CMSC624 Final Project STRIFE

Haoran Zhou[*]
University of Maryland
College Park, MD
hzhou127@umd.edu

Junran Yang[*]
University of Maryland
College Park, MD
jryang@umd.edu

Tianrui Guan[*]
University of Maryland
College Park, MD
rayguan@umd.edu

ABSTRACT

This is the final project of CMSC624 at University of Maryland. In this project we implemented Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning (*STRIFE*) [8], particularly, its clustering and parallel union-find algorithm. We also demonstrate the superior performance of *STRIFE* in comparison to other traditional concurrency control protocols like Locking, OCC and MVCC. Additionally, we make two modifications that successfully improve the throughput of the origin work on our benchmark and analyze the reasons behind the improvement. Our code is available on github at: https://github.com/zhr1201/STRIFE. This is a private repository, so if you need access, please contact any one of the authors.

KEYWORDS

Main-Memory Multicore DB, Online Transactional Processing (OLTP)

1 INTRODUCTION AND BACKGROUND

Nowadays, as the number of cores increases, the demand for scalability on multiple computing resources is even more pressing and challenging. There are various efforts working on various concurrency control to reduce contention and guarantee serializabilty, but most time it could be the bottleneck of a database system. In the paper Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning [8], the authors designed a new batch-based Online Transactional Processing (OLTP) system that partition incoming transactions into batches. Based on the access pattern, the transactions are divided into groups that is conflict free across groups. The groups can be executed in parallel on different cores without need for concurrency control, and transactions within the group should be executed with concurrency control protocol, which can guarantee serializability.

As the final project for CMSC624, we choose to implement what was described in the paper [8]. Particularly, we implement all parts of the basic clustering algorithm and parallel union find. We also integrated this method into the framework provided by CMSC624 course and measure the performance using our newly added benchmark method. We discovered that when the workloads fit the assumption of the original paper with high contention, it works better than the lock based concurrency control methods. However, we believe the workload assumption is too strong and STRIFE's clustering algorithm's performance is not ideal under certain types of workloads that do not fit the assumption of the model. Those types of workloads are analysed in detail to cast some insight into why the performance is bad. In addition we made some modifications to the existing algorithm, and show that the modifications results in better performance on those types of workloads that break the original STRIFE CC.

[*]All authors contributed equally to this project.

The main contribution of the work are the following:

- We complete the implementation of *STRIFE* [8]. (Sec 2).
- We propose a novel routine that remove the separate residual processing step of the original work to reduce the blocking time. Removing processing the residual by putting it in the next batch, we are able to increase parallelism by running the worker threads and partitioning the next batch at the same time. (Sec 3.1)
- We relax the restriction during allocation step 2.2.5 by allowing the transactions that access at most one special cluster and several non-special clusters to enter the corresponding queues. In this attempt, queues are not guaranteed to be conflict-free with each other, but they would have very little contention since only few transactions containing cold records in non-special clusters are allowed to enter each queue. However, the number of residuals is drastically reduced. We used locking to ensure consistency. (Sec 3.2)
- We extend CMSC624 project 2 framework in support of *STRIFE* and conduct extensive experiments, comparing throughput with other concurrency control protocols like two phase locking [3, 4, 6], serial and parallel optimal concurrency control (OCC) [5], and multi-version concurrency control (MVCC) [1].

The organization of the paper is as follow. In Section 2, we describe the overview and implementation of the clustering and parallel union find algorithm. In Section 3, we describe some modifications to the original paper and some other implementation details. In Section 4, we present a series of experimentation and analysis. Afterwards, we review some related work in Section 5, show some possible future directions and extensions on our implementation in Section 6 and conclude in Section 7.

2 ARCHITECTURE OVERVIEW

In this section, we give a overview of what we implement for this project: Parallel Union Find and Clustering Algorithm. We briefly go over the main points of those algorithms, and point out some confusions we have on some steps.

2.1 Parallel Union Find Algorithm

We implement what is described in the paper [8]: a concurrent version of union-find algorithm with path compression. In the original paper, it uses a data structure called Record to represent a node in the Union-Find data structure, and the clusters are represented by all the root nodes in the Union-Find data structure. To spot a special cluster in Union(), it requires the address of the Record to be greater than M. However, that requires moving the Records around in memory, thus requires modifying the parent pointer of the children nodes, which is a tremendous amount of overhead.

Instead of using address to identify special clusters as in the original paper, we use a boolean field in the data record to mark whether the records belongs to a special cluster, and use another field for storing the a global unique ID of that Record. If one record is marked as special, we set the boolean field and assign a ID larger than a particular threshold to that record. Doing this ensures the root will remain special as we swap the nodes so that the invariant is still guaranteed.

2.2 Clustering Algorithm

The pseudo code is shown in Figure 2. As described in *STRIFE*, there are five main steps: (1) *PREPARE* (2) *SPOT* (3) *FUSE* (4) *MERGE* and (5) *ALLOCATE*.

2.2.1 *Prepare step.*

We don't create an explicit access graph, but instead we extract the write set of the transactions in the batch along with metadata.

2.2.2 *Spot step.*

We randomly pick a transaction and identify if we can mark its data item as "special". If any of the data items accessed by this transaction has been marked as special, we reject this transaction; otherwise, we union all of its data items into a new special cluster. We repeat this process after k trials, which would end up with less than or equal to k special clusters.

2.2.3 *Fuse step.*

In this step, we process all remaining transactions in the batch, which we ignored or rejected during *Spot step.* If all of the data items accessed by the current transaction belong to at most one special cluster, we union them into the special cluster. If they belong to more than one special cluster, we only record the count.

2.2.4 *Merge step.*

We iterate through all special clusters pairwise, and merge two special clusters if there are enough many transactions that access data cross those two clusters.

2.2.5 *Allocate step.*

During this step, we try to allocate all transactions in the batch to different disjoint clusters according to what we constructed in previous steps, and assign the remaining transactions to the residual list.

2.3 Execution model

All the workloads are partitioned into different batches. Each batch is partitioned into several queues in the worklist and a residual list using the algorithm described above at Section 2.2. After partitioning, different queues can be executed in parallel without contention. After finishing all the transactions in the worklist, the transactions in the residual list is executed serially.

```
Input: List<Txn, ReadWriteSet> batch
Output: Queue<Queue<Txn>> worklist
        Queue<Txn> residuals
set<Cluster> special;
int count[k][k] = {0};
// Spot Step (initially each data node is a cluster)
i = 0;
repeat
    Pick a random transaction $T$ from batch;
    set<Cluster> C = { Find($r$) | $r \in$ T.nbrs };
    set<Cluster> S = { $c$ | $c \in C$ and $c$ is special };
    if $|S| == 0$ then
        c = C.first;
        foreach $other \in C$ do
            Union(c, other);
        c.id = i;
        c.count++;
        c.is_special = true;
        special.insert(c);
        i++;
until $k$ times;
// Fuse Step
foreach $T \in batch$ do
    set<Cluster> C = { Find($r$) | $r \in$ T.nbrs };
    set<Cluster> S = { $c$ | $c \in C$ and $c$ is special };
    if $|S| \leq 1$ then
        c = ($|S| == 0$) ? C.first : S.first;
        foreach $other \in C$ do
            Union(c, other);
        c.count++;
    else
        foreach $(c_1, c_2) \in S \times S$ do
            count[$c_1$.id][$c_2$.id]++;
// Merge Step
foreach $(c_1, c_2) \in special \times special$ do
    $n_1$ = count[$c_1$.id][$c_2$.id];
    $n_2$ = $c_1$.count + $c_2$.count + $n_1$;
    if $n_1 \geq \alpha \times n_2$ then
        Union($c_1$, $c_2$);
// Allocate Step
foreach $T \in batch$ do
    C = { Find($r$) | $r \in$ T.nbrs };
    if $|C| = 1$ then
        c = C.first;
        if $c.queue = \emptyset$ then
            c.queue = new Queue<Txn>();
            worklist.Push(c.queue);
        c.queue.Push(T);
    else
        residuals.Push(T);
return (worklist, residuals);
```

Figure 1: Pseudo-code of the original algorithm [8]

2.4 Analysis of the Algorithm

STRIFE uses hot set to represent the set of data that is accessed very often by the transactions. The assumption of the original paper is that the size of the hot set is small, hoping the spot step 2.2.2 is able to spot most of the hot data records. And in the fuse step 2.2.3 and merge step 2.2.4, it tries to merge some of the special clusters if there are a certain ratio of cross special cluster transactions exist. This merge step is used to reduce the number of partitions so that the number of cross partition transactions is reduced, which leads to shorter residual list. For example, we have special cluster 1 and special cluster 2, and transaction A accesses both special cluster 1 and special cluster 2. Without the fuse and merge steps, it will go into the residual list in the allocate step since it needs access to two special clusters. But if special cluster 1 and 2 get merged because many transactions span across these two clusters, transaction A will count as only accessing 1 cluster thus getting into one of the worklist. This step is necessary because we don't want our residual list to be too long which will reduce concurrency.

However, under certain situations, the above assumption might fail which will lead to bad partition performance and low throughput. According to the pseudo code in Figure 2 (line 33), we merge two special clusters if there are many transactions that access both clusters. However, some of those transactions that motivate the merge would not benefit from this merge: for example, if those transactions also access some records that do not belong to any other special clusters, those transactions would still end up in the residual list. A concrete case is that we have special cluster 1 and 2, the transaction A access special cluster 1, special cluster 2 and a cold data record. In this example, it will not fuse the cold record into the special clusters in the fuse step 2.2.3 because it accesses more than two special clusters. And even though special cluster 1 and 2 get merged in the merge step 2.2.4, this transaction will not get out of the residual list in the allocate step 2.2.5. This counter-intuitive design is the motivation for the adjustment of the clustering algorithm in Section 3.2.

This type of residual also exists even with perfect partitionable data access, particularly when a database has a very large set of hot data records, or there is no hot record and all the data records are accessed with almost the same probability. For a particular partition, because the size of the partition is very big, it might get spotted to different special clusters at the spot step 2.2.2 and there are lots of data records in that partition labeled as cold (not special). In that case, the example we mentioned above is also possible to exist within a single partition. We discovered that this type of residual ratio could be very high for some workloads of certain data access pattern, which motivates our design of Secion 3.1.

3 MODIFICATIONS

In this section, we focus on two modifications we make to the original algorithm: Residual-free Processing and Lenient Allocation.

3.1 Residual-free Processing

The first modification we make is on the overall pipeline. In the original work, we first process different collision-free clusters in parallel, and then process the residual once finished. However, there are two places that could potentially incur blocking: (1) the residuals have to wait until the clusters in this batch is finished and (2) the next around of clustering have to wait until the residual is finished.

Based on this observation, we make the following changes:

- We completely remove the residual processing by injecting residuals to the next batch. To prevent starvation, we could give the residuals higher priority during spot step (2.2.2). And periodically flush the residual queue to force those transactions to get executed.
- Without the residual steps, after we spawn many threads to process different clusters in parallel, we can immediately start to partition the next batch. We will block the main process to assign clusters to worker threads until we finish the current batch and start processing the next batch.

The most obvious reason that we can benefit from from repartitioning the residuals is that, not all transactions enter the residual list because the transaction spans multiple partitions in the database, sometimes it is just because of some random factors and get allocated to the residual list, as we have discussed in Section 2.4. If we repartition those transactions over and over again, chances are high that they will get partitioned correctly in the next few rounds.

3.2 Lenient Allocation

Our second modification is to allow less strict allocation step 2.2.5. In Section 2.4, we point out that during merge step 2.2.4, there could be plenty of transactions that access more than one special clusters and some other cold records outside those special clusters, and they would still remain in the residual list even after they contribute to the merge of two special clusters. To give stronger expression to the merge step, we also allocate those transactions to the corresponding clusters. Even if those clusters are not guaranteed to be conflict-free, the chance conflicts is rare. In our implementation, we decide to use a locking scheme to ensure correctness.

4 EVALUATION

4.1 Setup and Workload Generator

The experiments are conducted using a desktop computer with one Intel Core I7 9700 CPU. The CPU has 8 physicals cores and 8 physical threads without hyper-threading to maximize single core performance.

For evaluation purpose, we implement two new load generators to control the level of contention and partitionability in workloads. We use **Generator A** to compare the *STRIFE* algorithm with other CC protocols and **Generator B** to show the improvement of partition due to our modification.

4.1.1 Generator A. First, the set of all data items, D, is partitioned into two disjoint sets that complements each other: hot data set H and cold data set $D - H$. The hot data set is considerably small comparing to the cold data set so that it is more popular among and more frequently accessed by all transactions. The hot data set, H, is partitioned into several disjoint subsets $H_1, H_2, ..., H_n$. For each transaction T that accesses k data items, x data items are randomly picked from one hot data subset $H_i, i \in [n]$ and the rest $k - x$ items are randomly selected from the cold data set. We also set a certain proportion of transactions to be residuals that accesses data items across several hot data subsets. The reason that we cluster the hot data set and only allow each transaction to access one cluster is to guarantee the partitionability of transactions. The degree of contention in a workload is determined by several aspects: the proportion of hot data access in partitionable transactions, the proportion of residuals in all transactions and the size of data accessed by residuals.

Access Pattern	CC Protocols (throughput)						
	Serial	Locking A	Locking B	OCC	Parallel OCC	MVCC	STRIFE
Contention High	9084	13183	13121	13459	15959	1897	**16496**
Contention Medium	9186	14202	14160	12136	17548	1908	**18796**
Contention Low	9210	20610	20518	16008	**25615**	2560	20811
Perfectly Partitionable	947	3895	**3993**	2883	3974	954	3223

Table 1: This table describes the throughput of different concurrent control protocols under a modified benchmark based on CMSC624, assignment 2. Serial, Locking A, Locking B, OCC, Parallel OCC and MVCC are implementations directly from that assignment. STRIFE is the original version of the algorithm without any modifications. The workloads are generated by Generator A. Normally a transactions would randomly access few hot data items from one cluster of the hot data set, and several cold data items. In the experiment, we also inject some transactions that access cross many clusters in hot data set and use percentage of this type of transactions to control the level of contention. Contention high, medium and low corresponds to 10%, 8% and 5% of such transactions. In the last access pattern, we made a perfectly partitionable transactions, which means that those transactions could be partitioned without residual in an optimal solution. (See details in Section 4.1)

4.1.2 Generator B. This generator generates transactions without hot set, so all the data items are treated equally by this generator. Before generating any transactions, all the data items in the database are partitioned into a designated number of clusters. We specify the percentage of residual transactions, and generate partitionable transactions and residual transactions according to this percentage. For a partitionable transaction, we randomly choose a cluster in the database and generate its data set only from that cluster. And for residual transactions, we select its data set randomly from the whole database. Even if the transactions are perfectly partitionable, the cluster algorithm can fail to generate exact clusters matching the ground truth. We compare our modification and the original cluster algorithm using perfectly partitionable workload created by Generator B to show the throughput deference that reflected by the effectiveness of partitioning.

4.2 CC Protocols

We compare *STRIFE* against 6 types of CC protocols:

Serial runs the transactions in the serial order as they come in.

Locking A is the standard locking scheme that do not distinguish write locks and read locks.

Locking B is a motified version of Locking A, except that we allow read locks to be shared. Only write lock is exclusive.

OCC is optimistic concurrent control scheme. It execute the transactions before hand in a temporary buffer and only commit and apply the changes to stable storage if there is no conflict during the execution. Otherwise we will abort the transactions and put it back in the request list, or return to the user and proceed at the discretion of the client.

Parallel OCC will run OCC scheme in parallel to reduce latency. Usually the parallel version of OCC outperform the non parallel version.

MVCC is multi-version concurrency control scheme. It uses locking and store multiple versions of the data record in disks and support snapshot read. Usually it has more overhead than other schemes for managing multiple versions of data items.

In this experiment, we use generator A to generate all the workloads. The size of the hot set is 20, and is partitioned into 4 clusters, each transaction accesses 5 records from the hot set. Under this setting, we could achieve at most X4 speedup if all the transactions are partitioned perfectly and there is no residual transaction left. We used a batch size of 10k and used the execution time of 100k transactions to get the throughput. The contention is controlled by the number of hot set records that a residual transaction (cross partition transaction) accesses and the percentage of such transactions. Table 1 shows the throughput we get for running different CCs under different contention setup.

For high contention workload, each residual transaction accesses 10 hot data records randomly and the percentage of those transactions is 10%. For middle contention workload, each residual transaction accesses 8 hot data records randomly and the percentage of those transactions is 8%. Under high contention, Lock A and Lock B will block lots of transactions as we have discussed in Assignment 2. And in this case STRIFE gets better performance because it can partition the workloads and executes parts of the transactions without contention.

For low contention workload, each residual transaction accesses 2 hot data records randomly and the percentage of those transactions is 2%. Actually "low" is only a relative term here, and we can tell that actually the contention is still high according to the performance of OCC. In this case, the performance of Lock A and Lock B are better than under high contention. The performance gap between STRIFE and Lock is smaller.

And if we continue to reduce the residual transactions and have a perfect partionable workload. The Locking methods are the fastest, because if a transaction gets blocked by another for acquiring the lock, it means they are from the same partition and those transactions should also get assigned to the same worklist in STRIFE. Even though some transactions get blocked by the lock manager, different transactions from different partitions will not block each other, so there should be at least one transaction from each partition that is running at any point. Since STRIFE has extra partition overhead than the lock manager, it is slower in this case. (In this part, the transaction execution time is set longer to reduce the effect of partition overhead, so the throughput is 10 times smaller than the previous experiments.)

4.3 Ablation Study

Access Pattern	*STRIFE* Modifications (throughput)			
	Original	Residual-free	Lenient Allocation	Both
Contention High	15668	**20902**	11627	15436
Contention Low	5087	422[*]	9944	**14833**

Table 2: This is ablation study of *STRIFE*. We compared the original implementations with our modifications and run experiments with perfectly partitionable transactions with high and low contentions by adjusting database size. The workloads are generated by Generator B.

In this section we compare the original implementation of *STRIFE* with the improvement. In Table 2, we can see we improve the final throughput in both high contention and low contention.

High Contention: The parallelism of clustering and execution is very effective in high contention experiment. The reason why it can perform so good is that reduce the blocking time when the residual is waiting for clusters' execution to finish and when the partition algorithm is waiting for the residual to finish. The effect of locking is worse than the original implementation because there is many unnecessary overhead when locking on the hot record, which is guaranteed to be collision-free. We sacrifice too much to guarantee the correctness of our second modification 3.2. However, currently we do not have solutions to only lock the cold records because we have no advance knowledge on whether records are hot or cold.

Low Contention: In low contention case, the residual-free implementation is extremely useless. In most of time, it never return after a long period of time because the clustering algorithm could only identify few clusters each around, so too many transactions in the residual are pushed to the next batch for the next round of partition. This could repeat forever and the progress is made very slowly as a result. However, the locking work extreme well because instead of putting the residual in the next batch, we put more transactions in the residuals than the original algorithm does. It seems that the overhead of locking is trivial compared to the effectiveness of clustering a great number of residuals to run in parallel. It is worth noting that the version that implements both residual-free and lenient allocation has the greatest throughput. Based on the analysis, we conclude that a more lenient allocation would greatly reduce the number residual carried to the next batch, so the problem residual-free implementation encountered in low contention case will be resolved.

5 RELATED WORK

In this section, we review three related fields: (1) Different CC protocols with trade-offs in *STRIFE*; (2) Partitioned workload and multi-cores CC protocols; (3) Discussion on Serializability.

[*]In the experiments with Residual-free implementation in low contention case, the experiment often get stucked. Only occasionally it will give a very unfavorable results.

CC Protocols. In general, there are two types of concurrency control protocols: pessimistic protocols like 2-PL [3, 4, 6], or optimistic protocols like OCC [5] and MVCC [1]. The pessimistic protocols usually works better during high contention, but might have deadlock issue and long time blocking for time-consuming transactions. Pessimistic concurrency control would also have locking overhead for low contention access pattern, which could be the bottleneck of the database system. On the other hand, optimistic concurrency control would avoid locking overhead for low contention transactions, but might have a higher abort rate in case of high contention transactions.

In the implementation of *STRIFE*, we need to choose a CC methods to handle different special clusters in parallel. Here, using serial execution order or pessimistic concurrency control would work better because the transactions within a cluster is guaranteed to have high contention. Moreover, serial execution would have slightly better performance when the clusters are highly compact and have mostly overlapping write/read set.

Partitioned Workload and Multi-cores CC. There are many previous work on database partitions [2, 7, 9], all of which are static partition protocol and requires offline computation. *STRIFE* is more efficient in comparison because it uses dynamic clustering.

Serializability. Serializability is the strongest guarantee of isolation level. In most modern database systems, serializability becomes a standard consideration for implementation. It it an important aspect for concurrent scheme, which ensures executing concurrent transactions is equivalent to executing those transactions in some serial order. Serializability guarantee that each transactions would see a valid state of a database at all times. However, serializabiltiy in the context of distributed system would still have causal reverse: if a write transaction X happens before and caused a write transaction Y, it's still possible to have another read transaction Z that sees the effect of Y but not the effect of Z. With strict serializability, such causal reverse would guarantee not to happen.

Our implementation only guarantees serializability, not strict serializability. For example, in previous case, it's possible to have Y assigned in one of the cluster and Z and X assigned in the residual. As a result, Z might see the effect of Y but not X if Z is ordered before X in the residual.

6 FUTURE WORK

We have limited time for this work, but there are definitely some future works to be done and future directions to pursue:

- Access pattern of transactions can greatly affect the final throughput of the concurrent control scheme. We explore many different pattern setup and modified the transaction generators many times for analysis. This is a very interesting direction to pursue in order to better understand the effectiveness of the clustering algorithm and how to improve it further.

- For the Lenient Allocation Scheme in Section 3.2, there is too much overhead for locking, and extremely not efficient in our current implementation. A future work could be optimizing the locking scheme, and only lock those non-special record as they are not guaranteed to be collision-free across clusters.
- We didn't quantify the relationship between the STRIFE throughput and the partition results. We did analyse a few cases in the test we wrote, but a systematic analysis of this relation and how different kinds of workloads will affect the part ion results give us more insight into this problem.

7 CONCLUSION

In this project, we implemented the STRIFE concurrency control mentioned on [8]. The advantage and disadvantage of this method is analysed and verified by the experiments, and we compare the performance of STRIFE with other CC, particularly lock based CC, under several kinds of workloads. According to the disadvantage of STRIFE we have discussed at Section 2.4, we proposed two improved version of STRIFE and showed those modifications can result in higher throughput under certain types of workloads.

ACKNOWLEDGMENTS

To Guna Prasaad, Alvin Cheung and Dan Suciu for their work Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning [8], which our work is based on. And thank Dr. Daniel J. Abadi and TA Gang Liao for providing this amazing class CMSC624 at University of Maryland, College Park.

REFERENCES

[1] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[2] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[3] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

[4] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques.* Elsevier.

[5] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[6] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[7] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[8] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 527–542. https://doi.org/10.1145/3318464.3389764

[9] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 430–441. https://doi.org/10.1145/2452376.2452427

A APPENDIX: MODIFICATION OF THE CODE BASE

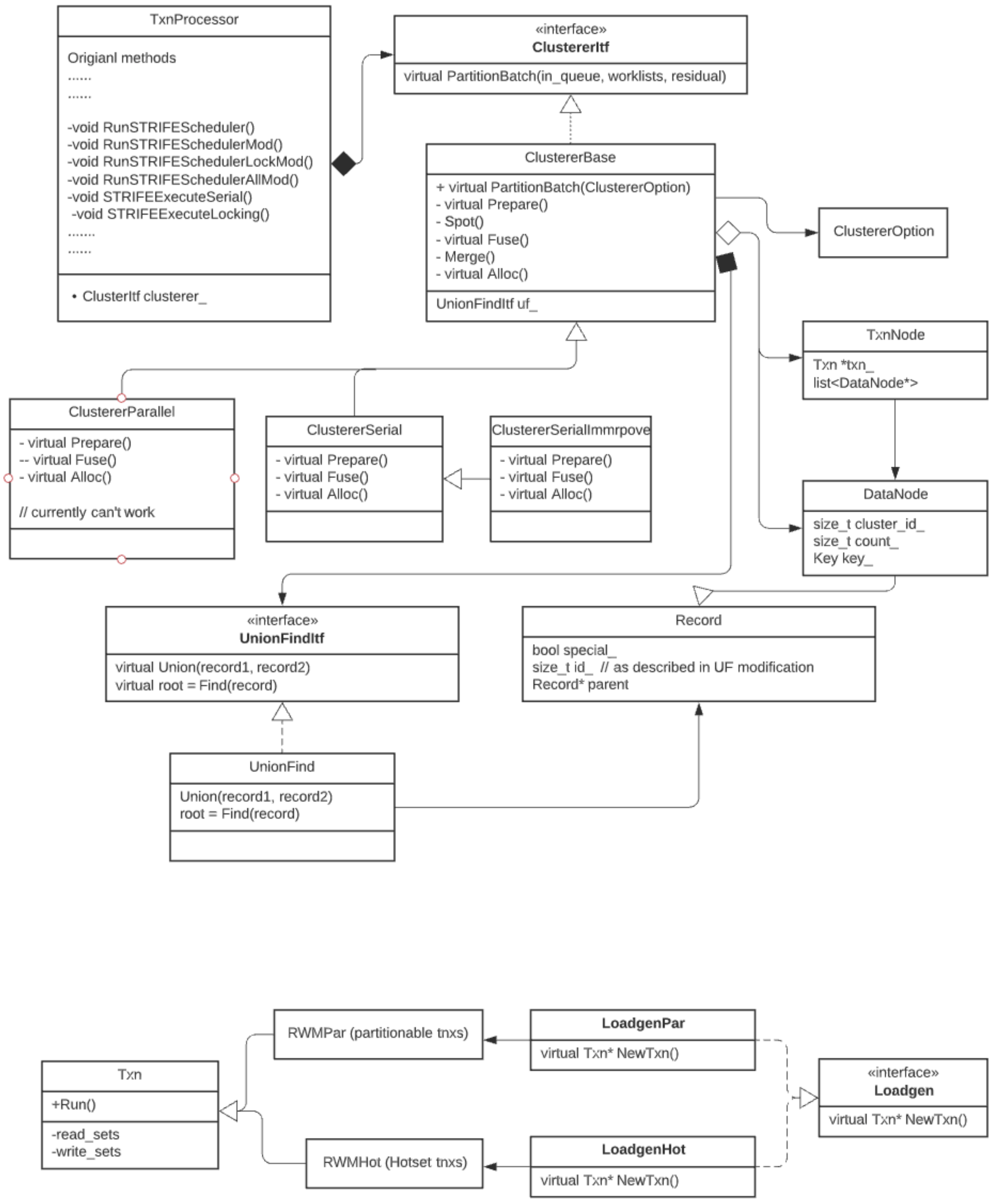

Figure 2: UML diagram of the parts we added to the original as2 code base.