Санкт–Петербургский государственный университет

Копылов Игорь Евгеньевич

Выпускная квалификационная работа

Перевод SQL диалекта между разными СУБД

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2021 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Технологии программирования»

Научный руководитель:
Кандидат технических наук, доцент СПбГУ, заведующий кафедрой технологии программирования. - Блеканов Иван Станиславович

Рецензент:
Кандидат физико-математических наук, доцент кафедры компьютерных технологий и систем - Коровкин Максим Васильевич

Санкт-Петербург
2025 г.

Содержание

Введение

Современные компании часто используют несколько систем управления базами данных (СУБД) одновременно: одни выбирают PostgreSQL для сложных аналитических запросов, другие — MySQL для высокой производительности веб-приложений, а некоторые — Oracle или MS SQL для корпоративных решений. Это разнообразие приводит к проблеме совместимости: SQL-диалекты разных СУБД имеют синтаксические и функциональные различия, из-за чего миграция или интеграция данных требуют ручного переписывания запросов. Такой процесс трудоёмок, подвержен ошибкам и усложняет взаимодействие между системами.

Традиционно проблема решается с помощью:

- ручного перевода — требует глубоких знаний диалектов и не масштабируется;
- промежуточных инструментов (ORM, ETL-системы) — добавляют накладные расходы и не всегда покрывают все особенности SQL [1, 2];
- специализированных конвертеров — часто ограничены поддержкой конкретных СУБД и устаревают с выходом новых версий [2].

Альтернативным подходом является использование языковых моделей (LLM), способных анализировать и преобразовывать текст с учётом контекста. Эти модели демонстрируют высокую эффективность в задачах машинного перевода, генерации кода и семантического анализа [3, 4, 5]. Благодаря архитектуре трансформеров

Обзор литературы

Исследования в области интеграции данных (Dong Halevy, 2005; Lenzerini, 2002) [1] подчеркивают сложности, возникающие при работе с разнородными источниками. В контексте SQL различия между диалектами изучались в работах, посвященных миграции БД (Klettke et al., 2015) [2], где отмечается, что ручной перевод запросов — ресурсоемкий и подверженный ошибкам процесс. Современные инструменты, такие как Apache Calcite, предлагают rule-based

преобразования, но их поддержка новых версий СУБД требует постоянного обновления

Традиционные методы преобразования SQL включают:

1. парсинг и синтаксические трансформации (на основе грамматик, как в ANTLR).

2. шаблонные замены (например, в pgloader для миграции PostgreSQL).

Однако эти методы плохо масштабируются на сложные запросы.

Машинное обучение применялось для SQL-генерации в Seq2SQL и IRNet (Guo et al., 2019)[6], но фокус был на генерации SQL из текста, а не на переводе между диалектами.

Трансформеры в обработке кода и SQL Архитектура Transformer стала основой для моделей, работающих с кодом:

Codex и демонстрируют способность к трансляции между языками программирования. TaBERT и GRAPPAадаптируют трансформеры для SQL, учитывая структуру запросов. Эти работы показывают, что LLM могут улавливать семантические паттерны в SQL, но их применение к переводу между диалектами требует дополнительных исследований.

В задачах генерации и преобразования SQL используются: Exact Match (EM) — точное совпадение с эталоном. BLEU, ROUGE — для оценки семантической близости (влияние заимствовано из NLP).

Критическая проблема — отсутствие эталонных датасетов для перевода SQL (аналогичных Spider для генерации), что осложняет валидацию новых методов.

Потенциал LLM в переводе SQL Исследования по few-shot learning и fine-tuning [7] специализированных моделей (например, Defog SQLCoder) показывают, что LLM могут адаптироваться к узким задачам, таким как SQL-оптимизация. Однако их эффективность для перевода между диалектами требует экспериментальной проверки, включая:

- влияние контекстного обучения (prompt engineering)

- обработку редких синтаксических конструкций

- интерпретируемость ошибок (анализ типичных сбоев)

Глава 1. Современные подходы к конвертации SQL-запросов: лингвистический анализ и архитектурные решения

1.1 Сравнение SQL-диалектов популярных СУБД

PostgreSQL — это объектно-реляционная система управления базами данных (СУБД) с открытым исходным кодом, активно развиваемая сообществом и поддерживаемая множеством коммерческих и некоммерческих организаций. Она изначально создавалась как расширяемая, надежная и соответствующая стандартам СУБД, и за последние десятилетия получила широкое распространение в промышленной, научной и веб-разработке [8].

Одной из ключевых особенностей PostgreSQL является строгое соблюдение стандарта SQL[8] (в частности, SQL:2008 и более новых редакций), что делает ее удобной для переносимости SQL-кода и взаимодействия с другими системами. В отличие от некоторых других СУБД, PostgreSQL не ограничивается базовым реляционным подходом, а реализует расширения, позволяющие использовать объектно-ориентированное проектирование. Это выражается, например, в поддержке наследования таблиц, пользовательских типов данных и перегрузки операторов и функций.

Сильной стороной PostgreSQL является ее расширяемость[8]. Пользователи могут добавлять новые операторы, функции, агрегаты, типы данных, языки программирования для хранимых процедур (например, PL/pgSQL, PL/Python, PL/Perl и другие), а также расширения (например, PostGIS для работы с геоданными). Такая архитектура делает PostgreSQL особенно гибкой при реализации специализированных решений.

С точки зрения хранения и обработки данных PostgreSQL предлагает развитую систему транзакций с поддержкой ACID и уровней изоляции, включая Serializable Snapshot Isolation[8]. Важной особенностью является Multi-Version Concurrency Control (MVCC) — механизм, обеспечивающий высокую параллельность операций и предотвращающий блокировки при чтении данных. Это позволяет эффективно обслуживать большое количество параллельных клиентов без потери производительности.

СУБД поддерживает сложные типы индексов: B-tree, Hash, GiST, SP-GiST,

GIN, BRIN, что делает возможным эффективную работу с разными типами запросов, включая полнотекстовый поиск, поиск по массивам и JSON-структурам, а также пространственные запросы. Также PostgreSQL предлагает расширенную поддержку JSON и JSONB, позволяя использовать ее как документно-ориентированную базу данных. В отличие от многих традиционных реляционных СУБД, PostgreSQL может эффективно работать с полу-структурированными данными, что делает ее универсальной платформой для различных типов приложений.

Еще одной важной особенностью PostgreSQL является развитая система прав доступа и гибкая ролевая модель[8]. Поддерживается управление пользователями на уровне базы, схем, таблиц, колонок и функций. Механизмы репликации и отказоустойчивости также активно развиваются: PostgreSQL поддерживает потоковую, логическую, синхронную и асинхронную репликации.

На уровне инструментов и экосистемы PostgreSQL хорошо интегрируется с языками программирования (Python, Java, Node.js, Go, C/C++), имеет широкую поддержку в ORM-средствах (например, SQLAlchemy, Django ORM, Hibernate). СУБД активно используется в облачных решениях (Amazon RDS/Aurora, Google Cloud SQL, Azure Database for PostgreSQL), что подтверждает её зрелость и востребованность.

MySQL — это одна из самых популярных систем управления реляционными базами данных (СУБД), широко используемая в веб-разработке, бизнес-приложениях и корпоративных системах[9]. СУБД известна своей простотой в установке и использовании, высокой скоростью обработки запросов и хорошей масштабируемостью, особенно в сценариях, связанных с веб-сервисами и распределёнными системами.

MySQL является СУБД с открытым исходным кодом, однако существуют разные редакции: Community Edition (бесплатная и с открытым кодом) и несколько коммерческих редакций, предлагаемых Oracle с дополнительными функциями, поддержкой и инструментами. Одной из сильных сторон MySQL является её производительность при работе с большим количеством простых запросов и быстрым чтением данных. Несмотря на то, что MySQL поддерживает стандарт SQL, она исторически была более либеральна в отношении его соблюдения, чем, например, PostgreSQL. Допускается выполнение некорректных

с точки зрения стандарта SQL операций без генерации ошибок (в зависимости от SQL режима). Однако с выходом новых версий в MySQL добавлены более строгие режимы работы и расширена поддержка стандартных SQL-функций.

Пример различий в поведении MySQL и PostgreSQL:

Листинг 1: Группировка без агрегатной функции

```
-- Example of a query with GROUP BY, where not all columns are aggregated:
SELECT department, employee_name
FROM employees
GROUP BY department;
```

В PostgreSQL данный запрос вызовет ошибку: *column 'employee_name' must appear in the GROUP BY clause or be used in an aggregate function*, поскольку он строго следует стандарту SQL.

В MySQL при отключённом режиме `ONLY_FULL_GROUP_BY` такой запрос выполнится без ошибок, но результат может быть непредсказуемым: MySQL вернёт одно произвольное значение `employee_name` для каждой группы `department`.

MySQL использует модульную систему хранения данных, что позволяет подключать различные движки хранения (storage engines). Самыми популярными являются InnoDB и MyISAM. В современных версиях именно InnoDB используется по умолчанию, поскольку он поддерживает транзакции, внешние ключи, блокировки на уровне строк и обеспечивает выполнение требований ACID. В отличие от PostgreSQL, где имеется единая модель хранения, в MySQL выбор движка позволяет гибко адаптировать СУБД под конкретные нужды (например, высокая скорость чтения у MyISAM или транзакционная надёжность у InnoDB).

Одной из особенностей MySQL является относительно ограниченная поддержка расширенных типов данных и функциональности по сравнению с Postgre. Пример различий в преобразовании типов (implicit casting):

В MySQL неявные преобразования типов допускаются во многих случаях, что может привести к неожиданным результатам. PostgreSQL, напротив, требует более строгого соответствия типов, что предотвращает потенциальные ошибки.

Листинг 2: Сравнение числового и строкового значения

```
SELECT *
FROM users
WHERE user_id = '123abc';
```

MySQL:

MySQL выполнит неявное преобразование строки '123abc' в число '123', проигнорировав некорректную часть строки. Запрос выполнится и вернёт всех пользователей с 'user_id = 123'.

PostgreSQL:

PostgreSQL выбросит ошибку: *invalid input syntax for integer: '123abc'*, поскольку строка '123abc' не может быть приведена к типу integer.

Это различие особенно критично при миграции запросов между СУБД, так как в PostgreSQL поведение MySQL будет считаться нарушением семантики SQL.

Поддержка JSON появилась только в версии 5.7, а полноценная работа с JSON-операциями — в 8.0[9]. Однако с каждой версией MySQL расширяет свою функциональность: появляются оконные функции, CTE-запросы (WITH), функции для работы с геоданными, улучшенная полнотекстовая индексация, репликация с фильтрацией и логическая репликация.

MySQL поддерживает репликацию данных как в синхронном, так и в асинхронном режиме. Классическая схема — master-slave (лидер-подчинённый), однако в новых версиях появилась поддержка multi-source replication и репликации между несколькими мастерами.

С точки зрения безопасности, MySQL предоставляет базовые механизмы управления доступом, поддержку SSL, шифрования данных на уровне столбцов и аутентификацию через плагины. Однако система разграничения прав более ограничена по сравнению с PostgreSQL, особенно если речь идёт о сложной иерархии ролей и разрешений.

MySQL активно используется в стеке LAMP (Linux, Apache, MySQL, PHP/Python/Perl) и поддерживается большинством хостинг-провайдеров и облачных платформ. Она легко интегрируется с широким спектром языков программирования, обладает поддержкой популярных ORM-библиотек и предоставляет инструменты для резервного копирования, мониторинга и настройки производительности.

SQLite крайне лёгкая и самодостаточная — нет необходимости устанавливать сервер, управлять пользователями, запускать службы. Всё, что требуется для работы — подключение библиотеки и создание/открытие файла базы данных[10]. Это делает её идеальной для встроенных систем, разработки прототипов, тестирования и приложений, где не требуется масштабируемость или сложная многопользовательская работа.

С точки зрения поддержки SQLite реализует значительную часть стандарта SQL-92, SQL-99 и частично SQL:2003, включая такие возможности как транзакции, вложенные запросы, индексы, представления, триггеры. Однако, многие расширенные функции, присутствующие в PostgreSQL и MySQL, в SQLite либо отсутствуют, либо реализованы упрощённо. Например, нет полноценной поддержки внешних ключей по умолчанию (она может быть включена вручную), нет расширенных типов индексов, хранимых процедур, параллельной обработки запросов.

SQLite использует механизм journal-based transaction logging и обеспечивает полную совместимость с ACID — транзакции либо выполняются целиком, либо полностью откатываются[10]. Однако, из-за своей архитектуры, SQLite не предназначена для работы с большим числом одновременных соединений. Она блокирует файл базы при записи, что может стать узким местом в многопользовательских веб-приложениях или в системах с интенсивной нагрузкой на запись.

С точки зрения производительности, SQLite показывает отличные результаты при работе с небольшими или средними объемами данных, особенно в read-heavy сценариях. За счёт отсутствия сетевых операций и серверной прослойки, время отклика при локальных операциях минимально. Но при этом отсутствует полноценное масштабирование — невозможно разнести обработку данных на разные узлы или реплицировать базу стандартными средствами, как в PostgreSQL или MySQL.

ClickHouse — это аналитическая колонкоориентированная система управления базами данных, разработанная в компании Яндекс для высокопроизводительной обработки аналитических запросов в реальном времени. Основное предназначение ClickHouse — хранение и анализ больших объёмов данных с возможностью выполнять сложные агрегирующие запросы с высокой скоростью.

Архитектурно система спроектирована с фокусом на OLAP-нагрузки (Online Analytical Processing), то есть на аналитическую обработку, в отличие от систем, ориентированных на транзакционные (OLTP) задачи[11].

Одной из фундаментальных особенностей ClickHouse является его колонкоориентированное хранение данных. Это означает, что данные одного столбца хранятся вместе, последовательно, в отличие от строкоориентированных СУБД, где данные одной строки хранятся целиком. Такой подход существенно ускоряет выполнение аналитических запросов, поскольку позволяет считывать с диска только те столбцы, которые действительно участвуют в запросе, минимизируя ввод-вывод и эффективно используя кэш. Кроме того, ClickHouse активно применяет сжатие данных, достигая высокой плотности хранения без потерь в производительности[11].

Система поддерживает собственный SQL-подобный язык запросов, который, несмотря на свои особенности и отклонения от стандарта SQL, достаточно выразителен для построения сложных агрегаций, оконных функций, вложенных подзапросов и работы с массивами, структурами, JSON-данными и другими типами. Одним из сильных мест ClickHouse является наличие богатого набора агрегатных функций и оптимизаций для их выполнения, в том числе таких, как приближённые подсчёты (например, HyperLogLog для оценки количества уникальных значений) и специализированные агрегаты для работы с временными рядами.

Система обладает горизонтально масштабируемой архитектурой[11], поддерживает репликацию, шардирование и отказоустойчивость. Сервер ClickHouse может быть развернут в виде единого узла или как распределённый кластер, где данные делятся по шардам и реплицируются между узлами для обеспечения высокой доступности и производительности. Для координации репликации используется ZooKeeper, который управляет метаданными и синхронизацией между репликами.

ClickHouse проектировался для обработки огромных объёмов данных с минимальной задержкой, и на практике способен обрабатывать миллиарды строк в секунду при условии правильного проектирования схемы данных и запросов. Для достижения этой цели используются механизмы векторной обработки, где данные обрабатываются не по одной строке, а блоками, что позволяет

эффективно использовать возможности современных процессоров и минимизировать накладные расходы на выполнение операций.

Важно отметить, что в ClickHouse отсутствует классическая поддержка транзакций и блокировок в стиле традиционных СУБД[11]. Модель согласованности основана на append-only подходе: данные записываются в виде immutable-частей, которые позже объединяются и сливаются с помощью фоновых процессов. Это обеспечивает высокую скорость записи и избегание блокировок, но при этом требует другого подхода к управлению изменениями данных и обработки ошибок.

ClickHouse предоставляет широкий инструментарий для администрирования, мониторинга и интеграции: доступен REST API, нативный клиент, поддержка драйверов для различных языков (Python, Go, Java и др.), а также интеграция с системами визуализации (например, Grafana, Redash, Superset). Кроме того, ClickHouse активно используется как backend для BI-систем и систем метрик, позволяя создавать дешёвые и быстрые решения для анализа данных в реальном времени.

Таблица 1: Сравнение диалектов и особенностей различных СУБД

№	**СУБД**	**Описание**	**Тип хранения**
1	PostgreSQL	Объектно-реляционная СУБД с расширенной поддержкой SQL, типов данных, транзакций, ACID, MVCC и JSONB. Широко применяется для сложных бизнес-приложений и аналитических задач. Поддерживает хранимые процедуры, репликацию и параллельные запросы.	(heap)

№	СУБД	Описание	Тип хранения
2	MySQL	Реляционная СУБД, популярная в веб-разработке. Поддерживает ACID через InnoDB, JSON (начиная с версии 5.7), хранимые процедуры и репликацию. Простая в настройке, подходит для CMS и SaaS-проектов.	(InnoDB)
3	SQLite	Лёгкая, встроенная СУБД. Используется как библиотека, не требует сервера. Применяется в мобильных и встраиваемых приложениях. Поддерживает транзакции, но ограничена в масштабировании и многопользовательской работе.	(один файл)
4	ClickHouse	Колонкоориентированная аналитическая СУБД, предназначенная для быстрого анализа больших объёмов данных. Не поддерживает транзакции в классическом понимании, но обеспечивает высокую скорость обработки запросов. Имеет встроенную поддержку кластеризации, шардирования и репликации.	по колонкам

1.2 Проблемы ручного и автоматизированного перевода SQL

Перевод SQL-запросов между диалектами различных СУБД остаётся одной из ключевых проблем при разработке кросс-платформенных решений[12], миграции данных и построении гибких систем хранения. Ручной подход к преобразованию SQL, несмотря на кажущуюся гибкость, сопряжён с рядом уязвимостей. Программист, сталкиваясь с различиями в синтаксисе (например, LIMIT vs TOP, ILIKE vs LOWER и т.д.), может допустить незаметные ошибки, которые приведут к некорректному поведению запросов, особенно в условиях сложной логики объединений, вложенных подзапросов или оконных функций. Кроме того, ручная адаптация SQL-кода трудоёмка, слабо масштабируется и плохо поддаётся сопровождению при увеличении числа поддерживаемых СУБД.

Автоматизированные rule-based-системы трансляции SQL, такие как SQLGlot или Apache Calcite, предлагают формальный подход на основе грамматик и деревьев разбора. Тем не менее, они обладают рядом ограничений. Во-первых, многие особенности SQL-диалектов не укладываются в обобщённую грамматику без потерь: поведение функций агрегации, оконные функции, особенности сортировки и даже простые типы данных могут интерпретироваться по-разному. Во-вторых, такие трансляторы часто теряют семантику запроса - например, специфичная оптимизация для ClickHouse может быть некорректно перенесена в PostgreSQL без учета индексации или распределённости. Даже при наличии расширяемых правил конвертации, поддержание актуальности соответствий между диалектами требует ручной доработки и глубокого знания как источника, так и целевой СУБД.

Использование ORM (Object-Relational Mapping) и ETL-систем (Extract, Transform, Load) также не избавляет от проблем несовместимости SQL. ORM-решения, такие как SQLAlchemy или Hibernate, хотя и абстрагируют работу с базой, в реальности ограничены подмножеством операций, поддерживаемых всеми целевыми СУБД. Запросы, сгенерированные ORM, часто не оптимальны, плохо интерпретируются в контексте конкретной базы и требуют ручной оптимизации на уровне 'native SQL'. ETL-инструменты, в свою очередь, сталкиваются с проблемой различий в типах данных, правилах преобразования дат и чисел, а также с различиями в транзакционных моделях. В результате авто-

матизация зачастую дополняется ручной постобработкой, что нарушает цель универсализации и увеличивает риски ошибок.

Агрегатные функции, такие как SUM, AVG, COUNT, по умолчанию игнорируют NULL, однако некоторые СУБД допускают изменения этого поведения через опции, а в контексте оконных функций или при использовании FILTER результаты могут существенно отличаться. Это может привести к незаметным логическим ошибкам, когда в одной системе запрос возвращает корректное среднее значение, а в другой - заниженное или вовсе NULL, особенно при отсутствии предварительной фильтрации.

Не менее коварной проблемой является неявное приведение типов. Разные СУБД по-разному обрабатывают ситуации, когда, например, строка сравнивается с числом, или когда происходит конкатенация значений различных типов [см. Листинг 1]. Это делает невозможным прямолинейный перенос логики, особенно в случае, если сравнения и арифметика происходят на данных, полученных из внешних источников (ETL), где типы могут быть нестрого определены[12]. В условиях сложных вложенных запросов это может вызывать неконсистентное поведение или некорректную фильтрацию данных.

Дополнительные сложности связаны с различиями в поведении сортировки и сравнении строк. Разные СУБД используют разные правила сопоставления символов (collation), что напрямую влияет на порядок сортировки, сравнение строк с разным регистром и работу операторов LIKE, ILIKE, = и <>. Например, PostgreSQL использует системный collation, который можно задавать при создании базы, тогда как SQLite чаще всего использует бинарное сравнение по умолчанию, чувствительное к регистру.

```
-- query 1
SELECT AGE WHERE name = 'Igor' and sername = 'Kopylov';
-- query 2
SELECT AGE WHERE name = 'igor' and sername = 'kopylov';
```

Это может привести к тому, что один и тот же запрос с условием в одной СУБД найдёт значение, а в другой - нет. Аналогично, сортировка строк с кириллическими символами, спецсимволами или пробелами может отличаться, особенно если в одной из систем настроен нестандартный порядок символов.

Сложные конструкции в SQL, такие как рекурсивные обобщённые таблич-

ные выражения (Common Table Expressions, CTE) и различные типы соединений (JOIN), представляют собой один из наиболее уязвимых аспектов при переносе запросов между диалектами разных СУБД. Несмотря на то, что синтаксис CTE стандартизирован в рамках [12], его реализация и поведение в конкретных системах существенно различаются. Например, PostgreSQL поддерживает рекурсивные CTE полностью и эффективно, включая возможность управления порядком обхода (ширина/глубина), использования оконных функций внутри рекурсивной части и задания условий остановки через LIMIT или FILTER[8]. В то же время в MySQL поддержка рекурсивных выражений появилась лишь с версии 8.0, и её реализация ограничена: отсутствуют многие возможности оптимизации, накладываются ограничения на максимальное число итераций, и рекурсия не всегда работает стабильно в условиях высокой вложенности или при работе с большими объёмами данных[9]. SQLite также поддерживает рекурсивные CTE, но при этом накладывает ограничения на глубину рекурсии и слабо масштабируется. В ClickHouse рекурсивные CTE на уровне SQL вообще не поддерживаются[11]: подобная логика требует императивной реализации на уровне внешнего приложения или процедурных скриптов. Это создаёт серьёзные затруднения при миграции запросов, построенных на рекурсивных обходах графов, иерархий или взаимосвязанных структур.

Листинг 3: PostgreSQL: Рекурсивный CTE

```
WITH RECURSIVE subordinates AS (
    SELECT id, name, manager_id
    FROM employees
    WHERE id = 1
  UNION ALL
    SELECT e.id, e.name, e.manager_id
    FROM employees e
    INNER JOIN subordinates s ON e.manager_id = s.id
)
SELECT * FROM subordinates;
```

Листинг 4: ClickHouse: Рекурсия не поддерживается

```
1. SELECT * FROM employees WHERE id = 1;
2. SELECT * FROM employees WHERE manager_id IN (1.);
```

В PostgreSQL (см. [листинг 3.]) реализована полноценная поддержка рекурсивных обобщённых табличных выражений (CTE)[8]. Это позволяет выразить иерархический обход структуры данных (например, иерархии сотрудников) на уровне одного SQL-запроса. В приведённом примере запрос начинается с начальника с id = 1, а затем с помощью рекурсивного объединения (UNION ALL) извлекаются все подчинённые на всех уровнях вложенности. Такой подход прост, декларативен и масштабируем.

В ClickHouse отсутствует поддержка рекурсивных CTE на уровне SQL. В связи с этим, для выполнения аналогичной задачи требуется реализовывать итеративную логику вручную, обычно вне СУБД (см. [листинг 4.]). Пример показывает, как осуществляется последовательный выбор подчинённых на каждом уровне: сначала выбирается начальный элемент (id = 1), затем подчинённые, у которых manager_id входит в набор id предыдущего уровня, и так далее. Этот подход требует организации цикла на уровне внешнего приложения (например, на Python или другом языке), что усложняет архитектуру и снижает удобство использования ClickHouse для подобных задач.

Ещё более сложной проблемой становятся различия в поведении операций JOIN, особенно в части их семантики, порядка выполнения и оптимизации. На базовом уровне все СУБД поддерживают INNER JOIN, LEFT JOIN, RIGHT JOIN и FULL OUTER JOIN, однако детали их реализации различаются. Например, в PostgreSQL оптимизатор умеет преобразовывать JOIN-выражения в соответствии с их селективностью, может использовать хэшированные, вложенные или индексированные соединения. MySQL, в зависимости от движка (InnoDB, MyISAM), по-разному трактует порядок выполнения соединений, причём в некоторых случаях порядок таблиц в запросе напрямую влияет на результат. Кроме того, в MySQL могут возникать ошибки или неожиданное поведение при соединении таблиц, содержащих поля с одинаковыми именами, особенно если не используются алиасы или явные указания имён[9]. В SQLite поведение соединений упрощено и зачастую не оптимизируется, что при больших данных приводит к существенной деградации производительности. В ClickHouse JOIN-операции ограничены: RIGHT JOIN и FULL JOIN поддерживаются с оговорками, а таблицы должны быть предварительно подготовлены (распределены или материализованы в определённой форме). Более того, в ClickHouse реко-

мендуется избегать тяжелых JOIN, поскольку система оптимизирована под денормализованные схемы и массивные агрегаты, а не под сложные реляционные запросы.

Семантические пробелы в SQL-запросах между диалектами СУБД представляют собой тонкую, но критически важную категорию несовместимостей, способную повлиять как на корректность вычислений, так и на логическую целостность приложения. Одной из наиболее значимых проблем является различная точность числовых операций. Хотя спецификация SQL допускает поддержку типов с фиксированной и плавающей точкой (например, DECIMAL, NUMERIC, FLOAT, DOUBLE), их реализация и интерпретация различаются. Так, в PostgreSQL арифметические операции над NUMERIC по умолчанию сохраняют высокую точность, что делает её подходящей для финансовых расчётов[8]. В MySQL тип DECIMAL также поддерживается, но точность может снижаться при преобразованиях или использовании агрегатных функций[9]. В SQLite нет строгой типизации: типы данных зачастую носят рекомендательный характер, и арифметика над числами может приводиться к REAL, теряя точность[10]. В ClickHouse Decimal-типы реализованы как отдельные структуры, но их поддержка в функциях и выражениях ограничена, особенно при смешанных типах[11]. Как следствие, один и тот же запрос, например вычисляющий налог с точностью до двух знаков, может в разных СУБД давать отличающиеся результаты, что особенно критично в отчётности и финансовых расчётах.

Дополнительное расхождение проявляется в реализации транзакционной модели[12]. PostgreSQL обеспечивает полную поддержку транзакций, включая уровни изоляции и управление блокировками, что позволяет гарантировать ACID-семантику[8]. В MySQL поддержка транзакций зависит от используемого движка: только InnoDB предоставляет полноценную транзакционность, в то время как MyISAM её не поддерживает[9]. SQLite реализует транзакции на уровне файловой системы, что надёжно в условиях одиночного подключения, но не масштабируется в многопользовательской среде[10]. ClickHouse же вообще не поддерживает традиционную транзакционную модель, полагаясь на модель «вставки-как-иммутабельного блока» и отказоустойчивость через репликацию[11]. Эти различия означают, что перенос логики, зависящей от BEGIN ... COMMIT, особенно в рамках ETL-процессов или агрегирующих

транзакций, требует полной переработки механизма согласованности.

Проблемы конвертации типов данных остаются ещё одной критичной точкой. Даже базовые типы, такие как BOOLEAN, DATE, TIMESTAMP, реализуются по-разному. PostgreSQL, например, строго различает DATE, TIMESTAMP WITH TIME ZONE и TIMESTAMP WITHOUT TIME ZONE, тогда как MySQL допускает неявные преобразования между ними. SQLite не имеет нативных типов дат и времени: все они хранятся как строки, числа или текст в ISO-формате, что делает невозможным использование ряда функций без дополнительного преобразования. В ClickHouse даты и время представлены специализированными типами (Date, DateTime64 и т.д.), но их интерпретация зависит от часового пояса сервера, а не пользователя, что может стать причиной временных рассогласований. Проблемы возникают и при переносе данных: значение TRUE в одной системе может интерпретироваться как 1, "true"или даже нераспознанная строка в другой.

Помимо логических различий, автоматическая генерация SQL-запросов в рамках ORM, ETL или систем трансляции часто страдает от практических ограничений. Генерируемый код, как правило, не учитывает индексные структуры, статистику таблиц, объёмы данных и специфики схем[**?**]. В результате такие запросы не только громоздки, но и малопригодны для выполнения в продуктивной среде. Например, ORM-генераторы могут использовать избыточные LEFT JOIN, даже если данные в подтаблицах гарантированно существуют, или строить неэффективные подзапросы вместо оконных функций. Отсутствие адаптации под конкретные схемы выражается в невозможности учёта денормализации, шардирования или кастомных оптимизаций: запрос работает "на всех"базах одинаково плохо, не раскрывая потенциала конкретной СУБД[**?**].

Особенно остро эти проблемы проявляются при попытке использовать универсальные подходы для аналитических СУБД - таких как ClickHouse. Эти системы проектировались не для совместимости с традиционным SQL, а для высокоскоростной обработки больших массивов данных. Они требуют иного подхода: упора на колоночное хранение, предварительное агрегирование, денормализацию и пакетную загрузку. Типовые SQL-запросы, особенно созданные автоматически, плохо сочетаются с архитектурными особенностями таких СУБД. Даже поддержка SQL-синтаксиса в них зачастую формальна: например,

в ClickHouse GROUP BY требует строгого соответствия агрегаций, а подзапросы ограничены по глубине и функциональности.

На основании проведённого анализа особенностей различных диалектов SQL и рассмотрения проблем, возникающих при их трансляции и миграции, можно сделать ряд важных выводов, подчёркивающих сложность и многоаспектность задачи перевода SQL-кода между СУБД. Несмотря на кажущуюся универсальность языка SQL, на практике каждая система управления базами данных реализует собственный диалект, в котором синтаксические, семантические и операционные различия оказываются весьма значительными. PostgreSQL, MySQL, SQLite, ClickHouse и другие СУБД следуют различным подходам в трактовке базовых концепций, таких как работа с типами данных, агрегация, сортировка, соединения, рекурсивные конструкции, транзакционность и точность вычислений.

С одной стороны, эти различия порождают риски при ручной миграции SQL-кода: от синтаксических ошибок и несовпадений ключевых слов до более сложных семантических ловушек, таких как различия в поведении с NULL, типами данных и порядком сортировки. Разработка запросов, универсальных для нескольких СУБД, практически невозможна без учёта множества нюансов и компромиссов. Даже при формально одинаковом SQL-запросе поведение и производительность могут кардинально отличаться. Это особенно критично в ситуациях, где важна точность расчётов, например, в финансовых системах или при аналитической отчётности.

С другой стороны, существующие средства автоматизированного преобразования SQL (включая rule-based парсеры, ORM-решения и ETL-платформы) демонстрируют ограниченную применимость[12]. Они либо опираются на простые шаблонные правила, не способные корректно интерпретировать сложную бизнес-логику, либо генерируют избыточные и неэффективные конструкции, не учитывающие архитектурные особенности целевой СУБД. Особенно остро это проявляется в отношении аналитических систем (ClickHouse), где стандартные SQL-подходы требуют полной переработки. Кроме того, проблема усиливается отсутствием обратной связи с хранилищем схемы, что делает невозможным автоматический учёт денормализации, шардирования и специфических ограничений.

Таким образом, задача трансляции SQL между диалектами выходит за рамки простой технической процедуры. Она требует системного подхода, включающего глубокое понимание как синтаксических конструкций, так и особенностей исполнения, хранения, планирования запросов и организации данных. В случае серьёзной миграции или построения кросс-СУБД совместимых решений, необходимо рассматривать применение промежуточных языков представления, создание абстракций поверх SQL, а также использование тестовых наборов и семантических валидаций для контроля соответствия результатов[**?**]. Только сочетание инженерной строгости, грамотной архитектуры и контекстно-зависимой адаптации может обеспечить корректную и эффективную реализацию переноса SQL-логики между различными СУБД, минимизируя потери точности, производительности и функциональности.

Применение больших языковых моделей (LLM) для трансляции SQL-запросов между диалектами следует рассматривать прежде всего как исследовательский подход, направленный на изучение возможностей ИИ в области синтаксического и семантического сопоставления запросов. Несмотря на впечатляющие результаты в задачах генерации и понимания кода, LLM не заменяют инженерные решения и не гарантируют корректного поведения во всех случаях - особенно при работе со сложными запросами, зависящими от специфики конкретной СУБД.

Тем не менее, LLM могут эффективно использоваться в качестве ассистентов - для предложений по преобразованию запросов, автоматического выявления потенциальных несовместимостей, генерации тест-кейсов или упрощения запросов для начального этапа миграции. Их применение должно сопровождаться валидацией и контролем со стороны специалиста, а также интеграцией в архитектуру более широких инструментов миграции и анализа SQL.

1.3 Обзор существующих исследований в области преобразования SQL

Лексический и синтаксический анализ SQL-запросов - это фундаментальный этап в процессе автоматизированного преобразования между диалектами различных СУБД. Этот метод предполагает предварительное структурное по-

нимание SQL-кода путём его разбиения на элементы (лексемы) и построения синтаксической структуры, которая может быть затем интерпретирована, преобразована и сгенерирована заново с учётом особенностей целевого диалекта [14]. Ключевым преимуществом подхода является его формальная строгость и масштабируемость: при корректной реализации он обеспечивает высокий уровень контроля над преобразуемым кодом и минимизирует ошибки за счёт соблюдения грамматических правил.

Процесс начинается с лексического анализа (tokenization), в ходе которого SQL-текст разбивается на последовательность элементарных единиц - токенов. Это могут быть ключевые слова (SELECT, FROM, WHERE), идентификаторы, операторы (+, =, <>), литералы ('abc', 123), спецсимволы и комментарии. Лексер (обычно реализуемый средствами регулярных выражений или генераторов на основе формального описания) устраняет пробелы, интерпретирует границы токенов и передаёт результат синтаксическому анализатору[14].

Следующий этап - синтаксический анализ (парсинг). Он предполагает сопоставление полученной последовательности токенов с правилами формальной грамматики, описывающей язык SQL в терминах контекстно-свободной грамматики (CFG). В этой задаче широко используются инструменты вроде ANTLR (Another Tool for Language Recognition), YACC/Bison, PEG-парсеры или кастомные LL/LR-алгоритмы. На основе грамматики создаётся парсер, который строит абстрактное синтаксическое дерево (AST - Abstract Syntax Tree). AST представляет собой древовидную структуру, где каждый узел соответствует синтаксической конструкции (например, оператор SELECT, список выражений, подзапрос и т.д.). Это дерево является основной моделью, на которой работает транслятор между диалектами.

Однако здесь возникают важные сложности. Во-первых, грамматики SQL очень объёмны и разнообразны, и в реальности ни одна СУБД не реализует строго стандартный SQL. Каждый диалект имеет расширения: специальные операторы (UPSERT, RETURNING), типы данных (JSONB, GEOMETRY), кастомные функции, синтаксис LIMIT, TOP, FETCH, различные вариации JOIN и оконных функций. Поэтому универсальная грамматика, охватывающая всё многообразие SQL-диалектов, на практике крайне сложна в поддержке. Парсеры часто «ломаются» при встрече незнакомой конструкции, особенно если грамматика

неполная или противоречивая. Для этого в крупных системах реализуется мультиграмматический подход, где поддерживаются отдельные правила под разные диалекты, либо используются фреймворки вроде SQLGlot, которые абстрагируют SQL в общий промежуточный формат.

Во-вторых, конфликты парсинга (shift-reduce, reduce-reduce и др.) возникают при неоднозначности грамматик, особенно в случае вложенных выражений и контекстно-зависимого синтаксиса. Например, одна и та же конструкция может трактоваться как подзапрос или табличное выражение в зависимости от положения. Решение таких конфликтов требует ручной доработки грамматики, внедрения предикатов разбора, lookahead-правил и других механизмов, усложняющих реализацию.

Кроме того, AST не всегда отражает семантику запроса. Например, выражения с NULL, агрегатные функции, вложенные CASE и оконные функции требуют более глубокого анализа. Поэтому на базе AST часто строится внутреннее представление (IR - intermediate representation), которое моделирует смысловую структуру запроса и позволяет безопасно трансформировать его в другой диалект[14]. На этом этапе возможна оптимизация, переупорядочивание, замена конструкций (LIMIT на FETCH FIRST, CONCAT на ||, IFNULL на COALESCE и т.д.) и дальнейшая генерация SQL в целевом синтаксисе.

Подход с использованием парсера на базе формальной грамматики имеет широкое применение: он лежит в основе таких инструментов, как Apache Calcite, SQLGlot, JSQLParser, Presto SQL Parser, а также промышленных систем трансляции SQL в ETL-платформах. Однако его эффективность напрямую зависит от полноты грамматики, корректной реализации разбора, учёта контекста и поддержки специфики целевой СУБД.

В заключение можно сказать, что лексико-синтаксический анализ является мощным, но технически сложным методом трансляции SQL между диалектами. Его реализация требует хорошей инженерной культуры, наличия формальных описаний грамматик, обширных тестов и постоянной актуализации под меняющиеся версии СУБД. Тем не менее, при правильной архитектуре этот метод обеспечивает наилучшее качество трансформации и масштабируемость.

1.4 Синтаксический анализ и построение AST

Синтаксический анализ начинается после лексической обработки и предполагает сопоставление токенов с правилами формальной грамматики SQL. Результатом является построение абстрактного синтаксического дерева (AST - Abstract Syntax Tree), которое отражает иерархическую структуру выражения.

Рассмотрим пример SQL-запроса:

Листинг 5: PostgreSQL: Исходный запрос

```
SELECT name, age
FROM users
WHERE status = 'active'
ORDER BY age DESC
LIMIT 10;
```

На его основе строится AST следующего вида:

```
SelectStatement
 Projection
    Column(name)
    Column(age)
 FromClause
    Table(users)
 WhereClause
    Condition(status = 'active')
 OrderBy
    age DESC
 Limit
     10
```

На следующем этапе AST может быть трансформировано в промежуточное представление (IR), которое отражает семантику запроса:

```
IR {
  projection: ["name", "age"],
  source: "users",
```

```
  filter: Eq("status", "active"),
  sort: [("age", DESC)],
  limit: 10
}
```

Это позволяет безопасно транслировать запрос в другой диалект, например ClickHouse:

Листинг 6: ClickHouse: Преобразованный запрос

```
SELECT name, age
FROM users
WHERE status = 'active'
ORDER BY age DESC
LIMIT 10;
```

Хотя на первый взгляд запросы идентичны, важно учитывать различия в поведении функций и типов, а также форматах LIMIT/OFFSET, агрегации и оконных функций, которые в ClickHouse могут потребовать иной реализации.

Конфликт разбора: в случае конструкции вида

```
SELECT (SELECT COUNT(*) FROM orders) AS total_orders;
```

возможно неоднозначное толкование вложенного выражения - как подзапроса или табличного выражения. Такие случаи требуют расширенного lookahead и ручной правки грамматики для корректного распознавания структуры.

Таким образом, синтаксический анализ с построением AST и IR - это критически важный этап, обеспечивающий точную трансформацию SQL между диалектами, особенно в условиях различий реализаций стандартов и семантики.

Методы шаблонных замен и rule-based трансформаций являются одними из наиболее широко используемых подходов при автоматическом преобразовании SQL-запросов между диалектами. Они опираются на использование набора правил и шаблонов, которые соответствуют определённым синтаксическим или семантическим конструкциям и преобразуют их в эквивалентные конструкции в целевом диалекте. Эти методы достаточно просты в реализации и эффективны при решении узкоформулированных задач, однако обладают целым рядом ограничений, связанных с хрупкостью и недостаточной универсальностью.

На базовом уровне шаблонные замены реализуются через регулярные выражения или шаблоны строк. Например, типичный случай - замена конструкции LIMIT N OFFSET M на OFFSET M ROWS FETCH NEXT N ROWS ONLY, характерную для стандартного SQL:2008, или TOP N в диалекте SQL Server. Такие замены возможны при чётко определённой структуре запроса и отсутствии вложенных конструкций. Часто они применяются к «плоскому» SQL без вложенных подзапросов и выражений, так как регулярные выражения плохо справляются с рекурсивными или сильно контекстно-зависимыми конструкциями.

На следующем уровне rule-based системы используют контекстные деревья и заранее заданный набор правил трансформации. Эти правила описываются в виде шаблонов AST (абстрактных синтаксических деревьев) и указывают, какие узлы и в каком контексте следует заменять. Например, правило может гласить: если в дереве встречается функция IFNULL(x, y), заменить её на COALESCE(x, y) при преобразовании в PostgreSQL. В более продвинутых реализациях возможно учитывать не только структуру запроса, но и типы данных, уровень вложенности, наличие агрегатных функций и другие семантические признаки. В таких системах используется понятие «правила сопоставления и замены» (pattern matching + rewriting).

Среди известных реализаций такого подхода - SQLGlot, Apache Calcite, ZetaSQL от Google. Например, SQLGlot использует большое количество грамматик для разных диалектов и содержит встроенные механизмы трансформации между ними, управляемые через словарь правил. Apache Calcite идёт дальше, предлагая унифицированную платформу для SQL-парсинга, оптимизации и генерации кода, включая rule-based ревайтер (переписыватель) запросов. Он позволяет строить цепочки преобразований, направленных как на трансляцию диалекта, так и на оптимизацию плана выполнения.

Однако несмотря на удобство, rule-based подходы страдают от хрупкости. Их главный недостаток - неспособность обобщать логику вне заранее определённых случаев. В реальных сценариях, когда запросы нестандартизованы, динамически генерируются или содержат многоуровневую вложенность, система правил либо выдаёт ошибку, либо генерирует неверный результат. Кроме того, при добавлении новых диалектов или расширений правил быстро растёт сложность сопровождения: для каждой пары «источник-назначение» может по-

требоваться отдельный набор правил, что ведёт к экспоненциальному росту конфигураций.

1.5 Пример шаблонной и rule-based трансформации

Для иллюстрации рассмотрим следующий SQL-запрос в синтаксисе SQL Server:

Листинг 7: SQL Server: исходный запрос с TOP

```
SELECT TOP 5 name, age
FROM employees
WHERE department = 'Sales'
ORDER BY age DESC;
```

В рамках шаблонной замены он может быть преобразован в стандарт SQL:2008 следующим образом:

Листинг 8: Standard SQL: FETCH FIRST

```
SELECT name, age
FROM employees
WHERE department = 'Sales'
ORDER BY age DESC
FETCH FIRST 5 ROWS ONLY;
```

Такое преобразование может быть выполнено на основе регулярного выражения или строкового шаблона, например:

```
regex: ^SELECT TOP (\d+) (.+) FROM
replace: SELECT \2 FROM ... FETCH FIRST \1 ROWS ONLY
```

Однако этот метод не учитывает возможное наличие вложенных подзапросов или подстановки ‘TOP‘ в более сложном контексте.

Rule-based подход, использующий AST (Abstract Syntax Tree), формализует правило как:

```
IF node.type == 'Top' AND dialect == 'sqlserver':
    REWRITE to LimitFetch(count=node.value, order=node.order)
```

В случае PostgreSQL такое правило может выдавать результат:

Листинг 9: PostgreSQL: Эквивалентный запрос с LIMIT

```
SELECT name, age
FROM employees
WHERE department = 'Sales'
ORDER BY age DESC
LIMIT 5;
```

Таким образом, rule-based подход позволяет выразить более точные условия преобразования, включая сопоставление узлов AST и генерацию корректного синтаксиса в зависимости от целевого диалекта.

Тем не менее, оба подхода плохо масштабируются на случаи с вложенными SELECT, оконными функциями или диалект-специфическими функциями. Например, конструкция `IFNULL()` в MySQL требует отдельного правила замены на `COALESCE()` в PostgreSQL, и таких правил может быть сотни.

Чтобы компенсировать эти недостатки, гибридные подходы совмещают rule-based трансформации с полноценным парсингом и эвристиками. В таких системах запрос сначала разбирается в AST, затем преобразуется на основе правил, но при этом применяется семантический анализ: например, проверка типов, раскрытие алиасов, устранение дублирующих условий. Также возможно использование графовых моделей: представление запроса в виде направленного ациклического графа (DAG), где узлы соответствуют операциям (фильтрация, проекция, соединение), а рёбра - потокам данных. Такой подход характерен для компиляторов запросов в распределённых СУБД и позволяет не только переводить SQL, но и оптимизировать его с учётом характеристик целевой системы - например, заменить последовательные фильтры на объединённые, переставить JOIN-операции, устранить избыточные столбцы.

В ряде случаев подключаются и эвристики - например, автоматическое определение наилучшей конструкции JOIN (INNER, LEFT, USING), выбор индексов, переименование колонок для устранения конфликтов, динамическая адаптация под ограничение длины идентификаторов и т.д. Такие методы требуют сложной логики и знания особенностей каждой СУБД, но значительно улучшают итоговое качество миграции.

В целом, подходы на основе шаблонов и правил являются необходимой частью любого SQL-транслятора, но достигают максимальной эффективности только в сочетании с глубокой структурной обработкой и оптимизацией. Их роль особенно велика при необходимости поддерживать большое число простых преобразований, но при этом важно понимать их ограничения и избегать применения в критичных участках без дополнительной проверки.

Применение методов машинного обучения, особенно нейросетевых моделей, в задаче генерации и преобразования SQL-запросов представляет собой относительно новое и активно развивающееся направление. Оно особенно актуально в условиях растущего числа диалектов SQL и усложнения запросов, используемых в аналитике, BI-системах и ETL-платформах. В отличие от традиционных rule-based или парсинг-ориентированных методов, модели машинного обучения стремятся абстрагироваться от строгих грамматик и использовать статистические закономерности в синтаксисе и семантике запросов.

Одним из первых подходов, применённых в этой области, стали Seq2Seq-модели (sequence-to-sequence), основанные на рекуррентных нейронных сетях (RNN) и их модификациях, таких как LSTM (Long Short-Term Memory)[13]. Идея заключалась в том, чтобы представить SQL-запрос как последовательность токенов, которую можно «перевести» в другую последовательность - то есть запрос в другом диалекте - аналогично машинному переводу. Такие модели обучаются на парах запросов в разных диалектах или на паре «естественный язык - SQL» (в задачах типа text-to-SQL).

Однако применение Seq2Seq-моделей оказалось ограниченным. Основная проблема - недостаточная способность моделей улавливать долгосрочные зависимости, особенно при наличии вложенных подзапросов, сложных JOIN-ов, оконных функций или CASE-конструкций[13]. Даже с использованием LSTM-механизмов модели часто теряли важный контекст и генерировали синтаксически или семантически некорректные запросы. Особенно плохо такие архитектуры масштабировались к большим SQL-базам с разнообразными диалектами.

Значительный прогресс в этом направлении стал возможен с появлением механизмов внимания (attention) и генеративных трансформерных архитектур. Механизм внимания позволяет модели фокусироваться на релевантных частях входной последовательности при генерации каждого токена выходной последо-

вательности, что критически важно для SQL, где, например, WHERE-условия зависят от ранее объявленных таблиц, а ORDER BY - от выходной схемы. Один из первых примеров такого подхода - SQLNet, в котором использовался attention-механизм поверх RNN для генерации SQL из текстового описания запроса.

На следующем этапе внимание было положено в основу всей архитектуры - в частности, Transformer-моделей, на которых основаны современные языковые модели вроде BERT, GPT, Codex и других. Эти модели обладают способностью учитывать весь контекст запроса одновременно и эффективно обучаться на больших корпусах кода. В результате стало возможным не только преобразовывать SQL-запросы между диалектами, но и генерировать SQL по описанию на естественном языке, выполнять автоматическое исправление запросов, предсказание следующего оператора, а также рекомендации по оптимизации.

Особую перспективу представляет fine-tuning предобученных языковых моделей. Например, адаптация моделей типа BERT для задач понимания структуры SQL - как при классификации подзапросов, так и при определении семантики функций и агрегаций. Модели, подобные Codex (наследник GPT, обученный на коде), демонстрируют способности к трансляции между диалектами SQL при наличии соответствующего обучения. Они могут распознавать, что, например, NVL в Oracle эквивалентен COALESCE в PostgreSQL, или автоматически заменить LIMIT/OFFSET на FETCH FIRST. При этом генерация может учитывать не только синтаксис, но и бизнес-контекст, схемы таблиц и предыдущую историю запросов.

Тем не менее, у таких подходов есть ограничения. Во-первых, качество генерации сильно зависит от обучающего корпуса: модели могут выдавать корректные конструкции в популярных диалектах (например, PostgreSQL или MySQL), но ошибаться в более нишевых или нестандартизованных (ClickHouse, Teradata и др.). Во-вторых, генеративные модели склонны к «галлюцинациям» - то есть созданию синтаксически правильных, но логически некорректных запросов, особенно если исходные данные неоднозначны. Кроме того, они требуют значительных вычислительных ресурсов как на стадии обучения, так и в инференсе.

В целом, машинное обучение и генеративные модели открывают новые горизонты для автоматизации SQL-перевода и понимания, особенно в сочета-

нии с традиционными методами - парсингом, rule-based трансформациями и эвристиками. Их развитие позволяет строить более интеллектуальные системы миграции и интеграции данных, адаптированные к реальному разнообразию СУБД и запросов.

1.6 Архитектура трансформеров и механизм внимания

Архитектура трансформеров (Transformers) представляет собой один из наиболее значительных прорывов в области обработки естественного языка и генерации текста, включая такие специфические задачи, как синтаксический разбор, машинный перевод, генерация SQL и кода. Впервые предложенная в статье “Attention Is All You Need” [3], она заменяет традиционные рекуррентные и сверточные механизмы полностью вниманием (attention), позволяя моделям эффективно обрабатывать весь входной контекст параллельно. Это делает трансформеры особенно подходящими для работы с длинными последовательностями и сложными зависимостями — именно такими, какие встречаются в SQL-запросах и других формальных языках.

Ключевая идея архитектуры трансформера заключается в использовании механизма самовнимания (self-attention), который позволяет каждому токену «обращаться» ко всем другим токенам входной последовательности и взвешивать их значимость при формировании своего представления. В отличие от RNN, где токены обрабатываются последовательно, трансформер способен анализировать все токены одновременно, что значительно ускоряет обучение и делает возможным масштабирование на большие датасеты.

Трансформеры состоят из двух основных блоков: энкодера и декодера, каждый из которых включает в себя слои self-attention и позиционно-зависимой обработки. Энкодер принимает входную последовательность (например, SQL-запрос или текст) и преобразует её в набор векторов признаков — контекстуализированных представлений токенов. Декодер затем использует эти представления, чтобы поэтапно генерировать выходную последовательность (например, запрос в другом диалекте SQL), опираясь на механизмы внимания как к ранее сгенерированным токенам, так и к исходному контексту энкодера.

Позиционная информация, утраченная при отказе от рекуррентности,

восстанавливается с помощью позиционного кодирования — добавления к каждому токену вектора, который отражает его положение в последовательности. Это позволяет модели понимать порядок слов или операций, что критично для синтаксически чувствительных языков, таких как SQL.

Модели на базе трансформеров, такие как BERT, GPT, T5 и Codex, отличаются своими стратегиями обучения: одни фокусируются на предсказании пропущенных токенов (BERT), другие — на автогрессивной генерации текста (GPT), третьи обучаются в формате «вход → выход» как универсальные преобразователи (T5). Эти архитектурные вариации позволяют адаптировать трансформеры под широкий спектр задач, от анализа естественного языка до генерации формализованных SQL-запросов и перевода между диалектами СУБД.

Одним из ключевых компонентов архитектуры трансформеров является механизм масштабированного скалярного внимания. Его основная задача — определить, какие элементы входной последовательности наиболее информативны для формирования контекстуального представления каждого отдельного элемента. В отличие от рекуррентных моделей, где взаимодействие между токенами ограничено соседством и порядком, внимание обеспечивает прямой доступ к любому другому элементу последовательности, что особенно важно при работе с языками программирования и запросами на SQL.

Пусть заданы три матрицы: запросов $Q \in \mathbb{R}^{n \times d_k}$, ключей $K \in \mathbb{R}^{n \times d_k}$ и значений $V \in \mathbb{R}^{n \times d_v}$, полученные путём линейных преобразований входных эмбеддингов. Здесь n — длина входной последовательности, d_k и d_v — размерности признакового пространства для ключей и значений соответственно.

Механизм внимания вычисляет матрицу взаимного соответствия между запросами и ключами с использованием скалярного произведения, масштабированного на корень из размерности пространства признаков:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

Матричное произведение $QK^\top$ формирует матрицу размерности $n \times n$, в которой каждый элемент отражает степень релевантности между i-м запросом и j-м ключом. Деление на $\sqrt{d_k}$ необходимо для стабилизации градиентов при больших значениях d_k, что позволяет избежать чрезмерно больших значений в

выходе softmax.

После нормализации с помощью функции softmax по строкам формируются вероятностные веса, отражающие значимость каждого элемента последовательности относительно текущего запроса. Эти веса используются для взвешенного суммирования соответствующих векторов из V, в результате чего получается новое представление для каждого токена, обогащённое глобальным контекстом всей последовательности.

Таким образом, attention-механизм реализует отображение:

$$(Q, K, V) \mapsto \mathbb{R}^{n \times d_v},$$

обеспечивая для каждого входного токена контекстуализованное векторное представление, учитывающее как локальные, так и дальние зависимости. Такая структура делает возможным параллельную обработку последовательности и эффективное извлечение семантических связей между её элементами.

Следует отметить, что в архитектуре трансформера используется так называемое многоголовое внимание (*multi-head attention*), при котором несколько независимых экземпляров механизма внимания обучаются параллельно, что позволяет модели захватывать различные типы зависимостей и аспектов входной последовательности. Однако базовой единицей является именно масштабированное скалярное внимание, которое и определяет фундаментальную природу информационного взаимодействия в трансформерах.

1.7 Механизм многоголового внимания

Механизм многоголового внимания (*Multi-Head Attention*) расширяет идею масштабированного скалярного внимания, позволяя модели одновременно учитывать различные аспекты взаимосвязей между элементами последовательности [3]. Вместо одного механизма внимания, многоголовое внимание использует h параллельных «голов», каждая из которых представляет собой независимый экземпляр механизма Scaled Dot-Product Attention с собственными параметрами.

Для каждой головы $i \in \{1, \ldots, h\}$ определяются отдельные линейные преобразования входного эмбеддинга:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V,$$

где $X \in \mathbb{R}^{n \times d_{\text{model}}}$ — входная последовательность эмбеддингов, $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ — обучаемые матрицы преобразования для i-й головы, а $d_k = d_v = d_{\text{model}}/h$ — размерность каждой подголовы.

Затем для каждой головы i независимо вычисляется результат внимания:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right) V_i.$$

После получения h выходов всех голов они конкатенируются вдоль признакового измерения и пропускаются через финальное линейное преобразование:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O,$$

где $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ — обучаемая матрица проекции, восстанавливающая исходную размерность модели d_{model}.

Основное достоинство многоголового внимания заключается в том, что каждая голова может обучаться выявлять различные шаблоны, зависимости и уровни абстракции: одни головы могут фокусироваться на локальных синтаксических связях, другие — на глобальных семантических отношениях. Такое разбиение позволяет модели извлекать более разнообразную и выразительную информацию из последовательности без увеличения времени выполнения, так как все головы могут быть обработаны параллельно.

1.8 Позиционное кодирование

Несмотря на выразительность и эффективность механизма внимания, как одиночного, так и многоголового, он по своей природе не учитывает информацию о порядке элементов во входной последовательности[3]. В отличие от рекуррентных и сверточных моделей, архитектура трансформера не обладает встроенной последовательной структурой, поскольку операции внимания симметричны относительно перестановки входов. Это означает, что без дополнительной информации модель не может различить, какой токен идёт раньше или

позже в последовательности. Однако порядок элементов критически важен как в естественном языке, так и в формализованных синтаксах, включая SQL.

Для устранения этой проблемы в архитектуру трансформеров вводится механизм *позиционного кодирования* (*Positional Encoding*), который позволяет явно добавлять информацию о позиции каждого токена в последовательности. Эта информация объединяется с исходными эмбеддингами токенов до подачи на вход слоям внимания.

Наиболее распространённой формой позиционного кодирования является *синусоидальное позиционное кодирование*, предложенное в оригинальной работе “Attention Is All You Need”. Оно основано на использовании тригонометрических функций с разными частотами и имеет следующую форму:

$$\mathrm{PE}_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad \mathrm{PE}_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$

где pos — позиция токена в последовательности, i — индекс размерности признакового пространства, а d_{model} — общее число признаков эмбеддинга. Таким образом, каждый токен получает вектор размерности d_{model}, значения которого периодически зависят от позиции, что позволяет трансформеру извлекать относительную и абсолютную позиционную информацию.

Позиционные векторы добавляются к эмбеддингам слов перед первым слоем внимания:

$$X_{\text{input}} = X_{\text{embedding}} + \mathrm{PE}.$$

Достоинством синусоидального кодирования является то, что оно не требует обучения и позволяет модели экстраполировать на более длинные последовательности, чем те, которые были представлены в обучающем наборе[3]. Кроме того, за счёт периодичности функций модель может легко вычислять относительные смещения между позициями, что может быть полезно, например, при обработке вложенных SQL-запросов, где важны иерархические зависимости.

В альтернативных архитектурах возможны и другие способы позиционного кодирования, включая обучаемые позиционные эмбеддинги, но синусоидальный метод остаётся предпочтительным в задачах, где требуется интерпре-

тируемость и устойчивость при работе с переменными длинами входа.

Таким образом, позиционное кодирование играет фундаментальную роль в архитектуре трансформеров, компенсируя отсутствие априорной информации о порядке токенов и обеспечивая необходимую контекстуализацию на уровне синтаксической структуры.

1.9 Резидуальные соединения и слойная нормализация

После того как позиционное кодирование добавляется к эмбеддингам, полученные векторы подаются в стек трансформерных блоков, каждый из которых состоит из двух ключевых подкомпонентов: механизма многоголового внимания и позиционно-зависимого полносвязного слоя. Для стабилизации обучения и эффективной передачи градиентов в глубокой архитектуре, каждый из этих компонентов оборачивается *резидуальным соединением* с последующей *слойной нормализацией*.

Резидуальные соединения (*residual connections*) были впервые предложены в контексте глубоких нейронных сетей для изображения, но нашли широкое применение и в трансформерах. Основная идея заключается в том, чтобы сохранить исходное представление x и добавить к нему результат нелинейного преобразования $f(x)$, полученного в текущем блоке:

$$\text{Res}(x) = x + f(x).$$

Таким образом, информация, прошедшая через механизм внимания или полносвязный слой, агрегируется с первоначальным представлением токена. Это предотвращает затухание градиентов и позволяет модели сохранить как локальную, так и модифицированную информацию одновременно. В контексте трансформера функция $f(x)$ соответствует либо выходу слоя внимания, либо выходу полносвязной сети в каждом подблоке.

После применения резидуального соединения производится *слойная нормализация* (*Layer Normalization*), которая стабилизирует значения активаций, приводя их к общему масштабу. В отличие от пакетной нормализации, слойная нормализация нормализует активации вдоль измерения признаков для каждого токена независимо:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta,$$

где μ и σ — среднее и стандартное отклонение по признаковому измерению, а γ и β — обучаемые параметры масштабирования и смещения. Такая нормализация ускоряет сходимость модели и повышает её устойчивость к колебаниям масштабов активаций в процессе обучения.

Итоговая структура трансформерного блока для одного токена включает в себя последовательность из:

- многоголового внимания;
- резидуального соединения;
- слойной нормализации;
- полносвязной feed-forward сети;
- ещё одного резидуального соединения;
- финальной слойной нормализации.

Каждая из этих операций выполняется над всей последовательностью параллельно, что обеспечивает высокую вычислительную эффективность трансформеров[3]. Сочетание резидуальных путей и нормализации позволяет глубокой архитектуре сохранять и обобщать информацию на разных уровнях абстракции без потери градиентов и ухудшения представлений.

Таким образом, резидуальные соединения и слойная нормализация являются критически важными элементами, обеспечивающими как стабильность обучения, так и способность модели к извлечению сложных структурных зависимостей — в том числе в задачах преобразования и генерации SQL-запросов.

1.10 Позиционно-зависимая feed-forward сеть и структура энкодера

После механизма внимания и последующих операций нормализации и резидуального соединения каждый блок энкодера включает в себя компонент

позиционно-зависимой полносвязной сети (Position-wise Feed-Forward Network, FFN), которая применяется независимо к каждому токену последовательности. В отличие от рекуррентных структур, где взаимодействие между токенами реализуется на уровне вычислений, трансформер реализует межтокенную зависимость исключительно через внимание, а полносвязная сеть служит для индивидуального преобразования каждого токена.

Структура FFN одинакова для всех позиций и задаётся двухслойной нейронной сетью с нелинейной функцией активации, чаще всего — ReLU:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2,$$

где $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ — обучаемые матрицы весов, b_1, b_2 — смещения, а $d_{\text{ff}} \gg d_{\text{model}}$ — внутренняя размерность (часто $d_{\text{ff}} = 2048$, $d_{\text{model}} = 512$). FFN применяется ко всем позициям независимо, но с одними и теми же параметрами, то есть операция полностью параллелизуема.

Таким образом, каждый трансформерный блок энкодера состоит из двух основных подблоков: многоголового внимания и feed-forward сети. Оба подблока оборачиваются в резидуальные соединения и слойную нормализацию. Формально, выход одного слоя энкодера можно выразить рекурсивно через входной вектор x следующим образом:

$$z_1 = \text{LayerNorm}(x + \text{MultiHead}(x)),$$
$$z_2 = \text{LayerNorm}(z_1 + \text{FFN}(z_1)).$$

Для повышения представительной мощности используется стек таких идентичных блоков, называемый *многослойным энкодером* (*stacked encoder*). Типичное количество слоёв варьируется от 6 до 12, в зависимости от размера модели и сложности задачи. Каждый следующий слой получает на вход выход предыдущего:

$$x^{(l+1)} = \text{EncoderLayer}(x^{(l)}),$$

где $x^{(0)}$ — эмбеддинги токенов с добавленным позиционным кодированием.

Итоговая архитектура энкодера представляет собой стек из N слоёв, каж-

дый из которых выполняет серию преобразований:

$$\text{Encoder}(X) = \text{EncoderLayer}_N(\cdots \text{EncoderLayer}_2(\text{EncoderLayer}_1(X))).$$

На выходе энкодера формируется последовательность векторов размерности d_{model}, каждый из которых содержит контекстуализированное представление соответствующего токена, обогащённое глобальной информацией о всей входной последовательности. Эти представления затем могут быть переданы либо в декодер (в случае задач генерации), либо напрямую в классификатор или другую архитектуру, в зависимости от цели.

Таким образом, энкодер трансформера представляет собой мощную архитектуру, сочетающую параллельную обработку, глобальное внимание и нелинейные преобразования, что делает его особенно эффективным в задачах синтаксического и семантического анализа языков, включая перевод SQL-диалектов.

1.11 Архитектура декодера трансформера

В отличие от энкодера, задача которого заключается в извлечении глобальных признаков из входной последовательности, декодер трансформера отвечает за поэтапную генерацию выходной последовательности[3]. Это особенно актуально для задач машинного перевода и трансформации SQL-диалектов, где требуется пошагово порождать корректный и семантически эквивалентный запрос на другом диалекте.

Каждый слой декодера включает три подблока: (1) механизм многоголового *маскированного* самовнимания, (2) механизм многоголового внимания к выходам энкодера, и (3) позиционно-зависимую feed-forward сеть. Все три подблока, как и в энкодере, обернуты в резидуальные соединения с последующей слойной нормализацией.

Пусть y — частично сгенерированная выходная последовательность. Маскированный механизм самовнимания (Masked Multi-Head Self-Attention) используется для того, чтобы предотвратить «заглядывание вперёд»: в момент генерации токена y_t модель должна иметь доступ только к $y_1, \ldots, y_{t-1}$. Это достигается с помощью маски нижнего треугольника, которая зануляет внимания к будущим

позициям:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V,$$

где M — матрица маски с $-\infty$ на недопустимых позициях (будущих токенах) и нулями на разрешённых.

Второй блок внимания в декодере — *межмодальный* или *source-target attention*, который позволяет декодеру обращаться к выходам энкодера, то есть к представлениям входной последовательности. Здесь ключи и значения берутся из энкодера, а запросы — из выхода предыдущего слоя декодера:

$$\text{CrossAttention}(Q^{\text{dec}}, K^{\text{enc}}, V^{\text{enc}}).$$

Этот механизм критичен для задач перевода, так как он позволяет учитывать семантику исходного выражения при генерации каждого токена.

Третьим компонентом каждого слоя декодера является позиционно - зависимая feed-forward сеть, идентичная по структуре тому, что используется в энкодере. Она применяется отдельно к каждому токену выходной последовательности.

Таким образом, один слой декодера реализует следующий набор операций:

$$\begin{aligned} z_1 &= \text{LayerNorm}(y + \text{MaskedMultiHead}(y)), \\ z_2 &= \text{LayerNorm}(z_1 + \text{CrossAttention}(z_1, x^{\text{enc}})), \\ z_3 &= \text{LayerNorm}(z_2 + \text{FFN}(z_2)). \end{aligned}$$

Как и энкодер, декодер состоит из стека N идентичных слоёв. После прохождения через весь стек декодера полученные представления передаются в линейный выходной слой с функцией softmax, преобразующей их в распределение вероятностей по словарю:

$$P(y_t \mid y_{<t}, x) = \text{softmax}(W_o z_3 + b_o),$$

где W_o и b_o — обучаемые параметры выходного слоя.

Важно отметить, что в отличие от энкодера, декодер требует автоагрегирования (аворегрессии) при генерации: на каждом шаге выходной токен подаётся

обратно в декодер для генерации следующего. Эта особенность делает декодер менее параллелизуемым во время инференса, но сохраняет корректную последовательную зависимость между токенами.

Таким образом, архитектура декодера трансформера объединяет в себе внимательное изучение как уже сгенерированной последовательности, так и входного контекста, полученного из энкодера, обеспечивая высокое качество и гибкость генерации — в том числе в задачах перевода и интерпретации SQL-запросов между различными диалектами.

1.12 Полная архитектура трансформера и её применение к задаче трансляции SQL

Модель трансформера представляет собой симметричную архитектуру, состоящую из двух основных компонентов: энкодера и декодера, каждый из которых включает стек из N идентичных слоёв. Энкодер преобразует входную последовательность в набор контекстуализированных векторов признаков, содержащих как лексическую, так и синтаксическую информацию. Декодер, в свою очередь, на основе этих векторов и ранее сгенерированных выходных токенов, пошагово строит целевую последовательность.

Полный путь обработки входной последовательности X и генерации выходной Y можно представить следующим образом:

:

$$X_{\text{embed}} = XW_e + PE,$$
$$x^{(0)} = X_{\text{embed}},$$
$$x^{(l+1)} = \text{EncoderLayer}(x^{(l)}), \quad l = 0, \ldots, N-1,$$

:

$$Y_{\text{embed}} = YW_d + PE,$$
$$y^{(0)} = Y_{\text{embed}},$$
$$\text{for } t = 1, \ldots, T:$$
$$y_t = \text{Decoder}(y_{<t}, x^{(N)}),$$
$$\hat{y}_t = \arg\max \text{softmax}(W_o y_t + b_o),$$

где W_e, W_d — матрицы эмбеддингов, PE — позиционное кодирование, $x^{(N)}$ — выход энкодера, $\hat{y}_t$ — предсказанный токен.

В контексте задачи трансляции SQL-диалектов такая архитектура оказывается особенно продуктивной. Входной SQL-запрос, написанный на одном диалекте (например, MySQL), интерпретируется энкодером как синтаксически и семантически насыщенная структура. Механизм внимания позволяет учесть не только линейный порядок токенов, но и их функциональные зависимости — такие как вложенные подзапросы, оконные функции и агрегаты.

Декодер, используя выход энкодера, поэтапно генерирует эквивалентный запрос на целевом диалекте (например, PostgreSQL), учитывая контекст и грамматические различия. Маскированное внимание позволяет сохранить строгую автоагрессию, а межмодальное внимание даёт возможность точно отразить смысл исходного выражения. Такой подход не требует жёстко заданных правил замены или ручного соответствия между конструкциями диалектов, что даёт значительное преимущество по сравнению с rule-based системами.

Кроме того, возможность предобучения трансформеров на больших корпусах SQL-запросов и последующего дообучения на конкретных парах диалектов делает их пригодными для применения даже в условиях ограниченного количества размеченных данных. Это превращает трансформеры в универсаль-

ный инструмент для автоматизации междиалектного преобразования, а также для задач интерпретации, исправления и оптимизации SQL-запросов в целом.

Глава 2. Разработка методов обработки SQL запросов на основе LLM

2.1 Использование LLM для работы с кодом и SQL

Хотя архитектура трансформера разрабатывалась в контексте обработки естественного языка, её универсальность и способность к моделированию длинных зависимостей делают её особенно эффективной и в задачах синтаксически строго определённых формальных языков, таких как SQL и языки программирования (например, Python, Java, C++). Адаптация трансформеров к этим задачам требует учёта особенностей грамматической строгости, семантической точности и ограниченности допустимых конструкций.

В случае SQL или языков программирования входной и выходной текст можно трактовать как последовательности токенов, полученные через лексический анализ (tokenization). В отличие от обычного текста, здесь токены имеют фиксированное множество типов (ключевые слова, идентификаторы, скобки, операторы и т.д.), а грамматическая корректность выходной последовательности критична. Поэтому одна из стратегий адаптации — замена byte-pair encoding (BPE) на специализированный токенизатор, построенный на основе формальной грамматики языка.

1. Архитектурные адаптации. В рамках энкодер-декодерной архитектуры трансформера необходимо учитывать специфику целевого формального языка. Например, для SQL характерны вложенные структуры, выражения с переменной арностью и строгая типизация. Поэтому возможно внедрение структурного внимания (structured attention), позволяющего учитывать древовидную природу синтаксического представления запроса.

Также используются модификации, такие как:

- *Tree-based positional encoding*, где позиционные коды формируются не по линейному порядку, а по узлам абстрактного синтаксического дерева.

- *Type-aware embeddings*, при которых каждому токену сопоставляется не только эмбеддинг, но и его типовая метка (например, `ColumnName`, `Literal`, `FunctionCall`).

2. Обучение на параллельных корпусах. Для задач трансляции между языками или диалектами, таких как MySQL $\rightarrow$ PostgreSQL или SQL $\rightarrow$ LINQ, используются параллельные корпуса запросов. При их отсутствии возможно обучение с использованием искусственно сгенерированных данных, правил трансформации, а также через weak supervision, где корректность выходного кода проверяется с помощью симуляции выполнения или парсера целевого языка.

3. Контроль корректности и исполнение. В отличие от перевода на естественный язык, результат трансляции должен быть не только синтаксически, но и семантически валиден. Одной из стратегий обеспечения корректности является включение дополнительных механизмов:

- *Execution-guided decoding* — модель проверяет возможные предсказания с помощью внешнего SQL-интерпретатора;
- *Grammar-constrained decoding* — на этапе генерации используются ограничения, заданные контекстно-свободной грамматикой;
- *Pointer networks* — механизм для копирования идентификаторов и имён таблиц из входного запроса, повышающий точность и согласованность.

Таким образом, трансформеры не просто применимы к задаче перевода между языками программирования, но при соответствующих адаптациях становятся мощным инструментом для генерации, трансформации, оптимизации и интерпретации программного кода. В частности, в области SQL это открывает путь к созданию универсальных движков миграции запросов, интеллектуальных систем автокоррекции и средств визуального проектирования, которые способны оперировать абстракциями, а не только строками текста.

2.2 Специализированные архитектуры для генерации и трансляции SQL-запросов: пример SQLNet и ограничения применения готовых моделей

Одной из первых специализированных архитектур, предназначенных для генерации SQL-запросов из текста или трансляции между диалектами, стала модель *SQLNet*. Она была предложена как альтернатива seq2seq-моделям с декодером общего назначения и ориентирована на строго формализованную структуру SQL.

В отличие от классических трансформеров, SQLNet использует схему пошаговой генерации компонентов запроса: сначала определяется тип операции (`SELECT`, `WHERE`, `GROUP BY`), затем — аргументы этих операций (имена колонок, условия, агрегаты). Такой подход основан на декомпозиции задачи генерации SQL на несколько подзадач с ограниченным словарём и предопределённой грамматикой.

Основные компоненты архитектуры SQLNet включают:

- *Sketch-based decoding* — модель не генерирует SQL-последовательность напрямую, а заполняет «шаблон» запроса, используя классификаторы и копирующие механизмы;
- *Column-attention* — внедрение механизма внимания, чувствительного к структуре входной таблицы и её схемы;
- *Execution-guided training* — стратегия обучения, при которой промежуточные гипотезы проверяются путём исполнения и отбрасываются, если они дают неверный результат.

Несмотря на архитектурные преимущества таких моделей, их использование на практике сталкивается с рядом ограничений:

Во-первых, многие существующие модели, в том числе SQLNet, PICARD, IRNet, RatSQL и даже крупные языковые модели вроде Codex или GPT-3.5/4, не всегда имеют открытые веса или лицензии, позволяющие производственное внедрение. Некоторые доступны лишь через API (с ограничениями скорости,

конфиденциальности и стоимости), другие вовсе не имеют общедоступной реализации.

Во-вторых, такие модели, как правило, требуют значительного количества обучающих данных, включая синтаксически корректные пары SQL-запросов. Наличие таких корпусов (например, Spider, WikiSQL) ограничено англоязычными источниками и не охватывает диалектные особенности промышленных СУБД.

Наконец, интеграция готовых моделей в реальные ETL-пайплайны и системы миграции усложняется отсутствием контроля над точностью и корректностью результата. Даже минимальная ошибка в синтаксисе SQL может привести к аварийному завершению задания или, что хуже, к некорректному результату обработки данных.

Таким образом, несмотря на архитектурную зрелость и потенциал специализированных моделей SQL-перевода, их практическое применение требует адаптации, тонкой настройки, дообучения на собственных данных и оценки качества в конкретном контексте применения.

2.3 Семейство моделей Gemini от Google и их применимость к задачам трансляции кода

Одним из наиболее значимых достижений в развитии универсальных моделей для генерации и трансформации кода и языка стало семейство мультимодальных моделей Gemini, разработанное Google DeepMind[5]. Эти модели, включая `Gemini 1, 1.5, 2.0, 7.0`, основаны на архитектуре, сочетающей свойства трансформеров нового поколения с масштабным обучением на объединённых корпусах текстов, программного кода и структурированных данных.

В отличие от узкоспециализированных архитектур (например, SQLNet или PICARD), Gemini представляет собой обобщённую языковую модель, обученную на множестве формальных и естественных языков. Она демонстрирует высокие результаты в задачах:

- трансляции между языками программирования (например, Python $\rightarrow$ Java, или SQL $\rightarrow$ LINQ);
- автогенерации запросов по текстовому описанию (text-to-SQL);

- интерпретации, рефакторинга и объяснения кода;
- анализа и генерации в мультимодальных форматах (текст+таблицы+изображения).

Модель Gemini использует модифицированную архитектуру трансформера с масштабированием вплоть до сотен миллиардов параметров. Одной из отличительных черт является более эффективный механизм смешанного внимания (*mixture-of-experts attention*), позволяющий избирательно активировать подмножество вычислительных блоков в зависимости от контекста[5]. Также сообщается об использовании обучающих стратегий, совмещающих self-supervised learning и reinforcement learning from human feedback (RLHF), что позволяет модели лучше справляться с задачами, требующими точности, таких как генерация SQL.

С практической точки зрения Gemini предлагает доступ через API и интеграцию в экосистему Google (Vertex AI, Colab, Firebase), однако на момент написания статьи большая часть моделей семейства остаётся проприетарной. Лишь ограниченные версии могут быть использованы в открытом доступе или на локальных вычислительных ресурсах.

2.4 Применение моделей Gemini в рамках данного исследования

В ходе выполнения данного исследования была предпринята попытка интеграции языковой модели семейства Gemini для решения задач трансляции SQL-запросов между диалектами различных СУБД. Благодаря универсальности архитектуры и способности модели к генерации синтаксически осмысленного кода, Gemini продемонстрировала потенциал в задачах сопоставления структур входного и выходного запроса, автоматического выбора ключевых операций и синтаксических конструкций, характерных для целевой СУБД.

Однако на практике возникли существенные ограничения, связанные прежде всего с требованиями к вычислительным ресурсам. Полноценное fine-tuning модели потребовало бы доступа к распределённым графическим ускорителям и значительным объёмам аннотированных данных. В условиях ограниченного доступа к ресурсам и отсутствии возможности переобучения всей модели, было

принято решение ограничиться частичным дообучением, сфокусированным на отдельных линейных слоях выходного декодера и адаптационных слоях между эмбеддингами и головами внимания.

Такой подход позволил адаптировать модель к целевой задаче без нарушения общей структуры трансформера и с минимальными затратами ресурсов. Подробное описание методики дообучения, а также архитектурных модификаций, использованных в ходе эксперимента, приводится в следующей главе.

2.5 Общая схема пайплайна решения

В данном разделе описывается архитектура системы автоматизированного перевода SQL-запросов между диалектами различных СУБД с использованием больших языковых моделей (LLM). Предложенное решение опирается на модульный пайплайн, включающий в себя этапы предобработки, трансформации, постобработки и оценки результатов. Такой подход позволяет обеспечить универсальность, расширяемость и применимость в различных сценариях миграции и интеграции данных.

Целью пайплайна является преобразование SQL-запроса, написанного в диалекте одной СУБД (например, PostgreSQL), в эквивалентный запрос в диалекте другой СУБД (например, ClickHouse), с сохранением семантики, корректностью выполнения и минимальной необходимостью ручной корректировки. Ниже представлены ключевые этапы обработки:

1. Формулировка цели и постановка задачи. Система должна обеспечить автоматическое преобразование SQL-запросов между диалектами, учитывая синтаксические и семантические особенности целевой СУБД.

2. Сбор и подготовка данных. Для обучения модели требуется специализированный датасет, содержащий пары SQL-запросов в различных диалектах. Поскольку открытых источников с такой структурой недостаточно, данные были собраны вручную и синтетически сгенерированы. Запросы нормализовались, токенизировались и аннотировались.

3. Выбор и настройка языковой модели. В качестве базовой модели использовались современные трансформеры, предобученные на коде и SQL (на-

пример, Codex, CodeT5+, SQLCoder). Модели дообучались (fine-tuning) с применением техник адаптивного обучения, таких как LoRA или PEFT, что позволяло достичь высокой точности при относительно малом объёме данных.

4. Предобработка входного SQL-запроса. На этом этапе осуществляется очистка, нормализация и структурный анализ запроса. Опционально производится определение исходного диалекта.

5. Генерация запроса в целевом диалекте с помощью LLM. Языковая модель принимает отформатированный запрос и, используя prompt - ориентированный ввод, возвращает эквивалентный запрос в другом диалекте. Для повышения качества использовались шаблоны контекста и обратная связь с валидационным модулем.

6. Постобработка и синтаксическая валидация. Сгенерированный запрос проверяется с использованием синтаксического анализатора (например, SQLGlot или ANTLR), устраняются дублирующие конструкции, корректируются имена таблиц и параметры LIMIT/OFFSET.

7. Оценка качества результата. Качество преобразования оценивается с помощью метрик ROUGE-1, ROUGE-L, Exact Match (EM). Кроме того, применяется семантическая проверка — запуск запросов на тестовых данных и сравнение возвращаемых результатов.

8. Анализ ошибок и обратная связь. В случае несовпадения результатов осуществляется анализ типичных ошибок, например: неправильная агрегация, потеря условий WHERE, некорректный порядок JOIN'ов. Результаты анализа используются для повторного обучения или корректировки prompt-шаблонов.

Предложенный пайплайн позволяет реализовать универсальную и расширяемую архитектуру для автоматизированного перевода SQL-запросов между диалектами различных СУБД. Благодаря использованию трансформерных моделей достигается высокая гибкость и адаптивность системы при работе с синтаксически и семантически разнообразными запросами.

2.6 Ограничения prompt-based подхода и обоснование выбора fine-tuning

В процессе адаптации языковой модели к задаче перевода SQL-запросов между диалектами была рассмотрена возможность применения метода few-shot learning, основанного на использовании контекстных примеров внутри prompt'а без изменения параметров модели. Однако данный подход продемонстрировал ряд существенных ограничений, включая нестабильность результатов, зависимость от порядка представления примеров и невозможность надёжной генерализации на структурно отличающиеся входы. Особенно затруднённой оказалась работа с аналитическими СУБД, такими как ClickHouse, имеющими выраженную специфику синтаксиса.

Вследствие этого было принято решение отказаться от few-shot learning и сосредоточиться исключительно на fine-tuning — дообучении модели на корпусе примеров, составленных по единому шаблону. Такой подход позволил добиться согласованного поведения модели и адаптации к заданной схеме базы данных.

Для генерации обучающих данных использовался промпт следующего вида:

```
<start_of_turn>user
You are an intelligent AI specialized in generating SQL queries.
Your task is to translate MY SQL into Clickhouse.
Please provide the SQL query corresponding to the given prompt
and context:
Prompt:
translate MY SQL into Clickhouse
Context:
Natural language: If a radius is 10, what is the lowest possible
mass?
ClickHouse schema: CREATE TABLE IF NOT EXISTS stars (
    id UInt32,
    Name String,
    Mass Float64,
    Radius Float64
```

```
) ENGINE = MergeTree() ORDER BY (id);
MY SQL query: SELECT MIN(Mass) FROM table_name WHERE Radius = 10;
<end_of_turn>
<start_of_turn>model
```

Такой формат объединяет описание задачи, естественно-языковый контекст, схему целевой таблицы и SQL-запрос в исходном диалекте. Приведение примеров к единому стилю обеспечило стандартизацию входных данных и позволило эффективно использовать fine-tuning для частичного дообучения модели.

2.7 Экспериментальная оценка эффективности метода

Одним из важнейших параметров в работе языковых моделей, влияющим на поведение генерации, является температура (temperature). В задачах вроде машинного перевода, генерации текста или креативных задач температура отвечает за степень случайности при выборе следующего токена. Однако в случае генерации SQL-запросов, где требуется максимально детерминированный и воспроизводимый результат, чрезмерная стохастичность приводит к деградации качества.

В рамках эксперимента была протестирована производительность модели на задаче генерации SQL при различных значениях температуры от 0.5 до 2.0. Для оценки качества использовались метрики ROUGE-1 и ROUGE-L, сопоставляющие сгенерированный SQL-код с эталонным решением.
Как видно из графика, при увеличении температуры наблюдается стремительное снижение качества генерации. При значении temperature = 0.5 достигаются наивысшие показатели (ROUGE-1: 0.7573, ROUGE-L: 0.7266), а при temperature 1.5 — резкое падение метрик до уровня, близкого к случайной генерации.

Это подтверждает, что низкие значения температуры критически важны для задач, требующих точности синтаксиса и семантики, таких как трансляция SQL. Даже незначительное увеличение параметра приводит к тому, что модель начинает "галлюцинировать"— вставлять неподходящие или синтаксически некорректные конструкции.

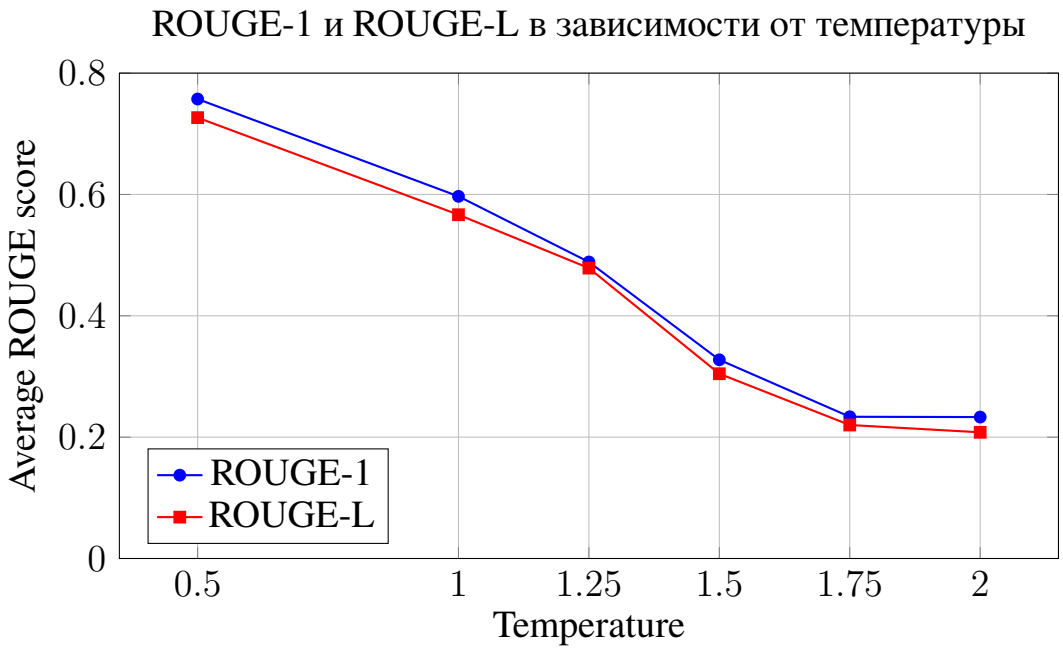

Рис. 1: Зависимость средних значений ROUGE-1 и ROUGE-L от параметра температуры

Для задач автоматического преобразования SQL-запросов между диалектами необходимо использовать низкие значения температуры (0.3–0.5), чтобы сохранить воспроизводимость, точность и корректность синтаксиса. В более широком смысле это подчёркивает, что даже мощные языковые модели должны использоваться в строгом режиме при решении инженерных задач, где критично точное соответствие ожидаемому результату.

Таблица 2: Сравнение качества преобразования различных SQL-диалектов в ClickHouse

№	СУБД	Описание	BLEU / ROUGE-1 / ROUGE-L
1	Postgres → ClickHouse	Преобразование запросов из PostgreSQL в ClickHouse, высокая точность соответствия.	0.232 / 0.8267 / 0.8181
2	MySQL → ClickHouse	Конвертация из MySQL в ClickHouse, лучшие показатели среди всех.	0.3 / 0.8449 / 0.838

№	СУБД	Описание	BLEU / ROUGE-1 / ROUGE-L
3	SQLite → ClickHouse	Перевод запросов из SQLite в ClickHouse, показатели чуть ниже, чем у MySQL.	0.28 / 0.832 / 0.8217
4	Все диалекты → ClickHouse	Объединённая модель, обученная на всех диалектах. Средние результаты.	0.064 / 0.4887 / 0.4697
5	Базовая модель → ClickHouse	Без дополнительного дообучения. Минимальные показатели качества.	0.1213 / 0.1157 / 0.1157

Ниже приведены примеры реальных вопросов на естественном языке, соответствующих им эталонных запросов и ответов, сгенерированных моделью, а также анализ особенностей перевода.

Пример 1:

Who had the high rebound total against golden state?

Листинг 10: Ground Truth (SQL$_{LITE}$)

```
SELECT Rebounds FROM nba_stats WHERE Team = 'Golden State';
```

Листинг 11: Ответ модели (ClickHouse)

```
SELECT High, rebounds FROM table_name WHERE Team = 'Golden State';
```

В данном примере модель сохранила условие фильтрации и имена столбцов, однако возникли избыточные поля в выборке, что требует доработки логики проекции столбцов.

Пример 2:

How many times have there been helps by Damon Stoudamire (13)?

Листинг 12: Ground Truth

```
SELECT COUNT(*) FROM table WHERE High_assists = 'Damon Stoudamire (13)';
```

Листинг 13: Ответ модели

```
SELECT COUNT(player_id) FROM players WHERE assists = 13 AND player_name =
```

Модель успешно распознала необходимость подсчёта количества записей, но сгенерированный запрос содержит незавершённое условие по имени игрока. Также модель использовала дополнительный столбец assists, который присутсвует в схеме бд, но это условие избыточно.

Пример 3:

What is the origin of aircraft in service in 1943 and retired in 1954?

Листинг 14: Ground Truth

```
SELECT National_Origin FROM table WHERE Retired = 1954 AND In_Service = 1943;
```

Листинг 15: Ответ модели

```
SELECT National_Origin FROM my_table WHERE Retired = 1954 AND In_Service
```

Запрос не завершён — отсутствует значение фильтра `In_Service = 1943`. Модель не смогла корректно сгенерировать полное условие.

Пример 4:

What is the Number of dances is 11 and competition finish is larger than 2.0 total

Листинг 16: Ground Truth

```
SELECT total FROM dance_competition WHERE number_of_dances = 11
AND competition_finish > 2.0;
```

Листинг 17: Ответ модели

```
SELECT Total FROM my_table WHERE Number_of_dances = 11 AND Competition_
```

Модель корректно начала запрос, правильно интерпретировала фильтрацию по Number__dances = 11, однако обрезала запрос, не дописав второе условие. Это указывает на недостаточную устойчивость к длинным условиям и возможно на нехватку обучающих примеров с двойными условиями.

Пример 5: *What were the points for when there were 30 tries against?*

Листинг 18: Ground Truth

```
SELECT Points, for_column FROM ClickHouse_Table
WHERE Tries = 30 AND against_column = 30;
```

Листинг 19: Ответ модели

```
POINTS = SELECT points FROM points WHERE tries = 30;
```

Выражение начинается с POINTS = SELECT, что нарушает SQL-формат. Не хватает второго условия (against_column = 30). Повторяется имя таблицы и поля: points FROM points. Модель в данном случае ошиблась и по синтаксису, и по содержанию.

2.8 Разработка pipeline

В рамках данной работы был реализован полный пайплайн, позволяющий обрабатывать SQL-запросы на входе, генерировать их представление в другом диалекте с помощью LLM, валидировать результат и возвращать готовый запрос.

Разработка пайплайна велась с учётом модульности и возможности гибкой адаптации под различные языковые модели и диалекты СУБД. Архитектура пайплайна отражает основные этапы жизненного цикла трансляции запроса: от текстовой предобработки до финальной оценки генерации. Каждый из этапов был реализован в виде отдельной функции или модуля, что позволило не только проводить поэтапную отладку, но и заменять компоненты (например, использовать локальные или облачные модели).

Ключевые компоненты пайплайна:

1. Модуль предобработки запроса выполняет очистку SQL-выражений от комментариев, нормализацию пробелов, приведение ключевых слов к верхнему регистру, а идентификаторов — к нижнему (по необходимости). Это обеспечивает стандартизированный ввод для модели и упрощает генерацию предсказуемого ответа.

2. Генератор на основе LLM отвечает за преобразование запроса из одного диалекта в другой. В данной работе использовалась дообученная языковая модель с архитектурой Transformer, настроенная на работу в режиме

`greedy decoding` (температура = 0, отключено сэмплирование). Генерация останавливается при достижении специального токена окончания , что позволяет избежать бессмысленного продолжения запроса.

3. Извлечение результата из ответа модели реализовано с использованием регулярных выражений, либо путём обрезки сгенерированного текста по шаблону prompt-а. Это позволяет избежать попадания в итог запроса лишней информации, например, отладочной или метаинформации.

4. Постобработка и валидация выполняется с помощью библиотеки `SQLGlot`, которая позволяет проверить корректность синтаксиса сгенерированного SQL-запроса в указанном диалекте (например, ClickHouse или PostgreSQL). Также осуществляется дополнительная нормализация: удаление лишних точек с запятой, пустых строк и пробелов.

Вся логика пайплайна реализована на языке Python, с использованием библиотек `transformers`, `sqlglot`, `nltk`, `evaluate` и `pandas`. Благодаря этому обеспечивается гибкость, возможность запуска как в локальной среде, так и на сервере, а также повторяемость экспериментов.

2.9 Вывод

В ходе выполнения данной дипломной работы была успешно решена задача разработки метода автоматизированного перевода SQL-запросов между диалектами различных систем управления базами данных (СУБД) с использованием больших языковых моделей (LLM). Основная цель — обеспечение сохранения семантики запросов, адаптация к новым СУБД без полного перепроектирования системы и минимизация ручных правок — была частично достигнута.

Заключение

Результат работы

1. Создание специализированного датасета, поскольку подходящих публичных коллекций для задачи преобразования SQL между диалектами не существует. Датасет позволил адаптировать LLM к специфике языков запросов и особенностям целевых СУБД.

2. Выбор и дообучение LLM с использованием технологии Low-Rank Adaptation (LoRA), что позволило эффективно использовать ограниченные вычислительные ресурсы и добиться устойчивой работы модели на базовых конструкциях SQL.

3. Разработка полного pipeline обработки запросов, включающего лексический и синтаксический анализ, трансформацию с сохранением логики, а также проверку корректности и соответствия синтаксису целевого диалекта.

4. Экспериментальная оценка, в ходе которой модель продемонстрировала успешное преобразование типичных запросов с фильтрами, агрегатными функциями и базовыми условиями, особенно в пределах стандартных SQL-конструкций.

Несмотря на положительные результаты, анализ ошибок выявил ограничения.

1. Сложности при работе со сложными запросами, включающими несколько фильтров, вложенные подзапросы и сложные арифметические операции.

2. Нарушения синтаксиса и потери логической целостности при генерации запросов с использованием алиасов и специфических конструкций ClickHouse.

3. Недостаток обучающих данных и ограниченные вычислительные возможности являются главными факторами, сдерживающими качество модели.

Перспективы развития

1. Масштабное дообучение на большом и разнообразном датасете, включающем сложные примеры и различные диалекты SQL.

2. Использование высокопроизводительных вычислительных платформ (GPU-кластеры) для обучения полноценных LLM без компромиссов.

3. Внедрение контроля синтаксиса и семантики на уровне инференса, включая проверку абстрактного синтаксического дерева (AST) и интеграцию с метаданными целевой СУБД.

4. Интеграция схемы данных и шаблонов типовых запросов конкретных СУБД (например, ClickHouse) для повышения точности и адаптивности преобразований.

В итоге, проведённое исследование подтвердило потенциал использования LLM для задачи автоматического перевода SQL-запросов, даже при ограниченных ресурсах. Полученные результаты создают основу для построения промышленных решений [16] и позволяют рекомендовать данный подход для локальных и образовательных проектов, где масштабные вычислительные мощности недоступны.

Список литературы

[1] Dong X.L., Halevy A.Y. Data Integration with Uncertainty // Proceedings of the VLDB Endowment. 2005.

[2] Klettke M., Rieping J., Winter A. Database migration and the role of schema mapping // Datenbank-Spektrum. 2015. № 15(3). С. 199–208.

[3] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. Attention Is All You Need // Proc. of NeurIPS. 2017.

[4] Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // Proc. of NAACL-HLT. 2019.

[5] Google Research. Gemini: Next-generation multimodal AI models // Google AI Blog. 2023.

[6] Guo D., Pasupat P., Liu J., Chang M. IRNet: Learning Semantic Parsing for Complex NL Queries over Databases // Proceedings of ACL. 2019.

[7] Ху Э. Дж., Шен Й., Уоллис П., Аллен-Цу З., Ли Ю., Ванг Ш., Ванг Л., Чен В. LoRA: низкоранговая адаптация больших языковых моделей // arXiv preprint arXiv:2106.09685. 2021.

[8] PostgreSQL Global Development Group. PostgreSQL Documentation. https://www.postgresql.org/docs/, 2025.

[9] Oracle Corporation. MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/, 2025.

[10] D. Richard Hipp. SQLite Documentation. https://www.sqlite.org/docs.html, 2025.

[11] ClickHouse Inc. ClickHouse Documentation. https://clickhouse.com/docs/, 2025.

[12] Zaharia M., Chowdhury M., Franklin M. J., Shenker S., Stoica I. Spark: Cluster Computing with Working Sets // Proc. of HotCloud. 2010.

[13] Sutskever I., Vinyals O., Le Q. V. Sequence to Sequence Learning with Neural Networks // Advances in Neural Information Processing Systems (NeurIPS). – 2014.

[14] Ахо А. В., Лам М. С., Сети Р., Ульман Д. А. Компиляторы: принципы, технологии и инструменты. — 2-е изд. — М.: Вильямс, 2007. — 944 с.

[15] Dong, X. L., & Halevy, A. A platform for personal information management and integration. 2005.

[16] Копылов Игорь, “Перевод SQL диалекта между разными СУБД” GitHub репозиторий, https://github.com/igorkopylov1/diploma_work, accessed May 19, 2025.

[17] С. К. Морозов, А. А. Богатырёв. Базы данных. Проектирование, реализация и сопровождение. — СПб.: Питер, 2023.