Открытая Олимпиада СПбГУ среди студентов и молодых специалистов «Petropolitan Science (Re)Search» в 2019/20 учебном году по предмету «Вычислительные технологии» для обучающихся и выпускников бакалавриата/специалитета

СОДЕРЖАНИЕ

Задание 1.

Представленный код реализует решение системы линейный алгебраических уравнений методом Гаусса с выбором главного элемента в столбце. Ниже приведен код данного метода на C++. Программа разрабатывалась в среде Visual Studio 2017.

```
#include "pch.h"
#include <iostream>
#include <math.h>
#include <Windows.h>
#include <ctime>
using namespace std;

int main(void)
{
	double acumtime = 0.0;
	double start, end;

		start = clock();
		SetConsoleCP(1251);
		SetConsoleOutputCP(1251);
		int i, j, k, k1, n;
		float r, s;
		cout << "Введите n " << endl;
		cin >> n;

		float **a = new float*[n];
		for (int count = 0; count < n; count++)
			a[count] = new float[n+1];

		for (i = 0; i < n; i++)
			for (j = 0; j < n + 1; j++)
			{
				cout << "Введите элемент " << "[" << i << "][" << j << "]  ";
				cin >> a[i][j];
			}
		cout << endl;

		int n1 = n + 1;
		float *x = new float[n + 1];

		for (k = 0; k < n; k++)
		{
			k1 = k + 1;
			s = a[k][k];
			j = k;
			for (i = k1; i < n; i++) {
				r = a[i][k];
				if (fabs(r) > fabs(s))
				{
					s = r;
					j = i;
				}
			}

			if (s == 0)
			{
				cout << "DET=0" << endl;
			}
```

```
                if (j != k)
                {
                        for (i = k; i < n1; i++)
                        {
                                r = a[k][i];
                                a[k][i] = a[j][i];
                                a[j][i] = r;
                        }
                }
                for (j = k1; j < n1; j++)
                        a[k][j] = a[k][j] / s;
                for (i = k1; i < n; i++)
                {
                        r = a[i][k];
                        for (j = k1; j < n1; j++)
                                a[i][j] = a[i][j] - r * a[k][j];
                }
        }

        for (i = n - 1; i >= 0; i--)
        {
                s = a[i][n];
                x[n] = 0;
                for (j = i + 1; j <= n; j++)
                        s = s - a[i][j] * x[j];
                x[i] = s;
        }

        cout << "Решение системы:" << endl;
        for (i = 0; i < n; i++)
        {
                cout << "x[" << i + 1 << "]=" << x[i];
                cout << endl;
        }

        end = clock();
        acumtime = (end - start) / CLOCKS_PER_SEC;
    cout << "time: " << acumtime << endl;
        delete[] x;
        for (int i = 0; i < n; i++)
           delete[] a[i];
        delete[] a;
}
```

Для внедрения параллельных вычислений будет рассмотрена технология OpenMP. Если говорить о том, можно ли реализовать программно применение технологии OpenMP в данном случае, то ответ положительный. С другой стороны, возникает вопрос о рациональности применения данной технологии в конкретном случае. Технологии параллельных вычислений применяются для больших данных с целью минимизации времени работы программы. В данном случае, все зависит от числа n (числа неизвестных переменных). В классических примерах используются СЛАУ 2-5-го порядка. Если использовать программу для решения данных СЛАУ, то нет

необходимости в применении технологии OpenMP, т.к. последовательное выполнение программы будет давать такой же результат по времени, а может, иногда даже и еще меньше.

Но чем больше число n, тем больше возрастает необходимость в применении технологии распараллеливания. Данный программный код построен на циклах for, к которым можно применять технологию OpenMP при определенных ограничениях. Первый цикл for имеет множество вложенных в него циклов. Вложенность достигает третьего уровня. В данном случае нерационально делать вложенное распараллеливание, т.к. это может перегружать машину, создавая больше потоков, чем она может обработать. В данном случае есть три варианта: установить распараллеливание во внешнем цикле, установить распараллеливание во внутренних циклах, либо попробовать использовать collapse(). Но Visual Studio 2017, нет возможности применения collapse(). Данная опция может применяться только начиная с версии OpenMP 3.0, а Visual Studio использует версию OpenMP 2.0. Можно также попробовать оптимизировать данный цикл, но это весьма проблематично. Данных цикл включает в себя множество других, которые также являются вложенными. Помимо этого, присутствуют еще и условия. Поэтому было принято решение остановится на распараллеливании внутренних циклов, т.к. это дает больше эффективности по сравнению с распараллеливанием внешнего цикла.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – прямой ход метода Гаусса – исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнетреугольному виду.

Сначала осуществляется поиск максимального элемента столбца. Между потоками можно разделить обработку элементов столбца (разные потоки должны обрабатывать разные элементы), в результате вычислений должен быть сформирован максимальный элемент столбца и его индекс. Разные потоки не должны одновременно пытаться изменять значение

максимального элемента или пытаться читать его в то время, когда другой пишет. Если поместить всю работу с максимальным элементом в критическую секцию – функция фактически останется последовательной, т.к. в один момент времени работал бы только один поток, а остальные находились бы в состоянии ожидания. В связи с этим, необходимо создать в каждом потоке свою локальную копию максимума и вычислить максимум в части массива, соответствующей потоку. Затем, для получения максимума всего столбца выполнить сравнение локальной копии с общим максимумом в критической секции.

После вычисления максимума выполняется перестановка строк, которую также можно распределить между потоками. Затем выполняется обнуление элементов j-того столбца, расположенных ниже k-той строки путем эквивалентных преобразований. Распараллеливание эквивалентных преобразований также можно распределить между потоками. Между потоками можно разделить каждую из строк, для которых выполняются преобразования.

Следующий цикл отвечает за обратных ход метода Гаусса, на котором определяется значение неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д. Применение распараллеливания в данном случае невозможно, т.к. итерации являются зависимыми (от значения x_{n-1} зависит значение x_{n-2}).

Также применено распараллеливание на вывод достаточно нестандартным способом: использование класса stringstream, который позволяет связать поток ввода-вывода со строкой в памяти. Это снижает время работы программы.

При проведении тестирования проверки времени выполнения программы с последовательным и параллельным вычислением, был использован еще один внешний цикл, который зацикливал выполнение

программы. По моему мнению, это дает примерно такой же результат, что и обработка СЛАУ с большим числом переменных. Применение такого метода обуславливается тем, что весьма проблематично найти СЛАУ с большим числом переменных. Также проверка осуществлялась всегда на конкретной СЛАУ без ввода пользователя, чтобы время выполнения программы не зависело от скорости ввода.

По результатам тестов можно прийти к следующему выводу: чем больше итераций в цикле, тем существенней различие между последовательным и параллельным вычислением. На рисунках 1-6 приведены результаты последовательного и паралельного тестирования при 100 итерациях в цикле. Время выполнения рассчитывалось, как среднеарифметическое. Также ниже приведен программный код с использованием технологии OpenMP, который использовался при проверке.

```
#include "pch.h"
#include <iostream>
#include <math.h>
#include <Windows.h>
#include <ctime>
#include <omp.h>
#include <sstream>
#define REPS 100
using namespace std;
std::stringstream buf;

int main(void)
{
	double acumtime = 0.0;
	double start, end;
	for (int rep = 0; rep < REPS; rep++)
	{
		start = clock();
		SetConsoleCP(1251);
		SetConsoleOutputCP(1251);
		int i, j, k, k1;
		const int n = 3;
		float r, s = 0.0;
		/*cout << "Введите n " << endl;
		cin >> n;*/

		float a[n][n + 1] = { {  1,   -2, 3, 10},   //матрица коэффициентов, где
последний столбец - правая часть системы
				      {2, 3, -1, -1},
				      {3, -1,  2,  13} };
		/*float **a = new float*[n];
		for (int count = 0; count < n; count++)
			a[count] = new float[n+1];

		for (i = 0; i < n; i++)
			for (j = 0; j < n + 1; j++)
```

```
	{
		cout << "Введите элемент " << "[" << i << "][" << j << "]  ";
		cin >> a[i][j];
	}
cout << endl;*/

int n1 = n + 1;
float *x = new float[n + 1];

for (k = 0; k < n; k++)
{
	x[n] = 0;
	k1 = k + 1;
	s = a[k][k];
	j = k;
	#pragma omp parallel
	{
		float loc_max = s;
		int loc_max_pos = j;
		#pragma omp for schedule(static) private(i)
		for (i = k1; i < n; i++)
		{
			r = a[i][k];
			if (fabs(r) > fabs(loc_max))
			{
				loc_max = r;
				loc_max_pos = i;
			}
		}

		#pragma omp critical
		{
			if (s < loc_max)
			{
				s = loc_max;
				j = loc_max_pos;
			}
		}

	}

	if (s == 0)
	{
		cout << "DET=0" << endl;
	}
	#pragma omp parallel
	{
		if (j != k)
		{
			#pragma omp for schedule(static) private(i)
			for (i = k; i < n1; i++)
			{
				r = a[k][i];
				a[k][i] = a[j][i];
				a[j][i] = r;
			}
		}
		#pragma omp for schedule(static) private(j)
		for (j = k1; j < n1; j++)
			a[k][j] = a[k][j] / s;

		#pragma omp for schedule(static) private(i,j)
		for (i = k1; i < n; i++)
		{
```

```
                                r = a[i][k];
                                for (j = k1; j < n1; j++)
                                        a[i][j] = a[i][j] - r * a[k][j];
                        }
                }
        }

        for (i = n - 1; i >= 0; i--)
        {
                s = a[i][n];
                for (j = i + 1; j <= n; j++)
                        s = s - a[i][j] * x[j];
                x[i] = s;
        }

        cout << "Решение системы:" << endl;

        #pragma omp parallel
        {
                #pragma omp for schedule(static) private(i)
                for (i = 0; i < n; i++)
                {
                        buf << "x[" << i + 1 << "]=" << x[i];
                        buf << endl;
                }
        }

        cout << buf.rdbuf();
        end = clock();
        acumtime += (end - start) / CLOCKS_PER_SEC;
        delete[] x;
    }
    cout << "time: " << acumtime / REPS << endl;
}
```

Ниже приведен окончательный программный код с использованием технологии OpenMP, удовлетворяющий требованиям задания.

```
#include "pch.h"
#include <iostream>
#include <math.h>
#include <Windows.h>
#include <ctime>
#include <omp.h>
#include <sstream>
using namespace std;
std::stringstream buf;

int main(void)
{
    double acumtime = 0.0;
    double start, end;
    start = clock();
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int i, j, k, k1, n;
    float r, s = 0.0;
    cout << "Введите n " << endl;
    cin >> n;

    float **a = new float*[n];
    for (int count = 0; count < n; count++)
        a[count] = new float[n+1];
```

```
for (i = 0; i < n; i++)
	for (j = 0; j < n + 1; j++)
	{
		cout << "Введите элемент " << "[" << i << "][" << j << "]  ";
		cin >> a[i][j];
	}
cout << endl;

int n1 = n + 1;
float *x = new float[n + 1];

for (k = 0; k < n; k++)
{
	x[n] = 0;
	k1 = k + 1;
	s = a[k][k];
	j = k;
	#pragma omp parallel
	{
		float loc_max = s;
		int loc_max_pos = j;
		#pragma omp for schedule(static) private(i)
		for (i = k1; i < n; i++)
		{
			r = a[i][k];
			if (fabs(r) > fabs(loc_max))
			{
				loc_max = r;
				loc_max_pos = i;
			}
		}

		#pragma omp critical
		{
			if (s < loc_max)
			{
				s = loc_max;
				j = loc_max_pos;
			}
		}

	}

	if (s == 0)
	{
		cout << "DET=0" << endl;
	}
	#pragma omp parallel
	{
		if (j != k)
		{
		#pragma omp for schedule(static) private(i)
			for (i = k; i < n1; i++)
			{
				r = a[k][i];
				a[k][i] = a[j][i];
				a[j][i] = r;
			}
		}
		#pragma omp for schedule(static) private(j)
		for (j = k1; j < n1; j++)
			a[k][j] = a[k][j] / s;
```

```
                #pragma omp for schedule(static) private(i,j)
                for (i = k1; i < n; i++)
                {
                        r = a[i][k];
                        for (j = k1; j < n1; j++)
                        a[i][j] = a[i][j] - r * a[k][j];
                }
        }
}

for (i = n - 1; i >= 0; i--)
{
        s = a[i][n];
        for (j = i + 1; j <= n; j++)
                s = s - a[i][j] * x[j];
        x[i] = s;
}

cout << "Решение системы:" << endl;

#pragma omp parallel
{
        #pragma omp for schedule(static) private(i)
        for (i = 0; i < n; i++)
        {
                buf << "x[" << i + 1 << "]=" << x[i];
                buf << endl;
        }
}

cout << buf.rdbuf();
end = clock();
acumtime = (end - start) / CLOCKS_PER_SEC;
cout << "time: " << acumtime << endl;

delete[] x;
for (int i = 0; i < n; i++)
        delete[] a[i];
delete[] a;

}
```

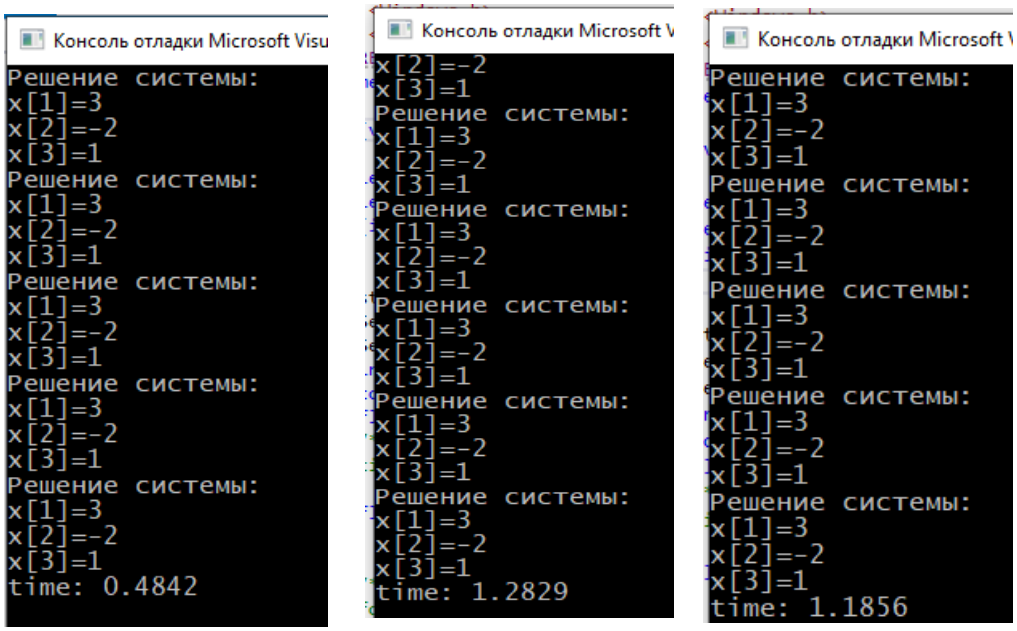


Рисунок 1-3 – Последовательное выполнение программы

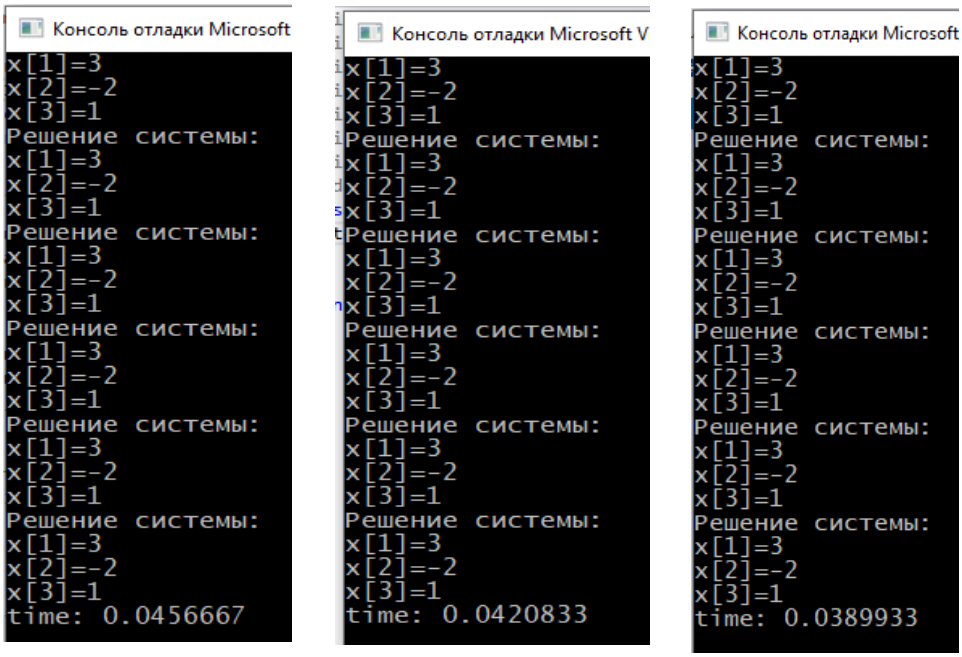


Рисунок 4-6 – Выполнение программы с использование технологии OpenMP

Таким образом, технология OpenMP в данном случае помогает распределить решение подзадач между различными нитями, что позволяет обеспечить лучшую балансировку вычислительной нагрузки.

Задание 2.

Реляционная схема базы данных службы доставки ресторана представлена на рисунке 7.

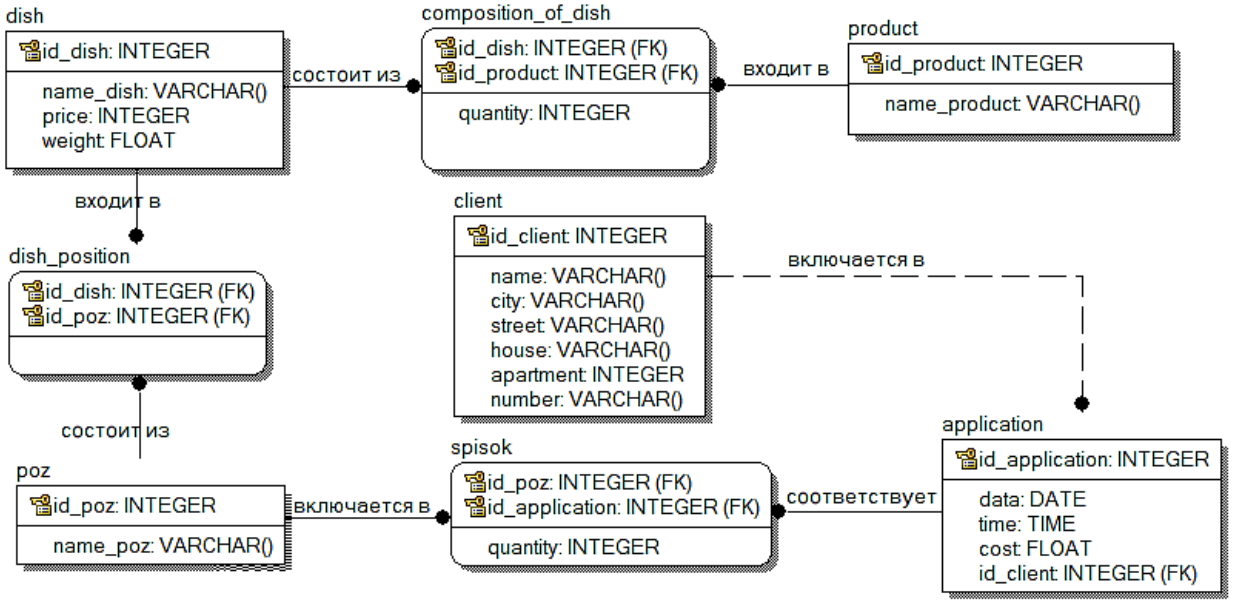


Рисунок 7 – Реляционная схема базы данных доставки ресторана

Меню состоит из блюд (таблица “dish”). Каждое блюдо имеет наименование (name_dish), цену (price), вес (weight). Каждое блюдо включает множество продуктов (таблица “product”). В то же время один продукт может быть включен во множество блюд. Связь «многие-ко-многим» необходимо было нормализовать с созданием промежуточной таблицы “composition_of_dish”. Составной первичный ключ таблицы “composition_of_dish” состоит из первичных ключей таблиц dish и product.

Требование о том, что различные вариации блюда считаются отдельными позициями, было реализовано следующим образом: наименование (name_dish) блюда может быть одинаковым, а вариацию блюда можно узнать при помощи id_dish из таблицы “composition_of_dish”. Позиции (id_poz) присваивается id_dish. Таким образом, разные вариации одного и того же блюда будут иметь разные id_dish и будут присвоены разным id_poz.

С учетом комбо-позиций в одну позицию может входить несколько блюд. Одновременно с этим одно блюдо может входить в несколько позиций.

Возникает связь «многие-ко-многим», которую необходимо нормализовать через промежуточную таблицу “dish_position”.

В журнале заказов должен хранится перечень позиций и их количество. Для того, чтобы это реализовать на физическом уровне необходимо ввести список позиций в заказе (промежуточная таблица “spisok”), так как одна позиция может включаться во множество заказов и один заказ может содержать множество позиций.

Также при приведении к 1НФ в сущности client поле адрес являлось неатомарным. Поэтому данное поле было разбито на поля: город (city), улица (street), дом (house), квартира (apartment). Помимо этого, поле телефон (number) для того, чтобы оно являлось атомарным должно включать не более одного номера.

При приведении ко 2НФ никаких преобразований не осуществлялось. Три таблицы имеют составной первичный ключ, две из которых имеют по одному неключевому атрибуту. Данные неключевые атрибуты полностью зависят от составного первичного ключа. В таблице “dish_position” отсутствуют неключевые атрибуты за неимением дополнительной информации.

При приведении к 3НФ и нормальной форме Бойса-Кодда также не было выявлено каких-то отклонений.

Ниже приведен код создания таблицы клиента на языке SQL.

```
CREATE TABLE client (
  id_client INT NOT NULL,
  name VARCHAR(40) NOT NULL,
  city VARCHAR(40),
  street VARCHAR(40),
  house VARCHAR(5),
  apartment INT,
  number VARCHAR(13),
```

```
  PRIMARY KEY (id_client));
```

Для полей house и number был применен тип данных VARCHAR(). Номер телефона может быть домашним, включать скобки. При требовании заказчика можно увеличить размерность, если требуются еще какие-то дополнительные знаки. Также номер дома может содержать дробь и буквы.

Ниже приведен код создания таблиц журнала заказов, позиций и списка позиций в заказе.

```
CREATE TABLE application (
  id_application INT PRIMARY KEY,
  data DATE,
  time TIME,
  cost FLOAT,
  id_client INT,
  FOREIGN KEY (id_client)  REFERENCES client (id_client)
   );
CREATE TABLE poz (
  id_poz INT NOT NULL,
  name_poz VARCHAR(40) NOT NULL,
  PRIMARY KEY (id_poz)
   );
CREATE TABLE spisok (
  id_application INT,
  id_poz INT,
  quantity INT,
  PRIMARY KEY (id_application , id_poz),
  FOREIGN KEY (id_application)  REFERENCES application (id_application),
  FOREIGN KEY (id_poz)  REFERENCES poz (id_poz)
   );
```

Запрос: Имея фиксированную дату X, вывести список клиентов, которые заказывали самую популярную в этот день позицию меню. Ниже приведена программная реализация на языке SQL.

```
SELECT DISTINCT name
FROM client JOIN application
ON client.id_client=application.id_client AND client.id_client IN(SELECT application.id_application
FROM spisok JOIN application
ON application.id_application = spisok.id_application
AND data='2020-02-08' AND id_poz=(SELECT id_poz
FROM spisok JOIN application
ON application.id_application = spisok.id_application
AND data='2020-02-08'
GROUP BY id_poz
HAVING SUM(kol) >= ALL(SELECT SUM(kol) FROM spisok JOIN application
ON application.id_application = spisok.id_application  AND data='2020-02-08'
GROUP BY id_poz)));
```

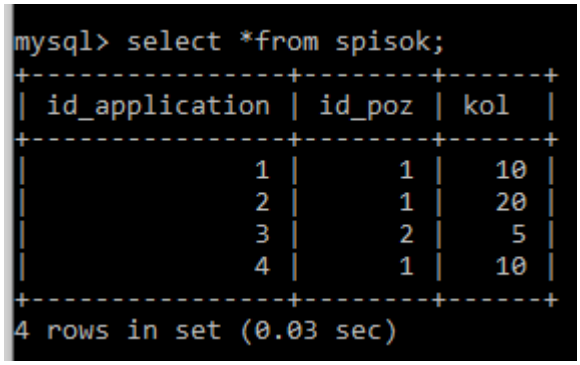
На рисунках 8-10 приведены примеры заполнения таблиц для реализации запроса.

```
mysql> select *from client;
+-----------+---------+-----------+
| id_client | name    | address   |
+-----------+---------+-----------+
|         1 | Петров  | Советская |
|         2 | Иванов  | Гагарина  |
|         3 | Сидоров | Ленина    |
+-----------+---------+-----------+
3 rows in set (0.03 sec)
```

Рисунок 8 – Пример заполнения таблицы client

```
mysql> select *from application;
+----------------+------------+-----------+
| id_application | data       | id_client |
+----------------+------------+-----------+
|              1 | 2020-02-08 |         1 |
|              2 | 2020-02-08 |         2 |
|              3 | 2020-02-08 |         3 |
|              4 | 2020-05-02 |         1 |
+----------------+------------+-----------+
4 rows in set (0.00 sec)
```

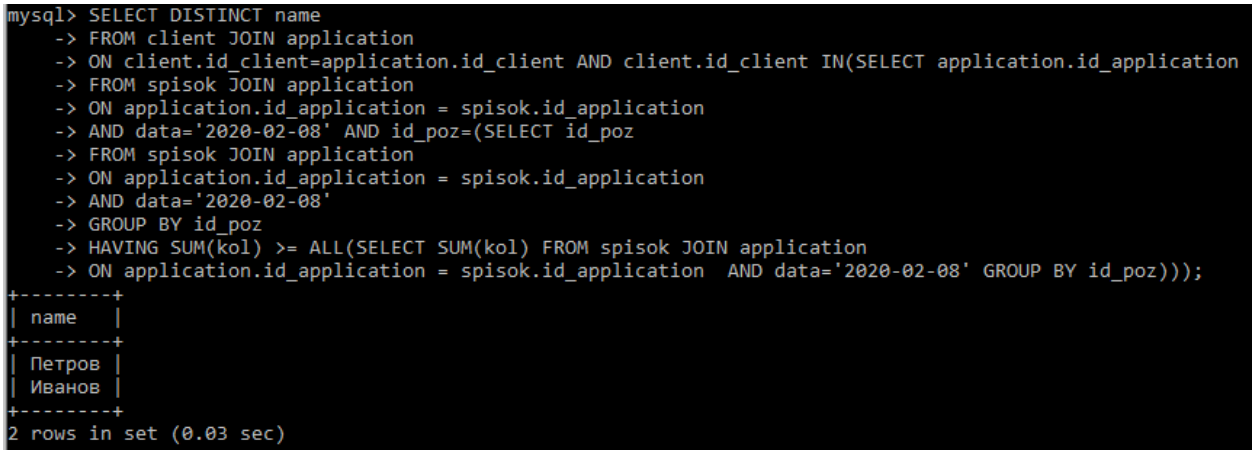
Рисунок 9 – Пример заполнения таблицы application



```
mysql> select *from spisok;
+----------------+--------+------+
| id_application | id_poz | kol  |
+----------------+--------+------+
|              1 |      1 |   10 |
|              2 |      1 |   20 |
|              3 |      2 |    5 |
|              4 |      1 |   10 |
+----------------+--------+------+
4 rows in set (0.03 sec)
```

Рисунок 10 – Пример заполнения таблицы spisok

На рисунке 11 представлен результат работы запроса.



```
mysql> SELECT DISTINCT name
    -> FROM client JOIN application
    -> ON client.id_client=application.id_client AND client.id_client IN(SELECT application.id_application
    -> FROM spisok JOIN application
    -> ON application.id_application = spisok.id_application
    -> AND data='2020-02-08' AND id_poz=(SELECT id_poz
    -> FROM spisok JOIN application
    -> ON application.id_application = spisok.id_application
    -> AND data='2020-02-08'
    -> GROUP BY id_poz
    -> HAVING SUM(kol) >= ALL(SELECT SUM(kol) FROM spisok JOIN application
    -> ON application.id_application = spisok.id_application  AND data='2020-02-08' GROUP BY id_poz)));
+--------+
| name   |
+--------+
| Петров |
| Иванов |
+--------+
2 rows in set (0.03 sec)
```

Рисунок 11 – Результат работы запроса с тестовыми данными

Примечание. Некоторые из таблиц на рисунках 8-10 имеют не совсем корректную структуру. Поэтому те поля, которые участвовали в запросе объявлены и заполнены корректно, остальные поля были созданы частично и некоторые из них не являются корректными (данные поля просто созданы для видимости). Основная цель тестовых данных и таблиц – проверка запроса.

Задание 3.

Big Data бесспорно является одним из самых популярных терминов в современном мире. Количество собранных данных продолжает расти. Поэтому анализ и обработка больших данных являются одной из наиболее важных задач, над которыми работают исследователи, чтобы найти наилучшие подходы для их обработки с высокой производительностью, низкой стоимостью и высокой точностью.

Большие данные определяются на основе «3V» [6]: volume (огромные объемы данных), variety (разнообразие форм и структур) и velocity (скорость обработки данных). Данные определения ввел аналитик компании Gartner Дуг Лейни (Doug Laney) в 2001 году.

Корни обработки технологий больших данных уходят в 2002 год, когда Дуг Каттинг (Doug Cutting) работал над проектом с открытым исходным кодом Nutch, целью которого была индексация веб-страниц и использование уже проиндексированных страниц для поиска. При работе он столкнулся с проблемами масштабируемости как с точки хранения, так и с точки зрения вычислений. В 2003 году Google опубликовал GFS (файловая система Google), а в 2004 году Nutch создал NDFS (распределенную файловую систему Nutch). В этом же году компания Google разработала свою парадигму программирования MapReduce. Авторами данной модели являются Джеффри Дин (Jeffrey Dean) и Санджай Гемават (Sanjay Ghemawat). В 2005 году Dough смог запустить Nutch на NDFS и использовал MapReduce. Это послужило зарождением Hadoop – платформы с открытым исходным кодом для хранения и обработки больших массивов данных. В январе 2006 года Hadoop был выделен в отдельный проект.

Таким образом, технология обработки больших данных началась с платформы Hadoop и парадигмы программирования MapReduce.

MapReduce – программная модель, предназначенная для распределенных вычислений. Реализован MapReduce преимущественно на языке Java. Алгоритм MapReduce включает в себя три основные стадии [1]:

1. Стадия Map. На данной стадии используется функция map(), которая берет заданный набор данных и преобразует его в другой набор данных, где отдельные элементы разбиваются на кортежи (пары «ключ/значение»).

2. Стадия Shuffle. На данной стадии происходит передача данных из Mapper в Reducer. Данная стадия необходима для Reducer, иначе будут отсутствовать входные данные. Эта стадия может начаться еще до завершения предыдущей, что существенно экономит время. Также происходит сортировка всех промежуточных пар «ключ-значение» по ключу. Таким образом, на одном рабочем узле лежат пары с одинаковым ключом.

3. Стадия Reduce. На данной стадии применяется функция Reduce, которая на вход получает ключ и список всех значений, которые были сгенерированы для этого ключа в качестве параметра. Ключи представлены в отсортированном порядке. Для каждого ключа вызывается функция Reduce, которая возвращает финальный результат. Таким образом, происходит свертывание данных и список преобразуется к единственному атомарному значению.

Программная модель MapReduce – сердце платформы Hadoop. Данная платформа представляет из себя целую экосистему, которая является масштабируемой, надежной средой, используемой для распределенных вычислений.

MapReduce был выпущен с начальными версиями платформы Hadoop. Но основной недостаток заключался в том, что фреймворк выполнял как задачу обработки, так и задачу управления ресурсами.

Map Reduce 2 – долгожданное обновление для методов, связанных с планированием и управлением ресурсами. Улучшения отделяют возможность управления ресурсами от логики, специфичной для MapReduce, и такое разделение было достигнуто благодаря появлению YARN в более поздних версиях Hadoop.

Ядро современной платформы Hadoop [1] представляет распределенная файловая система Hadoop (HDFS), Hadoop YARN, Hadoop MapReduce и

Hadoop Common (набор общих утилит и библиотек, которые поддерживают другие модули Hadoop).

Так как Hadoop представляет собой набор всех модулей, то он может включать в себя и другие языки программирования.

Hadoop решает проблемы, связанные с объемом и скоростью в горизонтальном масштабе. Он считается универсальным решением. Поэтому данная платформа стремительно набрала популярность. Помимо этого, Hadoop имеет открытый исходный код. Экосистема непрерывно развивалась в последние годы, что сделало ее максимально безошибочной.

Hadoop максимально эффективно работает с небольшим количеством больших файлов. Но данная платформа терпит неудачу, когда ставится задача обработки большого числа маленьких файлов. Также Hadoop осуществляет считывание и запись данных с диска, что делает операции чтения и записи весьма затратными.

Данная платформа поддерживает только механизм пакетной обработки. Hadoop не может производить вывод в режиме реального времени с низкой задержкой. Он работает только с данными, которые собираются и хранятся в файле заранее перед обработкой. Помимо этого, безопасность данной платформы является понятием неоднозначным. Hadoop использует аутентификацию Kerberos, которой сложно управлять.

Необходимость более быстрой обработки наборов данных привела к появлению Apache Spark. Это фреймворк с открытым исходным кодом, предназначенный для кластерных вычислений в режиме реального времени. В отличии от Hadoop, который основан на концепции пакетной обработки, Spark может обрабатывать данные в режиме реального времени и быть примерно в 100 раз быстрее. Основные особенности Spark: кластерные вычисления в памяти, параллелизм данных и отказоустойчивость. Архитектура основана на двух основных абстракциях: устойчивый распределенный набор данных (RDD) и направленный ациклический граф (DAG).

Экосистема Apache Spark включает в себя следующее [5]:

1. Spark Core – ядро, предназначенное для распределенной крупномасштабной и параллельной обработки данных. Основные задачи: управление памятью, устранение неисправностей, планирование, распределение и мониторинг заданий в кластере и взаимодействие с системами хранения.

2. Spark Streaming – компонент, который используется для обработки потоковых данных в реальном времени. Он обеспечивает высокую пропускную способность и отказоустойчивую обработку потоков данных.

3. GraphX – компонент, предназначенный для графов и выполнении параллельных вычислений над ними.

4. MLlib – библиотека машинного обучения.

5. Spark SQL – новый модуль в Spark, который объединяет реляционную обработку с API функционального программирования Spark. Он поддерживает запросы данных либо через SQL, либо через Hive Query Language.

6. SparkR – пакет R. Под R [2] здесь понимается как язык программирования, так и программная среда, предназначенная для работы со статистическими данными. Интегрированные среды разработки, такие как Eclipse и Visual Studio, поддерживают этот язык. Некоторые компании считают, что по своей популярности данный язык уже обогнал SQL.

Таким образом, Spark превосходит Hadoop, когда данные извлекаются из разных источников и необходимо получить потоковую обработку больших данных в режиме реального времени.

Из недостатков Spark можно перечислить следующее. Хранение данных в памяти может стать узким местом, когда речь идет об экономически эффективной обработке больших данных. Также гораздо удобнее запускать все на одном узле, Spark же требует распределения по нескольким кластерам. Spark потребляет огромное количество данных по сравнению с Hadoop.

Еще одна технология обработки больших данных – нереляционные базы данных NoSQL. Они предназначены для неструктурированных данных. Такие базы данных не требуют схем и определения типов данных. К основным

достоинствам можно еще причислить следующее: открытый исходный код, горизонтальную масштабируемость, распределенность, экономичность, высокую производительность, обработку данных в режиме реального времени и гибкость [4]. Примеры баз данных NoSQL: MongoDB, Redis и Cassandra.

Одна из разновидностей нереляционной базы данных – резидентная база данных, которая использует оперативную память для хранения данных. Резидентная база данных предназначена для достижения минимального времени отклика за счет исключения необходимости доступа к дискам. Идеально подходит для таких приложений как таблицы лидеров игр, торги в реальном времени и т.д.

Говоря об обработке больших данных, нельзя не упомянуть про искусственный интеллект. Практически любой современный продукт для обработки больших данных использует искусственный интеллект, который продвигается через машинное обучение [3]. Отдельно стоит выделить нейронные сети и глубинное обучение. Комбинация искусственного интеллекта и больших данных является причиной феноменального роста во многих отраслях.

Искусственный интеллект успешно применяется в аналитике больших данных. Это главным образом включает применение различных алгоритмов интеллектуального анализа к конкретному набору данных, которые затем помогут компании более эффективно принять решение. Типы аналитики больших данных: описательная (descriptive analytics), прогнозирующая аналитика (predictive analytics), предписывающая аналитика (prescriptive analytics), диагностирующая аналитика (diagnostic analytics).

Объединения технологий обработки больших данных с блокчейном дает отличные результаты. Самое большое преимущество блокчейна – это безопасность. Особенно эффективно применение данной комбинации для экономической отрасли, сферы государственного управления.

Благодаря быстрому росту данных и огромному стремлению организаций к повышению эффективности своей деятельности, технологии

обработки больших данных принесли на рынок столько зрелых идей и проектов. Это помогло решить многие бизнес-задачи и проблемы, а также качественным образом улучшить жизнь людей. Но объем информации до сих пор продолжает расти и есть риск, что те технологии, которые используются на данный момент времени в скором могут быстро потерять свою актуальность. Поэтому, можно сказать, что развитие технологий обработки больших данных находится в прямой зависимости от темпов роста данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. DeRoos Dirk. Hadoop For Dummies. Published by: John Wiley & Sons, Inc., 2014. 411 p.

2. Prasad Y. L. Big Data Analytics Made Easy. Published by: Notion Press, Inc.; 1 edition (December 3, 2016). 192 p.

3. Бринк Хенрик, Ричардс Джозеф, Феверолф Марк. Машинное обучение.-СПб.: Питер, 2017.-336 с.:ил. - (Серия «Библиотека программиста»).

4. Воронова, Л. И. Big Data. Методы и средства анализа: учебное пособие / Л. И. Воронова, В. И. Воронов. — Москва : Московский технический университет связи и информатики, 2016. — 33 с.

5. Изучаем Spark: молниеносный анализ данных / Карау Х., Конвински Э., Венделл П., Захария М. - М. : ДМК Пресс, 2015. - 304 с.

6. Силен Дэви, Мейсман Арно, Али Мохамед. Основы Data Science и Big Data. Python и наука о данных. - СПб.: Питер, 2017.-336 с.:ил. - (Серия «Библиотека программиста»).