Algoritmi e Basi di dati

Di
Gianluca
Mon-
dini

Indice

1 Algoritmi e strutture dati

1 Algoritmo ricorsivo

Una metodologia per programmare le funzioni ricorsive è la seguente:

1. Individuare i casi base in cui la funzione è definita immediatamente;
2. Effettuare le chiamate ricorsive su un insieme più piccolo di dati, cioè un insieme più vicino ai casi base;
3. Fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada nei casi base;

2 Complessità di un algoritmo

2.1 Definizione

È una funzione (sempre positiva) che associa alla dimensione del problema il costo della sua risoluzione in base alla misura scelta

$T_P(n)$ = Complessità con costo = tempo del programma P al variare di n

2.2 Ordine

$g(n)$ è di ordine $O(f(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che per ogni $n > n_0$ si ha $g(n) \leqslant c\, f(n)$

2.3 Regole

2.3.1 Regola dei fattori costanti

Per ogni costante positiva k, $O(f(n)) = O(k\, f(n))$

2.3.2 Regola della somma

Se $f(n)$ è $O(g(n))$, allora $f(n) + g(n)$ è $O(g(n))$

2.3.3 Regola del prodotto

Se $f(n)$ è $O(f1(n))$ e $g(n)$ è $O(g1(n))$, allora $f(n)g(n)$ è $O(f1(n)\, g1(n))$

2.3.4 Altre regole

Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$
Per ogni costante k, k è $O(1)$
Per $m \leqslant p$, n^m è $O(n^p)$
Un polinomio di grado m è $O(n^m)$

2.4 Teorema

Per ogni k, $n^k \in O(a^n)$, per ogni $a > 1$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

3 Complessità di un algoritmo ricorsivo

3.1 Esempio: fattoriale di un numero

La funzione è definita come

```
int fact (int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

Consideriamo come base il valore 0:

$$T(0) \in C[[n == 0]] + C[[\text{return } 1]] = O(1) + O(1) = O(1)$$

Per $T(n)$, se $n > 0$, abbiamo un tempo $O(1)$ complessivo per il test, la chiamata ricorsiva e la moltiplicazione più il tempo per l'esecuzione applicata a $n-1$. Quindi

$$T(n) = O(1) + T(n-1)$$

A questo punto rimpiazziamo gli $O(1)$ di $T(1)$ e $T(n)$ con simboli generici di costante, diversi fra loro perchè corrispondono a pezzi di programma diversi, e abbiamo la seguente relazione di ricorrenza:

$$T(0) = a$$

$$T(n) = b + T(n-1) \quad \text{per } n > 0$$

Proviamo a calcolare i valori di $T(n)$:

$$T(1) = b + T(0) = b + a$$

$$T(2) = b + T(1) = 2\,b + a$$

In generale, per $i \geqslant 0$, avremo $T(i) = i\,b + a$. Dimostriamo questo risultato con l'induzione naturale

- Base. $i = 0$. Abbiamo $T(0) = 0\,b + a = a$
- Induzione.
  Ipotesi. $T(i) = i\,b + a$
  Tesi. $T(i+1) = (i+1)\,b + a$
  Dim.

$$T(i+1) = b + T(i) \text{ per definizione di T}$$

$$= b + i\,b + a \quad \text{per ipotesi induttiva}$$

$$= (i+1)\,b + a$$

Quindi, poiché $T(n) = n\,b + a$, abbiamo $T(n) \in O(n)$

3.2 Dalla relazione di ricorrenza alla complessità

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(1) = a$
- $T(n) = b\,n + T(n-1)$

$$T(n) \in O(n^2)$$

- $T(1) = a$
- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(0) = a$
- $T(n) = b + T(n/2)$

$$T(n) \in O(\log(n))$$

- $T(0) = a$
- $T(n) = b + 2\,T(n/2)$

$$T(n) \in O(n)$$

- $T(1) = a$
- $T(n) = b + 2\,T(n-1)$

$$T(n) \in O(2^n)$$

- $T(0) = d$
- $T(n) = b\,n^k + T(n-1)$

$$T(n) \in O(n^{k+1})$$

Esercizio: fibonacci() ricorsivo

- $T(0) = T(1) = d$
- $T(n) = b + T(n-1) + T(n-2)$

$$T(n) \in O(2^n)$$

Esempio: fibonacci() non ricorsivo

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

Esempio: mergeSort()

- $T(0) = T(1) = d$

- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

Esempio: split()

- $T(0) = T(1) = d$
- $T(n) = b + T(n-2)$

$$T(n) \in O(n)$$

Esempio: merge()

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

4 Algoritmi di ordinamento

4.1 Merge sort

(necessita di revisione, probabilmente è sbagliato)

```
void mergeSort(int * arr, int start, int end) {
  int mid;
  if (start < end) {
    mid = (start + end) / 2;
    mergeSort(arr, start, mid);
    mergeSort(arr, mid + 1, end);
    combina(arr, start, mid+1, end);
  }
}

void combina(int arr[], int start, int mid, int end) {
  int iSx = start, iDx = mid;
  std::vector<int> tempResult;
  while (1) {
    if (arr[iSx] < arr[iDx]) {
      tempResult.push_back(arr[iSx++]);
    }
    else {
      tempResult.pushBack(arr[iDx++]);
    }
  }
}
```

4.2 Insertion sort

[necessita di revisione]

```
void insertSort(int x[], int n) {
  int i;
  int j;
  int app;
  for (i = 1; i < n; i++) {
    app = x[j];
    j = i - 1;
    while (j >= 0 && x[j] > app) {
      x[j+1] = x[j];
      j--;
    }
    x[j+1] = app;
  }
  return;
}
```

5 Alberi binari

- NULL è un albero binario
- Un nodo p più due alberi binari Bs e Bd forma un albero binario

5.1 Numero di foglie e di nodi

Un albero binario bilanciato con livello k ha

- $2^{k+1} - 1$ nodi

- 2^k foglie

5.2 Alcuni algoritmi

5.2.1 Contare il numero dei nodi

```
int nodes(Node* tree) {
  if (!tree) return 0;
  return 1 + nodes(tree -> left) + nodes(tree -> right);
}
```

5.2.2 Contare il numero delle foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  if (!tree -> left && !tree -> right) return 1;
  return leaves(tree -> left) + leaves(tree -> right);
}
```

5.2.3 Cercare un etichetta e resistuire un puntatore

Se il nodo non compare nell'albero, viene restituito NULL. Se l'albero contiene più di un'etichetta, viene restituito un puntatore al primo

```
Node* findNode(Infotype n, Node* tree) {
  // L'albero è vuoto, l'etichetta non può esserci
  if (!tree) return NULL;
  // Trovata l'etichetta, restituisco il puntatore
  if (tree -> label == n) return tree;
  // Cerco a sinistra
  Node* a = findNode(n, tree -> left);
  // Controllo se il puntatore della ricerca "a sinistra"
  // a resistuito qualcosa di interessante, altrimenti
cerco a destra
  if (a) return a;
  else return findNode(n, tree -> right);
}
```

5.2.4 Eliminare tutto l'abero

Alla fine il puntatore deve essere NULL

```
void delTree(Node* &tree) {
  if (tree) {
    delTree(tree -> left);
    delTree(tree -> right);
    delete tree;
    tree = NULL;
  }
}
```

5.2.5 Inserire un nodo

Inserisce un nodo (son) come figlio di father, sinistro se $c = 'l'$ oppure destro se $c = 'r'$.

Ritorna 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice

```
int insertNode(Node* &tree, InfoType son, InfoType father,
char c) {
  // Caso in cui l'albero sia vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  // Caso normale
  // Effettuo la ricerca di father con la funzione
  // di ricerca nodo (vedi sopra)
  Node * a = findNode(father, tree);
  // Se il nodo non è stato trovato, restituisco 0 e mi
fermo
```

```
  if (!a) return 0;
  // Inserimento come figlio sinistro
  if (c == 'l' && !a -> left) {
    a -> left = new Node;
    a -> left -> label = son;
    a -> left -> left = a -> left -> right = NULL;
    return 1;
  }
  if (c == 'r' && !a -> right) {
    a -> right = new Node;
    a -> right -> label = son;
    a -> right -> left = a -> right -> right = NULL;
    return 1;
  }
}
```

5.2.6 Compito 2 - Esercizio 4

Scrivere una funzione che, dato un albero binario ad etichette di tipo string, con puntatore alla radice t, restituisca come risultato il numero di nodi che hanno numero dispari di discendenti. Si calcoli la complessità della soluzione proposta in funzione del numero di nodi dell'albero.

```
int dispari(Node* t, int & nodi) {
  if (!t) {
    nodi = 0;
    return 0;
  }
  int cs, cd, nodis, nodid;
  cs = dispari(t -> left, nodis);
  cd = dispari(t -> left, nodid);
  nodi = nodis + nodid + 1;
  return cs + cd + ((nodis + nodid) % 2);
}
```

La funzione è $O(n)$

5.2.7 Compito 1 - Esercizio 4

Scrivere una funzione ricorsiva che, dato un albero binario a etichette intere, conta il numero di nodi che hanno più foglie maggiori o uguali a zero che minori di zero tra i propri discendenti.

```
int conta(const Node * t, int & pos, int & neg) {
  if (!t) {
    pos = 0;
    neg = 0;
    return 0;
  }
  if (!t -> left && !t -> right) {
    pos = (t -> info >= 0) ? 1 : 0;
    neg = (t -> info < 0)  ? 1 : 0;
    return 0;
  }
  int pos_left, pos_right;
  int neg_left, neg_right;
  int conta_left = conta(t -> left, pos_left, neg_left);
  int conta_right = conta(t -> right, pos_right,
neg_right);
  pos = pos_left + pos_right;
  neg = neg_left + neg_right;
  return (pos > neg) ? 1 : 0 + conta_left + conta_right;
}
```

6 Alberi generici

- Un nodo p è un albero
- Un nodo più una sequenza di alberi $A_1 ... A_n$ è un albero

6.1 Alcuni algoritmi

6.1.1 Contare il numero di nodi

Vedi l'algoritmo per gli alberi binari

6.1.2 Contare il numero di foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  // Caso della foglia
  if (!tree -> left) return 1 + leaves(tree -> right);
  // "Non caso" della foglia
  return leaves(tree -> left) + leaves(tree -> right);
```

6.1.3 Inserire un nodo in fondo ad una lista di fratelli

```
void addSon(InfoType x, Node* &tree) {
  // Caso in cui la lista sia vuota
  if (!tree) {
    tree = new Node;
    tree -> label = x;
    tree -> left = tree -> right = NULL;
  }
  else {
    addSon(x, tree -> right);
  }
}
```

6.1.4 Inserire un nodo son come ultimo figlio di father

Se l'albero è vuoto, lo inserisce come radice

```
int insert(InfoType son, InfoType father, Node* &tree) {
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  Node* a = findNode(father, tree);
  if (!a) return 0;
  addSon(son, a -> left);
  return 1;
}
```

6.1.5 Compito 1 - Esercizio 5

Scrivere una funzione che, dato un albero generico a etichette intere e memorizzazione figlio-fratello, conta il numero di nodi che hanno più figli maggiori o uguali a zero che minori di zero.

```
int conta(const Node* t) {
  if (!t) return 0;
  const Node * n;
  int pos = 0, neg = 0;
  for (n = t -> left; n != NULL; n = n -> right) {
    if (n -> info >= 0) pos++;
    else neg++;
  return (pos > neg) ? 1 : 0 + conta(t -> left) + conta(t
-> right);
}
```

7 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario tale che per ogni nodo p

- I nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p
- I nodi del sottoalbero destro di p hanno etichetta maggiore dell'etichetta p

7.1 Proprietà

- Non ci sono doppioni
- La visita simmetrica elenca le etichette in ordine crescente

7.2 Alcuni algoritmi

7.2.1 Cercare un nodo

```
Node* findNode(InfoType n, Node* tree) {
```

```
  if (!tree) return 0;
  if (n == tree -> label) return tree;
  if (n < tree -> label) {
    return findNode(n, tree -> left);
  }
  return findNode(n, tree -> right);
}
```

7.2.2 Inserire un nodo

```
void insertNode(InfoType n, Node* &tree) {
  // Albero vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = n;
    tree -> left = tree -> right = NULL;
    return;
  }
  // Caso n < radice
  if (n < tree -> label) {
    insertNode(n, tree -> left);
  }
  if (n > tree -> label) {
    insertNode(n, tree -> right);
  }
}
```

L'algoritmo ha complessità $O(\log(n))$

7.2.3 Restituire l'etichetta del nodo più piccolo di un albero ed eliminare il nodo che la contiene

```
void deleteMin(Node *& tree, InfoType &m) {
  if (tree -> left) // C'è un nodo più piccolo
    deleteMin(tree -> left, m);
  else {
    m = tree -> label; // restituisco l'etichetta
    Node * a = tree;
    // connetto il sottoalbero destro di
    // m al padre di m
    tree = tree -> right;
    // elimino il nodo
    delete a;
  }
}
```

7.2.4 Cancellare un nodo ?

```
void deleteNode(InfoType n, Node *& tree) {
  if (tree) {
    // n è minore della radice
    if (n < tree -> label) {
      deleteNode(n, tree -> left);
      return;
    }
    // n è maggiore della radice
    if (n > tree -> label) {
      deleteNode(n, tree -> right);
      return;
    }
    // n non ha figlio sinistro
    if (!tree -> left) {
      Node * a = tree;
      tree = tree -> right;
      delete a;
      return;
    }
    // n non ha figlio destro
    if (!tree -> right) {
      Node * a = tree;
      tree = tree -> left;
      delete a;
      return;
```

```
    }
    // n ha entrambi i figli
    deleteMin(tree -> right, tree -> label);
}
```

Questo algoritmo ha complessità $O(\log(n))$

8 Heap

Un heap è un albero binario quasi bilanciato con le seguenti proprietà:

- i nodi dell'ultimo livello sono addossati a sinistra
- in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti

L'heap viene memorizzato in un array

8.1 Calcolare le parentele

- Figlio sinistro di $i = 2\,i+1$
- Figlio destro di $i = 2\,i+2$
- Padre di $i = \frac{i-1}{2}$

8.2 Classe Heap

```
class Heap {
  private:
    int * h;
    int last;
    void up(int);
    void down(int);
    void exchange(int i, int j);
  public:
    Heap(int);
    ~Heap();
    void insert(int);
    int extract();
};
```

8.2.1 Costruttore

```
Heap::Heap(int n) {
  h = new int[n];
  last = -1;
}
```

8.2.2 Distruttore

```
Heap::~Heap() {
  delete h [n];
}
```

8.2.3 Inserimento

- Memorizza l'elemento nella prima posizione libera dell'array
- Fa risalire l'elemento tramite scambi figlio-padre per mantenere la proprietà dello heap

```
void Heap::insert(int x) {
  h[++last] = x;
  up(last);
}

// i è l'indice dell'elemento da far risalire
void Heap::up(int i) {
  // Se non sono sulla radice
  if (i > 0) {
    // Se l'elemento è maggiore del padre
    if (h[i] > h[(i-1)/2]) {
      // Scambia il figlio con il padre
      exchange(i, (i-1)/2);
      // chiama up sulla nuova posizione
      up((i-1)/2);
    }
```

```
    }
  }
```

La funzione termina in due casi:

- viene chiamata con l'indice 0 (radice)
- L'elemento è inferiore al padre

La complessità è $O(\log(n))$ perché ogni chiamata risale di un livello

8.2.4 Estrazione

- Restituisce il primo elemento dell'array
- Mette l'ultimo elemento al posto della radice e decrementa last
- Fa scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

```
int Heap::extract() {
  int r = h[0];
  h[0] = h[last--];
  down(0);
  return r;
}

// i è l'indice dell'elemento da far scendere
void Heap::down(int i) {
  // son = indice del figlio sinistro (se esiste)
  int son = 2*i+1;
  // se i ha un solo figlio (è l'ultimo dell'array)
  if (son == last) {
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio, altrimenti termina
      exchange(i, last);
    }
  }
  // se i ha entrambi i figli
  else if (son < last) {
    // son = indice del figlio maggiore tra i due
    if (h[son] < h[son+1]) son++;
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio
      exchange(i, son);
      // chiama down sulla nuova posizione
      down(son);
      // altrimenti termina (termina anche se i non ha
figli)
    }
  }
}
```

L'algoritmo ha complessità $O(\log(n))$

9 Ricerca hash

9.1 Alcuni algoritmi

9.1.1 Ricerca tramite hash

```
bool hashSearch (int* A, int k, int x) {
  int i = h(x);
  if (A[i] == 1) return true;
  else return false;
}
```

9.2 Metodo hash ad accesso non diretto

È possibile rilasciare l'iniettività e permettere che due elementi diversi abbiano lo stesso indirizzo hash. Si ha una collisione quando

$$h(x_1) = h(x_2)$$

Bisogno gestire le seguenti situazioni:

- Come cercare un elemento se il suo posto è occupato da un altro

- Come inserire gli elementi

9.2.1 Soluzione: hash modulare

Si scrive una funzione $h()$

$$h(x) = (x \ \% \ k)$$

In modo tale da essere sicuri di generare tutti e soli gli indici dell'array

Legge di scansione lineare Se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota.

L'inserimento è fatto con lo stesso criterio

Agglomerato Gruppo di elementi con indirizzi hash diversi (?)

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

Funzione di ricerca con scansione lineare

```
bool hashSearch(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j<k; j++) {
    int pos = (i+j) % k;
    if (A[pos] == -1) return false;
    if (A[pos] == x) return true;
  }
  return false;
}
```

Funzione di insierimento in presenza di cancellazioni

```
int hashInsert(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j < k; j++) {
    int pos = (i+j) % k;
    // -1: posizione vuota
    // -2: posizione disponibile
    if ((A[pos] == -1) || (A[pos] == -2)) {
      A[pos] = x;
      return 1;
    }
  }
  return 0;
}
```

10 Grafi

$$\text{Grafo orientato}=(N, A)$$

dove

- $N =$ insieme di nodi
- $A \subseteq N \times N =$ insieme di archi (coppie ordinate di nodi)

Un grafo orientato con n nodi ha al massimo n^2 archi

10.1 Rappresentazioni in memoria di grafi

10.1.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  Node * next;
};

Node* graph[N];
```

10.1.2 Tramite matrici di adiacenza

```
int graph[N][N];
```

10.2 Rappresentazioni in memoria di grafi con nodi ed archi etichettati

10.2.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  ArcType archLabel;
```

```
  Node * next;
};

Node * graph[N];

NodeType nodeLabels[N];
```

NodeType = char
ArcType = int

10.2.2 Tramite matrici di adiacenza

```
ArcType graph[N][N];
NodeType nodeLabels[N];
```

10.3 Cammino più breve

10.3.1 Algoritmo di Dijkstra

- Si applica ai grafi orientati
- Trova i cammini minimi **da un nodo a tutti gli altri**
- È basato sulla metodologia **greedy**

Vengono utilizzate due tabelle:

- **dist** (distanza)
- **pred** (predecessore)

con n elementi.

Vengono eseguiti n passi, e ad ogni passo:

1. Si sceglie il nodo con distanza minore in **dist**
2. Si aggiornano **pred** e **dist** per i suoi immediati successori

11 Templates

11.1 Esempi di definizione

```
template <class T1>
T1 somma(T1 a, T2 b) {
  return a + b;
}
```

11.2 Parametri costanti

11.3 Variabili statiche

Ogni "tipo di funzione" possiede la propria variabile statica indipendente

```
template<class Data>
void typeCounter(Data a) {
        static int c;
        c++;
        cout << "Il counter vale " << c << endl;
}
```

12 Derivazione

12.1 Specificatori d'accesso

- I campi **privati** di una classe non sono accessibili dalle sottoclassi nè dall'esterno
- I campi **protetti** di una classe sono accessibilit dalle sottoclassi, ma non dall'esterno
- I campi **pubblici** di una classe sono accessibili anche dall'esterno
- I campi privati, protetti e pubblici rimangono tali in tutta la gerarchia

12.2 Costruzione degli oggetti

Quando un oggetto viene costruito si costruisce prima la parte **BASE** ed in seguito quella **DERIVATA**.

Viene quindi prima chiamato il costruttore della classe base e poi quello della classe derivata.

Se la classe base ha dei costruttori, il costruttore di una classe derivata deve chiamarne uno nella lsita di inizializzazione. Può non chiamarlo esplicitamente se la classe base ha un costruttore di default, che in questo caso viene chiamato automaticamente.

12.3 Funzioni virtuali

2 Basi di dati

1 Normalizzazione

In informatica la **normalizzazione** è un procedimento volto all'eliminazione della ridondanza informativa e del rischio di incoerenza dal database. Esistono vari livelli di normalizzazione (*forme normali*) che certificano la qualità dello schema del database.

Questo processo si fonda su un semplice criterio: se una relazione presenta più concetti tra loro indipendenti, la si decompone in relazioni più piccole, una per ogni concetto. Questo tipo di processo non è sempre applicabile in tutte le tabelle, dato che in taluni casi potrebbe comportare una perdita d'informazioni.

1.1 Prima forma normale

Definizione: Si dice che una base dati è in 1NF (*prima forma normale*) se vale la seguente relazione per ogni relazione contenuta nella base dati: una relazione è in 1NF se e solo se:

1. tutte le righe della relazione hanno lo stesso numero di attributi
2. non presenta gruppi di attributi che si ripetono (ossia ciascun attributo è definito su un dominio con valori atomici)
3. tutti i valori di un attributo sono dello stesso tipo (appartengono allo stesso dominio)
4. esiste una chiave primaria (ossia esiste un insieme di attributi, che identifica in modo univoco ogni tupla della relazione)
5. l'ordine delle righe è irrilevante (non è portatore di informazioni)

1.2 Seconda forma normale

Definizione: Una base dati è invece in 2NF (*seconda forma normale*) quando è in 1NF e per ogni relazione tutti gli attributi non chiave dipendono funzionalmente dall'intera chiave composta (ovvero la relazione non ha attributi che dipendono funzionalmente da una parte della chiave).

Come esempio supponiamo di avere una tabella con gli esami sostenuti dagli studenti universitari. I campi di interesse potrebbero quindi essere i seguenti:

- "Codice corso di laurea"
- "Codice esame"
- "Matricola studente"
- "Voto conseguito"
- "Data superamento"

La tabella avrà quindi la seguente intestazione

id_corso_laurea	**id_esame**	**id_studente**	voto	data

La chiave candidata (tale terminologia è riservata alle superchiavi minimali, anche dette semplicemente chiavi) è rappresentata dalla tripla evidenziata, ossia da:

- "Codice corso di laurea"
- "Codice esame"
- "Matricola studente"

Essa infatti risulta essere l'insieme di chiavi minimale per poter identificare in modo univoco le tuple (i record) della tabella.

I campi "Voto conseguito" e "Data superamento", invece, sono campi *non chiave*, e fanno riferimento all'intera superchiave.

Difatti, se dipendessero solo da:

- "Codice corso di laurea" si perderebbero le informazioni relative allo studente e all'esame superato
- "Codice esame" si perderebbero le informazioni relative allo studente ed al corso di laurea a cui l'esame appartiene

- "Matricola studente'' si perderebbero le informazioni relative all'esame superato e al corso di laurea a cui lo studente è iscritto.
- "Codice corso di laurea", "Codice esame" si perderebbero le informazioni relative allo studente che ha superato l'esame
- "Codice corso di laurea", "Matricola studente" si perderebbero le informazioni relative all'esame superato
- "Matricola studente", "Codice esame" si perderebbero le informazioni relative al Corso di Laurea di appartenenza.

1.3 Terza forma normale

Definizione: Una base dati è in 3NF (*terza forma normale*) se è in 2NF e tutti gli attributi non-chiave dipendono dalla chiave soltanto, ossia non esistono attributi che dipendono da altri attributi non-chiave. Tale normalizzazione elimina la dipendenza transitiva degli attributi dalla chiave.

Per ogni dipendenza funzionale non banale $X \to Y$ è vera una delle seguenti condizioni:

- X è una superchiave della relazione
- Y è membro della chiave della relazione

Teorema: Ogni relazione può essere portata in 3NF.

Dalla soluzione dell'esame: Una relazione r è in 3NF se o la parte sinistra di ogni dipendenza è una superchiave di r, oppure ogni attributo nella parte destra di ogni dipendenza è contenuto in una chiave di r.

Da domanda d'esame: Lo studente fornisca la definizione di Terza Forma Normale

Una tabella r è in Terza Forma Normale se per ogni dipendenza funzionale non banale $X \to Y$ definita su di essa è verificata almeno una delle seguenti condizioni:

- X è superchiave di r
- Ogni attributo in Y è contenuto in almeno una chiave di r

Da domanda d'esame: Dare un esempio di relazione non BCNF ma 3NF

```
R(_A_, _B_, C, D)
AB -> CD
D -> A
```

1.4 Forma normale di Boyce e Codd

Definizione: Una relazione R è in forma normale di Boyce e Codd (BCNF) se e solo se è in 3NF e, per ogni dipendenza funzionale non banale $X \to Y$, X è una superchiave per R.

Dato un insieme di relazioni, non è possibile garantire sempre il raggiungimento della BCNF; in particolare il mancato raggiungimento di questo obiettivo è indice che la base dati è affetta da un'anomalia di cancellazione (ossia è possibile perdere dati a seguito di un'operazione di cancellazione).

Es: Facciamo un esempio molto banale, se abbiamo uno schema relazionale $R < ABCDE, ABC \to DE >$

Mettiamolo in forma canonica $ABC \to D, ABC \to E$.

Calcoliamo le chiavi: A, B e C non stanno a destra di nessuna dipendenza, quindi appartengono a tutte le chiavi

La chiusura di ABC è ABCDE quindi ABC è una chiave

Ora, visto che una chiave è una superchiave minimale (ovvero una superchiave con tutti attributi essenziali per derivare ogni attributo del sistema) lo schema relazionale è in BCNF

2 Definizioni

2.1 Miste

2.1.1 Chiusura transitiva di un insieme di dipendenze

Dalla soluzione dell'esame:

La chiusura di un insieme di dipendenze è $F^+ = \{X \to Y \mid F \Rightarrow X \to Y\}$

2.1.2 Chiusura di un insieme di attributi

Dalla soluzione dell'esame:

La chiusura di un insieme di attributi X^+ è costituito dagli attributi funzionalmente dipendenti da X

Da domanda esame:

Dare un esempio di uso della chiusura di un insieme di attributi
Individuazione della chiave primaria

2.1.3 Algoritmo per il calcolo della chiusura di un insieme di attributi X

Da domanda esame:

Definire l'algoritmo per il calcolo della chiusura di un insieme di attributi X

Si parte da un insieme uguale ad X, per ogni dipendenza $A \rightarrow B$, dove A appartiene ad X, B viene aggiunto ad X, finché non ci sono altre dipendenze tali che B non è contenuto nell'insieme costruito fino a questo punto.

2.2 ACID

- **atomicità**: la transazione è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla, non sono ammesse esecuzioni parziali;
- **consistenza**: quando inizia una transazione il database si trova in uno stato consistente e quando la transazione termina il database deve essere in un altro stato consistente, ovvero non deve violare eventuali vincoli di integrità, quindi non devono verificarsi contraddizioni (*inconsistenza*) tra i dati archiviati nel DB;
- **isolamento**: ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione;
- **durabilità**: detta anche **persistenza**, si riferisce al fatto che una volta che una transazione abbia richiesto un *commit work*, i cambiamenti apportati non dovranno essere più persi. Per evitare che nel lasso di tempo fra il momento in cui la base di dati si impegna a scrivere le modifiche e quello in cui li scrive effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul DB.

2.3 Database NOSQL

I dati sono conservati in documenti, non in tabelle.

I database NOSQL non hanno uno schema fisso come quelli relazionali.

2.3.1 Vantaggi e svantaggi

1. **Leggerezza computazionale**: i database NoSQL non prevedono operazioni di aggregazione sui dati, in quanto tutte le informazioni sono già raccolte in un unico documento associato all'oggetto da trattare. Negli ambienti SQL la complessità di queste operazioni, e quindi il peso computazionale, cresce con l'ingigantirsi della base di dati, del numero di tabelle e delle informazioni da trattare. Il NoSQL, invece, non ha limiti di dimensioni in questo senso. Così si ottengono migliori prestazioni e performance anche in ambienti di Big Data. Lo scotto da pagare a tutta questa flessibilità e alla proprietà di aggregazione dei database NoSQL è la **duplicazione delle informazioni**. In realtà, i costi sempre meno proibitivi dei sistemi di storage rendono questo svantaggio poco importante.
2. **Assenza di schema**: i database NoSQL sono privi di schema in quanto il documento JSON contiene tutti i campi necessari, senza necessità di definizione. In questo modo, possiamo arricchire le nostre applicazioni di nuovi dati e informazioni, definibili liberamente all'interno dei documenti JSON **senza rischi per l'integrità dei dati**. I database non relazionali, a differenza di quelli SQL, si rivelano quindi adatti a inglobare velocemente nuovi tipi di dati e a conservare dati semistrutturati o non strutturati.
3. **Scalabilità orizzontale garantita**: l'aggregazione dei dati e l'assenza di uno schema definito a priori offre l'opportunità di scalare orizzontalmente i database NoSQL senza difficoltà e senza rischi operativi.

Vantaggi di un sistemi NoSQL da compito d'esame

Quali sono i vantaggi di un DBMS NoSQL e per quali applicazioni sono rilevanti?

Scalabilità orizzontale, sharding, high availability.

Le applicazioni interessate a queste caratteristiche sono quelle che richiedono tempi di risposta molto bassi senza necessitare di controlli di consistenza molto forti.

2.3.2 Differenze con un database SQL

Da soluzione esame:

Nei DBMS non relazionali è possibile attuare sharding senza perdere di efficienza nell'esecuzione di query complesse, nei relazionali no.

Nei DBMS non relazionali si ha replicazione estesa di nodi anche a caldo e quindi scaling orizzontale invece che verticale.

Nei DBMS non relazionali non è possibile attuare controlli sull'integrità dei dati e quindi garantirne la consistenza, e non esiste uno standard condiviso per la memorizzazione e l'accesso ai dati.

Da domanda esame:

Quali sono le caratteristiche di un database SQL che non vengono mantenute in uno NoSQL?

Standardizzazione del modello e del query language

Portabilità

Controllo sull'integrità dei dati

Niente tabelle nella realizzazione fisica

2.3.3 Equivalente ACID per database NOSQL

Da una domanda di un compito:

Perché i sistemi NoSQL non sono basati sulle proprietà ACID. Su quale paradigma sono basati?

I sistemi NoSQL sono basati sul paradigma Base, ovvero Basically Available, Soft state, Eventual Consistency, invece che godere delle proprietà di Atomicità, Coerenza, Isolamento e Durabilità. La coerenza dei dati in un sistema sempre disponibile e altamente distribuito non è garantibile.

3 Schedule

3.1 VSR

Uno schedule viene detto *view-serializzabile (VSR)* se è view-equivalente ad uno seriale.

Per stabilire se due schedule sono view-equivalenti bisogna verificare che abbiano le stesse relazioni legge-da e le stesse scritture finali.

3.2 CSR

Uno schedule *conflict-serializzabile* è anche *view-serializzabile*.

Per capire se uno schedule è conflict-serializzabile è necessario che il grafo orientato i cui nodi rappresentazon le transazioni non contenga "cicli".

3.3 Locking a due fasi

Le due strategie viste finora non garantiscono la serializzabilità, ma la verificano a posteriori. Nei sistemi utilizzati in pratica si preferisce invece fare in modo che essa sia in qualche modo assicurata, adottando particolari accorgimenti e spesso strutture dati d'appoggio. Una delle tecniche più utilizzate si basa sul *locking*, che fa uso di una variabile di *lock* per descrivere lo stato di una risorsa rispetto alle operazioni che lo riguardano. Quando una transazione vuole utilizzare una risorsa deve *prima* richiederne il lock, attendere finché non gli viene concesso e rilasciarlo dopo il suo utilizzo. In particolare, nel *locking a due fasi* (2PL) una volta che una transazione ha rilasciato un lock qualsiasi, non può più acquisirne altri. Quindi, se una transazione t_i ha bisogno del lock su una risorsa già occupata da t_j, quest'ultima prima di rilasciarla dovrà acquisire il lock di tutte le risorse a cui deve ancora accedere, perché una volta unlockata una risorsa non potrà più lockarne nessuna.

Perché ci interessano tanto gli schedule generati con questa tecnica? *Perché sono serializzabili*.

Esistono due tipi di 2PL, a seconda della variabile di lock che viene utilizzata:

- a *2 stati*: il lock può essere *locked* (risorsa bloccata) o *unlocked* (risorsa rilasciata o comunque disponibile)
- a *3 stati*: il lock può essere *read locked*, *write locked* o *unlocked*. L'introduzione del read locked consente a due o più transazioni di condividere la stessa risorsa in lettura.

Ultima considerazione: gli schedule 2PL sono un sottoinsieme della classe di schedule CSR, che sono a loro volta sottoinsieme dei VSR. Ne consegue che se uno schedule è non view-serializzabile non è nemmeno conflict-serializzabile né tantomeno basato su 2PL!

3.3.1 Come determinare se uno schedule è generato da uno scheduer basato su 2PL (esempio)

Prendiamo lo schedule

$$R_1(x), R_1(t), R_2(z), W_3(x), W_1(x), R_1(y), W_3(t), W_2(x), W_1(y)$$

1. È molto utile raggruppare le operazioni per transazioni per facilitare le azioni di lock e unlock.
   - $t_1 = r_1(x), r_1(t), w_1(x), r_1(y), w_1(y)$
   - $t_2 = r_2(z), w_2(x)$
   - $t_3 = w_3(x), w_3(t)$
2. Ora dobbiamo stabilire operazione per operazione, mantenendo l'ordine dello schedule dato, a quale transazione dare il lock su quale risorsa. Conviene mantere queste informazioni in una tabella con una colonna per le operazioni, una per il lock/unlock e infine tante colonne quante sono le risorse, in cui indicare quale transazione ne detiene attualmente l'accesso. L'istruzione da scrivere nella colonna dei lock avrà la seguente forma:

   $$\text{lock}\,(\text{transazione}, \text{risorsa}) \Longrightarrow \text{OK}\,/\,\text{NO}$$

   oppure il complementare

   $$\text{unlock}(\text{transazione}, \text{risorsa}) \Longrightarrow \text{OK}\,/\,\text{NO}$$

3. Riempiamo la tabella con il nostro schedule

Operazione	**Lock**	**X**	**Y**	**Z**	**T**
$R_1(x)$	lock(t_1, x): OK	t_1			
$R_2(t)$	lock(t_1, t):OK	t_1			t_1
$R_2(z)$	lock(t_2, z): OK	t_1		t_2	t_1
$W_3(x)$	lock(t_1, y): OK	t_1	t_1	t_2	t_1
	unlock(t_1, x): NO	t_1	t_1	t_2	t_1

4. Osserviamo i passaggi effettuari a partire dall'operazione $W_3(x)$
   - La transazione 3 vuole effettuare una scrittura su x, quindi deve prima togliere il lock alla transazione che ne ha possesso, ovvero la t_1
   - Prima di unlockare una risorsa a t_1 devo prima bloccare tutte quelle risorse di cui in futuro avrà bisogno, quindi la y
   - A questo punto unlocko x dalla transazione 1, e qui sta il problema: l'operazione successiva dello schedule è $W_1(x)$, ma avendo appena unlockato non potrò più rilockoralo.

Quindi lo schedule in esame non è 2PL. Notare che in realtà noi già sapevamo l'esito della verifica: dato che lo schedule non era CSR, non poteva in nessun modo essere nemmeno 2PL.