**Оглавление**

[1. Постановка задачи 3](#_Toc137126612)

[2. Загрузка данных 3](#_Toc137126613)

[2.1. Объем загруженных данных 3](#_Toc137126614)

[2.2. Время загрузки данных из WB 3](#_Toc137126615)

[2.3. Описание структуры загруженных данных 4](#_Toc137126616)

[2.4. Алгоритм загрузки данных из WB 6](#_Toc137126617)

[3. Проведение экспериментов 7](#_Toc137126618)

[3.1. Алгоритм проведения экспериментов 7](#_Toc137126619)

[3.2. Проведение экспериментов в PostgreSQL с описанием запросов 8](#_Toc137126620)

[3.3. Проведение экспериментов в SQLite с описанием запросов 15](#_Toc137126621)

[3.4. Сравнение времени выполнения запросов в PostgreSQL и SQLite 19](#_Toc137126622)

[4. Заключение 23](#_Toc137126623)

[5. Список используемой литературы 23](#_Toc137126624)

[Приложения 24](#_Toc137126625)

[Приложение 1. DDL сценарий создания таблицы в PostgreSQL и SQLite 24](#_Toc137126626)

[Приложение 2. Программный код загрузки данных 25](#_Toc137126627)

[Приложение 3. Программный код проведения экспериментов 31](#_Toc137126628)

# **1. Постановка задачи**

Создать программу для выгрузки данных из внешнего источника, разобрать и структурировать исходные данные [1], загрузить их в СУБД PostgreSQL [2] и SQLite для проведения исследований [3]. Проанализировать возможности двух СУБД [4] посредством сравнения времени выполнения CRUD запросов к большому объему загруженных данных [5].

# **2. Загрузка данных**

## **2.1. Объем загруженных данных**

В данной работе мы используем 4 базы данных разного размера. Численные значения объема баз данных приведены в Таблице 2.1.

Таблица 2.1 – Объем загруженных данных в СУБД PostgreSQL и SQLite

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование базы данных** | | **Количество строк** | **Объем, МБ** |
| postgres\_1 | SQLite\_1 | 483 439 | 134 |
| postgres\_2 | SQLite\_2 | 966 899 | 268 |
| postgres\_3 | SQLite\_3 | 1 933 819 | 537 |
| postgres\_4 | SQLite\_4 | 3 867 659 | 1076 |

## **2.2. Время загрузки данных из WB**

Время загрузки данных из WB составило 1 час 40 минут. За это время удалось получить 451 147 строк с параметрами товаров маркетплейса, в каждой строке 36 параметров, описывающих конкретный товар (раздел 2.3). Для получения данных использовалось специальное API, которое позволяет получать данные пакетами по 10 000 строк. Это связано с ограничениями, установленными API по количеству данных, которые можно запросить за один раз. Получение, обработка и загрузка каждого блока данных в две СУБД занимает примерно 2 минуты.

## **2.3. Описание структуры загруженных данных**

В результате выгрузки данных из внешнего источника посредством специального API была получена таблица «wb\_products», содержащая 36 столбцов. DDL сценарий создания таблицы в СУБД продемонстрирован в Приложении 1. Фрагмент JSON файла, полученного от API, показан на Рисунке 2.1. Краткое описание каждого столбца приведено в Таблице 2.2.

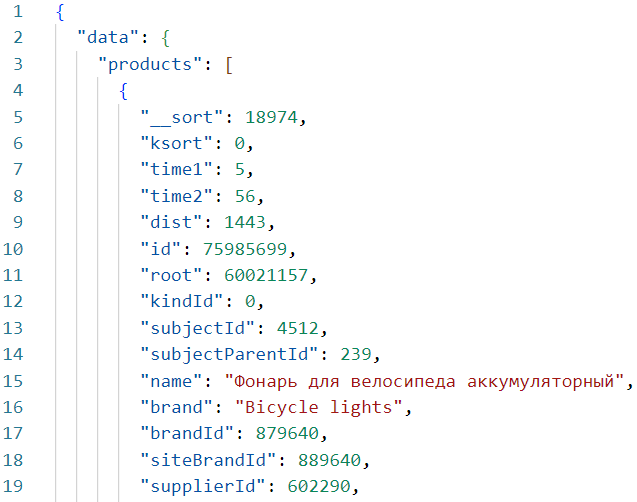


Рисунок 2.1 – Фрагмент полученного JSON файла

Таблица 2.2 – Описание столбцов таблицы «wb\_products»

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Наименование** | **Тип** | **Описание** |
| 1 | sort | INT | Категория товара |
| 2 | ksort | INT | Дополнительная категория товара |
| 3 | time1 | INT | Минимальное время доставки товара |
| 4 | time2 | INT | Максимальное время доставки товара |
| 5 | dist | INT | Расстояние до города товара |
| 6 | id | INT | Уникальный идентификатор товара |
| 7 | root | INT | Идентификатор корневого товара |
| 8 | kindId | INT | Вид идентификатора товара |
| 9 | subjectId | INT | Идентификатор типа товара |
| 10 | subjectidParentId | INT | Идентификатор корневого типа товара |

Продолжение таблицы 2.2.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Наименование** | **Тип** | **Описание** |
| 11 | name | TEXT | Наименование товара |
| 12 | brand | TEXT | Бренд товара |
| 13 | brandId | INT | Идентификатор бренда товара |
| 14 | siteBrandId | INT | Идентификатор сайта бренда товара |
| 15 | supplierId | INT | Идентификатор поставщика товара |
| 16 | sale | INT | Скидка на товар в процентах |
| 17 | priceU | INT | Стоимость товара без скидки в копейках |
| 18 | salePriceU | INT | Стоимость товара со скидкой в копейках |
| 19 | logisticsCost | INT | Затраты на логистику |
| 20 | saleConditions | INT | Условия продажи |
| 21 | pics | INT | Количество фотографий товара |
| 22 | rating | INT | Рейтинг товара на сайте |
| 23 | feedbacks | INT | Количество отзывов на сайте |
| 24 | panelPromoId | INT | Идентификатор категории рекламы |
| 25 | promoTextCat | TEXT | Текст категории рекламы |
| 26 | volume | INT | Количество товара в наличии |
| 27 | diffPrice | TEXT | Наличие разных цен на товар |
| 28 | colorsName | TEXT | Название цвета товара |
| 29 | colorsId | INT | Идентификатор цвета товара |
| 30 | sizeName | TEXT | Название размера товара |
| 31 | sizesOrigName | TEXT | Оригинальное имя размера товара |
| 32 | sizesRank | INT | Ранг размера товара |
| 33 | sizesOptionId | INT | Идентификатор опции размера товара |
| 34 | sizesWh | INT | Габариты товара |
| 35 | sizesSign | TEXT | Обозначение размера товара |
| 36 | link | TEXT | Ссылка на страницу товара |

## **2.4. Алгоритм загрузки данных из WB**

Последовательность действий при выгрузке данных из внешнего источника посредством специального API приведена на Рисунке 2.2. Программный код загрузки данных показан в Приложении 2.

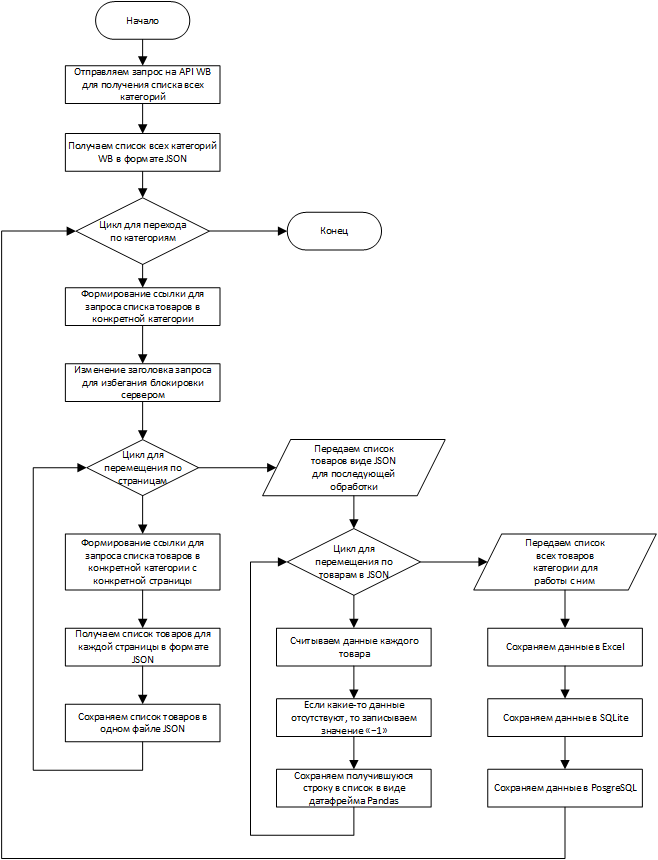


Рисунок 2.2 – Алгоритм загрузки данных из WB

# **3. Проведение экспериментов**

## **3.1. Алгоритм проведения экспериментов**

Последовательность шагов при анализе возможностей двух СУБД посредством выполнения CRUD запросов показана на Рисунке 3.1. Программный код проведения экспериментов представлен в Приложении 3.

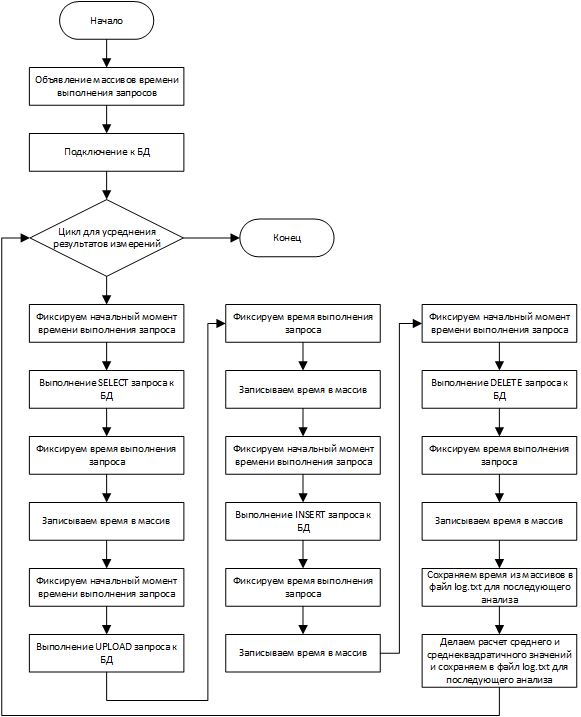


Рисунок 3.1 – Алгоритм проведения экспериментов

## **3.2. Проведение экспериментов в PostgreSQL с описанием запросов**

В рамках эксперимента требуется установить зависимость времени выполнения запросов CRUD к БД разных размеров, для каждого запроса рассчитать среднее и среднеквадратичное отклонение. Используются БД posgres1, posqres2, posqres3, posqres4 (раздел 2.1).

SELECT \* FROM wb\_products

SELECT. Выполняется запрос к базе данных, и выгружаются все строки из таблицы wb\_product. Используя встроенную в pgAdmin функцию Explain, получаем полную информацию по запросу Select. Данная функция позволяем получить графическую визуализацию запроса, аналитику физического плана и некоторые статистические параметры. Результаты выполнения Explain для запроса представлены на Рисунке 3.2.

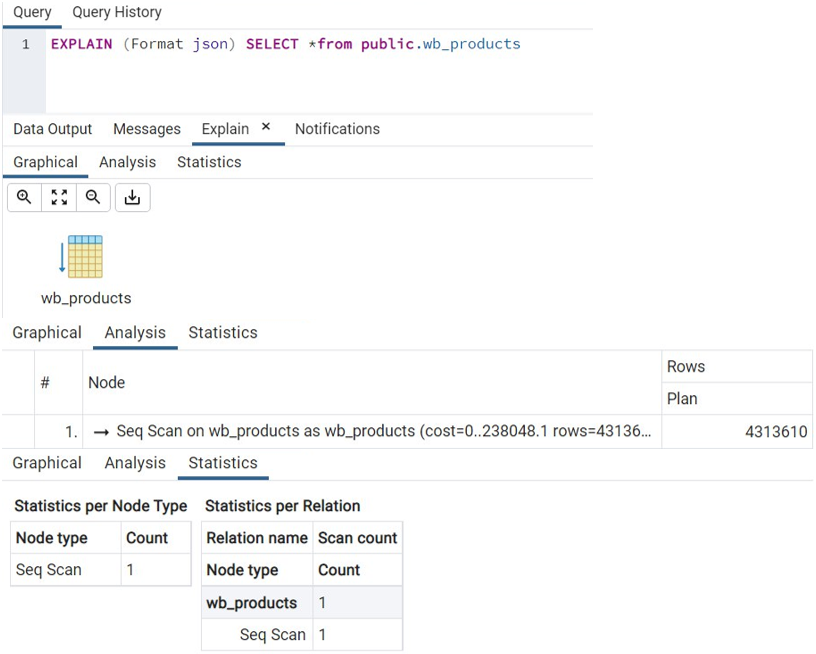


Рисунок 3.2 – Выполнение Explain для запроса Select

UPDATE wb\_products SET dist = 100 WHERE dist = 99

UPDATE. Используя встроенную в pgAdmin Explain функцию, получаем полную информацию по запросу Update. Результаты запроса представлены на Рисунке 3.3.

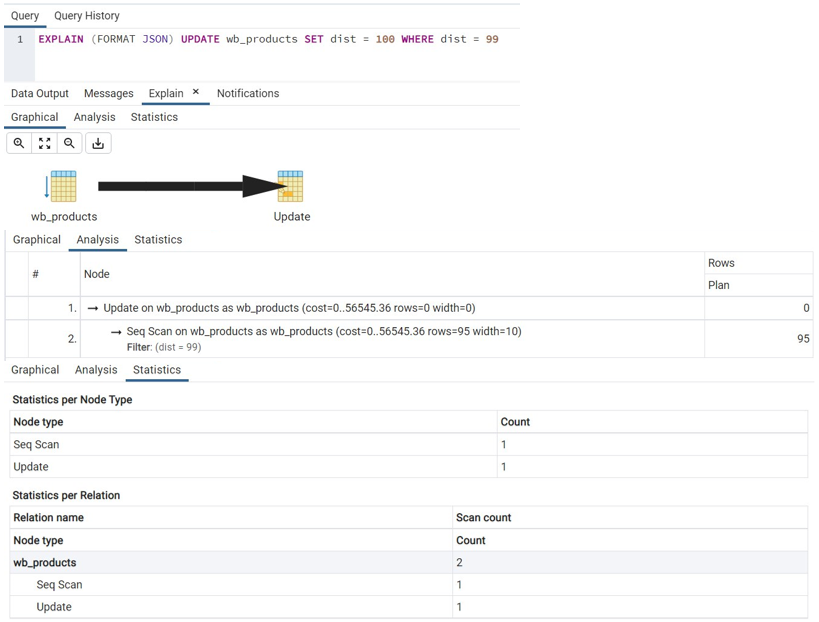


Рисунок 3.3 – Выполнение Explain для запроса Update

INSERT into wb\_products values (110099, 525755, 43, 31, 5566, 51446, 175520, 332, 41, 12, 'Блузка офисная для девочки школьная', 'Camicia', 822269, 832269, 373633, 68, 265000, 84800, 0, 0, 14, 5, 1276, 159147, 'ХИТЫВЕСНЫ', 32, '0', 'белый', 16777215, '42', '42', 166651, 20080, 120762, 'jf7LEXBl5Ze71c5hNa1Z2KnZyc4=', 'https://www.wildberries.ru/catalog/110665146/detail.aspx?targetUrl=BP')

INSERT. Используя встроенную в pgAdmin функцию Explain, получаем полную информацию по запросу Insert. В данном запросе требуется указать данные для всех 36 столбцов таблицы, описанных в Разделе 2.3. Результаты запроса представлены на Рисунке 3.4.

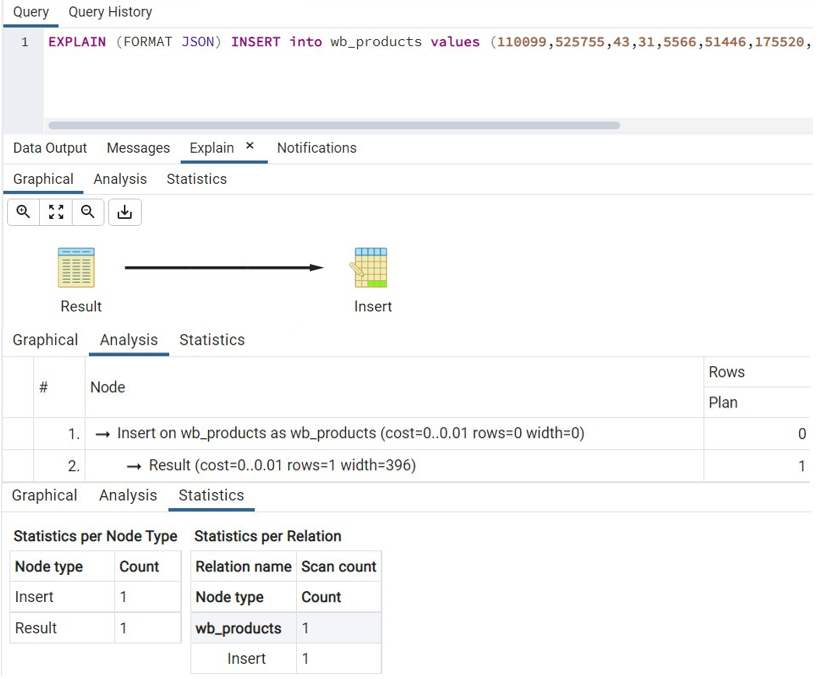


Рисунок 3.4 – Выполнение Explain для запроса Insert

DELETE FROM wb\_products WHERE id=80012708

DELETE. Используя встроенную в pgAdmin функцию Explain, получаем полную информацию по запросу Delete. Результаты запроса представлены на Рисунке 3.5.

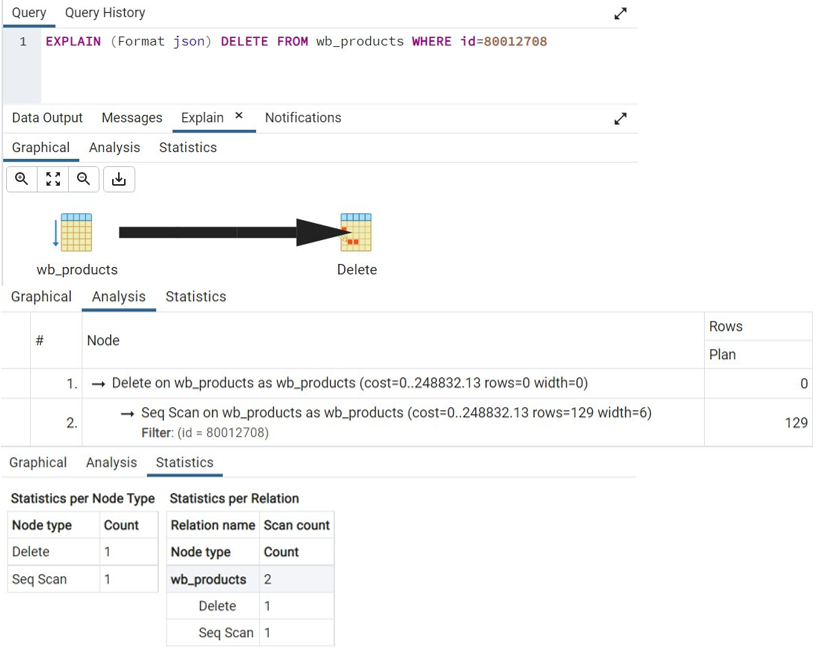


Рисунок 3.5 – Выполнение Explain для запроса Delete

Далее в соответствии с алгоритмом проведения экспериментов, описанном в Разделе 3.1, выполняем 20 CRUD запросов к каждой из двух СУБД для расчета среднего значения и среднеквадратичного отклонения.

Результат выполнения 20 запросов Select к СУБД PostgreSQL показан в Таблице 3.1 и визуализирован на Рисунке 3.6.

Таблица 3.1 – Результат выполнения 20 запросов Select в PostgreSQL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ Select** | **postgres1** | **postgres2** | **postgres3** | **postgres4** |
| 1 | 1,28805542 | 2,596428871 | 4,628715992 | 8,907176495 |
| 2 | 1,188027143 | 2,519071817 | 4,633093357 | 11,77524018 |
| 3 | 1,135818481 | 2,469276905 | 4,685551167 | 9,022065878 |
| 4 | 1,16257906 | 2,667906046 | 4,546559095 | 8,970579863 |
| 5 | 1,358062506 | 2,415505409 | 4,782265186 | 8,91914773 |

Продолжение таблицы 3.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ Select** | **postgres1** | **postgres2** | **postgres3** | **postgres4** |
| 6 | 1,150325537 | 2,400585651 | 4,807919264 | 8,809303284 |
| 7 | 1,127218008 | 2,415766478 | 4,590075731 | 9,069332123 |
| 8 | 1,145205021 | 2,460457325 | 4,522244215 | 9,291076422 |
| 9 | 1,173143148 | 2,605240345 | 4,535361528 | 8,969914436 |
| 10 | 1,138268471 | 2,493346453 | 4,524473906 | 9,092027187 |
| 11 | 1,19404006 | 2,418659687 | 4,509439945 | 9,087626219 |
| 12 | 1,133038521 | 2,404831409 | 4,529818535 | 8,938804626 |
| 13 | 1,139374495 | 2,404574871 | 4,506833315 | 9,220055342 |
| 14 | 1,161930323 | 2,449455738 | 4,473920107 | 9,351839542 |
| 15 | 1,178367376 | 2,406604052 | 4,481939793 | 9,035997391 |
| 16 | 1,178851366 | 2,656934977 | 4,478961229 | 8,801584482 |
| 17 | 1,170704842 | 2,412128448 | 4,470255613 | 9,122145176 |
| 18 | 1,238681555 | 2,452447414 | 4,429638624 | 8,950104952 |
| 19 | 1,177023649 | 2,42353344 | 4,492684126 | 8,88197422 |
| 20 | 1,158051729 | 2,381591797 | 4,447512388 | 8,918031216 |

Рисунок 3.6 – Результат выполнения 20 запросов Select в PostgreSQL

С увеличением количества строк в СУБД PostgreSQL увеличивается и время выполнения запроса Select. Расчеты среднего времени выполнения CRUD запросов показаны в Таблице 3.2 и на Рисунке 3.7, а расчеты среднеквадратичного отклонения времени выполнения запросов продемонстрированы в Таблице 3.3 и на Рисунке 3.8.

Как описано ранее в Разделе 2.1:

* количество строк в БД postgres\_1 – 483 439;
* количество строк в БД postgres\_2 – 966 899;
* количество строк в БД postgres\_3 – 1 933 819;
* количество строк в БД postgres\_4 – 3 867 659.

Таблица 3.2 – Среднее время выполнения CRUD запросов в PostgreSQL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **postgres1** | **postgres2** | **postgres3** | **postgres4** |
| **SELECT** | 1,179838336 | 2,472717357 | 4,553863156 | 9,156701338 |
| **UPDATE** | 0,250711727 | 0,575242329 | 0,896314692 | 3,074826372 |
| **DELETE** | 0,183049989 | 0,405863261 | 0,503250325 | 0,959520447 |
| **INSERT** | 0,010333824 | 0,002795243 | 0,007140672 | 0,004440761 |

Рисунок 3.7 – Среднее время выполнения CRUD запросов в PostgreSQL

Таблица 3.3 – Среднеквадратичное отклонение времени выполнения CRUD запросов для СУБД PostgreSQL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **postgres1** | **postgres2** | **postgres3** | **postgres4** |
| **SELECT** | 0,00499819 | 0,020905637 | 0,024398539 | 0,054754681 |
| **UPDATE** | 0,001152968 | 0,017804095 | 0,028019635 | 0,201439186 |
| **DELETE** | 0,002124666 | 0,005974271 | 0,005247237 | 0,003881541 |
| **INSERT** | 0,002062634 | 0,001912584 | 0,00084331 | 0,000451901 |

Рисунок 3.8 – Среднеквадратичное отклонение времени выполнения CRUD запросов для СУБД PostgreSQL

Исходя из роста функции среднего значения времени выполнения запросов Select и Update (рисунок 3.7) и среднеквадратичного отклонения (рисунок 3.8), мы можем сделать вывод, что чем больше количество строк в БД, тем больше времени требуется для выполнения запросов Select и Update.

Delete и Insert выполняются очень быстро, и изменения, связанные с увеличением базы данных, не существенно влияют на скорость выполнения запросов (рисунок 3.8).

## **3.3. Проведение экспериментов в SQLite с описанием запросов**

В результате экспериментов в СУБД SQLite с требуется установить зависимость времени выполнения запросов при работе с SQLite CRUD к БД разных размеров, для каждого запроса рассчитать среднее и среднеквадратичное отклонение. Для эксперимента используются БД SQLite\_1, SQLite\_2, SQLite\_3, SQLite\_4 (раздел 2.1). Используемые запросы точно такие же, как в Разделе 3.2.

Далее в соответствии с алгоритмом проведения экспериментов, описанном в Разделе 3.1, выполняем 20 CRUD запросов к каждой из двух СУБД для расчета среднего значения и среднеквадратичного отклонения.

Как описано ранее в Разделе 2.1:

* количество строк в БД SQLite\_1 – 483 439;
* количество строк в БД SQLite\_2 – 966 899;
* количество строк в БД SQLite\_3 – 1 933 819;
* количество строк в БД SQLite\_4 – 3 867 659.

Результат выполнения 20 запросов Update к СУБД SQLite приведен в Таблице 3.4 и продемонстрирован на Рисунке 3.9. Используется запрос UPDATE wb\_products SET dist = 100 WHERE dist = 99.

Таблица 3.4 – Результат выполнения 20 запросов UPDATE в СУБД SQLite

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ Update** | **SQLite\_1** | **SQLite\_2** | **SQLite\_3** | **SQLite\_4** |
| 1 | 0,144148588 | 0,563112974 | 1,089136628 | 1,147135258 |
| 2 | 0,183686495 | 0,646323204 | 1,094427347 | 2,963939667 |
| 3 | 0,136494398 | 0,301933527 | 0,582471371 | 3,009844303 |
| 4 | 0,154922485 | 0,339317799 | 0,607671022 | 2,854712248 |
| 5 | 0,173482656 | 0,340688229 | 1,067941904 | 2,674207211 |
| 6 | 0,165002346 | 0,318427563 | 1,123900175 | 2,344498634 |
| 7 | 0,157465935 | 0,326195955 | 1,17198348 | 2,504748344 |

Продолжение таблицы 3.4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ Update** | **SQLite\_1** | **SQLite\_2** | **SQLite\_3** | **SQLite\_4** |
| 8 | 0,153924227 | 0,316440582 | 1,107788801 | 2,270922422 |
| 9 | 0,15418601 | 0,315835953 | 0,664513111 | 1,159860849 |
| 10 | 0,206473589 | 0,324757576 | 0,644913673 | 1,129983425 |
| 11 | 0,187879086 | 0,339012146 | 0,614657879 | 1,18343401 |
| 12 | 0,13750267 | 0,303218842 | 0,614030838 | 1,212903738 |
| 13 | 0,171632528 | 0,323848248 | 0,600276947 | 1,240617275 |
| 14 | 0,187623501 | 0,313157797 | 0,592596054 | 1,30702734 |
| 15 | 0,148228168 | 0,312974453 | 0,605332136 | 1,160255909 |
| 16 | 0,139810801 | 0,310529232 | 0,637218237 | 1,212408066 |
| 17 | 0,142521143 | 0,621321678 | 0,611548901 | 1,145684481 |
| 18 | 0,188092947 | 0,569136381 | 1,148563862 | 1,217784166 |
| 19 | 0,141818047 | 0,55821228 | 1,028506041 | 1,154778719 |
| 20 | 0,143033504 | 0,669199705 | 1,120984554 | 1,158035517 |

Рисунок 3.9 – Результат выполнения 20 запросов UPDATE в СУБД SQLite

С увеличением количества строк в СУБД SQLite увеличивается и время выполнения запроса UPDATE. Расчеты среднего времени выполнения CRUD запросов показаны в Таблице 3.5 и на Рисунке 3.10, а расчеты среднеквадратичного отклонения времени выполнения запросов продемонстрированы в Таблице 3.6 и на Рисунке 3.11.

Таблица 3.5 – Среднее время выполнения CRUD запросов в СУБД SQLite

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **SQLite\_1** | **SQLite\_2** | **SQLite\_3** | **SQLite\_4** |
| **SELECT** | 0,000053138 | 0,000156999 | 0,000101864 | 0,000098162 |
| **UPDATE** | 0,160896456 | 0,405682206 | 0,836423147 | 1,702639079 |
| **DELETE** | 0,167783177 | 0,408836341 | 0,891940571 | 1,567275655 |
| **INSERT** | 0,017631091 | 0,015506065 | 0,016887665 | 0,012220359 |

Рисунок 3.10 – Среднее время выполнения CRUD запросов в СУБД SQLite

Таблица 3.6 – Среднеквадратичное отклонение времени для СУБД SQLite

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **SQLite\_1** | **SQLite\_2** | **SQLite\_3** | **SQLite\_4** |
| **SELECT** | 0,000012236 | 0,000036154 | 0,000023412 | 0,000022518 |
| **UPDATE** | 0,004098042 | 0,060455062 | 0,065282864 | 0,124940626 |
| **DELETE** | 0,006559406 | 0,038146823 | 0,050134814 | 0,067431881 |
| **INSERT** | 0,002358979 | 0,000855617 | 0,000988429 | 0,000482925 |

Рисунок 3.11 – Среднеквадратичное отклонение времени выполнения CRUD запросов для СУБД SQLite

Исходя из роста функции среднего значения времени выполнения запросов Delete и Update (рисунок 3.10) и среднеквадратичного отклонения (рисунок 3.11), мы можем сделать вывод, что чем больше количество строк в БД, тем больше времени требуется для выполнения запросов Delete и Update.

Select и Insert выполняются очень быстро, и изменения, связанные с увеличением базы данных, не существенно влияют на скорость выполнения запросов (рисунок 3.11).

## **3.4. Сравнение времени выполнения запросов в PostgreSQL и SQLite**

Сравним скорость выполнения запросов в PostgesSQL и SQLite по значению среднеквадратичного отклонения запросов CRUD для четырех баз данных с разным количеством строк (раздел 2.1). Для сравнения скорости запросов сделаем 4 таблицы (таблицы 3.7-3.10) и визуализируем графики среднеквадратичного отклонения для 4 запросов (рисунки 3.12-3.15).

Таблица 3.7 – Среднее время выполнения и среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Select

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SELECT** | **483 439** | **966 899** | **1 933 819** | **3 867 659** |
| **Среднее время выполнения** | | | | |
| **PostgreSQL** | 1,179838336 | 2,472717357 | 4,553863156 | 9,156701338 |
| **SQLite** | 0,000053138 | 0,000156999 | 0,000101864 | 0,000098162 |
| **Среднеквадратичное отклонение времени выполнения** | | | | |
| **PostgreSQL** | 0,00499819 | 0,020905637 | 0,024398539 | 0,054754681 |
| **SQLite** | 0,000012236 | 0,000036154 | 0,000023412 | 0,000022518 |

Рисунок 3.12 – Среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Select

Из Рисунка 3.12 видно, что SELECT запрос в SQLite выполняется существенно быстрее. И разница будет еще сильнее заметна на большем количестве строк, так как с увеличением базы данных растет время запроса.

Таблица 3.8 – Среднее время выполнения и среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Update

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **UPDATE** | **483 439** | **966 899** | **1 933 819** | **3 867 659** |
| **Среднее время выполнения** | | | | |
| **PostgreSQL** | 0,250711727 | 0,575242329 | 0,896314692 | 3,074826372 |
| **SQLite** | 0,160896456 | 0,405682206 | 0,836423147 | 1,702639079 |
| **Среднеквадратичное отклонение времени выполнения** | | | | |
| **PostgreSQL** | 0,001152968 | 0,017804095 | 0,028019635 | 0,201439186 |
| **SQLite** | 0,004098042 | 0,060455062 | 0,065282864 | 0,124940626 |

Рисунок 3.13 – Среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Update

Из Рисунка 3.13 видно, что Update запрос в SQLite и PostgreSQL выполняется, примерно одинаково. И с увеличением БД растет время запроса.

Таблица 3.9 – Среднее время выполнения и среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Delete

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DELETE** | **483 439** | **966 899** | **1 933 819** | **3 867 659** |
| **Среднее время выполнения** | | | | |
| **PostgreSQL** | 0,183049989 | 0,405863261 | 0,503250325 | 0,959520447 |
| **SQLite** | 0,167783177 | 0,408836341 | 0,891940571 | 1,567275655 |
| **Среднеквадратичное отклонение времени выполнения** | | | | |
| **PostgreSQL** | 0,002124666 | 0,005974271 | 0,005247237 | 0,003881541 |
| **SQLite** | 0,006559406 | 0,038146823 | 0,050134814 | 0,067431881 |

Рисунок 3.14 – Среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Delete

Из Рисунка 3.14 видно, что с увеличением БД время запроса растет в SQLite, а в PostgresSQL уменьшается или остаётся почти неизменным. Такая необычная зависимость может быть объяснена наличием оптимизирующего алгоритма в PostgresSQL для запроса Delete.

Таблица 3.10 – Среднее время выполнения и среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Insert

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INSERT** | **483 439** | **966 899** | **1 933 819** | **3 867 659** |
| **Среднее время выполнения** | | | | |
| **PostgreSQL** | 0,010333824 | 0,002795243 | 0,007140672 | 0,004440761 |
| **SQLite** | 0,017631091 | 0,015506065 | 0,016887665 | 0,012220359 |
| **Среднеквадратичное отклонение времени выполнения** | | | | |
| **PostgreSQL** | 0,002062634 | 0,001912584 | 0,00084331 | 0,000451901 |
| **SQLite** | 0,002358979 | 0,000855617 | 0,000988429 | 0,000482925 |

Рисунок 3.15 – Среднеквадратичное отклонение времени выполнения в СУБД PostgreSQL и SQLite запроса Insert

Из Рисунка 3.15 видно, что Insert запрос в SQLite и PostgreSQL выполняется, примерно одинаково. И с увеличением БД уменьшается время запроса благодаря оптимизации на большом количестве записываемых в СУБД данных.

# **4. Заключение**

Создали программу для выгрузки данных из внешнего источника, разобрали и структурировали исходные данные, загрузили их в СУБД PostgreSQL и SQLite для проведения исследований. Проанализировали возможности двух СУБД посредством сравнения времени выполнения CRUD запросов к большому объему загруженных данных.

# **5. Список используемой литературы**

1. Виноградов В., Виноградова М. В. Постреляционные модели данных и языки запросов / Москва: Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет), 2017. 100 с. ISBN 978-5-7038-4283-6. EDN XFIMAW.
2. Мангушева А.Р. Базы данных на СУБД PostgreSQL / Казань: Общество с ограниченной ответственностью "Редакционно-издательский центр "Школа", 2020. 93 с. ISBN 978-5-00162-258-1. EDN ACMIUL.
3. Голдовский Я.М. Введение в постреляционные базы данных: учебное пособие для студентов, обучающихся по направлению "Информатика и вычислительная техника" по дисциплине "Постреляционные базы данных" / Московский гос. ун-т путей сообщ. (МИИТ), Ин-т упр. и информ. технологий, Каф. "Вычислительные системы и сети". Москва: МИИТ, 2008. 92 с. EDN QMTPUH.
4. Парфенов Ю.П. Постреляционные хранилища данных: учебное пособие / Министерство образования и науки Российской Федерации, Уральский федеральный университет. Екатеринбург: Издательство Уральского университета, 2016. 120 с. ISBN 978-5-7996-1827-8. EDN WYZTQX.
5. Григорьев Ю.А., Плутенко А.Д., Плужникова А.Ю. Реляционные базы данных и системы NoSQL / Благовещенск: Амурский государственный университет, 2018. 424 с. ISBN 978-5-93493-308-2. EDN XTSEKL.

# **Приложения**

## **Приложение 1. DDL сценарий создания таблицы в PostgreSQL и SQLite**

Таблица товаров маркетплейса «wb\_products» состоит из 36 столбцов, описанных в Разделе 2.3. Для создания таблицы в СУБД PostgreSQL и SQLite выполняется одинаковый DDL сценарий:

CREATE TABLE IF NOT EXISTS wb\_products (

sort INT NOT NULL,

ksort INT NOT NULL,

time1 INT NOT NULL,

time2 INT NOT NULL,

dist INT NOT NULL,

id INT NOT NULL,

root INT NOT NULL,

kindId INT NOT NULL,

subjectId INT NOT NULL,

subjectParentId INT NOT NULL,

name TEXT NOT NULL,

brand TEXT NOT NULL,

brandId INT NOT NULL,

siteBrandId INT NOT NULL,

supplierId INT NOT NULL,

sale INT NOT NULL,

priceU INT NOT NULL,

salePriceU INT NOT NULL,

logisticsCost INT NOT NULL,

saleConditions INT NOT NULL,

pics INT NOT NULL,

rating INT NOT NULL,

feedbacks INT NOT NULL,

panelPromoId INT NOT NULL,

promoTextCat TEXT NOT NULL,

volume INT NOT NULL,

diffPrice TEXT NOT NULL,

colorsName TEXT NOT NULL,

colorsId INT NOT NULL,

sizeName TEXT NOT NULL,

sizesOrigName TEXT NOT NULL,

sizesRank INT NOT NULL,

sizesOptionId INT NOT NULL,

sizesWh INT NOT NULL,

sizesSign TEXT NOT NULL,

link TEXT NOT NULL);

## **Приложение 2. Программный код загрузки данных**

Для реализации программы загрузки данных из внешнего источника в СУБД PostgreSQL и SQLite был использован язык программирования Python. В программе использовалось 7 библиотек: random, tqdm, pandas, psycopg2, requests, sqlite3, json

Функция get\_catalogs\_wb получает все каталоги WB с помощью запроса к специальному API. После получения информации функция сохраняет данные в JSON файл для дальнейшей обработки.

url = 'https://www.wildberries.ru/webapi/menu/main-menu-ru-ru.json'

headers = {'Accept': "\*/\*", 'User-Agent': "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"}

response = requests.get(url, headers=headers).json()

with open('wb\_catalogs\_data.json', 'w', encoding='UTF-8') as file:

json.dump(response, file, indent=2, ensure\_ascii=False)

data\_list = []

for d in response:

try:

for child in d['childs']:

if target == child['url']:

data\_list.append({

'category\_name': child['name'],

'category\_url': child['url'],

'shard': child['shard'],

'query': child['query']})

else:

for sub\_child in child['childs']:

data\_list.append({

'category\_name': sub\_child['name'],

'category\_url': sub\_child['url'],

'shard': sub\_child['shard'],

'query': sub\_child['query']})

except:

# print(f'не имеет дочерних каталогов \*{d["name"]}\*')

continue

return data\_list

Функция search\_caterogory\_in\_catalog ищет совпадение категории со списком из файла с каталогами (рисунок 4). Это необходимо, чтобы отправлять серверу запросы только к существующим спискам товаров и избегать блокировки сервером из-за некорректного запроса:

try:

for catalog in catalog\_list:

if catalog['category\_url'] == target:

print(f'найдено совпадение: {catalog["category\_name"]}')

name\_category = catalog['category\_name']

shard = catalog['shard']

query = catalog['query']

return name\_category, shard, query

else:

# print('нет совпадения')

pass

except:

print('Данный раздел не найден!')

Функция get\_data\_from\_json забирает данные из JSON файла с данными о товарах и сохраняет их в список data\_list для последующей работы с ними в программе. Также в этой функции реализован поиск исключений, когда отсутствует какой-либо параметр товара.

data\_list = []

for data in json\_file['data']['products']:

# try:

# price = int(data["priceU"] / 100)

# except:

# price = 0

# print (data)

try:

datatime2 = data['time2']

except KeyError:

datatime2 = f'-1'

try:

datatime1 = data['time1']

except KeyError:

datatime1 = f'-1'

try:

datadist = data['dist']

except KeyError:

datadist = f'-1'

try:

datapanelPromoId = data['panelPromoId']

datapromoTextCat = data['promoTextCat']

except KeyError:

datapanelPromoId = f'-1'

datapromoTextCat = f'-1'

try:

datacolorsName = data['colors'][0]['name']

datacolorsid = data['colors'][0]['id']

except IndexError:

datacolorsName = f'-1'

datacolorsid = f'-1'

data\_list.append({

'\_\_sort': data['\_\_sort'],

'time1': datatime1,

'time2': datatime2,

'dist': datadist,

'id': data['id'],

'root': data['root'],

'kindId': data['kindId'],

'subjectId': data['subjectId'],

'subjectParentId': data['subjectParentId'],

'Наименование': data['name'],

'Бренд': data['brand'],

'id бренда': data['brandId'],

'siteBrandId': data['siteBrandId'],

'supplierId': data['supplierId'],

'Скидка': data['sale'],

'priceU': data['priceU'],

'salePriceU': data['salePriceU'],

'logisticsCost': data['logisticsCost'],

'saleConditions': data['saleConditions'],

'pics': data['pics'],

'rating': data['rating'],

'feedbacks': data['feedbacks'],

'panelPromoId': datapanelPromoId,

'promoTextCat': datapromoTextCat,

'volume': data['volume'],

'diffPrice': data['diffPrice'],

'colors\_name': datacolorsName,

'colors\_id': datacolorsid,

'sizes\_name': data["sizes"][0]["name"],

'sizes\_origName': data['sizes'][0]['origName'],

'sizes\_rank': data['sizes'][0]['rank'],

'sizes\_optionId': data['sizes'][0]['optionId'],

'sizes\_wh': data['sizes'][0]['wh'],

'sizes\_sign': data['sizes'][0]['sign'],

'Ссылка': f'https://www.wildberries.ru/catalog/{data["id"]}/detail.aspx?targetUrl=BP'

})

return data\_list

Функция get\_content отправляет запрос к серверу, получает список товаров с параметрами в конкретной категории и сохраняет их в JSON файл для последующей работы с ними. Также в этой функции реализован обход ограничений сервера на огромное количество однотипных запросов с помощью изменения заголовка запроса на заголовки самых популярных браузеров.

desktop\_agents = [

'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36',

'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36',

'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_1) AppleWebKit/602.2.14 (KHTML, like Gecko) Version/10.0.1 Safari/602.2.14',

'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36',

'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36',

'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36',

'Mozilla/5.0 (Windows NT 10.0; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0']

headers = {'User-Agent': choice(desktop\_agents), 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8'}

data\_list = []

for page in tqdm(range(1, 101)):

# url = f'https://wbxcatalog-ru.wildberries.ru/{shard}' \

# f'/catalog?appType=1&curr=rub&dest=-1029256,-102269,-1278703,-1255563' \

# f'&{query}&lang=ru&locale=ru&sort=sale&page={page}' \

# f'&priceU={low\_price \* 100};{top\_price \* 100}'

url = f'https://catalog.wb.ru/catalog/{shard}/catalog?appType=1&curr=rub&dest=-1075831,-77677,-398551,12358499' \

f'&locale=ru&page={page}&priceU={1 \* 100};{1000000 \* 100}' \ f'&reg=0&regions=64,83,4,38,80,33,70,82,86,30,69,1,48,22,66,31,40&sort=popular&spp=0&{query}'

data = requests.get(url, headers=headers).json()

if len(get\_data\_from\_json(data)) > 0:

data\_list.extend(get\_data\_from\_json(data))

else:

print(f'Сбор данных завершен.')

break

# print (data\_list)

return data\_list

Функция save\_database сохраняет данные из списка товаров в постоянное хранилище СУБД SQLite и СУБД PostgreSQL для последующей работы с ними

data = pd.DataFrame(data\_list)

data = list(data.itertuples(index=True, name=None))

conSQLite.cursor().executemany(

"INSERT into wb\_products values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)",

data)

conSQLite.commit()

conPostgreSQL.cursor().executemany(

"INSERT into wb\_products values (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)",

data)

conPostgreSQL.commit()

Функция parser\_all выполняет сбор данных и сохранение товаров из всего списка каталогов, используя функции, описанные выше.

target = url

catalog\_list = get\_catalogs\_wb(target)

try:

# поиск введенной категории в общем каталоге

name\_category, shard, query = search\_category\_in\_catalog(target, catalog\_list=catalog\_list)

# сбор данных в найденном каталоге

data\_list = get\_content(shard=shard, query=query, low\_price=low\_price, top\_price=top\_price)

# сохранение найденных данных

#save\_excel(data\_list, f'{name\_category}\_from\_{low\_price}\_to\_{top\_price}')

save\_database(data\_list, conSQLite, conPostgreSQL)

except TypeError:

print('Ошибка! Возможно не верно указан раздел. Удалите все доп фильтры с ссылки')

except PermissionError:

print('Ошибка! Вы забыли закрыть созданный ранее excel файл. Закройте и повторите попытку')

Код основной функции, в ней происходит подключение к базам данных PostgreSQL и SQLite и создание таблицы с помощью DDL сценария из Приложения 1 (рисунок 10).

conSQLite = sqlite3.connect("PBD\_DZ\_SQLite.db")

conPostgreSQL = psycopg2.connect(user="postgres",

password="1234",

host="127.0.0.1",

port="5432",

database="PBD\_DZ\_PostgreSQL")

Create\_wb\_products = '''CREATE TABLE IF NOT EXISTS wb\_products (…);'''

conSQLite.cursor().execute(Create\_wb\_products)

conSQLite.commit()

conPostgreSQL.cursor().execute(Create\_wb\_products)

conPostgreSQL.commit()

low\_price = 1

top\_price = 500000

with open('wb\_catalogs\_data.json', 'r', encoding='UTF-8') as json\_file:

data = json.load(json\_file)

data\_list = []

listID = [128313, 128604, 9156, 62057]

for d in data:

try:

if d['id'] not in listID:

for child in d['childs']:

try:

for child1 in child['childs']:

try:

if child1['id'] not in listID:

for child2 in child1['childs']:

try:

for child3 in child2['childs']:

data\_list.append(

child3['url'])

except:

data\_list.append(child2['url'])

except:

data\_list.append(child1['url'])

except:

data\_list.append(child['url'])

except:

continue

for item in data\_list:

print(item)

parser\_all(item, low\_price, top\_price, conSQLite, conPostgreSQL)

conSQLite.close()

conPostgreSQL.close()

Также в основной функции запускается сбор данных из всех каталогов, которые удалось получить от Wildberries на прошлых этапах. При получении данных из каталогов требуется погружаться на несколько уровней по иерархии в связи с глубокой вложенностью структуры JSON файла.

В результате работы программного кода загрузки данных в обе СУБД было записано 451 147 строк с 36 параметрами в каждой строке (рисунок 12).

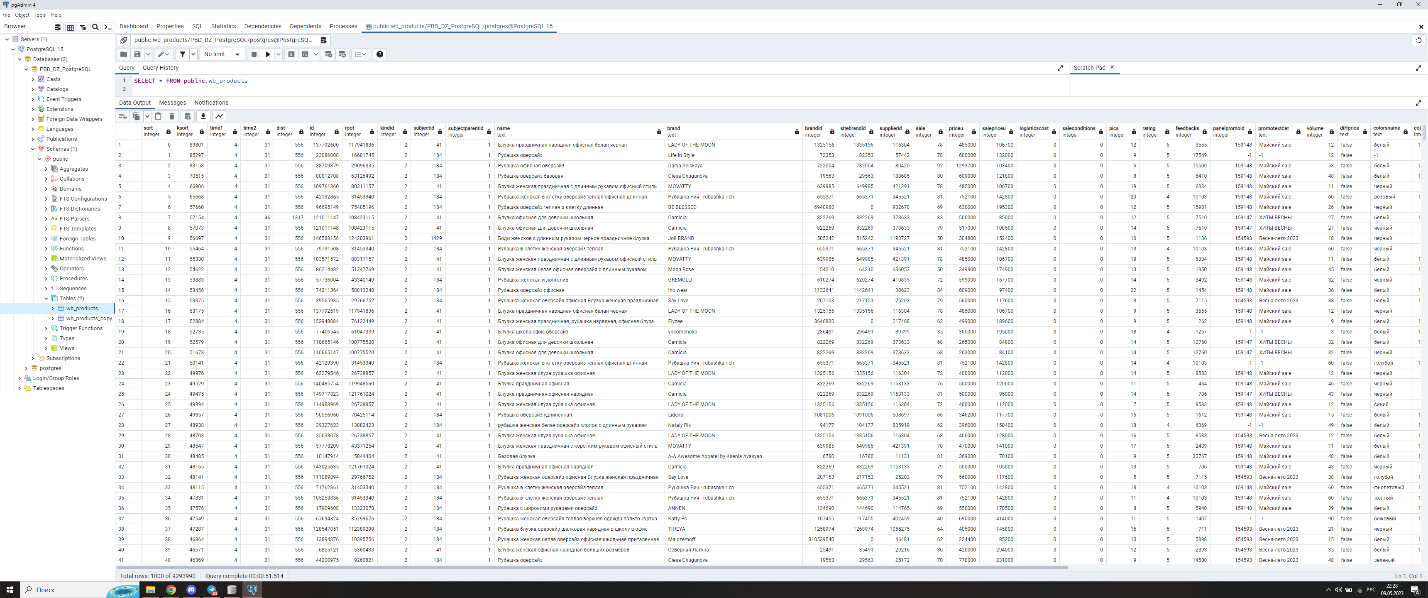


Рисунок 4 – Полученная база данных в СУБД PostgreSQL

## **Приложение 3. Программный код проведения экспериментов**

Для реализации программного кода для проведения экспериментов был использован язык программирования Python. В программе использовалось 5 библиотек: datetime, psycopg2, sqlite3, math, time.

Функция arrayStrToFile записывает в текстовый файл log.txt строку или массив строк, которые подаются на вход функции.

def arrayStrToFile(array):

with open('logs.txt', 'a', encoding='utf-8') as file:

date = datetime.now().strftime('%Y-%m-%d\_%H:%M:%S')

if(isinstance(array, str)):

file.write(f'{array}\n')

else:

for elem in array:

file.write(f'{elem}\n')

Функция sred\_sum\_sv считает среднее значение и среднеквадратичное отклонение времени запроса и записывает их в файл.

def sred\_sum\_cv(time\_array, name):

n = len(time\_array)

sred\_sum\_select = sum(time\_array)/len(time\_array)

sum\_otckl = 0

for i in range(n):

sum\_otckl = sum\_otckl+(time\_array[i]-sred\_sum\_select)\*\*2

sred\_cv\_otckl = math.sqrt(sum\_otckl/(n-1))

print(f"\nСреднее значение времении запроса {name}\n{round(sred\_sum\_select,5)}")

print(f"\nСреднеквадратическое отклонение времени запросы {name}\n{round(sred\_cv\_otckl,5)}")

arrayStrToFile(name)

arrayStrToFile(time\_array)

arrayStrToFile(f"Среднее значение времении запроса {name}")

arrayStrToFile([sred\_sum\_select])

arrayStrToFile(f"Среднеквадратическое отклонение времени запросы {name}")

arrayStrToFile([sred\_cv\_otckl])

В начале основной функции программы выполняется подключение к СУБД PostgreSQL с помощью библиотеки psycopg2 или к СУБД SQLite с помощью библиотеки sqlite3 (рисунок 16).

def main():

time\_select = [] # Мссив времени выполнения запросов Select

time\_update = [] # Массив времени выполнения запросов Update

time\_delete = [] # Массив времени выполнения запросов Delete

time\_insert = [] # Массив времени выполнения запросов Insert

n = 20 # Количество запросов

try:

connection = psycopg2.connect(user="postgres",

password="1234",

host="127.0.0.1",

port="5432",

database="postgres"

cursor = connection.cursor()

postgreSQL\_select\_Query = "SELECT \* FROM wb\_products"

print("Выбор строк из таблицы mobile с помощью cursor.fetchall")

После требуется n раз провести эксперимент для дальнейшего получения среднего значения и среднеквадратичного отклонения для СУБД PostgreSQL и SQLite.

for i in range(n):

# Запрос в базу данных

PostgreSQL\_Select\_1 = time.time()

cursor.execute(postgreSQL\_select\_Query)

PostgreSQL\_Select = time.time()-PostgreSQL\_Select\_1

selectData = cursor.fetchall()

time\_select.append(PostgreSQL\_Select)

print(PostgreSQL\_Select, PostgreSQL\_Select\_1)

print(len(selectData))

# Обновление таблицы wb\_products в PostgreSQL

PostgreSQL\_update\_1 = time.time()

cursor.execute(

'''UPDATE wb\_products SET dist = 100 WHERE dist = 99''')

PostgreSQL\_update = time.time()-PostgreSQL\_update\_1

time\_update.append(PostgreSQL\_update)

# Добавление selectData[0] в таблицу wb\_products в PostgreSQL

PostgreSQL\_Insert1 = time.time()

cursor.execute('''INSERT into wb\_products values (

%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,

%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)''', selectData[0])

# cursor.execute('''INSERT into wb\_products values (

# ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,

# ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''', selectData[3])

connection.commit()

PostgreSQL\_Insert = time.time()-PostgreSQL\_Insert1

time\_insert.append(PostgreSQL\_Insert)

В конце основной функции программы выполняется запись результатов эксперимента в файл log.txt и происходит закрытие соединения с подключенной СУБД.

# Записываем результаты в log

arrayStrToFile("")

arrayStrToFile(f"========== База данных {len(selectData)} ==========")

arrayStrToFile("")

sred\_sum\_cv(time\_select, "Select")

sred\_sum\_cv(time\_update, "Update")

sred\_sum\_cv(time\_delete, "Delete")

sred\_sum\_cv(time\_insert, "Insert")

# Отслеживаем ошибку psycopg2.OperationalError

except psycopg2.OperationalError as error:

print("Ошибка при работе с PostgreSQL", error)

# Независимо от результата закрываем cursor

finally:

if connection:

cursor.close()

connection.close()

print("\nСоединение с PostgreSQL закрыто")

Фрагмент текстового файла log.txt, в котором записаны все экспериментов для обеих СУБД, продемонстрирован на Рисунке 19.

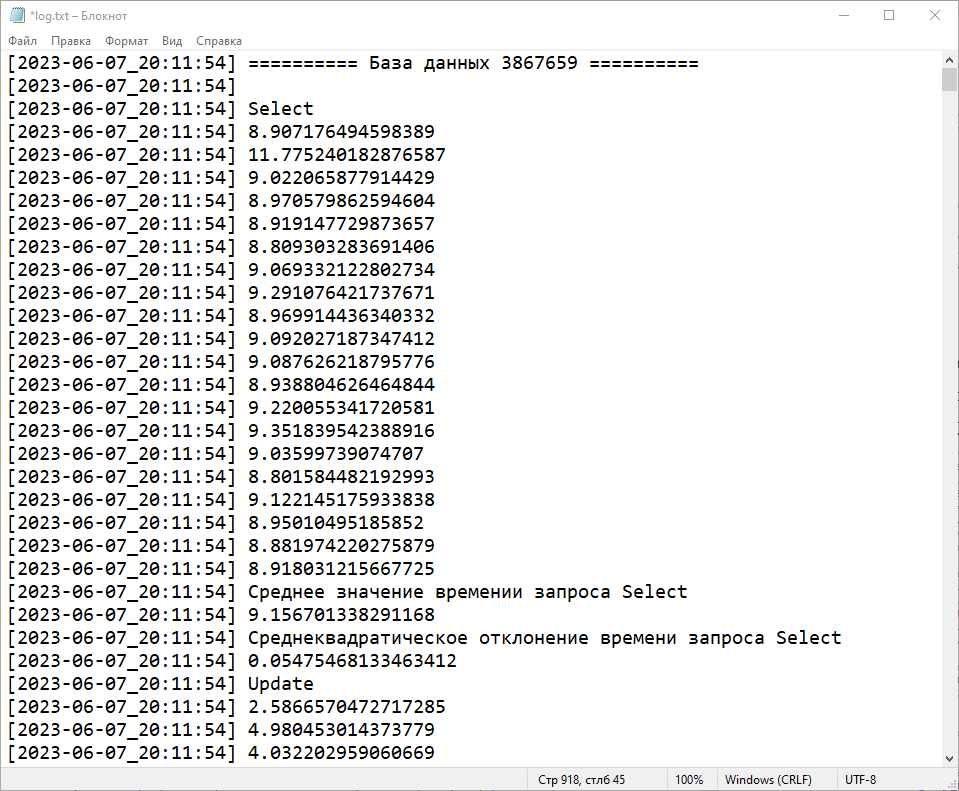


Рисунок 5 – Эксперименты; фрагмент текстового файла log.txt