

6.830 Problem Set 3

Assigned: 11/3

Due: 12/1

The purpose of this problem set is to give you some practice with concepts related to query optimization and concurrency control and recovery.

1 Parallel Query Processing

Recall that the standard way to execute a query in a shared-nothing parallel database is to horizontally partition all tables across all nodes in the database. Recall also that hashing can be used to repartition one or both tables as needed to compute joins in a distributed fashion. Suppose you are trying to compute the following query:

```
SELECT *
FROM R, S
WHERE R.a > v1
AND S.b > v2
AND R.c = S.d
```

You are given the following:

- Both tables are 4,000 MB,
- The disk can read at 50 MB/sec (for the purposes of this problem, you may ignore differences between sequential and random I/O),
- The network on each node can transmit data at 40 MB/sec, regardless of the number or rate at which other nodes are simultaneously transmitting,
- A computer cannot send over the network and read or write from its disk at the same time,
- The selectivity of both selection predicates is 0.1,
- Each tuple in R joins with exactly one tuple in S,
- Each machine in your distributed database has 400 MB of memory.

Question 1: Suppose both tables are stored on a single node. Describe the best query plan for executing this query on that node.

Question 2: Ignoring CPU costs, estimate the time to answer this query on a single node.

Question 3: Now, suppose that the tables are hash-partitioned on R.a and S.b across a 4 node distributed database. Describe the best distributed query plan for executing this query.

Question 4: Ignoring CPU costs, estimate the time to answer this query on the distributed database.

Question 5: Now, suppose that the tables are hash-partitioned on R.c and S.d across a 4 node distributed database. Describe the best distributed query plan for executing this query.

Question 6: Ignoring CPU costs, estimate the time to answer this query on the distributed database.

2 Locking

You have a database of movies and actors, with a `moviesacted` table that represents the many-to-many relationship between movies and actors, as follows:

```
movies : (id int, title char(100), year int)
actors : (id int, name char(100), salary int)
moviesacted : (movieid int, actorid id,)
```

There are 3 clustered B+Trees: one on `movies.id`, one on `actors.id`, and one on `(moviesacted.movieid, moviesacted.actorid)`.

There are 20,000 actors and 10,000 movies, the `bookauths` table has 100,000 rows, and integers occupy 4 bytes. Data pages hold 1,000 bytes, and each index page holds 100 pointers and key values.

The earliest movie in the database is data 1990, and each year has approximately the same number of movies.

For each of the following queries:

- Indicate the query plan that the database would most likely use to answer the query.
- Estimate the number of read (S) and write (X) locks that would be acquired by the execution of your plan assuming the use of page level locking. Assume that locks must be acquired on both index and data pages before they are read or written.
- Estimate the the number of read and write locks that would be required by your plan assuming row locking? Suppose that locks must be acquired for each distinct value read from an index (e.g., if the actor with id “10” is read from the database, a single index lock is acquired.)

Question 7:

- a. `SELECT title FROM movies WHERE year > 2004`
- b. `UPDATE actors`
`SET salary = '$10M'`
`WHERE name = 'Brad Pitt'`

Question 8:

Describe a situation in which switching from page to row locks would likely make processing of some workload 2x as fast. You can use different queries than in the above questions; please explicitly write down the transactions you have in mind. Explain the reasons for the 2x increase in performance.

3 ARIES

Question 9:

Suppose an ARIES-based database system (using page-level logging) runs the following transaction over a single-table database of animals with the schema `<name char(100), species char(100), cageid int>`. Assume that the table is stored in a heap file on disk but that records were inserted in cageid order, such that records in the file are sorted by cageid (but the database system does not maintain this sorted order as additional updates happen.) Each disk page is 2000 bytes, and there are initially 5 animals in cage 1, 5 animals in cage 2, and 6 animals in cage 3 (and no other cages).

```
BEGIN TRANSACTION
SELECT name
  FROM animals
 WHERE animals.cageid > 1
UPDATE animals
  SET cageid = 2
  WHERE animals.cageid = 3
INSERT INTO animals
  VALUES ('billy', 'goat', 1)
COMMIT
```

If this is the only transaction that has ever run on the system, what will the contents of the log look like after this transaction completes? Show the log records, indicating their type, the transaction they apply to, and any additional data needed to describe the state of the records. You don't need to write out the contents of the before and after images in detail, but you should describe what they consist of.

The ARIES paper mentions that CLR's allow it to avoid performing an UNDO more than once if there is a crash during recovery: "... ARIES will rollback only those actions that had not already been undone. This is possible since history is repeated for such transactions and since the last CLR written for each transaction points (directly or indirectly) to the next non-CLR record that is to be undone." (p. 112).

Question 10: You might think that it would be enough to have each CLR record just point to the next record that is to be undone – that is, to omit the redo information in the PageID and Data fields of the log record (p. 113), so that a CLR contained only the fields LSN, Type, TransID, and UndoNxtLSN. Explain why this would be a bad idea.

Question 11:

Suppose an ARIES-based database crashes. When it comes back online, you find the log and disk pages look as follows (here, the notation UPDATE T1, X means that transaction T1 updated page X.) Assume that update data pages are only flushed to disk when a CHECKPOINT is taken.

```
BEGIN T1
BEGIN T3
UPDATE T1, X
UPDATE T3, Z
CHECKPOINT
COMMIT T1
BEGIN T2
UPDATE T2, X
UPDATE T3, Y
COMMIT T3
BEGIN T4
CHECKPOINT
UPDATE T4, Y
*crash*
```

- a. What transactions were running at the time of the crash?
- b. What transactions will the system UNDO?
- c. What data pages will the system modify during the REDO pass of recovery?
- d. What data pages will the system modify during the UNDO pass of recovery?

4 Two phase commit

One limitation of two-phase commit is that the subordinate nodes are required to wait for the coordinator to determine the outcome of the transaction once they have entered the PREPARED state. This means that if the coordinator crashes after sending out PREPARE messages, workers may need to remember information about the transaction for a long time (until the coordinator comes back up.)

Dana Bass observes that if the coordinator sent out PREPARE messages, it has almost certainly completed processing its portion of the query. This means that in the event of a coordinator failure, it should, in theory, be possible for one of the subordinates to assume the role of coordinator and collect the votes of the subordinates itself, with the original coordinator learning the outcome of the transaction when it comes back up.

Ben Bitdiddle decides to implement Dana's approach. He modifies the code that runs on subordinates so that when a node N does not receive an outcome message (telling it whether to COMMIT or ABORT) from the coordinator, and is unable to contact the coordinator to determine the outcome, it sets a timer. When this timer expires, N sends a message to all the other nodes telling them it is taking over as coordinator (in the event that two nodes send out such a message simultaneously, the node with the lower id becomes the coordinator.) N then begins the standard two-phase commit protocol, sending PREPARE messages, receiving votes, and sending outcomes.

Question 12: Dana points out that the original coordinator C may still have been crashed at the time N took over as the coordinator. She points out that when C comes back from the crash, it will abort the transaction (because it didn't log a COMMIT record for it, and it was the coordinator.) How should Ben modify his new commit protocol to ensure that it operates correctly when C comes up? Your solution should also handle the case of multiple coordinator failures (e.g., that N crashes too!)

Question 13: Ben's protocol still requires N to wait for C to come back up before it can forget about the state of the transaction (since it has to receive an ACK from C after it sends the outcome record.) Why is it any better than the basic two phase commit protocol discussed in class? (Hint: Think about the state that nodes have to maintain in the two versions of the protocol.)

5 Column Stores

After taking 6.830, Ben BitDiddle reads the C-Store paper and decides to implement a column-oriented database of his own. He has a typical star schema with fact table, $\text{FACT}(A_1, \dots, A_n)$ and runs a collection of queries, parameterized by $L \leq n$:

```
SELECT  $A_1, \dots, A_L$ 
FROM FACT
```

He bases his implementation on SimpleDB, but treats each column as a separate table. Columns are stored on disk in the same order (so the first value of column 1 corresponds to the same tuple as the first value in columns 2... n .) To perform the above query, he uses $L - 1$ "merge" operators. A merge operator simply loops over its inputs, repeatedly reading one tuple from each input, concatenating these tuples together, and outputting wider tuples (like a merge join, but without the sort phase). His implementation of the merge operator reads a page into memory and iterates through the tuples in the page before moving on to the next page. He runs these operators using the conventional tuple-at-a-time iterator model we studied in class. Ben's database is configured to use 4 KB pages. Disk seeks take 10 ms, and the disk can sequentially read at 50 MBytes/sec.

Question 14: Contrary to his expectation, Ben finds that when he runs queries with $L > 1$, the performance of his system is worse than in a standard row-store running the same query. Explain why this is likely the case, and propose a simple solution to the problem.