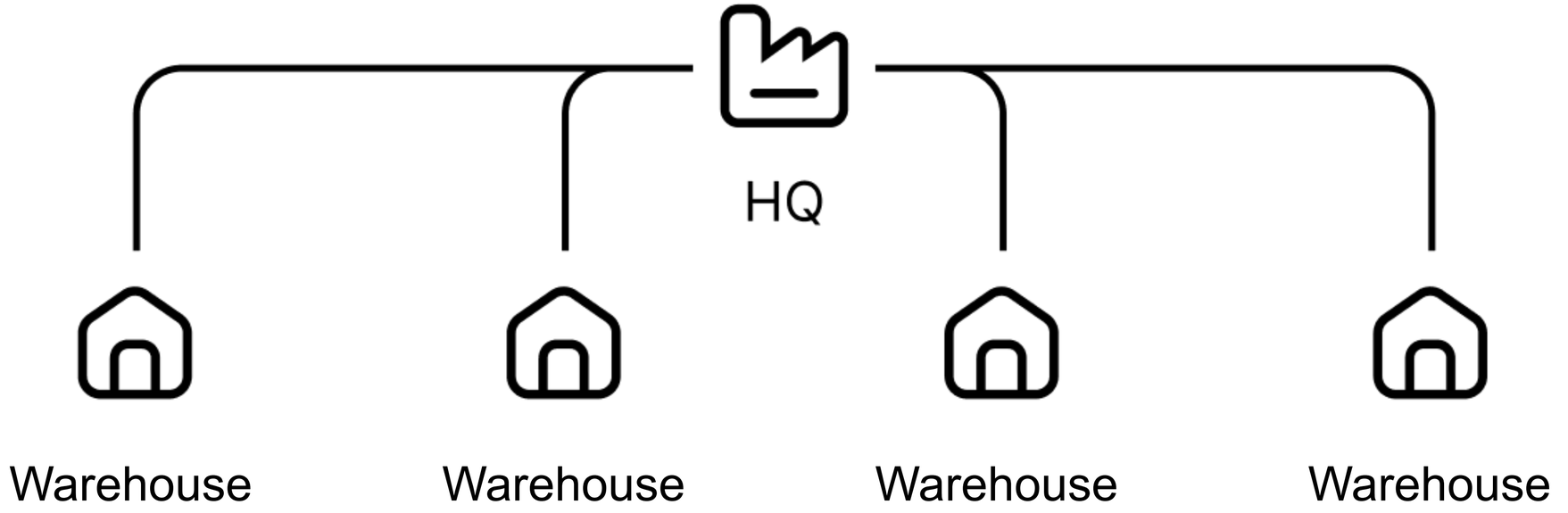
CockroachDB

«Единственная объективная методика оценки производительности OLTP»

YugabyteDB

«Наиболее реалистичное и объективное измерение производительности OLTP-систем»

TPC-C — эмуляция e-commerce организации



Логика TPC-C

- Число складов задаётся при запуске
- Каждый склад обслуживает 10 районов, примерно 100 МБ данных
- В каждом районе есть терминал
- Пользователи делают заказы и оплачивают их
- Иногда проверяется статус заказа
- Выполняется доставка
- На складах проводится инвентаризация

Терминалы и транзакции TPC-C

Терминалов
сотни тысяч

Каждый терминал либо спит, либо выполняет интерактивную транзакцию

Транзакции состоят из 5-10 запросов в СУБД и очень простой и быстрой логики между ними

Проект CMU Benchbase

Multi-DBMS SQL Benchmarking Framework на основе JDBC

Написан в Carnegie Mellon. Под руководством проф. Энди Павло

Легко добавлять новые СУБД и бенчмарки

Единственная широко известная реализация TPC-C

YugabyteDB использует форк Benchbase

Сначала мы пошли тем же путём

Требования к клиенту для запуска
15 000 складов

В оригинальном Benchbase TPC-C

150 000
потоков ОС

600
GB RAM

Запускали TPC-C
на 5 серверах
(каждый 128 ядер
и 512 GB RAM)

YDB работал
на 3 таких серверах!

Масштабируемся

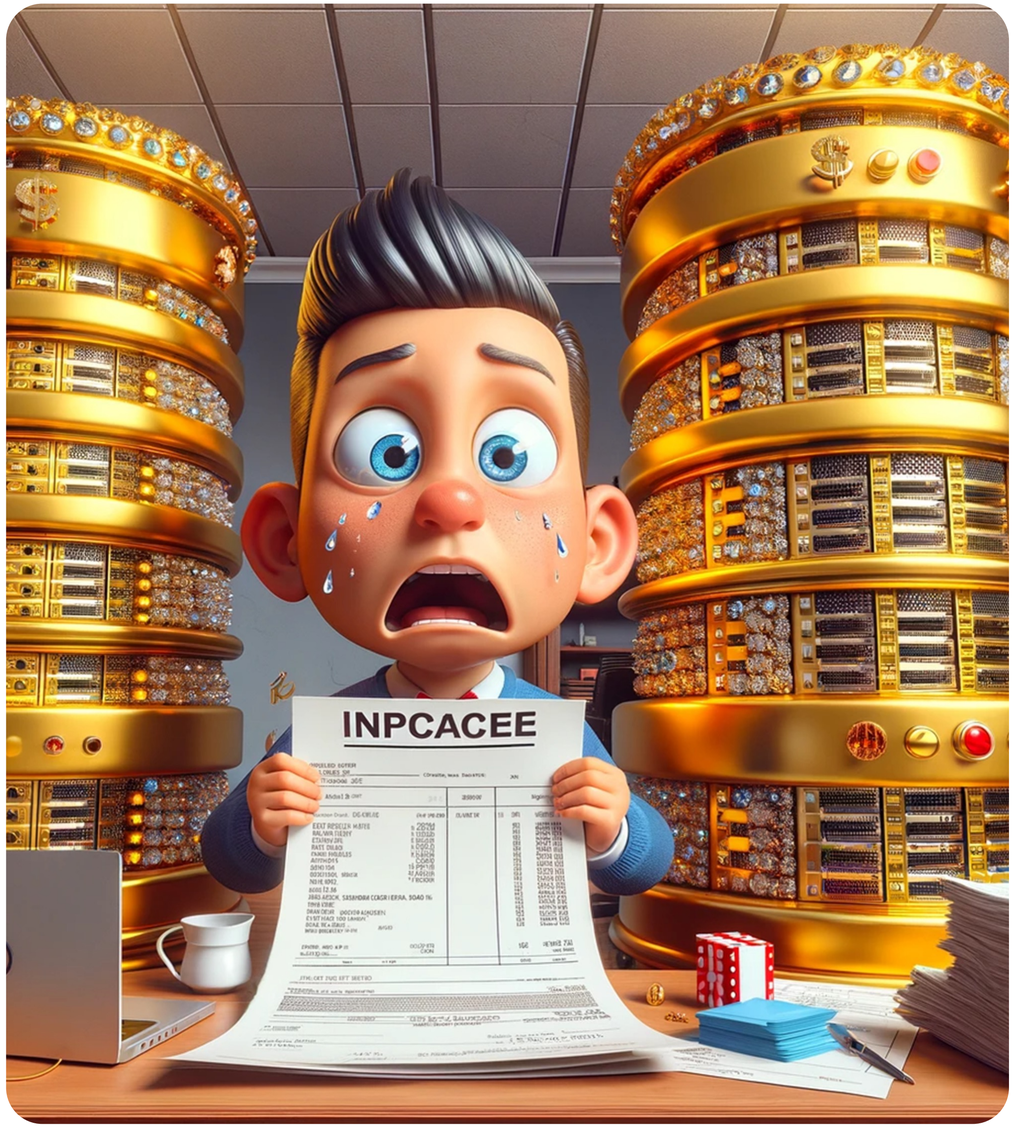
Хотим прогрузить СУБД,
в которой 9, 15, 30, 60, 81 серверов

- YDB
- CockroachDB
- YugabyteDB

$10,000

стоит один такой эксперимент в AWS

И одним экспериментом не обойтись



Форкнули benchbase

- github.com/ydb-platform/tpcc и github.com/ydb-platform/tpcc-postgres
- Заменили потоки ОС на виртуальные потоки Java
- Уменьшили потребление памяти

Стало

1 CPU core — **1000** складов

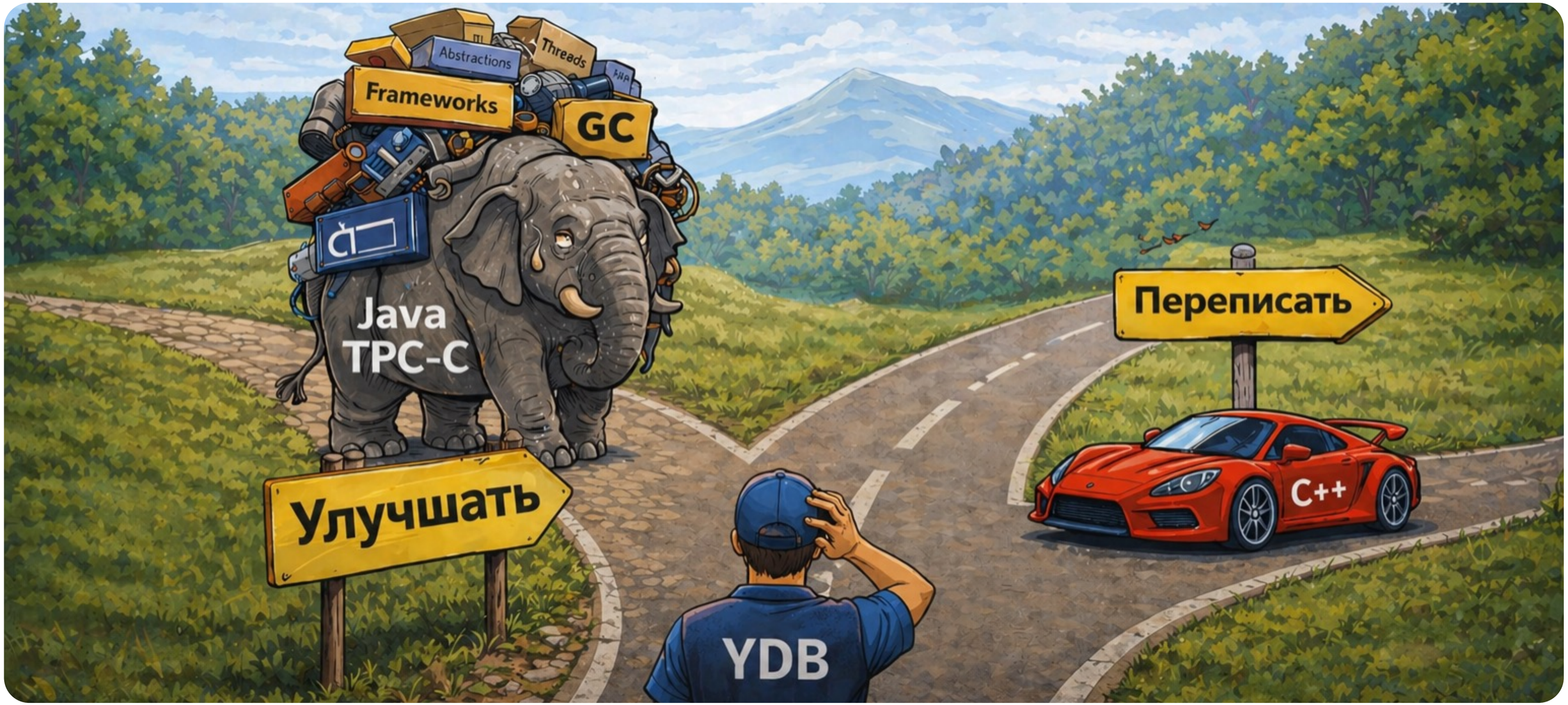
6 MiB RAM на каждый склад

15 000 складов

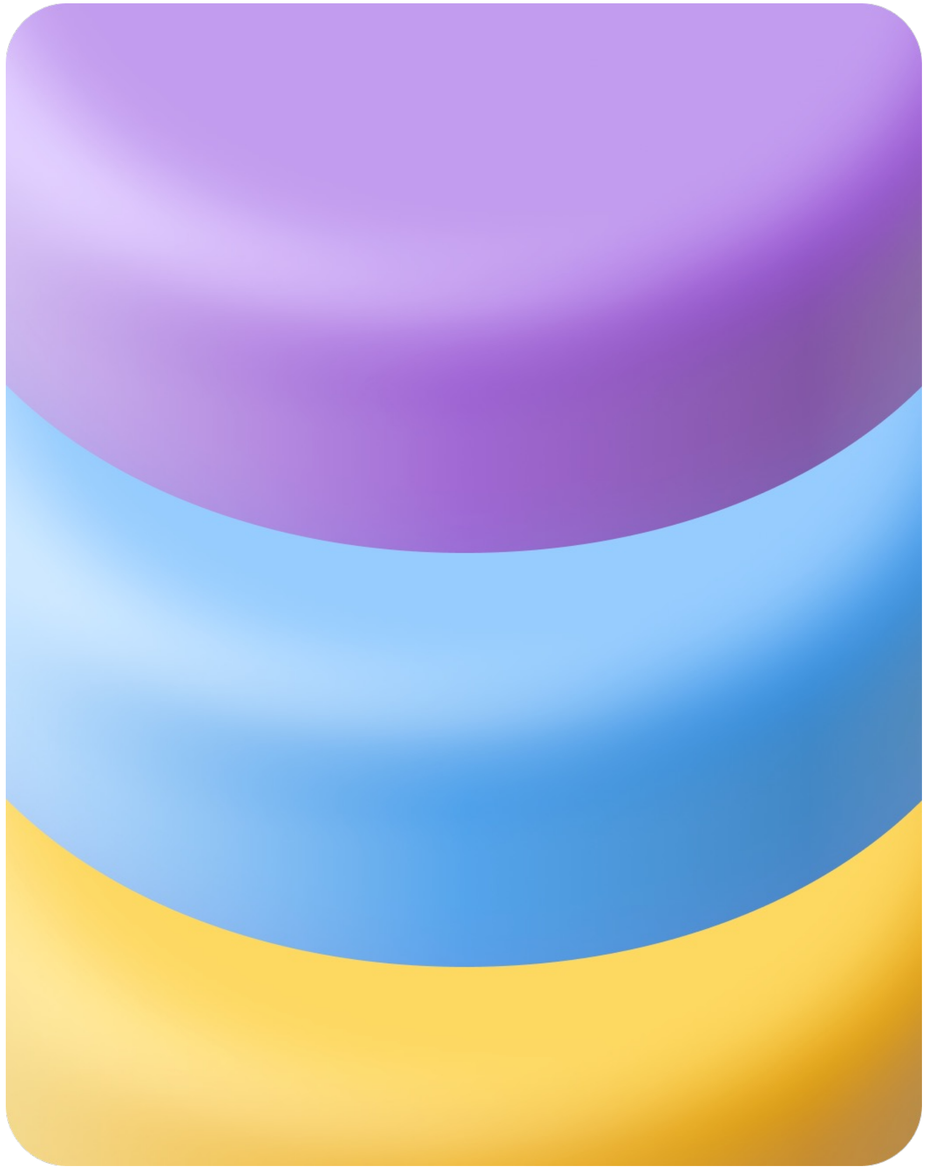
90 GiB RAM вместо **600** GiB

Разумное число потоков ОС

Улучшать нельзя переписать



Проектируем TPC-C на C++



02

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

```
// Terminal
while (Now() < EndTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);

  // simulate user typing / preparing request
  Sleep(KeyingTime(type));

  auto result = RunTx(input);

  // simulate user thinking after response
  Sleep(ThinkTime(type));

  Stats.Record(type, result);
}
```

```
// Simplified transaction
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2

  // r3-r9

  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();
}
```

Терминалы и транзакции TPC-C

Терминалы
много спят:
синхронный
системный вызов

Транзакции
большую часть
времени ждут
ответа от СУБД

В упрощённом
коде транзакции
используется
асинхронный API,
но мы синхронно
ждём завершения

Один терминал — один поток?

Достоинства

- Очень удобно писать/читать такой код
- ОС обеспечивает честное разделение времени между терминалами
- ОС отвечает за сохранение и восстановление контекста потока-терминала

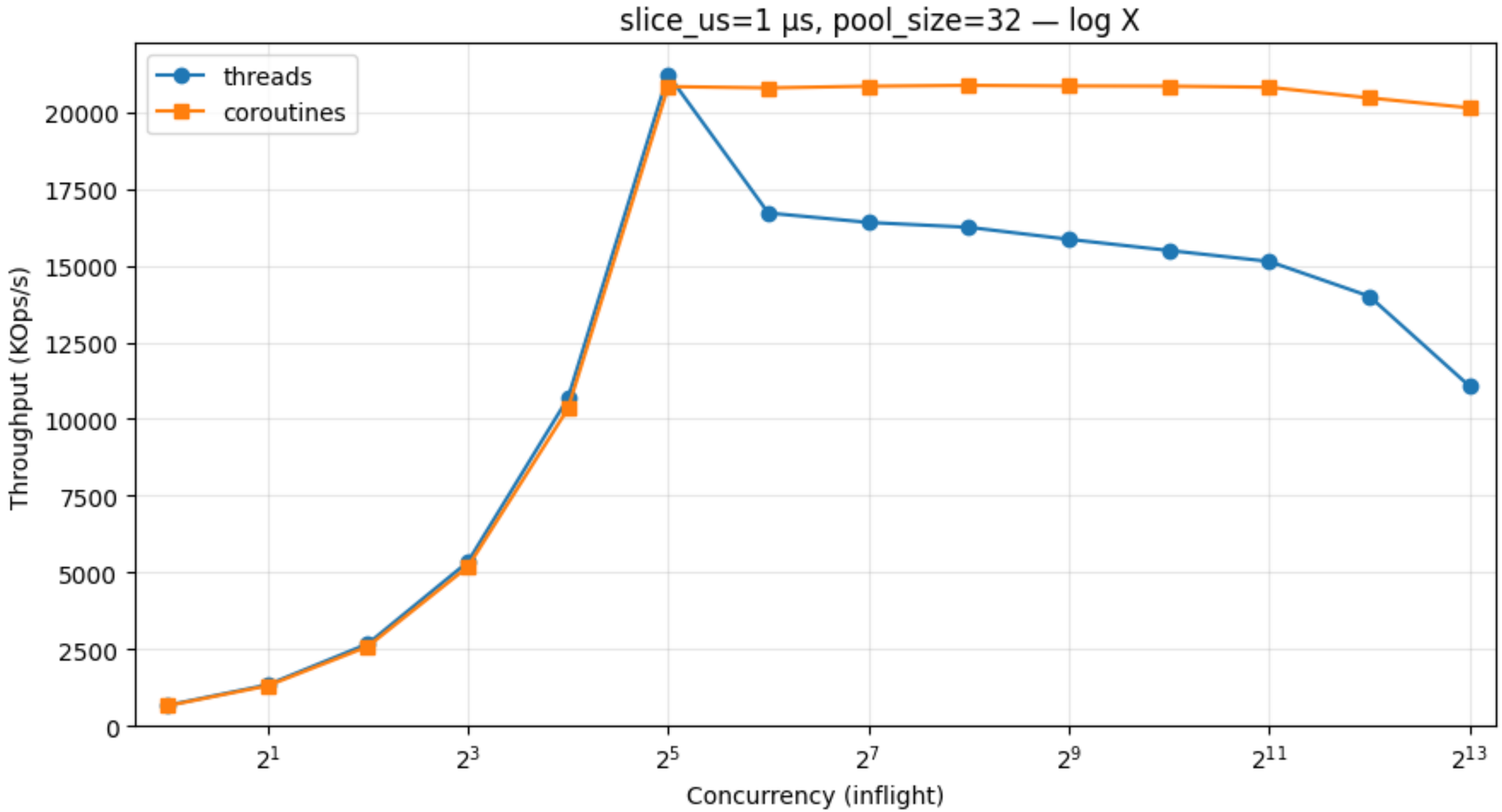
Недостатки

- Много потоков не создать: максимум несколько десятков тысяч, а в TPC-C сотни тысяч терминалов
- Потенциально высокое потребление памяти: стеки потоков и page-level fragmentation

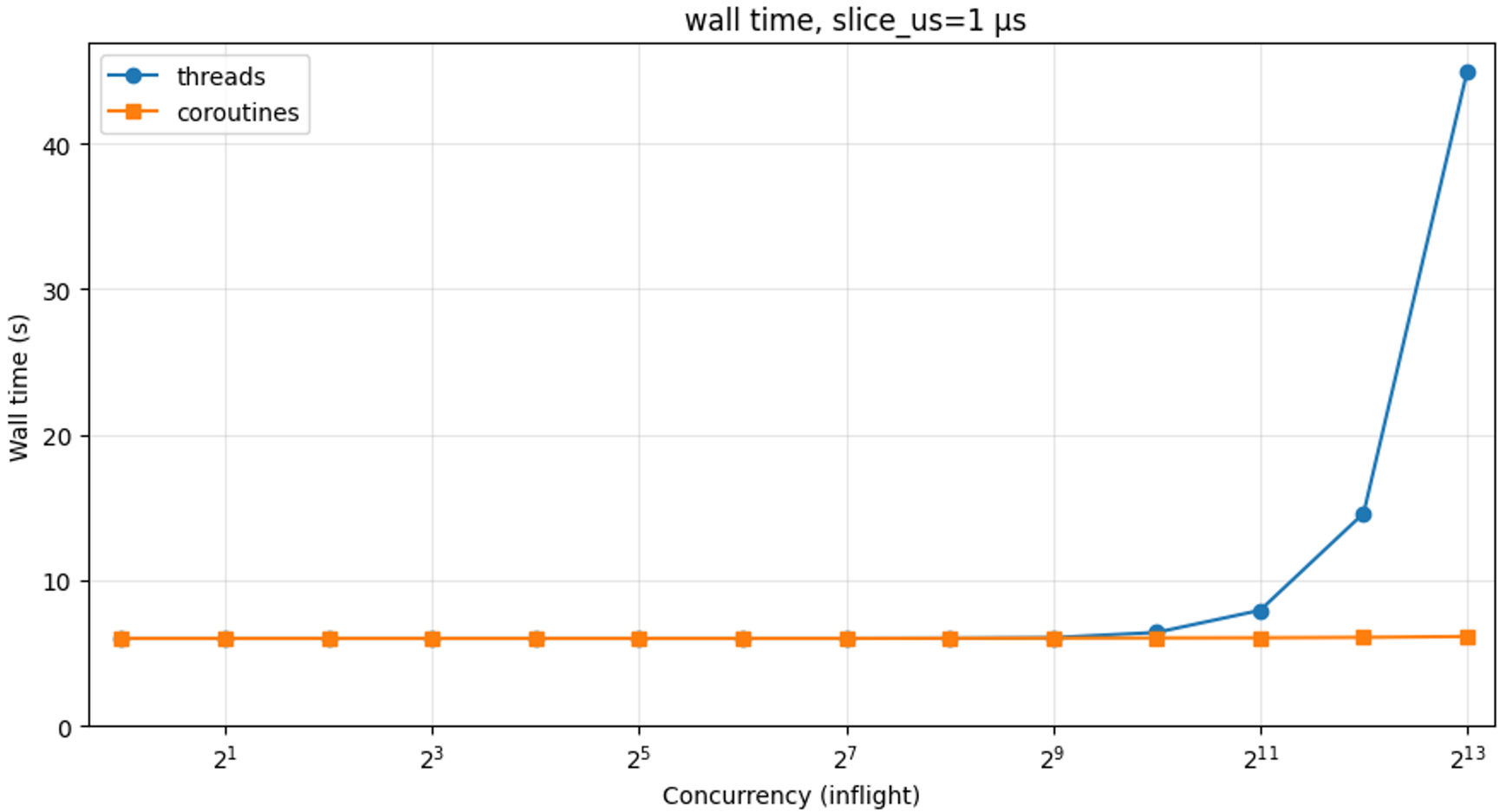
Микробенчмарк: корутины против потоков

- Создаём N workers: потоки или корутины
- 32 физических ядра (все на одной NUMA-ноде)
- 1 секунда warmup + 5 секунд выполнение
- Каждый делает короткое вычисление в течение 1 мкс, затем yield
- Измеряем суммарное число операций в секунду
- Измеряем общее время выполнения теста

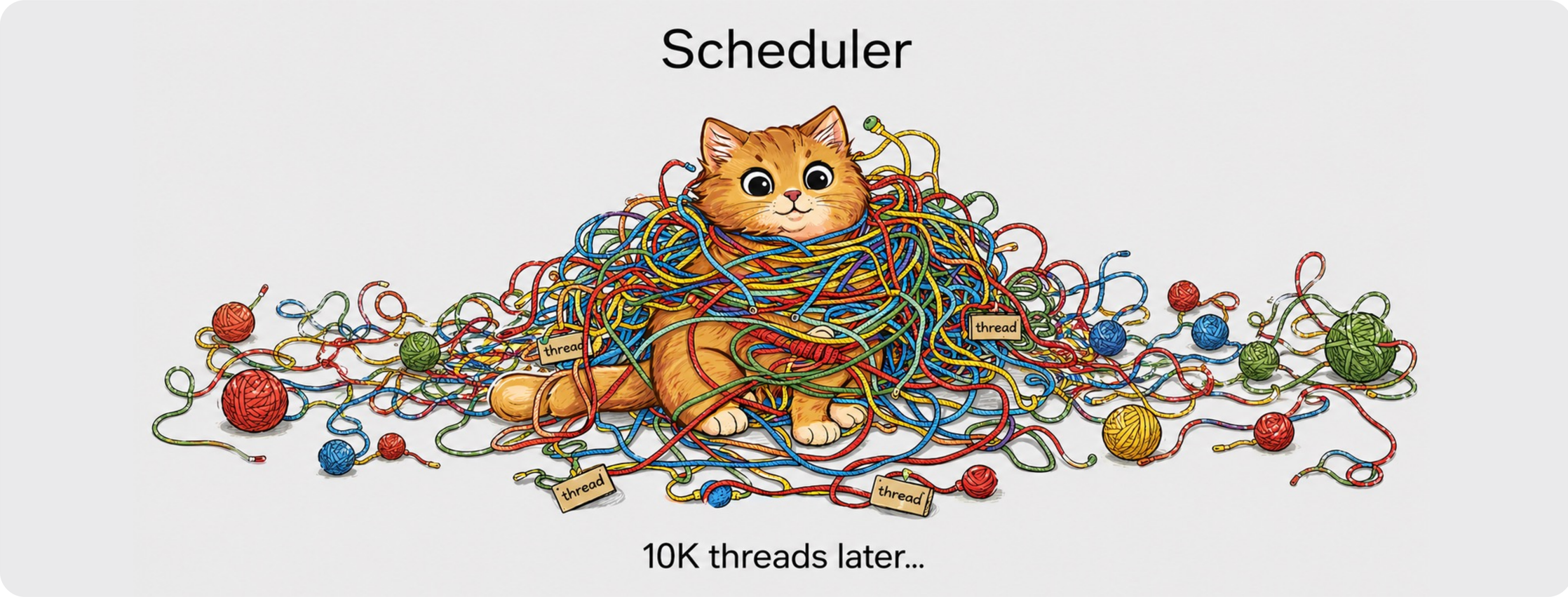
Микробенчмарк: корутины против потоков



Микробенчмарк: корутины против потоков



Потоки не масштабируются.
Они запутываются.



Async API: future

```
// thread1
auto future = tx.Query(sql);
// do something
if (future.HasValue()) {
    return;
}
// do something // <----------------------------->
if (future.HasValue()) {
  return;
}
```

```
// thread2
//
//
// DBMS processes the query on another host
//
// DBMS replied
promise.SetValue(value);
```

Async API: future

Отсутствует в стандартной библиотеке

Без этого неудобно писать код

Но есть практически во всех сторонних библиотеках

Цепочка (композиция)

```
auto future = tx.Query(sql1)
  .Apply([&](auto r1) {
    // process r1
    return tx.Query(sql2);
  });
```

Callback

```
tx.Query(sql1)
  .Subscribe([&](auto r1) {
    std::cout << “query 1 is ready”;
  });
```

Пулы и future

`After()`

наш таймер

`ContinueWith()`

цепочки из `future()`,
выполняющиеся
в одном пуле

```
auto future1 = pool.After(sleepInterval);

auto future2 = pool.ContinueWith(future1, [&](auto) {
    return RunTx(input, pool);
  });
```

Async API: транзакция на future

1. `Query()` возвращает `future` на результат

2. `Query()` внутри себя хранит `promise`, через который проставляет результат, когда он готов

3. Используем цепочки из `future (Apply)`: результат `sql1` подаётся на вход lambda, которая на его основе запускает `sql2`

```
auto tx = db.BeginTransaction();

auto f1 = tx.Query(sql1);
auto f2 = f1.Apply([](auto r1) {
  // process r1
  return tx.Query(sql2);
});

return f2;
```

```
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).GetValueSync();
  // process r1

  auto r2 = tx.Query(sql2).GetValueSync();
  // process r2


  // r3-r9


  auto r10 = tx.Query(sql10).GetValueSync();
  // process r10

  return tx.Commit().GetValueSync();

}
```

```
Future<TxResult> RunTx(Input input) {
  auto tx = db.BeginTransaction();
  return tx.Query(sql1)
    .Apply([&](auto r1) {
      // process r1
      return tx.Query(sql2);
    })
    // r3-r9
    .Apply([&](auto r10) {
      // process r10
      return tx.Commit();
    });
}
```

Async API: кто что выполняет?

1. RunTx() выполняется в потоке пользователя

```
Future<TxResult> RunTx(Input input) {
  auto tx = db.BeginTransaction();
  return tx.Query(sql1)
    .Apply([&](auto r1) {
      // process r1
      return tx.Query(sql2);
    })
    // r3-r9
    .Apply([&](auto r10) {
      // process r10
      return tx.Commit();
    });
}
```

Async API: кто что выполняет?

1. `RunTx()` выполняется в потоке пользователя

2. Всё, что после первого `Apply()` выполнит тот, кто проставляет значение `promise`.
В нашем случае это поток SDK базы

```
Future<TxResult> RunTx(Input input) {
  auto tx = db.BeginTransaction();
  return tx.Query(sql1)
    .Apply([&](auto r1) {
      // process r1
      return tx.Query(sql2);
    })
    // r3-r9
    .Apply([&](auto r10) {
      // process r10
      return tx.Commit();
    });
}
```

Async API: кто что выполняет?

1. `RunTx()` выполняется в потоке пользователя

2. Всё, что после первого `Apply()` выполнит тот, кто проставляет значение `promise`.
   В нашем случае это поток SDK базы

3. Большая часть пользовательского кода переезжает на выполнение в другие потоки, которые мы не контролируем

А что если мы возьмём и заблокируем все потоки SDK?

```
Future<TxResult> RunTx(Input input) {
  auto tx = db.BeginTransaction();
  return tx.Query(sql1)
    .Apply([&](auto r1) {
      // process r1
      return tx.Query(sql2);
    })
    // r3-r9
    .Apply([&](auto r10) {
      // process r10
      return tx.Commit();
    });
}
```

Async API:
а что с терминалом?

1. Теперь `RunTx()` возвращает `future`: блокироваться на ней мы не можем

2. Ещё надо что-то делать со сном

```
// Terminal (single user)
while (Now() < EndTs) {
    auto type = PickTransaction();

    // simulate user typing / preparing request
    Sleep(KeyingTime(type));

    auto future = RunTx(input); // DB interaction

    // simulate user thinking after response
    Sleep(ThinkTime(type));

    Stats.Record(type, result);
}
```

Async API: неблокирующий терминал

```
struct TTerminalState {
  Promise<void> Done;
  TInstant EndTs;

  TransactionType Type;
  Input Input;
};
```

Async API: неблокирующий терминал

```
struct TTerminalState {
  Promise<void> Done;
  TInstant EndTs;

  TransactionType Type;
  Input Input;
};

Future<void> RunTerminal(TInstant endTs, IPool& pool) {
  auto state = std::make_unique<TTerminalState>();
  state->EndTs = endTs;

  auto future = state->Done.GetFuture();

  pool.Schedule([state = std::move(state), &pool]() mutable {
    TerminalStep(std::move(state), pool);
  });

  return future;
}
```

Async API: неблокирующий терминал

```
void TerminalStep(std::unique_ptr<TTerminalState> state, IPool& pool) {
  if (Now() >= state->EndTs) {
    state->Done.SetValue();
    return;
  }
  state->Type = PickTransaction();
  state->Input = MakeInput(state->Type);
```

Async API: неблокирующий терминал

```
void TerminalStep(std::unique_ptr<TTerminalState> state, IPool& pool) {
  if (Now() >= state->EndTs) {
    state->Done.SetValue();
    return;
  }
  state->Type = PickTransaction();
  state->Input = MakeInput(state->Type);

  auto f1 = pool.After(KeyingTime(state->Type));
```

Async API: неблокирующий терминал

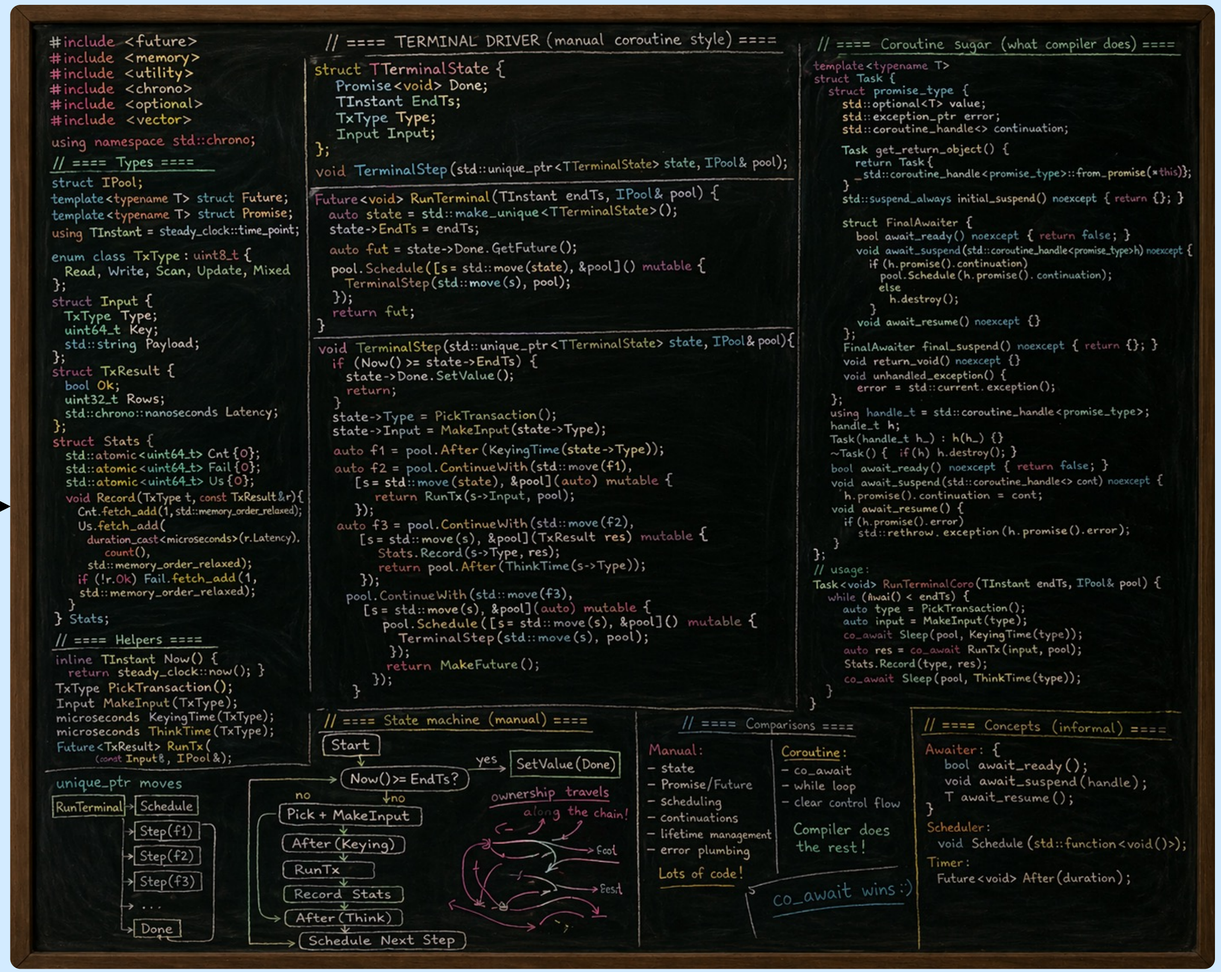
```
void TerminalStep(std::unique_ptr<TTerminalState> state, IPool& pool) {
  if (Now() >= state->EndTs) {
    state->Done.SetValue();
    return;
  }
  state->Type = PickTransaction();
  state->Input = MakeInput(state->Type);

  auto f1 = pool.After(KeyingTime(state->Type));
  pool.ContinueWith(f1, [state = std::move(state), &pool](auto) mutable {
    auto f2 = RunTx(state->Input, pool);

    pool.ContinueWith(f2, [state = std::move(state), &pool](auto result) mutable {
      Stats.Record(state->Type, result);
```

Async API: неблокирующий терминал

```
void TerminalStep(std::unique_ptr<TTerminalState> state, IPool& pool) {
  if (Now() >= state->EndTs) {
    state->Done.SetValue();
    return;
  }
  state->Type = PickTransaction();
  state->Input = MakeInput(state->Type);

  auto f1 = pool.After(KeyingTime(state->Type));
  pool.ContinueWith(f1, [state = std::move(state), &pool](auto) mutable {
    auto f2 = RunTx(state->Input, pool);

    pool.ContinueWith(f2, [state = std::move(state), &pool](auto result) mutable {
      Stats.Record(state->Type, result);

      auto f3 = pool.After(ThinkTime(state->Type));
      pool.ContinueWith(f3, [state = std::move(state), &pool](auto) mutable {
        pool.Schedule([state = std::move(state), &pool]() mutable {
          TerminalStep(std::move(state), pool);
        });
      });
    });
  });
}
```

Было — стало: callback hell

```
void RunTerminal(TInstant endTs) {

  while (Now() < EndTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    Sleep(KeyingTime(type));

    auto result = RunTx(input);

    Sleep(ThinkTime(type));

    Stats.Record(type, result);
  }
}
```



Async API: транзакция с пулом

```
Future<TxResult> RunTx(Input input, IPool& pool) {
  auto tx = db.BeginTransaction();

  auto f1 = tx.Query(sql1);
  auto f2 = pool.ContinueWith(f1, [&](auto r1) {
    // process r1
    return tx.Query(sql2);
  });

  // ...

  auto f10 = pool.ContinueWith(f9, [&](auto r9) {
    // process r9
    return tx.Query(sql10);
  });

  return pool.ContinueWith(f10, [&](auto r10) {
    // process r10
    return tx.Commit();
  });
}
```

Было — стало: транзакция

```
TxResult RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = tx.Query(sql1).WaitGetResult();
  // process r1

  auto r2 = tx.Query(sql2).WaitGetResult();
  // process r2


  // r3-r9


  auto r10 = tx.Query(sql10).WaitGetResult();
  // process r10

  return tx.Commit().WaitGetResult();

}
```



```
Future<TxResult> RunTx(Input input, IPool& pool) {
  auto tx = db.BeginTransaction();

  auto f1 = tx.Query(sql1);
  auto f2 = pool.ContinueWith(f1, [&](auto r1) {
    // process r1
    return tx.Query(sql2);
  });

  // r3-r9

  auto f10 = pool.ContinueWith(f9, [&](auto r9) {
    // process r9
    return tx.Query(sql10);
  });

  return pool.ContinueWith(f10, [&](auto r10) {
    // process r10
    return tx.Commit();
  });
}
```

Получили эффективный код, но

- Бизнес-логика обёрнута в пулы и цепочки фьюч — намного хуже читается
- Большие лямбды сложно читать. А если выносить цепочки в функции, то код расползается по файлу
- Хотелось бы сохранить эффективность и вернуть читаемость

Транзакция на корутинах

- Код почти не отличается от первоначального синхронного
- Использует асинхронный API: убраны `GetValueSync()`
- Так же эффективен, как код с фьючами
- можно создавать сотни тысяч корутин

```
TFuture<TxResult> RunTx(Input input) {
  auto tx = db.BeginTransaction();

  auto r1 = co_await tx.Query(sql1);
  // process r1

  auto r2 = co_await tx.Query(sql2);
  // process r2

  // r3-r9

  auto r10 = co_await tx.Query(sql10);
  // process r10

  co_return tx.Commit();
}
```

Терминал на корутинах

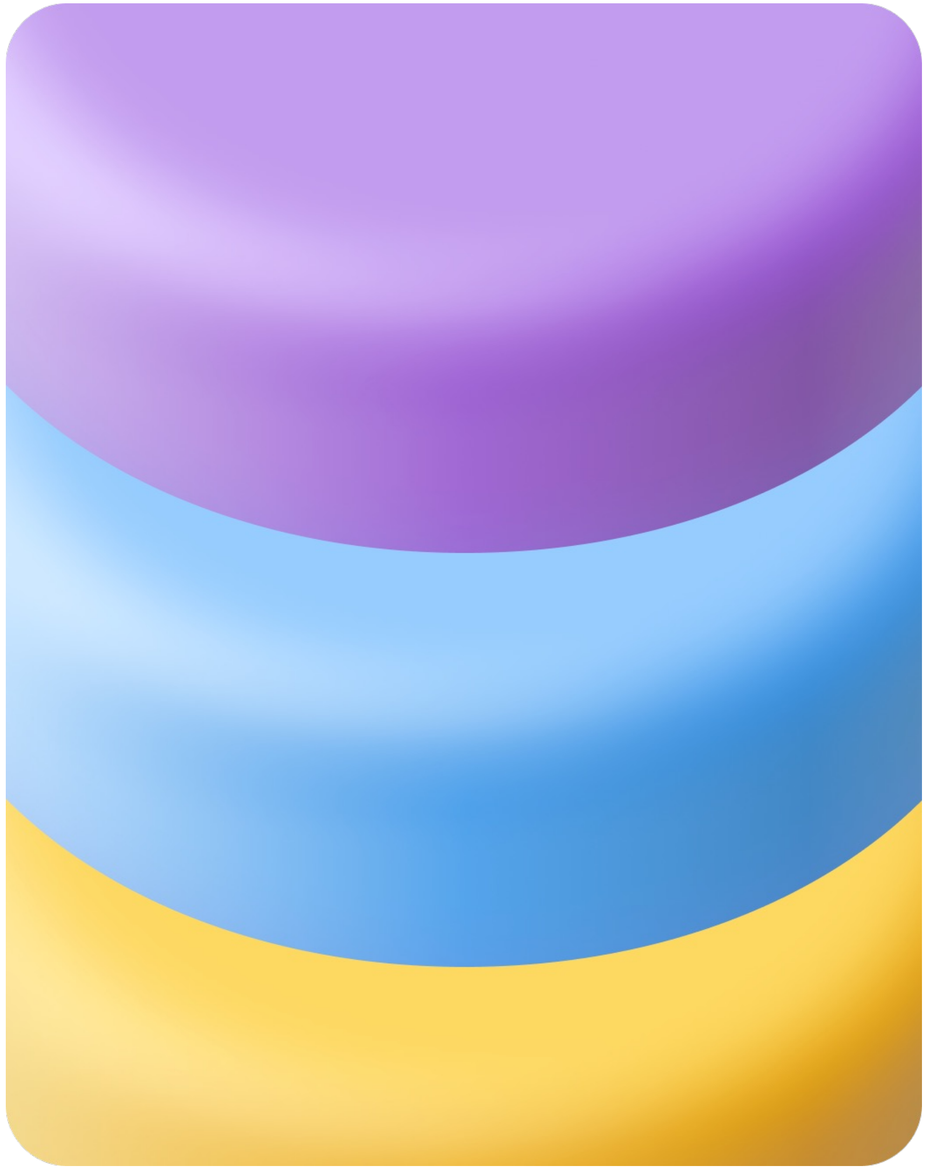
1. Асинхронный код на фьючах эффективен, но его сложнее писать и сопровождать

2. Синхронный код был прост для понимания. Но неэффективен и много потоков не создать

3. Корутины выглядят слишком привлекательно. Что же там под капотом?

```
TFuture<void> RunTerminal(TInstant endTs) {
  while(Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}
```

Грокаем корутины



03

Прозрачная синхронная версия

- Внутри `GetValueSync()` и `Sleep()` системные вызовы
- Операционная система сохраняет контекст потока: регистры, стек
- ОС сама решает, какой поток запустить следующим — запускается другой терминал
- Когда закончится сон или просигналит `Wait()` и появится свободный CPU, ОС восстановит состояние потока и терминал продолжит свою работу

```
tx.Query(sql1).GetValueSync();

Sleep(KeyingTime(type));
```

Корутины — синтаксический сахар

- Очень похоже на синхронный код: «блокируемся» в `co_await`, состояние сохраняется и восстанавливается по готовности
- ОС ничего не знает о корутинах
- Компилятор генерирует дополнительный код, никакой другой магии нет
- Компилятор не делает ничего, что нельзя было бы написать в старых версиях C++. Что мы и попробуем сделать далее

```
co_await Sleep(KeyingTime(type));
auto result = co_await RunTx(input);
```

Терминал без сахара: выделяем state

```
void RunTerminal(TInstant endTs) {

  while (Now() < EndTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    Sleep(KeyingTime(type));

    auto result = RunTx(input);

    Sleep(ThinkTime(type));

    Stats.Record(type, result);
  }
}
```

```
enum class EState {
  Sleep,
  RunTx,
  ProcessResult,
  Terminate,
};
```

Терминал без сахара: выделяем state

```
void RunTerminal(TInstant endTs) {

  while (Now() < EndTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    Sleep(KeyingTime(type));

    auto result = RunTx(input);

    Sleep(ThinkTime(type));

    Stats.Record(type, result);
  }
}
```

```
enum class EState {
  Sleep,
  RunTx,
  ProcessResult,
  Terminate,
};

struct TState {
  EState State;
  TFuture<void> StateDone;

  TInstant endTs;
  TransactionType Type;
  Input Input;
  std::optional<TxResult> Result;
};

void RunTerminal(TState* state);
```

Терминал — стейт-машина

1

```
void RunTerminal(TStatefulTask* state) {
  if (Now() >= state->EndTs) {
    state->State = EState::Terminate;
    return;
  }

  auto& pool = *state->Pool;

  switch (state->State) {
  case EState::Sleep: {
    return;
  }
  case EState::RunTx: {
    return;
  }
  case EState::ProcessResult: {
    return;
  }
}
```

```
case EState::Sleep: {
  state->Type = PickTransaction();
  state->Input = MakeInput(state->Type);
  state->Result.reset();

  auto& sleepTime = KeyingTime(state->Type);
  state->StateDone = pool.After(sleepTime);
  state->State = EState::RunTx;
  return;
}
```

Терминал — стейт-машина

2

```
void RunTerminal(TStatefulTask* state) {
  if (Now() >= state->EndTs) {
    state->State = EState::Terminate;
    return;
  }

  auto& pool = *state->Pool;

  switch (state->State) {
  case EState::Sleep: {
    return;
  }
  case EState::RunTx: {
    return;
  }
  case EState::ProcessResult: {
    return;
  }
}
```

```
case EState::RunTx: {
  state->StateDone = RunTx(state->Input, pool)
    .Apply([state](auto result) {
      state->Result = std::move(result);
    });


  state->State = EState::ProcessResult;
  return;
}
```

Терминал — стейт-машина

3

```
void RunTerminal(TStatefulTask* state) {
  if (Now() >= state->EndTs) {
    state->State = EState::Terminate;
    return;
  }

  auto& pool = *state->Pool;

  switch (state->State) {
  case EState::Sleep: {
    return;
  }
  case EState::RunTx: {
    return;
  }
  case EState::ProcessResult: {
    return;
  }
}
```

```
case EState::ProcessResult: {
  Stats.Record(state->Type, *state->Result);

  state->StateDone = pool.After(ThinkTime(state->Type));
  state->State = EState::Sleep;
  return;
}
```

От стейта к таске

```
struct TStatefulTask {
  EState State = EState::Sleep;
  // ...
  void Resume() {
    RunTerminal(this);
  }

  bool Done() const {
    return State == EState::Terminate;
  }
};
```

```
TStatefulTask CreateTerminal(TInstant endTs, IPool& pool) {
  return TStatefulTask{
    .State = EState::Sleep,
    .Pool = &pool,
    .EndTs = endTs,
  };
}

auto state = std::make_shared<TStatefulTask>(CreateTerminal(endTs, pool));

std::function<void()> driver;
driver = [state] {
  if (state->Done()) {
    return;
  }

  state->Resume();
  state->Pool->ContinueWith(state->StateDone, driver);
};

driver();
```

От стейта к таске

```
struct TStatefulTask {
  EState State = EState::Sleep;
  // ...
  void Resume() {
    RunTerminal(this);
  }

  bool Done() const {
    return State == EState::Terminate;
  }
};
```

```
TStatefulTask CreateTerminal(TInstant endTs, IPool& pool) {
  return TStatefulTask{
    .State = EState::Sleep,
    .Pool = &pool,
    .EndTs = endTs,
  };
}

auto state = std::make_shared<TStatefulTask>(CreateTerminal(endTs, pool));

std::function<void()> driver;
driver = [state] {
  if (state->Done()) {
    return;
  }

  state->Resume();
  state->Pool->ContinueWith(state->StateDone, driver);
};

driver();
```

От стейта к таске

```
struct TStatefulTask {
  EState State = EState::Sleep;
  // ...
  void Resume() {
    RunTerminal(this);
  }

  bool Done() const {
    return State == EState::Terminate;
  }
};
```

```
TStatefulTask CreateTerminal(TInstant endTs, IPool& pool) {
  return TStatefulTask{
    .State = EState::Sleep,
    .Pool = &pool,
    .EndTs = endTs,
  };
}

auto state = std::make_shared<TStatefulTask>(CreateTerminal(endTs, pool));

std::function<void()> driver;
driver = [state] {
  if (state->Done()) {
    return;
  }

  state->Resume();
  state->Pool->ContinueWith(state->StateDone, driver);
};

driver();
```

Ключевые моменты

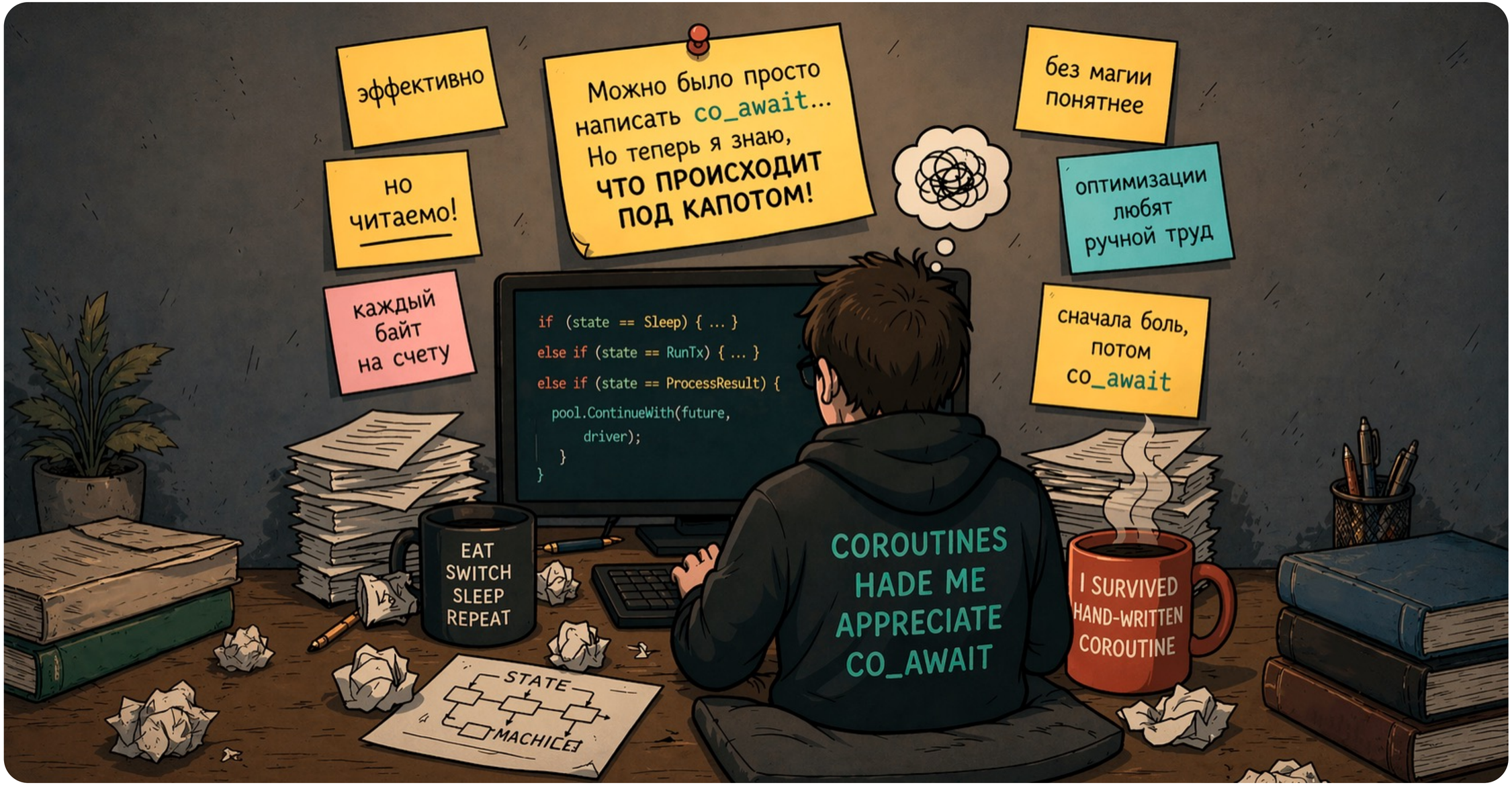
1 Преобразование в state-машину:

- `struct TStatefulTask;`
- `void RunTerminal(TStatefulTask* state);`

2 Создание таски

3 Нужна сущность, которая вызывает у таски Resume(), когда таска готова к выполнению. В примере это драйвер, на практике обычно пишут Scheduler

```
void RunTerminal(TInstant endTs) {

  while (Now() < EndTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    Sleep(KeyingTime(type));

    auto result = RunTx(input);

    Sleep(ThinkTime(type));

    Stats.Record(type, result);
  }
}
```


эффективно
но читаемо!
каждый байт на счету
Можно было просто написать co_await...
Но теперь я знаю, ЧТО ПРОИСХОДИТ ПОД КАПОТОМ!
без магии понятнее
оптимизации любят ручной труд
сначала боль, потом co_await
if (state == Sleep) { ... }
else if (state == RunTx) { ... }
else if (state == ProcessResult) {
pool.ContinueWith(future,
driver);
}
}
EAT
SWITCH
SLEEP
REPEAT
COROUTINES
HADE ME
APPRECIATE
CO_AWAIT
I SURVIVED
HAND-WRITTEN
COROUTINE
STATE
MACHICE

Корутины в C++20

Если в функции встречается одно из этих слов

`co_await` `co_return` `co_yield`

то это не функция, а корутина

```
TFuture<void> RunTerminal(TInstant endTs) {
  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}
```

Функция не функция

1

Корутина выглядит
как функция, которую
можно вызвать

2

На самом деле
при её вызове вызывается
«конструктор» корутины

3

Сама функция
преобразована
в стейт-машину

4

Вызов стейт-машины спрятан
в виде `resume()` внутри `task`.
Корутина может быть запущена
сразу или потом, подробнее позже

```
auto task = RunTerminal(endTs);

// compiler transforms into
auto task = CreateTerminalTask(endTs);
task.handle->resume();
```

Функция не функция

Недостающие запчасти?

Корутина должна вовращать тип T,
у которого есть `T::promise_type`

```
TFuture<void> Sleep(TDuration d);
TFuture<TxResult> RunTx(Input input);

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}

// usage
auto task = RunTerminal(endTs);
```

Недостающие запчасти?

- Корутина должна вовращать тип T, у которого есть `T::promise_type`
- Справа от `co_await` должен быть `awaitable` тип

```
TFuture<void> Sleep(TDuration d);
TFuture<TxResult> RunTx(Input input);

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}

// usage
auto task = RunTerminal(endTs);
```

Недостающие запчасти?

- Корутина должна вовращать тип T, у которого есть `T::promise_type`
- Справа от `co_await` должен быть `awaitable` тип
- В реализации `TFuture` в YDB и `folly::Future` всё это уже есть

```
TFuture<void> Sleep(TDuration d);
TFuture<TxResult> RunTx(Input input);

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}

// usage
auto task = RunTerminal(endTs);
```

Недостающие запчасти?

- Корутина должна вовращать тип T, у которого есть `T::promise_type`
- Справа от `co_await` должен быть `awaitable` тип
- В реализации `TFuture` в YDB и `folly::Future` всё это уже есть
- Тем не менее полезно понимать, что внутри

```
TFuture<void> Sleep(TDuration d);
TFuture<TxResult> RunTx(Input input);

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    co_await Sleep(KeyingTime(type));

    auto result = co_await RunTx(input);
    Stats.Record(type, result);

    co_await Sleep(ThinkTime(type));
  }
}

// usage
auto task = RunTerminal(endTs);
```

TFuture<void>::promise_type

```
// auto task = RunTerminal(endTs);
// transforms into

auto* promise = new promise_type;
auto task = promise->get_return_object();

auto h = MakeCoroutineHandle(promise);

if (promise->initial_suspend() == suspend_never) {
  h.resume();
}




task.GetValueSync();
```

```
template <typename... Args>
struct coroutine_traits<TFuture<void>, Args...> {
  struct promise_type {
    TPromise<void> promise;

    TFuture<void> get_return_object() {
      return promise.GetFuture();
    }

    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }

    void return_void() {
      promise.SetValue();
    }

    void unhandled_exception() {
      promise.SetException(std::current_exception());
    }
  };
};
```

TFuture<void>::promise_type

```
// auto task = RunTerminal(endTs);
// transforms into

auto* promise = new promise_type;
auto task = promise->get_return_object();

auto h = MakeCoroutineHandle(promise);

if (promise->initial_suspend() == suspend_never) {
  h.resume();
}




task.GetValueSync();
```

```
template <typename... Args>
struct coroutine_traits<TFuture<void>, Args...> {
  struct promise_type {
    TPromise<void> promise;

    TFuture<void> get_return_object() {
      return promise.GetFuture();
    }

    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }

    void return_void() {
      promise.SetValue();
    }

    void unhandled_exception() {
      promise.SetException(std::current_exception());
    }
  };
};
```

TFuture<void>::promise_type

```
// auto task = RunTerminal(endTs);
// transforms into

auto* promise = new promise_type;
auto task = promise->get_return_object();

auto h = MakeCoroutineHandle(promise);

if (promise->initial_suspend() == suspend_never) {
  h.resume();
}




task.GetValueSync();
```

```
template <typename... Args>
struct coroutine_traits<TFuture<void>, Args...> {
  struct promise_type {
    TPromise<void> promise;

    TFuture<void> get_return_object() {
      return promise.GetFuture();
    }

    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }

    void return_void() {
      promise.SetValue();
    }

    void unhandled_exception() {
      promise.SetException(std::current_exception());
    }
  };
};
```

TFuture<void>::promise_type

```
// auto task = RunTerminal(endTs);
// transforms into

auto* promise = new promise_type;
auto task = promise->get_return_object();

auto h = MakeCoroutineHandle(promise);

if (promise->initial_suspend() == suspend_never) {
  h.resume();
}




task.GetValueSync();
```

```
template <typename... Args>
struct coroutine_traits<TFuture<void>, Args...> {
  struct promise_type {
    TPromise<void> promise;

    TFuture<void> get_return_object() {
      return promise.GetFuture();
    }

    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }

    void return_void() {
      promise.SetValue();
    }

    void unhandled_exception() {
      promise.SetException(std::current_exception());
    }
  };
};
```

Awaitable TFuture<void>

```
TFuture<TxResult> RunTx(Input input);

// auto result = co_await RunTx(input);
// transforms into

TFuture<TxResult> tmp = RunTx(input);
auto awaiter = TFutureAwaiter<TxResult>
 { std::move(tmp) };

if (!awaiter.await_ready()) {
 // suspend current coroutine
 awaiter.await_suspend(
  std::coroutine_handle<>::from_address(...));
 return; // return to caller / to the resumer
}


// either Future was already ready,
// or coroutine was resumed later

auto result = awaiter.await_resume();
```

```
template <typename T>
struct TFutureAwaiter {
  TFuture<T> Future;


  bool await_ready() {
    return Future.IsReady();
  }


  void await_suspend(std::coroutine_handle<> handle) {
    Future.Subscribe([handle]() { handle.resume(); });
  }


  T await_resume() {
    return Future.Get();
  }
};


template <typename T>
TFutureAwaiter<T> operator co_await(TFuture<T>&& future) {
  return TFutureAwaiter<T>{std::move(future)};
}
```

Awaitable TFuture<void>

```
TFuture<TxResult> RunTx(Input input);

// auto result = co_await RunTx(input);
// transforms into

TFuture<TxResult> tmp = RunTx(input);
auto awaiter = TFutureAwaiter<TxResult>
 { std::move(tmp) };

if (!awaiter.await_ready()) {
 // suspend current coroutine
 awaiter.await_suspend(
  std::coroutine_handle<>::from_address(...));
 return; // return to caller / to the resumer
}


// either Future was already ready,
// or coroutine was resumed later

auto result = awaiter.await_resume();
```

```
template <typename T>
struct TFutureAwaiter {
  TFuture<T> Future;


  bool await_ready() {
    return Future.IsReady();
  }


  void await_suspend(std::coroutine_handle<> handle) {
    Future.Subscribe([handle]() { handle.resume(); });
  }


  T await_resume() {
    return Future.Get();
  }
};


template <typename T>
TFutureAwaiter<T> operator co_await(TFuture<T>&& future) {
  return TFutureAwaiter<T>{std::move(future)};
}
```

Awaitable TFuture<void>

```
TFuture<TxResult> RunTx(Input input);

// auto result = co_await RunTx(input);
// transforms into

TFuture<TxResult> tmp = RunTx(input);
auto awaiter = TFutureAwaiter<TxResult>
 { std::move(tmp) };

if (!awaiter.await_ready()) {
 // suspend current coroutine
 awaiter.await_suspend(
  std::coroutine_handle<>::from_address(...));
 return; // return to caller / to the resumer
}

// either Future was already ready,
// or coroutine was resumed later

auto result = awaiter.await_resume();
```

```
template <typename T>
struct TFutureAwaiter {
  TFuture<T> Future;


  bool await_ready() {
    return Future.IsReady();
  }


  void await_suspend(std::coroutine_handle<> handle) {
    Future.Subscribe([handle]() { handle.resume(); });
  }


  T await_resume() {
    return Future.Get();
  }
};


template <typename T>
TFutureAwaiter<T> operator co_await(TFuture<T>&& future) {
  return TFutureAwaiter<T>{std::move(future)};
}
```

Awaitable TFuture<void>

```
TFuture<TxResult> RunTx(Input input);

// auto result = co_await RunTx(input);
// transforms into


TFuture<TxResult> tmp = RunTx(input);
auto awaiter = TFutureAwaiter<TxResult>
 { std::move(tmp) };

if (!awaiter.await_ready()) {
 // suspend current coroutine
 awaiter.await_suspend(
  std::coroutine_handle<>::from_address(...));
 return; // return to caller / to the resumer
}


// either Future was already ready,
// or coroutine was resumed later

auto result = awaiter.await_resume();
```

```
template <typename T>
struct TFutureAwaiter {
  TFuture<T> Future;


  bool await_ready() {
    return Future.IsReady();
  }


  void await_suspend(std::coroutine_handle<> handle) {
    Future.Subscribe([handle]() { handle.resume(); });
  }


  T await_resume() {
    return Future.Get();
  }
};


template <typename T>
TFutureAwaiter<T> operator co_await(TFuture<T>&& future) {
  return TFutureAwaiter<T>{std::move(future)};
}
```

А где же потоки и пулы?

1. Все терминалы при вызове `RunTerminal()` начнут выполняться в текущем потоке

```
struct promise_type {
 std::suspend_never initial_suspend() noexcept;
};

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
 while (Now() < endTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);


  co_await Sleep(KeyingTime(type));

  auto result = co_await RunTx(input);
  Stats.Record(type, result);

  co_await Sleep(ThinkTime(type));
 }
}

// usage

std::vector<TFuture<void> tasks;
for (int i = 0; i < Terminals.size(); ++i) {
 tasks.emplace_back(RunTerminal(endTs));
}
```

А где же потоки и пулы?

1. Все терминалы при вызове `RunTerminal()` начнут выполняться в текущем потоке

2. Терминал выполняется в текущем потоке до первого `co_await`

```
struct promise_type {
 std::suspend_never initial_suspend() noexcept;
};

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
 while (Now() < endTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);


  co_await Sleep(KeyingTime(type));

  auto result = co_await RunTx(input);
  Stats.Record(type, result);

  co_await Sleep(ThinkTime(type));
 }
}

// usage

std::vector<TFuture<void> tasks;
for (int i = 0; i < Terminals.size(); ++i) {
 tasks.emplace_back(RunTerminal(endTs));
}
```

А где же потоки и пулы?

1. Все терминалы при вызове `RunTerminal()` начнут выполняться в текущем потоке

2. Терминал выполняется в текущем потоке до первого `co_await`

3. Дальше терминалы выполняются в потоке, который проставляет значение `future`

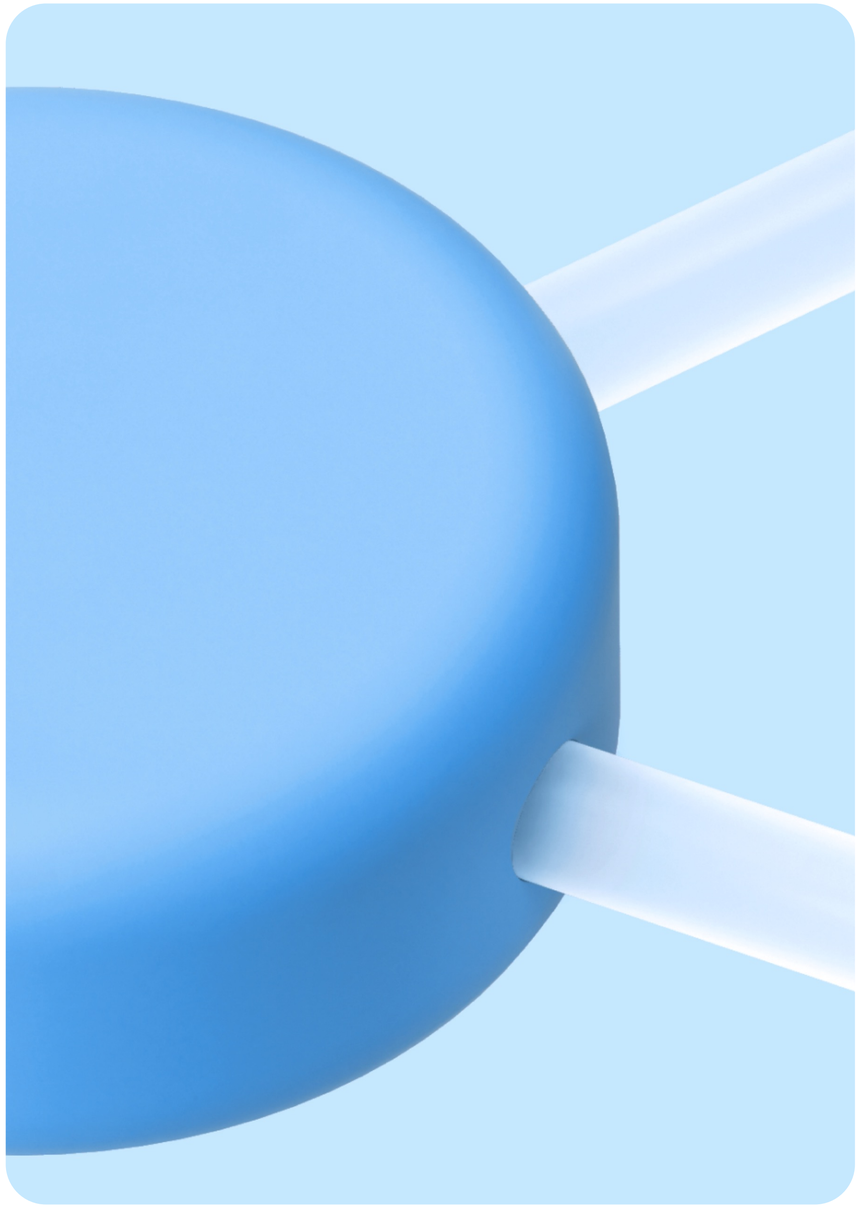
Очень напоминает первый вариант на фьючах без пулов!

```
struct promise_type {
 std::suspend_never initial_suspend() noexcept;
};

// coroutine
TFuture<void> RunTerminal(TInstant endTs) {
 while (Now() < endTs) {
  auto type = PickTransaction();
  auto input = MakeInput(type);


  co_await Sleep(KeyingTime(type));

  auto result = co_await RunTx(input);
  Stats.Record(type, result);

  co_await Sleep(ThinkTime(type));
 }
}

// usage

std::vector<TFuture<void> tasks;
for (int i = 0; i < Terminals.size(); ++i) {
 tasks.emplace_back(RunTerminal(endTs));
}
```

co_await и перекладка в пул

1. Добавляем монотонно растущий уникальный `TerminalID`
2. В пуле N-потоков. Терминал выполняется всегда в потоке `threadHint = TerminalID % N`
3. В начале `RunTerminal()` перекладываем в нужный поток при необходимости

```
TFuture<void> RunTerminal(TInstant endTs) {
  auto threadHint = TerminalID % Pool.size();
  co_await TScheduleOnPool(pool, threadHint);
  while (Now() < context.EndTs) {
  // ...
}
```



```cpp
// await until coroutine is resumed inside thread pool
struct TScheduleOnPool {
  TScheduleOnPool(IThreadPool& pool, size_t threadHint)
  : Pool(pool)
  , ThreadHint(threadHint)
  {}

  bool await_ready() {
    return Pool.IsInThisThread(threadHint);
  }

  bool await_suspend(std::coroutine_handle<> h) {
    // already on pool thread → continue immediately
    if (Pool.IsInThisThread(threadHint)) {
      return false;
    }

    // schedule continuation into pool
    Pool.Schedule(h, ThreadHint);
    return true;
  }

  void await_resume() {}

  IThreadPool& Pool;
  size_t ThreadHint;
};
```

Back to the future в нужном потоке

1. Во многих реализациях `future` код корутины будет продолжен в том потоке, который проставил значение `future`

2. В асинхронном API СУБД часто это поток СУБД

3. Мы не хотим выполнять наш код в «чужих» потоках

```
auto r1 = co_await tx.Query(sql1);
```

Back to the future в нужном потоке

1. Во многих реализациях `future` код корутины будет продолжен в том потоке, который проставил значение `future`

2. В асинхронном API СУБД часто это поток СУБД

3. Мы не хотим выполнять наш код в «чужих» потоках

```
auto r1 = co_await tx.Query(sql1);


auto r1 = co_await TAwaitFutureOnPool(
  tx.Query(sql1),
  pool,
  threadHint);
```

```cpp
template <typename T>
struct TAwaitFutureOnPool {
  TAwaitFutureOnPool(TFuture<T>& future, IThreadPool& pool, size_t threadHint)
    : Future(future)
    , Pool(pool)
    , ThreadHint(threadHint)
  {}

  bool await_ready() {
    return Future.HasValue();
  }

  void await_suspend(std::coroutine_handle<> h) {
    Future.Subscribe([this, h](const TFuture<T>&) {
      Pool.Schedule(h, ThreadHint);
    });
  }

  T await_resume() {
    return Future.GetValue();
  }

  TFuture<T>& Future;
  IThreadPool& Pool;
  size_t ThreadHint;
};
```

Финальный терминал

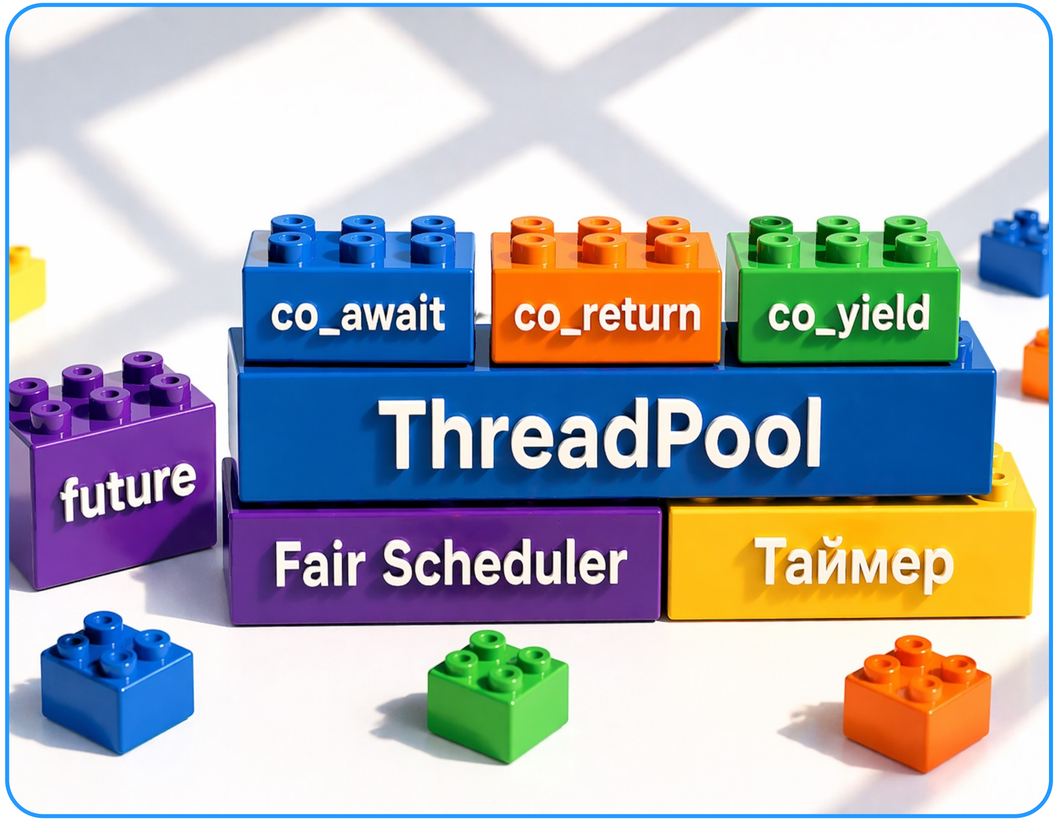
```
TFuture<void> Sleep(TDuration d);
TFuture<TxResult> RunTx(Input input);

TFuture<void> RunTerminal(TInstant endTs, IThreadPool& pool, ui64 terminalId) {
  auto threadHint = terminalID % pool.size();

  co_await TScheduleOnPool(pool, threadHint);

  while (Now() < endTs) {
    auto type = PickTransaction();
    auto input = MakeInput(type);

    auto f1 = Sleep(KeyingTime(type));
    co_await TSuspendWithFuture<void>(f1, pool, threadHint);

    auto f2 = RunTx(input);
    auto result = co_await TSuspendWithFuture<TxResult>(f2, pool, threadHint);
    Stats.Record(type, result);

    auto f3 = Sleep(ThinkTime(type));
    co_await TSuspendWithFuture<void>(f3, pool, threadHint);
  }
}
```

Стандарт даёт нам только три слова

1. Кроме трёх слов из стандарта
   и небольшой помощи со стороны
   компилятора, ничего нет

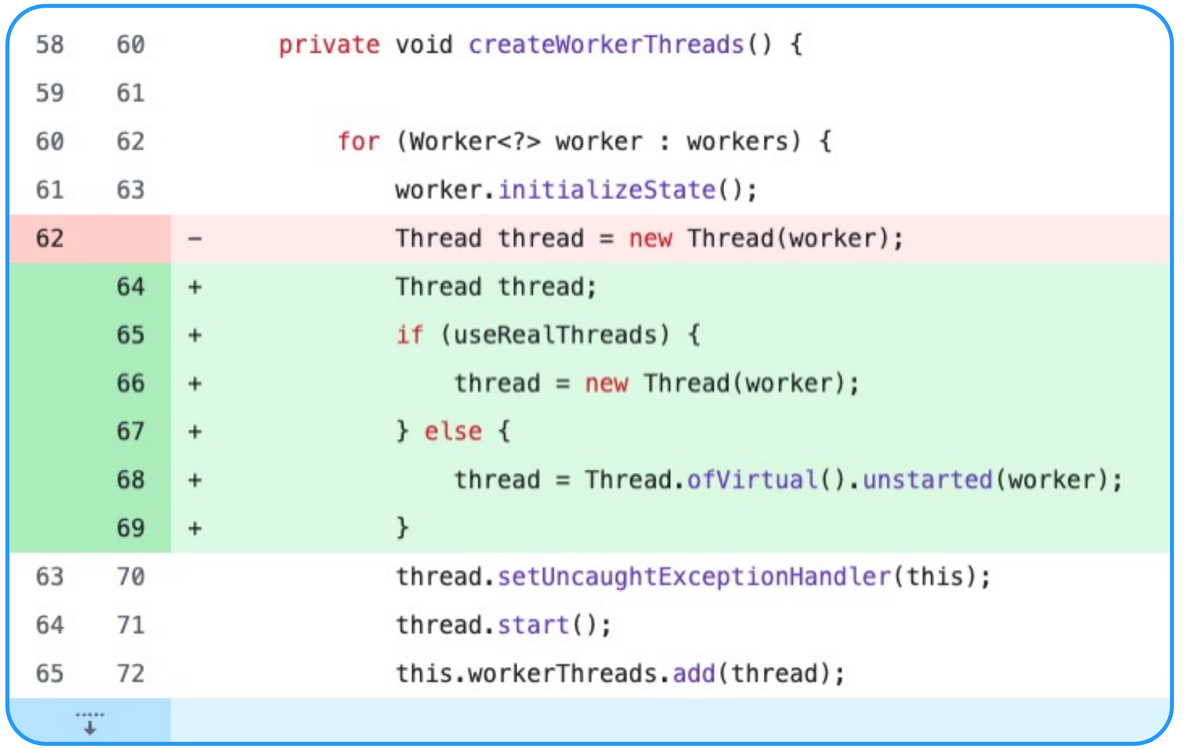
2. Фьючи, пул потоков, планировщик
   и таймер — пришлось написать самим.
   Всё это есть и в других сторонних
   библиотеках

3. «Сырые» корутины скорее
   для разработчиков библиотек,
   а не приложений. Если использовать
   обёртки, то всё не так плохо



Не наброс, но в Java...

- Переход на virtual threads — одна строка
- Но есть нюанс:
  «Как мы начали использовать виртуальные потоки Java 21 и на раз-два получили дедлок в TPC-C для PostgreSQL»

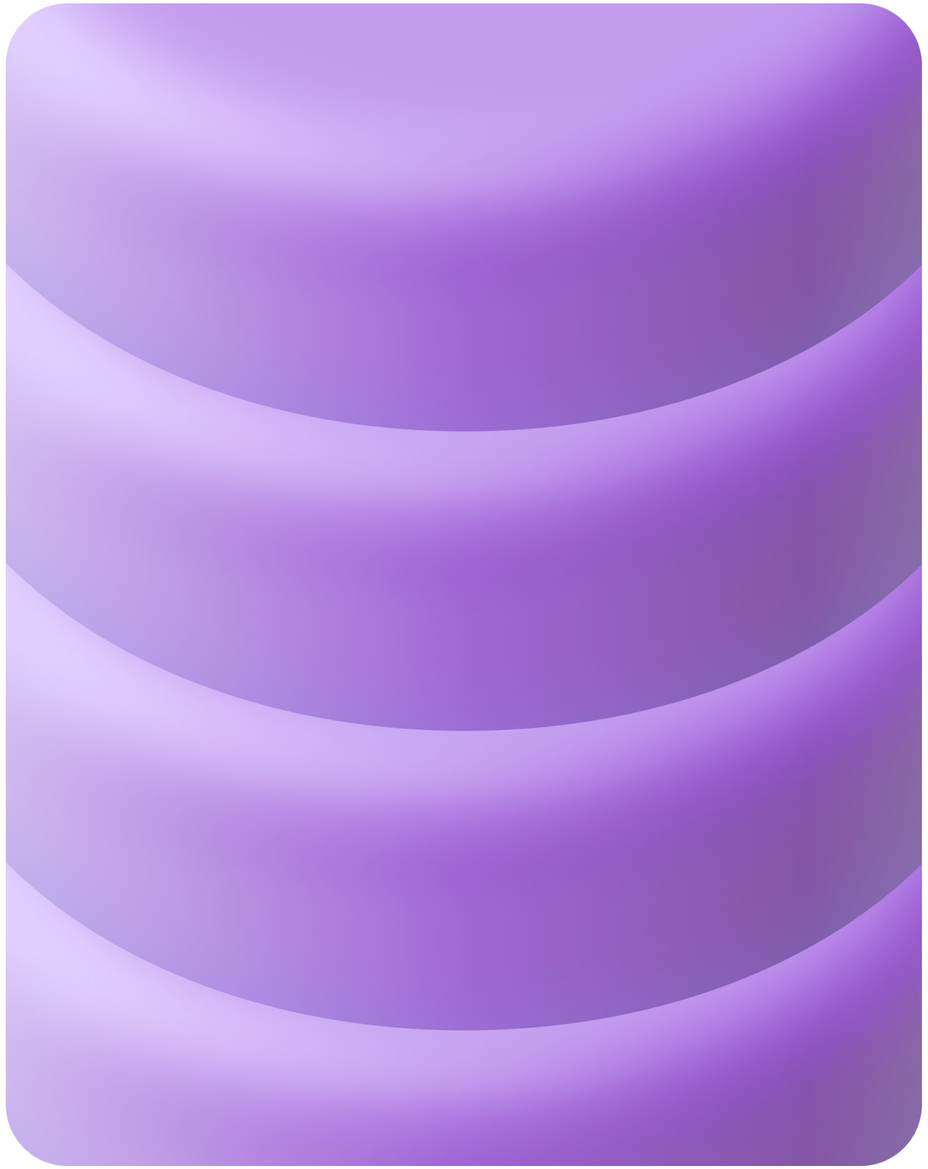


```
58  60          private void createWorkerThreads() {
59  61
60  62              for (Worker<?> worker : workers) {
61  63                  worker.initializeState();
62      -           Thread thread = new Thread(worker);
    64  +           Thread thread;
    65  +           if (useRealThreads) {
    66  +               thread = new Thread(worker);
    67  +           } else {
    68  +               thread = Thread.ofVirtual().unstarted(worker);
    69  +           }
63  70              thread.setUncaughtExceptionHandler(this);
64  71              thread.start();
65  72              this.workerThreads.add(thread);
```

А что в других языках?

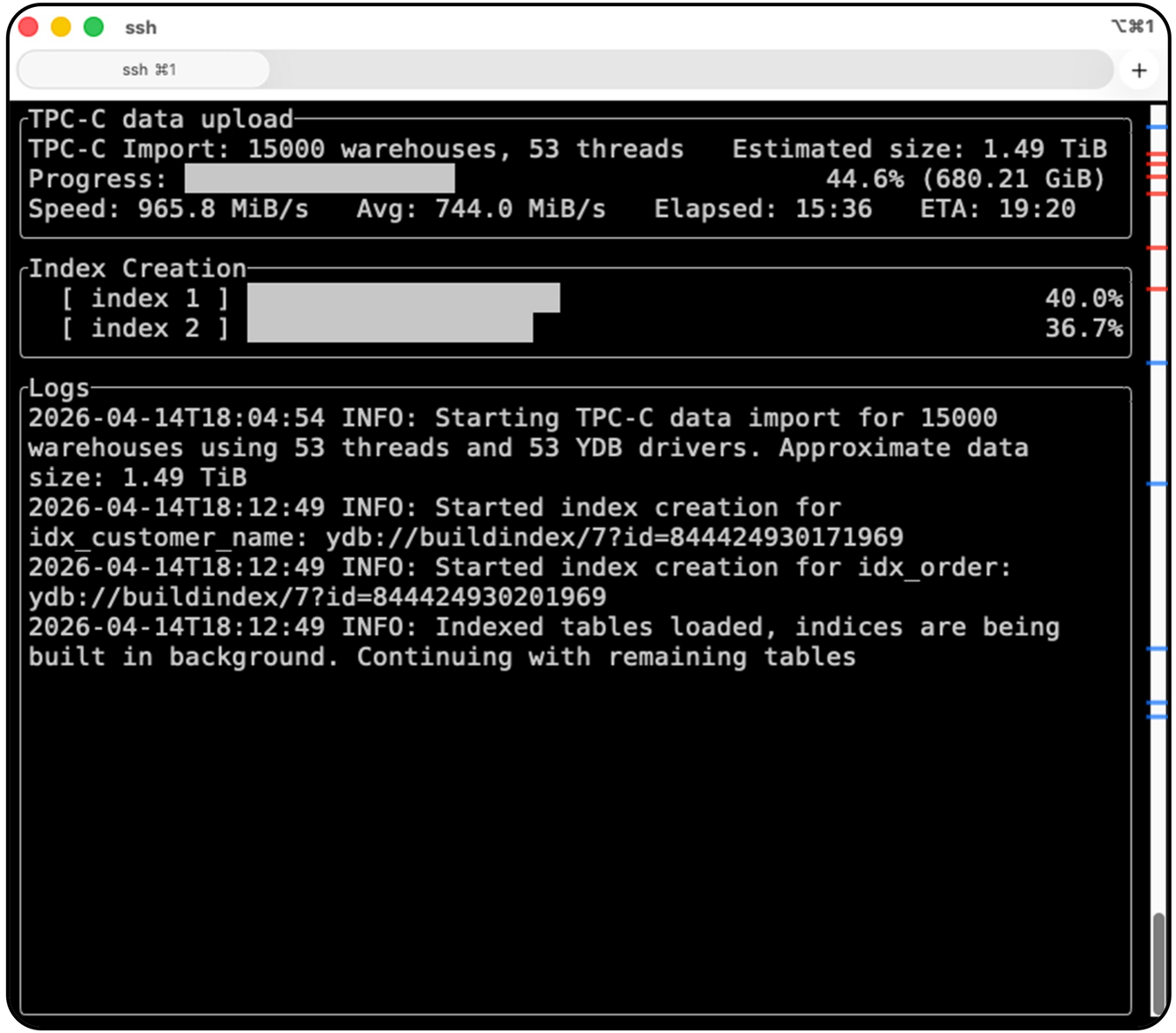
Фича	C++	Java	Go
Тип	stackless	stackful (virtual threads)	stackful (goroutines)
Runtime	нет	JVM	Go runtime
Scheduler	нет	JVM	Go runtime
Итеграция I/O	нет	есть	супер
Входной порог	крайне высокий	низкий	очень низкий
Контроль	полный	низкий	низкий

Наш TPC-C

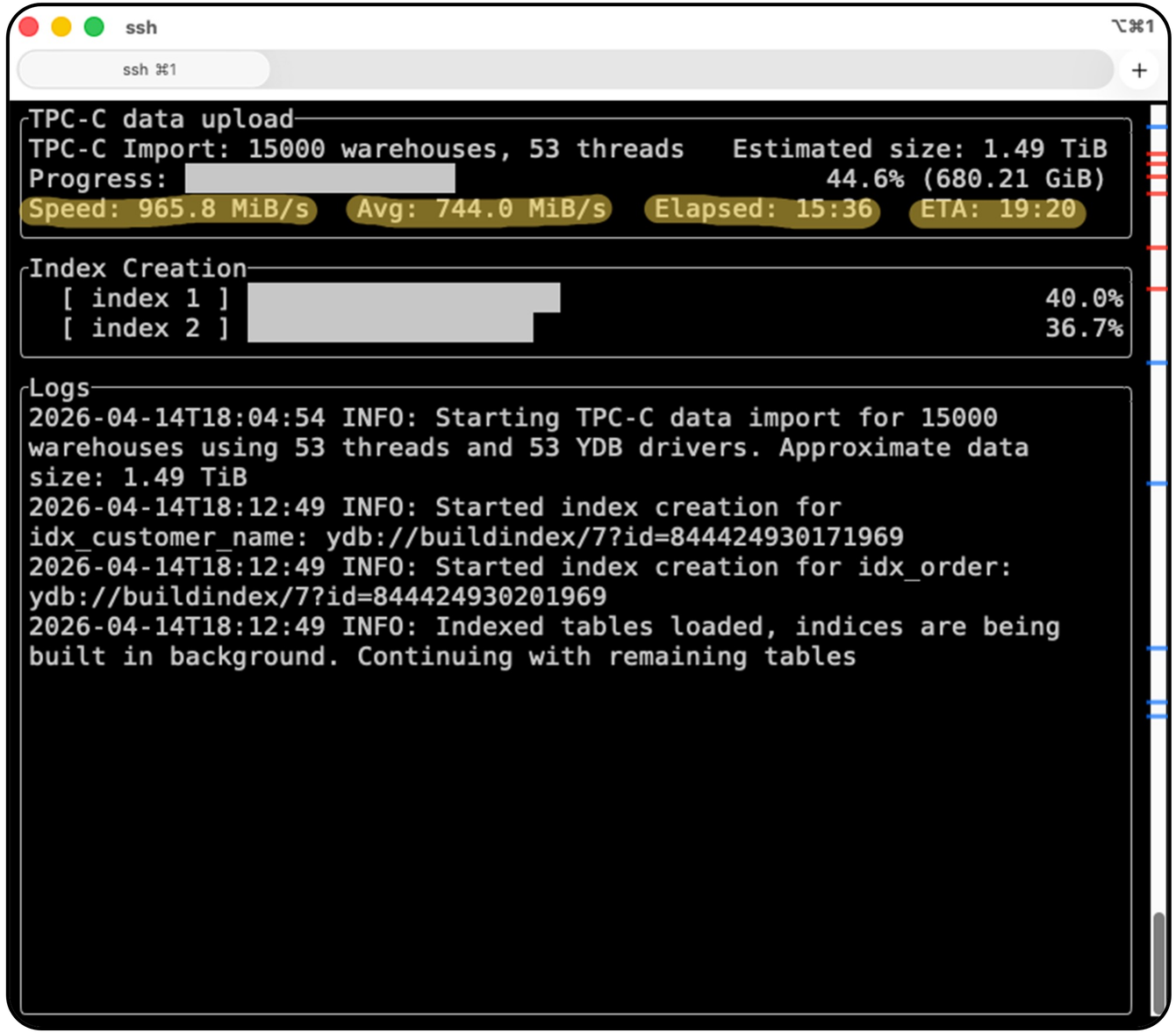


04

TPC-C import

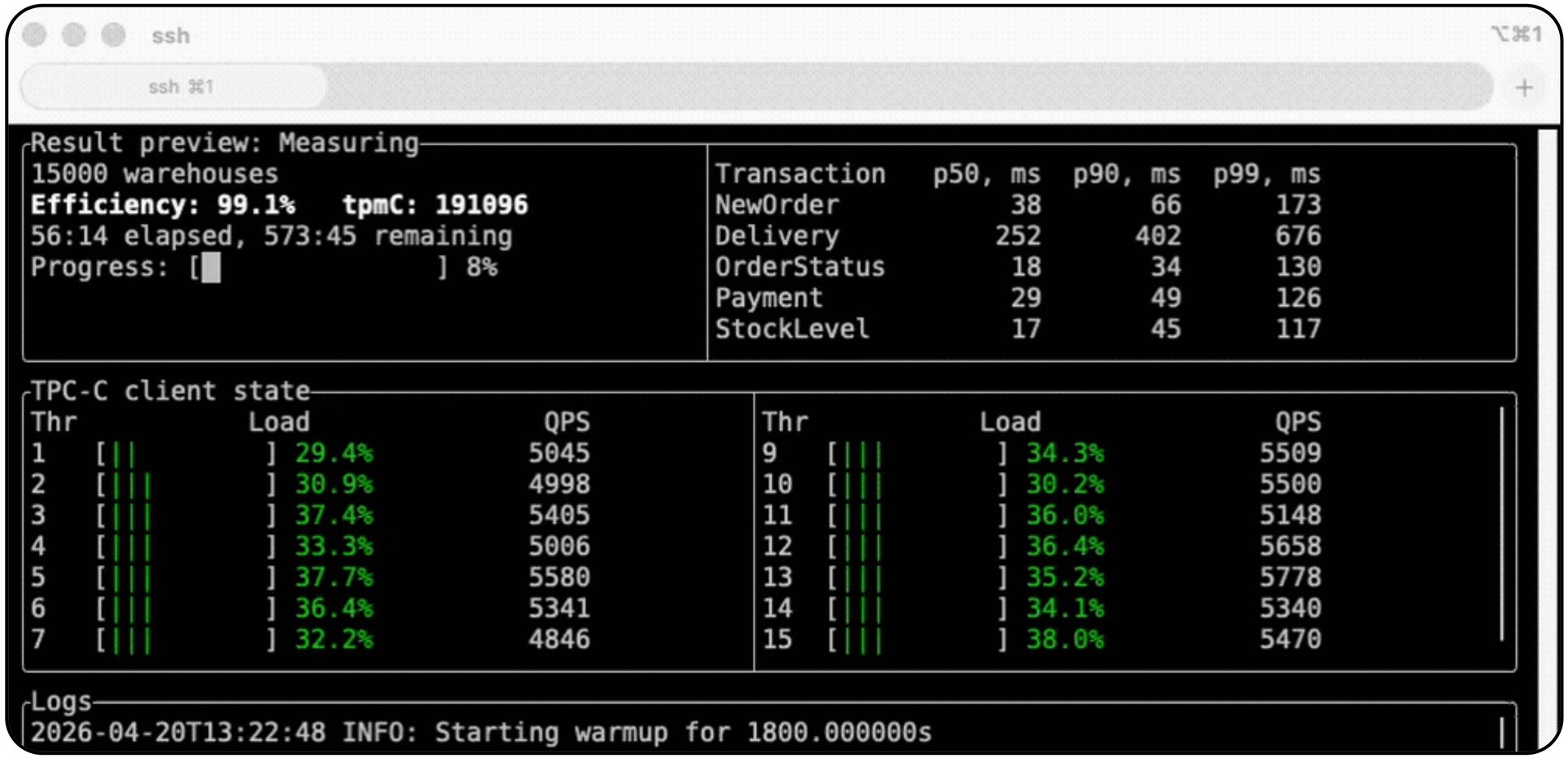

ssh
⌥⌘1
ssh ⌘1
TPC-C data upload
TPC-C Import: 15000 warehouses, 53 threads Estimated size: 1.49 TiB
Progress: 44.6% (680.21 GiB)
Speed: 965.8 MiB/s Avg: 744.0 MiB/s Elapsed: 15:36 ETA: 19:20
Index Creation
[index 1] 40.0%
[index 2] 36.7%
Logs
2026-04-14T18:04:54 INFO: Starting TPC-C data import for 15000 warehouses using 53 threads and 53 YDB drivers. Approximate data size: 1.49 TiB
2026-04-14T18:12:49 INFO: Started index creation for idx_customer_name: ydb://buildindex/7?id=844424930171969
2026-04-14T18:12:49 INFO: Started index creation for idx_order: ydb://buildindex/7?id=844424930201969
2026-04-14T18:12:49 INFO: Indexed tables loaded, indices are being built in background. Continuing with remaining tables

TPC-C import

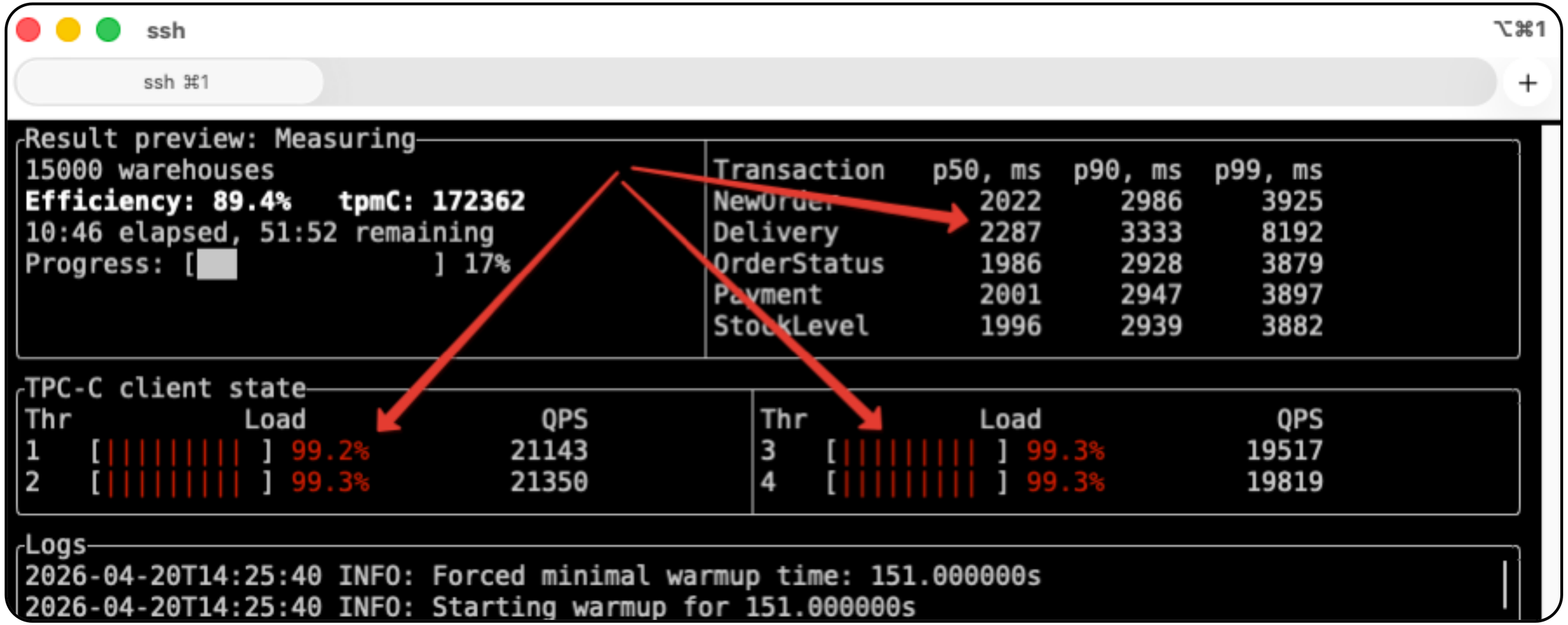

ssh
⌥⌘1
ssh ⌘1
TPC-C data upload
TPC-C Import: 15000 warehouses, 53 threads Estimated size: 1.49 TiB
Progress: 44.6% (680.21 GiB)
Speed: 965.8 MiB/s Avg: 744.0 MiB/s Elapsed: 15:36 ETA: 19:20
Index Creation
[index 1] 40.0%
[index 2] 36.7%
Logs
2026-04-14T18:04:54 INFO: Starting TPC-C data import for 15000 warehouses using 53 threads and 53 YDB drivers. Approximate data size: 1.49 TiB
2026-04-14T18:12:49 INFO: Started index creation for idx_customer_name: ydb://buildindex/7?id=844424930171969
2026-04-14T18:12:49 INFO: Started index creation for idx_order: ydb://buildindex/7?id=844424930201969
2026-04-14T18:12:49 INFO: Indexed tables loaded, indices are being built in background. Continuing with remaining tables

TPC-C run

- CPU: среднее потребление 25 ядер
- RAM: 520 MiB, вместо 600 GiB
- потоки: 500 вместо 150000



TPC-C run: не хватило потоков



Корутины ещё не гарантия скорости

Мы обнаружили значительные тормоза, не связанные с корутинами:

- thundering herd или запускаем 150K терминалов разом
- релизная сборка с debuginfo сильно замедляет gRPC
- нашли интересный gRPC channel bottleneck, о чём написали пост

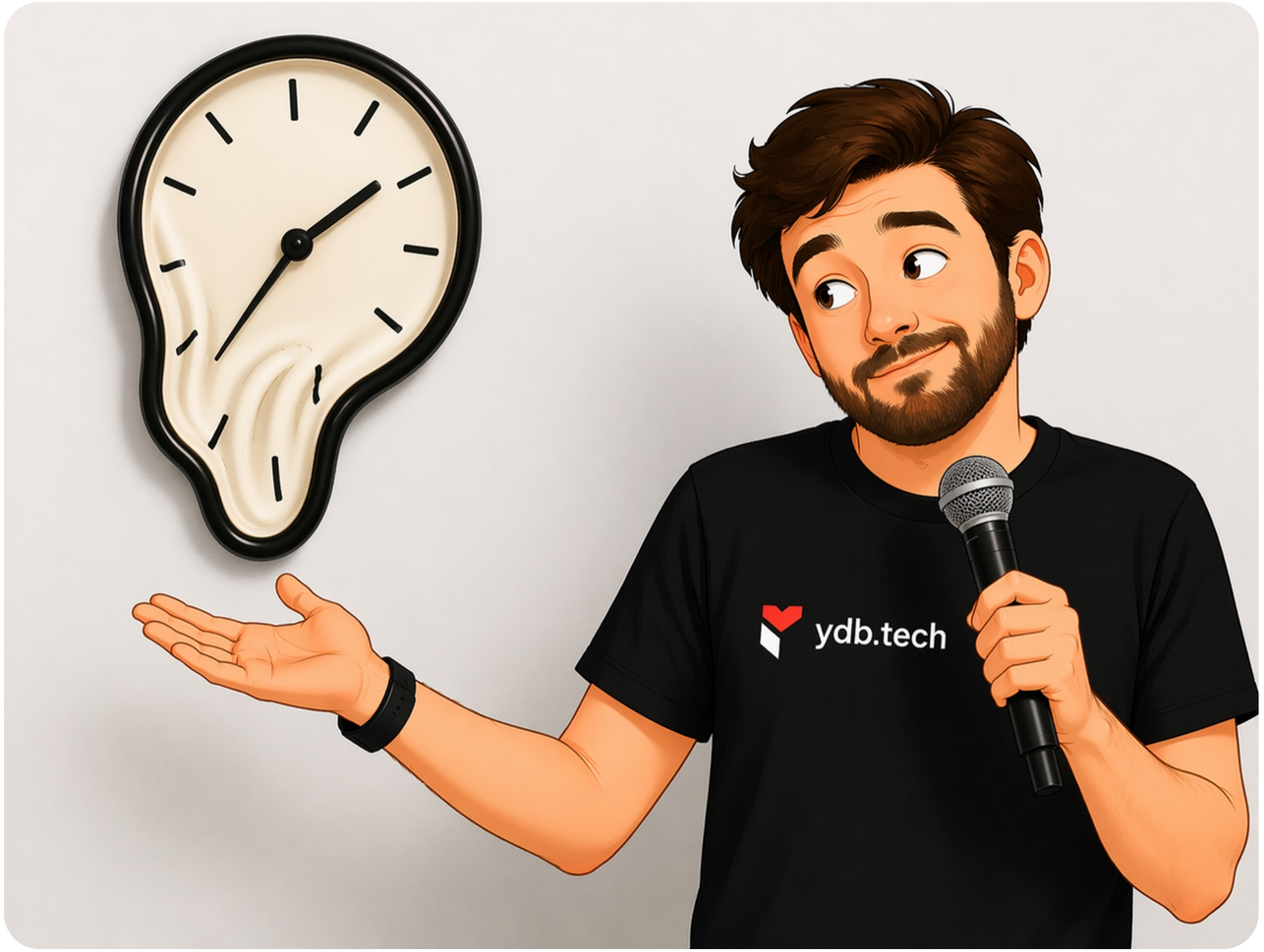
Бонус: теперь TPC-C и для PostgreSQL

- Использовали Opus 4.6, интересный эксперимент. Без работы мы, программисты, пока не останемся
- будем использовать для тестирования YDB в режиме совместимости с PostgreSQL
- код

Не хватило времени рассказать

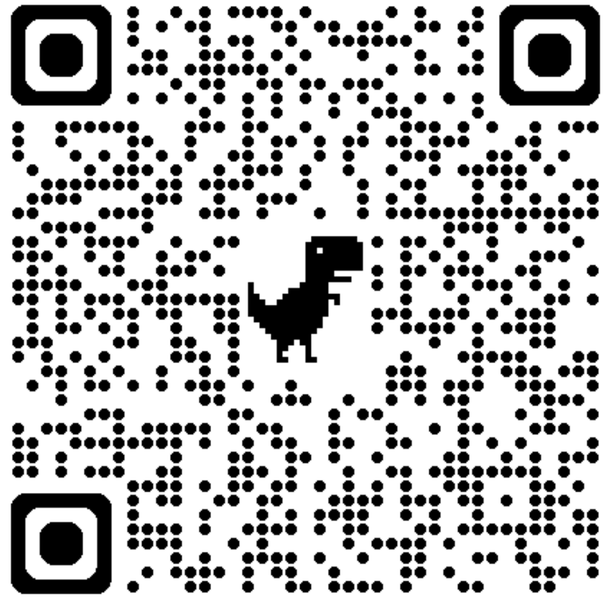
- FIFO Scheduler
  для наших корутин
- Timer thread
- Тонкости корутин

Но всё есть в коде
и дополнительных
материалах! ☺



Заключение

- Корутины позволяют писать эффективный и понятный код
- К сожалению, стандартная библиотека не предоставляет рантайма для корутин
- Мы написали свой рантайм и сделали эффективную реализацию TPC-C



Слайды, исходный код и другие материалы

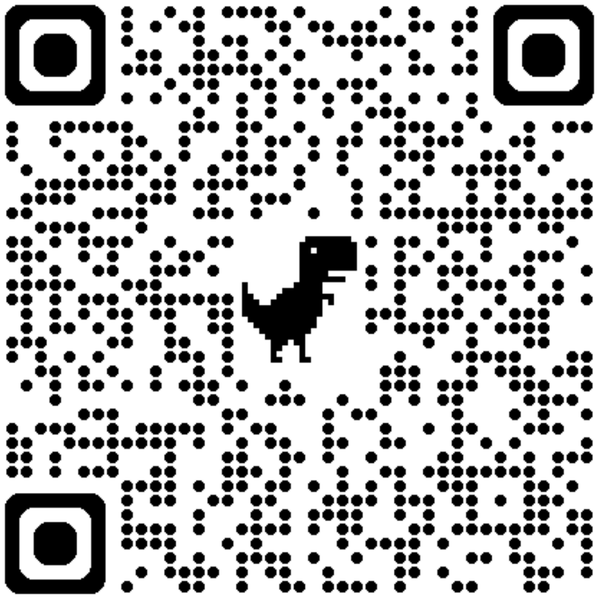
YDB

C++ Russia

Разработка библиотек vs. приложений

Заключение

- Корутины позволяют писать эффективный и понятный код
- К сожалению, стандартная библиотека не предоставляет рантайма для корутин
- Мы написали свой рантайм и сделали эффективную реализацию TPC-C



Слайды, исходный код и другие материалы

YDB

C++ Russia