Методологические рекомендации к практическим занятиям.

В качестве контроля на практических занятиях студенты закрепляют полученные на лекциях и самостоятельно изученные знания в виде собеседования. Каждая тема подразумевает свой список вопросов, представленных в рекомендациях к каждому разделу.

Критерии оценивания (модуль1):

20 баллов, если:

– изученный материал изложен полно, определения даны верно;

– ответ показывает понимание материала;

– обучающийся может обосновать свои суждения, применить знания на практике, привести необходимые примеры, не только по учебнику и конспекту, но и самостоятельно составленные.

12-16 баллов, если:

– изученный материал изложен достаточно полно;

– при ответе допускаются ошибки, заминки, которые обучающийся в состоянии исправить самостоятельно при наводящих вопросах;

– обучающийся затрудняется с ответами на 1-2 дополнительных вопроса.

4-8 балла, если:

– материал изложен неполно, с неточностями в определении понятий или формулировке определений;

– материал излагается непоследовательно;

–обучающийся не может достаточно глубоко и доказательно обосновать свои суждения и привести свои примеры;

– на 50% дополнительных вопросов даны неверные ответы.

0 баллов, если:

– при ответе обнаруживается полное незнание и непонимание изучаемого материала;

– материал излагается неуверенно, беспорядочно;

– даны неверные ответы более чем на 50% дополнительных вопросов.

Критерии оценивания (модуль2):

40 баллов, если:

– изученный материал изложен полно, определения даны верно;

– ответ показывает понимание материала;

– обучающийся может обосновать свои суждения, применить знания на практике, привести необходимые примеры, не только по учебнику и конспекту, но и самостоятельно составленные.

24-32 балла, если:

– изученный материал изложен достаточно полно;

– при ответе допускаются ошибки, заминки, которые обучающийся в состоянии исправить самостоятельно при наводящих вопросах;

– обучающийся затрудняется с ответами на 1-2 дополнительных вопроса.

8-16 баллов, если:

– материал изложен неполно, с неточностями в определении понятий или формулировке определений;

– материал излагается непоследовательно;

–обучающийся не может достаточно глубоко и доказательно обосновать свои суждения и привести свои примеры;

– на 50% дополнительных вопросов даны неверные ответы.

0 баллов, если:

– при ответе обнаруживается полное незнание и непонимание изучаемого материала;

– материал излагается неуверенно, беспорядочно;

– даны неверные ответы более чем на 50% дополнительных вопросов.

Содержание

Практическая работа 1.

Вычислительная сложность алгоритмов (4 часа)

Для закрепления пройденного материала по данному разделу предлагаются следующие вопросы для собеседования:

1. Какую роль играет понятие алгоритма в программировании?

   Определение алгоритма. Пример. Роль и место алгоритма в программировании.

2. Какие свойства алгоритма относятся к важнейшим?

   Описать основные свойства алгоритма. Обосновать, почему они считаются важнейшими.

3. Из каких элементов строятся блок-схемы?

   Что такое блок-схемы. Зарисовать элементы, описать их.

4. Как можно классифицировать алгоритмы?

   Описать известные классификации алгоритмов.

5. Что такое сложность алгоритма?

   Определение сложность алгоритма. Зачем ее определять. Какие показатели используются для оценки сложности.

6. Как можно классифицировать алгоритмы в соответствии с их временной сложностью?

   Описать классификацию алгоритмов по временной сложности. Привести примеры.

7. Вычислительная сложность алгоритма. Какая вычислительная сложность больше: константная, квадратичная, логарифмическая, экспоненциальная, факториальная? Обозначение вычислительной сложности алгоритма. Что такое вычислительная сложность в лучшем и худшем случае? Чем объясняется различная алгоритмическая сложность алгоритмов?

Студентам необходимо разобрать дома все вопросы, подготовить краткий конспект для ***устного ответа***.

Помимо лекционного материала, рекомендуется использовать для подготовки следующие ресурсы:

https://training.ru/#!/News/420?lang=ru – Оценка сложности алгоритмов

https://habr.com/ru/post/104219/ - Оценка сложности алгоритмов

http://window.edu.ru/resource/402/77402/files/alg_prog.pdf - Основы алгоритмизации и программирования.

В программировании, вычислительную сложность алгоритмов обычно оценивают по количеству действий, которые выполняет алгоритм и по количеству используемой памяти.

Чаще всего именно эти два критерия играют основную роль. Перед использованием алгоритма нужно правильно оценить, как именно сложность алгоритма будет зависеть от количества входных данных, чтобы «бесконечное» время работы алгоритма не стало сюрпризом в самый неподходящий момент.

Количество входных данных принято обозначать буквой ***n***. Важно понимать, что нам нужно оценить не то, сколько именно операций потребуется алгоритму при конкретном количестве входных данных, а то, как он себя поведет при увеличении количества входных данных. То есть, мы хотим получить функцию изменения количества операций, которые выполнит алгоритм, в зависимости от количества входных данных ***n***.

О-нотация

В подавляющем большинстве случаев для обозначения оценки сложности алгоритмов используют так называемую О-нотацию, в математике такое обозначение используют для сравнения асимптотического поведения функций.

О-нотация определяет функцию, назовем ее g(n), которая показывает, как будет изменяться вычислительная сложность алгоритма с изменением количества входных данных в худшем для алгоритма случае.

Пример «грубой» асимптотической оценки

n	log(n)	n log(n)	n^2	n^3	2^n	n!
10	1	10	100	1 000	1 024	3 628 800
100	2	200	10 000	1 000 000	1.E+30	9.E+157
1 000	3	3 000	1 000 000	1 000 000 000	1.E+301	очень много
10 000	4	40 000	100 000 000	1 000 000 000 000	очень много	очень много
100 000	5	500 000	10 000 000 000	1 000 000 000 000 000	очень много	очень много
1 000 000	6	6 000 000	1 000 000 000 000	1 000 000 000 000 000 000	очень много	очень много
10 000 000	7	70 000 000	100 000 000 000 000	1 000 000 000 000 000 000 000	очень много	очень много
100 000 000	8	800 000 000	10 000 000 000 000 000	1 000 000 000 000 000 000 000 000	очень много	очень много

Примеры анализа алгоритмов

Алгоритм поиска минимального элемента массива, приведенный выше, выполнит N итераций цикла. Трудоемкость каждой итерации не зависит от количества элементов массива, поэтому имеет сложность $T^{iter} = \mathcal{O}(1)$. В связи с этим, верхняя оценка всего алгоритма $T_n^{min} = \mathcal{O}(n) \cdot \mathcal{O}(1) = \mathcal{O}(n \cdot 1) = \mathcal{O}(n)$. Аналогично вычисляется нижняя оценка сложности, а в силу того, что она совпадает с верхней — можно утверждать $T_n^{min} = \Theta(n)$.

Алгоритм пузырьковой сортировки (bubble sort) использует два вложенных цикла. Во внутреннем последовательно сравниваются пары элементов и если оказывается, что элементы стоят в неправильном порядке — выполняется перестановка. Внешний цикл выполняется до тех пор, пока в массиве найдется хоть одна пара элементов, нарушающих требуемый порядок [2].

```
1.  начало; пузырьковая сортировка массива array из N элементов
2.  nPairs := N; количество пар элементов
3.  hasSwapped := false; пока что ни одна пара не нарушила порядок
4.  для всех i от 1 до nPairs-1 выполнять:
5.    если array[i] > array[i+1] то:
6.      swap(array[i], array[i+1]); обменять элементы местами
7.      hasSwapped := true; найдена перестановка
8.    nPairs := nPairs - 1; наибольший элемент гарантированно помещен в конец
9.   если hasSwapped = true - то перейти на п.3
10.  конец; массив array отсортирован
```

Трудоемкость функции swap не зависит от количества элементов в массиве, поэтому оценивается как $T^{swap} = \Theta(1)$. В результате выполнения внутреннего цикла, наибольший элемент смещается в конец массива неупорядоченной части, поэтому через N таких вызовов массив в любом случае окажется отсортирован. Если же массив отсортирован, то внутренний цикл будет выполнен лишь один раз.

Таким образом:

- $T_n^{bubble} = \mathcal{O}(\sum_{i=1}^{n} \sum_{j=1}^{n-i} 1) = \mathcal{O}(\sum_{i=1}^{n} n) = \mathcal{O}(n^2)$;
- $T_n^{bubble} = \Omega(1 \cdot \sum_{j=1}^{n-i} 1) = \Omega(n)$.

В **алгоритме сортировки выбором** массив мысленно разделяется на упорядоченную и необработанную части. На каждом шаге из неупорядоченной части массива выбирается минимальный элемент и добавляется в отсортированную часть [2].

```
1.  начало; selection_sort - сортировка массива array из N элементов методом выбора
2.  для всех i от 1 до N выполнять:
3.    imin := indMin(array, N, i)
4.    swap(array[i], array[imin])
5.  конец; массив array отсортирован
```

Для поиска наименьшего элемента неупорядоченной части массива используется функция indMin, принимающая массив, размер массива и номер позиции, начиная с которой нужно производить поиск. Анализ сложности этой функции можно выполнить аналогично тому, как это сделано для функции min — количество операций линейно зависит от количества обрабатываемых элементов: $T_{n,i}^{indMin} = \Theta(n — i)$.

У сортировки выбором нет ветвлений, которые могут внести различия в оценку наилучшего и наихудшего случаев, ее трудоемкость:
$T_n^{select} = \Theta(\sum_{i=1}^{n}(T_{n,i}^{indMin} + T^{swap})) = \Theta(\sum_{i=1}^{n}(n-i)) = \Theta(\frac{n-1}{2} \cdot n) = \Theta(n^2)$.

Практическая работа 2.

Структуры данных: списки, стеки, очереди.(8 часов)
Вопросы для собеседования:

1. Классификация структур данных. Классификация сложных структур по организации взаимосвязей между элементами.
2. Список. Виды списков. Способы задания списков. Почему используется класс при работе со списками, а не структура (struct) при реализации на языке C#. Какое действие нельзя выполнять со структурой?
3. Как определить список при помощи класса (одного и двух)? Практическое задание на разработку программного кода по этой части касаются работы со ссылками next, prev.
4. Стек, основные операции в стеке. Как реализовать стек, способы и их достоинства и недостатки?
5. Очередь. Добавление и удаление из очереди. Как реализовать очередь, способы и их достоинства и недостатки?
6. Переменные ссылочного типа и обычные. В чем разница? Задания по участку кода определить, какие переменные указаны.
7. Сортировка массивов. Три вида простых сортировок и их алгоритмическую сложность. Сортировка шелла и быстрая сортировка. Вычислительная сложность быстрой сортировки. Как работает алгоритм быстрой сортировки?

В качестве дополнения к теоретическим ответам, студентам предлагается программно реализовать работу со структурами данных рассматриваемого типа.

Допускается реализация на любом подходящем языке программирования. В ***задачах на реализацию предлагается***:

- Описать структуру типа список: два вида. Выполнить несколько простейших произвольных операций со списком. Рассказать о действиях, которые нельзя выполнять со списком.
- Описать структуру типа стек. Выполнить заполнение стека. Удаление элементов стека.
- Описать структуру типа очередь. Выполнить добавление элементов. Удаление элементов.
- Определить, какие переменные указаны:
  int a;
  int arr[5];
  char d = 's';
  bool k = true;

```
int *a = new int;
int *b = new int(5);
```

- **Что делает этот код?**

```
void Make_Single_List(int
n,Single_List** Head){
  if (n > 0) {
    (*Head) = new Single_List();
    cout << "Введите значение ";
    cin >> (*Head)->Data;
    (*Head)->Next=NULL;
     Make_Single_List(n-1,&((*Head)-
>Next));
  }
}
```

```
void
Print_Single_List(Single_List*
Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_Single_List(Head->Next);
  }
  else cout << "\n";
}
```

```
Single_List*
Delete_Item_Single_List(Single_List*
Head,
      int Number){
  Single_List *ptr;
  Single_List *Current = Head;
  for (int i = 1; i < Number && Current
!= NULL; i++)
    Current = Current->Next;
  if (Current != NULL){
    if (Current == Head){
      Head = Head->Next;
      delete(Current);
      Current = Head;
    }
    else {
      ptr = Head;
      while (ptr->Next != Current)
        ptr = ptr->Next;
      ptr->Next = Current->Next;
      delete(Current);
      Current=ptr;
    }
  }
  return Head;
}
```

```
bool
Find_Item_Single_List(Single_List*
Head, int DataItem){
  Single_List *ptr;
  ptr = Head;
  while (ptr != NULL){
    if  (DataItem  ==  ptr->Data)
return true;
    else ptr = ptr->Next;
  }
  return false;
}
```

```
void   Delete_Single_List(Single_List*
Head){
  if (Head != NULL){
    Delete_Single_List(Head->Next);
    delete Head;
  }
}
```

```
struct list {
               type elem;
               list *next, *pred;
                   }
list *headlist ;
```

- Подписать участки кода по их функциям.

```
//
void Make_Stack(int n, Stack* Top_Stack){
  if (n > 0) {
    int tmp;//вспомогательная переменная
    cout << "Введите значение ";
    cin >> tmp; //вводим значение информационного поля
    Push_Stack(tmp, Top_Stack);
    Make_Stack(n-1,Top_Stack);
  }
}

//
void Print_Stack(Stack* Top_Stack){
  Print_Single_List(Top_Stack->Top);
}

//
void Push_Stack(int NewElem, Stack* Top_Stack){
  Top_Stack->Top      =Insert_Item_Single_List(Top_Stack-
>Top,1,NewElem);
}

//
int Pop_Stack(Stack* Top_Stack){
  int NewElem = NULL;
  if (Top_Stack->Top != NULL) {
    NewElem = Top_Stack->Top->Data;
    Top_Stack->Top  =  Delete_Item_Single_List(Top_Stack-
>Top,0);
  }
  return NewElem;
}

//
bool Empty_Stack(Stack* Top_Stack){
  return Empty_Single_List(Top_Stack->Top);
}

//
void Clear_Stack(Stack* Top_Stack){
  Delete_Single_List(Top_Stack->Top);
}
```

В качестве последнего задания по теме предлагается найти ответы на следующие вопросы самостоятельно:

1. Что такое труднорешаемые задачи?
2. Что такое недетерминировано-полиномиальные алгоритмы?
3. К какому семейству языков относится Scheme? Чем он отличается от традиционных императивных языков?
4. Как происходит процесс разработки программ на Scheme?
5. Что такое REPL?
6. Как классифицируются выражения в Scheme?
7. Что такое комбинация?

Рекомендации к выполнению

Используя все доступные источники, поисковые инструменты, найти развернутые ответы на вопросы. В конце занятия представить их. Оценка в соответствии с критериями.

Практическая работа 3.

Структуры данных: графы(6 часов + 1 к.р.)

Данный раздел посвящен изучению графовых моделей, а также графов как структуры данных. Вопросы к собеседованию:

1. Графы. Определение. Способы задания графа. Чем граф отличается от дерева?
2. Что такое циклический граф, ориентированный и неориентированный? Ориентированный и неориентированный граф. Взвешенный граф.
3. Поиск в глубину и ширину.
4. Кратчайший путь в графе от вершины. Алгоритм Дейкстры. Практические задания касаются итераций работы алгоритма на примере.

В дополнение к вопросам студентам необходимо решить следующие задачи.

Задача 1. На рисунке справа схема дорог Н-ского района изображена в виде графа; в таблице слева содержатся сведения о протяжённости каждой из этих дорог (в километрах).

	П1	П2	П3	П4	П5	П6
П1		10			8	5
П2	10			20	12	
П3				4		
П4		20	4		15	
П5	8	12		15		7
П6	5				7	

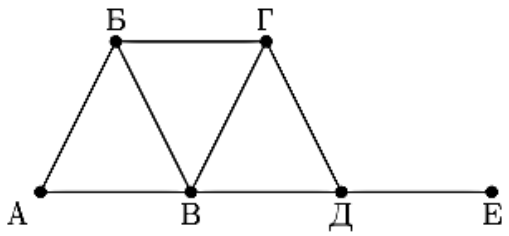


Рис. 4: Граф дорог Н-ского района

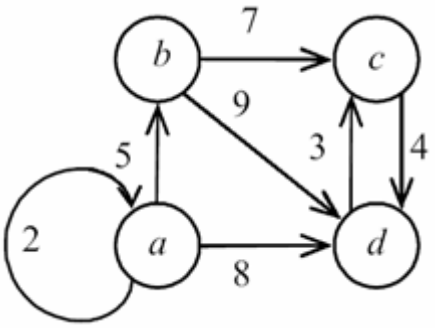
Так как таблицу и схему рисовали независимо друг от друга, то нумерация населённых пунктов в таблице никак не связана с буквенными обозначениями на графе. Определите, какова протяжённость дороги из пункта Б в пункт В. В ответе запишите целое число — так, как оно указано в таблице.

Задача 2. Построить граф, у которого вершины имеют следующие степени:

А – 7, Б – 3, С – 1

Задача 3. Построить Эйлеров граф

Задача 4. К заданному графу построить матрицу смежности, матрицу весов, инцидентности и список ребер



Задача 5. Привести пример Эйлерова графа.

Задача 6. Привести пример графа, имеющего Эйлеров путь.

Практические задания на программную реализацию.

1. Написать код для списка смежности
2. Реализовать матрицу смежности
3. Реализовать поиск в глубину
4. Реализовать поиск в ширину
5. Написать программу поиска кратчайшего пути
6. Реализовать Алгоритм Дейкстры на произвольном примере. Код функции см.ниже

```
void Dijkstra(int n, int **Graph, int Node){
  bool *S = new bool[n];
  int *D = new int[n];
  int *P = new int[n];
  int i, j;
  int Max_Sum = 0;
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      Max_Sum += Graph[i][j];
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      if (Graph[i][j] == 0)
        Graph[i][j] = Max_Sum;
  for (i = 0 ; i < n ; i++){
    S[i] = false;
    P[i] = Node;
    D[i] = Graph[Node][i];
  }
  S[Node] = true;
  P[Node] = -1;
  for ( i = 0 ; i < n - 1 ; i++ ){
    int w = 0;
    for ( j = 1 ; j < n ; j++ ){
      if (!S[w]){
        if (!S[j] && D[j] <= D[w])
          w = j;
      }
      else w++;
    }
    S[w] = true;
    for ( j = 1 ; j < n ; j++ )
```

```
      if (!S[j])
        if (D[w] + Graph[w][j] < D[j]){
          D[j] = D[w] + Graph[w][j];
          P[j] = w;
        }
  }
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",D[i]);
  cout << endl;
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",P[i]+1);
  cout << endl;
  delete [] P;
  delete [] D;
  delete [] S;
}
```

Для подготовки к контрольной работе рекомендуется повторить следующие разделы:

- Структуры данных: списки, работа с односвязными и двусвязными списками
- Графы: виды графов, заданиче графа с помощью матриц. Алгоритм Дейкстры(по шагам и итерациям)
- Пространства имен.

Справочный материал

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут обращаться к членам с помощью полного имени для каждого идентификатора, например std::vector<std::string> vec; , или else с std::vector<std::string> vec; для одного идентификатора (using std::string) или using std::string для всех идентификаторов в пространстве имен (using namespace std;). Код в файлах заголовков всегда должен содержать полное имя в пространстве имен. В следующем примере показано объявление пространства имен и продемонстрированы три способа доступа к членам пространства имен из кода за его пределами. (https://docs.microsoft.com/ru-ru/cpp/cpp/namespaces-cpp?view=msvc-170)

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Практическая работа 4. (4 часа)

Распараллеливание алгоритмов

Параллелизм на уровне алгоритмов

Данный вид параллелизма предполагает замену последовательных алгоритмов некоторых вычислений на параллельные. Это касается алгоритмов поиска, сортировки и т.п. Организация процесса распараллеливания осуществляется за счет использования различных средств параллельного программирования, таких как, специальные библиотеки, переменные окружения, директивы компилятора и т.п. Примером может служить технология OpenMP.

Параллельный алгоритм – алгоритм, предназначенный для реализации на параллельной вычислительной системе.

Parallel **Extensions** to the *.NET* Framework (другие названия - *Parallel* FX *Library*, PFX) - это библиотека, разработанная фирмой Microsoft, для использования в программах на базе управляемого (managed) кода. Она позволяет распараллеливать задачи, в которых могут использоваться специальные - координирующие (*coordinating*) - структуры данных. Тем самым, библиотека PFX упрощает написание *параллельных программ*, обеспечивая *увеличение производительности* при увеличении числа ядер или числа процессоров, исключая многие сложности современных моделей *параллельного программирования*. Первая версия библиотеки была представлена 29 ноября 2007 года, впоследствии выходили обновления в декабре 2007 и июне 2008 годов. На момент написания данных лекций, библиотека PFX вошла в состав *.NET* 4 *CTP* и *Visual* Studio 2008/2010.

Parallel **Extensions** обеспечивает несколько новых способов организации *параллелизма*:

- *Параллелизм* при декларативной обработке данных. Реализуется при помощи параллельного интегрированного языка запросов (PLINQ) - параллельной реализации *LINQ*. Отличие от *LINQ* заключается в том, что запросы выполняются параллельно, обеспечивая *масштабируемость* и загрузку доступных ядер и процессоров.
- *Параллелизм* при *императивной* обработке данных. Реализуется при помощи библиотечных реализаций параллельных вариантов основных *итеративных* операций над данными, таких как циклы for и foreach. Их выполнение автоматически распределяется на все доступные ядра/процессоры *вычислительной системы*.
- *Параллелизм* на уровне задач. Библиотека *Parallel* Extensions обеспечивает высокоуровневую работу с пулом рабочих потоков, позволяя явно структурировать параллельно *исполняющийся код* с помощью легковесных задач. *Планировщик* библиотеки *Parallel* Extensions выполняет диспетчеризацию и управление исполнением этих задач, а также единообразный механизм обработки *исключительных ситуаций*.

Parallel Extensions - это управляемая (managed) библиотека и для своей работы она требует установленный *.NET* Framework 3.5.

Базовые принципы разработки распараллеливания алгоритмов на центральном процессоре

Разработка параллельных алгоритмов состоит из следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо. 2. Определение для сформированного набора подзадач информационных взаимодействий.

3. Масштабирование подзадач, определение количества процессоров.

4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

Этапы 1-4 могут повторяться, в случае необходимости, например для улучшения эффективности алгоритма. Если желаемые показатели не достигаются, то следует изменить математическую модель задачи. Приведенная схема является общей, в этой связи в каждом конкретном случае последовательность этапов может меняться. Например, если заранее не известно точное число процессоров, но известны границы решающего поля, разработку алгоритма можно начать с масштабирования базового набора задач и только потом перейти к декомпозиции и выявление связей по информации. Схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений приведена на рисунке.

На этапе декомпозиции осуществляется выделение базовых подзадач. В процессе декомпозиции предъявляются минимальные требования обеспечить:

- примерно равный объем вычислений в выделяемых подзадачах;
- минимальный информационный обмен данными между процессорами. На этапе анализа информационных зависимостей между подзадачами различают следующие типы этих зависимостей:
- локальные (на соседних процессорах) и глобальные (в которых принимают участие все процессоры) схемы передачи данных;
- структурные (соответствующие типовым топологиям коммуникаций) и произвольные способы взаимодействия;

- статические (задаваемые на этапе проектирования) или динамические (определяемые в ходе выполняемых вычислений);

- синхронные (следующая операция выполняется после выполнения предыдущей операции всеми процессорами) и асинхронные способы взаимодействия (процессы могут не дожидаться полного завершения действий по передаче данных) подзадачи обладают высокой степенью информационной взаимозависимости.

Этап масштабирования параллельного алгоритма выполняется в случае, если количество подзадач (областей данных) отличается от числа процессоров. Тогда осуществляется переход на этап декомпозиции. При этом, число подзадач уменьшают за счет укрупнения области исходных данных. В первую очередь следует объединить области, для которых подзадачи обладают высокой степенью информационной взаимозависимости.

На этапе закрепления задач за процессорами следует учитывать информационные взаимодействия между областями данных этих задач. Такие задачи целесообразно размещать на процессорах, связанных прямыми линиями передачи данных.

Приведенная схема может использоваться для построения параллельного алгоритма, который характеризуется параллелизмом данных и параллелизмом задач. Если имеет место параллелизм данных, то задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Должно соблюдаться требование равномерная загрузка процессоров.

При этом следует учитывать возможность различной производительности процессоров. Эффективность параллельной программы зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных. Вычислительная задача разбивается на несколько самостоятельных подзадач в том случае если в ней отсутствует параллелизм по данным. Каждый процессор занимается решением отдельной подзадачи. В данном случае имеет место параллелизм задач. Количество задач влияет на количество процессоров. При обеспечении равномерной загрузки процессоров и минимизации обмена данными между ними можно ожидать значительного ускорения. Эффективность кода предполагает анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсоемких частей.

Платформа .NET Framework — это технология, которая поддерживает создание и выполнение веб-служб и приложений Windows. При разработке платформы .NET Framework учитывались следующие цели.

- Обеспечение согласованной объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, для локального выполнения кода, распределенного в Интернете, либо для удаленного выполнения.
- Предоставление среды выполнения кода, в которой:

    - сведена к минимуму вероятность конфликтов в процессе развертывания программного обеспечения и управления его версиями;
    - гарантируется безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним изготовителем;
    - исключаются проблемы с производительностью сред выполнения скриптов или интерпретируемого кода;
- обеспечиваются единые принципы разработки для разных типов приложений, таких как приложения Windows и веб-приложения;
- обеспечивается взаимодействие на основе промышленных стандартов, которое гарантирует интеграцию кода платформы .NET Framework с любым другим кодом.

Платформа .NET Framework состоит из общеязыковой среды выполнения (среды CLR) и библиотеки классов .NET Framework. Основой платформы .NET Framework является среда CLR. Среду выполнения можно считать агентом, который управляет кодом во время выполнения и предоставляет основные службы, такие как управление памятью, управление потоками и удаленное взаимодействие. При этом средой накладываются условия строгой типизации и другие виды проверки точности кода, обеспечивающие безопасность и надежность. Фактически основной задачей среды выполнения является управление кодом. Код, который обращается к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. Библиотека классов является комплексной объектно-ориентированной коллекцией повторно используемых типов, которые применяются для разработки приложений — начиная с обычных приложений, запускаемых из командной строки, и приложений с графическим интерфейсом (GUI) и заканчивая приложениями, использующими последние технологические возможности ASP.NET, такие как веб-формы и веб-службы XML.

Платформа .NET Framework может размещаться неуправляемыми компонентами, которые загружают среду CLR в собственные процессы и запускают выполнение управляемого кода, создавая таким образом программную среду, позволяющую использовать средства как управляемого, так и неуправляемого выполнения. Платформа .NET Framework не только предоставляет несколько базовых сред выполнения, но также поддерживает разработку базовых сред выполнения независимыми производителями.

Например, ASP.NET размещает среду выполнения и обеспечивает масштабируемую среду для управляемого кода на стороне сервера. ASP.NET работает непосредственно со средой выполнения, чтобы обеспечить выполнение приложений ASP.NET и веб-служб XML, обсуждаемых ниже в этой статье.

Список вопросов к собеседованию:

1. Что такое специальная форма? Приведите примеры специальных форм.
2. Что такое процедура-предикат?
3. Понятие рекурсии. Перечислите виды рекурсии.
4. Чем опасна древовидная рекурсия?
5. Что такое функция высших порядков в Scheme и для чего её можно использовать?
6. Перечислите виды окружений.
7. Нормальный и аппликативный порядок вычислений.
8. .NetFramework Платформа. Каким образом достигается возможность разработки кроссплатформенных приложений? Код MSIL, native код, JIT компилятор.

Практическая работа 5.

Алгоритмы для анализа социальных сетей(6 часов)

Силовые алгоритмы визуализации графов — класс алгоритмов визуализации графов в эстетически приятном виде. Их цель — расположить узлы графа в двумерном или трёхмерном пространстве так, что все рёбра имели бы более-менее одинаковую длину, и свести к минимуму число пересечений рёбер путём назначения сил для множества рёбер и узлов основываясь на их относительных положениях, а затем путём использования этих сил либо для моделирования движения рёбер и узлов, либо для минимизации их энергии.

Как только силы на узлах и рёбрах определены, поведение всего графа под действием этих сил может быть итеративно промоделировано, как если бы это была физическая система. В такой ситуации силы, действующие на узлы, пытаются стянуть их ближе или оттолкнуть их друг от друга подальше. Это продолжается, пока система не придёт в состояние механического равновесия, то есть положение узлов не меняется от итерации к итерации. Положение узлов в этом состоянии равновесия используется для генерации рисунка графа.

Для сил, определённых из пружин, идеальная длина которых пропорциональна расстоянию в графе, мажорирование стресса даёт очень хорошее поведение (то есть монотонную сходимость) и математически элегантный путь минимизации этой разницы и, следовательно, к хорошему размещению вершин графа.

Можно также использовать механизмы, которые ищут минимум энергии более прямо, а не по физической модели. Такие механизмы, являющиеся примерами общих методов глобальной оптимизации, включают имитацию отжига и генетические алгоритмы.

Следующие свойства являются наиболее важными ***преимуществами*** силовых алгоритмов:

- Результаты хорошего качества

По меньшей мере для графов среднего размера (до 50—500 вершин), полученные результаты обычно имеют очень хорошие рисунки графов по следующим критериям: однородность длин рёбер, равномерное распределение вершин и симметрия. Последний критерий наиболее важен и трудно достижим в других типах алгоритмов.

- Гибкость

Силовые алгоритмы могут быть легко приспособлены и расширены для дополнительных эстетических критериев. Это делает алгоритмы более универсальными классами алгоритмов визуализации графов. Примерами существующих расширений являются алгоритмы для ориентированных графов, визуализация трёхмерных графов[6], кластерная визуализация графов, визуализация графов с ограничениями и динамическая визуализация графов.

- Интуитивность

Поскольку алгоритмы основаны на физических аналогах привычных объектов, наподобие пружин, поведение алгоритмов относительно просто предсказать и понять. Этого нет в других типах алгоритмов визуализации графов.

- Простота

Типичные силовые алгоритмы просты и могут быть реализованы в несколько строк кода. Другие классы алгоритмов визуализации, такие как алгоритмы на основе ортогональных размещений, обычно требуют куда больше работы.

- Интерактивность

Ещё одним преимуществом этого класса алгоритмов является аспект интерактивности. При рисовании промежуточных этапов графа пользователь может проследить, как меняется граф, прослеживая эволюцию от беспорядочного месива в хорошо выглядящую конфигурацию. В некоторых средствах интерактивного рисования графа пользователь может отбросить один или несколько узлов из состояния равновесия и наблюдать миграцию узлов в новое состояние равновесия. Это даёт алгоритмам преимущество для динамических и онлайновых систем визуализации графов.

- Строгая теоретическая поддержка

В то время как простые силовые алгоритмы часто появляются в литературе и на практике (поскольку они относительно просты и понятны), начинает возрастать число более обоснованных подходов. Статистики решали подобные задачи в многомерном шкалировании (англ. multidimensional scaling, MDS) с 1930-х годов, а физики также имеют длинную историю работы со связанными задачами моделирования движения n тел, так что существуют вполне вызревшие подходы. Как пример, подход мажорирования стресса к метрическим MDS может быть применен для визуализации графа и в этом случае можно доказать монотонную сходимость. Монотонная сходимость, свойство, что алгоритм будет на каждой итерации уменьшать напряжение или цену размещения вершин, важно, поскольку это гарантирует, что размещение, в конечном счёте, достигнет локального минимума и алгоритм остановится. Глушение колебаний приводит к остановке алгоритма, но не гарантирует, что будет достигнут истинный локальный минимум.

Программные приложения для анализа социальных сетей

Для анализа социальных сетей существует множество приложений для моделирования взаимодействий и процессов в сети, для вычисления определенных параметров сети и для визуализации графа сети.

Например, приложения по визуализации сети ВКонтакте (см. http://www.yasiv.com/vk) или Facebook (http://www.touchgraph.com/facebook). В них используются различные методы и алгоритмы, которые описаны ранее в данной работе. К наиболее известным средствам автоматического анализа социальных взаимодействий относятся: NetMiner (http://www.netminer.com/index.php), NetworkX (http://networkx.lanl.gov), SNAP (http://snap.stanford.edu), UCINet (http://www.analytictech.com/ucinet), Pajek (http:// vlado.fmf.uni-lj.si/pub/networks/pajek), ORA (см. http://www.casos.cs.cmu.edu/projects/ora), Cytoscape (http://www.cytoscape.org) и др. Для подобных приложений важным требованием является возможность

обрабатывать очень большое количество данных. В связи с этим процесс обработки часто распараллеливают. Существуют приложения, которые моделируют «теорию шести рукопожатий», которые выстраивают цепочку из связей (друзей) между двумя пользователями сети: для русскоязычной сети ВКонтакте (http://ienot.ru/hand), для англоязычных сетей (http://www. sixdegrees.org, http://sixdegrees.com). Эти цепочки, как правило, действительно получаются небольшой длины.

Аспектный анализ тональности (aspect-based sentiment analysis) — это подвид анализа тональности, чья задача заключается в определении отношения к конкретному аспекту основного предмета обсуждения. Все подходы к анализу тональности можно разделить на три группы.

Первая — ***подходы на основе правил (rule-based).*** Чаще всего в них используются вручную заданные правила классификации и эмоционально размеченные словари. Эти правила обычно на основе эмоциональности ключевых слов и их совместного использования с другими ключевыми словами рассчитывают класс текста. Несмотря на высокую эффективность в текстах из какой-то определенной тематики, методы на основе правил плохо обобщают. Кроме того, они крайне трудоёмки в создании, особенно когда нет доступа к подходящему словарю настроений. Последнее особенно характерно для русского языка, потому что на нем не так много источников, как на английском, особенно в сфере анализа тональности. Крупнейшие русскоязычные словари настроений — RuSentiment и LINIS Crowd. Но в них есть только информация о тональности от позитивной до негативной, без характеристик эмоций. Таким образом, не существует альтернатив таким мощным англоязычным подборкам с обширными эмоциональными характеристиками, как SenticNet, SentiWordNet и SentiWords.

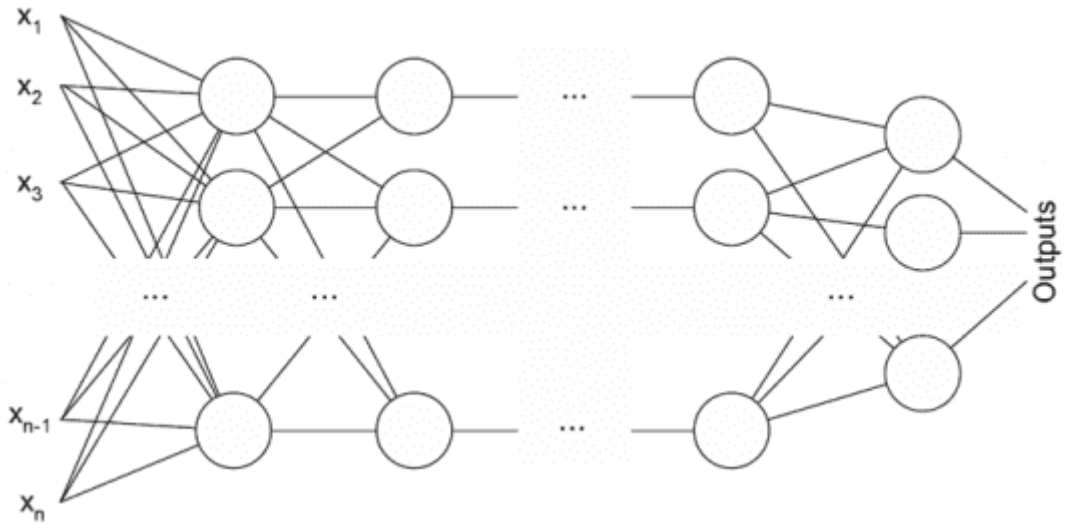
Вторая группа — подходы на основе машинного обучения. Они используют автоматическое извлечение признаков из текста и применение алгоритмов машинного обучения. Классическими алгоритмами классификации полярности являются ***наивный байесовский классификатор, дерево решений, логистическая регрессия и метод опорных векторов***. В последние годы внимание исследователей привлекают методы глубокого обучения, которые значительно превосходят традиционные методы в анализе тональности.

В простых подходах для представления текста в векторном пространстве обычно используется модель «мешок слов» (bag of words). В более сложных системах для генерирования эмбеддингов слов применяются модели дистрибутивной семантики, например, Word2Vec, GloVe или FastText. Также есть алгоритмы генерирования эмбеддингов на уровне предложений или параграфов, которые предназначены для переноса обучения в разных задачах обработки естественного языка. К таким алгоритмам относятся ELMo, Universal Sentence Encoder, BERT, ERNIE и XLNet. Одним из их главных недостатков с точки зрения генерирования эмбеддингов является потребность в больших массивах текстов для обучения. Это справедливо для всех методов машинного обучения, потому что всем алгоритмам обучения с учителем нужны для обучения размеченные наборы данных.

Третья группа — гибридные подходы. Они объединяют в себе подходы двух предыдущих видов. Например, гибридный фреймворк для анализа тональности персидского языка, в котором сочетаются лингвистические правила, а также модули свёрточных нейросетей и LSTM для классификации настроений. В гибридной модели аспектного анализа ALDONAr сочетаются онтология настроений для захвата информации о настроениях, BERT для получения эмбеддингов слов и два слоя CNN для расширенной классификации тональности.

Архитектуры нейронных сетей для решения задач NLP

Многослойный персептрон



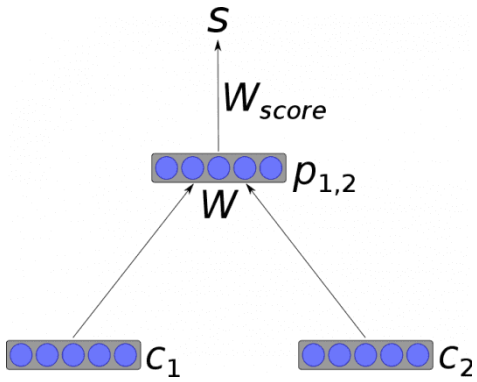
Многослойный персептрон состоит из 3 или более слоев. Он использует нелинейную функцию активации, часто тангенциальную или логистическую, которая позволяет классифицировать линейно неразделимые данные. Каждый узел в слое соединен с каждым узлом в последующем слое, что делает сеть полностью связанной. Такая архитектура находит применение в задачах распознавания речи и машинном переводе.

Сверточная нейронная сеть (Convolutional neural network, CNN) содержит один или более объединенных или соединенных сверточных слоев. CNN использует вариацию многослойного персептрона. Сверточные слои используют операцию свертки для входных данных и передают результат в следующий слой. Эта операция позволяет сети быть глубже с меньшим количеством параметров.

Семантический разбор, поиск парафраз, распознавание речи — тоже приложения CNN.

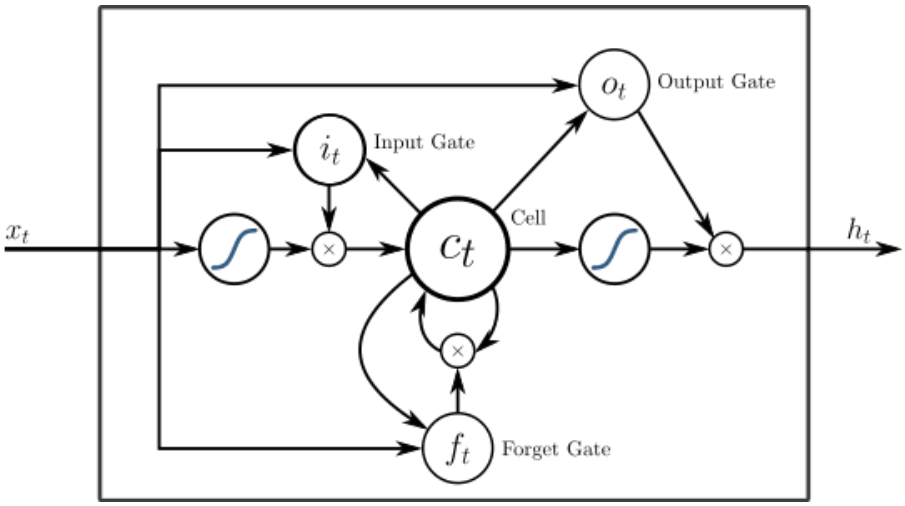
Рекурсивная нейронная сеть — тип глубокой нейронной сети, сформированный при применении одних и тех же наборов весов рекурсивно над структурой, чтобы сделать скалярное или структурированное предсказание над

входной структурой переменного размера через активацию структуры в топологическом порядке. В простейшей архитектуре нелинейность, такая как тангенциальная функция активации, и матрица весов, разделяемая всей сетью, используются для объединения узлов в родительские объекты.



Рекуррентная нейронная сеть, в отличие от прямой нейронной сети, является вариантом рекурсивной искусственной нейронной сети, в которой связи между нейронами — направленные циклы. Последнее означает, что выходная информация зависит не только от текущего входа, но также от состояний нейрона на предыдущем шаге. Такая память позволяет пользователям решать задачи NLP: распознание рукописного текста или речи.

Сеть долгой краткосрочной памяти (Long Short-Term Memory, LSTM) — разновидность архитектуры рекуррентной нейросети, созданная для более точного моделирования временных последовательностей и их долгосрочных зависимостей, чем традиционная рекуррентная сеть. LSTM-сеть не использует функцию активации в рекуррентных компонентах, сохраненные значения не модифицируются, а градиент не стремится исчезнуть во время тренировки. Часто LSTM применяется в блоках по несколько элементов. Эти блоки состоят из 3 или 4 затворов (например, входного, выходного и гейта забывания), которые контролируют построение информационного потока по логистической функции.



Практическая работа 6.

Хранение и обработка больших данных.(4 часа+2 к.р)

Понятие «big data» (большие данные) появилось в 2008 году, но еще до появления определения с большими данными уже встречались. Например, бизнес-аналитики компании «ВымпелКом» работали с big data в 2005 году, как утверждает Виктор Булгаков, руководитель департамента управленческой информации.

Чтобы точнее понять, относятся ли данные к big data или нет, смотрят на свойства информации (свойства определила Meta Group в 2001 году):

- Volume — объем (около 1 Петабайт).
- Velocity — регулярное обновление.
- Variety — данные могут быть не структурированы или иметь разнородные форматы.

К перечисленным факторам часто добавляют еще два:

- Variability (изменчивость) — всплески и спады данных, которые требуют определенных технологий для обработки.
- Value — различная сложность информации. Например, у данных о пользователях соцсетей и информации о транзакциях в банковской системе, разный уровень сложности.

Источниками могут быть:

- интернет — от соцсетей и СМИ до интернета вещей (IoT);
- корпоративные данные: логи, транзакции, архивы;
- другие устройства, которые собирают информацию, например, «умные колонки».

Сбор. Технологии и сам процесс сбора данных называют **дата майнингом** (data mining).

Сервисы, с помощью которых проводят сбор — это, например, Vertica, Tableau, Power BI, Qlik. Собранные данные могут быть в разных форматах: текст, Excel-таблицы, SAS.

В процессе сбора система находит Петабайты информации, которая после будет обработана **методами интеллектуального анализа**, который выявляет закономерности. К ним относят нейронные сети, алгоритмы кластеризации, алгоритмы обнаружения ассоциативных связей между событиями, деревья решений, и некоторые методы machine learning.

Кратко процесс сбора и обработки информации выглядит так:

- аналитическая программа получает задачу;
- система собирает нужную информацию, одновременно подготавливая её: удаляет нерелевантную, очищает от мусора, декодирует;
- выбирается модель или алгоритм для анализа;
- программа учится алгоритму и анализирует найденные закономерности.

Как хранят Big Data

Чаще всего «сырые» данные хранят в data lake — «озере данных». При этом хранят в разных форматах и степенях структурированности:

- строки и колонки из БД — структурные;
- CSV, XML, JSON-файлы, логи — полуструктурированные;
- документы, почтовые сообщения, pdf — неструктурированные;
- видео, аудио и изображения — бинарные.

Для хранения и обработки информации в data lake используют разные инструменты:

- **Hadoop** — платформа управления данными. Содержит один или несколько кластеров. Обычно используется для обработки, хранения и анализа больших объемов нереляционных данных: файлов журналов, записей интернет-трафика, данных датчиков, объектов JSON, изображений и сообщений в соцсетях.
- **HPPC (DAS)** — разработка LexisNexis Risk Solutions. Это суперкомпьютер, который обрабатывает информацию как в пакетном режиме, так и в режиме реального времени.
- **Storm** — фреймворк для обработки информации в реальном времени, разработан на Clojure.

Data lake — это не только хранилище. «Озеро» может включать в себя и программную платформу, например, Hadoop, кластеры серверов хранения и обработки данных, средства интеграции с источниками и потребителями информации и системы подготовки данных, управления и иногда инструментов машинного обучения. Также «озеро данных» можно масштабировать до тысяч серверов без остановки кластера.

Из озера информация поступает уже в «песочницы» — области исследования данных. На этом этапе разрабатываются сценарии для решения разных бизнес-задач.

Data lake чаще располагают в облаке, чем на собственных серверах. Для обработки big data нужны большие вычислительные мощности, а облачные технологии позволяют удешевить работу, поэтому компании прибегают к этим хранилищам.

Облачные технологии могут стать альтернативой собственному дата-сервису, потому что тяжело предсказать точную нагрузку на инфраструктуру. Если купить оборудование «про запас», то оно простаивает и приносит убытки. А если оборудование будет маломощным, то не хватит для хранения и обработки.

- Облако может хранить больше данных, чем физические серверы: место для хранения информации не закончится.
- Компания может создать собственную облачную структуру или взять в аренду мощности у провайдера.
- Облако экономически выгодно для компаний с быстро растущей нагрузкой или бизнесов, где часто тестируются различные гипотезы.

Когда данные получены и сохранены, их нужно проанализировать и представить в понятном для клиента виде: графики, таблицы, изображения или готовые алгоритмы. Из-за объема и сложности в обработке традиционные способы не подходят. С большими данными необходимо:

- обрабатывать весь массив данных (а это Петабайты);

- искать корреляции по всему массиву (в том числе скрытые);
- обрабатывать и анализировать информацию в реальном времени.

Поэтому для работы с big data разработаны отдельные технологии.

Технологии

Изначально это средства обработки неопределенно структурированных данных: СУБД NoSQL, алгоритмы MapReduce, Hadoop.

MapReduce — фреймворк для параллельных вычислений очень больших наборов данных (до нескольких Петабайт). Разработан Google (2004 год).

NoSQL (от англ. Not Only SQL, не только SQL). Помогает работать с разрозненными данными, решает проблемы масштабируемости и доступности с помощью атомарности и согласованности данных.

Hadoop — проект фонда Apache Software Foundation. Это набор утилит, библиотек и фреймворков, который служит для разработки и выполнения распределенных программ, работающих на кластерах из сотен и тысяч узлов. О нём уже говорили, но это потому, что без Hadoop не обходится практически ни один проект связанный с большими данными.

Также к технологиям относят языки программирования R и Python, продукты Apache.

Методы и средства работы с большими данными

Это дата майнинг, машинное обучение, краудсорсинг, прогнозная аналитика, визуализация, имитационное моделирование. Методик десятки:

- смешение и интеграция разнородных данных, например, цифровая обработка сигналов;
- прогнозная аналитика — использует данные за прошлые периоды и прогнозирует события в будущем;
- имитационное моделирование — строит модели, которые описывают процессы, как если бы они происходили в действительности;
- пространственный и статистический анализ;
- визуализация аналитических данных: рисунки, графики, диаграммы, таблицы.

Организация системы хранения данных

СХД должна быть масштабируемой, то есть гибкой, отказо- и катастрофоустойчивой. Необходимо обеспечивать ее соответствие стандартам и требованиям информационной и физической безопасности.

В случаях, когда требуется хранение больших объемов данных, важно не просто создать СХД, но и сделать ее оптимальной для решения конкретных задач компании.

Варианты подключений

- «Внутреннее» (подключения устройств и жестких дисков внутри одного хранилища: SCSI, Serial Attached SCSI (SAS), Serial ATA (SATA), Fibre Channel (FC). Накопитель устанавливается непосредственно на сервер.
- «Внешнее» (FC, Fibre Channel over Ethernet (FCoE), SCSI, iSCSI. Накопитель подключается к серверу с помощью шины).

- Кластерное (Infiniband). Подключение, организованное на основе кластеров (подсетей). Позволяет передавать данные с высокими скоростями за счет оптимальной маршрутизации

Основные элементы

СХД состоит из накопителей информации, серверов, инфраструктуры, обеспечивающей связь между ними, и системы управления.

Типы СХД

Системы хранения данных по типу накопителей информации делятся на три больших группы.

- **Дисковые**. Используются самые первые, распространенные и недорогие накопители. В современных условиях существенным недостатком становится то, что скорость передачи информации ограничивается скоростью вращения шпинделя, на котором закреплены пластины жесткого диска, однако современные дисковые СХД очень экономичные и «умные» в сравнении с их предшественниками.
- **Ленточные (кассетные)**. Мобильность кассет в сочетании с возможностью длительного хранения и восстановления информации делают их популярным средством для создания надежного электронного архива с физическим ограничением доступа к информации. Широко используются в мультимедийных библиотеках, где особенно важна низкая стоимость терабайта информации.
- **Флэш**. Полупроводниковые накопители отличаются высочайшей скоростью работы. Если у жесткого диска на обработку запроса уходит в среднем 6–7 мс, то для флэш-накопителей этот показатель достигает 0,1 мс. Таким образом, количество транзакций в секунду возрастает на 1–2 порядка. До недавнего времени флэш-накопители считались дорогими и использовались в гибридных системах вместе с дисковыми. Сейчас ситуация меняется и все чаще внедряются СХД полностью на флэш-накопителях, которые позволяют существенно сэкономить пространство серверов.

Говоря о технологиях хранения, невозможно обойти вниманием термин RAID. Redundant array of independent disks — избыточный массив независимых дисков — это технология виртуализации данных, которая объединяет несколько дисков в логический элемент для повышения производительности. В зависимости от выбранного типа RAID, технологии хранения делятся на два класса:

- **С использованием аппаратного RAID**. Более дорогое и не всегда оправданное решение, связанное с покупкой дополнительного компьютерного «железа» с собственной памятью и выделенным процессором. Аппаратный RAID требуется при наличии в системе как минимум четырех и более накопителей.
- **С использованием программного RAID**. В этой технологии используются контроллеры на материнской плате, которые не имеют своей памяти и выделенного процессора. Они используют от 2-5% ресурсов центрального процессора сервера. Не менее надежны, чем аппаратные решения, используются в небольших системах.

Устройства хранения

- **DAS**. Накопители ставятся непосредственно в сервер для получения дополнительного пространства со сравнительно быстрым доступом. Самый простой и недорогой вариант.
- **NAS**. Хранилище, подключаемое по сети. Отличается гибкостью и централизованным управлением, однако скорость доступа ограничена скоростью сети.
- **SAN**. Хранилище, подключаемое через оптико-волоконный кабель. Сочетает в себе все плюсы NAS с высокой скоростью доступа.
- **CAS.** Контентно-адресуемое хранилище данных — это аппаратно-программный комплекс, который позволяет накапливать огромное количество информации для долговременного хранения и обеспечивать доступ к данным по контентной адресации (образ данных хешируется и хеш используется для его нахождения). Основное применение CAS — это системы архивного, долговременного и неизменяемого хранения.
- **HSM.** Иерархическая система хранения данных — это технология хранения данных, которая позволяет автоматически переносить данные с «быстрых» дисков на «медленные» и обратно, когда это требуется вычислительным системам, тем самым обеспечивая снижение себестоимости хранения одного байта. Данные могут быть также перенесены на ленточные накопители.

Для создания хранилищ данных требуется разработка логической модели, которая будет полностью отражать ожидания клиента и возможности разработчика. После этого можно рассматривать технологические аспекты – например, размеры хранилища. Логическая модель может содержать тысячи атрибутов и связей.

Стоимость СХД варьируется в зависимости от масштаба, логической модели и оборудования. В одних случаях речь идет о сотнях тысяч рублей, в других – о десятках миллионов. На создание СХД может уйти от одного месяца до полугода. Важным фактором, который следует учитывать, является необходимость сервисной поддержки оборудования. Ее можно заказать непосредственно в представительстве мирового производителя или у компании Специальные Технологии РМ. Во втором случае стоимость владения СХД заметно снизится.