Demystifying Concurrency Control in DBMS

Gajendra Singh
Director - Academics, Sikkim Manipal University, Ghana Learning Center,
KnowledgeWorkz Ltd, Accra, Ghana
gajendra.singh@smughana.com

ABSTRACT: ***In today's era, sharing of information is common everywhere in everyday life. Customers share bank credit cards, students share books, teachers share knowledge. The list is endless. Despite the fact that we have learned how to share essential common resources, sharing is not easy because of the delay that it takes for us to acquire resources and get our tasks done in due time. When resources are lavish, delays are low, and sharing is relatively easy because we need to wait less. When resources are rare, delays are high, and sharing is much harder because we wait forever to use them. Adding more helpdesk counters reduces the delay of information to be given to the customers. However, when several tasks try to use the same resource or when tasks try to share information, it can lead to confusion and inconsistency. The task of concurrent computing is to solve that problem. This paper describes three key components of a high performance concurrent parallel database management system. First, Parallel Computing strategies that distribute the workload of a table across the available nodes while minimizing the overhead of concurrency. Second, Concurrency Control Locking Strategies. Third, Two Phase Locking Protocol that lock every item you touch, once you release your first lock, you can't acquire any more locks.***

KEYWORDS: Concurrency Control, Data Sharing, Locking

I. INTRODUCTION

To control the inconsistency and confusion of sharing information, we need to use some mechanism. One of the concepts is Concurrency control that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. This Technique, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity. [3] The Concurrency is about to control the multi-user access to the database. When many users try to access the same resource at the same time, concurrency control is required. To explain the concept of concurrency control, consider two travelers who go to the railway reservation counter at different places but at the same time to purchase a train ticket to the same destination on the same train. There's only one seat left in the coach, but without concurrency control, it's possible that both travelers will end up purchasing a ticket for that one seat. However, with concurrency control, the database wouldn't allow this to happen. Both travelers would still be able to access the train seating database, but concurrency control would preserve data accuracy and allow only one traveler to purchase the seat.

This example also demonstrates the importance of addressing this issue in a multi-user database. Obviously, one could quickly run into problems with the inaccurate data that can result inconsistency and inaccuracy from several transactions occurring simultaneously and writing over each other. The following section provides parallel computing strategies for implementing concurrency control.

II. PARALLEL COMPUTING

In parallel computing , database applications apply the concept of horizontal partitioning to allocate the tuples of each relation across multiple disk drives. The strategy used for partitioning a relation is independent of the storage structure used at each site. The database administrator (DBA) for such a system must consider a variety of alternative options for each relation. [4] Figure 1 as given below explains the strategies for Parallelization.

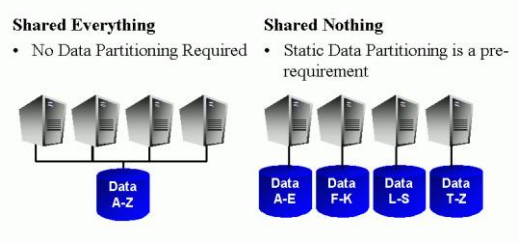

Figure 1: Design strategies for parallelization Non Shared versus Shared architecture. [4]

A. A DESIGN STRATEGIES FOR PARALLELIZATION – STATIC VERSUS DYNAMIC

One of the applications of concurrency is parallel computing. Without concurrency, we cannot image the concept of parallelism. And to execute a parallel program in parallel, you must have hardware with multiple processing elements so concurrent tasks execute in parallel.

For example, if a busy retail shop like Melcom group in Ghana has got only a single cash counter, the customers will form a single queue, and wait for their turn. If there are two cash counters, the task can be effectively split. The customers will form two queues and will be served twice as fast. This is an instance in which parallel processing is an effective solution. We can solve easily solve the critical problems with help of Parallel computing.

Another example is ray tracing which is a common approach for execution of images. The problem naturally contains a great deal of concurrency since in principle, each ray of light can be handled as an independent task. We can queue these tasks up for execution and use a pool of threads to run them in parallel on a parallel computer. In other words, we exploit the concurrency in ray tracing to create a parallel program to render a fixed sized image in less time. Parallelism is the idea of breaking down a single task into multiple smaller, distinct parts. Instead of one process doing all the work, the task can then be parallelized, having multiple processes working concurrently on the smaller units. This leads to tremendous performance improvements and optimal system utilization. The most critical part, [5] however, is to make the decision how to divide the original single task into smaller units of work. Traditionally, two approaches have been used for the implementation of parallel execution of database systems. The main difference is whether or not the physical data layout is used as a base – and static pre-requisite – for dividing, thus parallelizing, the work.

- **Static parallelism through physical data partitioning – non shared**

In non-shared database architectures, database files have to be partitioned on the nodes of a multi-computer system to enable parallel processing. Each node 'owns' a subset of the data and all access to this data is performed exclusively by the owning node, using a single process or thread with no provision for intra-partition parallelism (Instead of referring to a 'node' you can also find terms like 'virtual processors', a mechanism to emulate a non-shared node on a symmetric multiprocessor (SMP) machine; for simplicity reasons, we will refer to a node when discussing shared nothing architectures). In other words, a pure shared nothing system uses a partitioned or restricted access approach to divide the work among multiple processing nodes.

Data ownership by node changes relatively infrequently - database reorganization to address changing business needs, adding or removing nodes, and node failure are the typical reasons for change in ownership and always imply manual administration effort. Conceptually, it is useful to think of a non-shared system as being very similar to a distributed database. A transaction executing on a given node has to send messages to other nodes that own the data being accessed and coordinate the work done on the other nodes, to perform the required read/write activity. Message passing to other nodes, requesting to execute a specific operation (function) on their own data sets is commonly known as function shipping. On the other hand, if simply data is requested from a remote node, the complete data set must be accessed and shipped from the owning node to the requesting nodes (data shipping). This approach has some basic disadvantages and is not capable to address the scalability and high availability requirements of today's high-end environments:

• First, the non shared approach is not optimal for use on the shared SMP hardware. The requirement to physically partition data in order to derive the benefits of parallelism is clearly an artificial and outdated requirement on a shared everything SMP system, where every processor has direct, equal access to all the data.

• Second, the rigid partitioning-based parallel execution strategy employed in the shared nothing approach often leads to skewed resource utilization, e.g. when it is not necessary to access all partitions of a table, or when larger non-partitioned tables, owned by a single node, are part of an operation. In such situations, the tight ownership model that prevents intra-partition parallel execution fails to utilize all available processing power, delivering sub-optimal use of available processing power.

• Third, due to the fact of having a physical data partition to node relationship, shared nothing systems are not flexible at all to adapt to changing business requirements. When the business grows, you cannot easily enlarge your system incrementally to address

your growing business needs. You can upgrade all existing nodes, keeping them symmetrical and avoiding data re-partitioning. In most cases upgrading all nodes is too expensive; you have to add new nodes and to reorganize – to physically repartition – the existing database. Having no need for reorganization is always better than the most sophisticated reorganization facility.

• Finally, non shared systems, due to their use of a rigid restricted access scheme, fail to fully exploit the potential for high fault tolerance available in clustered systems. Undoubtedly, massively parallel execution based on a non shared architecture with the static data distribution can parallelize and scale under laboratory conditions. However, the above-mentioned deficiencies have to be addressed appropriately in every real-life environment to satisfy today's high end mission-critical requirements. A review and a more detailed discussion about the fundamental differences of the various cluster architectures and the disadvantages of shared nothing systems can be found in several Real Application Clusters (RAC) related white papers. [6]

- **Dynamic parallelism at execution time - shared**

To explain the concept of Dynamic Parallelism let us have let us focus on Oracle's dynamic parallel execution framework where all data is shared, and the decision for parallelization and dividing the work into smaller units is not restricted to any predetermined static data distribution done at database setup (creation) time. Every query has its own characteristics of accessing, joining, and processing different portions of data. Consequently, each SQL statement undergoes an optimization and parallelization process when it is parsed. When the data changes, if a more optimal execution or parallelization plan becomes available, or you simply add a new node to the system, Oracle can automatically adjust to the new situation. This provides the highest degree of flexibility for parallelizing any kind of operation:

• The physical data sub-setting for parallel access is dynamically raised for each query's requirement before the statement is executed.

• The degree of parallelism is optimized for every query. Unlike in a shared nothing environment, there's no necessary minimal degree of parallelism to invoke all nodes to access all data – the fundamental requirement to reach all of the data Operations can run in parallel, using one, some, or all nodes of a Real Application Clusters, depending on the current workload, the characteristics, and the importance of the query. As soon as the statement is optimized and parallelized, all subsequent parallel subtasks are known. The original process becomes the query coordinator; parallel execution servers (PX servers) are assigned from the common pool of parallel execution servers on one or more nodes and start working in parallel on the operation. Like in a shared nothing architecture, each parallel execution server in a shared everything architecture works independently on its personal subset of data.

Data or functions are sent between the parallel processes similar – or even identical – to the above discussed shared nothing architecture. When the parallel plan of a request is determined, every parallel execution server knows its data set and tasks, and the inter-process communication is as minimal as in a shared nothing environment. However, unlike the shared nothing architecture, each SQL statement executed in parallel is optimized without the need to take any physical database layout restrictions into account. This enables the most optimal data sub setting for each parallel execution, thus providing equal and in most cases even better scalability and performance than pure shared nothing architectures. Subsequent steps of a parallel operation are combined and processed by one Parallel Execution server whenever beneficial, reducing the necessity of function and/or data shipping even more.

III. CONCURRENCY CONTROL LOCKING STRATEGIES

There are two main concurrency control locking strategies which we can define as given below.

- **Pessimistic Locking:** In the concept of concurrency control strategy, an entity is locked till the time it exists in the database's memory. This bounds or prevents users from altering the data entity that is locked. There are two types of locks that fall under the category of pessimistic locking: write lock and read lock. [8]

With the write lock, everyone but the holder of the lock is prevented from reading, updating, or deleting the entity. With read lock, other users can read the entity, but no one except for the lock holder can update or delete it. [9]

- **Optimistic Locking**: In this locking, instances of simultaneous transactions, or collisions, are expected

to be infrequent. In distinction with pessimistic locking, optimistic locking doesn't try to prevent the collisions from occurring. Instead, it aims to detect these collisions and resolve them on the chance occasions when they occur.

Pessimistic locking be responsible for a guarantee that database changes are made safely. However, it becomes less possible as the number of simultaneous users or the number of entities involved in a transaction increase because the prospective for having to wait for a lock to release will increase.

Optimistic locking can improve the problem of waiting for locks to release, but then users have the potential to experience collisions when attempting to update the database.

A. LOCK BASED PROTOCOL

A technique that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose is called Lock. There are two types of operations, i.e. read and write, whose basic nature is different, the locks for read and write operation may behave differently.

There challenges are less when a read operation is performed by different transactions on the same data item. The value of the data item, if constant, can be read by any number of transactions at any given time.

While the write operation is rather different. When a transaction writes some value into a data item, the content of that data item remains in an inconsistent state, starting from the moment when the writing operation begins up to the moment the writing operation is over. If any other trancaction is allowed to read/write the value of the data item during the write operation, those transactions will read an inconsistent value or overwrite the value being written by the first transaction. In both the cases irregularities will creep into the database.

The Locking can be derived from simple rule here. If a transaction is reading the content of a sharable data item, then any number of other processes can be allowed to read the content of the same data item. But if a transaction is written into a sharable data item, then no other transaction will be allowed to read or write that same data item.

Based upon the rules we can classify the locks into two types.

- **Shared Lock:** Shared lock can be applied on a data item in order to read its content. The lock is shared means that any other transaction can acquire the shared lock on that same data item for reading purpose.

- **Exclusive Lock:** An Exclusive lock can be applied on a data item in order to both read/write into it. The lock is exclusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item.

The association between Shared and Exclusive Lock can be represented by the following table 1, which is known as **Lock Matrix**.

	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

Table 1. Lock Matrix

B. THE USE OF LOCKS

If in a transaction, a data item which we want to read/write should first be locked before the read/write is done. After the operation is over, the transaction should then unlock the data item so that other transaction can lock that same data item for their respective usage. Let us take an example to see that a transaction to deposit Ghana Cedi 100/- from account A to account B. The transaction should now be written as the following:

Lock-X (A); (Exclusive Lock, to both read A's value and modify it)
Read A;
A = A – 100;
Write A;
Unlock (A); (Unlocking A after the modification is done)
Lock-X (B); (Exclusive Lock, we want to both read B's value and modify it)
Read B;
B = B + 100;
Write B;
Unlock (B); (Unlocking B after the modification is done)

Any transaction that deposits 20% amount of account A to account C should now be written as given below:

Lock-S (A); (Shared Lock, we only want to read A's value)
Read A;
Temp = A * 0.2;
Unlock (A); (Unlocking A)
Lock-X (C); (Exclusive Lock, we want to both read C's value and modify it)
Read C;
C = C + Temp;
Write C;
Unlock (C); (Unlocking C after the modification is done)

Now it is clear how these locking mechanisms help us to create error free schedules. Lets see this example of an erroneous schedule:

T1	T2
Read A;	
A = A - 100;	
	Read A;
	Temp = A * 0.2;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

Based on common sense only, It has been detected that the Context Switching is being performed before the new value has been updated in A. T2 reads the old value of A, and thus deposits a wrong amount in C. If we had used the locking mechanism, this error could never have occurred. Now if we rewrite the schedule using the locks.

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
	Lock-S (A)
	Read A;
	Temp = A * 0.2;
	Unlock (A)
	Lock-X (C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Write A;	
Unlock (A)	
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	
Unlock (B)	

The Schedule cannot be prepared like the above even if we like, provided that we use the locks in the transactions. See the first statement in T2 that attempts to acquire a lock on A. This is not possible because T1 has not released the exclusive lock on A, and T2 just cannot get the shared lock it wants on A. It must wait until the exclusive lock on A is released by T1, and can begin its execution only after that.

So the proper schedule would look as given below:

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
Unlock (A)	
	Lock-S (A)
	Read A;
	Temp = A * 0.2;
	Unlock (A)
	Lock-X (C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	
Unlock (B)	

Finally this automatically becomes a very correct schedule. We need not apply any manual effort to detect or correct the errors that may crawl into the schedule if locks are not used in them.

IV. TWO PHASE LOCKING PROTOCOL

We can create any concurrent schedule by using locks. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks. [1] The Two Phase Locking Protocol assumes that a transaction can only be in one of two phases.

- **Growing Phase:** In this phase the transaction is able only acquire locks, but unable to release any lock. The transaction enters the growing phase as soon as it acquires the first lock it wants. From now on it has no option but to keep acquiring all the locks it would need. It cannot release any lock at this phase even if it has finished working with a locked data item. Ultimately the transaction reaches a point where all the lock it may need has been acquired. This point is called Lock Point.

- **Shrinking Phase:** Once the Lock Point has been reached, the transaction enters the shrinking phase. In this phase the transaction can only release locks, but cannot acquire any new lock. [8] The transaction enters the shrinking phase as soon as it releases the first lock after crossing the Lock Point. From now on it has no option but to keep releasing all the acquired locks. There are two different versions of the Two Phase Locking Protocol. One is called the Strict Two Phase Locking Protocol and the other one is called the Rigorous Two Phase Locking Protocol.

- **Strict Two Phase Locking Protocol** : According to this protocol, a transaction can release all the shared locks after the Lock Point has been reached, but it cannot release any of the exclusive locks until the transaction commits. This protocol helps in creating cascade less schedule. [7]

While creating a concurrent schedule there is typical problem called Cascading Scheeule. Let us Consider the following schedule once again.

T1	T2
Lock-X (A)	
Read A;	
A = A - 100;	
Write A;	
Unlock (A)	
	Lock-S (A)
	Read A;
	Temp = A * 0.2;
	Unlock (A)
	Lock-X (C)
	Read C;
	C = C + Temp;
	Write C;
	Unlock (C)
Lock-X (B)	
Read B;	
B = B + 100;	
Write B;	
Unlock (B)	

This schedule is hypothetically correct, but a very strange kind of problem may arise here. T1 releases the exclusive lock on A, and immediately after that the Context Switch is made. T2 acquires a shared lock on A to read its value, perform a calculation, update the content of account C and then issue COMMIT. Though, T1 is not finished yet. What if the remaining portion of T1 encounters a problem (power failure, disc failure etc.) and cannot be committed? In this case T1 should be rolled back and the old value of A should be restored. In such a case T2, which has read the updated (but not committed) value of A and calculated the value of C based on this value, must also have to be rolled back. We have to roll back T2 for no fault of T2 itself, but because we proceeded with T2 depending on a value which has not yet been committed. This phenomenon of rolling back a child transaction if the parent transaction is rolled back is called Cascading Rollback, which causes a notable loss of processing power and execution time.

While Using Strict Two Phase Locking Protocols, Cascading Rollback can be prevented. In Strict Two Phase Locking Protocols a transaction cannot release any of its acquired exclusive locks until the transaction commits. In such a case, T1 would not release the exclusive lock on A until it finally commits, which makes it impossible for T2 to acquire

the shared lock on A at a time when A's value has not been committed. This makes it impossible for a schedule to be cascaded. [10]

- **Rigorous Two Phase Locking Protocol**

In this type of Protocol, a transaction is not allowed to release either shared or exclusive locks until it commits i.e. until the transaction commits, other transaction might acquire a shared lock on a data item on which the uncommitted transaction has a shared lock; but cannot acquire any lock on a data item on which the uncommitted transaction has an exclusive lock. [11]

V. CONCLUSIONS AND FUTURE RESEARCH DIRECTIOINS

This Paper provides the conclusion of techniques to realize a better understanding of Concurrent, parallel, scalable, high performance database management system. We described the design strategies for parallelization – static verses dynamic to distribute the workload of a query across multiple nodes. The physical design of concurrent database management systems is an active area of research. From the above we can find the below points regarding concurrency control.

- While implementing the lock and unlock commands, we must ensure that these are atomic operations. An operating system synchronization mechanism should be used to ensure the atomicity of these operations when several instances of the lock manager can execute concurrently.
- We can prevent deadlock by giving priority to each transaction and confirming that lower priority transactions are not allowed to wait or higher priority truncations (or vice versa).
- We have found that the Oracle server uses a multiversion concurrency control scheme in which readers never wait; in fact, readers never get locks, and detect conflicts by checking if a block changed since they read it while.
- Now it is also clear that there are cases when pessimistic locking will perform better, it is not the case that optimistic concurrency control has no concurrency control overhead; rather, the locking overheads of lock-based approaches are replaced with the overheads of recording read-lists and write-lists of transactions, checking for conflicts, and copying changes from the private workspace. In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.

The Author has described the overall Concurrency control problem and have explained the ideas of concurrency and parallelism. I closed with a brief hint at how to think about concurrency algorithms. The parallel hardware required to run a concurrent program or transactions can be considered for future research area.

VI. REFERENCES

[1] Ambler, Scott. *Introduction to Concurrency Control*, 2006

[2] Andrew S. Tanenbaum, Albert S Woodhull (2006): "*Operating Systems Design and Implementation", 3rd Edition*, Prentice Hall, ISBN 0-13-142938-8

[3] Coronel, Carlos, Peter Rob (2004) "*Database Systems",* sixth edition. Thomson Course Technology

[4] DeWitt and J. Gray (1992) "Parallel database systems: the future of high performance database systems" *Communications of the ACM*, 35 (6): 85–98,.

[5] J. DeWitt and Gerber. R. (1985) "Multiprocessor hash-based join algorithms" in *Proceedings of the Very Large Databases Conference*,.

[6] K. Hua and C. Lee (1990) "An Adaptive Data Placement Scheme for Parallel Database Computer Systems" in *Proceedings of the Very Large Databases Conference*

[7] Kumar, V. (2012) "*Transaction Management Concurrency Control Mechanisms*"

[8] M. Livny, S. Khoshafian, and H. Boral (1987) "Multi-disk management algorithms" in *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77

[9] Oracle Technology Network *'Oracle 9i Real Application Clusters – Cache fusion delivers scalability'* of February 2002

[10] Ricardo (2012) Catherine. *Databases Illuminated*, second Ed. p386-387 Jones & Bartlett Learning

[11] Silberschatz, Avi; Galvin, Peter; Gagne, Greg (2008). *Operating Systems Concepts, 8th edition*.