Adaptive Quorums for Cloud Storage Systems

(extended abstract of the MSc dissertation)

Pilana Withanage Gayana Ranganatha Chandrasekara
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

Abstract—**Cloud storage systems rely on replication for reliability. Typically, each data object is stored in multiple nodes to ensure that data remains available in face of node or network faults. Quorum systems are a practical way to ensure that clients observe consistent data even if some of the replicas are slower or unavailable. Previous work has shown that the performance of a quorum based storage system can vary greatly depending on the workload characterisation and that significant gains can be achieved by carefully selecting the size of write and read quorums. In this work we are interested in multi-tenant storage systems, that are faced to heterogeneous works. In these systems, for optimal performance, different quorums may need to be applied to different data. Unfortunately, keeping different quorum systems for different objects significantly increases the amount of metadata that the storage system needs to manage. The challenge is to find suitable trade-offs among the size of the metadata and the performance of the resulting system. The thesis explores a strategy that consists in identifying which tenants and/or objects are the major sources of bottlenecks in the storage system and then performing fine-grain optimization for just those objects, while treating the rest in bulk. We have implemented a prototype of our system and assessed the merits of the approach experimentally.**

I. Introduction

The number of cloud-based applications that require the storage of large data sets keep increasing. Notable examples include Google, Facebook, Twitter, among many others, but this trend is common to many other applications. As a result, distributed storage systems are a fundamental component of any cloud-oriented infrastructure and have received a growing interest, with the emergence of many private, commercial, and open source implementations of this abstraction. Apache Cassandra [1], Amazon Dynamo [2], Openstack Swift [3] are some relevant examples of cloud storage systems that aim at providing high availability and throughput to the applications.

Most of the cloud storage systems, such as the ones listed above, have several configurations parameters that need to be carefully tuned for optimal performance. Naturally, the optimal configuration depends not only on application specific requirements (such as the intended reliability) but also of the workload characterization, such as the size of objects being read or written, distribution of the accesses to different objects, read/write ratio, among others [4], [5], [6], [7]. Furthermore, some of these factors may change in time, making hard of even impossible to resort to static or manual configuration. The dynamic adaptation of cloud storage systems, in face of observer changes in the workload, is therefore a challenging task that needs to be embraced [8].

It is possible to find in the literature several works that address the dynamic adaptation of cloud storage systems. Some examples include the dynamic adjustments of the replication scheme of objects based on observed changes to the read/write access patterns [4], automated adaptation of the replication protocol based on current operational conditions [5], dynamic data replica placement to improve locality patterns of data access with the help of machine learning techniques [6], and techniques to optimize system wide quorum sizes for both read and write operations according to the read/write ratio [7]. Still, most of these approaches consider single-tenant systems, i.e., the case where the system is optimized as a whole. However, many cloud storage systems are multi-tenant, i.e., the same storage infrastructure is shared by different applications, each with its own specific requirements and workload characterization. Ideally, one could attempt to optimize the system for each and every tenant. Unfortunately, this may be too costly and induce a significant metadata overhead. Furthermore, the optimization of tenants that have a residual share of the total system usage may have no perceived gain, not even for the tenant at hand, given the the overall system performance may be constrained by the configuration of the tenants that consume most of the systems bandwidth.

We address the dynamic adaptation of multi-tenant system by changing (object wise fine-grained adaptation) quorum configurations of popular objects detected based on the access patterns of objects in a distributed cloud storage system in order to maximise the throughput. We take Global Q-Opt which is a prototype developed for Openstack Swift by Maria Couceiro as the starting point because it is already capable of performing adaptive global quorum optimizations based on aggregated values by applying the same quorum configuration to all objects. Hence, our work (Q-Opt) extends this system by embedding state of the art hot spot detection algorithm and novel non-blocking quorum reconfiguration algorithms.

The rest of the document is structured as follows. Section II discusses the recent work performed related to our work. Section III introduces Q-Opt along with the design and implementation. Section IV shows the results of the experimental evaluation of Q-Opt and finally Section VI concludes this document by summarizing its key aspects and future work.

II. Related Work

Considerable work has been done in order to change the consistency levels of cloud storage systems dynamically and to gain a better throughput while satisfying the clients' consistency requirements. In this section we review some of the proposed solutions relevant to our work.

Application based adaptive replica consistency [9] determines the consistency of the data based on the requirements demand by the application. The system uses the read and write frequency and the variation of these two parameters as the key input matrix to trigger automated consistency adaptation. The hierarchical structure consists with one master, three deputy and many child nodes. Master node is elected in order to take care all the update operations in the system. Hence, the strong or weak consistency is defined as propagating updates to all nodes or only to deputy nodes respectively. Monitoring the frequency (high/low) of read/write operations they decide which consistency best suits for the current workload.

Harmony [10] is a system which extends the intuition of previous idea and adapts the consistency levels, by monitoring the system status in real time. To this end, Harmony considers different factors, including the data access pattern, the network latency, and the consistency requirements of the application, in particular if the application can tolerate stale reads or not. A stale read occurs when the read operation takes place after a write operation but the old value is still returned. It is developed as an external component to the Apache Cassandra cluster to gather statistics and compute the current stale read percentage of the cluster and then to decide the consistency level automatically comparing with the applications' stale read tolerance parameter (i.e This parameter needs to be specified by the user). The new consistency level is instantly communicated to the Cassandra driver component used by the application so that system adapts to fulfil applications' consistency requirements.

For the best of our knowledge, Global Q-Opt a prototype, for Openstack Swift, developed by Maria Couceiro, Matti Hiltunen, Paolo Romano, and Luís Rodrigues is the latest work related to adaptive quorum systems. It guarantees strong consistency and try to achieve higher throughput by adjusting the quorum configurations dynamically. Moreover, it uses Machine Learning (ML) techniques in predicting the correct quorum configuration. Hence, it collects statistics such as system wide read/write accesses, average read/write durations, successful replied read/write count etc from the proxy nodes of the cluster and pass them to ML module. Then the predicted quorum configuration is applied to all the proxy nodes using a non-blocking quorum reconfiguration algorithm.

Our work Q-Opt is an evolution of Global Q-Opt which is targeted to work with complex workload situations and still adapts to get higher throughput considering the popularity of the objects in the cloud storage.

III. Q-Opt

Q-Opt is designed to work with Software Defined Storage (SDS) where the external interface is represented by a set of proxy agents, and its data is distributed over a set of storage nodes. We denote the set of proxies by $\Pi = \{p_1, \ldots, p_P\}$, and the set of storage nodes by $\Sigma = \{s_1, \ldots, s_S\}$. In fact proxies and storage nodes are logical process which may be in practice mapped to a set of physical nodes using different strategies (e.g., proxies and storage nodes may reside in two disjoint sets of nodes, or each storage node may host a proxy). We assume that nodes may crash according to the fail-stop (non-byzantine) model. Furthermore, we assume that the system is asynchronous, but augmented with unreliable failure detectors which encapsulate all the synchrony assumptions of the system. Communication channels are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. We also assume that communication channels are FIFO ordered, that is if a process p sends messages $m1$ and $m2$ to a process q then q cannot receive $m2$ before $m1$.

As illustrated in Figure 1, Q-Opt is composed of three main components: the Autonomic Manager, the Reconfiguration Manager, and the Oracle.

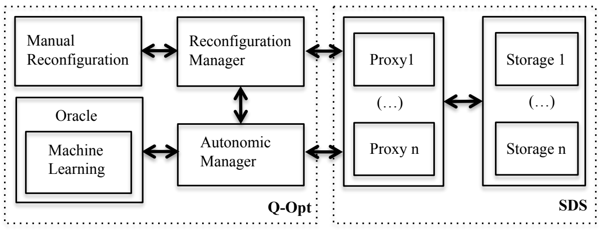

Figure 1. Architectural overview.

Autonomic Manager is responsible for orchestrating the self-tuning process of the system. Hence, it relies on an *Oracle* that encapsulates a black-box machine-learning based predictor that is responsible for determining the best quorum, given the workload of the monitored SDS application. *Reconfiguration Manager* is in charge of coordinating a non-blocking reconfiguration protocol that aims at altering the read/write quorum size used by the proxies for given data items.

A. Quorum Optimization

The Autonomic Manager is responsible for determining when to trigger a quorum reconfiguration. The pseudo code executed at the Autonomic Manager side is depicted in Algorithm 1. It executes the following tasks:

It first starts a new round by broadcasting the new round identifier, r, to all the proxies (line 5). Each proxy p_i then replies with (line 7):

```
1  int r=0; // Round identifier
2  // Fine-grain round-based optimization.
3  do
4      r=r+1;
5      broadcast [NewRound, r] to $\Pi$;
6      $\forall p_i \in \Pi$ :
7          wait received [RoundStats, r, topK$_i^r$, statsTopK$_i^{r-1}$,
           statsTail$_i^{r-1}$, th$_i^{r-1}$] from $p_i \vee$ suspect($p_i$);
8      statsTopK$^{r-1}$=merge(statsTopK$_1^{r-1}$, .., statsTopK$_P^{r-1}$);
9      topK$^r$=merge(topK$_1^r$,. . .,topK$_P^r$);
10     send [NewStats, r, statsTopK$^{r-1}$] to Oracle;
11     wait received [NewQuorums, r, quorumsTopK$^{r-1}$] from
       Oracle;
12     send [FineRec, r, $\langle$topK$^{r-1}$, quorumsTopK$^{r-1}\rangle$] to RM;
13     wait received [AckRec, r] from RM;
14     broadcast [NewTopK, r, topK$^r$] to $\Pi$;
15     th$^{r-1}$=aggregateThroughput(th$_1^{r-1}$,. . ., th$_P^{r-1}$) ;
16     $\Delta_{th}(\gamma)$ = throughput increase over last $\gamma$ rounds.;
17 while $\Delta_{th}(\gamma) \geq \theta$
18 // Tail optimization.
19 statsTail$^{r-1}$=merge(statsTail$_1^{r-1}$, ..,statsTail$_P^{r-1}$);
20 send [TailStats, statsTail$^{r-1}$] to Oracle;
21 wait received [TailQuorum, quorumTail$^{r-1}$] from Oracle;
22 send [CoarseRec, quorumTail$^{r-1}$] to RM;
23 wait received [AckRec, r] from RM;
```

Algorithm 1: Autonomic Manager pseudo-code.

- $topK_i^r$: A set of new "hotspots" objects that, according to the proxy's local accesses, should be optimized in the next round to obtain larger benefits. In order to be able to identify the "hotspots" on each proxy with low overhead, Q-Opt adopts a state of the art stream analysis algorithm [11] that permits to track the top-k most frequent items of a stream in an approximate, but very efficient manner.
- $statsTopK_i^{r-1}$: The ratio of write accesses and the size for each of the objects resulting in the top-k analysis of the previous round.
- $statsTailK_i^{r-1}$: Aggregate workload characteristics for the objects whose quorum size has not been individually optimized, i.e., the objects in the tail of the access distribution.
- th_i: The throughput achieved by the proxy during the last round.

Once the information sent by the proxies is gathered and merged (line 8 and 9), the merged statistics of previous round top-k is fed as *input features* to the Oracle (line 10), that outputs a *prediction* (line 11) of the right quorum to use *for each* object in the top-k set. In the current prototype, the Oracle only outputs the size W of the write quorum and the size R of the read quorum is derived automatically based on the system's replication degree, i.e., $R = N - W + 1$. If the output of the Oracle is different from the current quorum system for that object, a reconfiguration is triggered. In this case, the Autonomic Manager interacts with the Reconfiguration Manager (lines 12 and 13), which is in charge of orchestrating the coordination among proxy and storage nodes and adapt the current quorum configuration for the top-k objects identified in the previous round. Otherwise, if the current configuration is still valid, no reconfiguration is triggered. As a final step of a fine-grain optimization round, the Autonomic Manager broadcast the current top-k set to the proxies. Thus, each proxy can start monitoring the objects that belong to the current top-k set in the next round.

At the end of each round, the Autonomic Manager, based on the average throughput improvements achieved during the last γ rounds, decides whether to keep optimizing individual objects in a fine-grain manner or to stop. When the gains obtained with the fine-grain optimization of individual "hotspot" objects becomes negligible (i.e., lower than a tunable threshold θ), a final optimization step to tune the quorum configurations used to access the the remaining objects, i.e., the objects that fall in the tail of the access distribution. These objects are treated as bulk (lines 19 - 22): the same read/write quorum is assigned to all the objects in the tail of the access distribution based on its aggregate workload characterization.

B. Reconfiguration Manager

The Reconfiguration Manager (subsequently denoted RM) executes the required coordination among proxy and server nodes in order to allow them to alter the sizes of read and write quorums without endangering neither, consistency, nor availability during reconfigurations. This coordination enforced by Q-Opt is designed to preserve the following property, which is at the basis of all quorum systems that provide strong consistency:

Dynamic Quorum Consistency. *The quorum used by a read operation intersects with the write quorum of any concurrent write operation, and, if no concurrent write operation exists, with the quorum used by the last completed write operation.*

where two operations o_1, o_2 are concurrent if at the time in which a proxy starts processing o_2, the processing of o_1 by a (possibly different) proxy has not been finalized yet (or vice-versa). The availability of a SDS system, on the other hand, is preserved by ensuring that read/write operations can be executed in a non-blocking fashion during the reconfiguration phase, even despite the crash of (a subset of) proxy and storage nodes.

C. Algorithm overview

There are three different type of components involved in the execution of the reconfiguration algorithm: the storage nodes, the proxy nodes, and the RM. The purpose of the reconfiguration algorithm is to change the quorum configuration, i.e., the size of the read write quorums, used by the proxy servers.

The algorithm is coordinated by the RM. When the RM runs the reconfiguration algorithm we say that the RM *installs* a new quorum system. We denote the quorum system being used when the reconfiguration is started as the *old quorum* and the quorum system that is installed when the reconfiguration is concluded the *new quorum*. Old and new write and read quorums are denoted, respectively, as *oldW*, *oldR*, *newW*, and *newR*. Each quorum is associated with an *epoch number*, a sequential serial number that is incremented by the RM when some proxy is suspected to have crashed during a reconfiguration. We also assume that storage nodes maintain a variable, called *currentEpoch*, which stores the epoch number of the last quorum that has been installed by the RM. As it will be explained below, during the execution of the reconfiguration algorithm, proxy nodes use a special *transition quorum*, that is sized to guarantee intersection with both the old and new quorums.

We assume a fail stop-model (no recovery) for proxies and storage nodes, and that at least one proxy is correct. As for the storage nodes, in order to ensure the termination of read and write operations in the new and old quorum configuration, it is necessary to assume that the number of correct replicas is at least *max(oldR, oldW, newR, newW)*. For ease of presentation, we assume that the sets Σ and Π are static, i.e. nodes are not added to these sets, nor are they removed even after a crash. In order to cope with dynamic groups, one may use group membership techniques, e.g. [12], which are orthogonal to this work.

RM is equipped with an eventually perfect failure detection service [13] that provides, possibly erroneous, indications on whether any of the proxy nodes has crashed. An eventually perfect failure detector ensures *strong completeness*, i.e., all faulty proxy processes are eventually suspected, and *eventual strong accuracy*, i.e., there is a time after which no correct proxy process is ever suspected by the RM. The reconfiguration algorithm is indulgent [14], in the sense that in presence of false failure suspicions only the liveness of read/write operations can be endangered (as we will see they may be forced to re-execute), but neither the safety of the quorum system (i.e., the Dynamic Quorum Consistency property), nor the termination of the reconfiguration phase can be compromised by the occurrence of false failure suspicions. The failure detection service is encapsulated in the *suspect* primitive, which takes as input a process identifier $p_i \in \Pi$ and returns true or false depending on whether p_i is suspected to have crashed or not. Note that proxy servers are not required to detect the failure of storage servers nor vice-versa.

In absence of faults, the RM executes a two-phase reconfiguration protocol with the proxy servers, which can be roughly summarized as follows. In the first phase, the RM informs all proxies that a reconfiguration must be executed and instructs them to i) start using the transition quorum instead of the old quorum, and ii) wait till all the pending operations issued using the old quorum have completed. When all proxies reply, the RM starts the second phase, in which it informs all proxies that it is safe to start using the new quorum configuration.

This mechanism guarantees that the quorums used by read/write operations issued concurrently to the quorum reconfiguration intersect. However, one needs to address also the scenario in which a read operation is issued on an object that was last written in one of the previous quorum configurations, i.e., before the installation of the current quorum. In fact, if an object were to be last written using a write quorum, say *oldW*, smaller than the one used in the current configuration, then the current read quorum may not intersect with *oldW*. Hence, an obsolete version may be returned, violating safety. We detect this scenario by storing along with the object's metadata also a logical timestamp, $cfNo$, that identifies the quorum configuration used when the object was last written. If the version returned using the current read quorum was created in a previous quorum configuration having identifier $cfNo$, the proxy repeats the read using the largest read quorum used in any configuration installed since $cfNo$ (in case such read quorum is larger than the current one).

Since failure detection is not perfect, in order to ensure liveness the two-phase quorum reconfiguration protocol has to advance even if it cannot be guaranteed that all proxies have updated their quorum configuration. To this end, the RM triggers an epoch change on the back-end storage nodes, in order to guarantee that the operations issued by any unresponsive proxy (which may be using an outdated quorum configuration) are preventively discarded to preserve safety.

```
int epNo=0; // Epoch identifier
int cfNo=0; // Configuration round identifier
int curR=1, curW=N; // Sizes of the read and write quorums
// Any initialization value s.t. curR+curW>N is acceptable.
changeConfiguration(int newR, int newW)
    wait canReconfig;
    canReconfig = FALSE;
    cfNo++;
    broadcast [NEWQ, epNo, cfNo, newR, newW ] to $\Pi$;
    $\forall p_i \in \Pi$ :
      wait received [ACKNEWQ, epNo] from $p_i \vee suspect(p_i)$;
    if $\exists p_i : suspect(p_i)$ then
        tranR=max(curR,newR); tranW=max(curW,newW);
        epochChange(max(curR,curW),tranR,tranW);
    broadcast [CONFIRM, epNo, newR, newW ] to $\Pi$;
    $\forall p_i \in \Pi$ :
      wait received [ACKCONFIRM, epNo] from $p_i \vee suspect(p_i)$;
    if $\exists p_i : suspect(p_i)$ then
        epochChange(max(newR,newW),newR,newW);
    curR=newR; curW=newW;
    canReconfig = TRUE;
epochChange(int epochQ, int newR, int newW)
    epNo++;
    broadcast [NEWEP,epNo,cfNo,newR, newW ] to $\Sigma$;
    wait received [ACKNEWEP, epNo] from epochQ $s_i \in \Sigma$;
```
Algorithm 2: Reconfiguration Manager pseudo-code.

D. Quorum Reconfiguration Algorithm

The pseudo code for the reconfiguration algorithm executed at the Replication Manager side is depicted in Algorithm 2. The reconfiguration can be triggered by either the Autonomic Manager, or by a human system administrator, by invoking the *changeConfiguration* method and passing as arguments the new sizes for the read and write quorums, *newQ* and *WriteQ*. Multiple reconfigurations are executed in sequence: a new reconfiguration is only started by the RM after the previous reconfiguration concludes.

```
int lEpNo=0; // Epoch identifier
int lCfNo=0; // Configuration round identifier
set Q={}; // list of cfNo along with respective read/write quorum sizes
int curR=1, curW=N; // Sizes of the read and write quorums
// Any initialization value s.t. curR+curW>N is acceptable.
upon received [NewQ, epNo, cfNo, newR, newW ] from RM
    if lEpNo≤epNo then
        lEpNo=epNo;
        lCfNo=cfNo;
        Q=Q ∪ < cfNo, newR, newW > ;
        int oldR=curR; int oldW=curW;
        // new read/writes processed using transition quorum
        tranR=max(oldR,newR); tranW=max(oldW,newW);
        wait until all pending reads/writes issued using the old quorum complete;
        send [AckNewQ, epNo ] to RM;
upon received [Confirm, epNo, newR, newW ] from RM
    if lEpNo≤epNo then
        lEpNo=epNo;
        curR=newR; curW=newW;
        send [AckConfirm, epNo ] to RM;
```

Algorithm 3: Proxy pseudo-code (quorum reconfiguration).

Failure-free scenario. To start a reconfiguration, the RM broadcasts a **NewQ** message to all proxy nodes. Next, the RM waits till it has received an **AckNewQ** message from every proxy that is not suspected to have crashed.

Upon receipt of a **NewQ** message, see Algorithm 3, a proxy changes the quorum configuration used for its future read/write operations by using a *transition quorum*, whose read, respectively write, quorum size is equal to the maximum of the read, respectively write, quorum size in the old and new configurations. This ensures that the transition read (*tranR*), resp. write (*tranW*), quorum intersects with the write, resp. read, quorums of both the old and new configurations. Before replying to the RM with an **AckNewQ** message, the proxy waits until any "pending" operations it had issued using the old quorum completed.

If no proxy is suspected to have crashed, a **Confirm** message is broadcast to the proxy processes, in order to instruct them to switch to the new quorum configuration. Next, the RM waits for a **AckConfirm** reply from all the non-suspected proxy nodes. Finally, it flags that the reconfiguration has finished, which allow for accepting new reconfiguration requests.

The pseudo-code for the management of read and write operations at the proxy nodes is shown in Alg. 4 and Alg. 5. As already mentioned, in case a read operation is issued, the proxies need to check whether the version returned using the current read quorum was created by a write that used a write quorum smaller than the one currently in use. To this end, proxies maintain a set Q containing all quorum configurations installed so far[1] by the Autonomic Manager. If the version returned using the current read quorum was created in configuration $cfNo$, the proxy uses set Q to determine the value of the largest read quorum used in any

[1]In practice, the set Q can be immediately pruned whenever the maximum read quorum is installed.

```
1  upon received [Read, oId] from client $c$
2      while true do
3          broadcast [Read, oId, curEpNo] to $\Sigma$;
4          wait received [ReadReply, oId, val, ts, W] from $\Sigma' \subseteq \Sigma$
           s.t. $|\Sigma'|$=curR ∨ ([Nack, epNo,newR, newW ]
           ∧epNo > $lEpNo$);
5          if received [Nack, epNo, cfNo, newR, newW ] then
6              lEpNo=epNo; lCfNo=cfNo;
               curR=newR; curW=newW;
7              Q=Q ∪ < cfNo, newR, newW > ;
8              continue; // re-transmit in the new epoch
9          v= select the value with the freshest timestamp;
10         // Set of read quorums since v.cfNo till lCfNo;
11         $S = \{R_i :< q_i, R_i, \cdot > \in Q \wedge v.cfNo \leq q_i \leq lCfNo\}$;
12         if max(S)≤curR then
13             // safe to use cur. read quorum
14             send [ReadReply, oId, v] to client $c$;
15         else
16             // compute read quorum when v was created.
17             int oldR=max(S);
18             // obtain a total of oldR replies.
19             wait received [ReadReply, oId, val, ts] from $\Sigma' \subseteq \Sigma$
               s.t. $|\Sigma'|$=oldR ∨ ( [Nack, epNo,newR, newW ]
               ∧epNo > $lEpNo$);
20             if received [Nack, epNo, cfNo, newR, newW ] then
21                 lEpNo=epNo; lCfNo=cfNo;
                   curR=newR; curW=newW;
22                 Q=Q ∪ < cfNo, newR, newW > ;
23                 continue; // re-transmit in the new epoch
24             v= select the value with the freshest timestamp;
25             send [ReadReply, oId, v] to client $c$;
26             // write v using the current quorum
27             write($v$,$oId$,$v.ts$);
28         break;
29     end
```

Algorithm 4: Proxy pseudo-code (read logic).

```
upon received [Write, oId, value] from client $c$
    write (val, oId, getTimestamp());
    send [WriteReply, oId] to client $c$;
write(value v, objId oid, timestamp ts)
    while true do
        broadcast [Write, oId, val, ts, curEpNo] to $\Sigma$;
        wait received [WriteReply, oId] from $\Sigma' \subseteq \Sigma$ s.t.
        $|\Sigma'|$=curW ∨ ([Nack, epNo,newR, newW ]
        ∧epNo > $lEpNo$);
        if received [Nack, epNo, cfNo, newR, newW ] then
            lEpNo=epNo; lCfNo=cfNo;
            curR=newR; curW=newW;
            Q=Q ∪ < cfNo, newR, newW > ;
            continue; // re-transmit in the new epoch
        break;
    end
```

Algorithm 5: Proxy pseudo-code (write logic).

configuration since $cfNo$ till the current one. If this read quorum, noted oldR (see line 17 of Alg. 4), is larger than the current one, the read is repeated using oldR. Further, the value is written back using the current (larger) write quorum. Note that re-writing the object is not necessary for correctness. This write can be performed asynchronously, after returning the result to the client, and is meant to spare the cost of using a larger read quorum when serving future reads for the same object.

Coping with failure suspicions. In case the RM suspects some proxy while waiting for an **AckNewQ** or an **AckConfirm** message, the RM ensures that any operation running with an obsolete configurations is prevented from completing. To this end, the RM relies on the notion of *epochs*. Epochs are uniquely identified and totally ordered using a scalar timestamp, which is incremented by the RM whenever it suspects the failure of a proxy at lines 11 and 16 of Alg. 2. In this case, after increasing the epoch number, the RM broadcasts the **NEWEP** message to the storage nodes. This message includes 1) the new epoch identifier, and 2) the configuration of the transition quorum or of the new quorum, depending on whether the epoch change was triggered at the end of the first or of the second phase.

Next, the RM waits for acknowledgements from an *epoch-change quorum*, whose size is determined in order to guarantee that it intersects with the read and write quorums of any of the configurations in which the proxies may be executing. Specifically, if the epoch change is triggered at the end of the first phase, the size of the epoch-change quorum is set equal to the maximum between the size of the read and write quorums in the old configuration. It is instead set equal to the maximum between the size of the read and write quorums of the new configuration, if the epoch change is triggered at the end of the second phase.

When a storage node (see Alg. 6) receives an **NEWEP** message tagged with an epoch identifier larger than its local epoch timestamp, it updates its local timestamp and *rejects* any future write/read operation tagged with a lower epoch timestamp. It then replies back to the RM with an **ACKNEWEP** message. Whenever an an operation issued by a proxy in an old epoch is rejected, the storage node does not process the operation and replies with a **NACK** message, in which it specifies the current epoch number and the quorum configuration of this epoch.

Upon receiving a **NACK** message (see Algs. 4 and 5), the proxy node is informed of the existence of a newer epoch, along with the associated quorum configuration. Hence, it accordingly updates its local knowledge (i.e., its epoch and read/write quorum sizes), and re-executes the operation using the new epoch number and the updated quorum configuration.

E. Per-object quorum reconfiguration

As discussed, the above presented protocol allows for altering the quorum size used by the entire data store. However, extending the above presented protocol to allow for tuning independently the quorum sizes used to access different objects in the data store is relatively straightforward.

In particular, during the initial, round based optimization phase described in Section III-A, the RM is provided with a set of object identifiers and with their corresponding new quorum configurations. The RM forwards this information to the proxies via the *NewQ* message. The proxy servers, in their turn, shall store the mapping between the specified object identifiers and the corresponding write quorums, and use this information whenever they are serving a read or a write operation. Note that, in our prototype, we store this mapping in main memory, as only a reduced set of

```
int lEpNo=0; // Epoch identifier
int lCfNo=0; // Configuration round identifier
int curR=1, curW=N; //Sizes of the read and write quorums
// Any initialization value s.t. curR+curW>N is acceptable.
upon received [NEWEP,epNo, cfNo, newR, newW ] from RM
    if epNo $\geq$lEpNo then
        lEpNo=epNo;
        lCfNo=cfNo;
        curR=newR; curW=newW;
        send [ACKNEWEP, epNo] to Reconfiguration Manager;
upon received [Read, epNo, ...] or [Write, epNo, ...] from $\pi_i \in \Pi$
    if epNo <lEpNo then
        send [NACK, epNo, cfNo, newR, newW ] to $p_i$;
    else
        process read/write operation normally;
        if operation is a write then
            store lCfNo in the version metadata cfNo;
        else
            piggyback cfNo to the ReadReply message;
        end
    end
```
Algorithm 6: Storage node pseudo-code.

"hotspots" is individually optimized. This simple mechanism allows the proxy servers to determine the new and old quorum sizes to use on a per object basis. Also, when a fine-grained quorum reconfiguration is requested that only affects a set of data items D where the reconfiguration should wait only, in the first phase of the reconfiguration algorithm, for the completion of any pending operation (using the old quorum) targeting some of the items in D.

Any (read/write) access to objects whose quorum size has not been individually optimized can use a common, global quorum configuration and be treated exactly as shown in Section III-D.

IV. EVALUATION

In this section we present the results of an experimental study aimed at quantifying the impact on performance of using different read and write quorum sizes in OpenStack Swift. Then we assess three main aspects: the accuracy of the ML-based Oracle; the effectiveness of Q-Opt in automatically tuning the quorum configuration in presence of complex workloads; and the efficiency of the quorum reconfiguration algorithm.

A. Test-Bed

The experimental test-bed used to gather the experimental results presented in this document is a private cloud comprising a set of virtual machines (VMs) deployed over a cluster of 20 physical machines. The physical machines are connected via a Gigabit switch and each is equipped with 8 cores, 40 GB of RAM and 2x SATA 15K RPM hard drives (HDs). We allocate 10 VMs for the storage nodes, 5 VMs to serve as proxy nodes, and 5 VMs to emulate clients, i.e., to inject workload. Each client VM is statically associated with a different proxy node and runs 10 threads that generate a closed workload (i.e., a thread injects a new operation only after having received a reply for the previously submitted operation) with zero think time. Each proxy and client VM

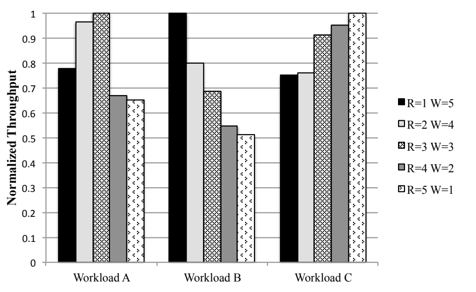

Figure 2. Normalized throughput of the studied workloads.

runs Ubuntu 12.04, and is equipped with 8 (virtual) cores, 10GB disk and 16GB of RAM memory. On the other hand, each storage node VM is equipped with 2 (virtual) cores, 100GB disk and 9GB of RAM memory.

Furthermore, the top-k module which would filter out the popular objects based on the access distribution is collocated in every proxy node, hence each proxy can individually query the top-k module with statistics of object accesses via that proxy. One of the proxy nodes would work as the master so that the ML module is collocated there. Hence in this set up proxy nodes independently query the top-k module to find the popular objects and then will send those objects' statistics to the master proxy where the ML module is queried (only by the master) in order to find the corresponding quorum configuration.

When we initiate the storage system we set the replication degree to 5 and use the default distribution policy that scatters object replicas randomly across the storage nodes (while enforcing that replicas of the same object are placed on different nodes)

In the first motivating experiment explained in section IV-B we have used a single tenant and a workload where all objects are accessed with the same profile. In the evaluation of the full system we consider more complex scenarios with skewed non-uniform workloads.

B. Impact of the read/write quorum sizes

We start by considering 3 different workloads that are representative of different application scenarios. Specifically, we consider two of the workloads specified by the well known YCSB [15] benchmark (noted Workload A and B), which are representative of scenarios in which the SDS is used, respectively, to store the state of users' sessions in a web application, and to add/retrieve tags to a collection of photos. The former has a balanced ratio between read and write operations, the latter has a read-dominated workload in which 95% of the generated operations are read accesses. We also consider a third workload, which is representative of scenario in which the SDS is used as a backup service (noted Workload C). In this case, 99% of the accesses are write operations. Note that such write-intensive workloads are frequent in the context of personal file storage applications, as in these systems a significant fraction of users exhibits an upload-only access pattern [16].

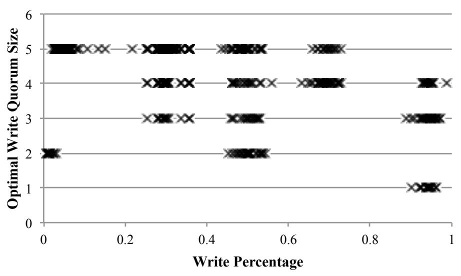

Figure 3. Optimal write quorum configuration vs write percentage.

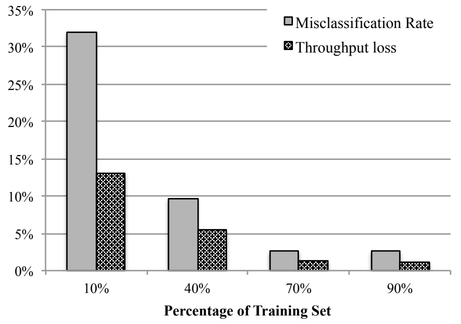

Figure 4. Oracle's misclassification rate and % throughput loss.

Figure 2 shows the throughput of the system (successful operations per second) normalized with respect to the best read/write quorum configuration for each workload. These results were obtained using one proxy node and 10 clients. The results clearly show that when increasing the size of the predominant operation quorum, the number of served operations decreases: configurations favouring smaller read quorums will achieve a higher throughput in read-dominated workloads, such as Workload B, and vice-versa, configurations favouring smaller write quorums achieve a higher throughput in write-dominated workloads, such as Workload C. Mixed workloads such as Workload A, with 50% read and 50% write operations, perform better with more balanced quorums, favouring slightly reading from less replicas because read operations are faster than write operations (as these need to write to disk).

In order to assess to what extent the relation between the percentage of writes in the workload and the optimal write quorum size may be captured by some linear dependency, we tested approx. 170 workloads, obtained by varying the percentage of read/write operations, the average object size, and the number of clients connected to the proxy in the domain. In Figure 3 we show a scatter plot contrasting, for each tested workload, the optimal write quorum size and the corresponding write percentage. The experimental data clearly highlights the lack of a clear linear correlation between these two variables, and has motivated our choice of employing black-box modelling techniques (i.e., decision trees) capable of inferring more complex, non-linear dependencies between the characteristics of a workload and its optimal quorum configuration.

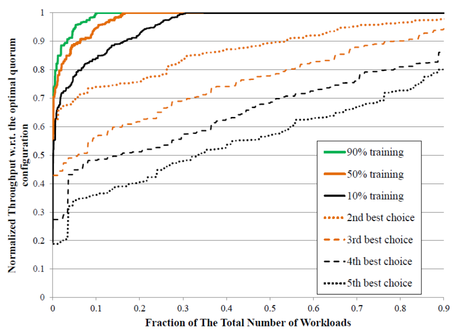

Figure 5. Cumulative distribution of the Oracle's accuracy.

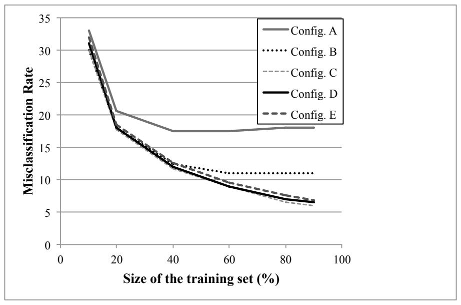

Figure 6. Oracle's misclassification rate when varying the set of features used.

V. Accuracy of the Oracle.

In order to assess the accuracy of the Oracle, we consider the same set of 170 workloads used in Figure 3. Figure 4 reports the misclassification rate and throughput loss (w.r.t. the optimal solution) when we vary the size of the training set, using the rest of available data as test set. The results are obtained as the average of 200 runs, in which we fed the C5.0 which builds a decision-tree classification model in an initial, off-line training phase, with different (and disjoint) randomly selected test and training sets. The results show that the ML-based oracle achieves very high accuracy, i.e. misclassification rate is lower than 10% and throughput loss is about 5%, if we use as little as the 40% of the collected data set as training set. Interestingly, we observe that the throughput loss is normally less than half of the misclassification rate: this depends on the fact that, in most of the misclassified workloads, the quorum configuration selected by the oracle yields performance levels that are quite close to the optimal ones.

Figure 5 provides an alternative perspective on our data set. It reports the cumulative distribution functions of the accuracy achieved by our predictor with different training set sizes, and contrasting them with the normalized performances using all possible configurations for each considered workload. The plot highlights that the choice of the quorum configuration has a striking effect on Q-Opt's performance: for instance, in about 20% of the workloads, the selection of the third best quorum configuration is 40% slower than the optimal one; in the worst case, the plot also shows that the worst (i.e., the 5-th best) choice of the quorum configuration can be even more that 5x slower than the optimal for some workloads.

Figure 6 allows us to asses to what extent the selection of the features used to generate the ML-based models impacts the model's accuracy. We used five different configurations for this experiment, from A to E. Each configuration progressively adds extra features, including previous configuration features. As the plot shows, the configuration A, which only includes the percentage of write transactions as a feature, achieves poor accuracy, generating a misclassification rate of around 20%. This result confirms the relevance of using a multi-variate model, capable of keeping into account additional factors besides the write percentage (as suggested also by the plot in Figure 3). Indeed, the plot clearly shows that, as we include among the provided features also the object size (Conf. B) and throughput (Conf. C) the misclassification rate gets considerably reduced. Our experimental data show also that adding additional features (e.g., conf. D and E, which include statistics on the latencies perceived by get and put operations) does not benefit accuracy, but, on the contrary, can lead to overfitting [17] phenomena that can ultimately have a detrimental effect on the learner's accuracy.

Finally, in Table I, we report the accuracy achieved by the considered learner when using the, so called, *boosting* technique. The boosting approach consists in training a chain of N learners, where the learner in position i is trained to learn how to correct the errors produced by the chain of learners in position 1,2,. . .,i. This technique has been frequently reported to yield significant accuracy improvements when used with weak learners. Our experiments do confirm the benefits of this technique, although the relative gains in accuracy are, at least for the considered data set, not so relevant to justify its additional computational overheads.

A. *Reconfiguration Overhead*

Since, Q-Opt uses a two-phase quorum reconfiguration protocol whose latency is affected by the number of pending operations each proxy has to finish before completing the first phase of the protocol. Thus, we expect to observe an increase in latency as the number of clients issuing concurrent operations increases. Figure 7 shows the quorum reconfiguration latency in absence of faults varying the number of clients from 15 to 150 (i.e., close to system's saturation that we estimate at around 165 clients). As expected, the results show a correlation between the latency of the reconfiguration and the request's arrival rate; however, the reconfiguration latency remains, in all the tested scenarios, lower than 15 milliseconds, which confirms the efficiency of the proposed reconfiguration strategy.

Figure 8 focuses on evaluating the performance of the lazy write-back procedure encompassed by the quorum reconfiguration protocol. Recall that this happens when a read operation gathers a quorum of replies which reveals that the last write applied to that object has been performed using a smaller write quorum than the one currently used. In

	10% Training		50% Training		90% Training	
	unboosted	boosted	unboosted	boosted	unboosted	boosted
Avg. Misclassification (%)	15	14	9	7	6	5
Avg. Distance from optimal (%)	4.6	4	2.2	1.6	1.6	0.9
Avg. Distance when misclassified (%)	14.1	12.9	10.9	9.5	9.5	7.6

Table I
IMPACT OF BOOSTING IN THE ORACLE'S PERFORMANCE WHEN VARYING THE TRAINING SET SIZE.

this case, the read operation is forced to wait for additional replies before returning to the client, and it has to write back the data item using the new (larger) write quorum. The experiment whose results are shown in Figure 8 is focused precisely on evaluating the overhead associated with these additional writes.

To this end we considered a worst-case scenario in which: i) the system is heavily loaded, ii) data items are accessed with a uniform distribution, and iii) the write quorum increases from 1 to 5 while the application is running a read-dominated (95%) workload. At the beginning of the experiment, we allow the system to run for 3 minutes until it stabilizes and then we trigger the bulk reconfiguration (i.e., for the entire set of 20K data items in the SDS) of the quorum and report throughput over time. Moreover, we implemented batching and asynchronous updates to the rewrite operations so that the system would not experience a significant performance degradation due to the bulk reconfiguration. Instead, the plot shows that, even in such a worst case scenario, there is a significant throughput gain and it increases further to align with the baseline throughput once the rewrite operations are completed for the bulk reconfiguration.

B. System Performance

Finally, Figure 9 evaluates the effectiveness of Q-OPT when faced with time in the presence of complex workloads. We compare few configuration against Q-OPT. *R1W5*, *R3W3* and *R1W5* are static configurations that force the system to use the same quorum for all the objects. *AllBest* uses the optimal quorum for each of the objects. Finally, *Top10%* uses the optimal quorum for each of the 10% most accessed objects. Notice that *AllBest* and *Top10%* are unachievable configurations in practice since it would require precise pre-knowledge about the workloads of each object. Those configurations will serve us as baselines.

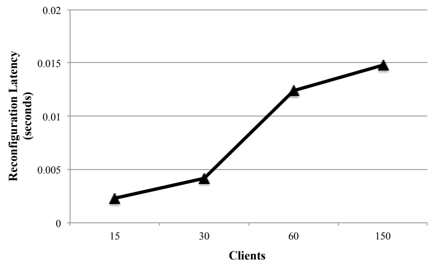

Figure 7. Reconfiguration latency while varying the number of clients.

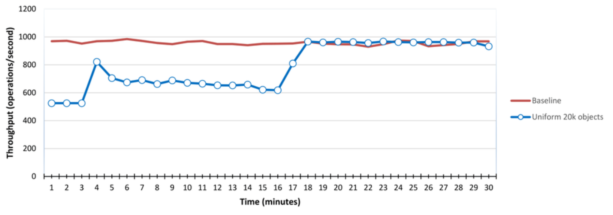

Figure 8. Evaluating the overheads associated with the bulk reconfiguration with batching and asynchronous updates. Write quorum is increased from 1 to 5 in presence of a read-dominated workload.

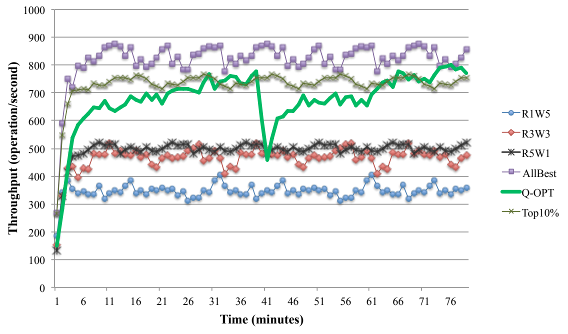

Figure 9. Q-OPT performance in comparison to other configurations.

In the experiment we combine two workloads, one read intensive and one write intensive, each of them representing a different tenant. This means that each workload accesses a non-overlapping set of objects. After 40 minutes, we swap the type of workload. Therefore, the read intensive workload becomes write intensive and viceversa. The idea is to observe how Q-OPT reacts to changes in the workloads. For this experiment, Q-OPT runs a fine-grain optimization round every minute. After 20 minutes, the fine-grain optimization phase ends and Q-OPT optimizes the tail. After the swapping (minute 40), Q-OPT starts again the fine-grain optimization phase and continues behaving as described for the first 40 minutes.

Q-OPT behaves as expected. During the first 20 minutes the throughput grows as the fine-grain optimization rounds advance, getting close to the *Top10%* baseline right before optimizing the tail. Once the tail is optimized (after minute 20), the throughput keeps growing even beyond the *Top10%* line and getting closer to the *AllBest* configuration. The *Top10%* configuration does not optimize the tail; therefore, it is expected that Q-Opt outperforms it, at least slightly. As expected, the plot shows that the performance of Q-Opt matches closely that of the optimal configurations. These results confirm the accuracy of Q-Opt's Oracle and highlight that the overheads introduced by the supports for adaptivity are very reduced. After swapping the workloads, Q-Opt experiences a noticeable decrement in the performance since it has to start the optimization phase from zero.

Furthermore, the figure shows that none of the static configurations is capable of achieving high throughput in comparison to the baselines and Q-Opt. In the worst case, the *R1W5* configuration is more than 2x slower than Q-Opt during stable periods. Even for the best static configuration (*R5W1*), Q-Opt still achieves around 45% higher throughput.

VI. Conclusions

Our work tackled the problem of automating the tuning of read/ write quorum configuration in distributed storage systems, a problem that is particularly relevant given the emergence of the software defined storage paradigm. The proposed solution, which we called Q-Opt, leverages on ML techniques to automate the identification of the optimal quorum configurations given the current application's workload, and on a reconfiguration mechanism that allows non-blocking processing of requests even during quorum reconfigurations. Q-Opt's optimization phase first focuses on optimizing the most accessed objects in a fine-grain manner. Then, it ends by assigning the same quorum for all the objects in the tail of the access distribution based on their aggregated profile. We integrated Q-Opt in a popular, open-source software defined storage and conducted an extensive experimental evaluation, which highlighted both the accuracy of its ML-based predictive model and the efficiency of its quorum reconfiguration algorithm.

Apart from the improvements done so we would like enhance this system to optimize based on tenants usage. For instance, selecting hot-objects system wide would be not fair for all the tenants using the system because all hot objects may belong to a certain tenant or few of the tenants. This will make poor user experience for the rest of the users and because a cloud storage is shared by vast amount of tenants finding out a proper way to improve fairness in usage would be worthy to research.

Acknowledgments

This work is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politècnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) funded by the European Union via the Erasmus Mundus program. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Prof. Paolo Romano, Manuel Bravo and Maria Couceiro.

References

[1] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[2] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. SOSP 2007*. ACM, 2007, pp. 205–220.

[3] Openstack-Swift, "Highly available, distributed, eventually consistent object/blob store. Accessed on 2015-01-03," http://docs.openstack.org/developer/swift, 2009.

[4] O. Wolfson, S. Jajodia, and Y. Huang, "An adaptive data replication algorithm," *ACM Trans. Database Syst.*, vol. 22, no. 2, pp. 255–314, Jun. 1997. [Online]. Available: http://doi.acm.org/10.1145/249978.249982

[5] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[6] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "Autoplacer: Scalable self-tuning data placement in distributed key-value stores," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 119–131. [Online]. Available: https://www.usenix.org/conference/icac13/technical-sessions/presentation/paiva

[7] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme, "Are quorums an alternative for data replication?" *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 257–294, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/937598.937601

[8] IMEX-Research, "The promise and challenges of cloud storage. Accessed on 2015-01-02," http://tiny.cc/sdpftx, 2015.

[9] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu, "An application-based adaptive replica consistency for cloud storage," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, Nov 2010, pp. 13–17.

[10] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. Perez, "Harmony: Towards automated self-adaptive consistency in cloud storage," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012, pp. 293–301.

[11] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. of the 10th ICDT*, Edinburgh,Scotland, 2005.

[12] K. Birman and R. V. Renesse, *Reliable Distributed Computing with the ISIS Toolkit*, R. V. Renesse, Ed., 1994.

[13] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[14] R. Guerraoui, "Indulgent algorithms (preliminary version)," in *Proc. PODC 2000*. ACM, 2000, pp. 289–297.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. SOCC 2010*. ACM, 2010, pp. 143–154.

[16] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: Understanding personal cloud storage services," in *Proc. IMC 2012*. ACM, 2012, pp. 481–494.

[17] T. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.