There are 12 questions and ?? pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. You can receive partial credit on questions. You have 80 minutes to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet. Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can’t understand your answer, we can’t give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ. NO PHONES, NO LAPTOPS, NO PDAS, ETC. YOU MAY USE A CALCULATOR.

Do not write in the boxes below

<table>
<thead>
<tr>
<th>1-4 (xx/30)</th>
<th>5-7 (xx/26)</th>
<th>8-10 (xx/24)</th>
<th>11-12 (xx/20)</th>
<th>Total (xx/100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \mu = 70.9, \sigma = 12.9 \]

Solutions
I Short Answer

1. [6 points]: Ben Bitdiddle is experimenting with shared nothing and shared disk parallel databases. Sometimes he finds that shared nothing systems perform better than shared disk systems, and sometimes he finds that shared disk systems perform better than shared nothing systems.

Describe one setting, workload, or scenario in which you would expect a shared disk system to perform better than a shared nothing system.

(Write your answer in the space below.)

A number of answers were accepted for this question. For example: A shared disk system might perform better if there are many read queries for the same row. These can be served in parallel on many servers in a shared disk system, but would have to be served on a single server if the data is partitioned in a shared nothing system.

2. [6 points]: Describe one setting, workload, or scenario in which you would expect a shared nothing system to perform better than a shared disk system.

(Write your answer in the space below.)

A number of answers were accepted for this question. For example: Lots of concurrent updates to random rows. A shared nothing system can run these in parallel on all servers. A shared disk system can also run them in parallel, but the servers must communicate to lock data and to ensure they have the latest page contents.
3. [8 points]: In the paper “C-Store: A Column Oriented DBMS”, different compression schemes are proposed for different types of data. For each of the following columns indicate which compression scheme is likely to offer the best compression ratio? Choose from:

- Type 1: Run Length Encoding
- Type 2: Bitmap encoding
- Type 3: Delta encoding
- Type 4: Generic compression tool (e.g., gzip or zip) – assume that this option gets about 20% reduction in data size regardless of input data.

(Circle the best compression scheme in each case.)

A. An unsorted column of last names of students at MIT

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
<th>Type 4</th>
</tr>
</thead>
</table>

Type 4: Names are not repeated, so run-length. They are many-valued, so bitmaps are a poor choice. Delta encoding won’t work well on this sort of variable-length string.

B. A sorted column of salaries of employees at MIT, assuming many employee salaries

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
<th>Type 4</th>
</tr>
</thead>
</table>

Type 3: If you have many employee salaries, you won’t have many repeated salaries, so run-length won’t help much. Bitmaps don’t work well for many-valued data. However, if sorted, the salaries will tend to be fairly close to each other, so you’ll have quite-small deltas.

C. An unsorted column of genders of students at MIT (assume genders are represented as 1 character strings)

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
<th>Type 4</th>
</tr>
</thead>
</table>

Type 2: With bitmap encoding, you can replace the one character (8 bits) with a much smaller number of bits (likely a single bit in this case). This means that a bitmap encoding will typically improve by a factor of 8:1. The list is unsorted, so run-length encoding will not be much of a win. ‘M’ - ‘F’ = +/-8, which takes 4 bits to store; so delta encoding will be worse than bitmap encoding.

D. A sorted column of birth-years (ages) of employees at MIT (MIT has thousands of employees)

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
<th>Type 4</th>
</tr>
</thead>
</table>

Type 1: Assume that it is unlikely that the age range from the youngest employee to the oldest employee at MIT is much more than 100 years. Also, assume conservatively that there are 1,000 employees. This means that, for any given possible age of an employee, there are about 10 employees of that age, so run-length encoding gives around a 10:1 savings; for more employees, the numbers are better. A bitmap would store 100 possible ages in 100 separate bitmaps, requiring more space than storing the age explicitly. Delta encoding may be able to cut the number of bits needed per entry, but it can’t cut it to less than one bit per person, which limits its potential.
You’re running a PNUTS system (see the paper by Cooper et al.). Data items X and Y both start with value zero. Here are two functions that use the API described in Section 2.2 of the PNUTS paper:

```python
fn1:
    x1 = read-any(X)
    x1 = x1 + 1
    write(X, x1) // X = x1
    write(Y, x1) // Y = x1

fn2:
    x1 = read-any(X)
    x2 = read-latest(X)
    y1 = read-any(Y)
    print x1, x2, y1
```

You execute two calls to fn1, at different sites, at the same time. After both calls to fn1 have returned, you execute fn2 at a third site. There is no activity in the system other than described here, and no crashes or network failures.

4. [10 points]: What output is it possible to see from fn2, given the design of PNUTS and the above scenario?

(Circle Yes for outputs that PNUTS could produce, and No for outputs that PNUTS could not produce.)

A. Yes / No 2, 2, 1 Yes, since the update to Y could take a long time for the message broker to propagate.

B. Yes / No 1, 2, 2 Yes, since the update to X could take a long time to propagate.

C. Yes / No 1, 1, 0 Yes, since fn1 and fn2 could both read X=0 and both write X=1.

D. Yes / No 2, 1, 1 No, since read-latest sees any update that read-any sees.

E. Yes / No 0, 0, 0 No, since both functions finished, so read-latest must see their changes.
5. [10 points]: Which of the following are true of BigTable, as described in the paper by Chang et al.? (Circle True or False for each statement.)

A. True / False  Adding a new column is expensive, since it requires moving existing columns to create space for the new column.
   False. One of the advantages of a column-oriented design is that adding new columns shouldn’t require moving existing columns.

B. True / False  BigTable is optimized for reads, and has comparatively slow writes.
   False. The tablet log/memtable scheme in Section 5.3 is optimized for writes, and the performance measurements in Figure 6 show that writes are often faster than reads.

C. True / False  BigTable stores each tablet on the disks of two tablet servers, using primary/backup replication.
   False. Tablets are stored and replicated by GFS (which may store more than two copies), and doesn’t rely on explicit tablet-level replication.

D. True / False  If the BigTable master crashes, clients must wait for it to be restarted before they can do more work.
   False. The paper says clients generally do not need to contact the BigTable master at all. Even if they did, BigTable will create a new master, so the old master need never restart.

E. True / False  BigTable stores tables in B+Trees on disk for fast primary-key lookups.
   False. BigTable stores data on disk (via GFS) as tablet logs and SSTables, neither of which is a B+Tree. It has an in-memory index that it uses for optimizing memory-resident lookups, which is also not a B+Tree.

6. [8 points]: Which of the following statements about the Aurora system, as described in the paper by Abadi et al., are true?  (Circle True or False for each statement.)

A. True / False  The primary purpose of QoS curves is to figure out whether to shed load.
   True: The QoS curves are global functions provided by the user to let Aurora decide whether to shed load and what to shed.

B. True / False  Input data streams are assumed to arrive in time order.
   False: Aurora is designed to handle situations where the data doesn’t arrive in order from the data feeds.

C. True / False  The output of the BSORT operator is totally ordered according to the value of the attribute A in the order clause.
   False: Aurora splits the input into windows and sorts each window. This produces unsorted output when the input is way out of order.

D. True / False  Aurora uses an iterator-style model for passing tuples between operators.
   False: Aurora uses a scheduler to determine which operator will be processed next.
II Parallel Databases

Dana Bass is trying to figure out how to best partition the tables in her 2-machine parallel database. The database she is using supports hash, range, and round-robin partitioning. When using hash and range partitioning she is required to specify the partitioning field, and when using range partitioning she must also specify the ranges of the partitioning attribute that are to be placed on each machine.

Her database consists of two large tables $A$ and $B$ of approximately the same size, each much larger than the sum of the two machines’ memories, with schemas $A(a, b)$ and $B(a, b)$. All fields are integers and are randomly and uniformly distributed. Each server stores a single heap file for each table in no particular order, and there are no indices.

7. [8 points]: For each of the following four workloads, specify which partitioning strategy from amongst the following will minimize the response time of the workload, assuming that CPU costs are negligible in comparison to network and disk I/O, that all disk I/O is sequential, and that disk bandwidth is comparable to network bandwidth.

Choose from these strategies:

a. Round robin partition both tables
b. Hash partition $A$ on $A.a$ and $B$ on $B.b$

b. Range partition $A$ on $A.a > 150$ and $B$ on $B.b > 150$

d. Range partition $A$ on $A.a > \min(A.a) + \frac{\max(A.a) - \min(A.a)}{2}$ and $B$ on $B.b > \min(B.b) + \frac{\max(B.b) - \min(B.b)}{2}$

(For each workload, circle the best partitioning a, b, c, or d. If several partitionings are equivalent, circle one.)

A. Transactions $T_1$ through $T_n$ issued concurrently, each of which inserts a new tuple into $A$, where transaction $T_i$’s tuple is $(i, 10)$; assume $n$ is unknown when the partitioning is performed.

a b c d

a or b: Both of them will distribute the tuples into the 2 machines equally.

B. Many transactions issued serially, each of which runs a query that joins $A.a = B.b$, with a predicate $B.b > 150$ (assume $A.a$ and $B.b$ are both randomly and uniformly distributed over the range $[0 \ldots 200]$).

a b c d

c: Round robin and hash partitions provide no optimization for this query. The range partition in c is the most optimal because it allows the query to be answered by scanning only 1/4th of the tuples.

C. Many transactions issued serially, each of which runs a range query of the form $B.a > x$ ($x$ is a variable, chosen uniformly and at random from the range $[0 \ldots \max(B.a)]$).

a b c d

a, b or d: None of the options help much, other than distributing the data into two equal partitions. Choice c is non-ideal because it doesn’t distribute the data into equal parts.
D. Many transactions issued concurrently, each of which runs a query of the form $A.a = y$ (y is a variable, chosen uniformly and at random from the range $[0 \ldots \max(A.a)]$).

b or d: Both $b$ and $d$ will cause each query to be run only on one of the machines, scanning half of the tuples. Round-robin partitioning requires each query to be run on both machines, so is non-ideal.
III Chitter

Alyssa P. Hacker is building an instant messaging system called Chitter. In Chitter, messages can be addressed to one or more other users. Chitter is supposed to provide the following transactional guarantees about messages:

- **All or nothing**: When a message is sent, it is delivered to all of its recipients, or none of them; if one message cannot be delivered (because, for example, the machine that is supposed to store it is unavailable), none of the recipients should receive it.

- **In order**: If user U1 and U2 both receive messages M1 and M2, then if U1 sees M1 before M2, U2 should also see M1 before M2.

Alyssa implements Chitter as a three-node distributed database. Each user is given a separate table for his or her incoming messages, and each user’s messages table is stored on one of the nodes. When a message is sent, it is written into the messages table of each user who receives it.

8. **[8 points]**: Initially, to ensure that Chitter provides the all-or-nothing property, Alyssa uses a database system that uses two-phase commit (in the standard form, without presumed abort or presumed commit), and writes each message as a part of a single transaction. Even though her machines are on a fast LAN with network round trip times of only 100 ns, she finds that it takes about 20 ms to commit a transaction for a message, even when only one transaction is running at a time. Explain, in one sentence, what is likely going on.

(Write your answer in the space below.)

*Two phase commit requires several forced log writes (a prepare message on each worker, followed by a commit on the coordinator.) Each of these probably takes about 10 ms.*
To improve the performance of Chitter, Ben Bitdiddle tells Alyssa that because messages are never deleted or updated, she can use a much simpler protocol than two-phase commit. Ben tells Alyssa about the Network Time Protocol (NTP), which can ensure that nodes in a cluster have clocks that are synchronized (i.e., agree with each other) to within 1 millisecond. He proposes Alyssa run NTP on her machines and then use the following protocol for sending a message:

- To send a message $m$, a client connects to one of the nodes $N$. The node assigns the message a 96-bit commit timestamp $TS_m$. To compute a timestamp, a node appends its unique 32-bit nodeid onto a 64-bit time in milliseconds since 1/1/2000 (the high order bits of the timestamp are the time); a node never issues the same timestamp for two messages.

- $N$ sends $m$ to each of the nodes that store the table for a user who should receive $m$, along with the value $TS_m$.

- When a node receives a message $m$ for a user $u$, it waits 2 milliseconds. If no messages arrive with a timestamp less than $TS_m$, the node “posts” $m$ by running a local transaction to write the message to $u$’s table. Otherwise, it posts the messages with timestamps less than $TS_m$ and then posts $m$.

9. **[8 points]**: Assuming that nodes can crash at any time, but that the network is never partitioned, delivers all messages, and never takes more than 100 ns to send messages between two nodes, does Ben’s protocol preserve the all-or-nothing property of Chitter? Why or why not?

(Write your answer in the space below.)

No. If a node receives a message and then crashes, it may lose the message while other nodes may display it, since there’s no logging in Ben’s protocol.
The protocol above does preserve the *in order* property. However, if it is modified such that nodes only wait 0.5 ms instead of 2 ms, the *in order* property can be violated.

10. [8 points]: Given an example of a sequence of messages and timestamps that could result in a violation of the Chitter *in order* property when nodes only wait 0.5 ms.

(Write your answer in the space below.)

There are several possible ways in which Ben’s protocol can fail. Here is one: suppose the following sequence of events (note that .5 ms is 500,000 ns):

100 ns  N2 produces message m₂ with timestamp {0, 2} (this notation means a message with timestamp 0 milliseconds was produced by node 2) for nodes {N₁, N₂} and sends it to N₁.

200 ns  m₂ arrives time at N₁

500,050 ns  N₁ produces message m₁ with timestamp {0, 1} for {N₁, N₂} and sends it to N₂

500,100 ns  N₂ posts message m₂ after waiting 500,000 ns (.5 ms). This is before it has received message m₁ which should be ordered earlier because the messages have the same time value and N₁ < N₂.

500,150 ns  m₁ arrives at N₂

500,200 ns  N₁ posts message m₁ before message m₂, since after N₁ receives m₂, it starts waiting and receives message m₁ at time 500,050 ns. It therefore (correctly) posts m₁ before m₂.

The in-order property is violated because the nodes post the messages in different orders. This is illustrated in the following diagram:

![Diagram showing the sequence of events and timestamps](image)

Note that with a 2 ms timeout this problem does not arise, because if two nodes timestamped messages in the same millisecond, and messages take 100 ns to propagate, each node will definitely hear the message from the other node before posting its own message (and the way we have constructed timestamps ensures nodes will agree on the order in which messages should be posted).

Other failure cases can arise if you assume three nodes, with one node sending the same message to two receivers, and the message taking different amounts of time to reach each receiver. Interestingly, the inaccuracy of time synchronization between the nodes does not matter in either case.
IV Logging

You have a database with logging and recovery as described in the ARIES paper. Your database uses strict two-phase locking. Your database has just two items in it, X with starting value 10, and Y with starting value 100.

You start three transactions at the same time, with transaction IDs (TIDs) TA, TB, and TC:

**TA:**
```
BEGIN TRANSACTION
  X = X + 1
  Y = Y * 3
END
```

**TB:**
```
BEGIN TRANSACTION
  Y = Y * 2
  X = X + 5
END
```

**TC:**
```
BEGIN TRANSACTION
  X = X * 10
END
```

These three transactions are the only activity in the system. The system crashes due to a power failure soon after you start the transactions. You are not sure whether or not any of them completed. You look at the disk while the system is down and see that, in the heap file, Y has the value 200. You restart the system and let the database recovery procedure complete. You query the database for the value of X, and it returns the value 110.
11. [12 points]: Please write down a log, as it would have appeared on the disk while the system was down, that is compatible with the above story. You need only include Update (U), Commit (C), and Abort (A) records. Specify the TID (TA, TB, or TC) and record type for each record, and for Updates, the item being written (X or Y) and the new value being written.

(Write your answer in the empty log below.)

<table>
<thead>
<tr>
<th>TID</th>
<th>Type</th>
<th>Item</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>Update</td>
<td>X</td>
<td>11</td>
</tr>
<tr>
<td>TB</td>
<td>Update</td>
<td>Y</td>
<td>200</td>
</tr>
<tr>
<td>TB</td>
<td>Abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TA</td>
<td>Update</td>
<td>Y</td>
<td>300</td>
</tr>
<tr>
<td>TA</td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC</td>
<td>Update</td>
<td>X</td>
<td>110</td>
</tr>
<tr>
<td>TC</td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The buffer manager writes Y to disk after TB sets Y = 200, and before TB aborts, which it is allowed to do under a STEAL policy. The buffer manager does not write Y to disk after TA commits, which it is allowed to do under a NO-FORCE policy. One reason TB might abort is that it deadlocks with TA.

12. [8 points]: ARIES stores both undo and redo information in log records. It is possible to change the design of ARIES to obtain a correct logging and recovery system that uses only redo information, so that an update record contains “after” value(s) but no “before” values (and no logical undo information). Describe:

A. What you would have to change about the rest of the system to get recoverability in such an ARIES-based “redo-only” system, and

B. The main reasons why the ARIES redo/undo approach is significantly better than this redo-only approach.

(Write your answer in the space below.)

Without undo, the buffer manager cannot write dirty pages to disk until the dirtying transaction commits. That is, the buffer manager must implement a NO-STEAL policy.

ARIES’ use of undo allows STEAL, which allows dirty pages of long-running transactions to be written to disk, which helps performance if other transactions need space for disk pages. If two transactions modify tuples on the same page, STEAL also allows one of them to commit without waiting for the other.

End of Quiz II