2015 IEEE International Conference on Big Data (Big Data)

PANOPTICON: A lock broker architecture for scalable transactions in the datacenter

Serafettin Tasci and Murat Demirbas
Computer Science & Engineering Department
University at Buffalo, SUNY

***Abstract*—For datacenter applications that require tight synchronization, transactions are commonly employed for achieving concurrency while preserving correctness. Unfortunately, distributed transactions are hard to scale due to the decentralized lock acquisition and coordination protocols they employ. We investigate the use of a centralized lock broker architecture to improve the efficiency/scalability for distributed transactions, and present the design and development of such a framework, called PANOPTICON.**

Panopticon achieves efficiency/scalability by divorcing locks from the data items and migrating locks to improve lock access locality. More specifically, the lock broker mediates the access to data shared across servers by migrating the associated locks like tokens, and in the process learns and improves the access locality of transactions.

Our experiments show that Panopticon performs better than distributed transactions as the number of data items and number of servers involved in transactions increase. Moreover, as the history locality (the probability of using the same objects in consecutive transactions) increase, Panopticon's lock migration strategies improve lock-access locality and result in significantly better performance. Finally, we also show that, by employing simple learning techniques, the broker can further improve the lock access locality and, hence, the performance of distributed transactions.

I. INTRODUCTION

Concurrent execution is a big challenge for distributed systems programming and datacenter computing in particular. For embarrassingly parallel data processing applications, concurrency is boosted and scalability is achieved by adding more servers, as in MapReduce [12]. However, for applications that require tighter synchronization, such as large-scale graph processing [14], [21], distributed scientific simulations [5], and backends of large-scale web-services, boosting concurrency in an uncontrolled/uncoordinated manner wreaks safety violations. In these applications, concurrent execution needs to be coordinated to prevent latent race conditions and synchronization bugs.

Transactions offer a nice abstraction for distributed systems developers as they provide serializability guarantees while allowing concurrency under the hood. Since transactions are easy to use, they are employed by modern distributed systems including Google Spanner [9], Google Megastore [4], and AWS RDS [2]. Unfortunately, transactions have some shortcomings. We identify the following two issues as the main bottlenecks for the scalability of distributed transactions.

Distributed transactions waste a lot of time in coordination. When several data items need to be locked at the same time, test-and-set approaches become inapplicable. Distributed coordination for setting locks (such as two phase locking and two phase commit) do not scale as their costs grow quickly with respect to the number of servers involved in the transactions [3].

Distributed transactions also waste a lot of time due to repetitive remote/nonlocal accesses. There is a big latency difference between accessing a server-local item versus item that is stored across the cluster. However, modern datacenter computing systems focus on consistent hashing and load-balancing of data to the servers and ignore access locality when assigning data for storage. When locks are coupled and tied to the data, this penalizes transaction latencies severely, especially in the lock acquisition phase. Even after several repetitions of the same transaction, each time the remote lock-access costs are paid again and again.

Contributions. To address the above scalability problems in transactional systems, we propose a lock broker architecture, called PANOPTICON. Panopticon achieves scalability by divorcing locks from the data items and tasking the broker with caching hot locks and migrating certain locks to servers to improve lock access locality.

Differing from centralized lock service solutions such as Chubby [6] or Zookeeper [17], Panopticon does not keep all the locks in the broker. Maintaining all the locks at the broker is not desirable since it kills all opportunities for local access: none of the servers can have local transactions in that setup. Instead Panopticon employs the broker as a cache for locks that receive across-server access. This way the broker also gets to observe transaction access patterns so that it can improve lock access locality of transactions by migrating locks when appropriate. If a lock gets consecutive accesses from one server, the broker migrates the locks to that server to improve lock-access locality further. Moreover, the broker can also learn some transaction access patterns and can proactively migrate locks to the servers even before they are requested by those servers.

We present Panopticon's novel lock broker architecture in Section II, where we detail the broker operations, lock migration rules, and optimizations such as batch locking, and lazy unlocking. We develop and build Panopticon leveraging the Hazelcast [16] platform, a popular lightweight open-source in-memory data grid for Java. Hazelcast uses

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

traditional distributed transaction processing with decentralized two phase locking protocol. We build on Hazelcast to implement the Panopticon lock broker architecture. We present our implementation of Panopticon in Section III and then use this implementation to compare and contrast Panopticon's improvements over traditional distributed transaction processing in Section IV. We discuss some design decisions and extensions/improvements to Panopticon in Section V. We compare and contrast Panopticon with other work on distributed transaction processing in Section VI.

We make the code for Panopticon as well as the TLA+ [19] specification for the lock-broker accessible freely at https://github.com/serafett/panopticon.git.

II. Panopticon Lock Broker

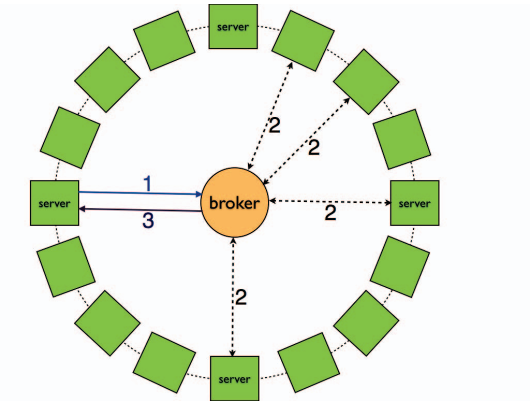

Figure 1. Panopticon lock broker

Panopticon maintains a lock for every data item, which can be any record in a key-value store. By default, the lock is kept where the data is, and this is advantageous for improving lock-locality and enabling server-local transactions without the need for contacting the broker. The server contacts the lock broker only if data at other servers needs to be accessed as part of a transaction.

The lock broker coordinates across-server sharing of the data in distributed transactions by migrating the corresponding locks like tokens. A server can request a lock anytime from the broker, and only from the broker. The broker gives the locks in a first-come first-serve manner. If the requested locks are at the broker, the broker responds immediately. Otherwise, the broker requests the locks from the corresponding servers first and forwards them to the requester when they are made available. The broker can request a lock back anytime from a server, and the server complies. (If the server is currently using the lock in an executing transaction, the server returns the lock after the transaction completes.) As the centralized authority for mediating access to data, the broker learns about the access patterns of transactions at runtime and manages the migration of locks to servers in a way that improves lock access locality as we discuss in Section II-A. We discuss transaction initiation and completion at the servers in more detail in Section II-B.

Note that when transactions involve multiple data items, this centralized lock broker solution gains an edge over the traditional distributed transaction processing. Decentralized distributed transactions employ two phase locking to prevent deadlocks, which require that the server initiating the transaction to contact the other servers for locks serially in a fixed order (in order to avoid deadlocks). Thus a decentralized distributed transaction execution that involves K other nodes completes at K round-trip times. Instead of this, it is more efficient to go to the broker and test/set all the locks at once when K is greater than 1. This is because a transaction execution via the lock broker complete in 2 round-trip times, regardless of how many other servers are involved in the transaction. In Panopticon, the lock request is sent at once to the broker, and the broker orders the K locks in parallel, so this takes only one round trip. The broker takes care of deadlock prevention since the broker serializes the transaction request order among transactions. We discuss this in more detail in Section II-C.

A. Tradeoffs and lock migration

Storing all the locks at the broker is not desirable because it kills all the lock-access locality for the servers. On the other hand, not storing any locks at the broker makes it slow for a server to request a lock it needs. Panopticon strives to find the sweet point in this tradeoff spectrum. We notice that there can be three types of locks hosted at the broker:

1) locks that receive across-server accesses,
2) locks that receive repetitive access from same server,
3) locks that receive no access for a long-time.

It is best to host *type 1* locks (locks that keep receiving across-server accesses) in the lock broker. And it is best to assign the *type 2* locks to the requesting server to avoid the overheads of repetitive requests from that server to the broker. We discuss the determination of the sweet point in the tradeoff between *type 1* and *type 2* locks next. [1]

In Panopticon, the broker gets to observe all transaction access patterns at runtime so it can differentiate between *type 1* and *type 2* locks given some rules for cut points. We use the following rule of thumb for declaring a lock to be of *type 2* and migrating that lock to a server: If k consecutive requests for a given lock l (held at the broker) comes from the same server w, then the broker migrates lock l to server w. From that point on, w treats l as its local lock. The lock locality of w is improved with this move, since w does not need to contact the broker for l again.

Note that this is not a permanent assignment. Later if another server y requests l, the broker migrates l back to

[1] For space-saving at the broker, we can employ least recently used (LRU) policy to expel *type 3* locks (locks that have not seen an access for a long-time) back to the original host (the server that hosts the corresponding data item).

itself, and gives y access to the lock. At this point, l is treated again as a *type 1* lock. When y is done with l, l is continued to be hosted at the broker (that is, until the k-consecutive rule is satisfied and l is migrated to another server). In our experiments, we observed that $k = 2$ is a good choice for most scenarios.

B. Transaction execution at the servers

The broker is oblivious to the state of the transactions. The broker maintains per lock information, but not per transaction information. The servers are the transaction managers and they maintain the transaction information. Transactions are initiated and executed by the servers distributedly after checking that all the locks are available at the server.

When a server initiates a transaction, it requests locks in batch as we explain in the next subsection. The server, as the transaction manager, is responsible for determining when to enter the transaction. When the server checks and finds that it has gotten all the locks, it starts the transaction and *authoritatively-owns* those locks, deferring the requests for these locks until the transaction ends. Within the time frame of initiating the transaction and entering the transaction, the server may have some of the locks but it may not authoritatively-own all those locks. For example, if a server requires locks of multiple data items $\{l_1, l_2, l_3, .., l_n\}$ for a transaction, it cannot authoritatively-own l_i until all data items l_j such that $l_j < l_i$ are locked in the current transaction. If the server is waiting for lock l_3 and has l_1, l_2, l_4; while l_1, l_2 are authoritatively-owned by the server, l_4 is not authoritatively-owned by the server. Therefore if another server asks for l_4, the server will have to return l_4 to the broker and then the broker will forward it to the requesting server. If the requesting server already authoritatively-owns l_3, it will authoritatively-own l_4 immediately after it receives it before replying any lock request. This way of managing the locks ensures progress and guarantees that livelocks as well as deadlocks are avoided.

After a transaction is finished, the server needs to unlock the data items, which means returning the locks back to the broker. In this phase we propose an optimization where the server performs a *lazy unlock*. Lazy unlocking means that the locks are released locally, but not transmitted back to broker until δ time elapses, where δ is empirically determined. Lazy unlocking provides efficiency benefits for cases when the server needs to access the same data items used in the terminated transaction immediately in the next transaction. Recall that if the same lock is requested by the same server k times in a row, that lock is migrated from the broker to that server. Therefore lazy unlock provides benefits between the first and k^{th} consecutive requests of a lock by the server. Instead of returning the lock back to the broker only to request it back afterwards, the lazy unlock mechanism provides a grace period at the server to avoid that inefficiency. This optimization is verifiably safe because if lazy-unlocked data is requested, the lock is immediately given back to the broker.

C. Transaction serialization at the broker

In Hazelcast transactions, two phase locking is employed to prevent deadlocks. Two phase locking requires that the server initiating the transaction needs to contact the other servers for locks in a total order. The server cannot contact another server until the current requested lock is acquired. Therefore, if the transaction needs to get the locks of multiple data items, this forces lock acquisition to be serial in nature and causes a significant increase in transaction time.

In Panopticon the servers are not prone to this problem. The servers can make the requests for all locks in batch and at once, because the broker takes care of serializing requests and deadlock prevention. When a server initiates a transaction, it requests locks in batch as we explain in the next subsection. The broker coordinates assigning/delivering of the requested locks in a first come first serve basis of the transaction requests. And when processing a lock request for a transaction, the broker assigns the locks in an increasing order to prevent deadlocks. The broker sorts the lock requests to form a total order based on the data item ID. When processing in this increasing order of locks, if the broker holds the lock, it forwards it to the requesting server. If the broker does not have the lock, it adds this server's name to the request-queue of the lock, and forwards the lock requests to the server that holds the lock. This is non-blocking but the replies may arrive in different order. When a lock becomes available, the broker will forward this lock to the server that is at the head of the queue it maintains for that lock. By employing batch requests to the broker, Panopticon avoids incremental lock requests in traditional decentralized distributed transactions and gains a big advantage in transaction execution time.

To test the protocol for correctness, we modeled the lock broker with TLA+ [19] and tested for the mutual exclusion invariants and deadlock. TLA+ is a tool for specifying distributed algorithms and model checking them. A TLA+ specification simply describes the set of all possible legal behaviours (execution traces) of a system. AWS reports [23] that they recently adopted TLA for use in many of their key distributed systems, including S3, DynamoDB, EBS, and a distributed lock manager.

Modeling with TLA+ helped us get a better understanding of the correctness reasoning for Panopticon. The proof of safety specification is relatively straightforward as in any token-based mutual exclusion algorithm: There is one token per data item, and it cannot be created/destroyed, and it can belong to one server at a given time. The liveness/starvation-freedom proof, on the other hand, is achieved by projecting a total order on requests and releases: Nodes form queues on this order, and waiting nodes raise on the order and make step by step progress towards the critical section. Our TLA+

model for the Panopticon lock broker is available at https://github.com/serafett/panopticon.git.

D. Transaction Prediction

Due to access locality between transactions, applications often contain some correlations between consequent transactions. We say that there is a correlation between two data items *i* and *j*, when it is highly probable that the next transaction will include item *j* given that the current transaction uses *i*. For such workloads, time series prediction can be employed to predict the items in the next transaction by looking at the items in the current transaction.

In particular, we use a first-order Markov model for prediction of the items in the next transaction. For this purpose, the broker uses a prediction table C which keeps at location $C[i, j]$ the number of times an item j is requested in the next transaction after item i is requested in the current transaction by the same server. With each new transaction request to the broker, the counts in the table are updated. Then the probability of an item d_j to be used in the next transaction is given by the equation:

$$Pr(j \in T_{t+1}) = \frac{C[i, j]}{\sum_{k=1}^{N} C[i, k]} \tag{1}$$

Then

$$\arg\max_j Pr(d_j \in T_{t+1})$$

is selected as the final prediction candidate. To avoid the cost incurred by incorrect predictions, we approve a candidate j only if its probability is significantly higher than other candidates and $C[i, j] \gg 0$.

Transaction prediction has two main benefits: First, it helps a server to receive locks for a transaction even before requesting them. When a server receives all item locks and starts a transaction, the predicted locks for the next transaction are also sent to the server if they are available at the broker. Secondly, when the predicted item locks are received, the server can read and cache data of those items. By this way, a server can avoid the latency of data reading from distributed storage after the transaction starts.

III. Implementation

In this section, we explain how we implement Panopticon over the Hazelcast platform. Hazelcast is an in-memory data grid (IMDG) which is designed to store data in the main memories of a cluster of machines. It ensures scalability by simple, on-the-fly cluster management. A typical deployment ensures that the data is evenly partitioned across all nodes in the cluster and automatic fail-over in case of a node failure.

As stated earlier, in this paper we use Hazelcast as the implementation choice. While it shares a similar feature set with other IMDGs, Hazelcast stands out as a lightweight, open-source solution with an easy to use Java API. Using Hazelcast is as simple as adding the Hazelcast *jar* file to the classpath and writing Java programs using distributed counterparts of java.util.Queue, Set, List, Map. By using an IMDG, we shift the burden of managing data items on a distributed cluster to the underlying data management platform (i.e. Hazelcast) and we only focus on lock management for distributed transactions.

A. Broker actions and data structures

All instances (i.e. machines) in a Hazelcast cluster have unique IDs, and we set the instance with ID 0 (which is denoted as the oldest node and special node in Hazelcast) as the broker and the other instances as servers. The broker does not perform computation and is responsible solely for lock management. The Panopticon broker has the following core data structures:

1) *LockTable*: keeps the current owner for each item lock.
2) *RequestTable*: keeps track of the requests for each lock. When a lock request arrives, if the lock is already available at the broker, the broker gives the lock to the requester. Else, it inserts the requester ID to the *requestTable* and sends a request to get the lock from the hosting server (only if a request has not been sent before). The *requestTable* stores multiple requests for the same data item in a FIFO queue.
3) *LeaseTable*: keeps track of the locks which were migrated to the servers.

B. Server actions and data structures

A server keeps the list of locks it owns in *lockList*. When the server initiates a transaction and requests locks with the *lockAll()* method, it first checks the *lockList* and sends a lock request to the broker only if it does not find some of the requested locks in this list. In addition the server keeps a boolean *requestList* to keep track of the list of requests for the locks it has. Note that these data structures are Lists, compared to the Table data structures in the broker, because for these locks, the other party is clear and unique: the broker. The servers interact only with the broker and not with other servers.

Whenever a server commits and exits a transaction, it calls *unlockAll()* method which checks its *requestList* and gives the requested locks to the broker. For the locks that are not requested, the server can keep them if the broker gave a lease for the lock (i.e., migrated the lock to this server) or if the locks are originally hosted at this server.

C. Communication and messaging

Messaging between the servers and the broker is done via ITopic publish-subscribe mechanism in Hazelcast which guarantees message ordering. We use $n+1$ topics where n is the number of servers. For server-to-broker communication we use a shared channel called *toBroker*. However broker-to-server communication uses separate channels for every

$server_i$ to ensure messages are published only to relevant servers. Whenever a new message from a registered ITopic arrives to a server or the broker, the message is parsed and handled based on the message type.

There are three types of messages in Panopticon: A *request* message is used to send a lock request to the broker or server holding the lock. Of course a server cannot send request messages to other servers directly; the broker does it on behalf of the server. A *reply* message is sent, to submit the lock to the requester when the lock becomes available. (Again, these messages might only be sent between a server-broker pair, never a server-server pair.) Finally, a *lease* message is used by broker to give/cancel leases of locks. For this purpose broker keeps a consecutive request list called *conseqList* that holds the consecutive requests to data items along with the id of the requesting server.

Finally, Panopticon does not assume reliable channels and is robust against message loss. To handle message losses, Panopticon employs message reply timeouts. If a request is not answered in a specified time, the server assumes the message is lost and resends the request. If the request message is sent successfully but the reply message is lost, then the replying server understands it when an identical request is received and resends the reply. Side effects of resending are avoided since the lock request and reply messages are idempotent.

IV. Experiments and Evaluation

A. Setup

To evaluate the performance of Panopticon, we performed experiments on AWS EC2 using up to 33 medium Linux instances, which have two EC2 compute units and 3.75 GB of RAM each. In our experiments, one instance is designated as the broker, and the remaining instances are servers. In all experiments, the transaction time is measured by averaging 1000 such transactions after a warmup period.

We compare Panopticon with the decentralized two phase locking based Hazelcast transactions. Panopticon-L denotes the basic Panopticon framework with lazy unlocking optimization that we described in Section II-B. In this Panopticon implementation, to save messages, we made the broker batch lock-replies and grant them together in one message to the server. We also performed experiments with Panopticon-S, which improves Panopticon with read staging. In Panopticon-S the broker sends lock-replies to the server as locks become available. And when a server receives a lock, it reads the data item immediately to stage a copy of the data-item at its cache before it receives all the locks and enters the transaction. By this way, it is possible to avoid delays incurred by costly distributed reads during a transaction. Panopticon-LS denotes the Panopticon-S version with lazy unlocking optimization.

We measure the effect of several parameters in Panopticon. History probability, Pr_Hist, denotes the probability of using the same objects in consecutive transactions. For example $Pr\_Hist = 0.7$ means that if a transaction uses 100 shared objects, 70 of them would be among the objects accessed by the previous transaction from the same server. Lease acquisition threshold, N_conseq, is used to determine the number of requests for giving a lease (i.e. migrating the lock). If the same server makes N consecutive requests to an object without any other server requesting it in the meanwhile, then the lease of the object is given to the server and the server can keep it until another request comes. Otherwise, it immediately returns the lock to the broker after its transaction finishes. $TxnSize$ denotes the number of data items accessed in a transaction. In all experiments, one read and one write operation is performed on each data item in a transaction. $TotalSize$ denotes the total number of shared data items between transaction servers. Note that decreasing $totalSize$ increases the contention in a workload. Therefore, despite we tested Panopticon with up to 64K shared data items, in these experiments we keep $totalSize \leq 16K$ to reveal the performance of Panopticon under contention.

B. Experiment Results

In our first set of experiments, by varying Pr_Hist, we measured the effect of the probability of selecting the same items in consecutive transactions. Since Hazelcast does not exploit the history of accesses, the results remain stable in Hazelcast regardless of the history probability. Figure 2.a shows that when the number of data items in a transaction is low ($txnSize = 10$), Panopticon performs better than decentralized transactions even with very low history probability. As the history probability increases, Panopticon benefits more from lock migration and data staging which causes the performance gap between Panopticon and decentralized transactions to increase. When the number of objects in a transaction is high ($txnSize = 100$), despite the increase in contention, the superiority of Panopticon becomes even more significant (Figure 2.b). In addition, we observe that while lazy unlocking (Panopt-L) helps more than data staging (Panopt-S), combining both optimizations (Panopt-LS) has an aggregated benefit.

In our second set of experiments, we evaluated the effect of N_conseq, the required number of consecutive requests for lease acquisition. When $N\_conseq = 1$, servers will keep the locks they acquired unless they receive a new request for this lock. When N_conseq becomes very high, servers will probably never attain enough consecutive requests to acquire the lease of the lock. As a result, locks will always be returned to the broker after a transaction completes in a server.

In Figure 3.a and Figure 3.b, we see the effect of N_conseq for history probabilities of 0.1 and 0.9 respectively. When $Pr\_Hist = 0.1$, setting the lease acquisition threshold to 1 will significantly hurt the transaction performance for all Panopticon variants. In this case, since servers

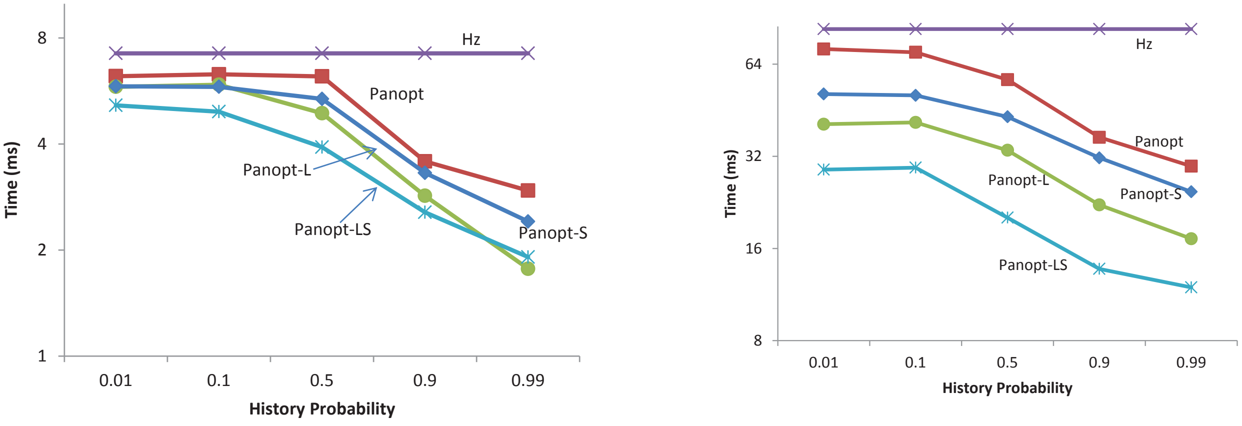

Figure 2. Change in time as the history probability increases with txnSize=10 (left) and txnSize=100 (right) using 4 servers and 1024 total data items

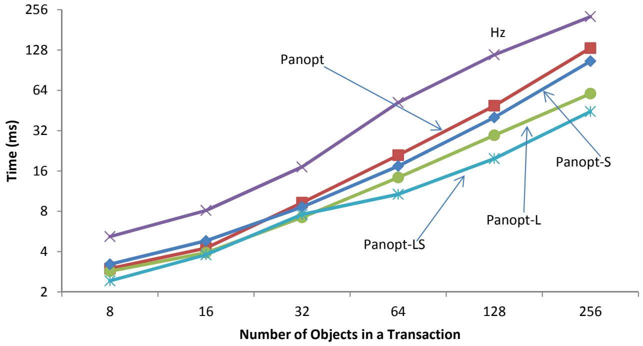

Figure 4. Comparison of Panopticon with decentralized locking as the number of locked items in a transaction changes

will not return the locks after transactions, whenever a new transaction starts, they will not be able to find the locks at the broker. Therefore, for most locks the cost of getting the lock will be 2 round-trip times. When $N\_conseq \geq 2$, this problem disappears since the servers will not be able to get any lease now.

On the other hand, when $Pr\_Hist = 0.9$, since servers will mostly have consecutive accesses to data items, earlier migration of locks to servers is desired. Therefore when N_conseq gets bigger, more locks are returned to the broker and performance deteriorates. servers will continue to access the same objects for more consecutive transactions. Note that Panopt-L and Panopt-LS always keep a good performance regardless of N_conseq, since both methods use the lazy-unlock optimization. With this optimization even when a server does not have the lease, by waiting until the next transaction starts, it avoids unnecessary returning of the locks to the broker.

In Figure 4, we varied txnSize, the number of data items accessed in a transaction, to see the effect of contention. Naturally, as the number of locked items increase, the duration of the transactions and also the contention for locks increases in all methods. However, as Figure 4 shows, Panopticon scales better than decentralized locking due to its batch locking and non-busy lock holding capabilities. In addition, in all experiments we have consistently observed that lazy unlocking and read staging optimizations improve the performance of Panopticon. This improvement becomes more evident as the number of locked objects in a transaction increases.

To evaluate the scalability of Panopticon, we kept the number of objects in a transaction fixed and measured the average transaction time with an increasing number of servers. While in Figure 5.a each transaction uses 4 data items from a pool of 1024 total items, Figure 5.b uses 4x more items per transaction from a 16x larger item set. In both figures, since the total number of objects is constant, the lock contention among servers increases significantly as more servers are employed.

Figure 5 shows that Panopticon performs remarkably well when the contention is low thanks to the data and lock caching mechanisms employed in Panopticon. When the number of servers increases, Panopticon scales linearly preserving its edge over decentralized locking all the time. In this figure, we did not include Panopticon and Panopticon-L since they are consistently outperformed by Panopticon-S and Panopticon-LS, that use read staging.

Finally, Figure 6 shows the effect of prediction on Panopticon's transaction performance. For this experiment, we used a specific workload in which for every item i, there is an item j such that probability of requesting j is significantly higher than other items in the next transaction if the current

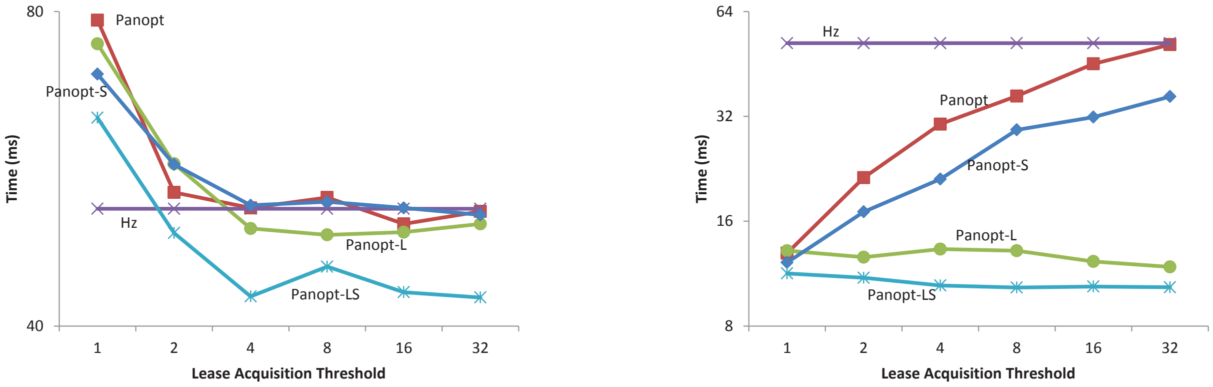

Figure 3. Change in time as the lease acquisition threshold increases with $Pr\_Hist = 0.1$ (left) and $Pr\_Hist = 0.9$ (right) using 4 servers and 1024 total data items

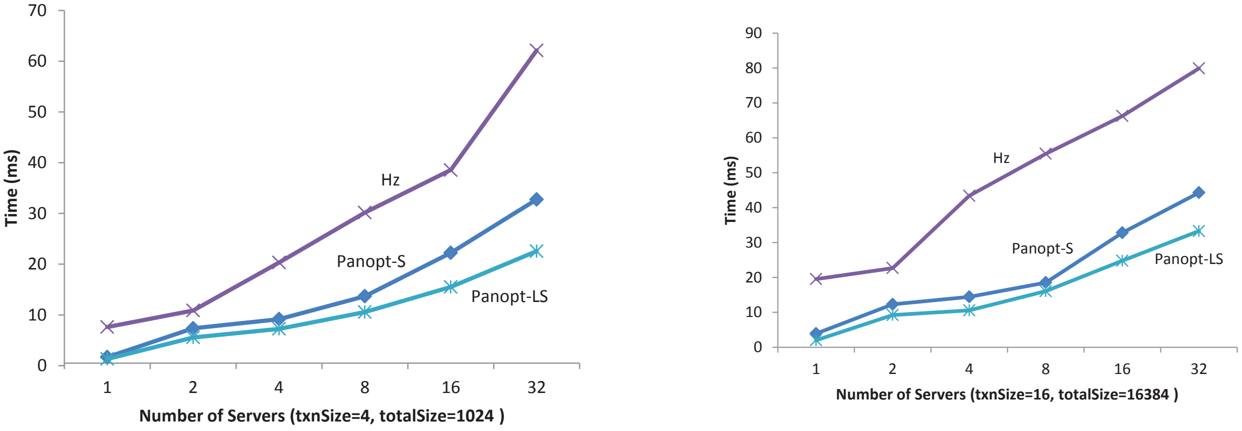

Figure 5. Scalability of Panopticon as the number of servers in the cluster increases

transaction includes i. The results show that prediction consistently improves transaction completion times.

Optimizations like lazy unlocking cannot replace prediction since they assume repeated access to same data items while prediction only requires a data access pattern in time. In other cases where there is no such pattern, using prediction may even hurt the performance of the system due to incorrect movement of locks.

These experiments also show that the single lock broker in Panopticon can scale well, since it does not keep track of transaction state information and it is not contacted for heavy-weight operations.

V. Discussion

By leveraging the simplified topology of the Panopticon, fault-tolerance to partitions can be added to the system. While there is no global agreement on a partition in a general distributed system setup (some nodes may detect a partition while others do not), the Panopticon setup simplifies the partition detection and handling significantly. Detection is very simple and there are no dissenting opinions on it, since it is dictated by the broker. Furthermore, unlike a general distributed system setup which is prone to arbitrary shaped partitioning, the partitions in the Panopticon are always simple and uniform. There are only 2 cases for a partition: 1) the broker is in the partition, and 2) the broker is not in the partition. We call the partition with the broker as the main partition. If the broker is not in the partition, this means the

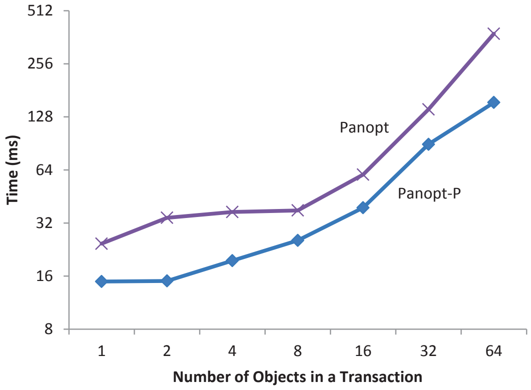

Figure 6. Effect of prediction on Panopticon performance (16 servers, 16384 total items)

server is partitioned away as an isolated single node and we call this as a single partition.

Similarly, in order to improve congestion control in Panopticon, the broker can also employ some simple rules. Since the broker gets to observe every across-server transaction request, it can notice when contention is increasing by just monitoring its *requestTable* entry queues. In future work, we will consider how to use this information to take corrective actions.

Finally, in order to achieve more scalability and avoid bottlenecks when using extremely large number of servers and locks, we plan employ a hierarchical composition of the brokers. For this we have k level-0 lock brokers each overseeing a cluster of servers and a top-level lock broker overseeing these k lock brokers. In this case, the level-0 brokers take on the role of servers for the top-level broker, and take on the role of tracking information for across cluster (i.e., across level-0 broker boundary) transactions.

Using hierarchical composition extends scalability of Panopticon. With such a setup, we can make Panopticon manage extremely large lock spaces which may not fit into the memory of a single broker. Moreover, using hierarchical composition of brokers at different datacenters, the Panopticon system can provide a partial answer to the across-datacenter/WAN transactions problem. Providing an efficient and complete system for across-datacenter transactions remains part of our future work.

VI. Related Work

A. Lock services

Google Chubby [7] is a centralized lock service that provides an interface similar to a distributed file system with advisory locks. Chubby depends on manual locking from the developers and is prone to the disadvantages of locking

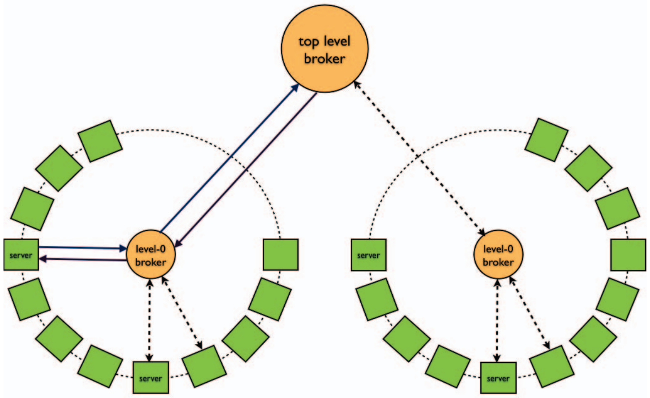

Figure 7. Hierarchical composition of Panopticon lock brokers

approaches. To keep the load light, Chubby provides coarse-grained locks instead of finer-grained locks. This locking scheme is more appropriate for loosely-coupled distributed systems: The Google File System [15] and BigTable [8] use Chubby as a lock service. ZooKeeper [17] is an opensource clone of Chubby.

The lock token idea has been employed in the distributed filesystems domain [22], [24]. GPFS [24] employs a centralized global lock manager in conjunction with local lock managers in each file system node, where the global lock manager can lease locks to local lock managers.

Differing from centralized lock servers, the Panopticon lock broker does not maintain all the locks, and rather it is a cache for locks that receive across-server access requests. Differing from previous work on lock brokers, Panopticon lock broker supports distributed transactions on multiple objects.

B. Transaction processing

Single-key transactional support. Since distributed transactions are costly and fail to satisfy the scalability requirements of web applications, several system designs have sacrificed the ability to support distributed transactions in lieu of supporting single key/object transactions in a statically partitioned setup. ElasTraS [10] provides ACID guarantees for transactions that are limited to a single object and single partition.

Limited multi-key transactional support. Since several applications requires collaboration, scalable and consistent multi-key access is critical for them. Google Megastore [4] and Megastore defines "entity groups" to partition the distributed datastore and provides ACID semantics to multi-key transactions that are confined within a predefined entity group. Megastore still has a limit of "a few writes per second per entity group" because higher write rates will cause even worse performance due to the conflicts and retries of the multiple leaders of the Paxos protocol employed for performing transactions. Many applications in Google used

Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. Examples of well-known Google applications that used Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine.

Limited wider transactional support. Relaxing the static entity groups restriction, Gstore [11] allows dynamic group formation. Key grouping requires a two-phase locking protocol which is a costly protocol, and Gstore prohibits transactions across these formed groups. To provide transactions over a distributed key-value store, Scalaris [25] employs Paxos. Similarly, CloudTPS [28] employs two-phase commit protocol to implement transactions over a distributed key-value store. CloudTPS makes the assumption that applications access only a few partitions in any of their transactions.

Sinfonia [1] provides multi-key transactional support by limiting the allowed operations in a transaction to support only a small subset of compare, and conditional read/write operations on the memory nodes. These "minitransactions" tradeoff expressivity of transactions with improved performance. The "ordering transactions with prediction" paper [13] proposes a similar architecture to Sinfonia, but address transactions that can have conflicts. Instead of using locks, they use OCC transactions, and suggests a prediction based ordering of them in advance (making reservations at the Object Managers), in order to reduce abort rates of transactions.

General distributed transactions. Recently a number of systems attempted to provide general unrestricted transactions. H-Store [18] partitions the database in to disjoint subsets that are assigned to a single-threaded execution engine assigned to one core on a node. H-Store's scalability relies on careful data partitioning across executor nodes, such that most transactions access only one executor node. Deuteronomy [20] introduces a distributed database architecture that emphasizes decoupling of transactional component from the data component. Calvin [27] employs a deterministic ordering guarantee to reduce the prohibitive costs associated with distributed transactions.

Spanner [9] is Google's multiversion distributed database that allows distributed transactions. Spanner employs Paxos at coordinators and two-phase commit across coordinators and uses accurate timekeeping with tightly synchronized atomic clocks as a means to improve the performance of distributed transactions. The coordinators manage and coordinate locks for data items by maintaining lock lists. Since many coordinators need to be coordinated for serialization of distributed transactions, two phase commit is employed for coordinating the coordinators. While distributed coordinator transactions using two phase commit inevitably take their toll, the Spanner team believes "*it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions*". Applications that use Spanner, such as Google's F1 advertising backend [26], can specify which datacenters contain which bits of data so that frequently read data can be located near users to reduce write latency.

Panopticon as compared to previous work. Most of the systems described above rely on some form of limitation on transactions that allows for an acceptable performance. Panopticon keeps things simple with a lock broker architecture, and eschews costly protocols for distributed coordination. As a result, Panopticon does not limit transactions and allows arbitrary multi-key/object transactions. Also different from these existing work, Panopticon divorces locks from the data items in an effort to improve lock access locality. Finally, different from the systems described above, Panopticon learns the access pattern of transactions on-the-fly and adaptively migrates locks and data items in order to improve access/lock locality in the system.

VII. Conclusion

Panopticon achieves scalability by divorcing locks from the data items and striving to improve lock access locality. The lock broker mediates the access to data shared across servers by migrating the associated locks like tokens, and in the process gets to learn about the access patterns of transactions.

We implemented Panopticon leveraging the Hazelcast in-memory data grid platform. Our experiments demonstrated Panopticon's improvements over Hazelcast's distributed transactions. The lock broker architecture performed significantly better as the number of data items and number of servers involved in transactions increase. This is because it is more efficient to go to the broker and test/set all the locks at once, instead of contacting other servers trying to acquire locks in increasing order in a serial manner. Also as the history locality (the probability of using the same objects in consecutive transactions) increase, Panopticon's lock migration strategies improved lock-access locality and resulted in significantly better performance.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[2] Amazon web services.". http://aws.amazon.com/rds.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *VLDB*, 2015.

[4] J. Baker, C. Bond, J. Corbett, JJ Furman, A. Khorlin, J. Larson, JM Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. *CIDR*, pages 223–234, 2011.

[5] S. Braun. A cloud-resolving simulation of hurricane bob (1991): Storm structure and eyewall buoyancy. *Mon. Wea. Rev.*, 130(6):15731592, 2002.

[6] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] J. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

[10] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2009.

[11] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, page 13, 2004.

[13] I. Eyal, K. Birman, I. Keidar, and R. van Renesse. Ordering transactions with prediction in distributed object stores. *LADIS*, 2013.

[14] Facebook graph search. http://www.facebook.com/about/graphsearch/.

[15] S. Ghemawat, H. Gobioff, and S-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37/5, pages 29–43. ACM, 2003.

[16] Hazelcast, in-memory data grid. http://www.hazelcast.com/.

[17] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[19] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[20] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[22] A. Mohindra and M. Devarakonda. Distributed token management in calypso file system. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 290–297, 1994.

[23] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. Use of formal methods at amazon web services. 2013.

[24] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[25] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48. ACM, 2008.

[26] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *VLDB*, 6(11):1068–1079, 2013.

[27] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[28] W. Zhou, G. Pierre, and C-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, pages 525–539, 2012.