Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All

Dixin Tang
The University of Chicago
totemtang@uchicago.edu

Hao Jiang
The University of Chicago
hajiang@uchicago.edu

Aaron J. Elmore
The University of Chicago
aelmore@cs.uchicago.edu

ABSTRACT

Use of transactional multicore main-memory databases is growing due to dramatic increases in memory size and CPU cores available for a single machine. To leverage these resources, recent concurrency control protocols have been proposed for main-memory databases, but are largely optimized for specific workloads. Due to shifting and unknown access patterns, workloads may change and one specific algorithm cannot dynamically fit all varied workloads. Thus, it is desirable to choose the right concurrency control protocol for a given workload. To address this issue we present adaptive concurrency control (ACC), that dynamically clusters data and chooses the optimal concurrency control protocol for each cluster. ACC addresses three key challenges: i) how to cluster data to minimize cross-cluster access and maintain load-balancing, ii) how to model workloads and perform protocol selection accordingly, and iii) how to support mixed concurrency control protocols running simultaneously. In this paper, we outline these challenges and present preliminary results.

1. INTRODUCTION

The demand for high-throughput transactional "many core" main-memory database management systems (DBMS) is growing due to dramatic increases in memory size and CPU core counts available for a single machine [21]. A high-end modern server can have hundreds of gigabytes to dozens of terabytes of main memory and dozens to hundreds of cores, such that most datasets are entirely stored in memory and are processed by a single server. Concurrency control enables ACID transactions to read and update records, while potentially executing operations across many cores. Due to the elimination of a traditional performance bottleneck, disk stalls, recent concurrency control protocols address new bottlenecks raised in a main-memory system. Some protocols try to minimize the concurrency control overhead [8], while other protocols strive to avoid single contention points across many cores [18]. These protocols are typically designed for specific workloads and exhibit high performance under their optimized scenarios.

The access patterns of a workload may be unknown before deployment of user-facing applications or be unpredictable and fluctuate in response to users' requests. For example, administrators may not know popular items or cannot estimate their hotness a priori; a set of records that become hot due to breaking news article may only remain popular for a short time; or the set of potential popular records can be unpredictable and the velocity of records becoming hot or cold is unknown. A challenge with an unknown and shifting workload is that databases are often not adaptive and use a fixed concurrency control protocol. Though such protocols achieve ideal performance in their optimized workload scenario, they fail to maintain the same performance with fluctuating workloads. Consider H-Store's concurrency control protocol using coarse-grained locking [8]. It works well for partitionable workloads, which means the whole dataset is partitioned into disjoint partitions such that a transaction is highly likely to access only one partition. But this approach suffers low throughput with increasing cross-partition transactions [21, 18]. Although reconfiguration can decrease cross-partition transactions for some scenarios [17], certain workloads may never exhibit access patterns that are amenable for partitioning. For these cases an optimistic protocol may be ideal if the workload mainly consists of read operations or has low-skew access patterns, otherwise a pessimistic protocol may be ideal.

In this paper we consider how workload features, such as record conflict rates and how partitionable the workload is, impact the comparative performance of concurrency control protocols. We illustrate the impact of workload features on a transactional main-memory database using three representative concurrency control protocols. Specifically, it includes a partition based concurrency control (*PartCC*) from H-Store [8], an optimistic concurrency control (*OCC*) based on Silo [18], and a no-wait two-phase locking method (*2PL*) based on VLL [21, 14]. We use a YCSB benchmark to test their performance under workload variation, with a detailed configuration given in Section 6. Under workload variation, the fluctuation of performance among concurrency control protocols is evident in Figure 1. We see the throughput of all concurrency control protocols greatly varies, and each protocol has cases where it performs best. At first, OCC works best when the workload is not partitionable and its record conflicts are low (WL1), then the workload becomes well partitioned and PartCC is the best choice (WL2), and finally 2PL is ideal when the workload

8th Biennial Conference on Innovative Data Systems Research (CIDR '17)
January 8-11, 2017 , Chaminade, California, USA.

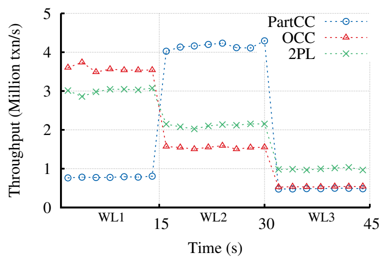

Figure 1: Throughput in the presence of workload variation

turns into non-partitionable with a high record conflict rate (WL3). For the three workloads, the best protocol has 4.5x, 2.6x and 1.9x higher throughput than the worst protocols. This highlights that no single concurrency control is ideal for all workloads and that the wrong concurrency control protocol can have significant performance degradation.

Instead of statically choosing a single concurrency control in a DBMS, multiple concurrency control protocols should be supported simultaneously and chosen according to the current workload. However, relying on database administrators to manually choose concurrency control algorithms is not a feasible solution since workload variation can be frequent or difficult to predict such that manual tuning fails to follow changes in demand. Therefore, concurrency control in transactional databases, in addition to preserving ACID properties, should adapt to workload changes without manual intervention or disrupting performance.

In this paper, we propose **adaptive concurrency control (ACC)**, a holistic solution to varied and dynamic workloads for main memory OLTP databases. ACC automatically clusters data and workers (e.g. cores), and selects the optimal concurrency control protocol for each cluster according to the current workload. Building ACC requires addressing the following major research challenges.

Data Clustering In ACC, we partition the dataset into clusters and assign each cluster with a concurrency control protocol. We also allocate dedicated CPU cores to each cluster for transaction execution. A good clustering algorithm should maintain that most transactions execute on a single cluster, cluster utilization is balanced, and overall throughput is maximized. This problem introduces the following challenges that have not been previously addressed. First, workloads on each cluster should be balanced with flexible assignment of cores to clusters to balance computing capacity and synchronization overhead. Second, the number of clusters is not determined in advance and may change on the fly. Finally, cross-cluster access costs may vary when different protocols are applied to the clusters.

Workload Modeling and Protocol Selection For a cluster of data, we need to choose an optimal protocol according to the current workload. The challenge is how to model the comparative performance of different concurrency control protocols given a workload. This is difficult because the model may encounter workloads that have not been studied before. To enable this model, databases statistics (or features) need to be collected online from runtime protocols. Feature extraction should be lightweight and fast such that ACC can have a timely response to workload changes. We embed feature extraction into transaction workers to parallelize this process. One key problem is, given a workload how to normalize features across different protocols when concurrency control workers exhibit various behaviors and signals for a feature.

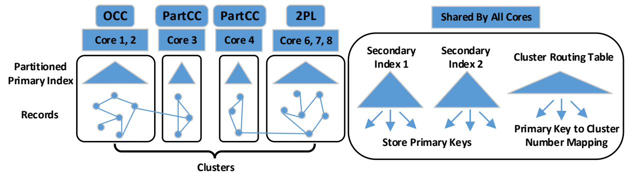

Figure 2: An ACC sample configuration

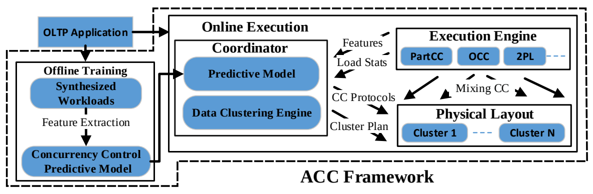

Figure 3: ACC components and overview

Mixing Concurrency Control Protocols As each cluster can have a different protocol, ACC needs to address challenges raised by mixed concurrency control protocol execution: how to maintain an isolation level (e.g. serializable) when transactions span clusters (and subsequently protocols) without introducing extra concurrency control overheads, and how to mix durability and recovery algorithms of different protocols. Protocols adopt various methods to guarantee properties of isolation and consistency, but they fail to maintain these properties when transactions span protocols without the use of distributed commit protocols. Previous work [20] proposes a general framework to address this issue by introducing an extra layer of locking mechanism to coordinate conflicts across different protocols. This extra locking mechanism may limit the scalability of the databases and become the performance bottleneck with high core counts. On the other hand, ACC will bridge various protocols such that conflicts among them can be reflected in each protocol without introducing any extra overhead. In addition, durability and recovery algorithms can be optimized for a protocol, but contradict with other implementations or optimizations. For example, PartCC often adopts command logging to record transactions applied to databases, while the logging algorithm of Silo logs record values rather than operations. A mechanism is required to manage dependencies between different logging algorithms to ensure correct recovery.

2. ACC OVERVIEW

ACC dynamically clusters the data store, assigns CPU cores to clusters, and selects a concurrency control protocol for each cluster. Figure 2 shows a sample configuration, where the data store is divided into four clusters such that the records co-accessed by transactions (denoted as solid lines connecting records) across clusters are minimized. Each cluster owns a partition of the primary index and corresponding records, and is assigned one or more CPU cores according to load statistics. We denote a cluster assigned with one core as a partition and assume a concurrency control worker or a transaction worker (i.e. a thread or process) occupies a physical core. The concurrency control for each cluster is customized according to its access patterns. For example, the first cluster is configured with OCC since it receives transactions having few conflicts. For a concurrency control protocol, we name the cluster it is running on as the *home cluster* and denote other clusters as *remote clusters*. The cluster routing table maintains the mapping from primary keys to cluster numbers and secondary indices point to primary keys; both are shared by all cores.

To enable dynamic configuration of concurrency control protocol for each cluster, ACC requires training a predictive model for protocol selection offline, and reconfiguring clusters organization and protocols online. The predictive model incorporates a set of effective features which cover major characteristics of a workload and uses machine learning classifiers to predict the optimal protocols. Figure 3 gives an overview of how offline model training and online execution works. We use synthesized workloads to explore workload variation as much as possible in the offline training. For example, we mix the stored procedures of an OLTP application to explore the effects of the number of write operations on candidate protocols. Note that it's possible to use an online workload trace (e.g. log) to train the model or improve it incrementally.

The core component of online execution is a central coordinator that controls the clustering and determines the protocol for each cluster. ACC monitors transactions' access patterns and extracts features from the active workload. ACC uses load statistics to generate a new clustering plan and lets our predictive model choose the optimal protocols using the extracted features. Then, according to the new plan clusters are reorganized, CPU cores are reassigned, and protocols are reconfigured accordingly.

3. DATASET CLUSTERING

ACC partitions a dataset into clusters and selects a concurrency protocol for each cluster. Most existing transactional database partitioning solutions target minimizing cross-partition access, such that transactions should access in average as small number of partitions as possible [3, 6, 7]. While taking this into account, ACC also needs to address core allocation, concurrency control synchronization overheads, and load-balancing when clustering data. For example, if one partition has only a very few transactions, ACC can merge it with another partition as it would be a waste of resource to allocate a core to it. Similarly, if one partition constitutes almost of all workload's requests, then we can merge all partitions together. When the workload changes ACC should dynamically recluster the dataset with low overhead. Therefore, ACC must address the following goals for clustering:

1. Minimize cross-cluster accesses.
2. Maintain load-balancing and maximize throughput.
3. Recluster the dataset on the fly with minimal overhead, while maintaining ACID properties.
4. Determine when to recluster the dataset.

In the rest of this section we describe ACC's initial approach to clustering, but believe that integrated solutions to the above problems require new techniques.

Clustering Many existing projects target partitioning a database to minimize cross-partition access [6, 7]. While these approaches work for many scenarios, they all require the number of expected partitions to be known in advance or rely on simple heuristics for adding partitions if a solution cannot be found [17]. In ACC we are facing several new challenges. First, cross-cluster access cost varies with multiple concurrency protocols. For example, it costs less for a transaction's home cluster running OCC to access a remote cluster running also OCC, while it costs more if the remote cluster runs PartCC since accessing the remote cluster requires locking the whole cluster. Second, the number of partitions is no longer a constant, but a variable that is determined by both the workload and the number of cores available. In addition, we also need to assign cores to partitions such that we maximize throughput for clusters, while avoiding over-assigning cores to limit synchronization overheads that can arise from too many workers [21].

Our initial prototype uses a Partition-Merge approach to generate the plan of cluster layouts and cores assignment. In the partition step, we employ existing partitioning techniques [3, 6, 7] to split the dataset into clusters as the same number of available cores, then compute the utilization for each cluster. The utilization of a cluster is defined to be the ratio of number of operations that fall into the given cluster to the total number of operations:

$$U(c) = \frac{|\{o : o \text{ access } c\}|}{|O|}$$

where c is a given cluster, O is the set of all operations, and $o \in O$ is a single operation.

In the merge step, we merge clusters considering utilization and cross-cluster access cost. To make sure no core is assigned to a cluster with low utilization, we put all clusters having utilization that is below a predefined threshold into a pile and merge the two clusters with the lowest utilization. If the merged cluster has a utilization that is above the threshold, it is moved out of the pile. Otherwise, the new cluster is put back to the pile and the first step is repeated. This process ends if 1) all the clusters have utilization that is greater than the threshold, or 2) there is only one cluster left. In the second case, the only cluster left is merged with the cluster with highest utilization.

To reduce cross-cluster access cost, we also merge clusters having high percentage of cross-cluster access. Let i, j be two records and e_{ij} be the access cost when they are involved in a single transaction. The affinity of two clusters is defined to be the ratio between cross-cluster access to the total records co-access within the two clusters:

$$A(c1, c2) = \frac{w_k \sum_{i \in c1, j \in c2} e_{ij}}{\sum_{i,j \in c1} e_{ij} + \sum_{i,j \in c2} e_{ij}}$$

where w_k is the weight introduced by concurrency control protocols running on these clusters. Clusters with high affinity will be merged together. Finally, we assign cores to the cluster that is proportional to their utilization, while factoring the overhead of additional worker contention.

Reclustering Another challenge ACC faces is to minimize the continuous need of reclustering. Reclustering requires reorganizing primary indices in each cluster, which introduces high latch contention to running transactions. To minimize this contention, we freeze the original indices, migrate data in frozen indices into newly built indices, and operate with tiered indices. This method allows original indices serve most transactions and avoids most latch contention. We leave incremental reclustering to future work.

For secondary indices, if they align to the partitioning of the primary indices, they are partitioned in the same way. Otherwise, if secondary indices are rarely modified, we replicate them among all cores to reduce latch contention. If they are frequently modified, we can dedicate a few cores to process index access requests, which minimizes the synchronization overhead of index latches.

4. WORKLOAD MODELING AND PROTOCOL SELECTION

ACC uses a model composed of classifiers to predict the optimal protocol for a workload. Obtaining this model requires solving three problems: how to engineer features to model workloads, how to efficiently extract features from a database at runtime, and how to synthesize workloads to train the model.

Workload Modeling To model workloads for candidate protocols, we first find workload characteristics that affect the protocols' comparative performance and design corresponding features to capture these effects. While ACC currently supports PartCC, OCC, and 2PL, we plan to extend it to more protocols and variants in the future.

ACC uses four features to model the comparative performance of PartCC, OCC, and 2PL given a workload. We track the ratio of read operations (`ReadRatio`), the average number of records accessed by transactions (`TransLen`), and the probability of concurrent transactions reading or writing the same records (`RecContention`). These features model record conflicts across transactions, which is a critical factor influencing the performance of PartCC, OCC and 2PL. The fourth feature `CrossCost` represents the cost of all cross-cluster transactions and influences the applicability of PartCC.

Feature Extraction The basic process of feature extraction involves concurrency control workers collecting workload statistics in parallel and reporting them to a centralized coordinator. The coordinator then extracts features from these details. With many cores, a potential bottleneck of this process is the centralized feature extraction, which will delay the response to workload variation. ACC removes this bottleneck by moving central extraction to concurrency control workers. However, applying this idea to extract consistent feature values among different protocols is challenging when transaction workers of various protocols use different signals to reflect the same feature. For example, detecting record conflicts in OCC may be reflected with transaction aborts, whereas in PartCC queue length and lock wait time may be used. We address this challenge by making feature extraction independent of running protocols. The basic idea is to build a separate process by sampling transactions or operations.

For our initial system we develop an algorithm to estimate concurrent record access to extract record contention (`RecContention`) without heavyweight synchronization between workers. Each record is associated with a counter for contention detection and each worker runs this algorithm independently. It is composed of two lightweight repetitive phases, a mark and a detection phase. In the mark phase, concurrency control workers sample a fixed number of record accesses and mark these records accessed by incrementing counters. In the detection phase, workers sample a configured number of record access and check records' counters. For each counter, if a counter is zero there is no contention on this record (i.e. not marked by other workers). If a counter is more than zero, the worker copies the counter's value and checks whether this record is marked by itself – and if so it decrements the local value. Then, the worker adds the value (i.e. number of other workers marking this record) to its local contention counter, which acts as a proxy for record contention. After a time period, it clears all marks and repeats the process.

Model Training We build a model by training on synthesized workloads. Using synthesized workloads can effectively explore a feature space, making the trained model robust to predict the ideal concurrency control. We synthesize workloads using various stored procedure mixes and access distributions (e.g. vary *theta* for Zipf) to evenly explore feature space using well known search algorithms. Note that it is also possible to use online workloads to train this model or improve it incrementally. For example, we can use workload trace (i.e. logs) to generate training cases by running all protocols in a database replica.

5. MIXED CONCURRENCY CONTROL

As clusters in ACC can use different concurrency control protocols, the use of simultaneous protocols requires addressing two challenges: how to enforce an isolation level across protocols with minimal overhead and how to enable fast logging and recovery with mixed protocols running.

Enforcing Isolation Across Protocols To guarantee a given isolation level, workers of mixed concurrency control need to detect and coordinate transaction conflicts among concurrently running protocols (e.g. OCC and 2PL workers writing the same record), which can be a performance bottleneck with mixed concurrency control. For example, Callas [20], which mixes concurrency control protocols, coordinates such conflicts by introducing an additional locking mechanism that is independent of different protocols; if a transaction reads or writes a record, Callas requires the transaction to acquire a lock beforehand. This locking mechanism limits the scalability of protocols that strive to eliminate lock overhead (e.g. OCC or PartCC). We identify that the primary cause of conflict coordination overhead is from different protocols accessing the same records. To eliminate such overhead, we propose a Data-oriented Mixed Concurrency Control (*DomCC*), where a subset of records (i.e. clusters) are assigned a *single* protocol to manage all concurrent record read and write operations, which we term as *a protocol mananging the cluster*. For example, a transaction that is dispatched to a cluster that is managed by OCC may also access records in a cluster that is managed by 2PL; here, this transaction will need to follow the concurrency control logic of 2PL when it accesses these records. Note that for each candidate protocol, only one protocol instance exists in the whole system regardless of the number of clusters using that protocol.

Figure 4 shows the framework of mixing the three protocols in DomCC. We see the the life cycle of a transaction is divided into four phases: Preprocess, Execution, Validation, and Commit. In the Preprocess phase, if the transaction needs to access clusters that are managed by PartCC, all partition locks of these clusters are acquired in DomCC. Note that the clusters a transaction needs to access are

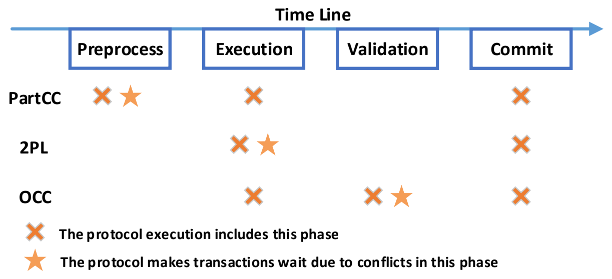
Time Line
Preprocess
Execution
Validation
Commit
PartCC
2PL
OCC
The protocol execution includes this phase
The protocol makes transactions wait due to conflicts in this phase

Figure 4: Mixing PartCC, OCC, and 2PL

known before it starts. Then, DomCC starts to execute the transaction logic. For each record access, it first examines the cluster it belongs to and subsequently the protocol managing the cluster; then it executes corresponding concurrency control logic according to the protocol managing the record. For example, if OCC manages a record, the transaction reads the timestamp and the value of this record using the logic of OCC; if 2PL manages the record, the transaction acquires a lock before reading or writing the record. Next, when the transaction enters the Validation phase, it executes OCC validation for records managed by OCC. Finally, in the Commit phase DomCC applies all writes and releases locks to make these writes visible to other transactions. Specifically, it releases partition locks for PartCC, record locks for 2PL, and write locks for OCC. Note that we assume a rigorous 2PL here.

We now show that the mixed execution of PartCC, OCC, and 2PL is conflict serializable. For two committed transactions $t(i)$ and $t(j)$, we consider four cases:

- If $t(i)$ and $t(j)$ conflict in requesting partition locks of PartCC, either $t(i)$ or $t(j)$ will wait in the Preprocess phase until the other transaction finishes. Therefore, in this case no conflict-cycle exists for the two transactions.
- If $t(i)$ and $t(j)$ conflict in requesting record locks of 2PL, either $t(i)$ or $t(j)$ will wait or abort in the Execution phase until the other transaction finished. Therefore, in this case no conflict-cycle exists for the two transactions.
- If $t(i)$ and $t(j)$ conflict in OCC, there is also no conflict-cycle since OCC can guarantee the conflict serializability of $t(i)$ and $t(j)$ and no conflicts exist for other two protocols.
- Otherwise, no conflicts exist; therefore, any schedules of $t(i)$ and $t(j)$ are serializable.

In addition to providing conflict serializability, DomCC can detect or avoid deadlocks. We assume any deadlock introduced within a single protocol can be avoided or detected by the protocol itself. Therefore, we must examine whether deadlocks exist across protocols. We observe that in each phase there is a distinct protocol making transactions wait due to transaction conflicts. Therefore, transactions blocked by a protocol will not wait for transactions blocked by other protocols in earlier phases. This means there is no wait-cycle across the three protocols, so DomCC can detect or avoid deadlocks. Finally, all candidate protocols release write locks at the end of transactions. Given that the OCC implementation is also strict, this guarantees that DomCC is strict and thus recoverable. While our particular configuration of DomCC provides these properties with no modification to the underlying protocols, an interesting area to explore is how to guarantee the same properties while minimizing lock time, adding more protocols or variants, and supporting different phase orderings.

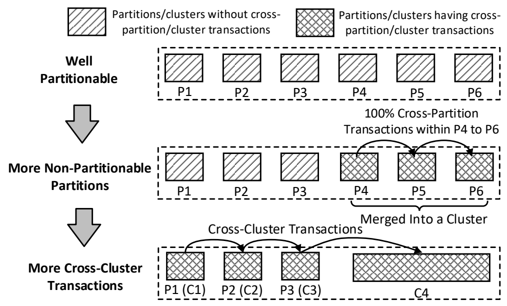
Partitions/clusters without cross-partition/cluster transactions
Partitions/clusters having cross-partition/cluster transactions
Well Partitionable
P1
P2
P3
P4
P5
P6
100% Cross-Partition Transactions within P4 to P6
More Non-Partitionable Partitions
Merged Into a Cluster
Cross-Cluster Transactions
More Cross-Cluster Transactions
P1 (C1)
P2 (C2)
P3 (C3)
C4

Figure 5: Increasing the percentage of cross-partition transactions in two ways: i) increase the number of non-partitionable partitions; ii) increase the percentage of cross-cluster transactions

Durability and Recovery To accomplish efficient logging and recovery in the presence of many cores, ACC exploits recovery algorithms from SiloR [23] that achieves durability and fast recovery using parallel logging. While originally designed for Silo OCC, we extend it to 2PL and PartCC. SiloR depends on time periods called epochs. A global epoch is periodically advanced and transactions belonging to the same epoch are logged together (i.e. batch logging) to guarantee serial order among transactions belonging to different epochs. The key to enabling SiloR is computing transaction IDs (TIDs) according to timestamps of the records a transaction has touched. For each record of 2PL, it is assigned a timestamp and 2PL can use the same method as OCC to compute TIDs. For PartCC clusters, we assign each cluster a timestamp such that records belong to the same cluster use the same timestamps and adopt Silo's approach to compute TIDs.

A challenge for recovery with mixed concurrency control is how to support multiple durability and recovery algorithms at the same time, which we believe is still an open question. Supporting multiple forms of logging is desirable if a logging algorithm specifically designed for a concurrency control protocol can achieve faster logging and recovery. Command logging designed for PartCC, for instance, can minimize overhead to transaction processing by only recording the procedure ID and parameter values (i.e. logical logging) and fully parallelizing recovery by taking advantage of partitioned executors. SiloR, on the other hand, does not rely on partitions but adopts value logging, which logs record values updated by transactions (i.e. physiological logging), to parallelize recovery. The overhead of SiloR is that value logging requires writing more data than command logging. The challenge of mixing the two algorithms is that it is difficult to identify dependencies between logged transactions and values and serialize them during recovery.

6. PRELIMINARY RESULTS

Prototype We have developed a prototype of several ACC components based on Doppel, an open-source main-memory OLTP database [11]. In our prototype, clients issue

Table 1: Workloads for testing mixed concurrency control

	Well Partitionable	Partially Partitionable	Not Partitionable
High Conflicts	WL1	WL3	WL5
Low Conflicts	WL2	WL4	WL6

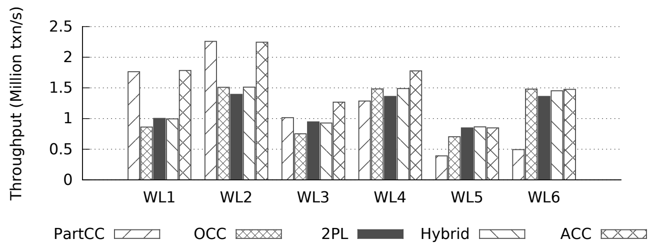

Figure 6: Throughput in various workloads

one-shot transaction requests using pre-defined stored procedures, where all parameters are provided when a transaction begins and transactions are executed to the completion without interacting with clients. Stored procedures are written in Go[1] and issue read/write operations using interfaces provided by the prototype. Each one-shot transaction is dispatched to a transaction worker, which runs this transaction to the end (commit or abort).

Transaction workers are extended to extract features from the current workload and report them to a central coordinator. The coordinator automatically selects and switches protocols online. Our prototype currently supports mixing PartCC of H-Store, OCC from Silo, and 2PL No-Wait based on VLL [14], which co-locates locks with records. Note that we have compared two 2PL variants No-Wait and Wait-Die, and find that 2PL No-Wait performs better than Wait-Die in most cases because of much lower synchronization overhead of lock management [13]. We additionally include a hybrid approach of 2PL and OCC (denoted as Hybrid) [19], which uses locks to protect highly conflicted records and uses validation for the other records. Here, we statically analyze a workload to configure the set of records using 2PL. Our prototype automatically selects protocols using a predictive model based on a DecisionTree [1] and supports switching protocols online. We are currently working on data clustering and recovery for mixed protocols.

Experiment Settings All experiments are run on a single server with four NUMA nodes, each of which has an 8-core Intel Xeon E7-4830 processor (2.13 GHz), 64 GB of DRAM and 24 MB of shared L3 cache, yielding 32 physical cores and 256 GB of DRAM in total.

We use YCSB [5], Smallbank [4], and TPC-C [2] in our experiments. We generate 10 million tuples for YCSB, each occupying 500 bytes. Each transaction in YCSB consists of 20 operations mixed with read-only and read-modify-write operations. We generate 10 million tuples for each table of Smallbank and use 32 warehouses for TPC-C. YCSB and Smallbank are partitioned by primary keys and TPC-C is partitioned via warehouse ID.

Mixed Concurrency Control We use YCSB to evaluate the performance benefits of mixed concurrency control under different numbers of cross-partition transactions and record conflicts rates. Key access is skewed using Zipf with *theta* = 1. We vary the number of cross-partition

[1]https://golang.org/

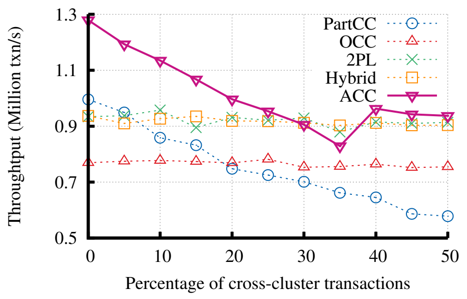

Figure 7: Increasing the percentage of cross-cluster transactions

transactions in two ways, which is shown in Figure 5. The workload is initially well partitionable, which means there is no cross-partition transactions. Note that each partition is also a cluster here. Then, we vary the number of non-partitionable partitions (e.g. P4-P6 in Figure 5). Within non-partitionable partitions, the percentage of cross-partition transactions is 100%. Therefore, these partitions (e.g. P4 to P6) can be merged into a cluster along with their respective cores. Next, we increase the percentage of cross-cluster transactions to make transactions access records in different clusters and potentially protocols.

We first compare the throughput of ACC with that of single protocols and Hybrid using the workloads in Table 1. For highly conflicted workloads we use 50% read-modify-write operations and for lowly conflicted workloads we use 100% read operations. For the partially partitionable workload, it includes 16 non-partitionable partitions and the other 16 partitions have no cross-partition transactions. Figure 6 shows the results. We see that ACC has almost the highest throughput in all workloads. For WL1 and WL2, it adopts PartCC since the workload is well partitionable. For non-partitionable workloads (WL5 and WL6), it uses 2PL and OCC respectively according to the record conflict rates. Finally, for partially partitionable workloads (WL3 and WL4) ACC outperforms other protocols due to mixing candidate protocols. Specifically, it can apply PartCC to the part of the workload that is partitionable and OCC or 2PL to the part of the workload that non-partitionable based on how conflicted the workload is. ACC has up to 3.0x, 2.1x, 1.8x, and 1.8x higher throughput than PartCC, OCC, 2PL, and Hybrid respectively.

Next, we increase the percentage of cross-cluster transactions. The test starts with the partially partitionable and highly conflicted workload (i.e. WL3). The result shown in Figure 7 demonstrates how the throughput of ACC varies with more cross-cluster transactions involved. At first, ACC mixes PartCC and 2PL by using PartCC to process the well-partitionable workload part and 2PL to process the rest of the highly conflicted and non-partitionable workload. With more cross-cluster transactions introduced partition conflicts increase; thus, ACC adopts 2PL for the whole system. Note that although ACC uses 2PL, it has slightly higher throughput than 2PL because it leverages the partial partitionability to avoid some conflicts as lock contention across cores is reduced due to clustering.

Adaptivity We then evaluate ACC's adaptivity in response to selecting a single protocol for the entire data store according to workload variation. If PartCC is used, the whole store is partitioned such that each CPU core owns a

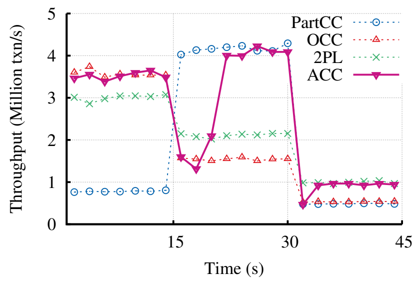

Figure 8: Adaptivity Test (YCSB)

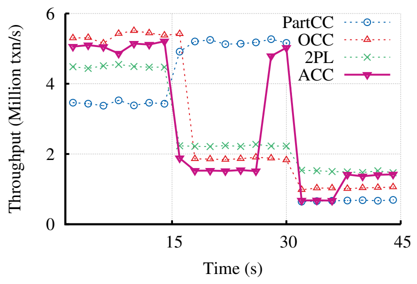

Figure 9: Adaptivity Test (Smallbank)

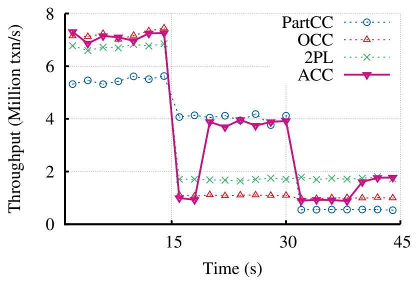

Figure 10: Adaptivity Test (TPC-C)

Table 2: Prediction Accuracy and F1 Score

	YCSB		Smallbank		TPC-C	
	Zipf	H/C	Zipf	H/C	Zipf	H/C
Accuracy	0.9	0.89	0.94	0.83	0.92	0.97
F1 Score	0.86	0.85	0.98	0.94	0.8	0.94

partition (i.e. a cluster). We currently generate partitioning plans offline. If OCC or 2PL is used, the partitioned stores are merged back into a single cluster (i.e. shared store). We use YCSB, Smallbank, and TPC-C to test ACC over the same workload variation in Figure 1. For the workloads with low and high record conflicts, we use read-only transactions (e.g. OrderStatus in TPC-C) and write-intensive transactions (e.g. NewOrder and Payment Mix in TPC-C) respectively. To generate non-partitionable workloads, we use 100% cross-partition transactions and skew partition access distribution (i.e. Zipf with *theta* = 0.3). Skewed partition access means a subset of partitions receive more transactions than other partitions. Partitionable workloads have no cross-partition transactions, as well as no skewed partition access. For all tests record access distribution is Zipf with *theta* = 1.5.

Figures 8–10 show the test results of how overall throughput evolves for three benchmarks. We see that ACC can adaptively choose the optimal protocol for the three workloads. Specifically, ACC starts with OCC for the workload that is not partitionable and has low record conflicts. When the workload becomes well-partitionable and has high record conflicts, ACC switches from OCC to PartCC, where it needs to partition the whole store first. ACC continues to process transaction requests during the partitioning and switches to PartCC when partitioning is done. Then, more partition conflicts are introduced in the workload; thus, ACC merges the partitioned store and switches from PartCC to 2PL. The dip in performance for ACC during workload shifts is due to lag while using the prior protocol and a short period where workers reorganize clusters and indices. The throughput improvement of ACC over PartCC, OCC, and 2PL can be up to 4.5x, 3.5x and 2.3x respectively, and ACC can achieve at least 95% throughput of the optimal protocol when it uses this protocol, demonstrating that the overhead of ACC is minimal.

Prediction Accuracy To test the accuracy of our predictive model, we train our model using a Zipf distribution and generate test cases using Zipf and Hot/Cold distribution with various mixes of stored procedures. Note that we only evaluate test cases that are not included in the training of our predictive model. For example, we use different *theta* values to train and test the model. We run each test cases for all protocols, extract features from our prototype, and label the test case with the protocol having the highest throughput. Then, we test how accurately our model can predict the optimal protocol using the extracted features. Table 2 shows the prediction accuracy and F1 scores using Zipf and Hot/Cold distribution in three benchmarks. We generate 1000 test cases for each benchmark. We see that the prediction accuracy (i.e. the ratio between the number of cases ACC predicts the optimal protocol over the total number of test cases) ranges from 83% to 97% and F1 scores is from 80% to 98%. This indicates that our predictive model can be accurate even under distributions that have never been trained before.

These preliminary results demonstrate the promise of ACC's ability to maximize throughput in the presence of varied workloads.

7. RELATED WORK

A large body of work exists relating to transactional database systems. Here, we discuss a limited set of projects related to partitioning, optimizing protocols for main-memory systems, and mixed concurrency control protocols.

Data Partitioning Many existing projects target generating database partitions. Andreev et. al. proposes a method using Linear Programming [3] to generate k-way balanced partitions. Schism [6] models the relationship between database tuples as a graph then use METIS [9] to partition it. Similarly, SWORD [12] uses hypergraphs to partition large databases with replication. To deal with large dynamic datasets, Leopard [7] proposes an online partitioning method based on scoring. These approaches require the number of expected partitions to be known in advance. However, this constant is not given in ACC but determined online according to the workload. In addition, ACC requires dynamic core allocation and considers the effect of multiple form of concurrency control, which makes the problem different. Recent efforts have explored partitioning when the number of partitions can expand or contract based on heuristics. E-Store [17] handles partitioning by using a two-tiered approach where hot records are partitioned separately from large cold blocks. Clay [15] incrementally partitions tuples based on affinity between records and the load capacity of underlying partitions.

Optimizing Single Protocols Several research projects focus on optimizing concurrency controls for main-memory databases. H-Store [8] proposes a partition based concurrency control using coarse-grained locking to minimize concurrency control overhead (e.g. latching and locking). Hekaton [10] improves the performance of 2PL and OCC by using multi-version based lock-free hash tables for main-memory databases. Silo [18] develops an OCC protocol that achieves high throughput by avoiding all centralized contention points among CPU cores. Doppel [11] proposes a scalable concurrency control protocol for communicative operations under high contention workloads. Tictoc [22] extracts more concurrency by lazily evaluating transactions' timestamps. These projects mainly consider optimizing a single concurrency control and are orthogonal to ACC.

Mixing Multiple Protocols Several projects mix multiple protocols in single database systems. Callas [20] provides a modular concurrency control mechanism to group transactions and provide customized concurrency control for each group. To enforce isolation across groups, each record is associated with a lock and transactions need to acquire these locks before any record operations. This has greatly introduced extra cost to original protocols especially for protocols trying to minimize locking overheads (e.g. PartCC and OCC), while our design allows a protocol to keep its execution unchanged. Meanwhile, Callas does not cover mixing logging and recovery algorithms. Hsync and MOCC [16, 19] adopt a hybrid 2PL and OCC concurrency control. ACC considers more protocols and supports adaptively switching among them.

8. CONCLUSION

In this paper, we introduce Adaptive Concurrency Control (ACC), a main-memory database system for many-core machines. It supports clustering a data store and dynamically selecting concurrency control protocols for each cluster. We outline new key challenges in data clustering, workload modeling, mixing protocols, and evaluate a prototype demonstrating promising preliminary results. We believe incorporating multiple state-of-the-art concurrency control protocols into one single database systems and adaptively clustering the data will address the critical challenge of varied and mixed OLTP workloads in the presence of many cores and massive main-memory systems.

Acknowledgments

The authors thank Neha Narula and Mike Stonebraker whose insights and discussions with lead to the initial ideas for ACC. The authors would also like to thank the reviewers, Zechao Shang, Sam Madden, and Mike Franklin who provided valuable feedback. This work was in part supported by the Intel Science and Technology Center for Big Data.

9. REFERENCES

[1] Scikit-Learn. http://scikit-learn.org.

[2] TPC-C. http://www.tpc.org/tpcc/.

[3] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[4] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[6] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[7] J. Huang and D. J. Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551, Mar. 2016.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[9] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[10] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[11] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, Broomfield, CO, Oct. 2014. USENIX Association.

[12] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *EDBT*, 2013.

[13] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1583–1598, New York, NY, USA, 2016. ACM.

[14] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 145–156. VLDB Endowment, 2013.

[15] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10:445–456, December 2016.

[16] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng. Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 463–478, New York, NY, USA, 2016. ACM.

[17] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[18] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[19] T. Wang and H. Kimura. Mostly-optimistic

concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, Oct. 2016.

[20] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, New York, NY, USA, 2015. ACM.

[21] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[22] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.

[23] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477, Berkeley, CA, USA, 2014. USENIX Association.