CS W186 Databases - Fall 2019
Guerilla Section 4: Parallel Query Processing; Transactions and Concurrency

Sunday, November 3, 2019

1 Types of Parallelism

For each of the following scenarios, state whether it is an example of:

- Inter-query parallelism
- Intra-query, inter-operator parallelism
- Intra-query, intra-operator parallelism
- No parallelism

1. A query with a selection, followed by a projection, followed by a join, runs on a single machine with one thread.

   Answer: No parallelism. It might look like a pipeline, but at any given point in time there is only one thing happening, since there is only one thread.

2. Same as before, but there is a second machine and a second query, running independently of the first machine and the first query.

   Answer: Inter-query parallelism.

3. A query with a selection, followed by a projection, runs on a single machine with multiple threads; one thread is given to the selection and one thread is given to the projection.

   Answer: Intra-query, inter-operator parallelism.

4. We have a single machine, and it runs recursive hash partitioning (for external hashing) with one thread.

   Answer: No parallelism, because there is only one machine and one thread. Don't confuse this with parallel hashing!

5. We have a multi-machine database, and we are running a join over it. For the join, we are running parallel sort-merge join.

   Answer: Intra-query, intra-operator parallelism. We have a single query and a single operator, but that single operator is going to do multiple things at the same time (across different machines).

2 Partitioning for Parallelism

1. Suppose we have a table of size 50,000 KB, and our database has 10 machines. Each machine has 100 pages of buffer, and a page is 4 KB.

   We would like to perform parallel sorting on this table, so first, we perfectly range partition the data. Then on each machine, we run standard external sorting.

   How many passes does this external sort on each machine take?

   Answer: **2 passes**.
   After range partitioning, each table will have 5,000 KB of data, or 1,250 pages. With 100 pages of buffer, this will take 2 passes to sort.

2. Suppose we were doing parallel hash join. The first step is to partition the data across the machines, and we usually use hash partitioning to do this.

   Would range partitioning also work? What about round-robin partitioning?

   Answer: Range partitioning also works, because items with the same key still end up on the same machine as required. Round-robin partitioning does not do that, so it does not work.

3. Suppose we have a table of 1200 rows, perfectly range-partitioned across 3 machines in order.

   We just bought a 4th machine for our database, and we want to run parallel sorting using all 4 machines.

   The first step in parallel sorting is to repartition the data across all 4 machines, using range partitioning. (The new machine will get the last range.)

   For each of the first 3 machines, how many rows will it send across the network during the repartitioning? (You can assume the new ranges are also perfectly uniform.)

   Answer: **100 rows, 200 rows, and 300 rows.**
   The original partitions were 3 ranges of 400 rows each; the new 4 ranges will have 300 rows each.
   The first machine held the first 400 rows originally, and now only needs to hold the first 300. It will send the remaining 100 rows over the network (to machine 2).
   The second machine held rows 501-800 initially, but now needs to hold rows 401-600. It will send rows 601-800 (200 rows) to machine 3.
   The third machine held rows 801-1200 initially, and similarly needs to send rows 901-1200 (300 rows) to machine 4.

3 Transactions and Concurrency

In this question, we will explore the key topics of transactions and concurrency: serializability, types of locks, two-phase locking, and deadlocks.

We will do this by actually running multiple transactions at the same time on a database, and seeing what happens. You may find it helpful (but not necessary) to draw some graphs:

- For the lock type questions, you may wish to draw a graph representing the whole database and which resources are being locked.
- For serializability questions, you may wish to draw a graph with a node for each transaction, and arrows if there are *conflicts* between transactions.
- For deadlock questions, you may wish to draw a graph with a node for each transaction, and arrows if a transaction is *waiting* for a lock held by another transaction.

We will use a database with tables A, B, C, ... and table A holds rows A1, A2, A3, ... and so on.

Consider the following sequence of operations:

```
Txn 1:  IX-Lock(Database)      (1)
Txn 1:  IX-Lock(Table A)       (2)
Txn 1:  X-Lock(Row A1)         (3)
Txn 1:  Write(Row A1)          (4)
Txn 1:  Unlock(Row A1)         (5)
Txn 1:  S-Lock(Row A2)         (6)
Txn 2:  IX-Lock(Database)      (7)
Txn 2:  IX-Lock(Table A)       (8)
Txn 2:  X-Lock(Row A1)         (9)
Txn 2:  Write(Row A1)          (10)
Txn 2:  S-Lock(Row A2)         (11)
Txn 2:  Read(Row A2)           (12)
```

1. Is Transaction 1 doing Two-Phase Locking so far?

   Answer: No; we have a lock (6) after unlock (5).

2. Is Transaction 1 doing Strict Two-Phase Locking?

   Answer: No; it's not even Two-Phase.

3. Is this schedule conflict-serializable so far? If not, what is the cycle?

   Answer: Yes, it is conflict-serializable; there is only one conflict ($4 \rightarrow 10$), so it is serializable to a serial order of Txn 1 $\rightarrow$ Txn 2.

4. Is this schedule serial so far?

   Answer: Yes! Since all of Txn 1's operations are before Txn 2's operations, it is actually already a serial order.

Continuing with all operations so far:

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B2)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  S-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

5. Is Transaction 2 doing Two-Phase Locking so far?

   Answer: Yes. All locks (< 17) are before unlocks (17, 18).

6. Is Transaction 2 doing Strict Two-Phase Locking?

   Answer: No! For strict two-phase, we must **commit** the transaction **before doing any unlocks**.

7. Is this schedule conflict-serializable so far? If not, what is the cycle?

   Answer: No, not anymore. We have T1 $\rightarrow$ T2 from conflict 4 $\rightarrow$ 10, and T2 $\rightarrow$ T1 from conflict 15 $\rightarrow$ 20. The cycle is T1 $\leftrightarrow$ T2.

8. Suppose we start a new transaction, Transaction 3. What kind of locks can Transaction 3 acquire on the whole database?

   The database currently has two IX locks held by Txn 1 and Txn 2. Looking at the compatibility matrix, we see that **IS and IX** are the locks compatible with IX locks, so these are the locks that Txn 3 can acquire on the database.

9. Given the above answers, what kind of locks can Transaction 3 acquire on Table A?

   On the parent of Table A (the whole database), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.
   Which of these can we actually acquire? Table A currently has two IX locks held by Txn 1 and Txn 2, so the compatibility matrix says **IS and IX** locks only.

10. Given the above answers, what kind of locks can Transaction 3 acquire on Row A3?

    On the parent of Row A3 (Table A), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.
    Which of these can we actually acquire? Row A3 has no locks currently, so we can acquire any of these! But this is a leaf node and I locks are for intermediate nodes only, so in practice we can only acquire **S and X locks**.

11. Given the above answers, what kind of locks can Transaction 3 acquire on Table B?

    On the parent of Table B (the whole database), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.
    Which of these can we actually acquire? Table B currently has an SIX lock, so the compatibility matrix says **IS locks** only.

12. Given the above answers, what kind of locks can Transaction 3 acquire on Row B2?

    On the parent of Row B2 (Table B), we know we can acquire IS locks (from the previous question). These locks allow us to acquire S and IS locks below it.

Which of these can we actually acquire? Row B2 has no locks currently, so we can acquire any of these! But this is a leaf node and I locks are for intermediate nodes only, so in practice we can only acquire the **S lock**.

13. What kind of locks can Transaction 1 acquire on Row B2?

    Transaction 1 currently holds an SIX lock on the parent of Row B2 (Table B), which allows it to acquire X and IX locks below it.
    Which of these can it actually acquire? Row B2 has no locks, so it can acquire either of them; however, I locks are for intermediate nodes only, so it can just acquire the **X lock**.

```
Txn 2:  IX-Lock(Table B)        (21)
Txn 3:  IX-Lock(Database)       (22)
Txn 3:  X-Lock(Table C)         (23)
Txn 3:  IX-Lock(Table A)        (24)
Txn 3:  X-Lock(Row A1)          (25)
Txn 1:  IX-Lock(Table C)        (26)
```

14. We have now entered a deadlock. What is the waits-for cycle between the transactions?

    T1 (26) waits on T3 (23).
    T3 (25) waits on T2 (9).
    T2 (21) waits on T1 (18).

15. We can end this deadlock by aborting the youngest transaction. Which transaction do we abort?

    The youngest transaction is **T3**, so we abort that one.

    Alternatively, we could have avoided this deadlock in the first place by using *wound-wait* or *wait-die*. Recall from lecture that these methods cause transactions to sometimes abort, according to a priority order, when they try to acquire locks.

    Let's say that the priority of the transaction is its number (Txn 1 is highest priority).

16. If we were using *wound-wait*, what is the **first operation** in this sequence that would cause a transaction to get aborted, and which transaction gets aborted?

    In *wound-wait*, the only waiting that happens is lower priority waiting for higher priority. If a higher priority transaction tries to wait, it will just abort ("wound") the transaction it is waiting on.
    At (21), T2 can wait on T1. At (25), T3 can wait on T2. **But at (26), T1 will not wait on T3; it will instead abort T3.**

17. If we were using *wait-die*, what is the **first operation** in this sequence that would cause a transaction to get aborted, and which transaction gets aborted?

    In *wait-die*, the only waiting that happens is higher priority waiting for lower priority. If a lower priority transaction tries to wait, it will just abort itself ("die") instead.
    At (21), T2 will not wait on T1, since T2 is lower priority. Thus, T2 will abort.