CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2022)
PROF. ANDY PAVLO

Homework #5 (by Chi Zhang, Tim Lee) – Solutions

Due: **Thursday Dec 4, 2022 @ 11:59pm**

IMPORTANT:

- **Upload this PDF** with your answers to **Gradescope by 11:59pm on Thursday Dec 4, 2022**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- **You have to use this PDF for all of your answers.**

For your information:

- Graded out of **100** points; **4** questions total

Revision : 2022/12/08 15:59

Question	Points	Score
Write-Ahead Logging	25	
Replication	20	
Two-Phase Commit	25	
Distributed Query Plan	30	
Total:	100	

Question 1: Write-Ahead Logging............................[25 points]

Consider a DBMS using write-ahead logging with physical log records with the STEAL and NO-FORCE buffer pool management policy. Assume the DBMS executes a non-fuzzy checkpoint where all dirty pages are written to disk.

Its transaction recovery log contains log records of the following form:

```
<txnId, objectId, beforeValue, afterValue>
```

The log also contains checkpoint, transaction begin, and transaction commit records.

The database contains three objects (i.e., A, B, and C).

The DBMS sees records as in Figure 1 in the WAL on disk after a crash.

Assume the DBMS uses ARIES as described in class to recover from failures.

LSN	WAL Record
1	`<T1 BEGIN>`
2	`<T1, B, 6, 7>`
3	`<T1, C, 42, 43>`
4	`<T2 BEGIN>`
5	`<T2, A, 33, 71>`
6	`<T1 COMMIT>`
7	`<T2, C, 43, 100>`
8	`<T3 BEGIN>`
9	`<T3, B, 7, 20>`
10	`<T2, C, 100, 67>`
11	`<`**`CHECKPOINT`**`>`
12	`<T3, B, 20, 42>`
13	`<T2, A, 71, 13>`
14	`<T2 COMMIT>`
15	`<T3, B, 42, 66>`

Figure 1: WAL

(a) **[6 points]** What are the values of A, B, and C in the database stored on disk before the DBMS recovers the state of the database?

- ☐ A=71, B=6, C=100
- ☐ A=13, B=66, C=67
- ☐ A=43, B:7, C=Not possible to determine
- ☐ A=71, B=42, C=42
- ☐ A=Not possible to determine, B:20, C=43
- ☐ A=43, B:20, C=Not possible to determine
- ■ **A:Not possible to determine, B=Not possible to determine, C:67**
- ☐ A=Not possible to determine, B=20, C:Not possible to determine
- ☐ A:71, B=Not possible to determine C=42

Question 1 continues...

☐ A,B,C:Not possible to determine

Solution: The checkpoint flushed everything to disk, but then the data objects A,B were modified by transactions after the checkpoint.

Since we are using STEAL, any dirty page could be written to disk, so therefore we don't know the contents of the database on disk at the crash.

(b) **[3 points]** What should be the correct action on T1 when recovering the database from WAL?

- ☐ redo all of T1's changes
- ☐ undo all of T1's changes
- ■ **do nothing to T1**

Solution: T1 committed before the checkpoint. All of its changes were written to disk. There is nothing to redo or undo.

(c) **[3 points]** What should be the correct action on T2 when recovering the database from WAL?

- ■ **redo all of T2's changes**
- ☐ undo all of T2's changes
- ☐ do nothing to T2

Solution: T2 committed after the checkpont, so that means the DBMS has to redo all of its changes.

(d) **[3 points]** What should be the correct action on T3 when recovering the database from WAL?

- ☐ redo all of T3's changes
- ■ **undo all of T3's changes**
- ☐ do nothing to T3

Solution: T3 never committed. All of its changes should only be undone.

(e) **[10 points]** Assume that the DBMS flushes all dirty pages when the recovery process finishes. What are the values of A, B, and C after the DBMS recovers the state of the database from the WAL in Figure 1?

- ☐ A=33, B=6, C=42
- ☐ A=13, B=66, C=67
- ☐ A=13, B=6, C=100
- ■ **A=13, B=7, C=67**
- ☐ A=71, B=20, C=42

Question 1 continues…

- ☐ A=33, B=42, C=100
- ☐ A=71, B=7, C=100
- ☐ A=13, B=42, C=67
- ☐ A=33, B=20, C=43
- ☐ A=71, B=66, C=43
- ☐ A=13, B=42, C=42
- ☐ Not possible to determine

Solution: A = 13 (rollback to the afterValue made by T2)
B = 7 (committed by T1)
C = 67 (rollback to the afterValue made by T2)

Question 2: Replication ....[20 points]

Consider a DBMS using active-passive, master-replica replication with multi-versioned concurrency control. All read-write transactions go to the master node (NODE A), while read-only transactions are routed to the replica (NODE B). You can assume that the DBMS has "instant" fail-over and master elections. That is, there is no time gap between when the master goes down and when the replica gets promoted as the new master. For example, if NODE A goes down at timestamp ① then NODE B will be elected the new master at ②.

The database has a single table `foo(id,val)` with the following tuples:

id	**val**
1	xx
2	yy
3	zz

Table 1: **`foo(id,val)`**

For each questions listed below, assume that the following transactions shown in Figure 2 are executing in the DBMS: (1) Transaction #1 on NODE A and (2) Transaction #2 on NODE B. You can assume that the timestamps for each operation is the real physical time of when it was invoked at the DBMS and that the clocks on both nodes are perfectly synchronized.

time	operation
①	`BEGIN;`
②	`UPDATE foo SET val = 'x';`
③	`UPDATE foo SET val = 'yyy' WHERE id = 3;`
④	`UPDATE foo SET val = 'z' WHERE id = 1;`
⑤	`COMMIT;`

(a) Transaction #1 – NODE A

time	operation
②	`BEGIN READ ONLY;`
③	`SELECT val FROM foo WHERE id = 3;`
④	`SELECT val FROM foo WHERE id = 1;`
⑤	`SELECT val FROM foo WHERE id = 1;`
⑥	`COMMIT;`

(b) Transaction #2 – NODE B

Figure 2: Transactions executing in the DBMS.

(a) Assume that the DBMS is using *asynchronous* replication with *continuous* log streaming (i.e., the master node sends log records to the replica in the background after the transaction executes them). Suppose that NODE A crashes at timestamp ⑤ before it executes the `COMMIT` operation.

i. **[6 points]** If Transaction #2 is running under `READ COMMITTED`, what is the return result of the `val` attribute for its `SELECT` query at timestamp ⑤? Select all that are possible.

☐ zz
☐ x
☐ yy
☐ yyy
☐ z

Question 2 continues...

■ **xx**
☐ None of the above

Solution: READ COMMITTED means that the transaction will only see the versions that were committed. That means at ⑤, Transaction #1 has not committed yet so therefore Transaction #2 cannot see any of its versions.

ii. **[7 points]** If Transaction #2 is running under the `READ UNCOMMITTED` isolation level, what is the return result of the `val` attribute for its `SELECT` query at timestamp ⑤? Select all that are possible.

☐ zz
■ **x**
☐ yy
☐ yyy
■ **z**
■ **xx**
☐ None of the above

Solution: READ UNCOMMITTED means that it will read any version of the tuple that exists in the database. But what version of tuple 1 that the transaction will read depends on whether the master node shipped the log record over before the query is executed. Since we are doing continuous log shipping, we have no idea. So it could read the version of the tuple that existed *before* Transaction #1 started (i.e., "xx") or after Transaction #1 executed the UPDATE query at ② (i.e., "x"), or after Transaction #1 executed the UPDATE query at ④ (i.e., "z").

(b) **[7 points]** Assume that the DBMS is using *synchronous* replication with *on commit* propagation. Suppose that both NODE A and NODE B crash at exactly the same time at timestamp ⑥ after executing Transaction #1's `COMMIT` operation. You can assume that the application was notified that the Transaction #1 was committed successfully.

After the crash, you find that NODE A had a major hardware failure and cannot boot. NODE B is able to recover and is elected the new master.

What are the values of the tuples in the database when the system comes back online? Select all that are possible.

☐ { (1,xx), (2,yy), (3,zz) }
☐ { (1,x), (2,x), (3,x) }
☐ { (1,x), (2,x), (3,yyy) }
■ **{ (1,z), (2,x), (3,yyy) }**
☐ { (1,xx), (2,x), (3,zz) }
☐ { (1,xx), (2,x), (3,x) }
☐ None of the above

Solution: Synchronous replication with On Commit propagation means that the replica received the log records from the master when Transaction #1 committed. The master sent the notification to the client that the txn committed only when it was guaranteed to be durable on disk on the master and the replica. When the system come back on-line,

Question 2 continues...

we know that the txn was also flushed to disk on the replica. Thus, the only correct state of the database is if Transaction #1 did execute. There cannot be any partial updates to the database.

Question 3: Two-Phase Commit .... [25 points]

Consider a distributed transaction T operating under the two-phase commit protocol with the early acknowledgement optimization. Let N_0 be the *coordinator* node, and N_1, N_2, N_3 be the *participant* nodes.

The following messages have been sent:

time	message
1	N_0 to N_2: "**Phase1:PREPARE**"
2	N_2 to N_0: "**OK**"
3	N_0 to N_1: "**Phase1:PREPARE**"
4	N_0 to N_3: "**Phase1:PREPARE**"

Figure 3: Two-Phase Commit messages for transaction T

(a) **[7 points]** Who should send a message next at time 5 in Figure 3? Select *all* the possible answers.

☐ N_0
■ N_1
☐ N_2
■ N_3
☐ It is not possible to determine

Solution: N_1 has to send a response to N_0 N_3 has to send a response to N_0

(b) **[6 points]** To whom? Again, select *all* the possible answers.

■ N_0
☐ N_1
☐ N_2
☐ N_3
☐ It is not possible to determine

Solution: N_1 has to send a response to N_0 N_3 has to send a response to N_0

(c) **[6 points]** Suppose that N_0 received the "**ABORT**" response from N_1 at time 5 in Figure 3. What should happen under the two-phase commit protocol in this scenario?

☐ N_0 resends "**Phase1:PREPARE**" to N_2
☐ N_1 resends "**OK**" to N_0
☐ N_0 sends "**Phase2:COMMIT**" all of the participant nodes
■ **N_0 sends "ABORT" all of the participant nodes**
☐ N_0 resends "**Phase1:PREPARE**" to all of the participant nodes
☐ It is not possible to determine

Question 3 continues…

Solution: The coordinator (N_0) will mark the transaction as aborted. 2PC requires that *all* participants respond with "`OK`".

(d) **[6 points]** Suppose that N_0 successfully receives all of the "`OK`" messages from the participants from the first phase. It then sends the "`Phase2:COMMIT`" message to all of the participants but N_1 and N_3 crash before they receives this message. What is the status of the transaction T when N_1 comes back on-line?

☐ T's status is *aborted*

[✓ **T's status is *committed***]

☐ It is not possible to determine

Solution: Once the coordinator (N_0) gets a "`OK`" message from *all* participants, then the transaction is considered to be committed even though a node may crash during the second phase. In this example, N_1 and N_3 would have restore T when it comes back on-line.

Question 4: Distributed Query Plan .... [30 points]

The CMUDB group is working on a brand new shared-nothing distributed database system called BusTub*. They are developing the distributed query engine.

Given the following schema:

```
CREATE TABLE t1(PARTITION KEY v1 int, v2 int);
CREATE TABLE t2(PARTITION KEY v3 int, v4 int);
CREATE TABLE t3(PARTITION KEY v5 int, v6 int);
CREATE TABLE t4(PARTITION KEY v7 int, v8 int);
```

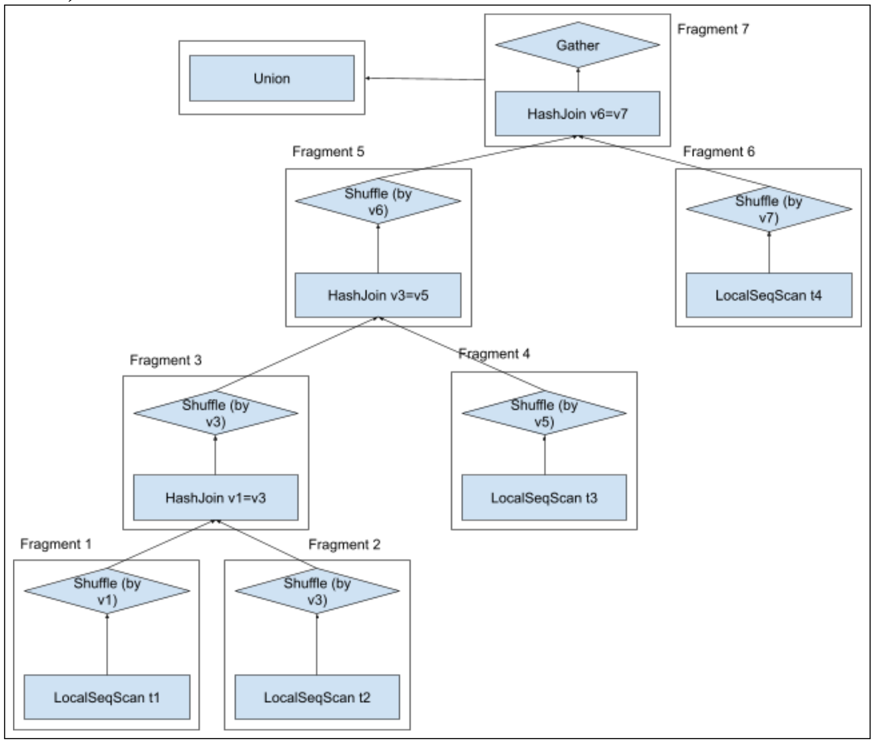
The database system partitions the tables by key range. That is to say, each node in the system manages rows of the table within a non-overlapping range of keys.

Given the following query:

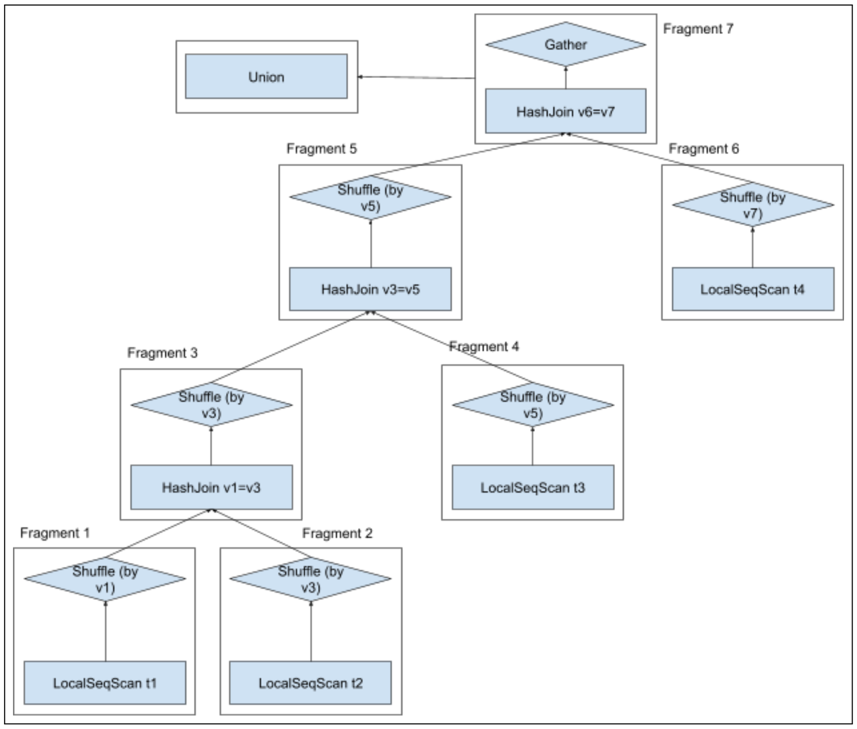
```
SELECT * FROM ((t1 INNER JOIN t2 ON v1 = v3)
    INNER JOIN t3 ON v3 = v5)
    INNER JOIN t4 ON v6 = v7;
```

(a) **[5 points]** Assume that the query optimizer doesn't know the ranges of data stored on each node and only knows the tables are sharded by some keys, what are the correct distributed query plans for this query?

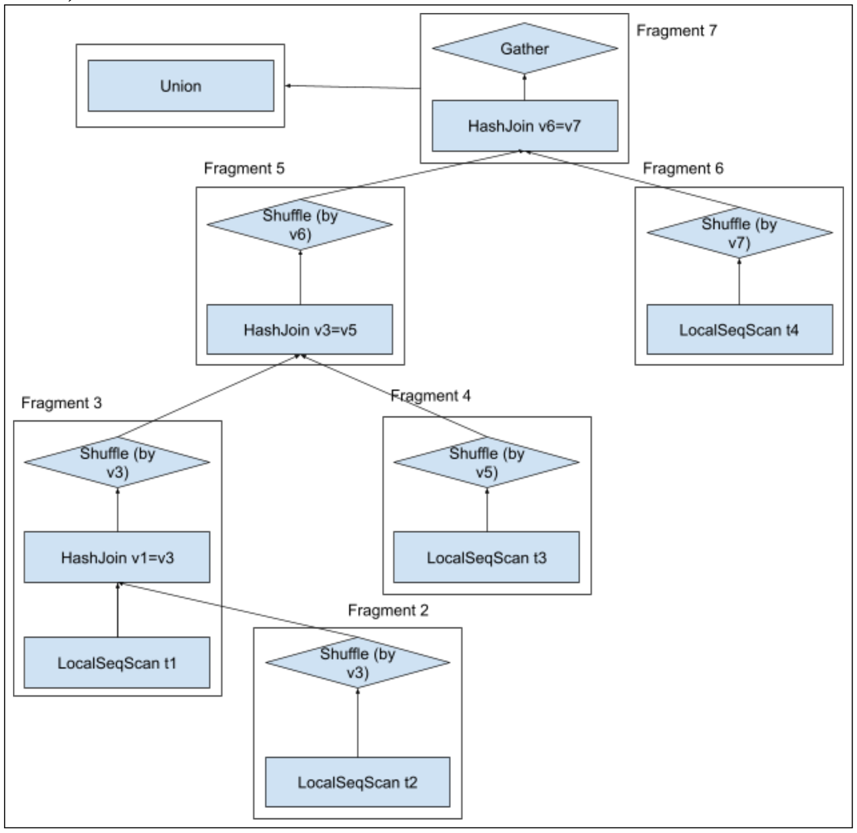
■ **A)**



☐ B)

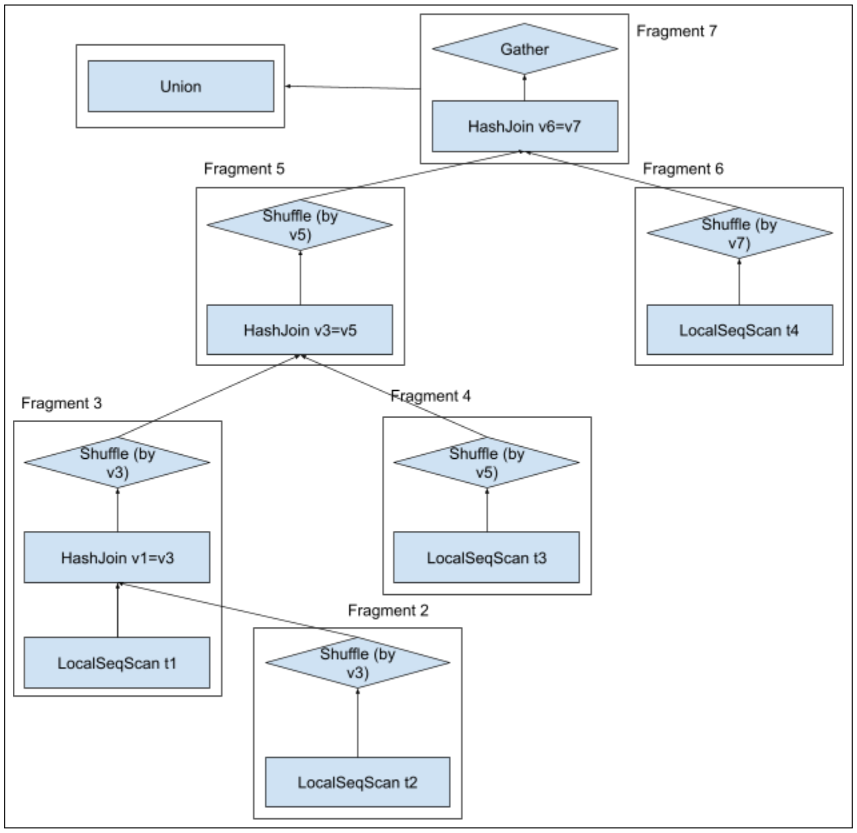


■ **C)**

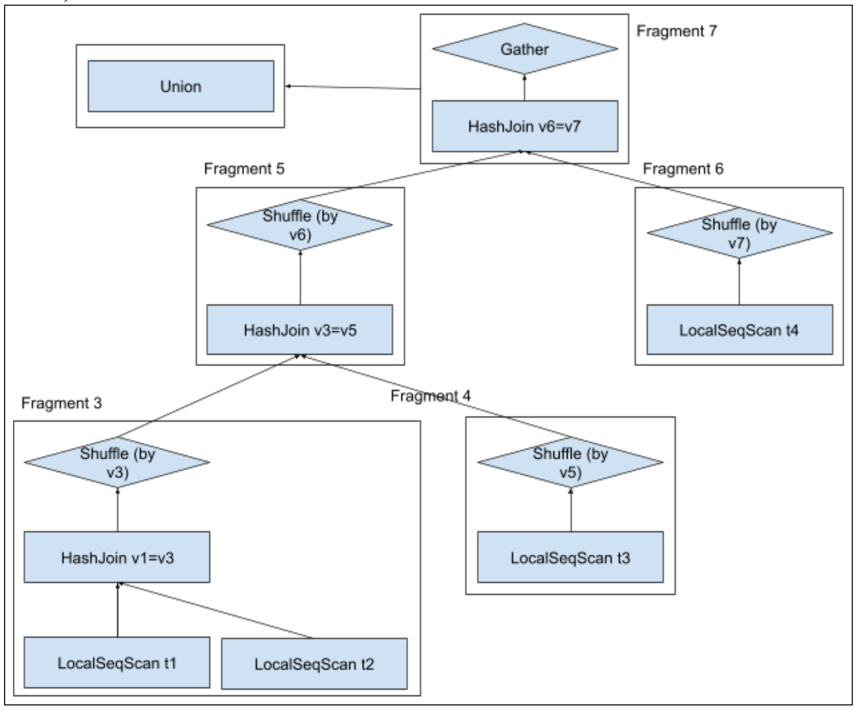


☐ D)

Question 4 continues…


Union
Fragment 7
Gather
HashJoin v6=v7
Fragment 5
Shuffle (by v5)
HashJoin v3=v5
Fragment 6
Shuffle (by v7)
LocalSeqScan t4
Fragment 3
Shuffle (by v3)
HashJoin v1=v3
LocalSeqScan t1
Fragment 4
Shuffle (by v5)
LocalSeqScan t3
Fragment 2
Shuffle (by v3)
LocalSeqScan t2

☐ E)


Union
Fragment 7
Gather
HashJoin v6=v7
Fragment 5
Shuffle (by v6)
HashJoin v3=v5
Fragment 6
Shuffle (by v7)
LocalSeqScan t4
Fragment 3
Shuffle (by v3)
HashJoin v1=v3
LocalSeqScan t1
LocalSeqScan t2
Fragment 4
Shuffle (by v5)
LocalSeqScan t3

Solution: For the query optimizer, its job is to generate a query plan that works given any schedule and any parallelism (i.e., when the range of table data to be joined are distributed on different nodes, when each fragment has different parallelism, etc.) Therefore, the optimizer should ensure within each fragment, the execution engine can access all data they need by inserting correct shuffles.

B and D are obviously incorrect given the left side of HashJoin v6 = v7 should be shuffled by v6, not v5. This ensures that the node executing fragment 7 can access data of the same range from the left join side and the right join side. If we shuffle by v5, it is possible that the row from the left side of the join and the row from the right side of the join cannot be matched using the condition v6 = v7.

E is incorrect because the optimizer cannot put table scan t1 and table scan t2 within the same fragment for a shared-nothing database system. If there is one node which contains t1 of only t1.v1 within 0 - 1000 and t2 of only t2.v3 within 1000 - 2000, the plan cannot be executed. There must be a shuffle on either side of HashJoin v1 = v3.

A and C are correct answers. For A the optimizer adds shuffles and splits as many fragments as possible. For C the optimizer merges table scan t1 and hash join v1 = v3 into a single fragment, which is also correct in any table data distribution – fragment 2 can simply shuffle the data using t1.v1's distribution and send the data to the corresponding row. Within fragment 3 in option C, data are distributed by the original distribution of table t1.

(b) **[5 points]** Assume there are 3 nodes in the system and the data ranges in each node are as follows:

	$t1.v1$	$t2.v3$	$t3.v5$	$t4.v7$
$Node\ 1$	0 - 999	0 - 999	1000 - 1999	0 - 999
$Node\ 2$	1000 - 1999	1000 - 1999	0 - 999	1000 - 1999
$Node\ 3$	2000 - 2999	2000 - 2999	2000 - 2999	2000 - 2999

Table 2: Data distribution for table $t1$ to $t4$

Assume the data are shuffled by range. Which is the best and correct schedule for the query?

☐ A)

```
1. HashJoin v1=v3: 0-999 on node 1, 1000-1999 on node 3,
   2000-2999 on node 2
2. HashJoin v3=v5: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
3. HashJoin v6=v7: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
4. Union: on node 1
```

☐ B)

Question 4 continues. . .

```
1. HashJoin v1=v3: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
2. HashJoin v3=v5: 0-999 on node 3, 1000-1999 on node 2,
   2000-2999 on node 1
3. HashJoin v6=v7: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
4. Union: on node 1, 2, 3
```

■ **C)**

```
1. HashJoin v1=v3: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
2. HashJoin v3=v5: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
3. HashJoin v6=v7: 0-999 on node 1, 1000-1999 on node 2,
   2000-2999 on node 3
4. Union: on node 1
```

Solution: For HashJoin v1=v3, all three nodes can perform joins locally, so we can schedule the query as in option C and won't need to send any data over the network.
For HahsJoin v3=v5, Node 3 can perform join locally, but Node 2 should send v5(0-999) to Node 1 and Node 1 should send v5(1000-1999) to Node 2 before joining locally.
For HashJoin v6=v7, Node 1, 2, 3 should shuffle rows by v6 so that v6(0-999) are on Node 1, v6(1000-1999) are on Node 2, and v6(2000-2999) are on Node 3. Note that v6 is not a partition key and therefore we have to add a shuffle anyways before the left side of the join.
For union, data should be gathered to a single node, and then be served to the client.

(c) **[5 points]** Given the following assumptions:
1. t1 contains 3000 rows and v1 has all values across (0-2999)
2. t2 contains 3000 rows and v3 has all values across (0-2999)
3. t3 contains 3000 rows, v5 has all values across (0-2999) and so as v6
4. t4 contains 300 rows and v7 values are uniformly distributed across 0-2999
5. All joins produce a number of rows equal to *min*{*cardinality of left child, cardinality of right child*}.

Using the data distribution of Table 2, how much data is expected to be transferred over the network when executing the plan with the best schedule in question (b)? (**Hint**: Calculate the answer by summing up all the expected (**rows** * **columns**) that would be shuffled before each `HashJoin` and `Union`.

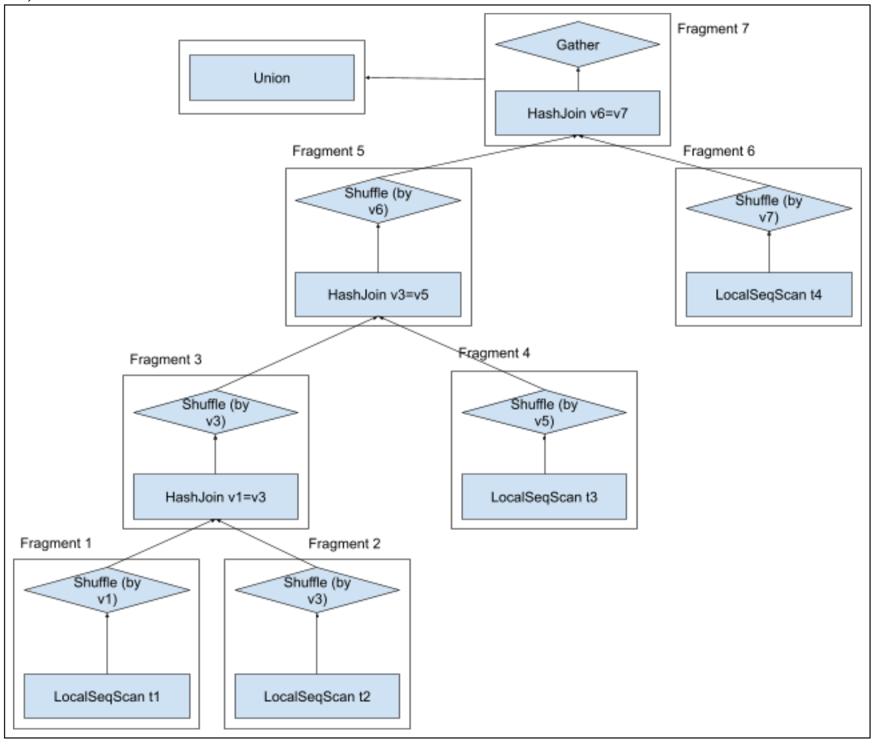
☐ 0 – 5000
☐ 5000 – 10000
■ 15000 – 22000
☐ 22000 – 40000

☐ ≥ 40000

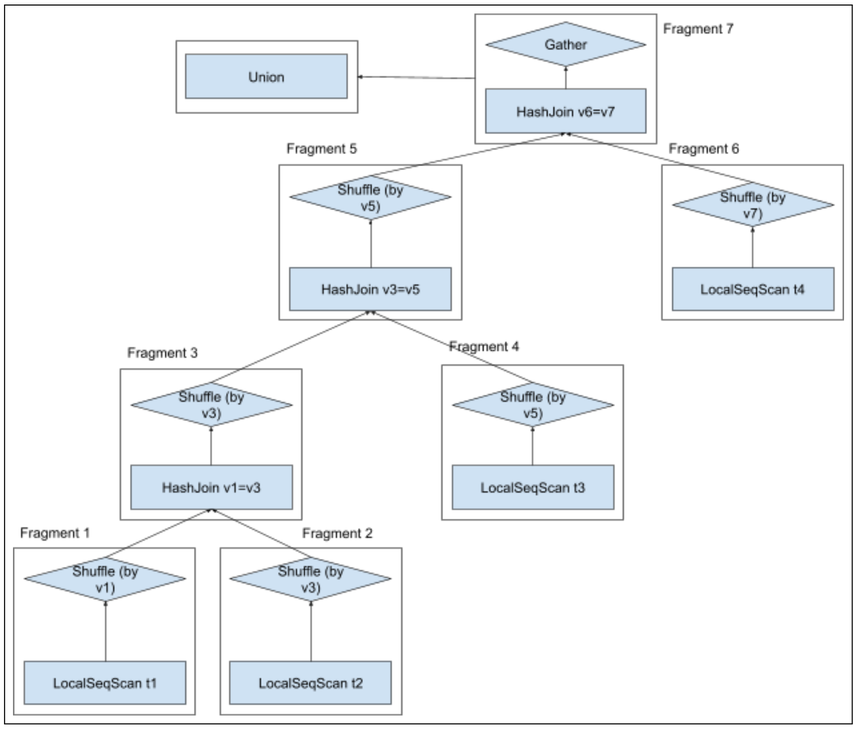
Solution: The right side shuffle of HashJoin v3=v5 requires transferring 2000 rows * 2 columns for t3 (t3.v5 within 1000 - 1999 are sent from node 1 to node 2, and t3.v5 within 0 - 999 are sent from node 2 to node 1). The left side shuffle of HashJoin v6=v7 requires shuffling 3000 rows * 6 columns * (2/3) probability (the v6 data are evenly distributed across 3 nodes, the expected proportion of rows that will be transferred to different nodes is 2/3). The gather requires shuffling 300 rows * 8 columns * (2/3) probability (the gather puts all rows to one node, so only 2/3 of the total rows are transferred to the destination node). So the total number of expected (rows * columns) transferred is 4000 + 12000 + 1600 = 17600.

(d) **[5 points]** An engineer realized that t4 is very small so they configured the table to be stored on all nodes. Which of the following plans is correct?

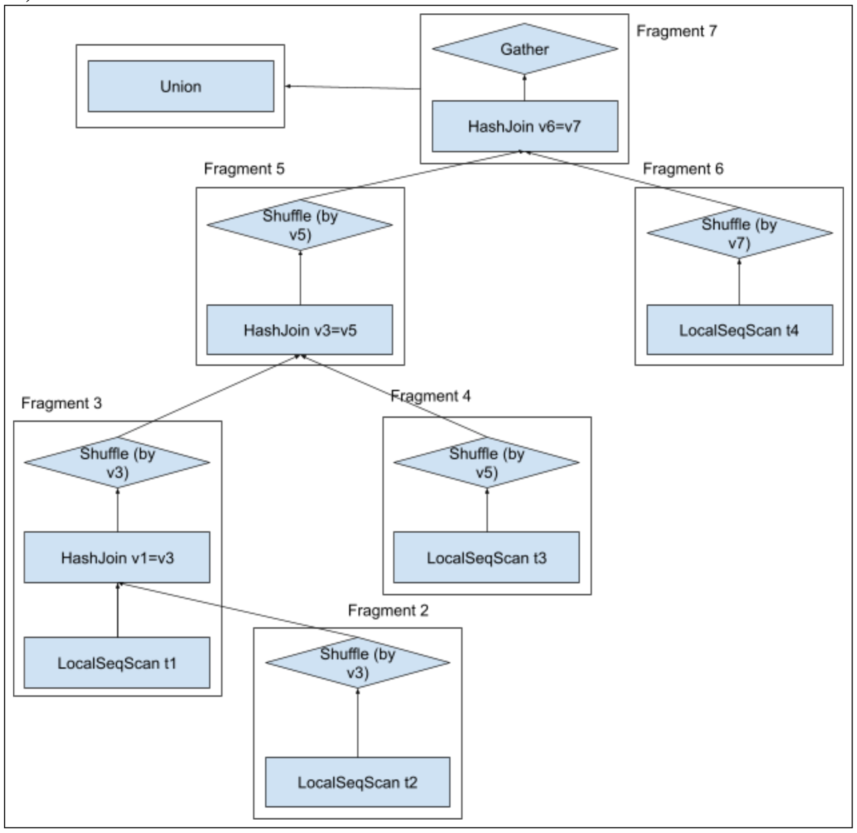
☐ A)



☐ B)

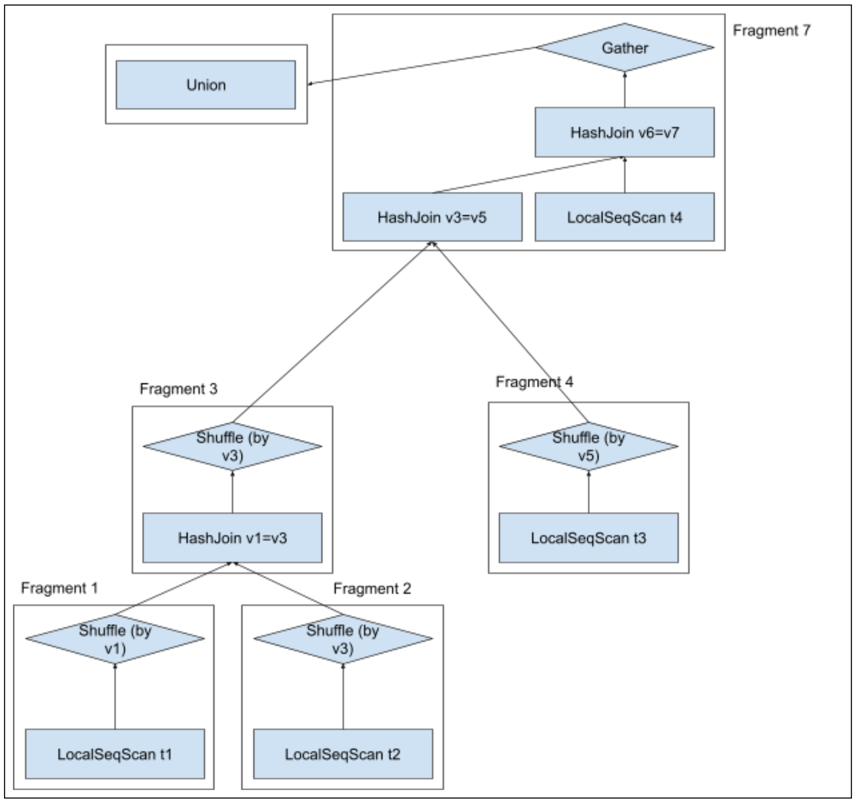


☐ C)



■ **D)**

Question 4 continues. . .



Solution: Now that every node has the full content of t4, we don't need to shuffle t4. Within fragment 7, the data is distributed by the join key v3 / v5. Given we have some data of HashJoin v3=v5 and full data of t4, the execution engine will be able to execute this fragment. Option A, B, C are incorrect because we have full data on every node and we will get a copy of table t4 on every node. This means that we will have 3 copies for each row after shuffling.

There are many ways of making A, B, C correct. For example, if we force table scan t4 fragment to become a "singleton" (i.e., the fragment will be scheduled on a single node instead of all nodes containing a copy of t4), then A will be correct. If we move table scan t4 into HashJoin v6=v7, B, C can also be correct even if the left side is not being shuffled by the join key.

(e) **[4 points]** In the correct case of question (d), how much data is expected to be transferred over the network, given the same assumptions in question (c)? (**Hint**: Calculate the answer by summing up all the expected (**rows** * **columns**) that would be shuffled before each `HashJoin` and `Union`.

- ☐ 0 – 5000
- [✓ 5000 – 10000]
- ☐ 15000 – 22000
- ☐ 22000 – 40000
- ☐ ≥ 40000

Question 4 continues. . .

Solution: The shuffle on the right side of HashJoin v3=v5 requires transferring 2000 rows * 2 columns. The union requires shuffling 300 rows * 8 columns * (2/3) probability. So the total number of expected (rows * columns) transferred is 5600.

(f) **[6 points]** The BusTub* developers decide to replicate data using multiple groups of Multi-Paxos to ensure high availability. Here is the latest setup of the database (Range = the range of the partition key of a table):

Range / Table	$t1$	$t2$	$t3$	$t4$
0 - 999	Paxos Group 1	Paxos Group 2	Paxos Group 3	Paxos Group 10
1000 - 1999	Paxos Group 4	Paxos Group 5	Paxos Group 6	Paxos Group 10
2000 - 2999	Paxos Group 7	Paxos Group 8	Paxos Group 9	Paxos Group 10

Table 3: Paxos Group IDs

Assuming the following status of the database (L = Leader, A = Acceptor):

Node / Paxos Group	1	2	3	4	5	6	7	8	9	10
Node 1	L				A	A	L			
Node 2	A	L				A	A	L		
Node 3	A	A	L				A	A	L	
Node 4		A	A	L				A	A	L
Node 5			A	A	L				A	A
Node 6				A	A	L				A

Table 4: Current Leader / Acceptor nodes of Paxos groups

What's the maximum number of nodes that can go down before this query cannot be successfully executed? (Assume reads / writes all go through the leader).

- ☐ 1
- [✓ **2**]
- ☐ 3
- ☐ 4
- ☐ 5

Solution: Paxos requires at least 2 nodes to achieve consensus in a 3-node setup. If there is only one node left for a single Paxos group, it cannot reach concensus. Note that for queries of pure reads (as in the query in this question) we still need at least 2 nodes for a single Paxos group because if there is only one left, it cannot know whether itself is a real leader (there might be network partition) and might serve stale or wrong data. Therefore, we can for example kick out node 1 and 4 while the query can still be executed.