Yandex Cloud Joker

Интеграция распределенных SQL хранилищ с Java-фреймворками на примере YDB

Курдюков Кирилл Алексеевич,
Разработчик YDB

Зачем возвращаться к теме JDBC, JPA, Spring Data и других технологий?

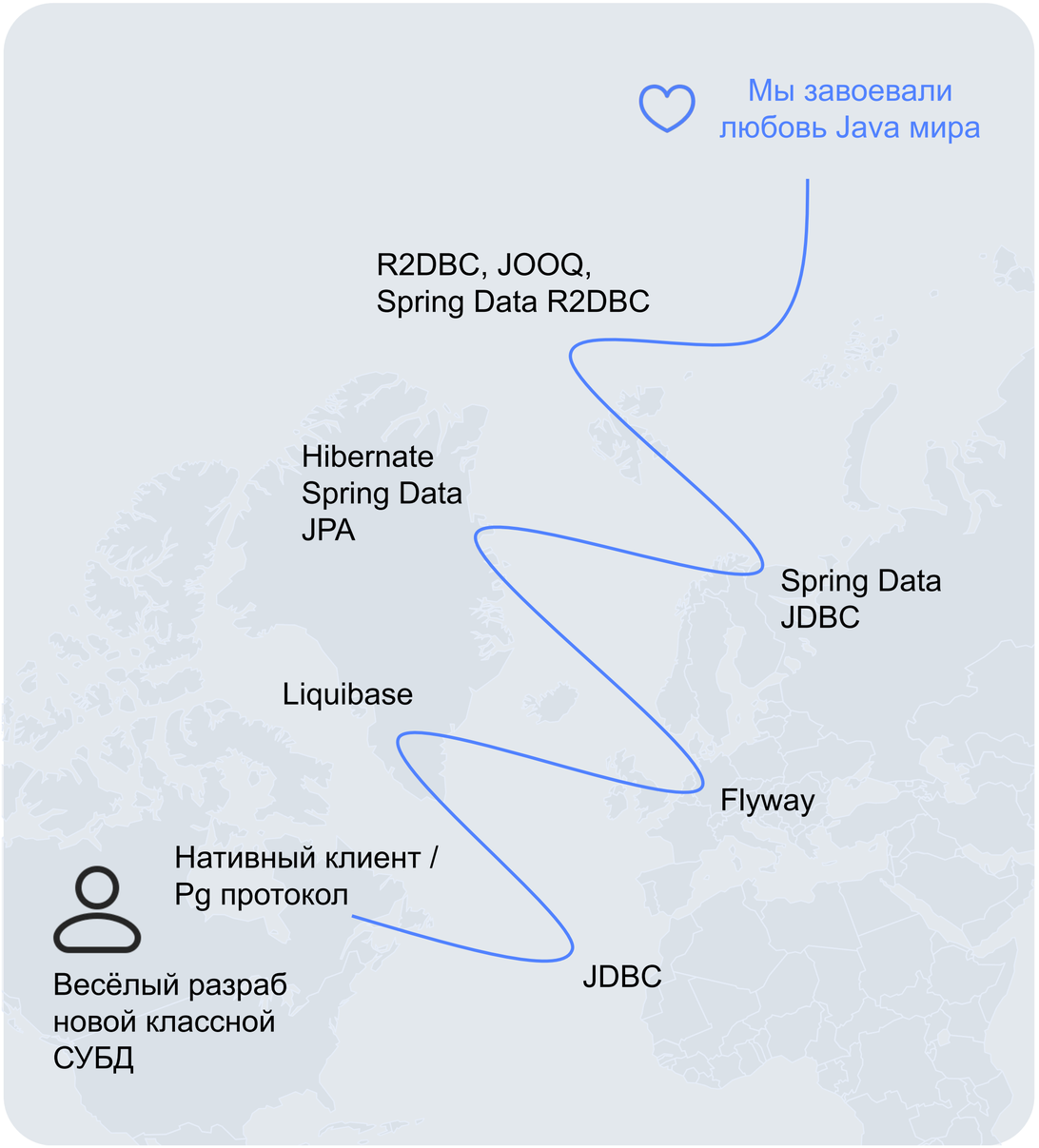
- Каждый инструмент детально разобран в отдельности

Зачем возвращаться к теме JDBC, JPA, Spring Data и других технологий?

- Каждый инструмент детально разобран в отдельности
- А что если взглянуть на этот ансамбль под другим углом?

Подход разработчика СУБД

На примере YDB пройдем по пути реализации стандарта JDBC до интеграции со всем основным Java стэком, оглядываясь на другие хранилища категории Distributed SQL.

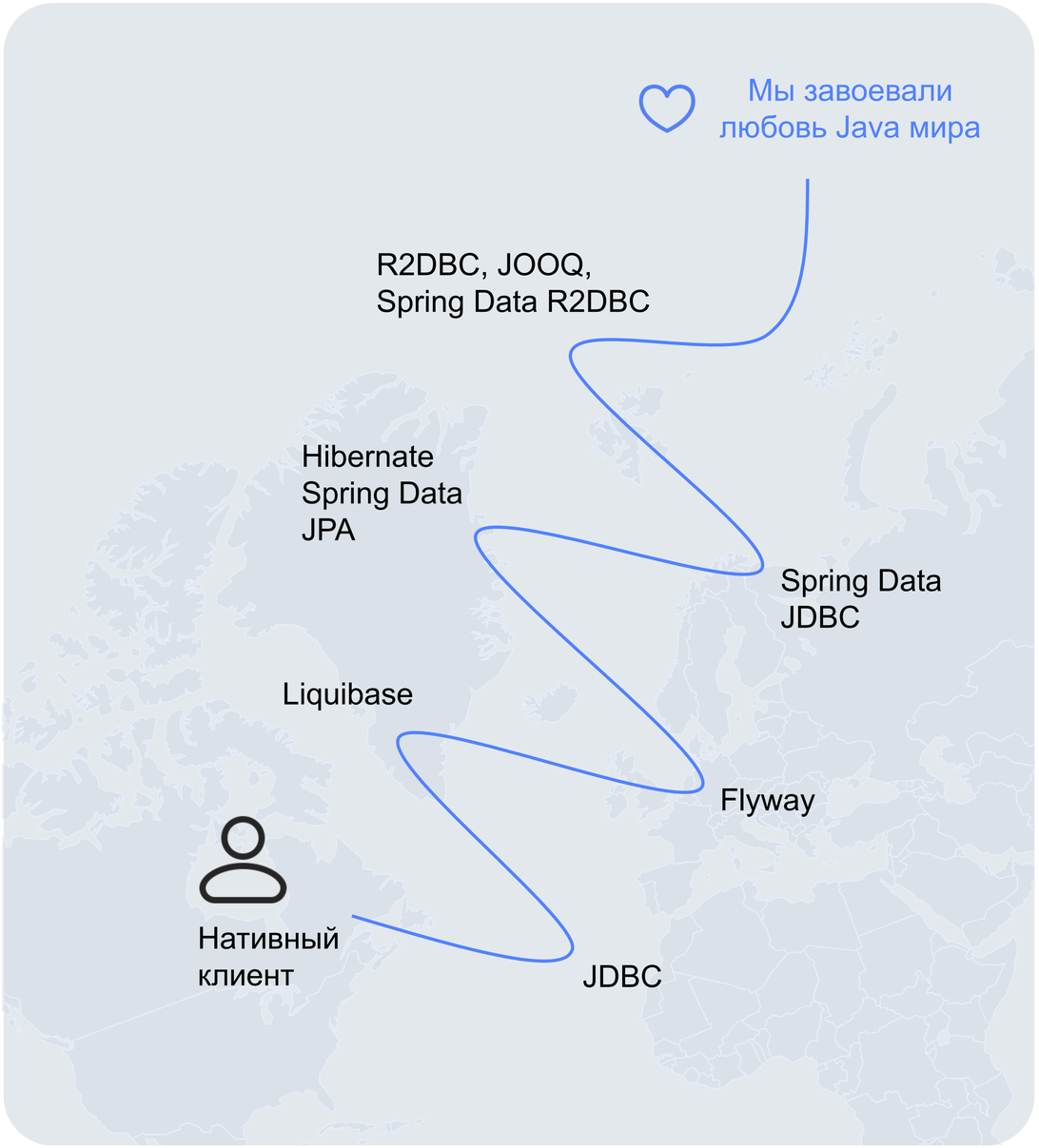


Нативный клиент

Если мы говорим о распределенных СУБД, то они, как правило, имеют своих нативных клиентов.

Примеры:

- YDB
- Google Spanner
- MongoDB
- Cassandra

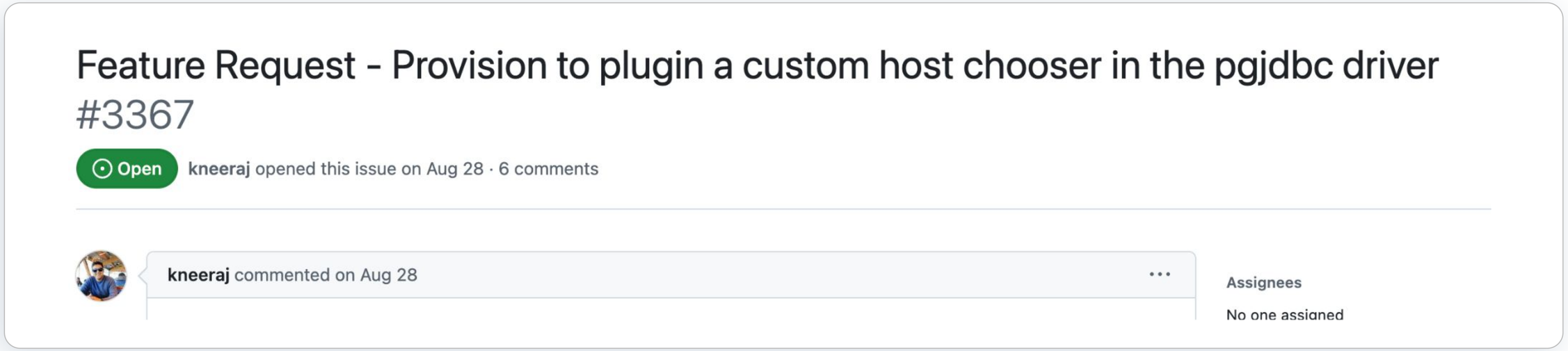


Реализация Pg протокола

Примеры:

- YugabyteDB (тащит бинарь Pg server)
- CockroachDB (Pg протокол на Go)

Проблемы с Pg подходом



Ждем такие issue для ADO.NET, database/sql…

Промежуточный итог

Нативный клиент дает больше гибкости (каких-либо оптимизаций). Разработка и архитектура полностью контролируется вендором.

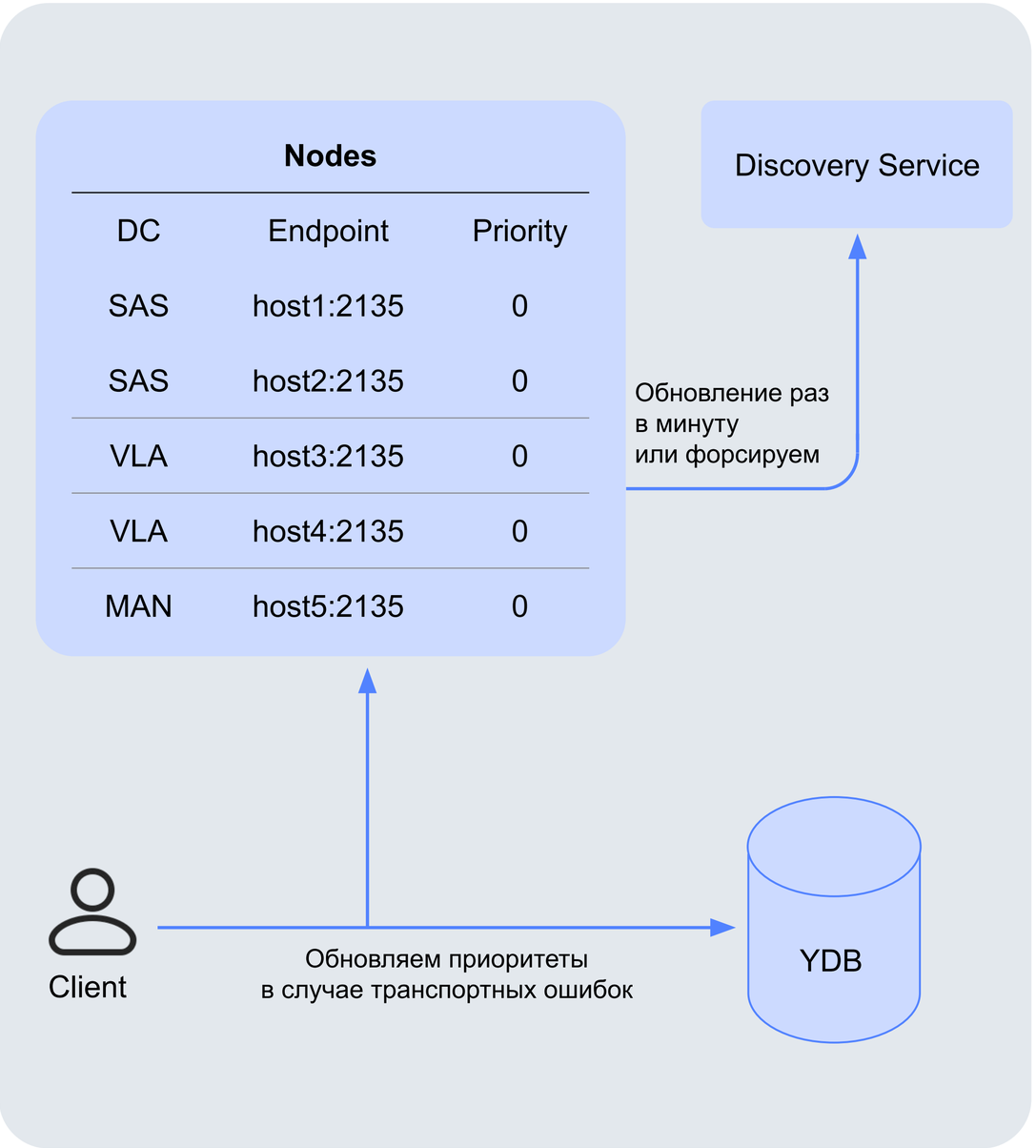
API новое и требуется поддержка всего Java стэка заново.

Мы эмулируем поведение PostgreSQL, таким образом, все инструменты, разработанные для PostgreSQL, теперь совместимы и с нашей СУБД.

Но мы же не Pg? PostgreSQL протокол становится точкой конфронтации различных вендоров.

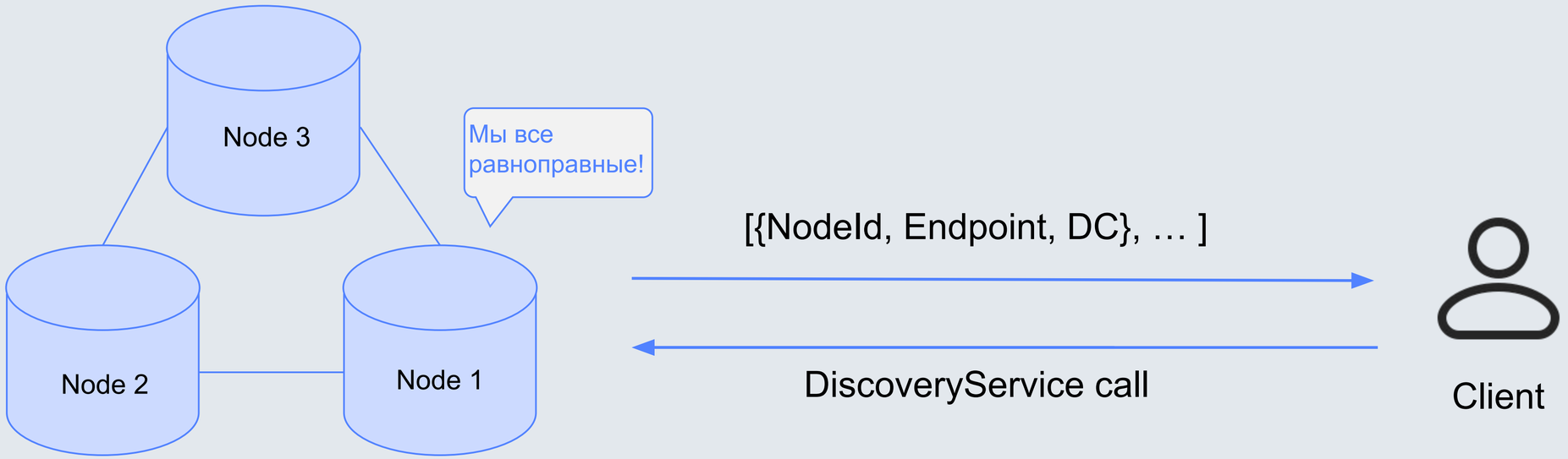
Какую задачу решает нативный клиент?

- Поддерживает топологию кластера
- Балансирует клиентские запросы
- Расставляет узлам приоритеты



Топология кластера на клиенте YDB

Важно: хосты узлов YDB необязательно являются статичными.



Топология кластера на клиенте YDB

Важно: хосты узлов YDB необязательно являются статичными.

Endpoint Map

NodeId	Endpoint	Priority
1	host1:2135	0
2	host2:2135	0
3	host3:2135	1000

Топология кластера на клиенте YDB

Важно: хосты узлов YDB необязательно являются статичными.

Endpoint Map

NodeId	Endpoint	Priority
1	host1:2135	0
2	host2:2135	0
3	host3:2135	1000

gRPC Channel Pool
size ~ O(размер кластера)

Взглянем на код нативного клиента?

```
String query
       = "DECLARE $seriesId AS Uint64; "
       + "DECLARE $seasonId AS Uint64; "
       + "SELECT sa.title AS season_title, sr.title AS series_title "
       + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
       + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

// Type of parameter values should be exactly the same as in DECLARE statements.
Params params = Params.of(
       "$seriesId", PrimitiveValue.newUint64(seriesID),
       "$seasonId", PrimitiveValue.newUint64(seasonID)
);

QueryReader result = retryCtx.supplyResult(
       session -> QueryReader.readFrom(session.createQuery(query, TxMode.SNAPSHOT_RO, params))
).join().getValue();

logger.info("--[ SelectWithParams ] -- ");

ResultSetReader rs = result.getResultSet(0);
while (rs.next()) {
   logger.info("read season with title {} for series {}",
           rs.getColumn("season_title").getText(),
           rs.getColumn("series_title").getText()
   );
}
```

Полный пример

Пишем руками YQL

```
String query
       = "DECLARE $seriesId AS Uint64; "
       + "DECLARE $seasonId AS Uint64; "
       + "SELECT sa.title AS season_title, sr.title AS series_title "
       + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
       + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

// Type of parameter values should be exactly the same as in DECLARE statements.
Params params = Params.of(
       "$seriesId", PrimitiveValue.newUint64(seriesID),
       "$seasonId", PrimitiveValue.newUint64(seasonID)
);

QueryReader result = retryCtx.supplyResult(
       session -> QueryReader.readFrom(session.createQuery(query, TxMode.SNAPSHOT_RO, params))
).join().getValue();

logger.info("--[ SelectWithParams ] -- ");

ResultSetReader rs = result.getResultSet(0);
while (rs.next()) {
   logger.info("read season with title {} for series {}",
           rs.getColumn("season_title").getText(),
           rs.getColumn("series_title").getText()
   );
}
```

Объявляем переменные

```
String query
        = "DECLARE $seriesId AS Uint64; "
        + "DECLARE $seasonId AS Uint64; "
        + "SELECT sa.title AS season_title, sr.title AS series_title "
        + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
        + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

// Type of parameter values should be exactly the same as in DECLARE statements.
Params params = Params.of(
        "$seriesId", PrimitiveValue.newUint64(seriesID),
        "$seasonId", PrimitiveValue.newUint64(seasonID)
);

QueryReader result = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery(query, TxMode.SNAPSHOT_RO, params))
).join().getValue();

logger.info("--[ SelectWithParams ] -- ");

ResultSetReader rs = result.getResultSet(0);
while (rs.next()) {
    logger.info("read season with title {} for series {}",
            rs.getColumn("season_title").getText(),
            rs.getColumn("series_title").getText()
    );
}
```

Исполняем запрос

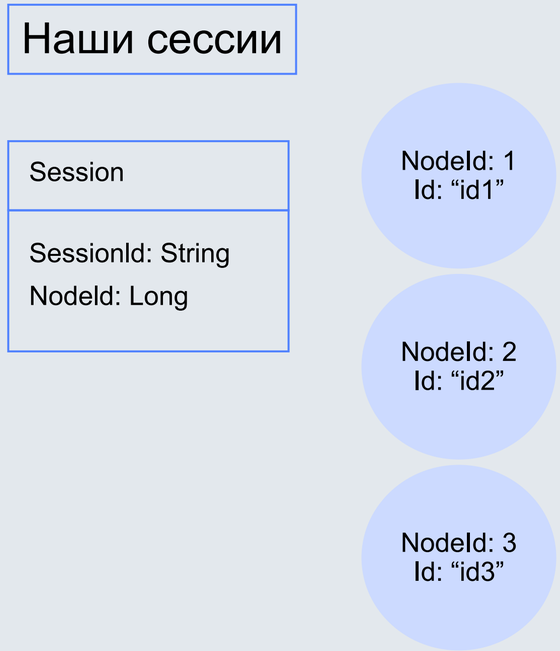
```
String query
        = "DECLARE $seriesId AS Uint64; "
        + "DECLARE $seasonId AS Uint64; "
        + "SELECT sa.title AS season_title, sr.title AS series_title "
        + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
        + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

// Type of parameter values should be exactly the same as in DECLARE statements.
Params params = Params.of(
        "$seriesId", PrimitiveValue.newUint64(seriesID),
        "$seasonId", PrimitiveValue.newUint64(seasonID)
);

QueryReader result = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery(query, TxMode.SNAPSHOT_RO, params))
).join().getValue();

logger.info("--[ SelectWithParams ] -- ");

ResultSetReader rs = result.getResultSet(0);
while (rs.next()) {
    logger.info("read season with title {} for series {}",
            rs.getColumn("season_title").getText(),
            rs.getColumn("series_title").getText()
    );
}
```

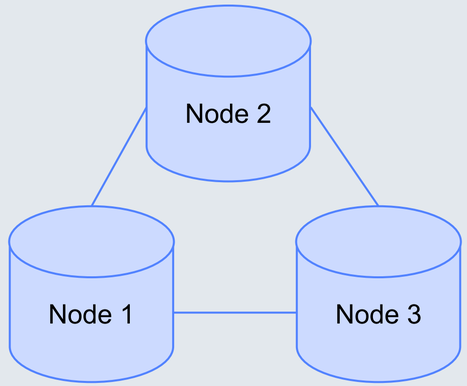
Читаем ResultSet

```
String query
        = "DECLARE $seriesId AS Uint64; "
        + "DECLARE $seasonId AS Uint64; "
        + "SELECT sa.title AS season_title, sr.title AS series_title "
        + "FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id "
        + "WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId";

// Type of parameter values should be exactly the same as in DECLARE statements.
Params params = Params.of(
        "$seriesId", PrimitiveValue.newUint64(seriesID),
        "$seasonId", PrimitiveValue.newUint64(seasonID)
);

QueryReader result = retryCtx.supplyResult(
        session -> QueryReader.readFrom(session.createQuery(query, TxMode.SNAPSHOT_RO, params))
).join().getValue();

logger.info("--[ SelectWithParams ] -- ");

ResultSetReader rs = result.getResultSet(0);
while (rs.next()) {
    logger.info("read season with title {} for series {}",
            rs.getColumn("season_title").getText(),
            rs.getColumn("series_title").getText()
    );
}
```

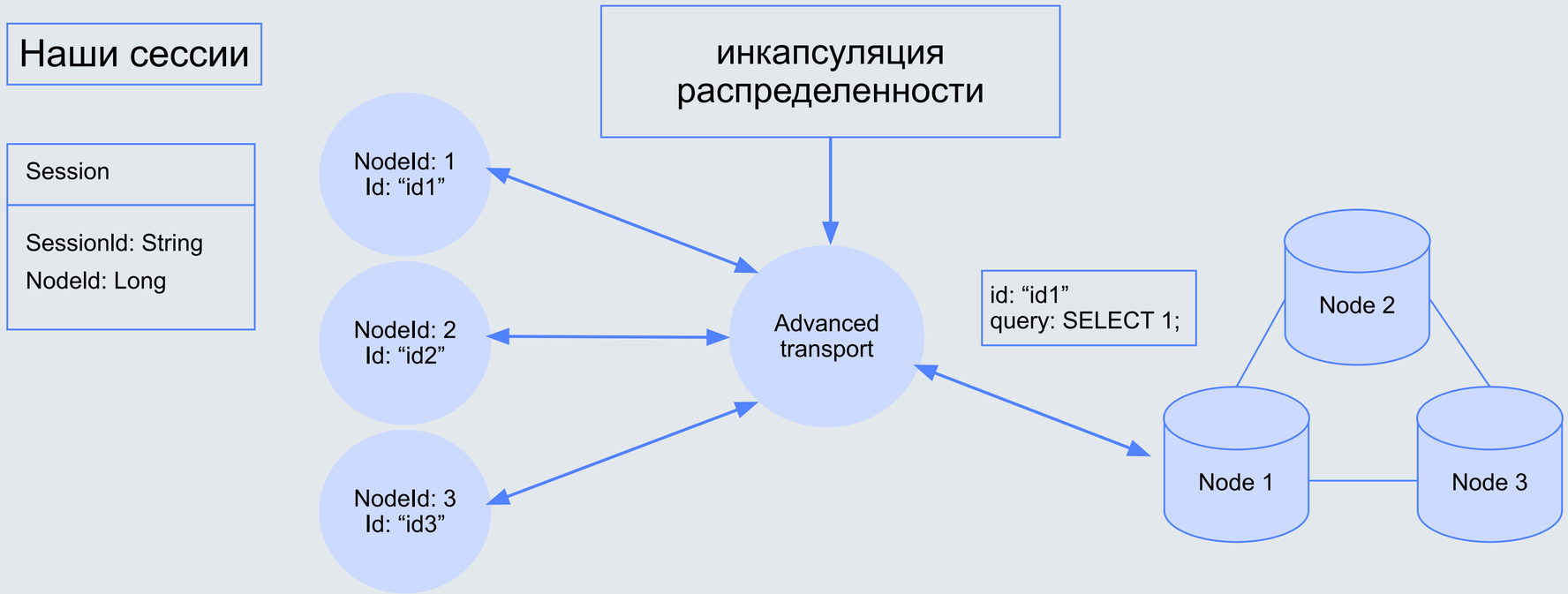
Хочешь исполнить запрос - держи сессию!

Сессия не считается долгоживущим ресурсом.



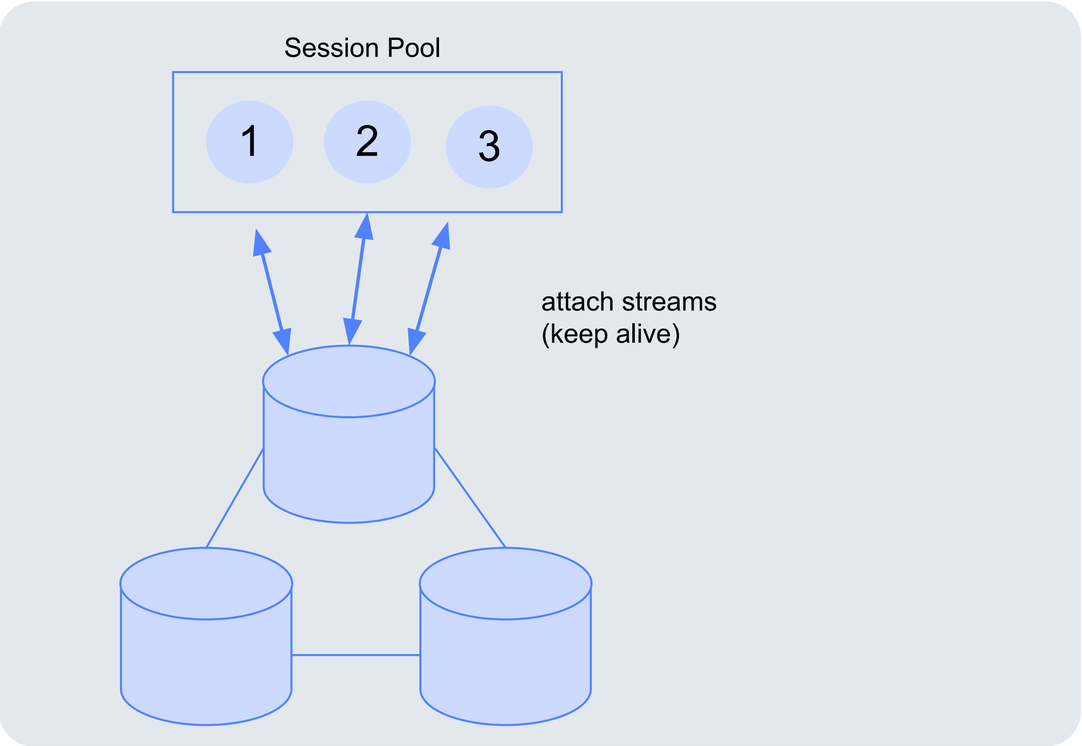


Хочешь исполнить запрос - держи сессию!

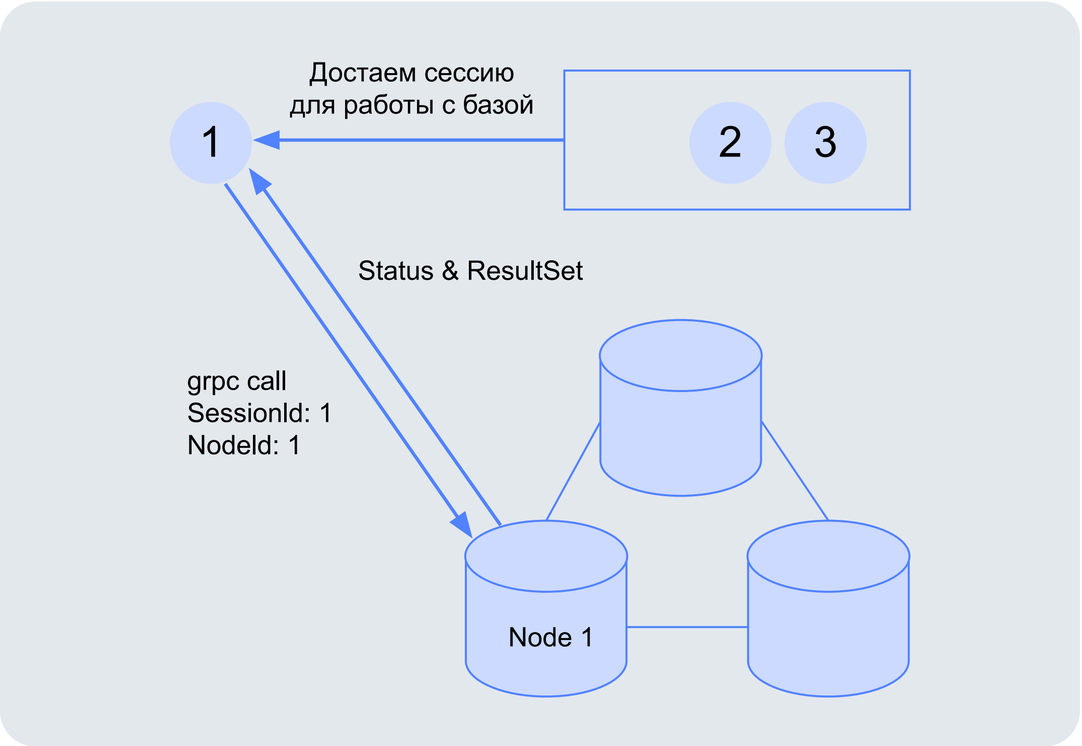


SessionPool

Пул поддерживает время жизни сессии.



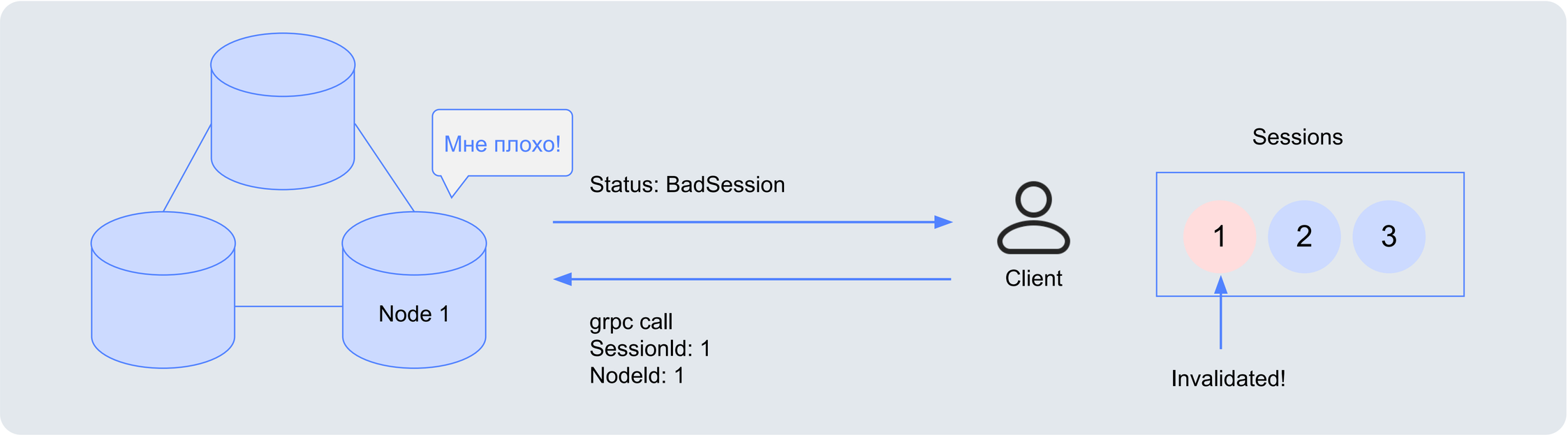
В старом API: когда сессия вне пула, её время жизни продлевается клиентскими операциями.



Обработка ошибок

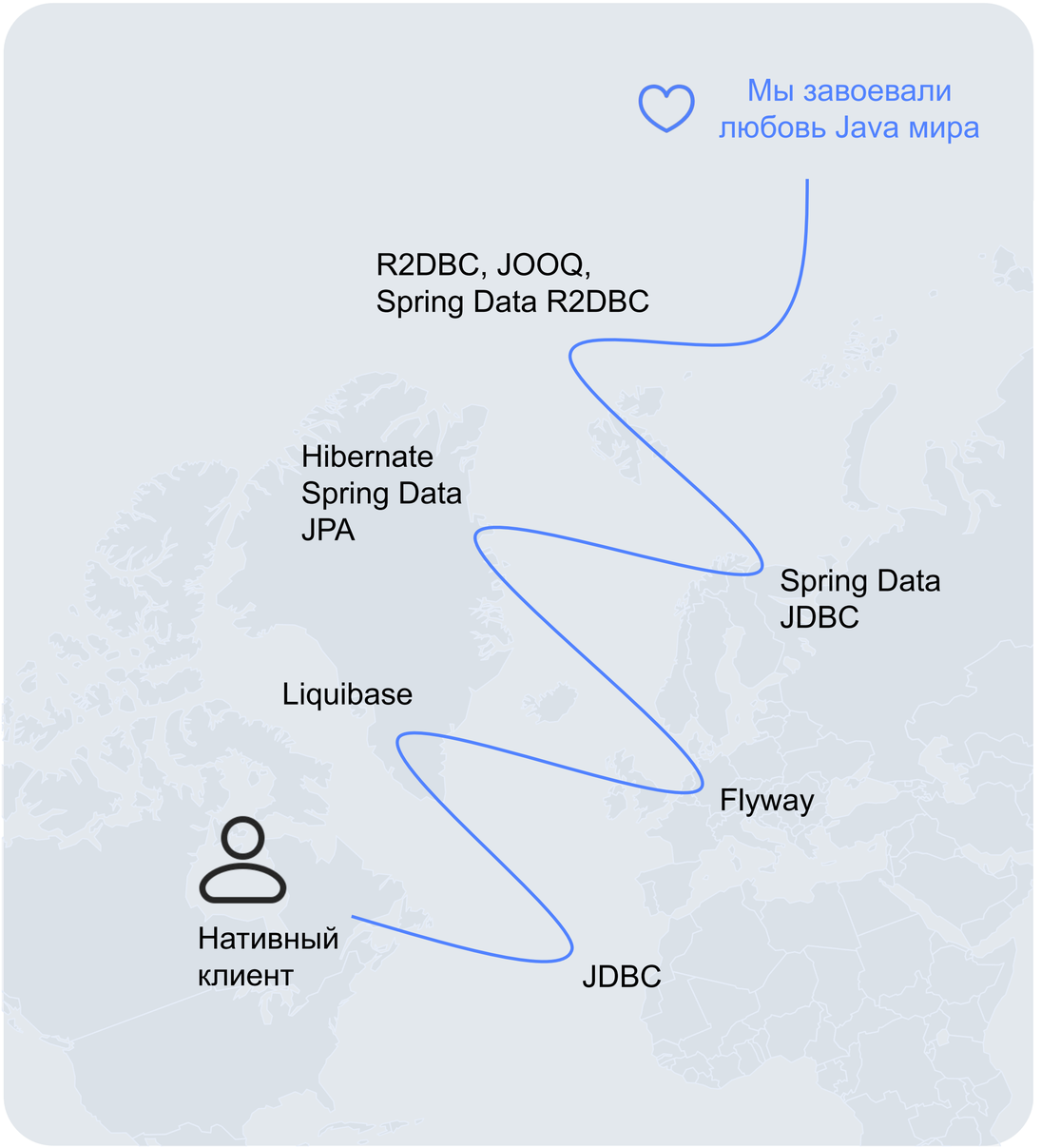
Ошибки делятся на два основных типа:

1. Серверные
2. Транспортные



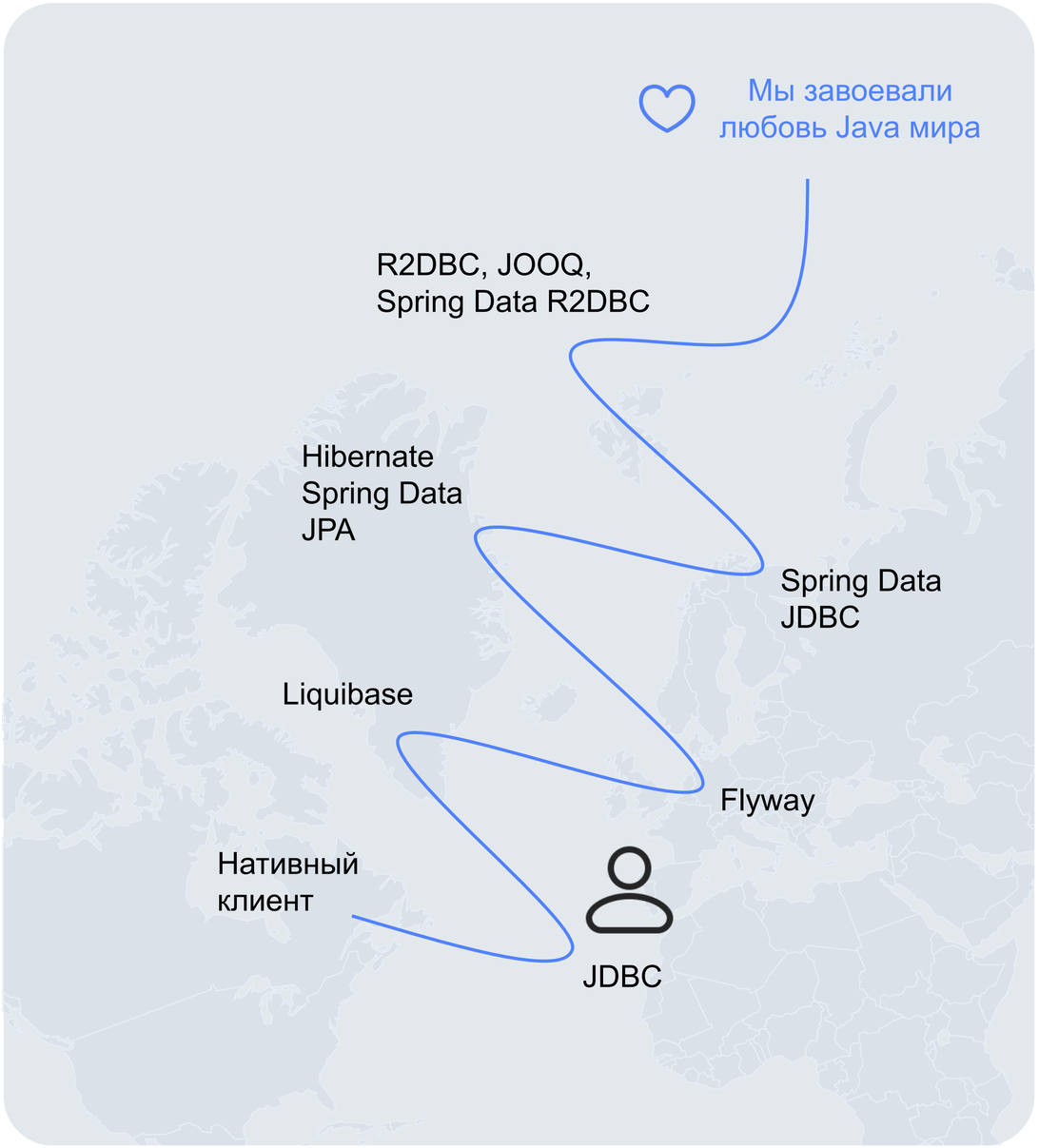
Нативный клиент итоги

1. Точка отсчета для взаимодействия с базой данных
2. Способ не самый удобный



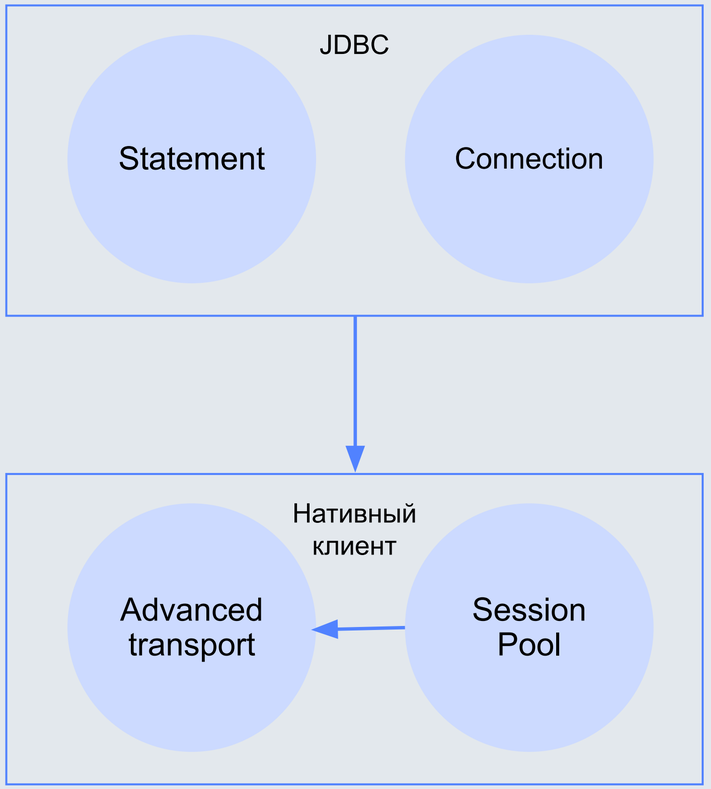
Введение JDBC

JDBC — это стандарт взаимодействия Java-приложений с различными СУБД.



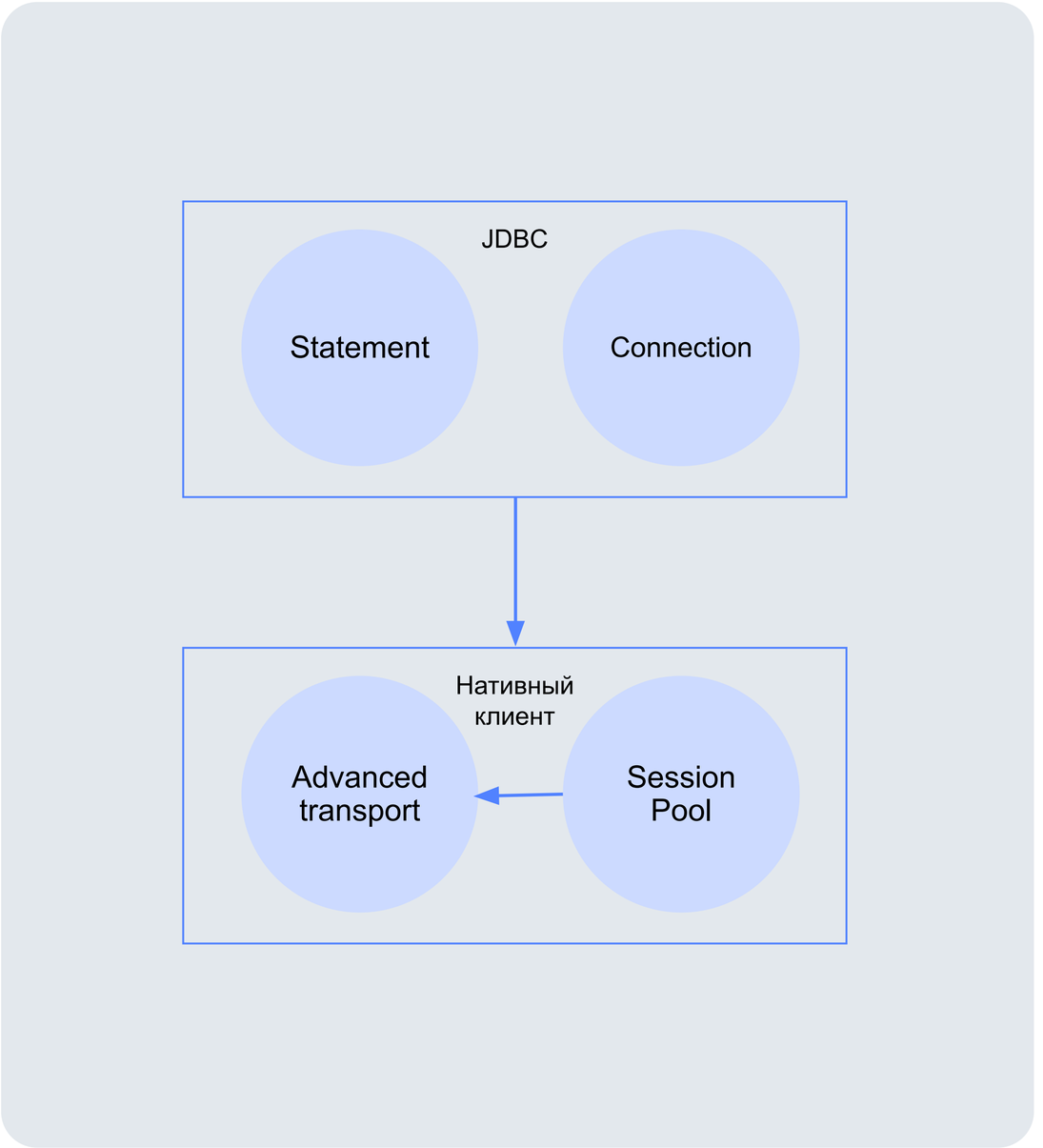
Реализация JDBC

YDB реализует JDBC, делегируя интерфейсы JDBC существующему нативному клиенту и синхронно вызывая его API (gRPC асинхронный транспорт), аналогично подходу, используемому Google Spanner.



Реализация JDBC

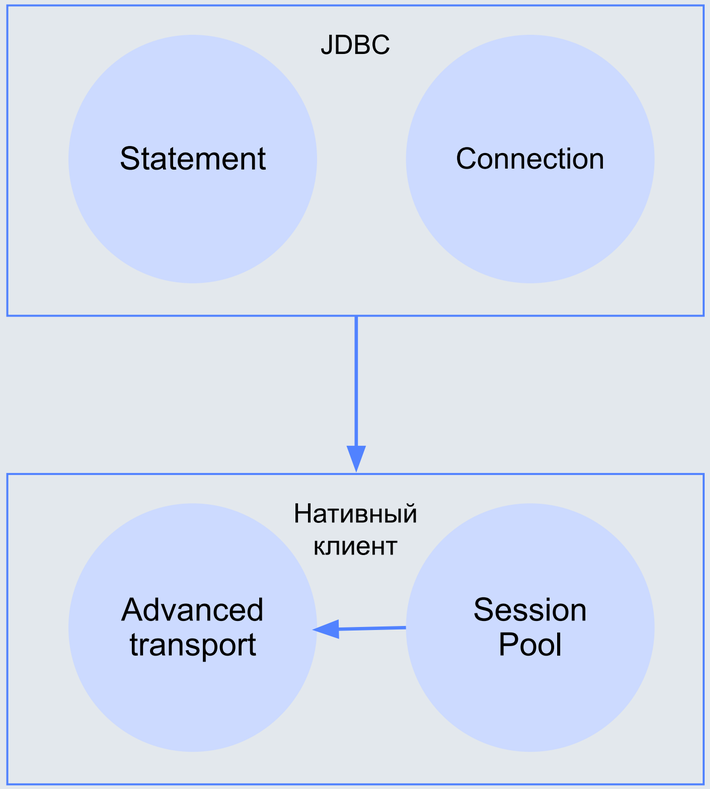
Основные аспекты, которые стоит учесть:



Реализация JDBC

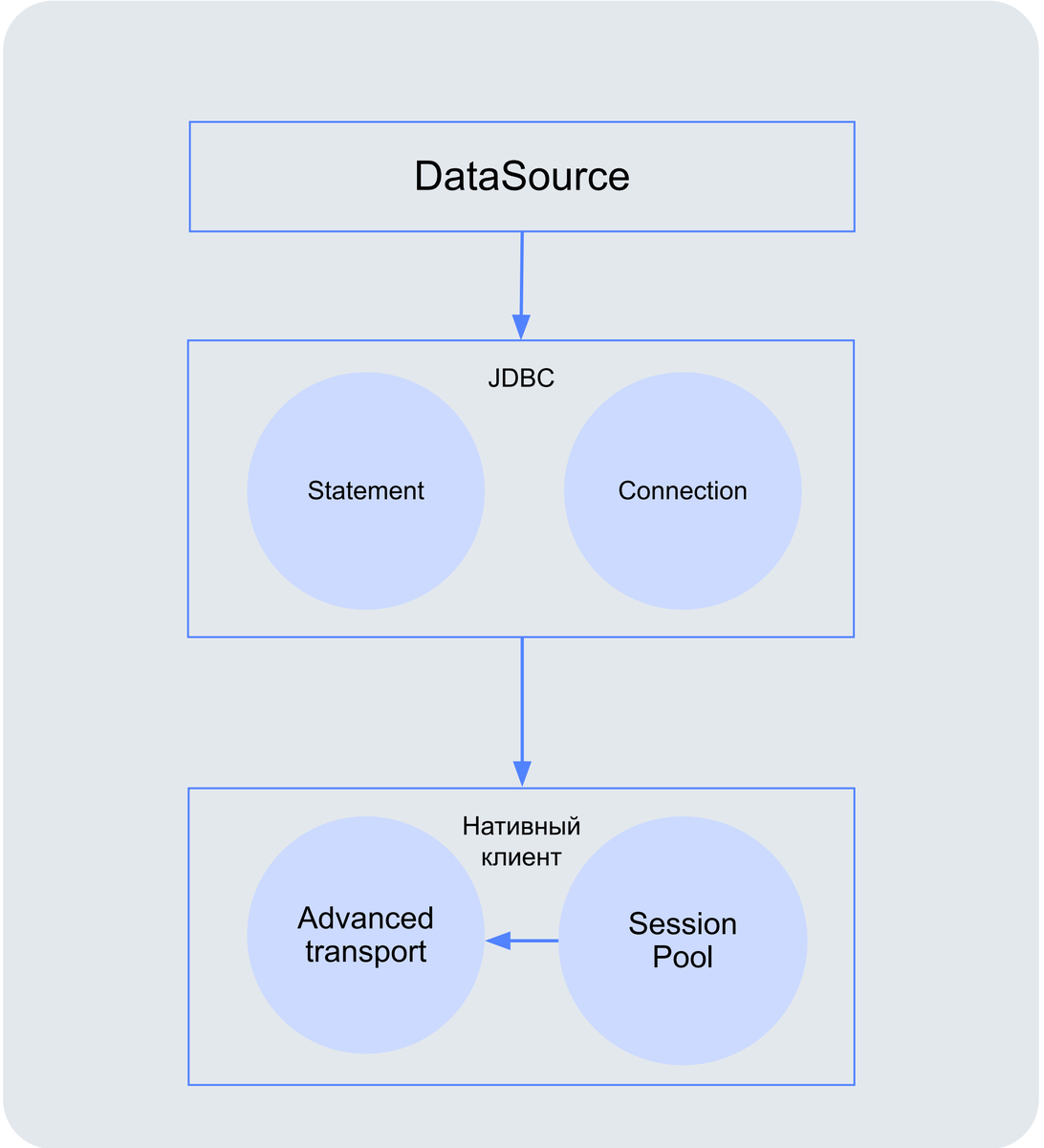
Основные аспекты, которые стоит учесть:

- JDBC используется вместе с внешним `javax.sql.DataSource`



DataSource

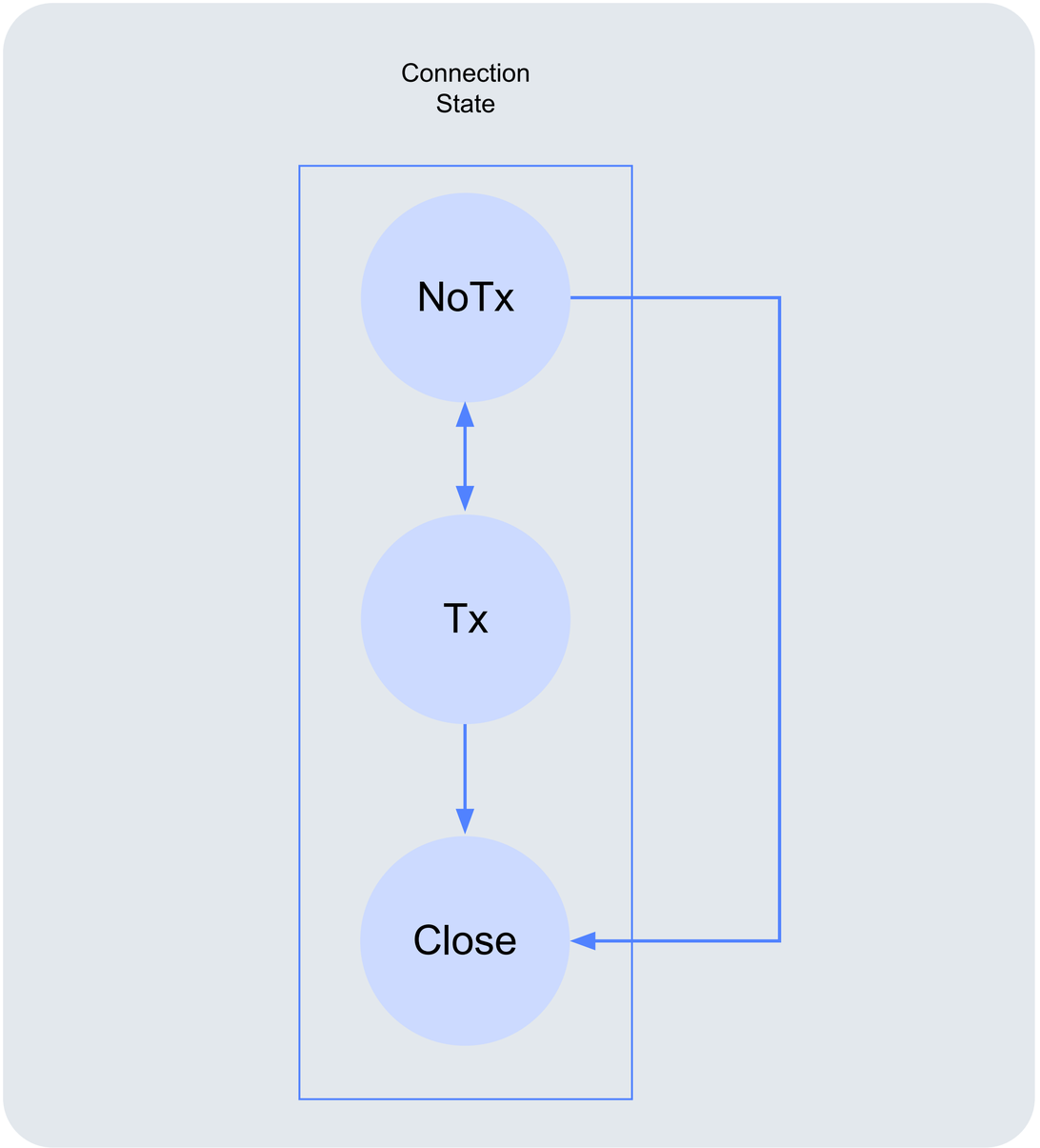
- Разные размеры пулов
- Время жизни сессии значительно короче
- Table Service API



Реализация Connection

Рассмотрим основной примитив JDBC `java.sql.Connection`:

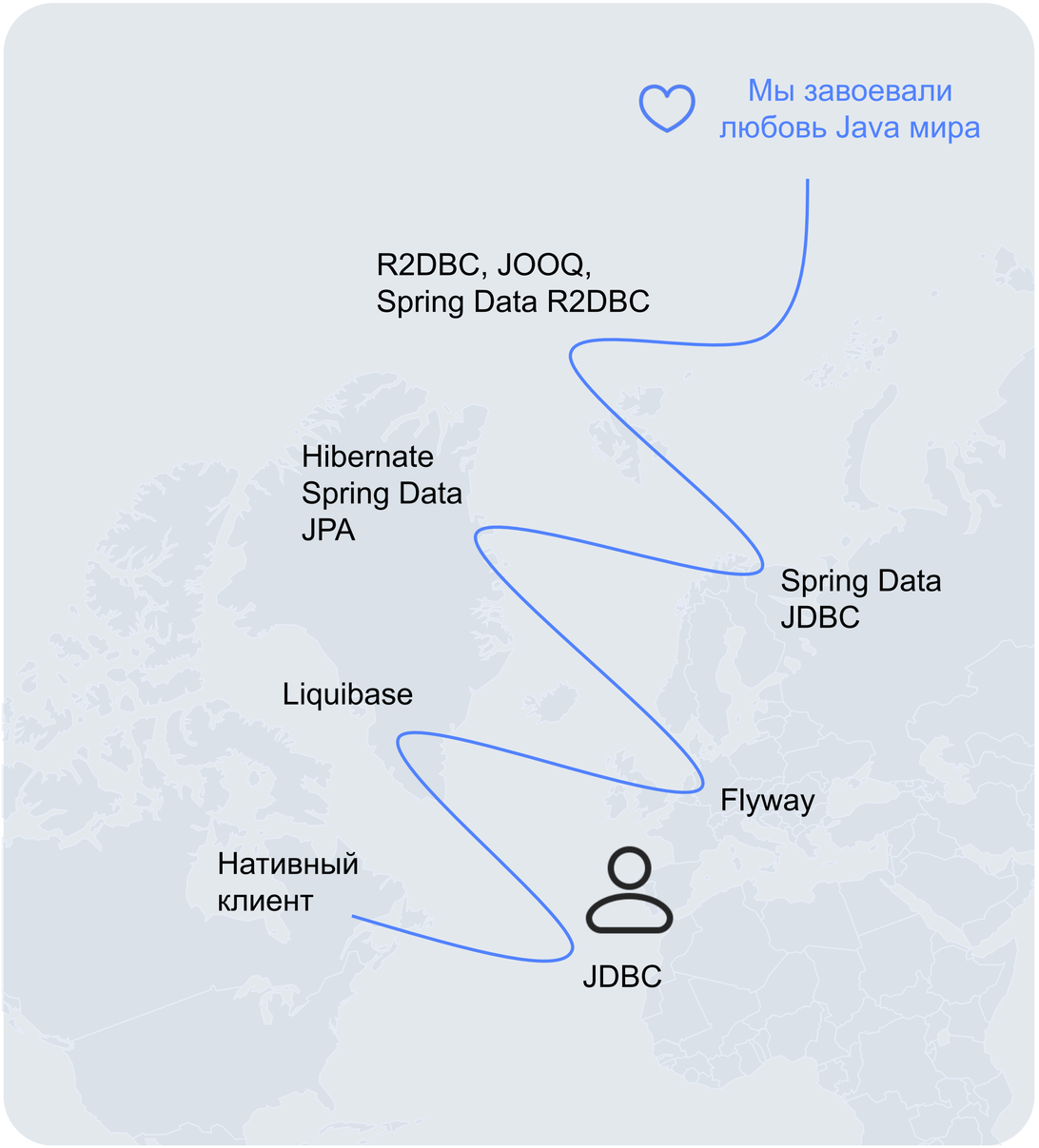
- **NoTx:** Для каждого используется сессия из SessionPool
- **Tx:** Удерживание сессии и сохранение метаданных транзакции
- **Close:** Соединение закрыто



Можно ли было прибивать сессию к Connection?

Основная причина виртуальных коннекшенов — `Table Service API`

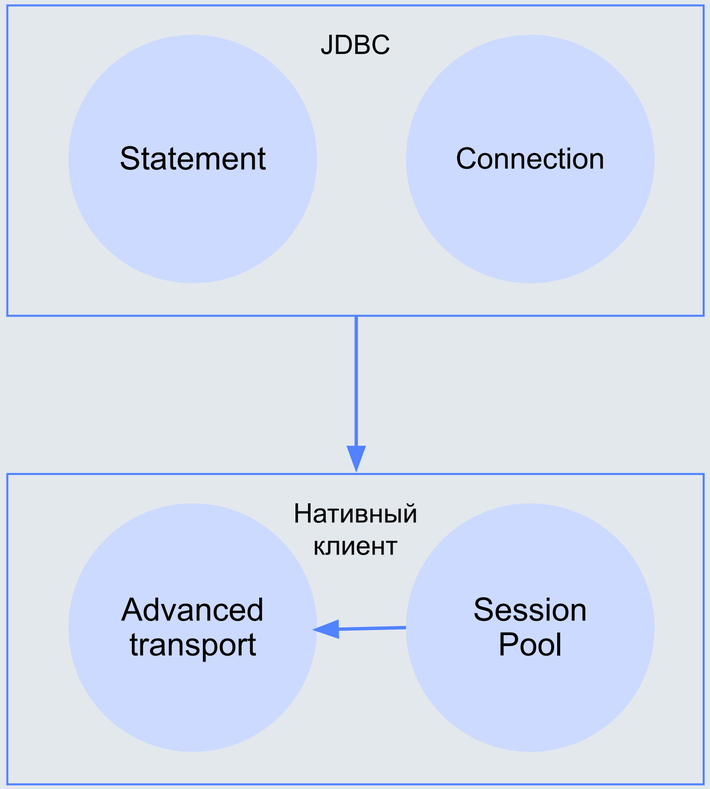
- Инвалидация коннекшена происходила бы через метод `isValid()` (точно бы происходила?)
- Сессии могут инвалидироваться сервером в самом пуле
- Зависимость от внешних имплементаций



Реализация JDBC

Основные аспекты, которые стоит учесть:

- JDBC используется вместе с внешним `javax.sql.DataSource`
- `rollback()` с оптимистическими блокировками



connection.rollback()

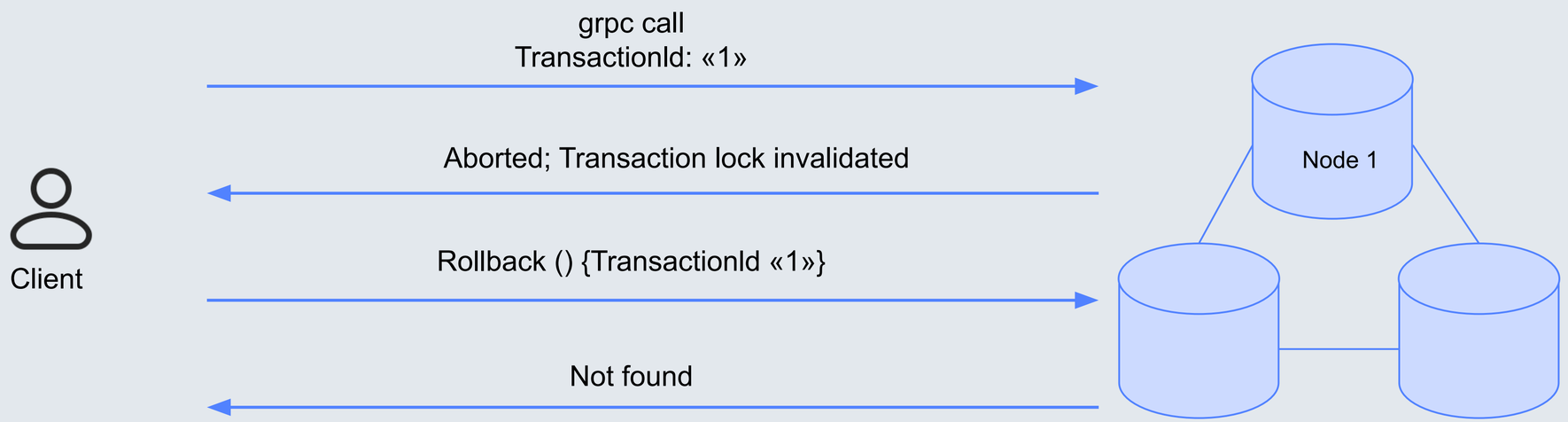
```
String updateString = "update COFFEES set SALES = ? where COF_NAME = ?";
String updateStatement = "update COFFEES set TOTAL = TOTAL + ? where COF_NAME = ?";

try (PreparedStatement updateSales = con.prepareStatement(updateString);
     PreparedStatement updateTotal =con.prepareStatement(updateStatement)) {
   con.setAutoCommit(false);
   for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
       //...
       updateSales.executeUpdate();
       updateTotal.executeUpdate();
       con.commit();
   }
} catch (SQLException e) {
   if (con != null) {
       try {
           System.err.print("Transaction is being rolled back");
           con.rollback();
       } catch (SQLException excep) {
           //...
       }
   }
}
```

connection.rollback()

```
catch (SQLException e) {
   if (con ≠ null) {
       try {
           System.err.print("Transaction is being rolled back");
           con.rollback();
       } catch (SQLException excep) {
           //...
       }
   }
}
```

А что не так с rollback()?

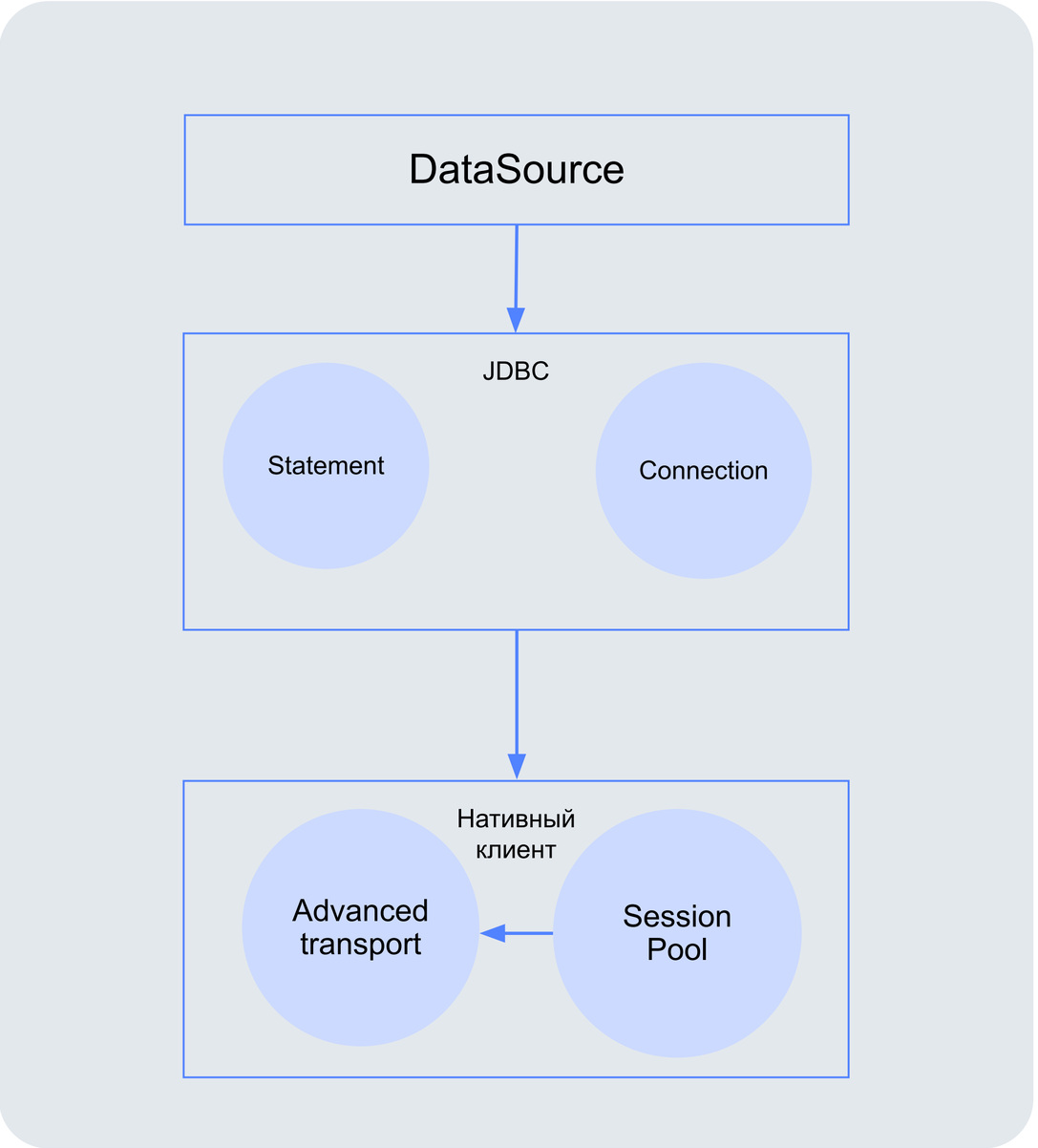
Сервер сам откатит транзакцию в случае инвалидации локов.
Дополнительный rollback ни к чему не приведет!



Реализация JDBC

Основные аспекты, которые стоит учесть:

- JDBC используется вместе с внешним `javax.sql.DataSource`
- `rollback()` с оптимистическими блокировками
- Объявление переменных: `VALUES (?,?,?)`



Parser

```
SELECT sa.title AS season_title, sr.title AS series_title
FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id
WHERE sa.series_id = ? AND sa.season_id = ?;
```

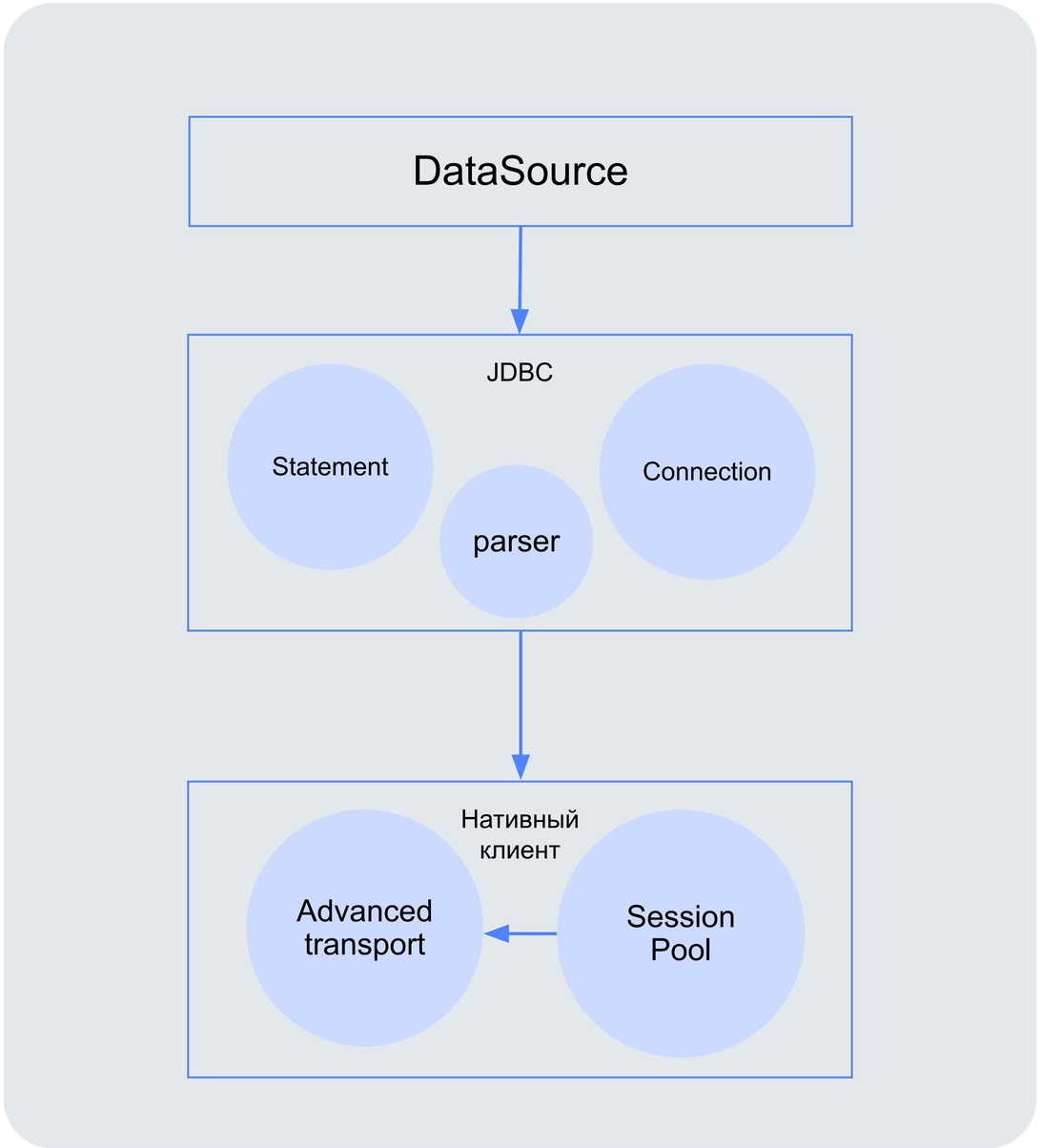
```
DECLARE $seriesId AS Uint64;
DECLARE $seasonId AS Uint64;
SELECT sa.title AS season_title, sr.title AS series_title
FROM seasons AS sa INNER JOIN series AS sr ON sa.series_id = sr.series_id
WHERE sa.series_id = $seriesId AND sa.season_id = $seasonId;
```

Реализация JDBC

Основные аспекты, которые стоит учесть:

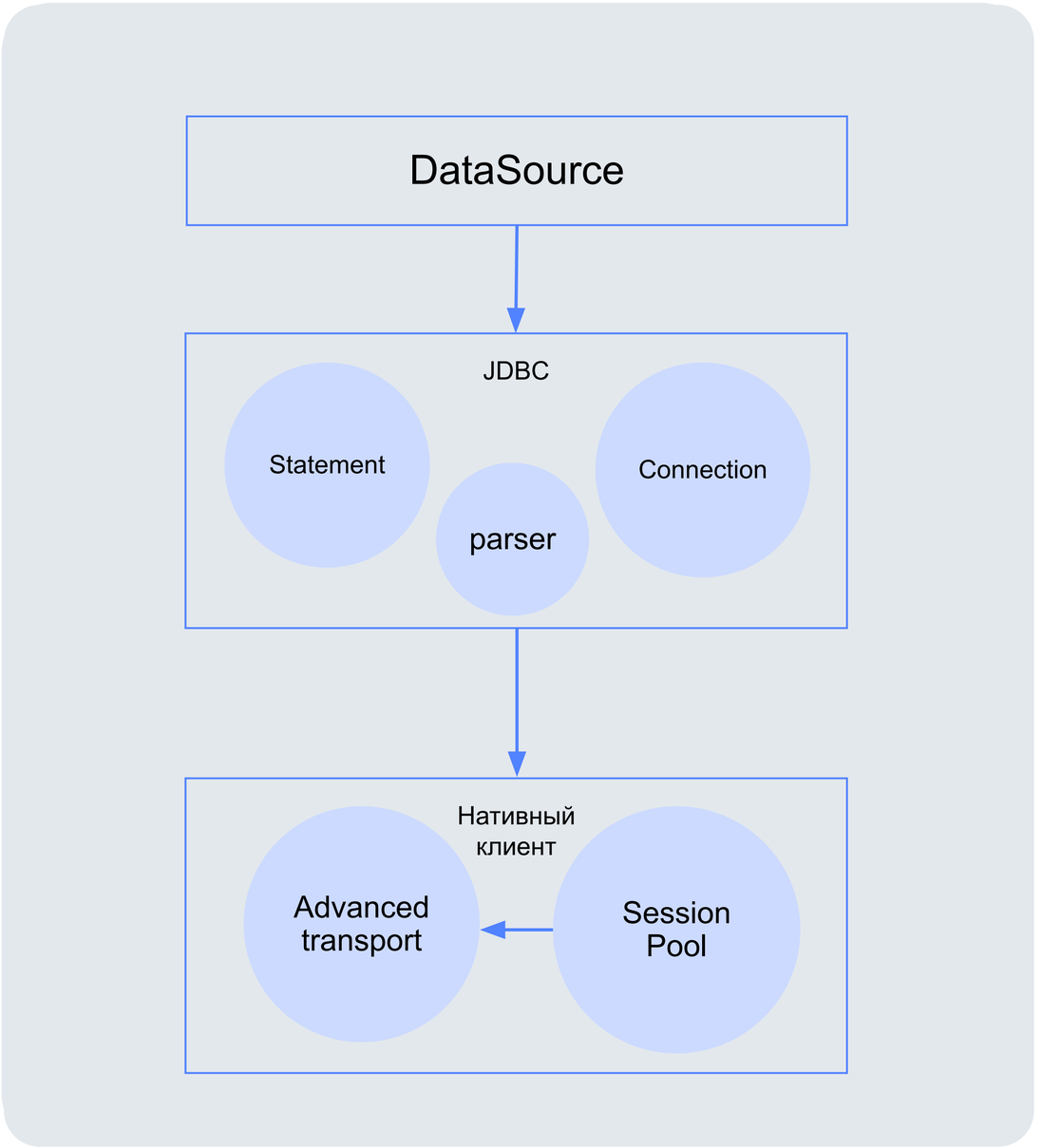
- JDBC используется вместе с внешним `javax.sql.DataSource`
- `rollback()` с оптимистическими блокировками
- Объявление переменных: `VALUES (?,?,?)`
- Подготовка batch запросов
- Строго типизированные параметры

… и много всяческой технической работы



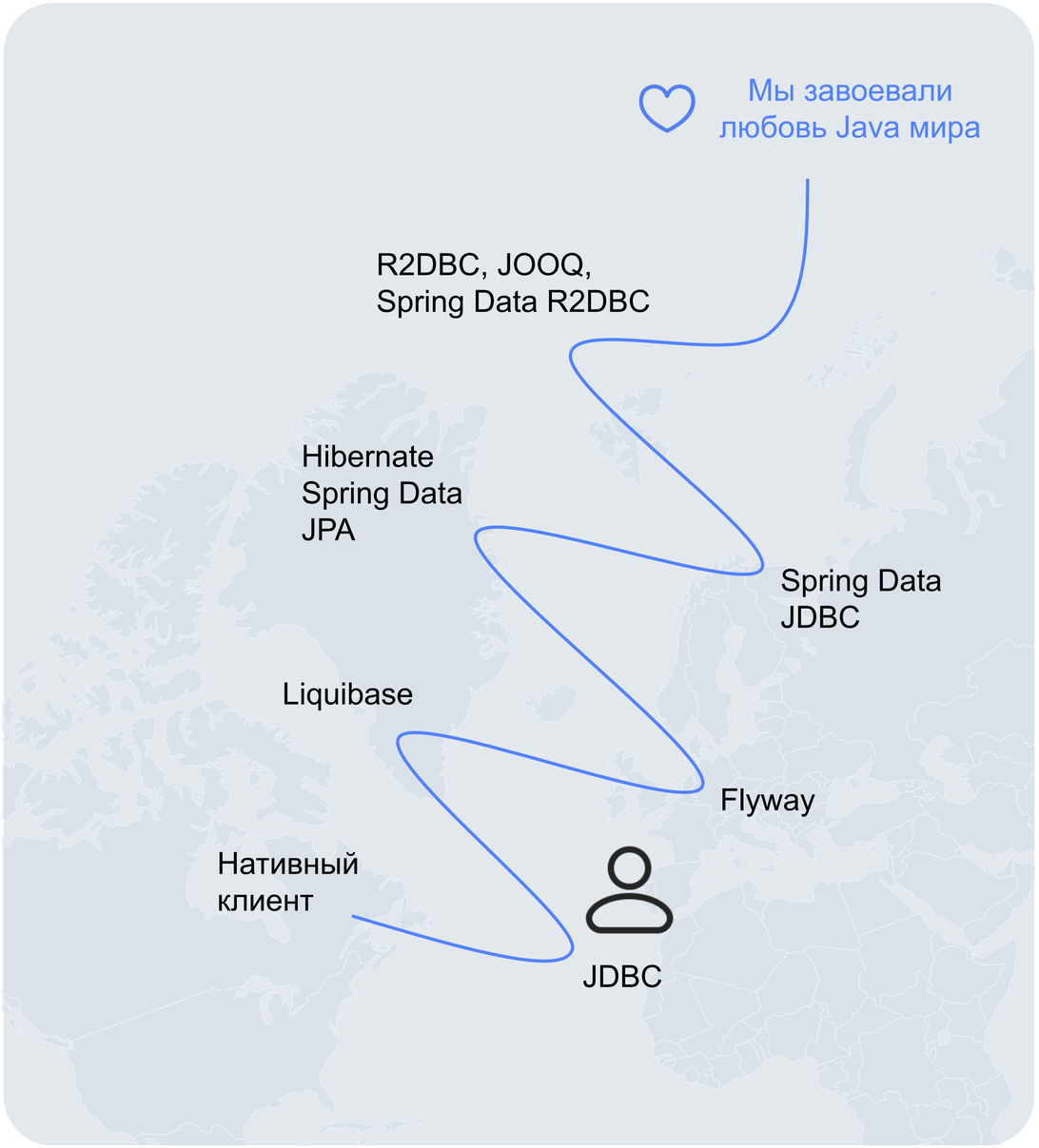
Основные ограничения

- `executeUpdate()`
- `setNull(int index)`
- `setSavepoint()`
- Отсутствие типа `Time`



JDBC итоги

1. Мы двигаем человечка
2. DataGrip
3. DBeaver
4. `SQLRecoverableException - важно!`
5. Переходим дальше!



Liquibase

- Безопасное управление миграциями
- Различные форматы схем: .sql, .xml, .json, .yaml
- Загрузка данных из .CSV

Liquibase: распределенная блокировка

```
| ID | LOCKED | LOCKEDBY | LOCKGRANTED |
|:---|:-------|:---------|:------------|
| 1  | false  | null     | null        |
```

```sql
-- acquire lock
SELECT LOCKED FROM DATABASECHANGELOGLOCK WHERE ID = 1;
```

Liquibase: распределенная блокировка

```
| ID | LOCKED | LOCKEDBY | LOCKGRANTED |
|:---|:-------|:---------|:------------|
| 1  | false  | null     | null        |
```

```sql
-- acquire lock
SELECT LOCKED FROM DATABASECHANGELOGLOCK WHERE ID = 1;
-- Если false, делаем запрос на взятие блокировки
UPDATE DATABASECHANGELOGLOCK SET LOCKED = true /* LOCKEDBY & LOCKGRANTED */ WHERE ID = 1;
-- Если UPDATE вернул 0, то блокировку взять не удалось, повторим взятие блокировки через 1 секунду.
```

Liquibase: распределенная блокировка

```
| ID | LOCKED | LOCKEDBY | LOCKGRANTED |
|:---|:-------|:---------|:------------|
| 1  | false  | null     | null        |
```

```sql
-- acquire lock
SELECT LOCKED FROM DATABASECHANGELOGLOCK WHERE ID = 1;
-- Если false, делаем запрос на взятие блокировки
UPDATE DATABASECHANGELOGLOCK SET LOCKED = true /* LOCKEDBY & LOCKGRANTED */ WHERE ID = 1;
-- Если UPDATE вернул 0, то блокировку взять не удалось, повторим взятие блокировки через 1 секунду.
-- run migrations..
```

Liquibase: распределенная блокировка

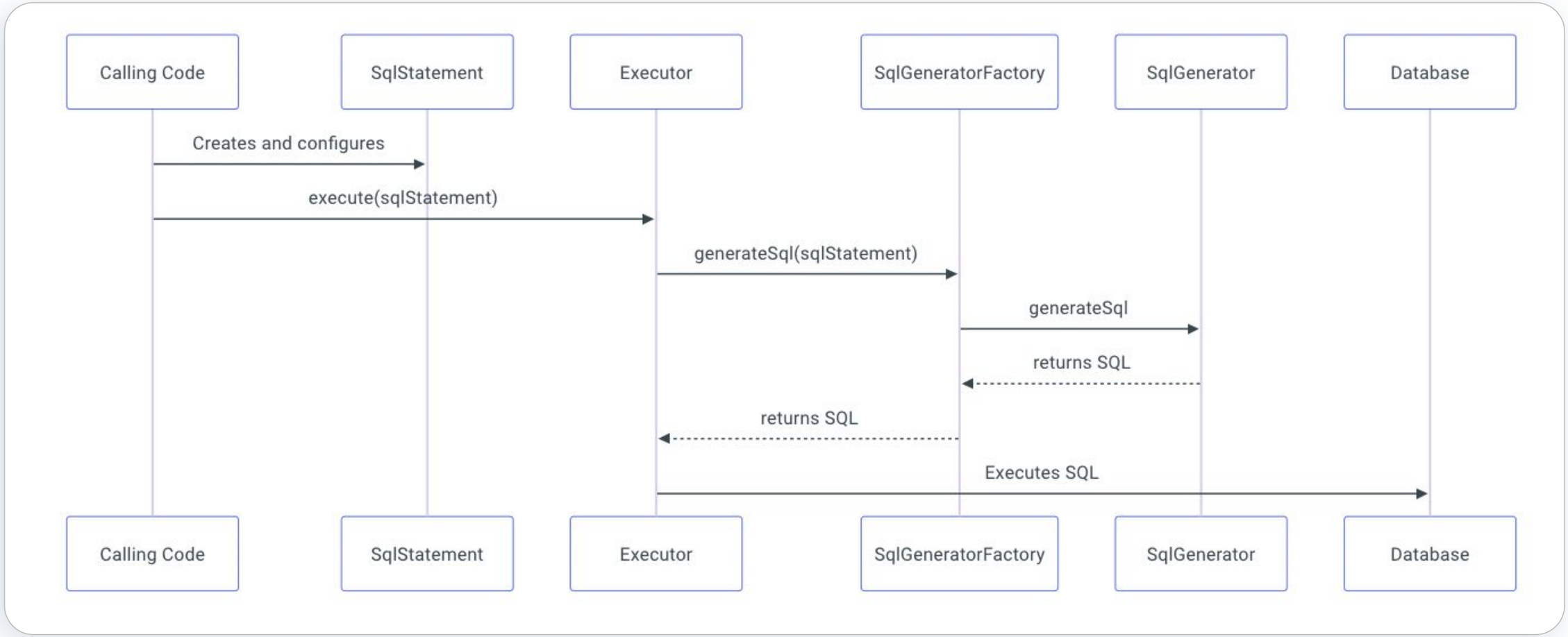
```
| ID | LOCKED | LOCKEDBY | LOCKGRANTED |
|:---|:-------|:---------|:------------|
| 1  | false  | null     | null        |
```

```sql
-- acquire lock
SELECT LOCKED FROM DATABASECHANGELOGLOCK WHERE ID = 1;
-- Если false, делаем запрос на взятие блокировки
UPDATE DATABASECHANGELOGLOCK SET LOCKED = true /* LOCKEDBY & LOCKGRANTED */ WHERE ID = 1;
-- Если UPDATE вернул 0, то блокировку взять не удалось, повторим взятие блокировки через 1 секунду.
-- run migrations..
-- release lock
UPDATE DATABASECHANGELOGLOCK SET LOCKED = false WHERE ID = 1;
```

Liquibase: распределенная блокировка для YDB

```
-- acquire lock начало транзакции

-- SERIALIZABLE

SELECT LOCKED FROM DATABASECHANGELOGLOCK WHERE ID = 1;

-- Если false, делаем запрос на взятие блокировки

UPDATE DATABASECHANGELOGLOCK SET LOCKED = true /* LOCKEDBY & LOCKGRANTED */ WHERE ID = 1;

COMMIT;

-- Transaction lock invalidated - означет, что блокировку не удалось взять, кто - то нас обогнал
```

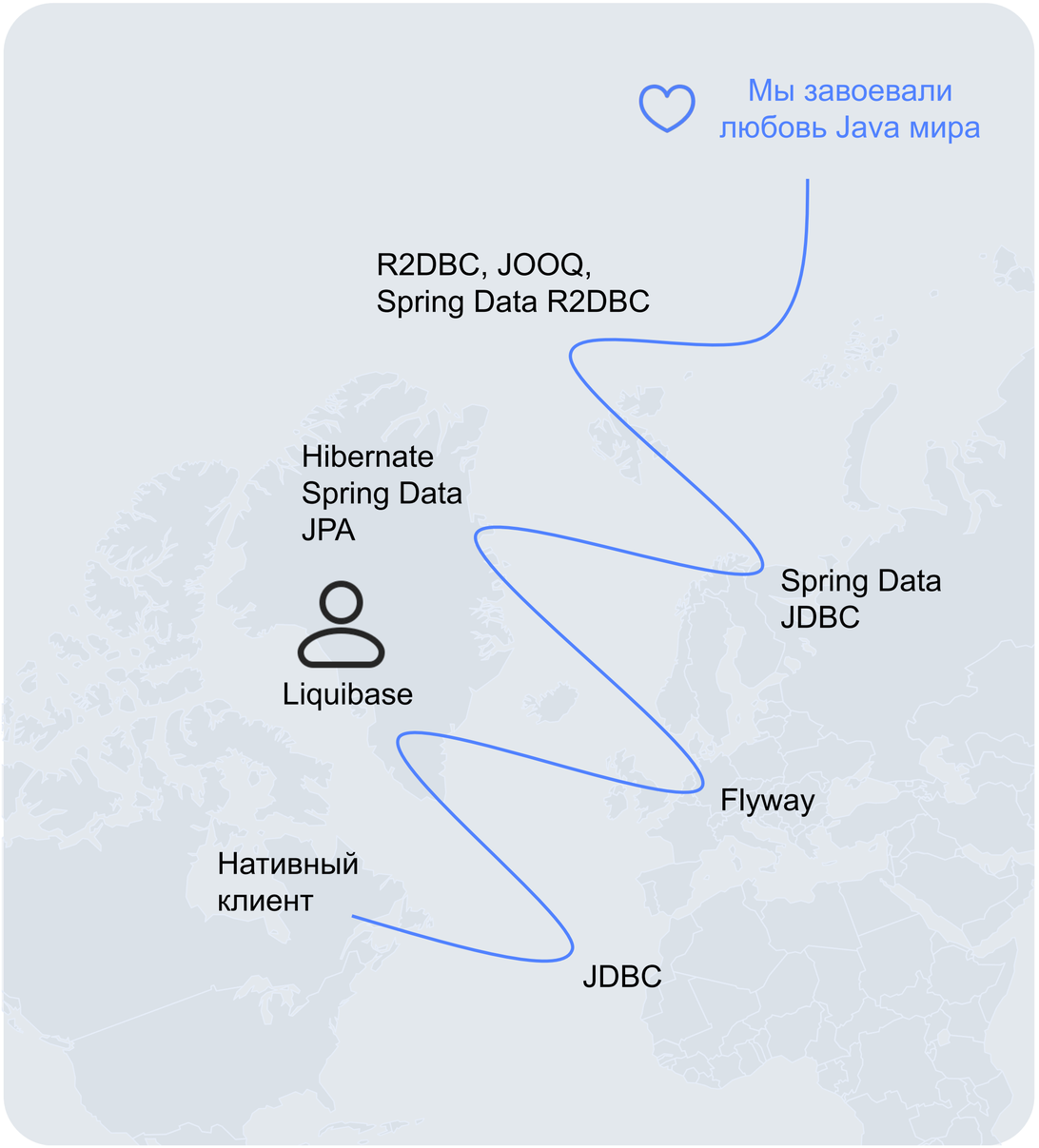
Оставшиеся пункты по Liquibase



https://contribute.liquibase.com/code/api/sqlgenerator-sqlgenerator/

Liquibase итоги

1. Двигаем человечка дальше
2. Можем смело использовать расширение в Spring Boot
3. Либо подкладывая .jar расширять утилиту liquibase
4. Двигаемся дальше!



Flyway

- Безопасное управление миграциями ..
- и все

Flyway: распределенная блокировка

- Pg: `SELECT * FROM lock FOR UPDATE`
- Oracle: `LOCK TABLE lock IN EXCLUSIVE MODE`
- CockroachDB, Google Spanner: `INSERT INTO c expired_time`

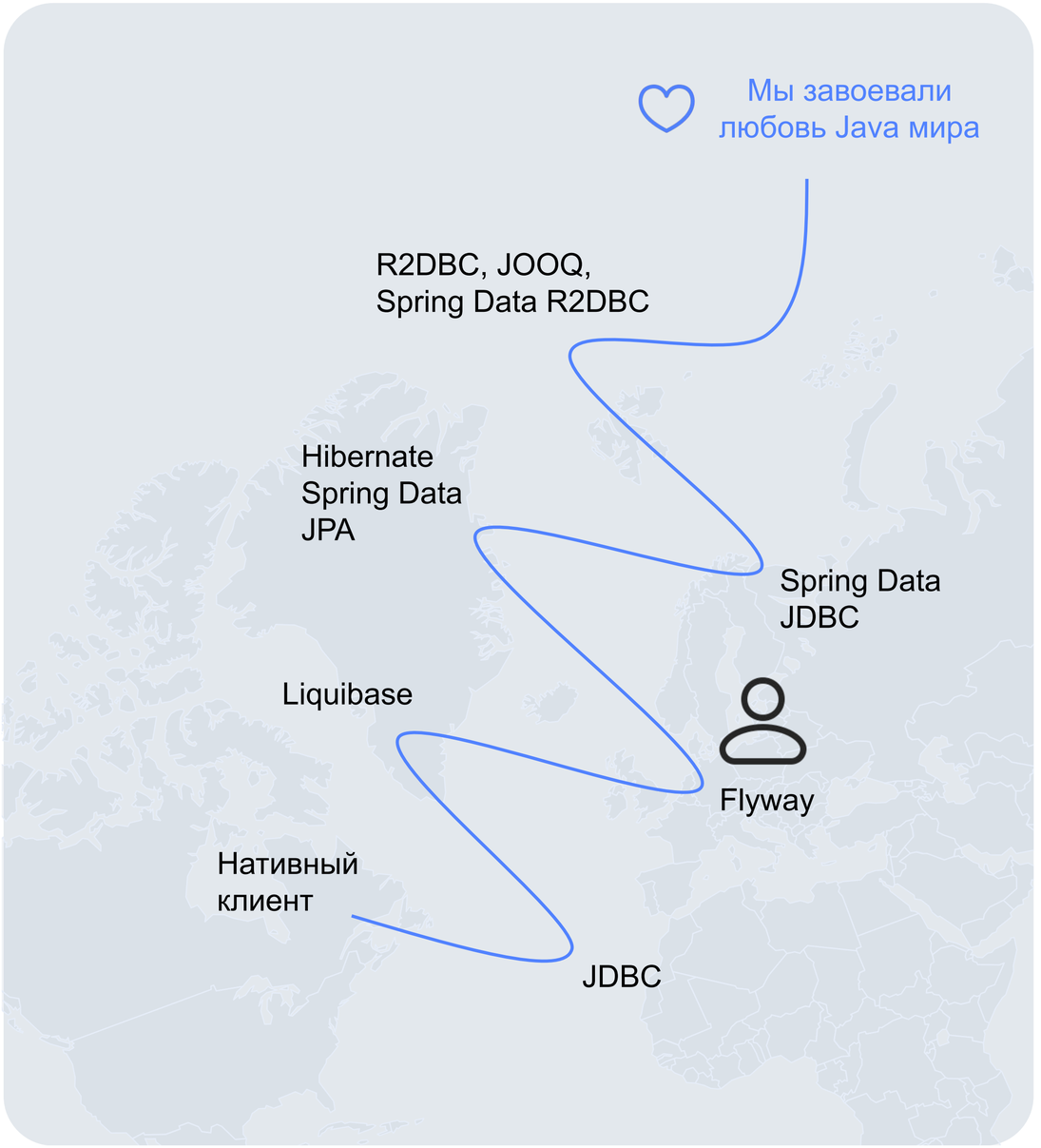
Flyway: распределенная блокировка для YDB

```
-- клиент пытается вставить фиктивную запись с ID = -100
INSERT INTO flyway_schema_history(installed_rank, version, description) VALUES (-100, ?, 'flyway-lock');
-- кто успел вставить такую запись, тот и лидер
-- отпускается блокировку
DELETE FROM flyway_schema_history WHERE installed_rank = -100
```

Важным отличием от CockroachDB и Google Spanner является то, что мы не проставляем тайм-аут истечения блокировки, а приостанавливаем процесс для расследования DBA, как сделано в Apache Ignite.

Flyway итоги

1. Двигаем человечка
2. Spring Boot с Flyway работает в штатном режиме
3. Утилиту Flyway можно расширить .jar диалекта



JPA (Hibernate)

- Генерация SQL: Hibernate отвечает за создание валидного SQL из объектно-ориентированного кода
- Маппинг сущностей
- Свой язык запросов (JPQL / HQL)
- Переносимость с одной СУБД на другую

Переносимость и диалекты

Диалект — инструмент, который помогает Hibernate генерировать SQL, соответствующий спецификациям различных СУБД, оставаясь переносимым при переключении технологии.

- hibernate-core основные диалекты
- hibernate-community-dialects
- Отдельные проекты, например YDB или Google Spanner

Диалект

Диалект не зависит напрямую от JDBC драйвера.

- Диалект определяется по URL к базе jdbc:{name}:.. (switch по name в hibernate-core)
- Вы можете явно указать диалект, если необходимо

Создавая новый диалект:

- Вы наследуютесь от существующего (CockroachDB наследуется от PostgreSQLDialect)
- Вы создаете новый от базового класса `org.hibernate.dialect.Dialect`
- Используете чужой (простой CRUD для новой условно СУБД)

YDB диалект

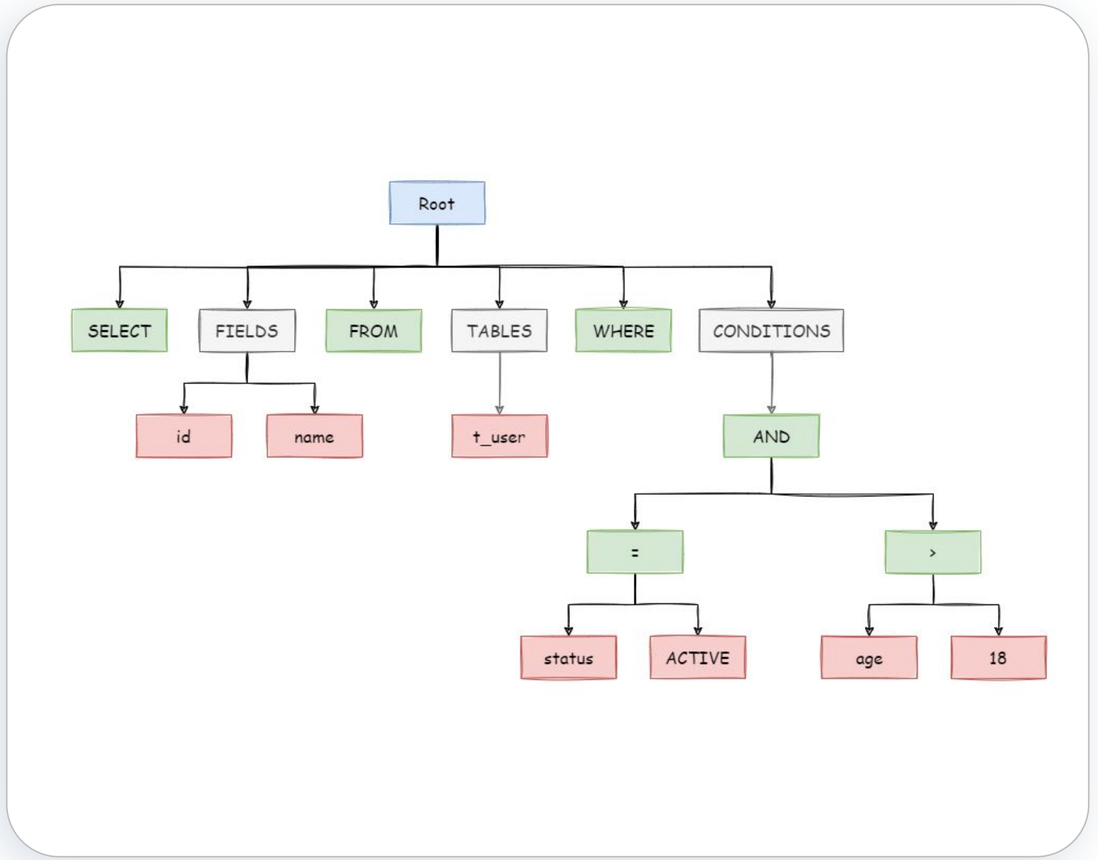
YDB обзавелась собственным полноценным диалектом, который наследуется от `org.hibernate.dialect.Dialect`. Примечательно, что таких диалектов несколько, учитывая версии Hibernate:

- Диалект для Hibernate 5
- Диалект для Hibernate 6
- Диалект (возможно) для будущих версий Hibernate 7, 8 и далее

```
spring.jpa.properties.hibernate.dialect=tech.ydb.hibernate.dialect.YdbDialect
spring.datasource.driver-class-name=tech.ydb.jdbc.YdbDriver
spring.datasource.url=jdbc:ydb:grpc://localhost:2136/local
```

Формирование SQL запроса к СУБД

Основная задача диалекта обход AST
при формировании SQL-запроса к базе данных.



Формирование SQL запроса к СУБД

Основная задача диалекта обход AST
при формировании SQL-запроса к базе данных:

- Формирование пагинации: `LIMIT ? OFFSET ?` вместо `OFFSET ? ROWS FETCH FIRST ? ROWS ONLY`

```
@Override
protected void renderOffsetFetchClause(
// params
) {
    if (fetchExpression ≠ null) {
        appendSql(" limit ");
        fetchExpression.accept(this);
    }

    if (offsetExpression ≠ null) {
        appendSql(" offset ");
        offsetExpression.accept(this);
    }
}
```

Формирование SQL запроса к СУБД

Основная задача диалекта обход AST
при формировании SQL-запроса к базе данных:

- Формирование пагинации: `LIMIT ? OFFSET ?` вместо `OFFSET ? ROWS FETCH FIRST ? ROWS ONLY`
- Оборачивать синтаксически сложные имена: `FROM `a-b`` вместо `FROM “a-b”`
- Формирование значение bool в запросе: `true, false` вместо 1 и 0
- `current_timestamp`: ряд функций для получения времени
- Фильтр по шаблону `LIKE / ILIKE .. ESCAPE ?`

```java
@Override
public char openQuote() {
    return '`';
}

@Override
public char closeQuote() {
    return '`';
}
@Override
public boolean supportsCaseInsensitiveLike() {
    return true;
}

@Override
public String getCaseInsensitiveLike() {
    return "ilike";
}
```

Дополнительные возможности диалекта

- Генерация схемы базы данных

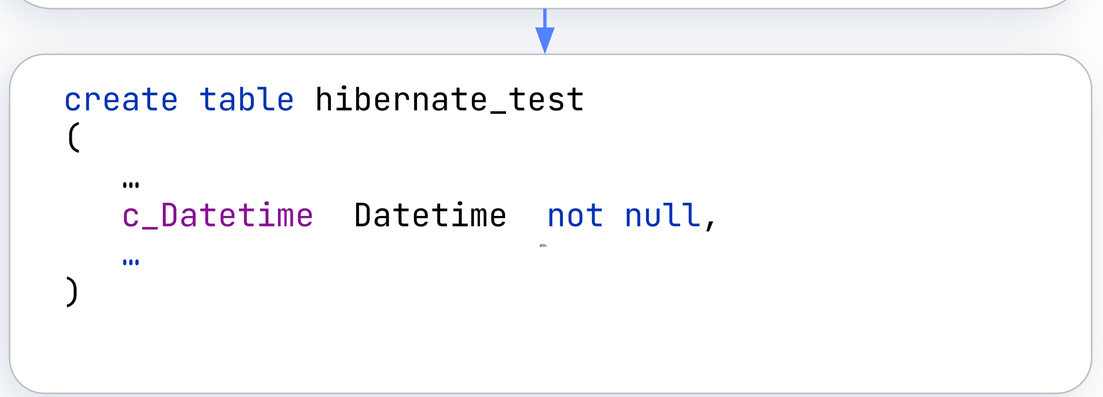
```
@Getter
@Setter
@Entity
@Table(name = "Groups")
public class Group {
    @Id
    @Column(name = "GroupId")
    private int id;
    @Column(name = "GroupName")
    private String name;
    @OneToMany(mappedBy = "group")
    private List<Student> students;
}
```

```
CREATE TABLE Groups (
    GroupId Int32 NOT NULL,
    GroupName Text,
    PRIMARY KEY (GroupId)
)
```

Дополнительные возможности диалекта

- Генерация схемы базы данных
- Регистрация новых типов, например `Datetime`

```java
@Entity
@Data
@Table(name = "hibernate_test")
public class TestEntity {
    @Id
    private Integer id;
    @Column(name = "c_Date", nullable = false)
    private LocalDate date;
    @Column(name = "c_Datetime", nullable = false)
    private LocalDateTime datetime;
    @Column(name = "c_Timestamp", nullable = false)
    private Instant timestamp;
}
```



Дополнительные возможности диалекта

- Генерация схемы базы данных
- Регистрация новых типов, например `Datetime`
- Добавлять hint'ы для запросов. Примеры (`USE INDEX` в MySQL, `VIEW` в YDB)
- Добавлять и переопределять Binder и Extractor для тех или иных типов данных

```
SELECT
    series_id,
    title,
    info,
    release_date,
    views,
    uploaded_user_id
FROM series VIEW views_index
WHERE views ⩾ someValue
```

Есть ли проблемы?

Синтаксические

В YQL можно писать следующего вида запрос:

```
SELECT A … FROM TABLE GROUP BY func(column) AS A
```

Вместо

```
SELECT func(column)… FROM TABLE GROUP BY func(column)
```

Группировка и проекция по результату функции становится не выразима.

Синтаксические

```
SELECT *
    FROM TableName VIEW IndexName
    WHERE …
```

Вставка VIEW, работает уже поверх сгенерированного SQL. А не обходе синтаксического дерева.

Регулярка для выражения `USE INDEX в` MySQL:

```
Pattern.compile( "^\\s*(select\\b.+?\\bfrom\\b.+?)(\\bwhere\\b.+?)$" )
```

Серверные

Обращение к колонкам вне проекции:

- ORDER BY
- GROUP BY

```
SELECT
    sa.title AS season_title,
    sr.title AS series_title
FROM seasons AS sa
        INNER JOIN series AS sr
ON sa.series_id = sr.series_id
WHERE sa.series_id = 1
ORDER BY sr.series_id, sa.season_id;
```

```
Can't get value: Status: GenericError, Issues:

Error: Member not found: sr.series_id
Error: Member not found: sa.season_id
```

Серверные

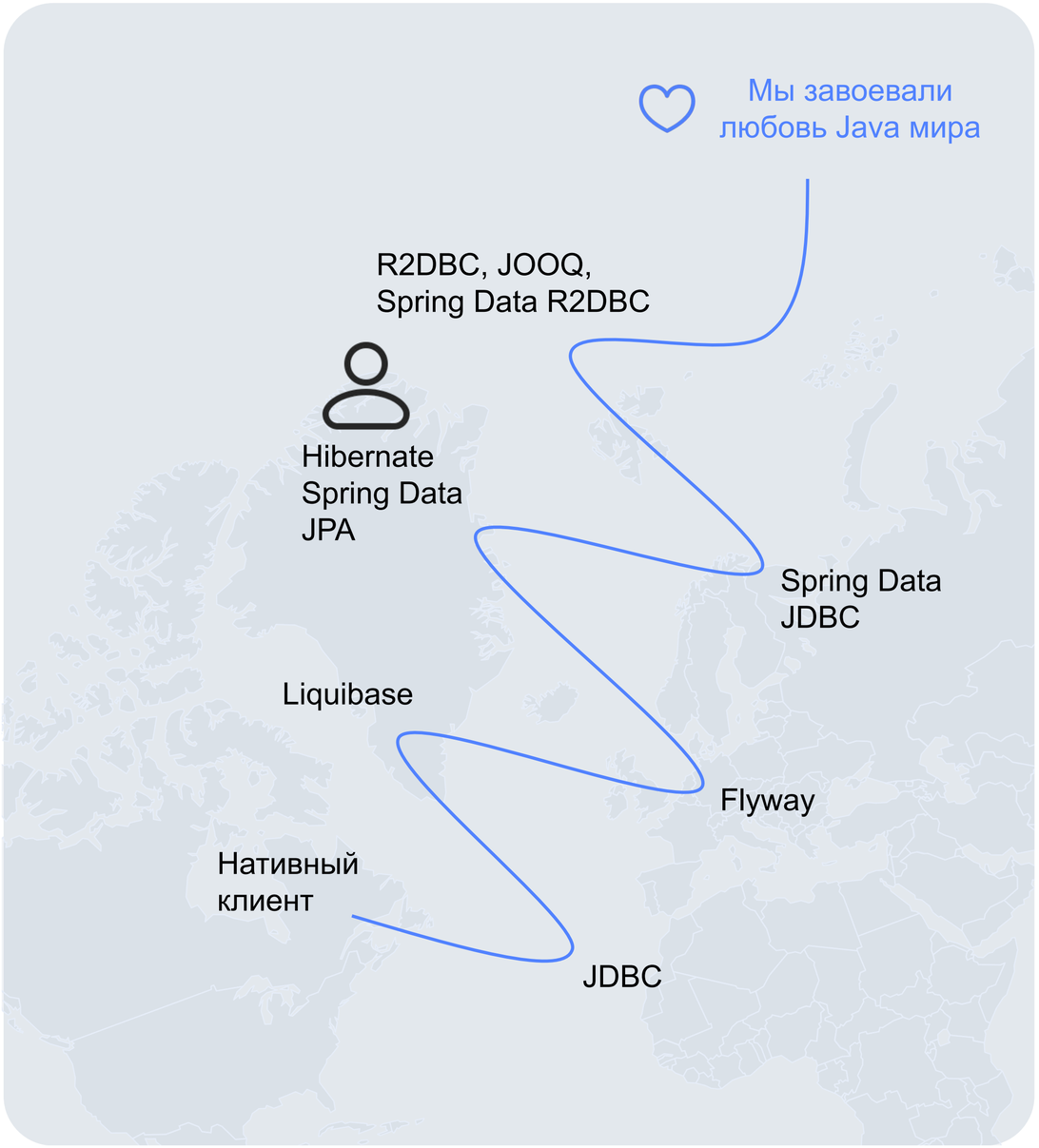
Нельзя обратиться к полю name, так как оно уже названо иначе

```
select
student0_.studentId as studenti1_1_,
student0_.groupId as groupid2_1_,
student0_.name as name3_1_
from Students student0_
order by student0_.name limit ?
```

```
Can't get value: Status: GenericError, Issues:

Error: Member not found: student0_.name
```

Hibernate итоги

1. Двигаем человечка
2. Диалект YDB для Hibernate
3. Spring Data JPA



Spring Data JDBC

“Spring Data JDBC, part of the larger Spring Data family, makes it easy to implement JDBC based repositories.”

Spring Data JDBC

```
interface SimpleUserRepository : ListCrudRepository<User, Long> {

    fun findByUsername(username: String): User?

}
```

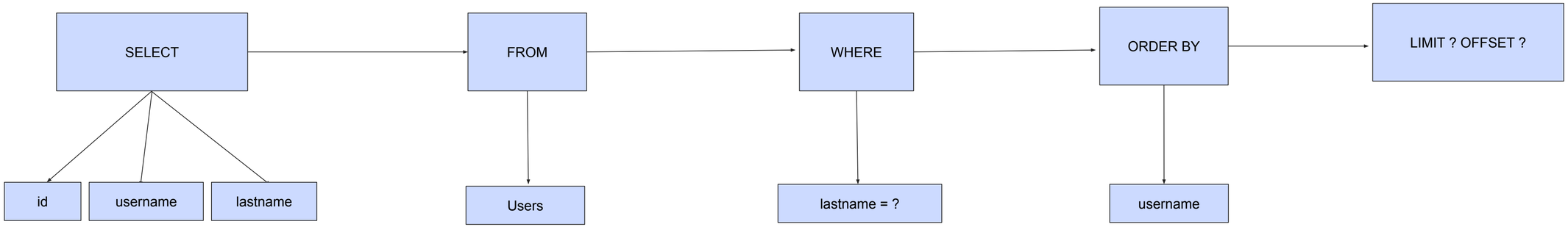
```
SELECT `Users`.`id` AS `id`, `Users`.`username` AS `username`,
       `Users`.`lastname` AS `lastname`, `Users`.`firstname` AS `firstname`
FROM `Users` WHERE `Users`.`username` = ?
```

Spring Data JDBC

```
fun findByLastnameOrderByUsernameAsc(lastname: String, page: Pageable): Slice<User>
```

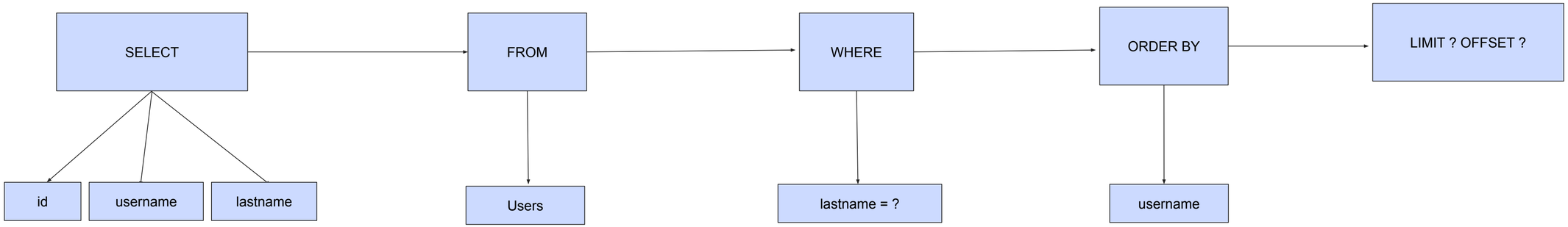
Spring Data JDBC

```
fun findByLastnameOrderByUsernameAsc(lastname: String, page: Pageable): Slice<User>
```



Spring Data JDBC

```
fun findByLastnameOrderByUsernameAsc(lastname: String, page: Pageable): Slice<User>
```




```
SELECT `Users`.`id` AS `id`, `Users`.`username` AS `username`,
       `Users`.`lastname` AS `lastname`, `Users`.`firstname` AS `firstname`
FROM `Users`
WHERE `Users`.`lastname` = ?
ORDER BY `Users`.`username` ASC LIMIT 6 OFFSET 5
```

Диалекты из коробки

Spring Data JDBC включает поддержку следующих баз данных:

- DB2
- H2
- HSQLDB
- MariaDB
- Microsoft SQL Server
- MySQL
- Oracle
- Postgres

Заметим, что их не так много

YDB диалект

```
org.springframework.data.jdbc.repository.config.DialectResolver$JdbcDialectProvider=\
tech.ydb.data.repository.config.YdbDialectProvider
```

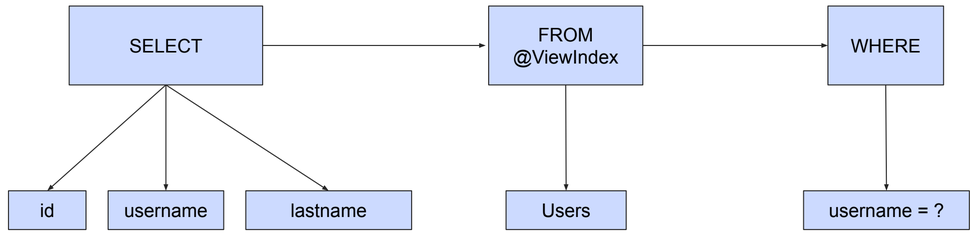
Провайдер представляет реализацию диалекта из модуля `spring-data-relational`, который может быть использован в дальнейшем в Spring Data R2DBC.

Возможности диалекта

- Обработка собственных аннотаций через spring-aop, например `@ViewIndex`
- Формирование пагинации: `LIMIT ? OFFSET ?`
- Процессинг всех имен: оборачиваем в backtick `
- Order by null: None для YQL
- LockMode — YDB не поддерживает

VIEW INDEX

```
@ViewIndex(indexName = "username_index")

fun findByUsername(username: String): User?
```




```
SELECT `Users`.`id` AS `id`, `Users`.`username` AS `username`,
       `Users`.`lastname` AS `lastname`, `Users`.`firstname` AS
`firstname`
FROM `Users` VIEW `username_index` AS `Users`
WHERE `Users`.`username` = ?
```

А есть тут проблемы?

чуть-чуть

LocalDate и LocalDateTime

Внезапные цепочки получения java.sql.JDBCType:

- java.time.LocalDate -> java.sql.Timestamp -> TIMESTAMP
- java.time.LocalDateTime -> java.time.LocalDateTime -> UNKNOWN

JdbcUtil вот такой вот странный.

Получается, не можем использовать LocalDate для типа Date. А для LocalDateTime не можем использовать null, как значение.

Расширение JdbcConverter

```
@Configuration
@EnableJdbcRepositories
@EnableJdbcAuditing
@Import(AbstractYdbJdbcConfiguration.class)
public class YdbJdbcConfiguration {}
```

```
@YdbType("Date")
private LocalDate dateColumn;
@YdbType("Datetime")
private LocalDateTime datetimeColumn;
```

Spring Data JDBC итоги

1. Двигаем человечка
2. Добавляем YDB диалект
   и все работает из коробки
3. Каких либо синтаксических
   ограничений нет
4. Автоинкремент только на подходе
   (используйте `Persistable`)

JOOQ

JOOQ — это библиотека для Java, которая позволяет создавать типобезопасные SQL-запросы путём генерации Java-классов из схемы базы данных и использования удобных конструкторов запросов.

Генерация Java классов

```
<strategy>
   <name>tech.ydb.jooq.codegen.YdbGeneratorStrategy</name>
</strategy>
<database>
   <name>tech.ydb.jooq.codegen.YdbDatabase</name>
   <excludes>.sys.*</excludes>
</database>
```

```
CREATE TABLE series
(
    series_id    Int64,
    title        Text,
    series_info  Text,
    release_date Date,
    PRIMARY KEY (series_id),
    INDEX        title_name GLOBAL ON (title)
);
```

```
public class SeriesRecord extends

UpdatableRecordImpl<SeriesRecord>


public class Series extends

TableImpl<SeriesRecord>
```

YdbDslContext

Расширение DslContext:

- UPSERT
- REPLACE
- VIEW

UPSERT / REPLACE по аналогии с INSERT

```
ydbDSLContext.upsertInto(SERIES)
    .set(record)
    .execute()
```

```
ydbDSLContext.replaceInto(SERIES)
    .set(record)
    .execute()
```

```
upsert into `episodes` (`series_id`, `season_id`, `episode_id`, `title`, `air_date`)
values (?, ?, ?, ?, ?)

replace into `episodes` (`series_id`, `season_id`, `episode_id`, `title`, `air_date`)
values (?, ?, ?, ?, ?)
```

VIEW

```
ydbDSLContext.selectFrom(SERIES.useIndex(Indexes.TITLE_NAME.name))
    .where(SERIES.TITLE.eq(title))
    .fetchOne()
```

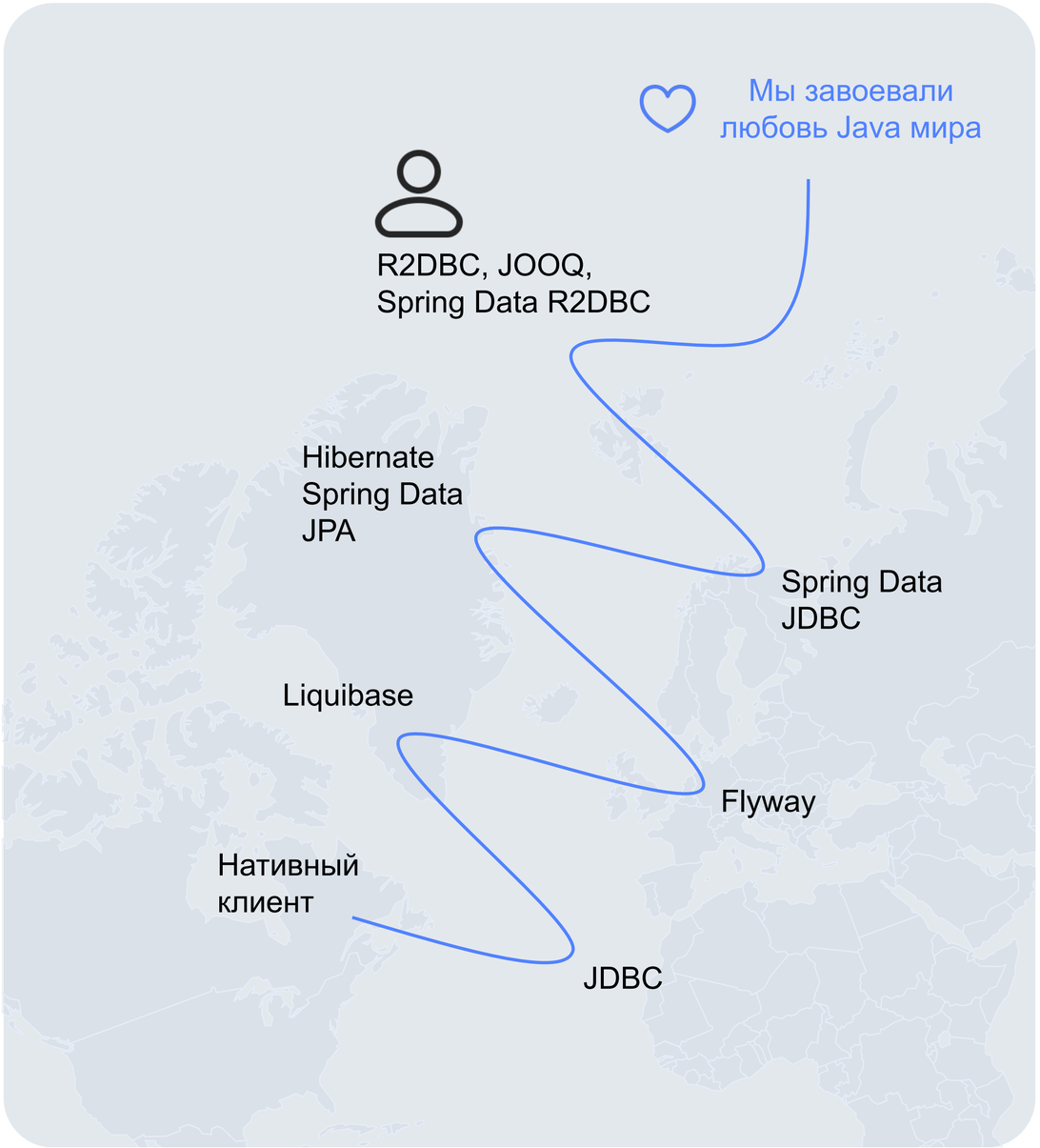
```
select `series`.`series_id`, `series`.`title`, `series`.`series_info`,
`series`.`release_date`
from `series` view `title_name` where `series`.`title` = ?
```

А есть тут проблемы?

чуть-чуть

JOOQ итоги

1. Возможность писать YQL специфичные билдеры
2. Полный контроль над SQL
3. Для Spring Boot нужно будет написать свой стартер
4. Версии - ай ай



R2DBC

:)

Практические выводы для YDB

- Политику ретраев теперь нужно настраивать клиенту явно
- Не бойтесь трогать диалекты, всячески расширять их для своих нужд

Мой рейтинг:

- JOOQ
- Spring Data JDBC
- Hibernate (Spring Data JPA)

Примеры

Концептуальные выводы

Важно понимать, что интеграция с Java-фреймворками, это не только клиентские «приседания», но также вызов серверу.

Готов ли язык запросов принимать такие сложносочиненные запросы?

Это все позволяет базе данных эволюционировать в соответствии с ожиданиями внешних клиентов и стандартов языков.

Благодарность!

Владимир Ситников
@vladimirsitnikv

Александр Горшенин
@alexandr268

Илья Криушенков
@ikriushenkov

Курдюков Кирилл Алексеевич
Tg: @ForeverTired