Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Фирсов Михаил Александрович

Экспериментальное исследование рекурсивного оператора в PosDB

Учебная практика

Научный руководитель:
ассистент кафедры ИАС Чернышев Г. А.

Санкт-Петербург
2022

Оглавление

Введение **3**

Постановка задачи **4**

1. Что такое рекурсивные запросы? **5**

1.1. Синтаксис рекурсивных запросов 5

1.2. Семантика рекурсивных запросов 5

1.3. Примеры 6

2. Реализация рекурсивных запросов в PosDB **8**

2.1. Устройство PosDB 8

2.2. Идеи реализации 9

2.3. Кортежная обработка 12

2.4. Обработка позиций 15

2.5. Уже написанные тесты 16

3. Обзор генераторов графов **17**

4. Реализация **18**

4.1. Интегрирование генератора в PosDB 18

4.2. Тесты на корректность 18

5. Тесты на производительность **20**

5.1. Поиск в ширину 20

5.2. Транзитивное замыкание 21

5.3. Выводы 22

Заключение **24**

Список литературы **25**

Введение

Реляционные базы данных хранят информацию в виде таблиц. Поэтому для хранения в них каких-нибудь древовидных структур приходиться использовать специальные атрибуты-ссылки на другие строки этой таблицы.

Иногда в базах данных хранятся иерархические данные. Для их обработки International Organization for Standardization (ISO) в стандарт SQL:1999 добавила специальные рекурсивные запросы. Они позволяют обрабатывать данные SQL таблиц рекурсивно. Задавая некоторое начальное условие, иначе говоря начальные узлы рекурсии, можно получить данные находящиеся в некотором иерархическом подчинении от этих начальных узлов.

В предыдущем семестре были продуманы и реализованы рекурсивные операторы TRecursive и PRecursive [6], позволяющие в дальнейшем реализовать в системе рекурсивные запросы. Но существующий набор тестов, проверяющих корректность работы этой реализации, описывает только небольшие деревья. В нем не было проверки на сравнение с работой аналогичных запросов в других СУБД, не было тестов с большими таблицами, что не придавало уверенности в надежности дальнейшего нагрузочного использования системы.

В настоящей работе мы исправляем эти недостатки: придумываем тесты, проверяющие корректность работы рекурсивных запросов с произвольными графами; убеждаемся в том, что производительность написанной реализации рекурсивных операторов лучше или не хуже производительности промышленных СУБД.

Постановка задачи

Целью настоящей работы является экспериментальное исследование работы ранее реализованных рекурсивных операторов в РСУБД PosDB. Для этого планируется выполнить сравнение производительности рекурсивных запросов в PosDB и в PostgreSQL. Однако, перед этим необходимо обеспечить уверенность в надежности полученных результатов, для чего было принято решение сначала реализовать расширенную проверку на корректность работы исследуемых операторов. Для достижения данной цели были поставлены следующие задачи:

1. Провести анализ существующих генераторов графов и выбрать тот, в котором есть возможность задавать размер графа, степень связности его вершин, а также обладающего командным интерфейсом для его запуска из других программ.

2. Создать набор мини-тестов для алгоритма на графах на корректность с использованием выбранного генератора.

3. Наладить экосистему бенчмаркинга: встроить выбранный генератор в PosDB, наладить с его помощью генерацию данных для PosDB и для PostgreSQL.

4. Выбрать несколько сложных алгоритмов на графах и реализовать их с помощью рекурсивных запросов в PosDB и PostgreSQL.

5. Выполнить замеры, в которых сравнивается производительность PosDB и PostgreSQL.

1. Что такое рекурсивные запросы?

1.1. Синтаксис рекурсивных запросов

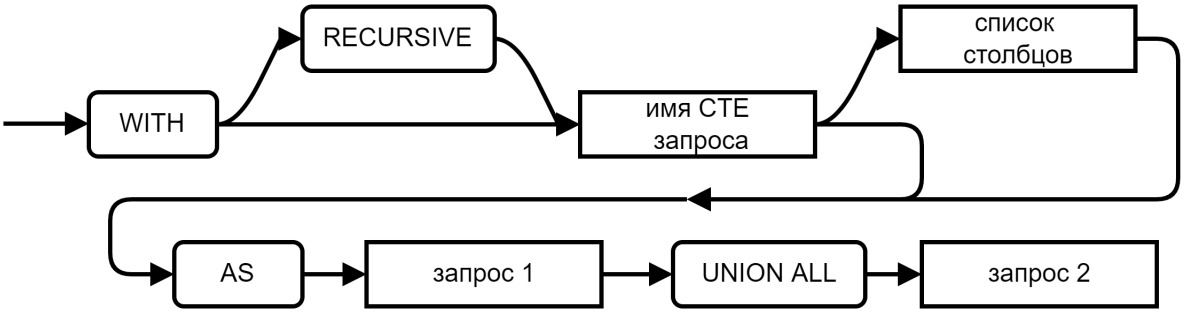

Рис. 1

Общий синтаксис рекурсивных запросов можно представить в виде диаграммы на Рис.1 или описать следующим шаблоном:

```
WITH [RECURSIVE] <имя CTE запроса> [(<список столбцов 0>)]
AS (<запрос 1>)
UNION ALL
<основной запрос 2>
```

1.2. Семантика рекурсивных запросов

- **Common Table Expression (CTE)** — обобщенное табличное выражение, которое можно использовать множество раз в запросе. CTE не сохраняет данные, а создает нечто вроде их временного представления [7].

- **<запрос 1>** является обычным, **<основной запрос 2>** — рекурсивным [5].

- **(<список столбцов 0>)** может быть выведен из списка столбцов запроса 1.

Количество столбцов CTE и рекурсивных элементов должно совпадать. Тип данных столбца в рекурсивном элементе должен совпадать с типом

данных соответствующего столбца в CTE. Рекурсивными могут быть только **монотонные запросы**. Если для любых входных наборов V_1, V_2, где $V_1 \subset V_2$, результат запроса на V_1 является подмножеством результата запроса на V_2, то такой запрос называется монотонным. Монотонность позволяет не пересчитывать результат рекурсивной части запроса на строках, полученных ранее, а просто добавлять в результат запроса данные, полученные подстановкой в CTE только строк, полученных на последнем шаге рекурсии. По этой причине, следующие конструкции не могут использоваться в рекурсивных запросах [5]:

- Агрегация
- Подзапрос, использующий рекурсивные запросы
- Разница множеств

1.3. Примеры

Worker_ID	Master_ID
1	NULL
2	5
3	1
4	3
5	4
6	5
7	5

Таблица 1

Worker_ID	Master_ID
1	NULL
3	1
4	3
5	4
2	5

Таблица 2

Пусть дана Таблица 1. Требуется найти всех косвенных подчиненных начальника (работника с Master_ID = NULL) вплоть до работников с неявным подчинения через 3 промежуточных начальников. Следующий запрос это сделает.

```
WITH RECURSIVE person(Worker_ID, Master_ID)
AS SELECT Table1.Worker_ID, Table1.Master_ID FROM
Table1 WHERE Table1.Master_ID = NULL
```

```
UNION ALL
SELECT Table1.Worker_ID, Table1.Master_ID FROM person JOIN Table1
ON Table1.Master_ID = person.Worker_ID
OPTION (MAXRECURSION 4);
```

Здесь person — это временное представление, которое неявно включает все пары (Table1.Worker_ID, Table1.Master_ID), найденные на данном шаге рекурсии. OPTION (MAXRECURSION 4) говорит о том, что нужно провести максимум 4 итерации рекурсивной части запроса. Результатом исполнения запроса станет Таблица 2.

2. Реализация рекурсивных запросов в PosDB

2.1. Устройство PosDB

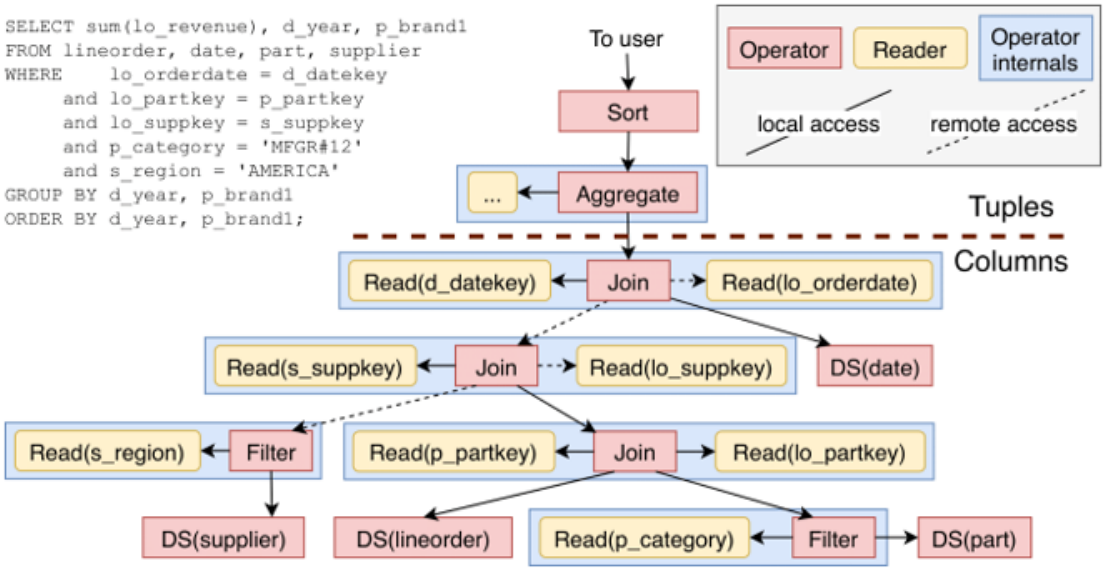

Рис. 2: Пример плана запроса в PosDB

PosDB — распределённая колоночная СУБД [4]. Главные особенности PosDB:

- PosDB хранит каждый атрибут таблицы в отдельном файле. Одна таблица может быть разделена на несколько частей — **партиций**, которые хранятся в разных файлах, возможно, на разных машинах.
- Все запросы на языке SQL, проходя через парсер, переводятся в специальный план запроса. **План запроса**, изображенный на Рис. 2, это диаграмма перехода табличных данных через специальные операторы.
- Исполнение запросов в PosDB основано на модели Volcano с поблочной обработкой: каждый оператор в плане пропускает через себя поток данных и реализует функции open(), next() и close(). next() возвращает блок кортежей или блок позиций. Поэтому большинство операторов делится на позиционные и кортежные.
- Позиционные операторы имеют специальные Reader-ы для чтения отдельных атрибутов таблицы.

- План запроса делится на позиционную и кортежную части. Момент преобразования позиций в кортежи называется **материализацией**.

- Каждый оператор в PosDB имеет два представления: в виде узла логического дерева плана запроса (пространство имён operators) и в виде узла физического дерева плана запроса (пространство имён phys_tree).

2.2. Идеи реализации

Для того, чтобы в дальнейшем реализовать рекурсивные запросы в парсере PosDB было предложено ввести два новых оператора плана запроса. Их иерархию можно представить в виде диаграммы на Рис. 3.

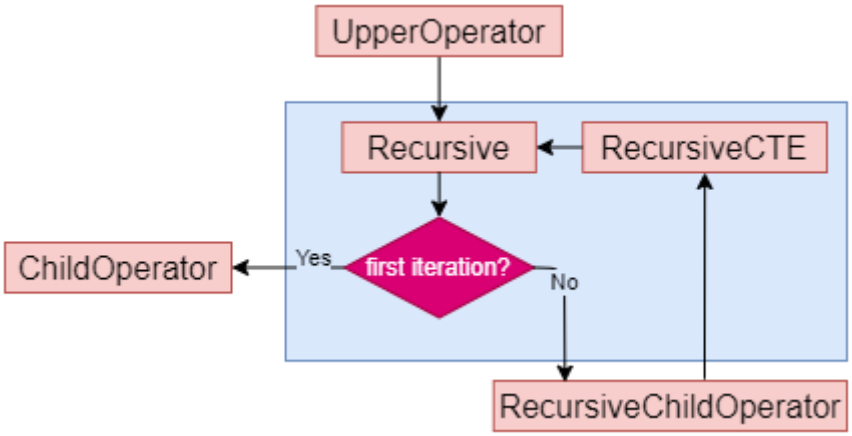

Рис. 3: Примерное изображение плана запроса с использованием Recursive и RecursiveCTE

- Recursive — хранит в себе указатели на RecursiveCTE, ChildOperator и RecursiveChildOperator. ChildOperator используется для нерекурсивной части запроса, с его помощью получим стартовые строки или позиции. Recursive-ChildOperator — обычный оператор, но внутри себя он либо явно, либо косвенно (через несколько промежуточных операторов) получает данные от RecursiveCTE.

- RecursiveCTE — хранит указатель на Recursive, от которого просит новые строки для их передачи методом Next в RecursiveChildOperator.

В PosDB есть два типа блоков: **TupleBlock**, который хранит кортежи, и **PMapBlock**, который хранит позиции. С учетом этого, необходимо ввести две следующих пары операторов:

- **TRecursive** и **TRecursiveCTE** — будут работать только с блоками кортежей.
- **PRecursive и PRecursiveCTE** — будут работать только с блоками позиций.

  Может возникнуть желание использовать гибридный оператор, у которого ChildOperator и RecursiveCTE возвращают блок типа PMapBlock, а RecursiveChildOperator — TupleBlock. При такой реализации, нам придётся кортежи, полученные от RecursiveChildOperator переводить обратно в позиции (это в некоторых случаях невозможно), чтобы использовать для второго и последующих шагов рекурсии. Поэтому использование гибридных операторов для реализации рекурсивных запросов невозможно.

Пример: Пусть задан следующий рекурсивный запрос:

```
WITH RECURSIVE person(Worker_ID, Master_ID)
AS SELECT Table1.Worker_ID, Table1.Master_ID FROM
Table1 WHERE Table1.Master_ID = NULL
UNION ALL
SELECT Table1.Worker_ID, Table1.Master_ID FROM person JOIN Table1
ON Table1.Master_ID = person.Worker_ID
OPTION (MAXRECURSION 4);
```

Дерево этого запроса с использованием введенных структур можно представить с помощью диаграммы на Рис. 4. Здесь:

- Левый оператор Materialize — это ChildOperator, он будет выполнен 1 раз для инициализации стартового набора кортежей.
- Join — это RecursiveChildOperator.

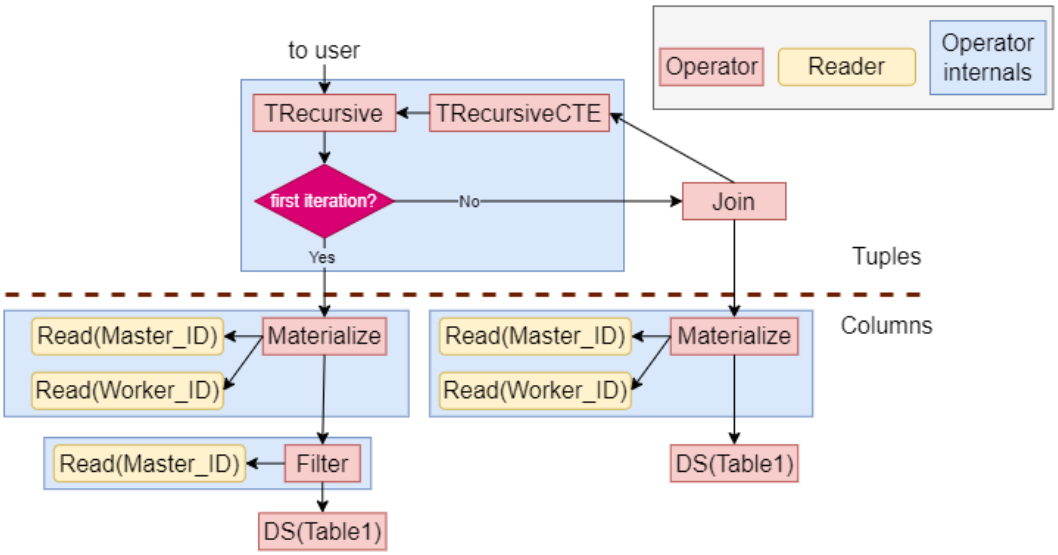

Рис. 4: Дерево запроса с использованием TRecursive и TRecursiveCTE

- Набор блоков кортежей текущего шага рекурсии хранится внутри TRecursive, назовём его curLevel, кроме того TRecursive будет хранить позицию блока в curLevel, который в следующий раз нужно передать TRecursive.

Ход вычислений будет следующим:

1. TRecursive запрашивает блок от левого Materialize, пока они не пусты, и сохраняет их в curLevel.

2. TRecursive передаёт выше все блоки из curLevel, пока не дойдёт до конца curLevel.

3. Чтобы получить блок нового шага рекурсии, TRecursive запросит блок у Join.

4. Join будет запрашивать блоки у TRecursiveCTE и у правого Materialize.

5. TRecursiveCTE будет просить блоки у TRecursive.

6. TRecursive будет, увеличивая внутренний счетчик, передавать TRecursiveCTE в ответ на его запросы блоки из curLevel.

7. Набрав новый блок определенного размера, Join отдаст его TRecursive. Если это непустой блок, то TRecursive сохранит его во временное хранилище nextLevel и перейдёт к шагу 3. Если это пустой блок, то curLevel

заменяется на nextLevel. Если теперь curLevel — это пустой набор блоков, то считаем, что TRecursive закончил свою работу, иначе TRecursive переходит к шагу два.

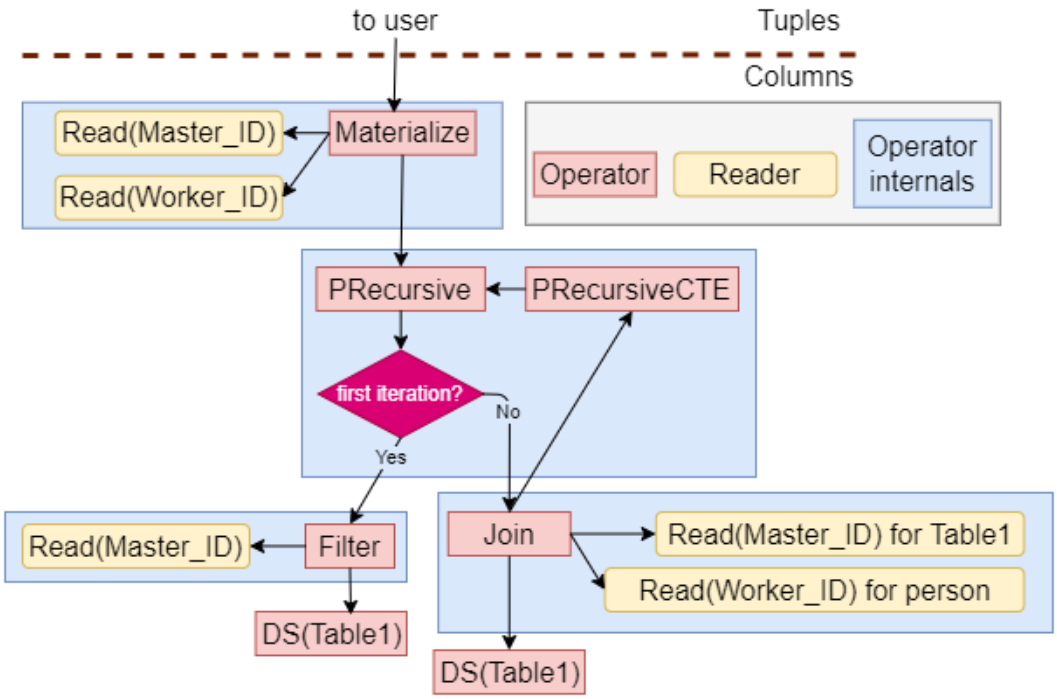

Рис. 5: Дерево запроса с использованием PRecursive и PRecursiveCTE

Вычисления для дерева запроса на рис. 5 с оператором PRecursive будут происходить похожим образом. Важным ограничением будет являться то, что мы можем работать только с позициями одной и той же таблицы, а значит оператор Join обязан возвращать позиции той же таблицы, что и левый оператор Filter. curLevel будет хранить не блоки кортежей, а блоки позиций. В остальном логика работы связки операторов PRecursive и PRecursiveCTE полностью совпадает с логикой TRecursive и TRecursiveCTE.

2.3. Кортежная обработка

Для обработки блоков кортежей были введены физические операторы, изображенные на Рис.6

Опишем поля и методы operators::TRecursive:

- **child** хранит указатель на нерекурсивный оператор, который возвращает кортежи нулевого шага рекурсии.

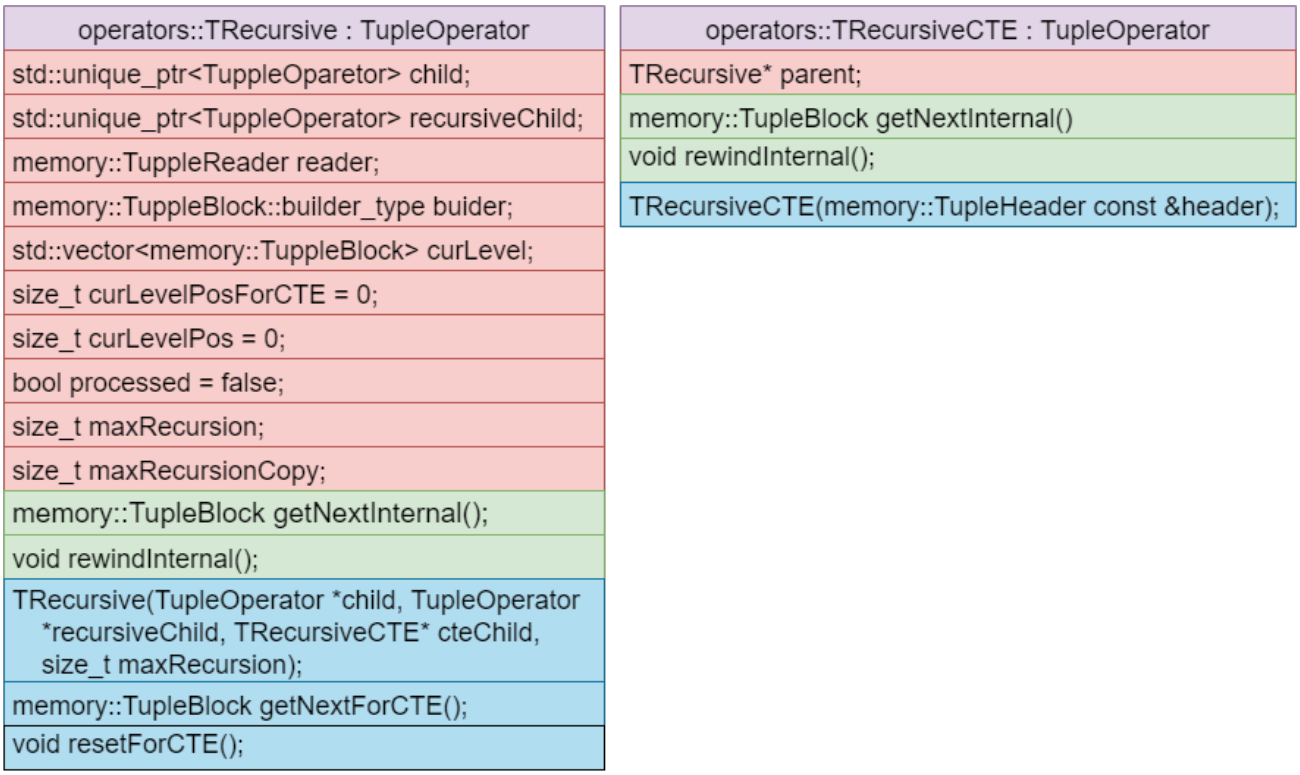

Рис. 6: Структура классов operators::TRecursive, operators::TRecursiveCTE

.

- **recursiveChild** хранит указатель на нерекурсивный оператор, один из потомков которого это operators::TRecursiveCTE.
- **reader** и **builder** нужны для чтения кортежей из специальных кортежных блоков.
- **curLevel** хранит блоки (наборы кортежей) текущего шага рекурсии.
- **curLevelPosForCTE** — позиция блока в curLevel, который нужно будет передать в operators::TRecursiveCTE методом **getNextForCTE**. Если он равен размеру curLevel, то передавать нужно пустой блок.
- **curLevelPos** — позиция блока в curLevel, который следующим нужно будет передать оператору, стоящему над operators::TRecursive, посредством метода **getNextInternal** базового класса **TupleOpeartor**. Если он равен размеру curLevel, то будем запрашивать блоки из recursiveChild, пока они не пусты, после чего обновим curLevel этими новыми блоками. Таким образом, в момент получения этих новых блоков

оператору operators::TRecursiveCTE мы будем передавать ещё старые блоки.

- **processed** — флаг, означающий, что все рекурсивные вызовы закончились. Его значение установим в истинное, если из recursiveChild не пришло ни одного непустого блока.
- **maxRecursionCopy** — максимальное количество итераций рекурсивной части запроса. Если этот параметр равен 0, то количество итераций рекурсии неограниченно.
- **maxRecursion** — счетчик с начальным значением maxRecursionCopy.
- **rewindInternal** обновляет весь оператор TRecursive. Наполняет curLevel за счет блоков, получаемых от child. Он также вызывается в конструкторе operators::TRecursive, для начальной инициализации curLevel.
- **resetForCTE** обнуляет curLevelPosForCTE.

Недостатком такого подхода является отсутствие возможности отследить цикл. Для его отслеживания можно хранить хэш-таблицу всех прошлых кортежей из curLevel. Каждый кортеж каждого блока, полученного от recursiveChild, проверять на существование в этой хэш-таблице. Однако даже такой подход не гарантирует отсутствие циклов. Простой контрпример — это запрос, который в рекурсивной части просто прибавляет 1 к одному из аргументов кортежа. На данный момент было принято решение оставить предотвращение появления циклов на руки пользователя системы.

Оператор operators::TRecursiveCTE делает следующее. В методе getNextInternal возвращает блок, полученный от parent. В методе rewindInternal вызывает метод resetForCTE объекта parent.

Для представления этих операторов физического дерева плана запроса были разработаны операторы логического дерева плана запроса, изображенные на Рис. 7

Для перехода от логического дерева к физическому выполняется вызов метода internalTransform узла логического дерева. Сперва завершается

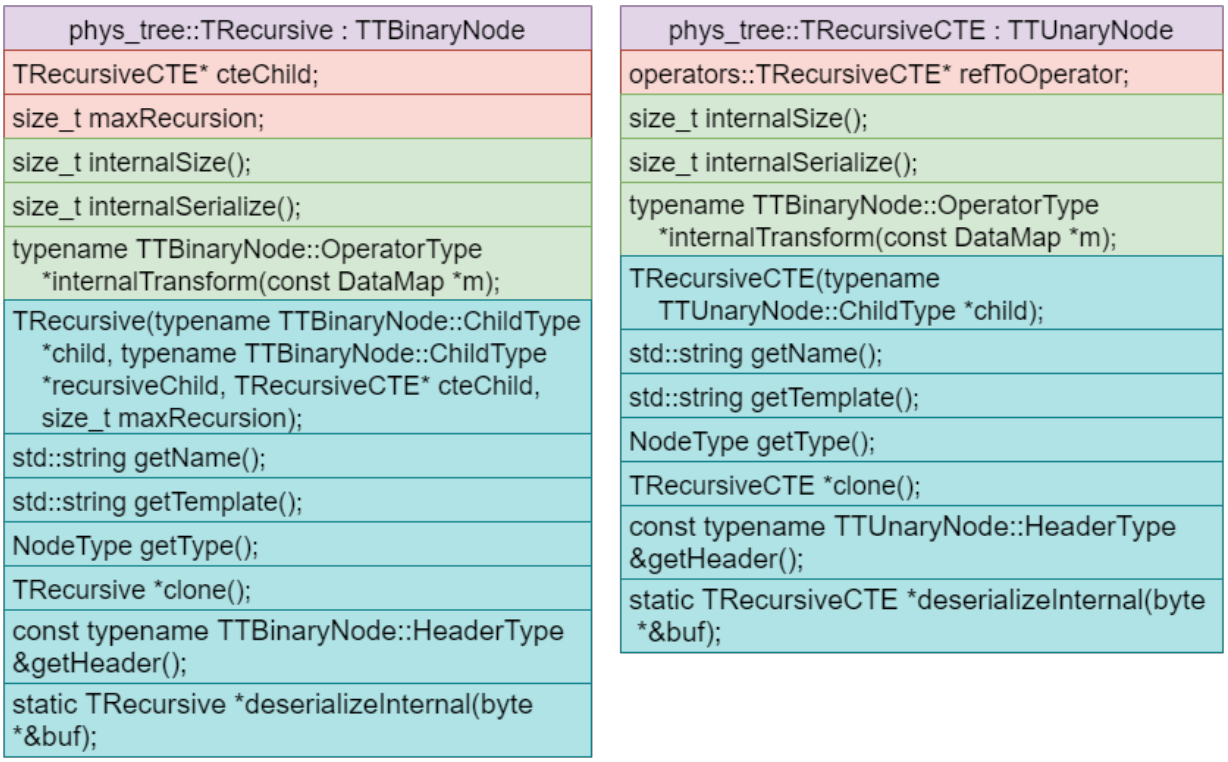

Рис. 7: Структура классов phys_tree::TRecursive, phys_tree::TRecursiveCTE

трансформация для листьев дерева, в затем на основе полученных листьев физического дерева строиться узлы на уровень выше и так далее до вершины дерева. Но нам для построения узла operators::TRecursiveCTE нужен узел operators::TRecursive, который будет построен позже.

Для решения этой проблемы было решено сделать следующее: phys_tree::TRecursive будет содержать указатель на phys_tree::TRecursiveCTE. У **cteChild** метод internalTransform закончит свою работу раньше и результат работы, т.е. указатель на узел физического дерева, сохранит в поле **refToOperator**. В методе internalTransform класса phys_tree::TRecursive мы у поля cteChild возьмем поле refToOperator и передадим в конструктор operators::TRecursive, внутри которого мы полю parent класса operators::TRecursiveCTE зададим значение this, т.е. указатель на нужный нам рекурсивный оператор физического дерева.

2.4. Обработка позиций

Для обработки блоков позиций были введены операторы PRecursive и PRecursiveCTE. Их реализация отличается от реализации кортежных опе-

раторов лишь типом используемых блоков.

2.5. Уже написанные тесты

Для проверки корректности написанных операторов была создана таблица аналогичная таблице из примера 1. Были написаны запросы на получение следующих данных из этой таблицы:

- Получение всех неявных подчиненных корня, листа, и двух узлов (не листьев и не корней) дерева подчинения. Их аналогом на языке SQL будет запрос вида:

```
WITH RECURSIVE person AS
SELECT T1.id, T1.master_id FROM T1 WHERE T1.id = k
# Здесь k - номер стартового узла
UNION ALL
SELECT T9.id, T9.master_id FROM T9 JOIN person
ON person.id = T9.master_id;
```

- Получение трёх строк в нерекурсивной части запроса и просто тождественный запрос в рекурсивной части запроса с использованием параметра MAXRECURSION. Аналогом на языке SQL будет запрос вида:

```
WITH RECURSIVE person AS
SELECT T1.id, T1.master_id FROM T1 WHERE T1.id < 3
UNION ALL
SELECT T1.id, T1.master_id FROM person
MAXRECURSION 4;
```

Были написаны 10 тестов: 5 для TRecursive, TRecursiveCTE; 5 для PRecursive, PRecursiveCTE.

3. Обзор генераторов графов

Таблица 3: Сравнение генераторов графов.

Генератор	наличие командного интерфейса	возможность варьировать размер графа	Возможность варьировать степень связности графа (среднее число рёбер на 1 вершину)
Brite [1]	+	+	+
GT-ITM [1]	+	+	+
GraphRNN [2]	-	+	+
GraphGen [3]	-	+	+

Были проанализированы следующие генераторы графов с открытым исходным кодом. В генераторах GraphGen [3] и GraphRNN [2] модификация входных параметров происходит за счет редактирования одного из исполняемых файлов, в Brite и GT-ITM это происходит через передачу генератору файла конфигурации графа. Brite наследует внутри себя механизм генерации GT-ITM, немного расширяя его [1]. В результате анализа в качестве используемого генератора был выбран Brite.

Генератор Brite работает на основе генератора графов Ваксмана, основанного на модели Эрдёша—Реньи. Он включает в себя характерные для сети характеристики, такие как размещение узлов на плоскости и использование функции вероятности для соединения двух узлов в модели Ваксмана, параметризованной расстоянием, разделяющим их на плоскости [1].

Два главных входных параметра, которые менялись в ходе тестирования, это n — число вершин в графе, m — среднее число рёбер на 1 вершину в графе. Brite генерирует разные характеристики вершин и узлов: положение на плоскости, вес и др. Для тестов были использованы только рёбра, а именно следующие их характеристики: номер ребра, из какой вершины выходит, в какую вершину входит.

Пример работы Brite изображен на Рис.8

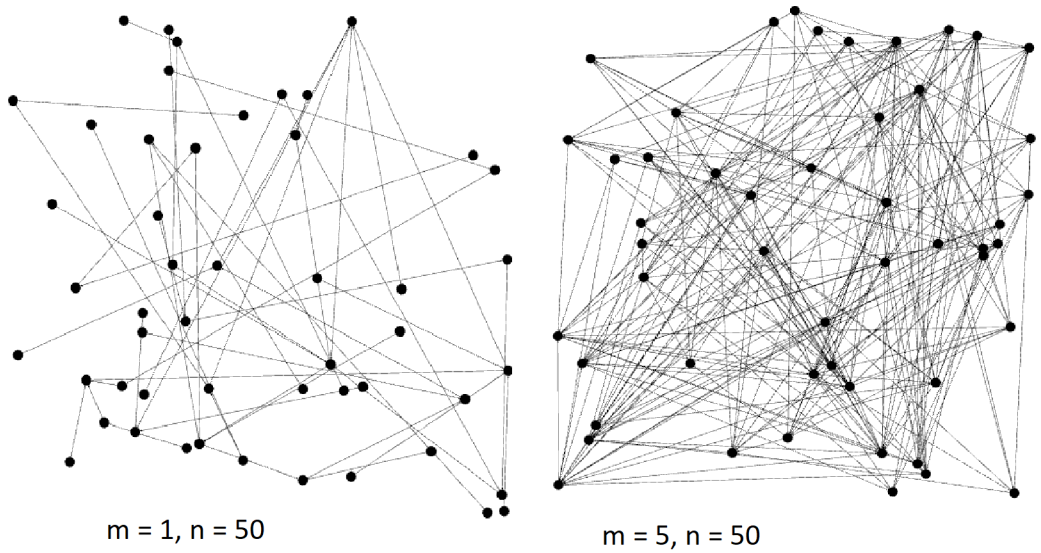

Рис. 8: Пример работы brite.

4. Реализация

4.1. Интегрирование генератора в PosDB

Brite был добавлен как submodule в git-репозиторий. У него есть графический и консольный интерфейс. Через последний на вход программе подаётся файл конфигурации, в котором отражены все входные параметры, среди которых есть n, m. На выходе Brite генерирует Brite-файл, который в последствии переводится в нужный формат под sql запрос к postgresSQL или под формат хранения таблиц в PosDB. Также чтобы не генерировать много раз схожие данные (при тестировании), для данных n, m сохраняется Brite-файл в специальной папке, добавленной в .gitignore.

4.2. Тесты на корректность

Была создана постоянная таблица edges с параметрами $n = 200$, $m = 2$ для тестирования корректности работы рекурсивных запросов с PostgreSQL. К ней были написаны запросы поиска в ширину с повторениями следующего вида:

```
WITH RECURSIVE edges_cte(id, from1, to1, depth) AS
    (SELECT edges.id, edges.from1, edges.to1, 0 FROM edges
```

```
    WHERE edges.from1 = 0
    UNION ALL
        SELECT edges.id, edges.from1, edges.to1, e.depth + 1
        FROM edges JOIN  edges_cte AS e
        ON edges.from1 = e.to1 AND e.depth < maxDepth)
SELECT edges_cte.id, edges_cte.from1, edges_cte.to1 FROM edges_cte;
```

Здесь варьирововались максимальная глубина поиска maxDepth, вид join-соединения: nested loop или hash join — и вид рекурсивного оператора: PRecursive или TRecursive. Для глубины 5 и 15 были сделаны тесты в двух режимах join-а. Для глубины 2, 16 и 20 только в режиме hash join-а. Всего к таблице edges было написано 14 тестов. Все тесты прошли postgres-верификацию, т.е. проверку на равенство с результатами в PostgreSQL.

5. Тесты на производительность

5.1. Поиск в ширину

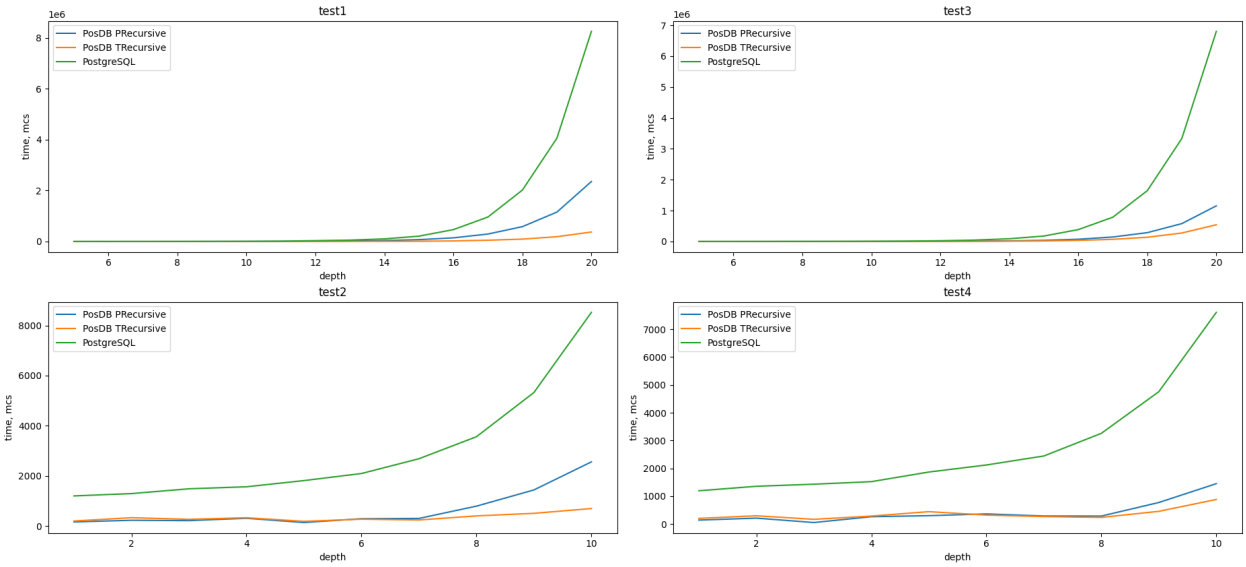

Рис. 9: Поиск в ширину в таблице edges ($n = 200$, $m = 2$).

Были проведены тесты аналогичные тестам на корректность на поиск в ширину с повторениями над той же таблицей edges. Test1 и test2 получают 3 столбца в качестве результата: id, from1, to1. Test3 и test4 — 1 столбец: id. На Рис. 9 изображены результаты их работы. В тестах 2, 4 было вычислено 100 запросов с усреднением времени работы, до запуска 1000 запросов была обработана без учета времени. В тестах 1, 3 — 10 запусков.

Был написан генератор, который для заданных n, m генерирует таблицу edges2 со столбцами id, from, to, name в PosDB и PostgreSQL. С его использованием были проведены тесты производительности при различных значениях n, m для PRecursive, TRecursive в PosDB и PostgreSQL. Тест аналогичен тесту на корректность, описанному выше для значений глубины поиска от 1 до 8 по 20 тестов со случайной стартовой вершиной с усреднением итогового значения времени. Перед измерением времени проходили 10 запросов со случайной стартовой вершиной и случайной глубиной для всех видов запросов: PRecursive, TRecursive в PosDB и PostgreSQL. Результаты можно видеть на Рис. 10.

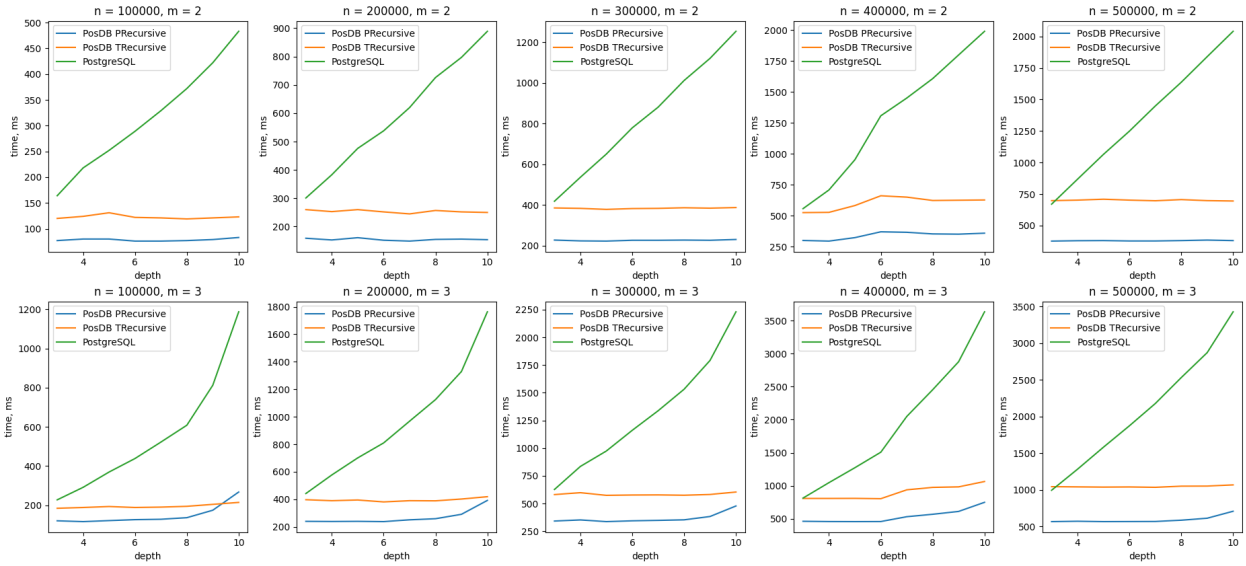

Рис. 10: Поиск в ширину в таблице edges2 при различных значениях n, m.

5.2. Транзитивное замыкание

Также был реализован алгоритм выполняющий первые несколько шагов поиска транзитивного замыкания. С использование PRecursive его реализовать невозможно, поэтому тесты в PosDB проводились только с помощью TRecursive.

```
WITH RECURSIVE edges_cte(from1, to1, depth) AS
    (SELECT edges.from1, edges.to1, 0 FROM edges
    UNION ALL
        SELECT edges.from1, e.to1, e.depth + 1
        FROM edges JOIN  edges_cte AS e
        ON edges.to1 = e.from1 AND e.depth < 4)
SELECT edges_cte.from1, edges_cte.to1 FROM edges_cte;
```

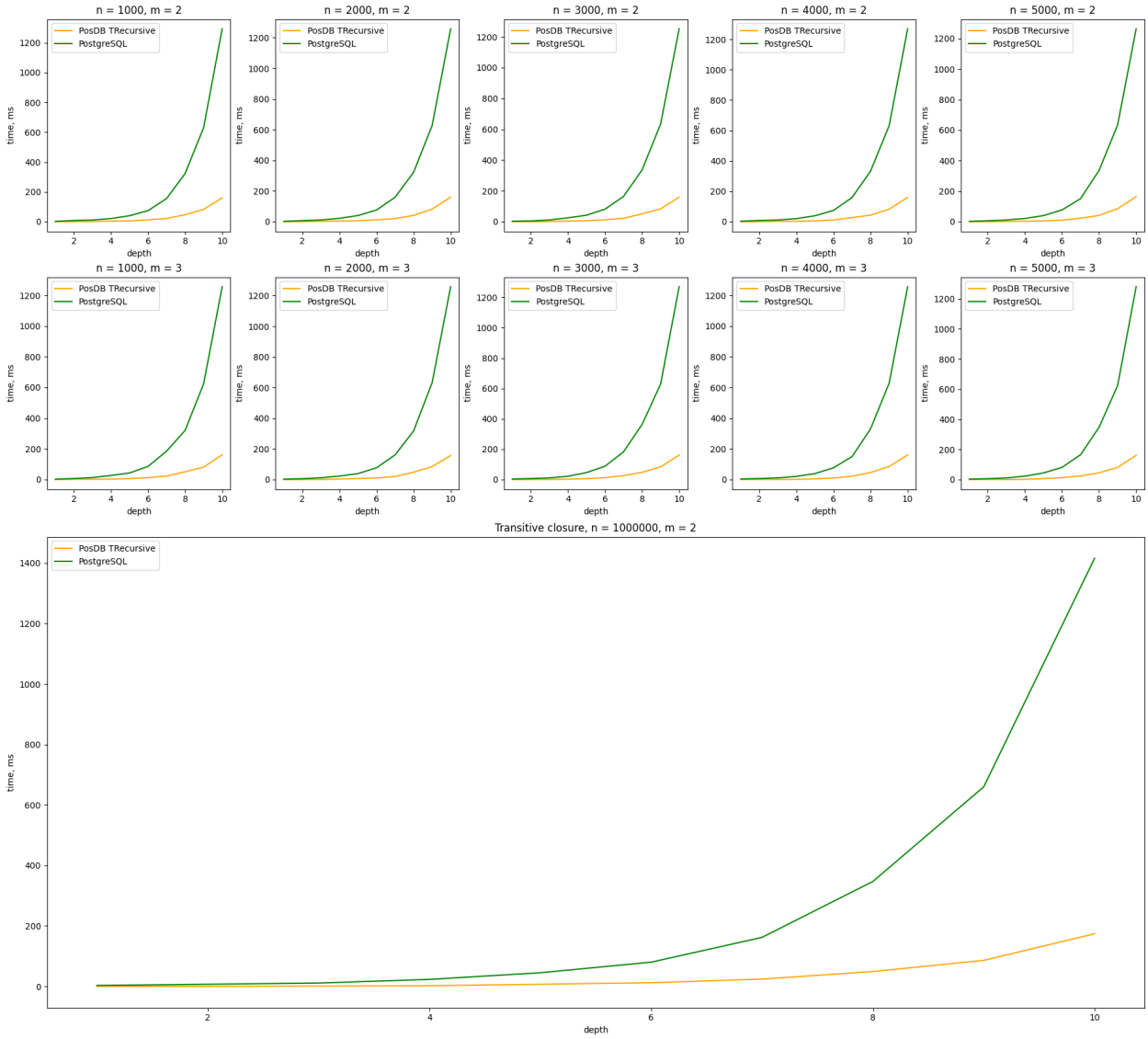

Рис. 11: Построение транзитивного замыкания в таблице edges2 при различных значениях n, m.

5.3. Выводы

На малых данных TRecursive работает быстрее PRecursive. Это может быть связано с тем, что с ростом глубины поиска число получаемых рёбер становится больше числа рёбер в таблице из-за возможных циклов в графе. Это приводит к тому, что материализовать сразу все нужные столбцы таблицы быстрее, чем по несколько раз читать пусть и не все её атрибуты. На больших данных PRecursive выполняет аналогичный запрос за меньшее время, чем TRecursive, т.к. он при непосредственном поиске будет работать только с двумя из трёх атрибутов (from, to), и только когда поймёт какие

строки таблицы нужны, материализует некоторые значения третьего атрибута (id).

Написанные планы запросов для PosDB работают значительно быстрее аналогичных запросов в PostgreSQL. Это может происходить из-за накладных расходов PostgreSQL на поддержку транзакций с их последующим возможным откатом. Были просмотрены планы этих двух видов запросов в PostgreSQL. В рекурсивной части запроса в нём используется Hash Join, точно также как и в построенном плане этих запросов для PosDB.

Заключение

В процессе работы были достигнуты следующие результаты:

- Проанализированы несколько генераторов рекурсивных запросов. Выбран наиболее подходящий из них под искомые параметры.
- Создан набор мини-тестов для алгоритма поиска в ширину на графах на корректность с использованием выбранного генератора.
- Налажена экосистема бенчмаркинга: выбранный генератор встроен в PosDB, с его помощью налажена генерация данных для PosDB и для PostgreSQL.
- Выбраны 2 сложных алгоритма на графах и реализованы с помощью рекурсивных запросов в PosDB и PostgreSQL.
- Выполнены замеры в которых сравнивается производительность PosDB и PostgreSQL.

Список литературы

[1] BRITE: Universal Topology Generation from a User's Perspective. — Online; accessed 2 May 2022. Access mode: https://www.cs.bu.edu/brite/publications/usermanual.pdf.

[2] Jiaxuan You Rex Ying Xiang Ren. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Model. — https://github.com/snap-stanford/GraphRNN. — 2018.

[3] Nikhil Goyal Harsh Vardhan Jain Sayan Ranu. GraphGen: A Scalable Approach to Domain-agnostic Labeled Graph Generation. — https://github.com/idea-iitd/graphgen. — 2020.

[4] Chernishev G. A., Galaktionov V. A., Grigorev V. D., Klyuchikov E. S., and Smirnov K. K. PosDB: An Architecture Overview // Program. Comput. Softw. — 2018. — jan. — Vol. 44, no. 1. — P. 62–74. — Access mode: https://doi.org/10.1134/S0361768818010024.

[5] Silberschatz Abraham, Korth Henry F., and Sudarshan S. Database Systems Concepts. — 7th ed. — McGraw-Hill Higher Education, 2020. — ISBN: 978-0-07-802215-9, 978-1-260-51504-6.

[6] М.А. Фирсов. Рекурсивные запросы в PosDB. — 2021.

[7] Рекурсия в MS SQL. — Online; accessed 29 October 2021. Access mode: https://www.fastreport.ru/ru/blog/show/recursion-mssql/.