Dealing Proactively with Data Corruption: Challenges and Opportunities

Nedyalko Borisov[†], Shivnath Babu[†], Nagapramod Mandagere[‡], Sandeep Uttamchandani[‡]

[†]*Computer Science Department, Duke University*
{nedyalko,shivnath}@cs.duke.edu

[‡]*IBM Almaden Research Center*
{nmandage,sandeepu}@us.ibm.com

***Abstract*—The danger of production or backup data becoming corrupted is a problem that database administrators dread. This position paper aims to bring this problem to the attention of the database research community, which, surprisingly, has by and large overlooked this problem. We begin by pointing out the causes and consequences of data corruption. We then describe the Proactive Checking Framework (PCF), a new framework that enables a database system to deal with data corruption automatically and proactively. We use a prototype implementation of PCF to give deeper insights into the overall problem and to outline a challenging research agenda to address it.**

I. The Dangers of Data Corruption

The "C" in ACID denotes the consistency of data managed by a Database Management System (DBMS). Consistency of data is defined in terms of properties that the data should satisfy [1]. The data can be seen as consisting of *data elements*, where an element can be any unit or container of data such as a record, a disk block, a table, a pair of records, or a group of tables. A state of the data has values for each of its elements. Each element e has certain properties that e's value should satisfy. A state is labeled *consistent* if it satisfies all properties for all elements; otherwise, the state is *corrupted*.

Some of the properties of an element e can be specified explicitly as constraints in the database schema. Examples include key constraints, referential-integrity constraints, check constraints, assertions, and triggers [1]. The DBMS will enforce these constraints on writes (insertions, deletions, and updates) to e. Other properties may have to be enforced implicitly, e.g., through usage policies such as giving write access to e to privileged users only.

To safeguard the consistency of data, DBMSs today depend on *correctness assumptions* like: any transaction run in the system in isolation and in the absence of system failures will take the data from one consistent state to another [1]. Justifications for assumptions like these include further assumptions like: if "someone" is given authorization to modify the state of the data, then she will understand and enforce properties of all data elements affected by the modifications [1]. Such overarching assumptions have many problematic aspects.

Hardware does not guarantee bit-level consistency: Hardware can corrupt bits, so it is impractical to assume that the bits that were expected to be written to an element e by the DBMS will be the same as what the DBMS gets back on a read of e. Alpha particles and cosmic rays can cause bit flips in the RAM and CPU [2]. Erratic disk arms and dust cause corruption of data stored on disk. Hardware is more abused today than ever for performance boosts (over-clocking) and power savings (under-clocking); which increases the chances of errors. The increased use of commodity hardware—whose cheaper prices compared to enterprise-grade hardware are partly due to less reliable components—poses similar concerns. Recent studies have shown that commodity hardware such as nearline SATA disks and RAM without error correcting codes (ECC) have considerably higher chances of causing corruption than their enterprise counterparts [3], [4], [2].

Trusting other systems to enforce consistency of the DBMS's data is risky: A DBMS may store its data on a file-system or storage area network (SAN), use a load balancer to manage resources, and copy data among nodes using network cards and switches. Software bugs in all of these systems have caused data corruption in the past (e.g., [3], [5]). Bugs in file-system and storage-engine software (which can run into 100,000+ lines of code) have caused problems such as *partial writes*, *lost writes*, and *misdirected writes* [6]. A partial write is a nonatomic write to a data element. A write issued by the DBMS may be acknowledged as successful by the storage layer even though the write is never performed (lost). In a misdirected write, the DBMS gets an acknowledgment of successful completion, but the actual write happens to a different location than what the DBMS intended. Application and system crashes are also known to cause *wild* writes.

The DBMS can have bugs too: The last few years have seen the emergence of a number of new DBMSs such as NoSQL systems, large-scale parallel databases, and MapReduce frameworks. New DBMS developers are pushing the envelope on increasing performance and availability by employing techniques like eventual consistency instead of the more time-tested ACID techniques. CouchDB (a popular NoSQL system) recently had a well-publicized bug that caused a race condition due to an overeagerness to prioritize performance over consistency. This bug led to writes not being committed to disk, causing data loss [7]. Hadoop had a similar problem [8].

Mistakes or malicious intent from humans: Misconfiguration of database parameters or administrative tasks like backups by database administrators (DBAs) can cause data corruption. We have seen erratic behavior from enterprise DBMSs when

[†]Authors Borisov and Babu are supported by NSF award number 0917062

free storage space runs out. Viruses, worms, and SQL-injection attacks can also corrupt production data.

Corruption of metadata can cause corruption of data: Consistency of metadata such as indexes in DBMSs and *inodes* (indexes used to map file offsets to blocks) in file-systems is just as important as consistency of data. For example, a lost write that causes an index update to be omitted can lead to incorrect query results when the index is used in query processing. Similarly, inode corruption can cause blocks of a file to "disappear" or become inaccessible; while superblock corruption can do the same to entire file-systems.

Data corruption can lead to three undesirable outcomes: data loss, system unavailability, and incorrect results.

Data loss: The most catastrophic outcome of data corruption is irretrievable loss of data. The recent bug in CouchDB caused writes to not be made persistent to disk, creating the potential for data loss when dirty buffers were evicted or lost from memory. Guarantees from replication are reduced in the presence of corruption since multiple replicas of a data element could become corrupted over a period of time due to independent or correlated hardware, software, or human errors.

Corruption of backups (archives) is a surprisingly common cause of data loss that often catches DBAs by surprise (e.g., [9]). A backup can become corrupted on its own or due to the spreading of corruption from the production system. Backups are often done in an incremental fashion where full dumps of the data are archived occasionally while changes (diffs) with respect to the last backup are archived more frequently. Corruption of any backup in the chain can lead to data loss. Suppose a disaster were to strike the production DBMS. In this situation, if one or more backups are found to be corrupted, then it may be impossible to recreate the version of the data as it existed before the disaster.

A tragic story here involves the social-bookmarking site Ma.gnolia.com which was gaining in popularity before it went out of business [10]. Ma.gnolia.com experienced an instance of file-system corruption that spread silently to data backups. Only when disaster struck did the administrators realize that data stored by most users was lost irretrievably. Ma.gnolia.com went out of business shortly after the problem happened.

System unavailability: Corruption of critical metadata in OSs, DBMSs, and file-systems can cause the system to crash or to be otherwise unavailable [11], [12]. OSs may be unable to boot in the presence of metadata corruption. We know of an instance where an enterprise DBMS kept crashing when DBAs attempted to bring it online from backups that happened to be corrupted; and the system was unavailable for a week before the problem was fixed manually.

Incorrect results: A recent corruption-injection study shows how the MySQL DBMS can crash or return incorrect results (usually no results or incomplete results) under data and metadata corruption [11]. A similar study with the state-of-the-art ZFS file-system showed that memory-level corruption can cause ZFS to crash or to return incorrect results [12].

A. Checks

Systems have developed a suite of *checks* to prevent, detect, and repair corruption.

- *Write-time checks:* These checks seek to prevent corruption on explicit writes to data elements. Before a write happens to a data element e, a write-time check can be applied to guarantee that the new value will not corrupt e. Constraints such as key and referential integrity used in DBMSs are popular examples of write-time checks [1].
- *Read-time checks:* A read-time check can be applied to a data element e to guarantee that the read will return a consistent value of e. Read-time checks usually depend on some additional information that is stored when writes to e happen. Checksums are the most popular examples of read-time checks. For example, a checksum computed from the contents of a disk block can be stored along with the block or elsewhere. On a read of the block, a checksum computed from the current contents of the block can be compared with the stored checksum to detect corruption caused by partial or wild writes. The Oracle *Hardware Assisted Resilient Data (HARD)* initiative is a major effort to enable the DBMS to generate checksums that can be validated by the underlying storage system to detect corruption caused anywhere in the I/O path [13]. Other examples of read-time checks include (a) comparing multiple replicas of a data element, and (b) verifying that the log sequence number (LSN) stored on a block is in agreement with the log. The LSN denotes the last persistent update made to the block.
- *Access-separated checks:* Write-time and read-time checks are *access-blended checks* that are done inline with the regular system execution as it serves a workload. Most systems have a set of *access-separated checks*—packaged commonly in the form of test suites—that can be applied outside the regular execution path. For example, MySQL has the *myisamchk* suite which supports five invocation options: fast, check-only-changed, check (default), medium-check, and extended-check. When invoked with one of these options, myisamchk applies checks of increasing sophistication and thoroughness to verify the consistency of tables and indexes in a database. *fsck* is a similar umbrella test suite for file-systems. Table I-A gives an overview.

B. Our Contributions

Access-blended checks are necessary but not sufficient: We argue in this paper that access-blended checks are necessary but not sufficient to deal comprehensively with data corruption problems faced by DBMSs. (The Oracle HARD initiative focuses on access-blended checks.) We offer two justifications for this argument. The primary justification is that data corruption can occur outside the regular execution path of the DBMS. For example, a bit flip in RAM or disk can corrupt any data element e at any point; even if e has not been accessed for a long time in the past nor will it be accessed for a long time in the future. Corruption of stored backup data can also occur outside the execution path in the production system.

TABLE I
EXAMPLE SUITES OF ACCESS-SEPARATED CHECKS

myisamchk check suite for the MySQL DBMS	
-F (Fast)	Verifies that tables have been closed properly and that all table flags are correct.
-C (Check only changed)	Verifies tables marked as changed since the last check. Tables are marked as changed if records were modified or a crash occurred during table access or repair.
-c (Default check)	Verifies that (i) table metadata (e.g., size, flags) matches the physical data, (ii) all deleted records are disposed properly, and (iii) all database page checksums are correct.
-m (Medium)	Extends default check and verifies that each record checksum matches the record data.
-e (Extended)	Extends medium and verifies that each index entry corresponds to a valid record in the table, and vice versa.
-r (Recovery)	Reads a table and repairs any corruptions found; it also updates the indexes.
Check suites for the XFS file-system	
xfs_check	Verifies the superblock, free space, and inode maps. Uses internal log to verify all inodes and corresponding data blocks, and each inode's relation in the file hierarchy.
xfs_repair	Repairs the superblock, and then uses the internal log to correct inodes and to find duplicate or orphan data blocks. Repairs each inode's relation in file hierarchy if needed.

The second justification is that checks vary from simple checksum comparisons to highly sophisticated ones that (i) traverse large data structures like an index or inode, or (ii) perform cross-comparisons across multiple data structures such as a table and the log or one or more indexes. Sophisticated checks can be resource-intensive and may need exclusive access to entire data structures. Running such checks in an access-blended fashion can be prohibitive from a performance overhead perspective as well as undesirable from the perspective of maintaining predictability of workload performance.

A self-managing DBMS should schedule access-separated checks proactively: Given the above justifications, access-blended checks, despite their usefulness, do not obviate the need for access-separated checks. The dominant usage mode of access-separated checks today is to run them in a postmortem fashion to track down and repair corruption(s) after one or more of the undesirable outcomes of corruption have occurred.

We conducted an informal survey among IT administrators in enterprise clients of a large company. Some of them run access-separated checks on data backups (usually once per day on the daily backup). However, none of these administrators had a well-defined process in place to deal with data corruption if it were to be detected by the checks. The rest of the administrators do not run any access-separated checks proactively. If corruption were to happen in these cases, then tedious manual steps will have to be taken for detection and repair; and high system downtime and data loss can result.

These observations lead to our position in this paper that a self-managing DBMS should: (i) incorporate access-separated checks as a first-class citizen (along with access-blended checks like checksums); and (ii) be responsible for scheduling access-separated checks proactively to significantly lower the chances of corruption leading to the undesirable outcomes of data loss, system unavailability, and incorrect results. This position opens up a challenging research agenda comprising questions such as how, when, and where to run resource-intensive checks, and reasoning about how low-level checks

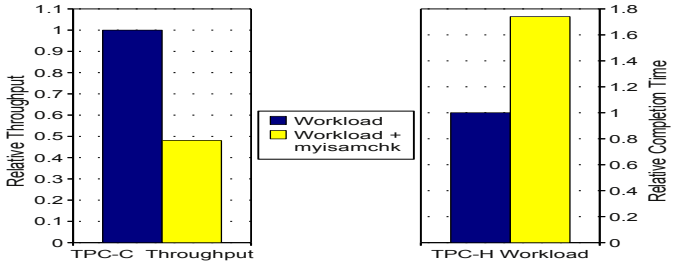

Fig. 1. Relative overhead of running MySQL's "myisamchk extended" access-separated check concurrently with two production workloads

can be scheduled in different systems like the DBMS, file-system, and storage system in order to meet higher-level consistency requirements at the application level. Section III proposes a solution framework—along with empirical results from an initial prototype—and Section IV uses this framework as a context to flesh out the research agenda.

II. EMPIRICAL EVALUATION OF CHECK OVERHEADS

We first consider the impact of running access-separated checks concurrently with the workload on the production DBMS. We consider an OLTP (read-write) workload and an OLAP (read-only) workload on a MySQL DBMS; the popular TPC-C (100 warehouses, 10 connections each) and TPC-H (scale factor of 20) benchmarks were used respectively. The production DBMS runs on the Amazon Cloud on an EC2 large instance provided by Amazon Web Services (AWS) [14]. EC2 large instances have 7.5GB of memory.

The Amazon Cloud provides three different types of storage: (i) instance storage, (ii) Simple Storage Service (S3), and (iii) Elastic Block Storage (EBS) volumes. Instance storage is the local storage on the EC2 instance. Instance storage does not persist beyond the lifespan of an EC2 instance, so the DBMS can use it to store temporary data only. EBS volumes are block-based storage units that persist independent of the lifespan of EC2 instances. The access latency and I/O throughput of EBS volumes are comparable to or better than that of instance storage; so we used a 50GB EBS volume as the persistent store of the production DBMS data. S3 storage offers a put and get interface, and is independent of EC2 instances. However, the data access rates from S3 are too slow for an enterprise DBMS. Since MySQL stores its data and indexes on a file-system, an XFS file-system is created on the EBS volume. Thus, the full production software stack consists of the OLTP/OLAP workload, MySQL DBMS, XFS file-system, and a 50GB EBS volume on a large EC2 instance.

Figure 1 shows the impact of running the access-separated checks concurrently with the production workload. The access-separated checks we consider are run using MySQL's *myisamchk extended* test (the "-e" option from Table I-A). The data element for this test is the set of all tables in the DBMS. As Figure 1 shows, the overhead introduced by the concurrent check execution is high: the throughput of the OLTP workload is reduced by around 50%, while the completion time of the OLAP workload is almost doubled.

III. A SOLUTION: PROACTIVE CHECKING FRAMEWORK

As discussed in Section I, a check operates on some data element e. The element should not be modified while the

check is running, otherwise the outcome of the check can be unpredictable. Thus, most checks need a read lock on the complete data element to prevent concurrent updates. If the check repairs any corruption detected, then the check would need a write lock on e. Recall that an element can be any unit or container of data such as a record, a disk block, a table, a pair of records, or a group of tables. The factors contributing to the high overhead in Figure 1 are the contention from locking and resource allocation due to the concurrent execution of access-separated checks. We propose a two-step solution to address this overhead:

- Take *snapshots* efficiently of the data element needed for an access-separated check, and run the check on each snapshot; thereby avoiding the blocking of updates to the same element by the production workload.
- Use planned resource allocation and isolation to run checks with low resource contention on the production workload.

We have implemented a *Proactive Checking Framework (PCF)* that automates these two steps, which we present next. While PCF has a fairly general design, our current prototype of PCF is based on mechanisms provided by the Amazon Cloud. Figure 2 shows the architecture of our current PCF prototype.

Snapshots: A snapshot is a point-in-time state of a data element. The technology to take snapshots has made considerable progress over the years. Today, snapshots can be taken at the storage, file-system, and database levels with minimal overhead on the production system. The typical use of a snapshot is to capture a state of a data element such that we can roll back to that state at any point of time.

Note that snapshots are different from backups. Unlike snapshots, backups always represent additional copies of the production data. *Copy-on-write* and *redirect-on-write* are two efficient techniques to take snapshots [15]. These techniques will create a new copy of a data element e only if the production system modifies e after the snapshot is taken.

PCF's *Snapshot Manager* takes efficient point-in-time snapshots of data elements in order to run access-separated checks. Our current implementation of the Snapshot Manager creates a snapshot of all the data in the software stack comprising MySQL, XFS, and EBS on the Amazon Cloud as follows [16]:

1. Take a global read lock on the database, and flush the DBMS-level dirty data to disk. This step uses MySQL's "FLUSH TABLES WITH READ LOCK" command with a preliminary flush done to reduce locking time.
2. Lock the file-system to prevent accesses, and flush its dirty data to disk. This step uses XFS's xfs_freeze command with a preliminary sync done to reduce locking time.
3. Create a snapshot of the EBS volume. This step uses the ec2-create-snapshot command provided by the Amazon Cloud. Once the command returns a *SnapshotID*, release the locks on the file-system and the database.

Steps 1-3 above complete in *less than a second* to return the SnapshotID. The snapshot is ready for use by the PCF at this point. In the background, the Amazon Cloud will create a backup of the snapshot on S3 storage. (The current Amazon Cloud API binds snapshots and backups together, and does not enable snapshots to be taken without backing up the data in the snapshot.) The data transfer from EBS to S3 is optimized by copying only the blocks that have changed since the previous snapshot; only the first snapshot will copy all the data blocks. The overall impact on the production system is minimal.

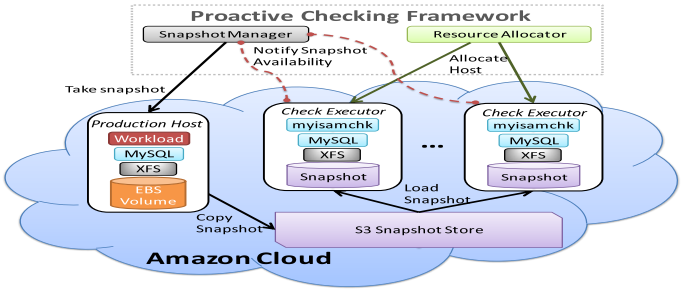

Fig. 2. Architecture of the Proactive Checking Framework (PCF)

The Snapshot Manager can be configured by the DBA to take snapshots at any rate. The default rate is one snapshot every ten minutes. The PCF has a configurable notification component that sends SnapshotIDs to the *Check Executors* for the snapshots on which checks should be run. By default, for every snapshot taken by the Snapshot Manager, a notification is sent to all Check Executors when the corresponding SnapshotID is available.

Check Executors: The PCF takes a list of access-separated checks as input. For each check, a computing unit called a Check Executor is allocated by PCF's Resource Manager to run the check on snapshots collected by the Snapshot Manager. In our current implementation, each Check Executor runs on a unique EC2 instance. The type of the instance is a configurable parameter that defaults to the EC2 large instance.

A Check Executor is in one of two states at any point of time: (i) *waiting state*, where it is waiting for the next SnapshotID notification; and (ii) *running state*, where it is running its assigned check on a snapshot. When a Check Executor receives a SnapshotID notification in the waiting state, it goes to the running state and takes the following steps:

1. The snapshot is loaded on the EC2 instance where the Check Executor is running. On the Amazon Cloud, snapshot data is accessible only through the EBS volume interface. This interface allows an EBS volume to be created from a SnapshotID. The volume is then attached to the Check Executor's EC2 instance for use.
2. The rest of the software stack (consisting currently of MySQL and XFS on top of EBS) is started, followed by the check. An Amazon Machine Image simplifies this step.
3. If the check detects a corruption, then any repair action associated with the check will be run. In this case, the Check Executor informs the PCF's notification component so that a notification can be sent to the DBA.

The Check Executor currently ignores new SnapshotID notifications received when it is in the running state. It goes back to the waiting state when the check completes.

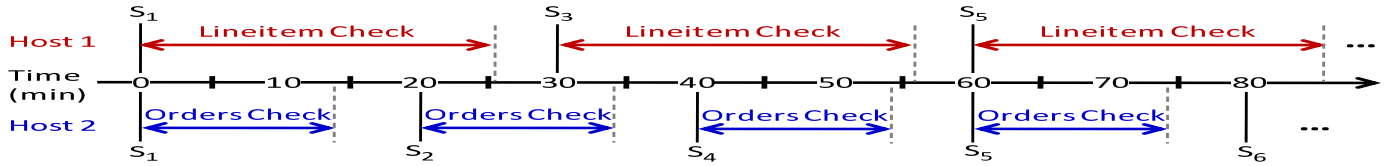

Fig. 3. An execution timeline of the Proactive Checking Framework on the Amazon Cloud. "myisamchk extended" (Table I-A) is used to run access-separated checks for the MySQL/XFS/EBS software stack

Figure 3 shows an execution timeline of PCF for the default settings. The Snapshot Manager collects snapshots from the production system described in Section II that runs a TPC-H workload on a MySQL DBMS, XFS file-system, and 50GB EBS volume on a large EC2 instance. PCF is given two access-separated checks to run: (i) "myisamchk extended" on the Lineitem table (denoted LineitemCheck), and (ii) "myisamchk extended" on the Orders table (denoted OrdersCheck).

PCF runs each check in a separate Check Executor. The Resource Manager allocates a EC2 host (of the default large instance type) to run each Check Executor: Host 1 runs LineitemCheck, and Host 2 runs OrdersCheck. Initially, both Check Executors are in the waiting state, and respond to the SnapshotID (denoted S_1) notification at Time 0 in Figure 3. Recall that the default SnapshotID collection and notification interval is 10 minutes. Both Check Executors are in the running state at Time 10, so they will ignore the SnapshotID notification at Time 10.

OrdersCheck on S_1 completes in around 14 minutes, so the Check Executor on Host 2 can run OrdersCheck on the SnapshotID (denoted S_2) at Time 20, the one at Time 40, and so on. LineitemCheck completes on average in around 25 minutes, so this check gets run once every three SnapshotID notifications when no repairs need to be done. Recall that any repair action associated with a check will be run by the Check Executor if a corruption is detected.

IV. Research Agenda to Deal with Corruption

The current implementation of PCF, although straightforward in principle, gives us a context to introduce a research agenda aimed at enabling DBMSs to handle data corruption automatically and proactively.

A. Specifying and Implementing Higher-level Requirements

PCF currently offers an interface to specify the access-separated checks to be run in a periodic fashion. This interface is low-level and focused on manual (DBA) use. A promising research direction is to raise the level of abstraction offered by PCF's interface to enable direct specification of higher-level requirements. An example requirement could be: "if the data used by the application's product recommendation feature gets corrupted, then revert this data back to a corruption-free version that is less than 1 hour old."

Such higher-level requirements usually come from *Recovery Time Objectives (RTOs)* and *Recovery Point Objectives (RPOs)* that exist at the application level. If the application were to crash or otherwise go down due to some cause (e.g., data corruption), then RTO limits the maximum tolerable window of application downtime. RPO limits the maximum tolerable window of recent data updates that can be lost when the application is brought back online. As we consider extending PCF to meet higher-level requirements, a number of challenges arise regarding what inputs to specify and how:

- Can a declarative language be developed to specify consistency requirements for data as well as recovery requirements if corruption were to occur?
- How can dependencies for checks be expressed, e.g., run check c_1 only if check c_2 fails or a block read error message is seen in the DBMS error log?
- A number of questions related to the past and future execution of the DBMS arise when a corruption is detected by an access-separated check on a data element e: (i) Can this corruption be ignored safely because the corruption will be handled by the system's access-blended checks if e were to be read in future? (ii) Has this corruption caused an incorrect result to be returned for a query in the past? (iii) Has this corruption already caused some data loss? and (iv) What actions should be taken to prevent future data loss due to this corruption?

B. Optimized Execution of Checks

The current implementation of PCF treats a check as a black-box script. Thus, checks can vary over the spectrum from checksum-based bit-level consistency checks, to checks for constraints that applications are supposed to enforce, and to auditing checks based on data-mining analysis of modifications to the data. There are plenty of opportunities for intelligent planning of check execution to meet the given requirements efficiently, especially in a pay-as-you-go cloud setting like the Amazon Cloud where resource usage is charged using resource-specific pricing models.

Suppose the actual requirement in the case of Figure 3 is to run LineitemCheck and OrdersCheck at least once per time interval of 60 minutes. In this case, allocating a single EC2 host h and running the checks one after the other on h would have been the optimal *checking plan*. This checking plan would no longer be satisfactory if the time interval was 30 minutes instead of 60. Note that running LineitemCheck and OrdersCheck one after the other on an EC2 large instance requires around 39 (25+14) minutes. The cost-optimal checking plan now could be to allocate two EC2 large instances as in Figure 3, or to run the checks concurrently on a single EC2 host with more resources than the large instance.

For optimized execution of checks, it is also important to ensure that the data needed by the check reaches the Check Executor efficiently. We found that the checks in Figure 3 will run around 2x faster if the data blocks they access in the EBS snapshot are already present on the EC2 host where the checks run. Since PCF currently uses snapshots as implemented in the Amazon Cloud, the required data blocks are pulled to the Check Executor as they are requested. Instead, we could imple-

ment a *push-based* architecture in PCF similar to how changed blocks from the production DBMS are replicated continuously to a hot-standby DBMS. While a push-based model will place higher overhead on the production DBMS, significant savings in check execution time as well as cost savings in a pay-as-you-go setting can result. We have implemented a similar architecture to meet DBMS tuning needs [17].

Data transfer costs can be avoided if we enable checks to be run on the production DBMS with minimal interference on the production workload. Adaptive resource-management techniques used to run administrative utilities like backups on the production DBMS could be used to run checks [18]; especially since efficient snapshot creation eliminates lock contention issues (Section III). With CPUs in enterprise servers moving towards 16-64 cores, it may become feasible to dedicate some cores to the critical task of data protection. The (additional) resource consumption from running access-separated checks on the production DBMS could be reduced by piggybacking on the I/O done for regular workload processing.

C. Rewriting Checks for Faster and Incremental Execution

The access-separated checks in wide use today were not written to execute efficiently on a continuous stream of data snapshots, presenting many promising opportunities [19]:

- These checks are written to run from scratch each time. Incremental versions of the checks can be developed to exploit how snapshot collection generates changes between successive snapshots. This idea has to be used with care since silent data corruption (e.g., bit flips, misdirected writes) can occur in blocks that are not updated explicitly.
- Most checks use a single-threaded execution model. Exploiting intra-check and inter-check parallelism—e.g., for efficient use of multicore hosts—has tremendous scope.
- It is possible to eliminate redundancies as well as to apply cross-optimizations for checks run at different levels of the software stack. For example, checks at the DBMS and file-system levels may both (redundantly) verify bit-level consistency of data in the same disk block. A file-system-level check that accesses certain files could be co-scheduled for better RAM and I/O utilization with DBMS-level checks that access data stored in the same files.
- Cross-optimizations may also be possible between access-separated checks and the access-blended checks that are invoked during regular workload processing. Exploiting patterns in the workload may be useful in this context.

D. Robust Check Execution under Unpredictable Events

In a setting like the Amazon Cloud, hosts could become slow or fail in an unpredictable fashion. (These hosts are actually virtual machines that are multiplexed along with multiple other virtual machines on the same physical infrastructure.) PCF should adapt to such unpredictable events while meeting higher-level requirements in the best way possible. Even the need for repairs are unpredictable events. Furthermore, production systems change over time, e.g., through hardware upgrades and software patches. Each change has a potential risk of corrupting data. PCF should be able to react to changes by deciding which extra checks (if any) to run and for how long. For example, when a patch is applied to the XFS file-system, xfs_check could be run more often than usual.

E. Automated Recovery from Data Corruption

Automated recovery greatly enhances the value of automated discovery of data corruption. Depending on the type of corruption discovered in an element e, one or more recovery strategies may be applicable such as:

- A corruption-free copy of e may exist due to replication.
- Replace the corrupted blocks of e from a previous snapshot[1], and redo block-level changes from the log.
- From a previous snapshot, load the database table that e belongs to. Redo required SQL-level inserts, deletes, and updates from the log. Rebuild all indexes on the table.
- Perform a complete disaster recovery starting by restoring the database and system stack from a backup.

The challenge is to choose the best strategy that will fix the corruption and meet the specified RTO and RPO. Information gathered during check execution can guide this choice.

V. Conclusions

In this paper, we pointed out the causes of data corruption as well as its consequences which can be as severe as entire companies going out of business. We outlined a challenging research agenda that will ultimately enable database systems to deal with data corruption automatically and proactively.

References

[1] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: a Large-scale Field Study," in *SIGMETRICS/Performance*, 2009.

[3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "An Analysis of Data Corruption in the Storage Stack," in *FAST*, 2008.

[4] B. Schroeder and G. A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1, 000, 000 Hours Mean to You?" in *FAST*, 2007.

[5] *Data corruption in Amazon S3*, glennbrunette.sys-con.com/node/991097/mobile.

[6] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity Lost and Parity Regained," in *FAST*, 2008.

[7] *Data corruption in CouchDB*, couchdb.apache.org/notice/1.0.1.html.

[8] *Log Corruption in HDFS*, issues.apache.org/jira/browse/HDFS-909.

[9] forums.oracle.com/forums/thread.jspa?threadID=1102491&tstart=45.

[10] *Data corruption at Ma.gnolia.com*, en.wikipedia.org/wiki/Gnolia.

[11] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton, "Impact of Disk Corruption on Open-source DBMS," in *ICDE*, 2010.

[12] Y. Zhang *et al.*, "End-to-end Data Integrity for File Systems: A ZFS Case Study," in *FAST*, 2010.

[13] M. Ault and M. Tumma, *Oracle 10g Grid & Real Application Clusters*, 1st ed. Rampant Techpress, 2004.

[14] *Amazon Web Services*, aws.amazon.com.

[15] *Snapshots Reinvented*, IBM, September 2008, ftp://public.dhe.ibm.com/storage/disk/xiv/ibm_xiv_snapshots_paper.pdf.

[16] *Running MySQL on Amazon EC2 with EBS*, aws.amazon.com/articles/1663.

[17] S. Duan, V. Thummala, and S. Babu, "Tuning Database Configuration Parameters with iTuned," in *VLDB*, 2009.

[18] S. S. Parekh, K. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang, "Managing the Performance Impact of Administrative Utilities," in *DSOM*, 2003.

[19] H. S. Gunawi *et al.*, "SQCK: A Declarative File System Checker," in *OSDI*, 2008.

[1] After a read-only snapshot/backup has been verified using access-separated checks, we can rely exclusively on access-blended checks to catch any future corruption in the snapshot/backup.