

**Basi di dati II**  
**Esame — 25 settembre 2012 — Compito A**

**Cenni sulle soluzioni**  
**(senza domande ripetute, le varianti del testo sono in rosso)**

Rispondere su questo fascicolo. Tempo a disposizione: due ore e trenta minuti.

Cognome \_\_\_\_\_ Nome \_\_\_\_\_ Matricola \_\_\_\_\_ Ordin. \_\_\_\_\_

**Domanda 1 (15%)**

Si consideri una relazione  $R(\underline{A} \ B \ C \ D \ E)$ , in cui gli attributi hanno tutti la stessa dimensione  $a = 4$  Byte, molto più piccola della dimensione del blocco pari a  $P = 4$  KByte. Si supponga che la relazione contenga  $N = 2.000.000$  ennuple e che le operazioni più frequenti su di essa siano le seguenti:

$o_1$  `SELECT * FROM R ORDER BY A`, con frequenza  $f_1 = 10$  operazioni nell'unità di tempo

$o_2$  `SELECT A, B, C FROM R ORDER BY A`, con frequenza  $f_2 = 200$  operazioni nell'unità di tempo

Valutare le due seguenti alternative di memorizzazione, calcolando il costo complessivo (riportare la formula che indica il numero di accessi nell'unità di tempo e il relativo valore numerico):

(i) memorizzazione della relazione  $R(\underline{A} \ B \ C \ D \ E)$  ordinata su  $A$

costo unitario di  $o_1$ :  $c_1 = N/(P/(5a)) = 5aN/P = 10^4$

costo unitario di  $o_2$ :  $c_2 = N/(P/(5a)) = 5aN/P = 10^4$

costo complessivo:  $c_1f_1 + c_2f_2 = N/(P/(5a)) = 55f_1aN/P = 2,1 \times 10^6$

(ii) memorizzazione delle proiezioni  $R1(\underline{A} \ B \ C)$  e  $R2(\underline{A} \ D \ E)$ , entrambe ordinate su  $A$  (supporre che il join possa essere eseguito con il metodo merge-join e che quindi il costo del join stesso sia trascurabile rispetto a quello delle due scansioni).

costo unitario di  $o_1$ :  $c_1 = 2N/(P/(3a)) = 6aN/P = 1,2 \times 10^4$

costo unitario di  $o_2$ :  $c_2 = N/(P/(3a)) = 3aN/P = 6 \times 10^3$

costo complessivo:  $c_1f_1 + c_2f_2 = 36f_1aN/P = 1,32 \times 10^6$

**Domanda 2 (10%)**

Considerare il documento XML qui sotto e definire uno schema XSD per il quale esso sia valido.

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
  <student>
    <firstName> Paolo </firstName>
    <lastName> Neri </lastName>
    <id> 281283 </id>
    <courses>
      <course>
        <name> Programmazione Orientata agli Oggetti </name>
        <shortName> POO </shortName>
        <record>
          <grade> 28 </grade>
          <date> 13/06/11 </date>
        </record>
      </course>
      <course>
        <name> Analisi e progettazione del software </name>
        <shortName> APS </shortName>
      </course>
      ...
    </courses>
  </student>
  <student>
    ...
  </student>
</students>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="students">
    <xsd:complexType>
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="student"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="student">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
        <xsd:element name="id" type="xsd:string"/>
        <xsd:element ref="courses"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:unique name="studentID">
      <xsd:selector xpath="."/>
      <xsd:field xpath="id"/>
    </xsd:unique>
  </xsd:element>
</xsd:schema>
```

```
<xsd:element name="record">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="grade" type="xsd:string"/>
      <xsd:element name="date" type="xsd:date"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="course">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="shortName" type="xsd:string"/>
      <xsd:element ref="record" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="courses">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="course"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

**Domanda 3** (15%)

Con riferimento a documenti come quello mostrato nella Domanda 2 (supponendolo memorizzato nel file `esame.xml`), rispondere alle seguenti interrogazioni

1. In XPath, trovare gli studenti che hanno superato POO con voto superiore a 24

```
//student[.//course[shortName="POO" and .//grade>24]]
```

2. In XPath, trovare gli studenti che hanno una media superiore a 27

```
//student[avg(.//grade)>27]
```

3. In XQuery, trovare gli studenti che hanno superato due esami con uno stesso voto

```
xquery version "1.0";
for $s in fn:doc("studenti.xml")//student
let $e1 := $s//course
let $e2 := $s//course

where $s2 >> $s1 and $e1//grade = $e2//grade

return $s
```

**Domanda 4 (20%)**

Una agenzia di viaggio on-line vende ai clienti pacchetti di viaggio proposti dai tour operator e desidera un sistema di supporto alle proprie attività che permetta di effettuare analisi sui pacchetti venduti in passato, al fine di comprendere sia le scelte dei clienti sulla base delle loro caratteristiche sia i profitti che derivano dalle vendite dei vari pacchetti. In particolare, si vuole realizzare un data mart con uno schema dimensionale che permetta di conoscere il numero di viaggi venduti, con il prezzo di vendita e il profitto per l'agenzia, rispetto a varie dimensioni, quali il tour operator, il cliente, con le sue caratteristiche fondamentali (fascia di età, professione, città e regione di partenza; si noti che alcune possono cambiare nel tempo), la destinazione (regione, località e sue caratteristiche), il tipo di sistemazione (categoria di albergo, appartamento, etc.).

Mostrare un possibile schema dimensionale, commentando brevemente (nello spazio disponibile) le eventuali scelte significative e chiarendo bene quale sia la grana dei fatti e il significato di ciascuna dimensione.

Sicuramente ci si aspettava una risposta dettagliata, con schemi delle tabelle e qualche commento. Qui sotto alcune riflessioni schematiche.

Una grana ragionevole prevede di considerare i singoli viaggi (con data, tour operator, tipo di sistemazione e destinazione e cliente come dimensioni).

In effetti, il cliente non è detto che debba essere una dimensione esplicita (dipende molto dalla eventuale necessità di ricordare il cliente in quanto tale). Le caratteristiche del cliente (fascia di età, professione e residenza potrebbero essere dimensioni a sé stanti. Se si usa la dimensione cliente, allora deve essere una “slowly changing dimension,” relativamente a fascia di età, professione e residenza.

Misure opportune sono il numero di viaggi (che se si usa la dimensione cliente potrebbe essere uguale a 1, se i viaggi sono individuali), il prezzo di vendita e il guadagno.

**Domanda 5** (10%)

Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

client 1	client 2
read(x)	read(x)
x = x + 10 write(x)	x = x + 20 write(x)
commit	commit

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su **multiversioni** e livelli di isolamento **SERIALIZABLE** e **READ COMMITTED**. Assumiamo che (come avviene di solito) **multiversioni** preveda

- **SERIALIZABLE**: le letture fanno riferimento allo stato della base di dati all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- **READ COMMITTED**: le letture fanno riferimento allo stato della base di dati all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED				SERIALIZABLE																																			
<table><tr><th colspan="2">client 1</th><th colspan="2">client 2</th></tr><tr><td>read(x)</td><td>legge 200</td><td>read(x)</td><td>legge 200</td></tr><tr><td>x = x + 10 write(x)</td><td>x vale 210 scrive 210</td><td>x = x + 20 xlock(x)</td><td>x vale 220 bloccata</td></tr><tr><td>commit</td><td></td><td>write(x) commit</td><td>scrive 220</td></tr></table> <p>anomalia: perdita di aggiornamento</p>				client 1		client 2		read(x)	legge 200	read(x)	legge 200	x = x + 10 write(x)	x vale 210 scrive 210	x = x + 20 xlock(x)	x vale 220 bloccata	commit		write(x) commit	scrive 220	<table><tr><th colspan="2">client 1</th><th colspan="2">client 2</th></tr><tr><td>read(x)</td><td>legge 200</td><td>read(x)</td><td>legge 200</td></tr><tr><td>x = x + 10 write(x)</td><td>x vale 210 scrive 210</td><td>x = x + 20 xlock(x)</td><td>x vale 220 bloccata</td></tr><tr><td>commit</td><td></td><td>abort read(x) x = x + 20 write(x)</td><td>legge 210 x vale 230 scrive 230</td></tr></table> <p>non c'è anomalia; l'abort è dovuto al fatto che x è stato modificato dopo l'avvio della transazione che sta scrivendo</p>				client 1		client 2		read(x)	legge 200	read(x)	legge 200	x = x + 10 write(x)	x vale 210 scrive 210	x = x + 20 xlock(x)	x vale 220 bloccata	commit		abort read(x) x = x + 20 write(x)	legge 210 x vale 230 scrive 230
client 1		client 2																																					
read(x)	legge 200	read(x)	legge 200																																				
x = x + 10 write(x)	x vale 210 scrive 210	x = x + 20 xlock(x)	x vale 220 bloccata																																				
commit		write(x) commit	scrive 220																																				
client 1		client 2																																					
read(x)	legge 200	read(x)	legge 200																																				
x = x + 10 write(x)	x vale 210 scrive 210	x = x + 20 xlock(x)	x vale 220 bloccata																																				
commit		abort read(x) x = x + 20 write(x)	legge 210 x vale 230 scrive 230																																				

**Domanda 6** (10%)

Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

client 1	client 2
read(x) x = x + 10 write(x)	
	read(x) x = x + 20 write(x)
commit	commit

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su **2PL** e livelli di isolamento **SERIALIZABLE** e **READ COMMITTED**. Assumiamo che (come avviene di solito) **2PL** preveda

- **SERIALIZABLE**: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- **READ COMMITTED**: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED				SERIALIZABLE			
client 1		client 2		client 1		client 2	
read(x) x = x + 10 write(x)	legge 200 x vale 210 scrive 210			read(x) x = x + 10 write(x)	legge 200 x vale 210 scrive 210		
		slock(x)	bloccata			slock(x)	bloccata
commit		read(x) x = x + 20 write(x) commit	legge 210 x vale 230 scrive 230	commit		read(x) x = x + 20 write(x) commit	legge 210 x vale 230 scrive 230
non c'è anomalia				non c'è anomalia			

**Domanda 7** (20%)

Considerare le relazioni sotto schematizzate, su cui si deve effettuare un hash join (sulla base del campo numerico). Si supponga che il fattore di blocco sia pari a 2 per entrambe le relazioni (e quindi che esse, come mostrato dalle divisioni fra le celle, occupino rispettivamente 7 e 10 blocchi) e che i buffer disponibili siano 4. Come noto, l'hash join (in questo caso, come in molti altri) si può eseguire in due passate, suddividendo prima entrambe le relazioni (per mezzo di una stessa funzione hash) in un numero di porzioni (dette *bucket*) il cui quadrato sia superiore al numero di blocchi della più piccola delle due relazioni, (in questo caso quindi 3) e poi confrontando i record nei bucket omologhi.

Mostrare (1) i bucket che si otterrebbero per le due relazioni (con una funzione hash che calcola il resto della divisione per 3, che nell'esempio corrisponde sempre al valore dell'ultima cifra); (2) il contenuto dei buffer all'inizio della seconda fase; (3) il contenuto dei buffer dopo l'esecuzione di sette chiamate al metodo `next()` sullo scan che implementa l'hash join; (4) i record prodotti dalle prime sette chiamate di `next()`.

R1

A	901
B	931
C	330
D	660
E	362
F	362
G	900
H	390
I	390
L	900
M	362
N	391
O	392
P	601

Bucket per R1

C	330
D	660
G	900
L	900
H	390
I	390
A	901
B	931
N	391
P	601
E	362
F	362
M	362
O	392

Buffer all'inizio  
della seconda fase

C	330
D	660
G	900
L	900
H	390
I	390
330	...
900	...

Record prodotti  
dalle prime sette  
chiamate di `next()`

C	330	...
G	900	...
L	900	...
D	660	...
H	390	...
I	390	...
A	901	...

Bucket per R2

R2

901	...
931	...
932	...
902	...
330	...
900	...
660	...
301	...
302	...
630	...
631	...
360	...
361	...
362	...
390	...
391	...
392	...
600	...
601	...
602	...

330	...
900	...
660	...
630	...
360	...
390	...
600	...
901	...
931	...
301	...
631	...
361	...
391	...
601	...
392	...
362	...
302	...
932	...
902	...
602	...

Buffer dopo sette  
chiamate di `next()`

A	901
B	931
N	391
P	601
901	...
931	...



**Basi di dati II**  
**Esame — 25 settembre 2012 — Compito B**

**Cenni sulle soluzioni**  
**(senza domande ripetute, le varianti del testo sono in rosso)**

Rispondere su questo fascicolo. Tempo a disposizione: due ore e trenta minuti.

Cognome \_\_\_\_\_ Nome \_\_\_\_\_ Matricola \_\_\_\_\_ Ordin. \_\_\_\_\_

**Domanda 1 (15%)**

Si consideri una relazione  $R(\underline{A} \ B \ C \ D \ E)$ , in cui gli attributi hanno tutti la stessa dimensione  $L = 4$  Byte, molto più piccola della dimensione del blocco pari a  $P = 4$  KByte. Si supponga che la relazione contenga  $R = 1.000.000$  ennuple e che le operazioni più frequenti su di essa siano le seguenti:

$o_1$  `SELECT * FROM R ORDER BY A`, con frequenza  $f_1 = 10$  operazioni nell'unità di tempo

$o_2$  `SELECT A, B, C FROM R ORDER BY A`, con frequenza  $f_2 = 200$  operazioni nell'unità di tempo

Valutare le due seguenti alternative di memorizzazione, calcolando il costo complessivo (riportare la formula che indica il numero di accessi nell'unità di tempo e il relativo valore numerico):

(i) memorizzazione della relazione  $R(\underline{A} \ B \ C \ D \ E)$  ordinata su  $A$

costo unitario di  $o_1$ :  $c_1 = R/(P/(5L)) = 5LR/P = 5 \times 10^3$

costo unitario di  $o_2$ :  $c_2 = R/(P/(5L)) = 5LR/P = 5 \times 10^3$

costo complessivo:  $c_1 f_1 + c_2 f_2 = R/(P/(5L)) = 55 f_1 a R/P = 1,05 \times 10^6$

(ii) memorizzazione delle proiezioni  $R1(\underline{A} \ B \ C)$  e  $R2(\underline{A} \ D \ E)$ , entrambe ordinate su  $A$  (supporre che il join possa essere eseguito con il metodo merge-join e che quindi il costo del join stesso sia trascurabile rispetto a quello delle due scansioni).

costo unitario di  $o_1$ :  $c_1 = 2R/(P/(3L)) = 6LR/P = 6 \times 10^3$

costo unitario di  $o_2$ :  $c_2 = R/(P/(3L)) = 3LR/P = 3 \times 10^3$

costo complessivo:  $c_1 f_1 + c_2 f_2 = 36 f_1 a R/P = 6,6 \times 10^5$

**Domanda 5** (10%)

Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

client 1	client 2
read(x)	read(x)
x = x + 10 write(x)	x = x + 20 write(x)
commit	commit

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su **2PL** e livelli di isolamento **SERIALIZABLE** e **READ COMMITTED**. Assumiamo che (come avviene di solito) **2PL** preveda

- **SERIALIZABLE**: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- **READ COMMITTED**: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

SERIALIZABLE				READ COMMITTED			
client 1		client 2					
read(x)	legge 200	read(x)	legge 200				
x = x + 10 xlock(x)	x vale 210 bloccata	x = x + 20 xlock(x)	x vale 220 bloccata				
abort slock(x)	bloccata	write(x) commit	scrive 220				
read(x)	legge 220						
x = x + 10 write(x) commit	x vale 230 scrive 230						
non c'è anomalia							

  

client 1		client 2	
read(x)	legge 200	read(x)	legge 200
x = x + 10 write(x)	x vale 210 scrive 210	x = x + 20 xlock(x)	x vale 220 bloccata
commit		write(x) commit	scrive 220

anomalia: perdita di aggiornamento

**Domanda 6** (10%)

Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

client 1	client 2
read(x) x = x + 10 write(x)	
	read(x) x = x + 20 write(x)
commit	commit

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su **multiversioni** e livelli di isolamento **SERIALIZABLE** e **READ COMMITTED**. Assumiamo che (come avviene di solito) **multiversioni** preveda

- **SERIALIZABLE**: le letture fanno riferimento allo stato della base di dati (con scritture in commit) all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- **READ COMMITTED**: le letture fanno riferimento allo stato della base di dati (con scritture in commit) all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED				SERIALIZABLE			
client 1		client 2		client 1		client 2	
read(x) x = x + 10 write(x)	legge 200 x vale 210 scrive 210			read(x) x = x + 10 write(x)	legge 200 x vale 210 scrive 210		
		read(x) x = x + 20 xlock(x)	legge 200 x vale 220 bloccata			read(x) x = x + 20 xlock(x)	legge 200 x vale 230 bloccata
commit		write(x) commit	scrive 220	commit		abort	
anomalia: perdita di aggiornamento				non c'è anomalia; l'abort è dovuto al fatto che x è stato modificato dopo l'avvio della transazione che sta scrivendo			

**Domanda 7** (20%)

Considerare le relazioni sotto schematizzate, su cui si deve effettuare un hash join (sulla base del campo numerico). Si supponga che il fattore di blocco sia pari a 2 per entrambe le relazioni (e quindi che esse, come mostrato dalle divisioni fra le celle, occupino rispettivamente 7 e 10 blocchi) e che i buffer disponibili siano 4. Come noto, l'hash join (in questo caso, come in molti altri) si può eseguire in due passate, suddividendo prima entrambe le relazioni (per mezzo di una stessa funzione hash) in un numero di porzioni (dette *bucket*) il cui quadrato sia superiore al numero di blocchi della più piccola delle due relazioni, (in questo caso quindi 3) e poi confrontando i record nei bucket omologhi.

Mostrare (1) i bucket che si otterrebbero per le due relazioni (con una funzione hash che calcola il resto della divisione per 3, che nell'esempio corrisponde sempre al valore dell'ultima cifra); (2) il contenuto dei buffer all'inizio della seconda fase; (3) il contenuto dei buffer dopo l'esecuzione di sette chiamate al metodo `next()` sullo scan che implementa l'hash join; (4) i record prodotti dalle prime sette chiamate di `next()`.

R1

A	931
B	901
C	300
D	660
E	362
F	362
G	930
H	390
I	390
L	930
M	362
N	391
O	392
P	601

Bucket per R1

C	300
D	660
G	930
L	930
H	390
I	390

A	931
B	901
N	391
P	601

E	362
F	362
M	362
O	392

Buffer all'inizio  
della seconda fase

C	300
D	660

G	930
L	930

H	390
I	390

300	...
930	...

Record prodotti  
dalle prime sette  
chiamate di `next()`

C	300	...
G	930	...
L	930	...
D	660	...
H	390	...
I	390	...
A	931	...

R2

931	...
901	...
902	...
932	...
300	...
930	...
660	...
301	...
302	...
630	...
631	...
360	...
361	...
362	...
390	...
391	...
392	...
600	...
601	...
602	...

Bucket per R2

300	...
930	...
660	...
630	...
360	...
390	...
600	...

931	...
901	...
301	...
631	...
361	...
391	...
601	...

392	...
362	...
302	...
902	...
932	...
602	...

Buffer dopo sette  
chiamate di `next()`

A	931
B	901

N	391
P	601

--	--

931	...
901	...