Student ID (*Matricola*)					
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	6	6	6	6
7	7	7	7	7	7
8	8	8	8	8	8
9	9	9	9	9	9

Computing Infrastructures

Course 095897

P. Cremonesi, M. Gribaudo

28-07-2015

Last Name / Cognome:

....

First Name / Nome:

....

Answers must be given exclusively on the answer sheet (last sheet): DO NOT FILL ANY BOX IN THIS SHEET

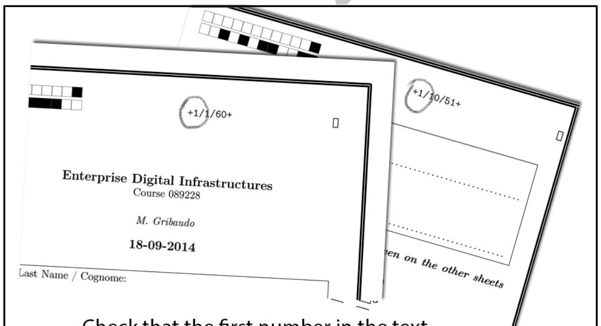
Students must use pen (black or blue) to mark answers (no pencil).
Students are permitted to use a non-programmable calculator.

Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.

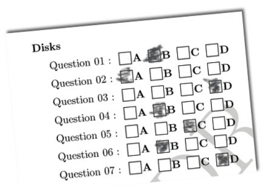
Students are NOT permitted to use mobile phones and similar connected devices.

Scores: correct answers +1 point, unanswered questions 0 points, wrong answers -0.333 points.

Reserve questions **must NOT** be answered, unless explicitly stated during the exam.



Check that the first number in the text and in the answer sheet is the same

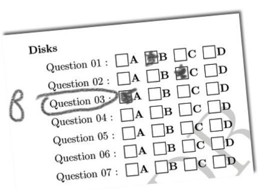


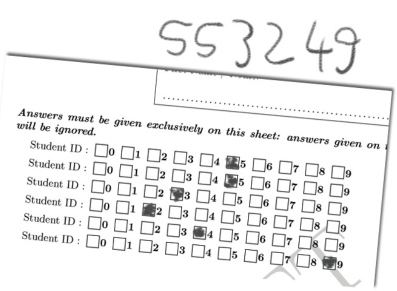
Do not use crosses to mark the answers!

Completely fill the box!

If you make a mistake:

1) circle the word "Question"
2) write the correct answer to its side





Enter the numbers of your "Student ID" from left to right

Disks

Consider the following set of disks connected in RAID 6, with generator $g = 2$:

Block A1	Block A2	Block A3	Block A4	Block A5	Block A6
D0	D1 2	D2	D3	P 11	Q 19
Block B1	**Block B2**	**Block B3**	**Block B4**	**Block B5**	**Block B6**
D0 3	D1 1	D2	P	Q 21	D3 1
Block C1	**Block C2**	**Block C3**	**Block C4**	**Block C5**	**Block C6**
D0 5	D1 3	P	Q	D2 0	D3 1

Figure 1: A RAID 6 configuration.

Question 1 What data is contained in blocks $A3$ and $A4$?

[A] $D_2 = 1,\ D_3 = 2$ [B] Cannot be computed [C] $D_2 = 2,\ D_3 = 0$ [D] $D_2 = 0,\ D_3 = 0$

SOLUTION:
Cannot be computed because there are three missing blocks

Question 2 What data is contained in blocks $B3$ and $B4$?

[A] $D_2 = 2,\ P = 7$ [B] $D_2 = 0,\ P = 5$ [C] Cannot be computed [D] $D_2 = 1,\ P = 7$

SOLUTION:
$D_2 = 2,\ P = 7.$

Question 3 What data is contained in blocks $C3$ and $C4$?

[A] Cannot be computed [B] $P = 9,\ Q = 19$ [C] $P = 7,\ Q = 15$ [D] $P = 10,\ Q = 21$

SOLUTION:
$P = 9,\ Q = 19.$

Question 4 Data in block $A2$ is changed to $D_1 = 5$. Which are the new values of the parity blocks?

[A] Cannot be computed [B] $P = 8,\ Q = 21$ [C] $P = 9,\ Q = 27$ [D] $P = 14,\ Q = 25$

SOLUTION:
$P = P_{old} - D_{old} + D_{new} = 11 - 2 + 5 = 14,$
$Q = Q_{old} - 2D_{old} + 2D_{new} = 19 - 4 + 10 = 25$

Let us now consider that the disks have the capacity of 1 TB, and that their $MTTF = 1250$ days and $MTTR = 25$ days.

Question 5 Which is the mean time to data loss ($MTTDL$) of the RAID?

[A] $MTTDL = 26041$ days [C] $MTTDL = 2083$ days
[B] $MTTDL = 52083$ days [D] $MTTDL = 5208$ days

SOLUTION:

$MTTDL$52083 days.

Question 6 Which will be the ($MTTDL$) if the same number of disks would be organized as a $RAID5 + 0$, with two groups of three disks each?

- [A] $MTTDL = 5208$ days
- [B] $MTTDL = 52083$ days
- [C] $MTTDL = 26041$ days
- [D] $MTTDL = 2083$ days

SOLUTION:
$MTTDL$5208 days.

Question 7 **Reserve -** *Do not answer unless explicitly stated during the exam*
The total capacity of the system is in both $RAID6$ or $RAID5 + 0$ configurations:

- [A] 3TB
- [B] 4TB
- [C] 6TB
- [D] 5TB

SOLUTION:
4TB.

Performance evaluation

A storage system is composed by a storage controller whose demand is $D_{CTRL} = 15$ ms, and 50 identical disks, whose service time is $S_{Disk} = 600$ ms. Requests are equally distributed among the disks, so that each one gets the same fraction of requests. The system can be described with an open model.

Question 8 The *demand* of each *Disk* is:

- [A] $D_{Disk} = 600\text{ms}$
- [B] $D_{Disk} = 30\text{s}$
- [C] $D_{Disk} = 30\text{ms}$
- [D] $D_{Disk} = 12\text{ms}$

SOLUTION:
$v_{Disk} = 1/50$ (since the request are equally distributed).
$D_{Disk} = S_{Disk} \cdot v_{Disk} = \frac{600}{50} = 12\text{ms}$.

Question 9 The *maximum arrival rate* that the system can handle is:

- [A] $\lambda_{max} = 83.33$ req/s.
- [B] $\lambda_{max} = 1.667$ req/s.
- [C] $\lambda_{max} = 66.67$ req/s.
- [D] $\lambda_{max} = 1.626$ req/s.

SOLUTION:
$\lambda_{max} = \frac{1}{D_{max}} = \frac{1}{0.015} = 66.67$ req/s.

Question 10 The *minimum response time* that the system can offer is:

- [A] $R_{min} = 615\text{ms}$
- [B] $R_{min} = 30.015\text{s}$
- [C] $R_{min} = 12\text{ms}$
- [D] $R_{min} = 27\text{ms}$

SOLUTION:
$R_{min} = \sum D_k = 15 + 12 \cdot 50 = 615$ ms.

When the utilization of the controller is $U_{Ctrl} = 90\%$, the response time of the system is 1.2 s. In this scenario:

Question 11 The *average number of requests in the system* is:

- [A] $N = 60$
- [B] $N = 72$
- [C] $N = 40$
- [D] Cannot be computed since Z is missing.

SOLUTION:
$U_{Ctrl} = X \cdot D_{Ctrl}$. $X = U_{Ctrl}/D_{Ctrl} = 0.9/0.015 = 60$ req/s
$N = X \cdot R = 60 \cdot 1.2 = 72$

Question 12 The *utilization* of each disk U_{disk} is:

- [A] $U_{Disk} = 72\%$
- [B] The disks saturate
- [C] $U_{Disk} = 90\%$
- [D] $U_{Disk} = 15\%$

SOLUTION:
$U_{Disk} = X \cdot D_{Disk} = 60 \cdot 0.012 = 72\%$.

Dependability

In the following questions we will assume that both failure and repair events follow exponential distributions.

Question 13 The analysis of the failure behavior of a two components system reveals that the system is down only when both its components are down. The two components A and B have the following characteristics: $\lambda_A = 0.005\ days^{-1}$, $MTTR_A = 2\ days$, $\lambda_B = 0.002\ days^{-1}$ and $MTTR_B = 10\ days$. The reliability of the system at $t = 100\ days$ is equal to:

- [A] 0.9286
- [B] 0.4965
- [C] 0.95134
- [D] 0.5834

SOLUTION:
$R_A(100) = e^{-0.005*100} = 0.6065$ $R_B(100) = e^{-0.002*100} = 0.8187$ $R_{sys} = 1 - (1 - 0.6065)(1 - 0.8187) = 0.92865$

Question 14 In the previous case, how many additional replicas of the component B are required to achieve a reliability at $t \to +\infty$ greater than 0.9 ?

- [A] 0
- [B] 42
- [C] 2
- [D] It is not possible

SOLUTION:
Not possible. For $t \to +\infty$ we have $R(t) \to 0$

Question 15 The MTTF computed without repair of the system described in Question 13 is:

- [A] 557.143
- [B] 142.857
- [C] 350
- [D] 99

SOLUTION:
$MTTF_{Par} = MTTF_A + MTTF_B - 1/(1/MTFF_A + 1/MTTF_B) = 557.143$

Consider now the components A and B organized as in Figure 1 and assume they have the characteristics specified in Question 13.

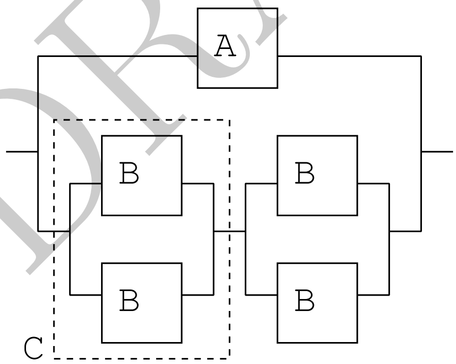


Figure 2: RBD.

Question 16 The MTTF computed without repair of block C is:

- [A] 400
- [B] 75
- [C] 750
- [D] 250

SOLUTION:
$MTTF = MTTF_B(1 + 1/2) = 500 * (3/2) = 750$

Question 17 The availability of the whole system is equal to:

- [A] 1
- [B] 0.999992
- [C] 0.56625
- [D] 0.92

SOLUTION:
$A_A = 200/(200 + 2) = 0.990099$ $A_B = 500/510 = 0.980392$
$A_{sys} = 1 - (1 - A_A)(1 - (1 - (1 - A_B)^2)^2) = 0.999992$

Question 18 **Reserve** - *Do not answer unless explicitly stated during the exam*
Consider a generic component D with $MTTF_D = 100$. Compute the minimum integer value of t such that the failure probability of the component is greater than 0.6.

A 77 B 92 C 3 D 10

SOLUTION:
$1 - e^{-t/100} \geq 0.6 \;\; 0.4 \geq e^{-t/100} \;\; ln 0.4 \geq -t/100 \;\; t \geq -100 ln(0.4) \;\; t \geq 92$

Cloud Computing

Question 19 Which of the following is a fundamental property of cloud computing?

- [A] aggregation of resources
- [B] sharing of resources
- [C] pay-per-usage
- [D] encapsulation

Solution. encapsulation, sharing and aggregation are properties of server virtualization; cloud computing in the form of saas does not necessary require virtualization

Question 20 Which of the following is an advantage of IaaS with respect to SaaS?

- [A] Easier to manage
- [B] Pay-per-usage
- [C] Accessible any-moment any-where
- [D] More flexible

Solution. Both have a pay-per-usage model and can be made accessible any where any moment

ITIL and DevOps

Question 21 Which is the definition of *problem* in ITIL?

- [A] The malfunctioning of a service which is violating the service level agreements
- [B] The unknown cause of recurring incidents
- [C] An incident difficult to solve
- [D] Whenever the user is not able to use a service

Question 22 Which is the main goal of the Change Management process?

- [A] Approving beneficial changes while ensuring minimal disruption to IT services
- [B] Planning for recovery of services in case of failure of changes
- [C] Planning for the availability and capacity of IT services
- [D] Producing requests-for-change (RFC) to fix problems so that they will not happen again in the future

Question 23 What is Infrastructure-as-a-Code?

- [A] Changes to an environment must be approved by the change management process.
- [B] Use of either Docker or Vagrant as provisioning tools
- [C] It is a form of cloud-computing based on virtualization
- [D] Changes to an environment are always applied via scripts which are versioned as traditional software

Question 24 **Reserve** - *Do not answer unless explicitly stated during the exam*
Which of the following is one of the goals of the Change Management process in ITIL

- A Upgrade applications
- B Buy new hardware for the data-center
- C Keep track of the configuration of the data-center
- D Evaluate costs and benefits of fixing bugs

Virtualization

Question 25 Which is the corresponding **alternative** technology to *type 1* hypervisors?

- A Hosted hypervisor
- B Hardware-assisted hypervisor
- C Micro-kernel hypervisor
- D Monolitic hypervisor

Question 26 Which is the definition of *bare-metal* hypervisor?

- A The hypervisor is running directly on the hardware
- B The system is not virtualized
- C The hypervisor is using ring -1 of the processor to provide hardware-assisted virtualization
- D The hypervisor is using binary code translation

Question 27 How many CPU levels are required to efficiently run a traditional operating system – such as Linux or Windows – within a virtual machine without using software emulation?

- A Five
- B Four
- C Two
- D Three

Question 28 Which is the corresponding **alternative** technology to *binary code translation*?

- A Hardware-assisted
- B Micro-kernel
- C Hosted
- D Para-virtualization

Question 29 Which is the main advantage of full virtualization with respect to OS-level virtualization?

- A Full virtualization provides better sharing of resources
- B Full virtualization provides light-weight virtual machines
- C Full virtualization does not allow to run different host operating systems
- D Full virtualization provides better insulation between virtual machines

Big Data

The following Apache Spark code computes the TF-IDF (Term Frequency - Inverse Document Frequency) schema for product descriptions contained into the dataset *products.csv*. Before being processed, stopwords contained into the file *stopwords.txt* are removed from each product description. TF and IDF scores are then computed (in this exercise we consider product descriptions as documents). Read carefully the code below and answer the questions. Pay attention, the code may contain some errors!

IMPORTANT NOTES:

Lines starting with three sharps ### (e.g., lines 1, 6 and 25) contain comments that explain what a function or a fragment of code is supposed to compute.

Lines starting with a single sharp # (e.g., lines 28, 39 and 45) report the output of the previous print call.

Use comments to understand what the content of an RDD or the outcome of an instruction should be.

The correct answer to <FILL IN> questions is the only one that generates an RDD having the "exact" structure to ensure the correctness of the whole program.

```
1  ### imports and custom type definitions
2  from collections import namedtuple
3
4  Product = namedtuple("Product", ["id", "name", "description", "manufacturer", "price"])
5
6  ### Helper functions to parse a line of the products file
7  def removeQuotes(s):
8      return ''.join(c for c in s if c != '"')
9
10 def splitLine(line):
11     return removeQuotes(line).split(',')
12
13 def parseLine(line):
14     line_split = splitLine(line)
15     if line_split[0] == "id":
16         return (line_split, 0)
17     else:
18         return (Product(
19                 id=line_split[0],
20                 name=line_split[1],
21                 description=line_split[2],
22                 manufacturer=line_split[3],
23                 price=line_split[4]), 1)
24
25 ### Parse the products file
26 productsRawRdd = sc.textFile("data/products.csv")
27 print productsRawRdd.take(3)
28 # [u'"id","name","description","manufacturer","price"',
29 # u'"g11448761432933644608","spanish vocabulary builder","expand your vocabulary! contains fun lessons
30 # that both teach",,6.95',
31 # u'"g8175198959985911471","topics presents: museums of world","5 cd-rom set. step behind the velvet rope
32 # to examine some",,12.9']
33
34 productsRdd = (productsRawRdd
35             .map(parseLine)
36             .filter(lambda x: x[1] != 0)
37             .map(lambda x: x[0]))
38 print productsRdd.take(3)
39 # [Product(id=u'g11448761432933644608', name=u'spanish vocabulary builder', description=u"expand your
40 # vocabulary! contains fun lessons that both teach', manufacturer=u'', price=u'6.95'),
41 # Product(id=u'g8175198959985911471', name=u'topics presents: museums of world', description=u'5 cd-rom set.
42 # step behind the velvet rope to examine some', manufacturer=u'', price=u'12.9')]
```

```

print "The total number of products is %d" % productsRdd.count()
# The total number of products is 200

### Import stopwords
stopfile = sc.textFile("data/stopwords.txt")
stopwords = set(stopfile.collect())
print stopwords
# set([u’all’, u’just’, u’being’, u’over’, u’both’, u’through’, u’have’, u’further’, u’their’, u’if’, ...
# u’again’, u’no’, u’when’, u’same’, u’any’, u’how’, u’other’, u’which’, u’you’, u’yours’, u’once’])

# Tokenize and filter descriptions
def tokenizeAndFilterStopwords(s):
    return [c for c in s.lower().split(’ ’) if c != ’’ and c not in stopwords]

productsToTokens = productsRdd.map(lambda x: (x.id,tokenizeAndFilterStopwords(x.description))
print productsToTokens.take(2)
# [(u’g11448761432933644608’, [u’expand’, u’vocabulary!’, u’contains’, u’fun’, u’lessons’, u’teach’]),
# (u’g8175198959985911471’, [u’5’, u’cd-rom’, u’set.’, u’step’, u’behind’, u’velvet’, u’rope’, u’examine’])]

biggestRecord = productsToTokens.<FILL IN>
print "The biggest record is %s with %d tokens" % (biggestRecord[0][0], len(biggestRecord[0][1]))
# The biggest record is g18431066094306345892 with 43 tokens

### Compute the Inverse Document Frequency of each token t
### i.e., the number of products N divided by the number of product descriptions n(t) that contain t
def idfs(corpus):
    N = corpus.count()
    uniqueTokens = corpus.map(lambda x: x[1])
    tokenCountPairTuple = uniqueTokens.map(lambda x: (x, 1))
    tokenSumPairTuple = tokenCountPairTuple.reduceByKey(lambda x, y: x + y)
    return (tokenSumPairTuple.map(lambda x: (x[0], N / float(x[1]))))
productsIdfs = idfs(productsToTokens)
print productsIdfs.take(5)
# [(u’lessons’, 40.0), (u’fun’, 25.0), (u’contains’, 66.7), (u’vocabulary’, 200.0),
# (u’teach’, 200.0)]

print "There are %d unique tokens" % productsIdfs.count()
# There are 2289 unique tokens
print ’The rarest token among descriptions is "%s"’ % productsIdfs.<FILL IN>
# The rarest token among descriptions is "re-pedalling"

### Compute Term Frequencies
### i.e. the frequency of each token within each product description
def tf(product):
    counts = {}
    tot_count = 0
    for t in product[1]:
        t_count = counts.get(t, 0) # if t is in counts return its value, otherwise 0
        t_count += 1
        counts[t] = t_count
        tot_count += 1
    return [(product[0], (token, float(count) / tot_count)) for token, count in counts.items()]
print tf(productsToTokens.first())
# [(u’g11448761432933644608’, (u’lessons’, 0.0625)), (u’g11448761432933644608’, (u’fun’, 0.0625)),
# (u’g11448761432933644608’, (u’contains’, 0.0625)), (u’g11448761432933644608’, (u’vocabulary!’, 0.0625)),
#(u’g11448761432933644608’, (u’teach’, 0.0625)), (u’g11448761432933644608’, (u’expand’, 0.0625))]

### Compute the TF-IDF schema for each product
### NOTE: The TF-IDF score for a term is simply the product if its TF and IDF scores
productsFs = (productsToTokens
```

```
104             .flatMap(tf)
105             .map(lambda x: (x[1][0], (x[0], x[1][1]))))
106 productsJoin = productsFs.join(productsIdfs)
107 productsTfIdf = (productsJoin
108                 .map(<FILL IN>)
109                 .reduceByKey(lambda x, y: x + y))
110 productTfIdfSchema = productsTfIdf.filter(lambda x: x[0] == u’g1144876143293364608’)
111 print "The TF-IDF schema for product %s is" % productTfIdfSchema.first()[0]
112 print productTfIdfSchema.first()[1]
113 # The TF-IDF schema for product g1144876143293364608 is
114 # [(u’lessons’, 2.5), (u’fun’, 1.5625), (u’vocabulary!’, 12.5), (u’teach’, 12.5),
115 # (u’expand’, 4.166666666666667), (u’contains’, 4.166666666666667)]
```

Question 30 Is line 58 correct?

- A No, it should have been *productsToTokens = productsRdd.map(lambda x: x in tokenizeAndFilterStopwords(x))*
- B No, it should have been *productsToTokens = productsRdd.map(lambda x: tokenizeAndFilterStopwords(x.description))*
- C Yes, it is correct.
- D No, it should have been *productsToTokens = productsRdd.map(tokenizeAndFilterStopwords))*

SOLUTION:
Yes, it is correct.

Question 31 Complete line 63.

- A biggestRecord = productsToTokens.sortByKey(lambda x: len(x[1])).first()
- B biggestRecord = productsToTokens.takeOrdered(1, lambda x: -len(x[1]))
- C biggestRecord = (productsToTokens
  .map(lambda x: (len(x[1]), x[0]))
  .sortByKey(False)
  .take(1))
- D biggestRecord = productsToTokens.map(lambda x: len(x[1])).first()

SOLUTION:
biggestRecord = productsToTokens.takeOrdered(1, lambda x: -len(x[1]))

Question 32 Is line 71 correct?

- A No, it should have been *uniqueTokens = corpus.map(lambda x: x[1]).distinct()*
- B Yes. It is correct.
- C No, it should have been *uniqueTokens = corpus.flatMap(lambda x: set(x[1]))*
- D No, it should have been *uniqueTokens = corpus.map(lambda x: set(x[1]))*

SOLUTION:
No, it should have been *uniqueTokens = corpus.flatMap(lambda x: set(x[1]))*

Question 33 Complete line 108.

- A lambda x: (x[1][0][1], [x[0], x[1][0][0] * x[1][1]])
- B lambda x: (x[1][0][0], [x[0], x[1][0][1] * x[1][1]])
- C lambda x: (x[1][0][0], [(x[0], x[1][0][1] * x[1][1])])
- D lambda x: (x[1][0][1], [(x[0], x[1][0][0] * x[1][1])])

SOLUTION:
lambda x: (x[1][0][0], [(x[0], x[1][0][1] * x[1][1])])

Question 34 Which line is variable *productsFs* (line 103) actually computed at for the first time (either partially or totally)?

[A] 107 – 109 [B] 106 [C] 103 – 105 [D] 111

SOLUTION:
productsFs gets computed at line 111 when the first action (first) is applied to the RDD, due to the lazy evaluation mechanism of Apache Spark.

Question 35 Which of these is a possible output for *productsJoin.first()* (line 106)? (Hint: look at previous *print* instructions to understand which is the content of the joined RDDs.)

[A] (u'lessons', 40.0, (u'g114487614329336446O8', 0.0625))
[B] (u'lessons', (40.0, (u'g114487614329336446O8', 0.0625)))
[C] (u'lessons', ((u'g114487614329336446O8', 0.0625), 40.0))
[D] (u'lessons', (u'g114487614329336446O8', 0.0625), 40.0)

SOLUTION:
(u'lessons', ((u'g114487614329336446O8', 0.0625), 40.0))

Question 36 **Reserve -** *Do not answer unless explicitly stated during the exam*
Complete line 82.

[A] *takeOrdered(1, lambda x: -x[1])[0][0]*
[B] *takeOrdered(1, lambda x: x[1])[0][0]*
[C] *takeOrdered(1)[0][0]*
[D] *takeOrdered(1, lambda x: -x[1])[0][1]*

SOLUTION:
takeOrdered(1, lambda x: -x[1])[0][0]

Disclaimer: this document is intended as a support to the exam. It should not replace the study. No guarantee is given on the correctness of the formulas contained: we assume no responsibility for errors that might occur in the exam due to mistakes in this document. However we did our best to ensure the correctness of the material here included.

Do not take notes on this document.

Table I. PERFORMANCE: VARIABLES

Variable	Definition
T	length of an observation interval
B	Busy time
C	Number of completions
A	Number of arrivals
W	Jobs per service time
N	number of users
U	utilization
Z	average think time of a user
X	system throughput
λ	arrival rate
S	service time
R	system response time
X_k, U_k S_k, R_k	measure for resource k

Table II. PERFORMANCE: RELATIONS

Relations
$\lambda = \frac{A}{T}$
$X = \frac{C}{T}$
$U = \frac{B}{T}$
$S = \frac{B}{C}$
$N = \frac{W}{T}$
$R = \frac{W}{C}$
$X = \lambda$ if stable

Table III. PERFORMANCE: LAWS

Law	Definition
Visits	$V_k = \frac{C_k}{C}$
Demand	$D_k = V_k S_k$
Utilization	$U_k = X_k S_k = X_0 D_k$
Little's	$N_k = X_k R_k$
Response time	$R = \frac{N}{X} - Z$
Forced flow	$X_k = X_0 V_k$
Queue length	$N_k - U_k$
Queue time	$R_k - D_k$

Table IV. PERFORMANCE: BOUNDS

Bounds	Open	Closed
Resp. Time	$R \geq \sum D_k$	$\max(\sum D_k, ND_{\max} - Z) \leq R \leq N \sum D_k$
Throughput	$X \leq \frac{1}{D_{\max}}$	$\frac{N}{N \sum D_k + Z} \leq X \leq \min\left(\frac{1}{D_{\max}}, \frac{N}{\sum D_k + Z}\right)$
N^*		$N^* = \frac{\sum D_k + Z}{D_{\max}}$

Table V. AVAILABILITY: VARIABLES

Variable	Definition
λ	Failure rate
$F(t)$	Failure probability
$R(t)$	Reliability
A	Availability
$MTTF$	Mean Time to Failure
$MTTR$	Mean Time to Repair
AFR	Annualized Failure Rate
P_{on}	# power hours per year

Table VI. AVAILABILITY: RELATIONS

Relations
$\lambda = 1/MTTF$ †
$R(t) = 1 - F(t)$
$R(t) = e^{-\frac{t}{MTTF}}$ †
$R(t) \approx 1 - \frac{t}{MTTF}$ if $t \ll MTTF$ †
$AFR = e^{-\frac{P_{on}}{MTTF}}$ †
$R_{Ser.}(t) = \prod R_i(t)$
$R_{Par.}(t) = 1 - \prod(1 - R_i(t))$
$MTTF_{Ser.} = \left(\sum \frac{1}{MTTF_i}\right)^{-1}$ †
$MTTF_{Ser.} = \frac{MTTF}{N}$ if *i.i.d.* †
$MTTF_{Par.} = MTTF \sum_{n=1}^{N} \frac{1}{n}$ if *i.i.d.* †
$MTTF_{Par2} = MTTF_1 + MTTF_2 - \frac{MTTF_1 \cdot MTTF_2}{MTTF_1 + MTTF_2}$ †
$A = \frac{MTTF}{MTTF + MTTR}$
$A_{Ser.} = \prod A_i$
$A_{Par.} = 1 - \prod(1 - A_i)$
† Exponential assumption

Table VII. RAID: VARIABLES

Variable	Definition
$MTTF$	Mean Time to Failure
$MTTR$	Mean Time to Repair
$MTTDL$	Mean Time to Data Loss
N	Number of disks in the array
G	Number of disks in a group

Relations

$MTTDL_{RAID0} = \frac{MTTF}{N}$
$MTTDL_{RAID1} = \frac{MTTF^2}{2 \cdot MTTR}$
$MTTDL_{RAID1+0} = \frac{MTTF^2}{N \cdot MTTR}$
$MTTDL_{RAID0+1} = \frac{2 \cdot MTTF^2}{N^2 \cdot MTTR}$
$MTTDL_{RAID5} = \frac{MTTF^2}{N \cdot (N-1) \cdot MTTR}$
$MTTDL_{RAID6} = \frac{2 \cdot MTTF^3}{N \cdot (N-1) \cdot (N-2) \cdot MTTR^2}$
$MTTDL_{RAID5+0} = \frac{MTTF^2}{N \cdot (G-1) \cdot MTTR}$
$MTTDL_{RAID6+0} = \frac{2 \cdot MTTF^3}{N \cdot (G-1) \cdot (G-2) \cdot MTTR^2}$

Table IX. RAID PARITY: VARIABLES

Variable	Definition
D_i	Data on i-th disk ($0 \leq i < N$)
P	Parity data
Q	Second Parity data
g	Parity generator

Table X. RAID 5 AND 6 PARITY P

Parity Computation	$P = \sum_{i=0}^{N-1} D_i$
Parity Update	$P^{new} = P^{old} + D_i^{new} - D_i^{old}$

Table XI. RAID 6 PARITY Q

Parity Computation	$Q = \sum_{i=0}^{N-1} g^i D_i$
Parity Update	$Q^{new} = Q^{old} + g^i (D_i^{new} - D_i^{old})$

Table XII. RAID PARITY: RECONSTRUCTION

Failed	Reconstruction
D_i	$D_i = P - \sum_{j \neq i} D_j$
P	$P = \sum_{i=0}^{N-1} D_i$
Q	$Q = \sum_{i=0}^{N-1} g^i D_i$
D_i and P	$D_i = g^{-i} \left(Q - \sum_{j \neq i} g^j D_j \right)$
D_i and D_k	Solve the system of equations: $\begin{cases} P = D_i + D_j + \sum_{k=0, k \neq i,j}^{N-1} D_k \\ Q = g^i D_i + g^j D_j + \sum_{k=0, k \neq i,j}^{N-1} g^k D_k \end{cases}$

Table XIII. DISKS: VARIABLES

Variable	Definition
T_s	Mean seek time
T_{sMax}	Max seek time
T_t	Mean transfer time (for one block)
B	Block size or Page Size
r_t	Transfer rate
T_l	Rotational latency time
r_r	Rotational speed
T_c	Controller overhead
T_{tP}	Mean transfer time (for one page)
T_{rP}	Mean read time (for one page)
T_b	Mean service time (for one block)
T_a	Mean service time (for one file)
F	File size
l	data locality
N	stripe width

$T_b = T_s + T_l + T_t + T_c$
$T_b = (T_s + T_l + T_t)/N + T_c$ †1
$T_b = T_{sMax} + 2T_l + T_t/N + T_c$ †2
$T_l = \frac{1}{2 \cdot r_r}$
$T_t = \frac{B}{r_t}$
$T_a = \lceil F/B \rceil \cdot T_b$ ⋆
$T_a = \lceil F/B \rceil \cdot [T_t + T_c + (1-l)(T_s + T_l)]$ °
$T_a = \lceil F/B \rceil \cdot [T_c + (T_t + (1-l)(T_s + T_l))/N]$ °,†
$T_a = \lceil F/B \rceil \cdot [T_c + T_t/N + (1-l)(T_{sMax} + 2T_l)]$ °
†1 for RAID 0, coarse grained
†2 for RAID 0, fine grained
⋆ for one file, without locality
° for one file, with locality

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `combineByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

RDD Persistence

One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

You can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

In addition, each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes, or store it off-heap in Tachyon. These levels are set by passing a `StorageLevel` object (Scala, Java, Python) to `persist()`. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY` (store deserialized objects in memory). The full set of storage levels is:

Storage Level	**Meaning**
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.

Note: *In Python, stored objects will always be serialized with the Pickle library, so it does not matter whether you choose a serialized level.*

Spark also automatically persists some intermediate data in shuffle operations (e.g. `reduceByKey`), even without users calling `persist`. This is done to avoid recomputing the entire input if a node fails during the shuffle. We still recommend users call `persist` on the resulting RDD if they plan to reuse it.

Answer sheet:

Last Name / Cognome:

....

First Name / Nome:

....

Answers must be given exclusively on this sheet: answers given on the other sheets will be ignored.

Student ID : 0 1 2 3 4 5 6 7 8 9

Student ID : 0 1 2 3 4 5 6 7 8 9

Student ID : 0 1 2 3 4 5 6 7 8 9

Student ID : 0 1 2 3 4 5 6 7 8 9

Student ID : 0 1 2 3 4 5 6 7 8 9

Student ID : 0 1 2 3 4 5 6 7 8 9

Disks

Question 01 : **A B C D**

Question 02 : **A B C D**

Question 03 : **A B C D**

Question 04 : **A B C D**

Question 05 : **A B C D**

Question 06 : **A B C D**

Question 07 **(R)**: **A B C D**

Performance Evaluation

Question 08 : **A B C D**

Question 09 : **A B C D**

Question 10 : **A B C D**

Question 11 : **A B C D**

Question 12 : **A B C D**

Dependability

Question 13 : **A B C D**

Question 14 : **A B C D**

Question 15 : **A B C D**

Question 16 : **A B C D**

Question 17 : **A B C D**

Question 18 **(R)**: **A B C D**

Cloud Computing

Question 19 : **A B C D**

Question 20 : **A B C D**

ITIL and DevOps

Question 21 : **A B C D**

Question 22 : **A B C D**

Question 23 : **A B C D**

Question 24 **(R)** : **A B C D**

Virtualization

Question 25 : **A B C D**

Question 26 : **A B C D**

Question 27 : **A B C D**

Question 28 : **A B C D**

Question 29 : **A B C D**

BigData

Question 30 : **A B C D**

Question 31 : **A B C D**

Question 32 : **A B C D**

Question 33 : **A B C D**

Question 34 : **A B C D**

Question 35 : **A B C D**

Question 36 **(R)** : **A B C D**