Cost of Ensuring Safety in Distributed Database Management Systems

Maitrayi Sabaratnam, Maitrayi.Sabaratnam@idi.ntnu.no
Norwegian University of Science & Technology(NTNU)

Øystein Torbjørnsen, ClustRa AS
Oystein.Torbjornsen@clustra.com

Svein-Olaf Hvasshovd, NTNU
Svein-Olaf.Hvasshovd@idi.ntnu.no

Abstract

Generally, applications employing Database Management Systems (DBMS) require that the integrity of the data stored in the database be preserved during normal operation as well as after crash recovery. Preserving database integrity and availability needs extra safety measures in the form of consistency checks. Increased safety measures inflict adverse effect on performance by reducing throughput and increasing response time. This may not be agreeable for some critical applications and thus, a tradeoff is needed.

This study evaluates the cost of extra consistency checks introduced in the data buffer cache in order to preserve the database integrity, in terms of performance loss. In addition, it evaluates the improvement in error coverage and fault tolerance, and occurrence of double failures causing long unavailability, with the help of fault injection. The evaluation is performed on a replicated DBMS, ClustRa [9].

The results show that the checksum overhead in a DBMS inflicted with a very high TPC-B-like workload caused a reduction in throughput up to 5%. The error detection coverage improved from 62% to 92%. Fault injection experiments shows that corruption in database image went down from 13% to 0%. This indicates that the applications that require high safety, but can afford up to 5% performance loss can adopt checksum mechanisms.

1 Introduction

A system behavior can be specified by standard semantics, failure semantics, and stochastic behavior [4]. **Standard semantics** describes how the system intend to function. **Failure semantics** describes in what ways the system can fail. Defining the failure behavior of a system is essential for designing the recovery actions in a fault tolerant system. Failures outside the failure semantics are defined as **catastrophic failures**. **Stochastic behavior** expresses the minimum probability for the standard behavior and the maximum probability for a catastrophic failure. **Safety** is defined as non-occurrence of catastrophic failures [10].

Failure semantics are classified into performance (giving late response), response (value failure- giving wrong response, state failure- performing wrong state transition), omission (not responding to an input), and crash (after omitting a response, no further response is given until the crashed server is restarted) failure semantics [4]. A crash is classified further into **amnesia** crash (restart from a predefined state independent of the state prevailed during the crash) and **atomic** crash (restart from the state at crash).

A DBMS server obeying ACID[1] [7] properties have performance, omission, and atomic crash failure semantics. In other words, DBMSs may give a belated response to a client request, e.g., due to resource contention. They may omit a response when a transaction is aborted, e.g., in order to break a dead lock. They omit responding to client requests during server crashes. A DBMS recovers from a crash at its last committed state. The requirements concerning response time is included into the DBMS standard semantics. The upper bound for recovery time or unavailability is included in the failure semantics.

Response failures and non-atomic crash failures are outside the DBMS failure semantics. Value failures occur when the DBMS returns an incorrect response to a client request. State failures occur when the database image does not conform to the changes made or intended to be made by the committed transactions. This can happen due to residual software faults that perform incorrect updates or recovery, or corrupt the database image accidentally as a result of a wild pointer. An inconsistent database image can be contagious, when newer updates are performed based on the infected values. Non-atomic crash recovery may leave the

[1] Atomicity- either all the operations belong to a transaction are executed or none of them are executed; Consistency- a transaction leads a database from a consistent state to another consistent state; Isolation- the changes made by a transaction will not be visible to other transactions until this transaction commits; Durability- the effect of the committed transactions will not be lost.

database image in an inconsistent state after the crash recovery; Sometimes, recovery may cause longer recovery time or unavailability than that specified in the failure semantics. Such catastrophic events may cause severe economic or fatal consequences. In this study, we define corruption in database image and long DBMS unavailability during recovery (see Sec. 5.5 for *double failures*) as **catastrophic events**.

Fault injection studies performed on DBMSs demonstrate the possibility of safety violations in DBMSs. Ng and Chen [12] showed that the database got corrupted in 2.3% to 2.7% cases even when a reliable memory for caching data was integrated into POSTGRES DBMS. A study conducted by the authors on ClustRa DBMS points out the necessity of introducing some form of consistency check in order to guard the database image from corruption [13]. Consistency checks can be introduced in the software at different levels to achieve different safety levels. But they do not come for free. The more the consistency checks, the higher the adverse impact on transaction response time. This may not be agreeable for some critical applications. Therefore, it is necessary to quantify the cost of different safety levels in order to determine the cost-performance tradeoff.

The primary purpose of this study is to evaluate the cost of introducing consistency checks (Sec. 2) to prevent corruption in database image in terms of performance loss (Sec. 4). In addition, it evaluates the effectiveness of the consistency checks in guarding the database from corruption, with the help of fault injection (Sec. 5). ClustRa DBMS [9] V2.0(beta) is used as the test bed for the experiments (Sec. 3). Sec. 6 summarizes the results and presents the conclusion derived.

2 Consistency Checks for DBMSs

In this section, we shall look at the checks or error detection capabilities that can be incorporated into the DBMS software. This excludes the operating system capabilities that detect address or data access violations, or the programming language type checking capabilities at runtime.

2.1 Checksums

One way to detect errors is to incorporate consistency checks, assertions, and exceptions wherever appropriate, based on the semantic rules pertaining to DBMSs. Another way is to attach checksums to the important, DBMS-specific objects, such as, data buffer, log record, lock, transaction, message, and execution-instruction [8][2]. These two ways are complementary and implementing both will strengthen the robustness of the DBMS, but as we mentioned earlier, the cost will be too high and therefore, a choice among the prescribed consistency checks is necessary.

The data buffer is a key data structure in a DBMS. Part or whole of the database image is cached here for processing. The ultimate corruption caused by malignant hardware or software that can give catastrophic impact on integrity is the one that occur in database image. Sullivan and Stonebraker [16] studied the use of hardware memory protection to guard the data buffer cache as an effort to improve the software fault tolerance in POSTGRES DBMS. The overhead was 7-11% of the processing time for CPU-bound workload. Our study evaluates the cost of introducing checksums in the data buffer structure and the improvement on error detection and fault tolerance after the introduction of checksums. Our study concentrates on safety problems given that an error is occurred in data buffer area, rather than the fault behavior or the mean time to catastrophic failures.

Checksums are generally used to detect errors and not to mask them. A checksum attached to an object is calculated whenever the object is updated and is stored within the object or at another fault tolerant location. when the object is accessed, its checksum is recalculated and compared with the stored checksum. The probability that a 32-bit checksum does not detect an error is as low as $2.3 * 10^{-10}$ [8]. Therefore, one can safely assume that most of the overlay that are not detected by the operating system checks or the runtime type checking will be detected by the prescribed checksum mechanism.

Checksums can be added to the objects (e.g., a transaction, lock, log, or data buffer) as well as the access methods (e.g., hash, B-tree, etc.) to the objects. Fig. 1(a) shows a generic data structure of objects and an organization structure to access them. Fig. 1(b) shows transaction objects and a hash table as the access method. A transaction object can be accessed by the hash access method. Here, a checksum field is attached to the access method as well as to each of the objects found in the hash buckets. The hash vector and object checksums must be updated appropriately. Checksums must be recalculated and checked against the stored checksums, whenever the objects are accessed. Checksums connected to access methods may be tested only at the

[2] *Data buffers* in the memory are used as temporary working area. Database images stored on disk pages are retrieved from the disk and cached in the data buffers before a transaction performs any operations on the data. *Log records* register the changes made by the transactions on the database image and are used by aborting transactions or recovery to bring the database image to a consistent state. *Locks* are used to serialize the concurrent transaction operations that conflict on the same data item. *Transaction* structures encapsulate the state and other information pertaining to a transaction. *Messages* include *execution-instructions* and other information exchanged between participant and coordinator process involving in a transaction protocol, as well as communication between the client and the DBMS server. For example, the checksum attached to a log record can be used to detect corruption of the log record when it is being used in recovery or transaction aborts.

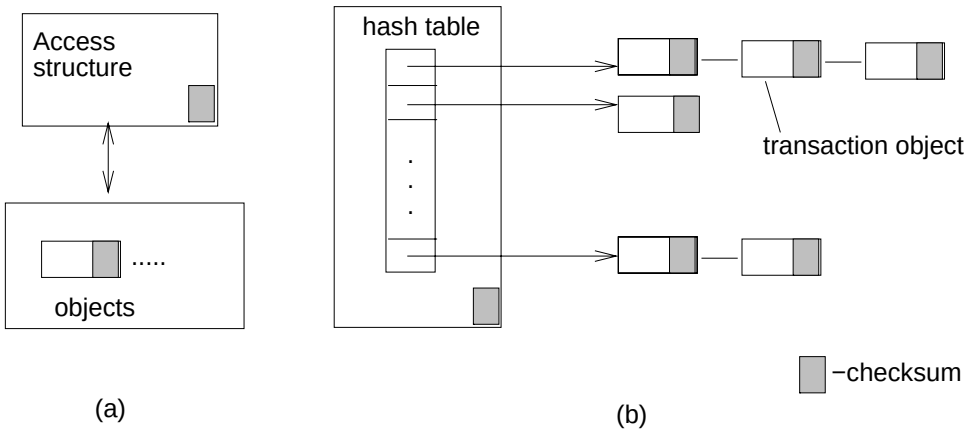

Figure 1. (a) Checksums are attached to generic objects as well as to their access method. (b) An example of transaction objects accessed by hashing the transaction identities.

restart of a DBMS instead of each access, for performance reasons (compromising error latency).

Class of errors detected by checksums: A **checksummed object** is defined as the object covered by a checksum. An **update-time-window** is defined as the time period between an update of an object and the update of the corresponding checksum. Checksums help to detect accidental corruption taking place on checksummed objects outside the update-time-window. Such corruption can be caused by malignant software faults, such as, wild pointers. (Sullivan and Chillarege have presented the common software faults that causes overlays [14, 15, 3] based on field error reports for operating system IBM MVS and database management system IMS). Errors that occur concurrently during the update-time-window will *not* be detected by the checksum mechanism. This class of errors are not covered by the hardware memory protection mechanism adopted in [16] either. A good example for corruption that are not detected by the checksum are the faults in the software that performs the updates wrongly and then updates the checksum accordingly. In addition, comes the traditional class of errors that is not covered by checksums, e.g., a double bit-flip on the same bit-position of two words where an XOR of machine-words is used as the checksum.

2.2 Checksum Granularity in Data Buffers

Since we concentrate on data buffers, we shall look at its structure before we proceed. A data buffer area consists of used and unused data buffers. A data buffer accommodates a disk page containing part of the database image. Each buffer has a header part and a data part as shown in Fig. 2. Data part consists of data records. Header part consists of administrative data needed to maintain the data part as well as to maintain the data structure (see Fig. 6 of the buffer area.

Checksum Granularity and Computation Cost:

Finer granularity checksums, e.g., those attached to parts of an object, reduce computation cost but increase error latency. For example, a data buffer usually has a header part of administrative data and a body part containing records. Different parts are accessed for different purposes. Attaching checksums to parts reduces computation costs at the access, but trades off error detection in the unaccessed part. For example, assume a data buffer page of 4K, having data records with the size of 50 bytes and a header with the size of 40 bytes. It may suffice for a transaction accessing only one record from a page, to recompute the checksums only for that record and the header, and check them against the corresponding stored-checksums (see Fig. 2(b)). If the checksum was attached to the buffer page level (see Fig. 2(a)), then for each record access, the checksum for the whole page have to be recomputed. This will take an order more instructions. But checking the whole page will reduce the error latency for the other non-hot-spot data records residing on that page. The choice of the first method needs more data storage capacity than the second method.

3 ClustRa as the Test Platform

ClustRa is a distributed DBMS providing highly available, high performance, scalable, and fault tolerant platform for applications requiring soft real-time response time, e.g., applications in telecommunications. It is a replicated DBMS with a shared-nothing architecture, running

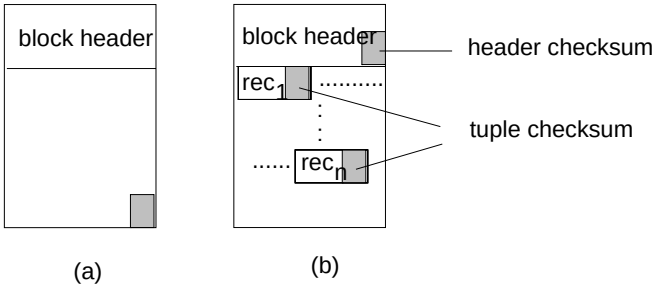

Figure 2. Checksums attached to different granularities: (a) buffer page (b) buffer header and records.

on COTS UNIX or NT work-stations. It achieves fault tolerance by replicating data and DBMS process on different computers connected by duplicated communication lines with high-bandwidth. A ClustRa process is referred to as **ClustRa-node** or simply **node**. Single node and communication line failures are masked. High availability is achieved by a supervisor process monitoring the ClustRa DBMS process at each node. ClustRa process is restarted by the supervisor if it does not show any sign of life within a timeout period. Besides, the supervisors at different nodes exchange *I'm alive* messages. A virtual partition management protocol is used to maintain a consistent set of available nodes and services[1, 9].

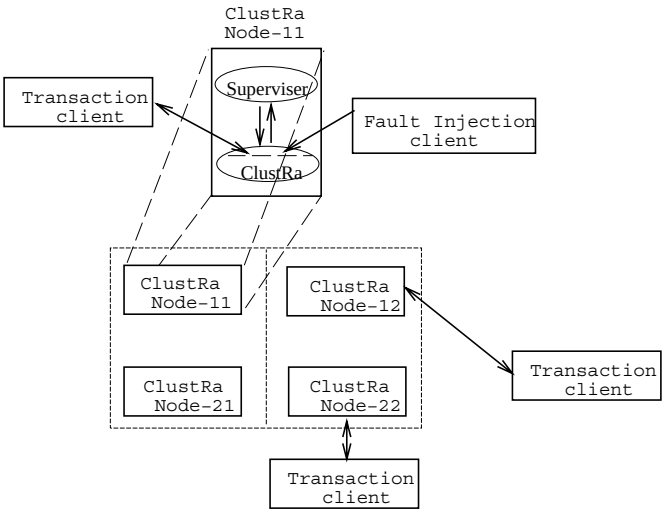

Figure 3. The experiment setup (ClustRa Node-11 is zoomed). Initially, Node-11 and Node-12 are active, and Node-21 and Node-22 are spare.

A four-ClustRa-node cluster running on two SUN work stations is used in the experiment. Each work station has two 200 MHz UltraSPARC processors and runs two ClustRa processes (one active and spare), as illustrated in Figure 3.

ClustRa-Node-11 and ClustRa-Node-12 are **active**, i.e., having data and ClustRa-Node-21 and ClustRa-Node-22 are **spare**, i.e., having no data. Node-11 and Node-12 are **counterpart nodes** containing replicas of the database, i.e., for the data fragment Node-11 has the **primary replica**, Node-12 has the **hot standby replica**, and vice versa. If a node having a primary replica fails, its counterpart hot standby node **takes over** the role of the primary. The crashed node is restarted by the supervisor process. When it finishes recovering the database content and catches up with the changes made on the database while it was down, with the help of the counterpart node, it **takes back** the primary role in order to balance the load on the nodes [2]. If the crashed node could not repair within a timeout period, then the spare node copies data from the acting primary and **takes back** the primary role.

There are two sets of experiments conducted in this study: **Cost evaluation** experiment and **fault tolerance evaluation** experiment. The first one evaluates the cost of the checksum, introduced in data record and data buffer header levels, as described in Sec. 2, measured in terms of performance loss. The other one evaluates the efficiency of the fault tolerance of ClustRa - detecting errors, masking node failures, and repairing the failed nodes. Detection of a malignant error leads to the crash of a ClustRa node where the error is detected. A node crash is masked by the hot standby node taking over. The crashed node or the spare node repairs itself and takes back role of the crashed node. This reestablishes the fault tolerance level of the system.

4 Experiments for Cost Evaluation

The experiment conducted to evaluate the cost of checksums introduced in data buffer structure, is described in this section. Subsection 4.1 presents the workload generated on the system during the measured period. Subsection 4.2 describes the experiment structure, steps performed during an experiment run, and the measures collected. Subsection 4.3 analyzes the results.

4.1 Workload Generator (WG):

The checksum overhead is expected to give less impact on performance when the load at the system is low than when it is high. In order to evaluate the checksum overhead at different loads, different workloads are generated by,

1. varying the number of transaction clients (TC) using the DBMS concurrently and

2. the number of data records a transaction updates. When the number of records accessed by a transaction is low, overhead for sending and receiving messages dominate the response time rather than the calculation cost for checksums.

Twenty different workloads are generated by combining the number of transaction clients (2, 4, 6, and 8) and the number of records accessed by each transaction (1, 2, 4, 6, and 8). Four-record access transaction resembles TPC-B standard [6]. A TC is allowed to function for a maximum of 300 seconds. Access to a data record is uniformly distributed, having no hot-spots. The database size is proportional to the number of TCs, where each TC adds around 500KB to the database. The workload chosen is CPU-bound, i.e., the disk latency is avoided.

4.2 Experiment Process:

Forty experiments are conducted on two DBMS implementations- one with checksums and the other without checksums- using the 20 workloads described above. Each experiment consists of 10 runs, totaling 400 runs. Each run takes around 10 minutes of clock time, giving 66 hours.

An experiment run consists of starting the DBMS and the workload generator, stopping the DBMS, and analyzing the logs, and is conducted by an experiment manager (EM). ClustRa processes are started on four ClustRa nodes as illustrated in Figure 3. The workload generator is started 240 seconds after the DBMS is started. The workload is maintained for around 300 seconds, and then the DBMS is stopped. Then, the information found in the DBMS log is analyzed and the performance metrics is calculated.

The performance information logged by the DBMS are 1) throughput, 2) response time, and 3) the % of transactions that finished within 15 milli-seconds. The last one can be used to measure the ability to handle the load. The information is calculated and written out for every 10-second time window. The data belonging to the stable workload period is extracted, i.e., the data belonging to the first three and the last three 10-second-windows are left out. Throughput and response time for an experiment is calculated by taking the average for the stable period and again for the ten runs.

4.3 Results

Figures 4 and 5 show the impact of checksum overhead in throughput and response time respectively. As mentioned earlier, the impact of checksum overhead is more significant in high transaction load than in low load. The maximum performance loss observed is 7%.

The difference in response time in 6-client case for 4, 6, and 8 record-access are: 391, 683, and 825 micro-seconds

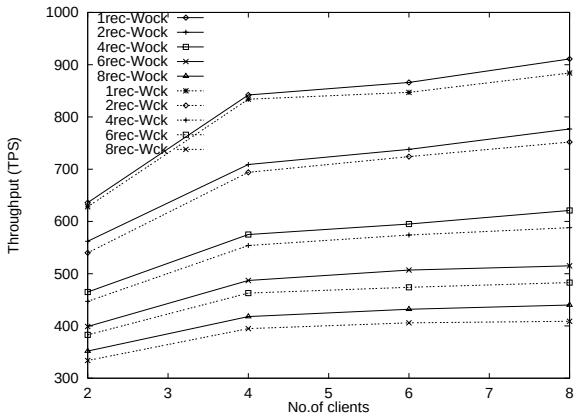

Figure 4. Impact of checksum overhead in throughput.

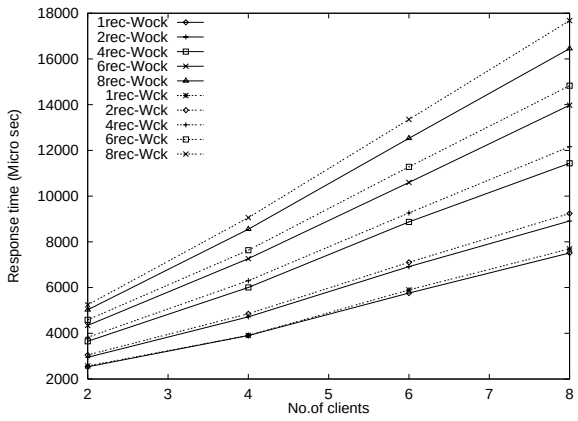

Figure 5. Impact of checksum overhead in response time.

respectively. The figures corresponding to 8-client case are: 726, 852, and 1228 micro-sec. The corresponding figures for throughput are: 21, 33, and 36 transactions for 6-client case and 33, 32, and 31 transactions for 8-client case. The system reaches a saturated state when the load increases beyond a limit. This occurred in 6-client-8-record-access, 8-client-6-record-access, and 8-client-8-record-access cases, where the % of transactions finished within 15 mil.sec. interval sank lower than the 95% limit (lowest observed is 55% in 8-client-8-record-access-with-checksum case). This thrashing effect is clear in the constant throughput differences in 8-client cases and 6-client, 6 and 8 record-access cases.

5 Experiments for Fault Tolerance Evaluation

5.1 Fault model:

The fault injection experiment is designed to validate the robustness of the data buffer guarded by checksums. The underlying fault model is *transient software overlay errors that corrupt the data buffer area*. There are 2 reasons for the choice: 1) As mentioned in Sec. 2, the main goal of the DBMSs is to maintain the integrity of the database image and overlay errors in the buffer area may cause severe integrity problems. Any software faults that ultimately result into corrupting data buffer area directly or by propagation are of interest for the study. 2) As pointed out by Sullivan and Chillarege [14] in a study performed on operating systems and DBMSs, the software faults causing overlay errors are hard to trace and their impact are much higher than the regular errors.

Common software faults causing overlay errors are: assignment faults, initialization faults, wild pointers, copy overflow, type mismatch, memory allocation, and undefined state [14, 15, 11]. Overlay errors may corrupt any part of code or data. The data buffer corruption is a subset of the errors resulting from the software faults causing overlays. We inject errors directly into the data buffer area in order to accelerate the database corruption, instead of injecting faults that result into direct or indirect corruption in data buffer area. The drawback of injecting errors in this way is that the mapping between representative faults and the resulting errors may not be captured in a representative manner. Christmansson and Chillarege [5] have addressed issues relevant to generating representative error sets that can be used in fault injection experiments, based on the field defect data for IBM OS.

5.2 Workload Generator (WG):

For the fault tolerance evaluation experiment, the WG starts four parallel transaction clients (TC). The TCs send single-record transactions in a back-to-back manner, where each transaction performs either insert, delete, update, or read operation on one record. The total load consists of 55% update transactions, 20% reads, and the rest 25% is distributed among inserts and deletes such that at equilibrium, 75% of the inserted data exists. The size of the database is around 5 MB at equilibrium. Further, the transaction load is selected in order to exploit the system resources like CPU at the server at a considerably high level, but at the same time give enough spare capacity to enable the take-over processing. The workload is designed to accelerate the access of a corrupted location by reducing the database size and increasing the number of transactions accessing the corrupted records.

5.3 Experiment Process:

Different types of errors are injected. They corrupt different components of a randomly-chosen data buffer in use in order to accelerate the error activation. Buffers are arranged in a B-tree structure, as shown in Figure 6(a). The header part of a buffer (see Figure 6(b)) consists of the following administrative data: buffer identifier, number of records in the buffer, number of free bytes in the buffer, high water mark - the position where a new record will be inserted, and a pointer to its next buffer (found in leaf buffers only). A record also has an administrative part and a data part. Administrative part contains: a key descriptor describing the number of fields used to identify a record uniquely and an administrative descriptor stating whether a record has the knowledge about the log record that contains information about the last change a transaction made on the record and whether the record has checksum. Data part of an index record contains access path to a buffer in the next level of the tree. Data part of a leaf buffer contains user data. In the following sections, errors overlaid on these components are referred to as error types: **BufferId, NoOfRecords, NoOfBytesFree, HighWater, NextPointer, KeyDescriptor, AdmDescriptor, NextLevelPointer**, and **UserData**.

There were 30 runs conducted for each of the nine error types mentioned above. The fault injection experiment was conducted for the DBMS with checksum, totaling 270 runs. The clock time used by each run is around fifteen minutes, giving a total of 68 hours. In order to create different fault scenarios as possible, each run was started with a different seed such that the choice of a buffer to be corrupted and the injection point in time varied. Therefore, the experiment did not evaluate the repeatability of the error impact.

An experiment has the same steps as that of the one described in Section 4. In addition, the following aspects are added: a fault injector client is started 30 seconds after the WG is started. Database content is dumped at the end of the experiment before the DBMS is stopped. The TCs kept a local copy of the database and perform the transactions on the local database, just to check the ClustRa behavior. They checked the content of the replies sent from the DBMS with their own. Any discrepancies are logged. In the analysis phase, TC's database content is compared with the DBMS's, and the logs from the TCs is scanned in order to find discrepancies.

5.4 Fault Injector:

Fault injector has a client part and a server part. A fault injector client (FIC) injects an error type mentioned ear-

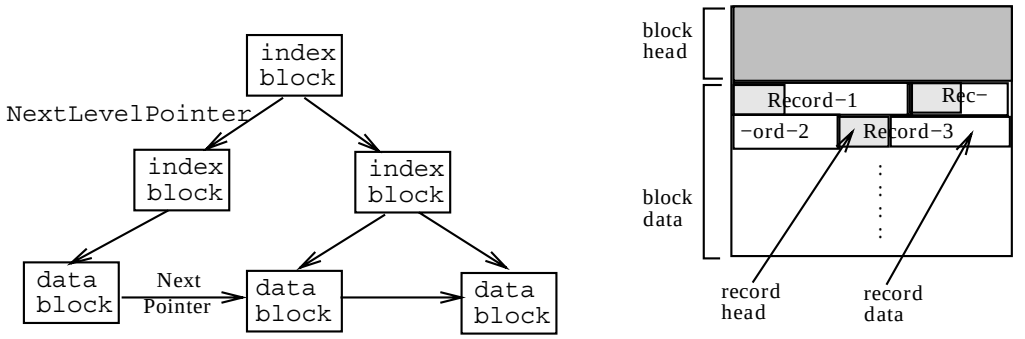

Figure 6. a) Arrangement of buffers in a B+-tree structure. b) the layout of a buffer.

lier into the data buffer area. Like TCs, FICs can also be started on a non-ClustRa node. A FIC is started at a point in time distributed uniformly between 30-60 seconds after the workload generator is started. FIC is given the error type and the relevant parameters by the EM. For example, if the error type is BufferId, then the parameters will be a seed value used by EM to repeat the run if necessary, a data table identifier, and a flag saying whether the chosen buffer is an index buffer, a data buffer, or any of them. FIC then sends a message containing the fault injection request together with the parameters. The server part of the fault injector is an extention of the DBMS server interface that handles requests from a FIC. At the receipt of the message from a FIC, this request is handled like the requests from the TCs. To process this request a small piece of software code is added to the DBMS server to a) access a buffer component as specified by the parameters and b) overlay a random bit pattern accordingly. This takes very few machine instructions. In addition, comes one extra message-receiving cost per run.

5.5 Results

In this section, fault tolerance and safety metrics gathered from the experiment are presented. It is compared with corresponding data presented in [13], where the DBMS operated without checksum mechanism. The improvement in the metrics after the introduction of checksum measures is discussed. At the end, the reliability growth in the software between the versions is briefed.

Fault tolerance issues: Coverage parameters for error detection, masking, and establishing of the fault tolerance level are evaluated and presented in Table 1. **Error detection coverage:** The second column shows the number of runs where the injected errors are detected. All the injected errors are detected by the checksum mechanism except the 22 errors that went undetected in the case of AdmDescriptor error type. AdmDescriptor indicates whether a checksum test is required. This gives flexibility to applications.

Error Type	# errors detected	# failures masked	# online repair
NextLevelPointer	30	30	30
UserData	30	30	30
KeyDescriptor	30	30	30
AdmDescriptor	8	8	6
BufferId	30	30	30
NextPointer	30	30	30
NoOfRecords	30	30	30
NoOfBytesFree	30	30	30
HighWater	30	30	30
Total	248	248	246
Coverage	(92%)	(100%)	(99%)

Table 1. Error coverage.

Checksum mechanism can be turned off dynamically or can be assigned to only parts of data in the database. When the code was scrutinized, it is found that only two bits are allocated to the AdmDescriptor component which has 0-3 as the legal values. Any corruption of it will lead to a legal value, but in the worst case, a corruption can turn the checksum test off. Fortunately, this corruption itself will not lead to any problem. But any eventual corruption in other parts of the record may go undetected. Solution is to have the checksum mechanism permanently or include it with the help of a compiler option, but this will reduce the flexibility enjoyed by the applications.

Error masking coverage: All the detected errors lead to the ClustRa process to be crashed. This node failure is masked by the takeover by the counterpart node.

Reestablishing the fault tolerance level: When a node is crashed, the system functions with reduced fault tolerance level until the repair of the crashed node is finished. To reestablish the fault tolerance level as soon as possible, the crashed node is restarted by the node supervisor and it performs online repair. The system is vulnerable during this

Metric	Without check, SW version 1.1	With check, SW version 2.1
Detection coverage	62%	92%
Data corruption	13%	0%
Double failures	1%	0%
Masking coverage	98%	100%
Unsuccessful repair	15%	1%

Table 2. Improvement in fault tolerance and reliability growth.

period, since the crash of the counterpart node will lead to a **double failure** which is catastrophical. When a double failure occurs, a spare node cannot take over online, since it has no active source node to copy data from. This requires recovery from a backup medium which may lead to longer unavailability and loss of the transactions committed recently.

Two cases of AdmDescriptor error type which went undetected in the primary node caused error propagation to the counterpart node which made it to crash. It could not finish repair within the rest of the experiment period, which was around 10 minutes.

Catastrophic events: As defined in Sec. 1, user data corruption and double failures are defined as catastrophic events that threatens the system safety. In the 270 runs, there were no double failures or user data corruption observed.

Table 2 shows the different coverage metrics when the system was operating with and without checksum mechanism and for different versions of the DBMS software (see [13] for more details of the data presented in column 2). Checksum mechanism contributes directly to the improvement of detection coverage and data corruption. Improvement in the software design and implementation resulting from the correction of the faults revealed by the fault injection experiment on version 1.1 as well as the other testing procedures contributed apparently to the improvement in the other metrics presented in the table.

6 Summary and Conclusions

In order to improve safety, checksums are attached to the data buffer headers and data records. Checksums are checked at each access of the buffer and record, and at flushing of a buffer from the cache to the disk. The latter prevents corruption infecting persistent data. The cost of improving safety is evaluated. The checksum overhead in a DBMS inflicted with a very high TPC-B-like workload caused a reduction in throughput up to 5%. The error detection coverage improved from 62% to 92%. Integrity of the database has improved; Fault injection experiments shows that user data corruption went down from 13% to 0%. This indicates that the applications that require high safety, but can afford up to 5% performance loss can adopt checksum mechanisms.

References

[1] A. Abbadi, D. Skeen, and F. Christian. An Efficient Fault-Tolerant Protocol for Replicated Data Management. *ACM Distributed Database Systems*, pages 259–73, 1988.

[2] S. Bratsberg, Ø. Grøvlen, S. Hvasshovd, B. Munch, and Ø. Torbjørnsen. Providing a Highly Available Database by Replication and Online Self-Repair. *International Journal of Engineering Intelligent Systems*, 4(3):131–139, 1996.

[3] R. Chillarege and N. Bowen. Understanding Large System Failures - A Fault Injection Experiment. *Proc. 19th. Ann. Int'l Symp. Fault Tolerant Computing*, pages 356–363, 1989.

[4] F. Christian. Understanding Fault Tolerant Systems. *Communications of the ACM*, 34(2), Feb. 1991.

[5] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proc. 26th. Ann. Int'l Symp. on Fault Tolerant Computing*, 1996.

[6] J. Gray, editor. *"The Benchmark Handbook for Database and Transaction Processing Systems"*. Morgan Kaufmann Publishers Inc., 1991.

[7] J. Gray and A. Reuter. *"Transaction Processing: Concepts and Techniques"*. Morgan Kaufmann Publishers Inc., 1993.

[8] S.-O. Hvasshovd and Ø. Torbjørnsen. Improved Safety in a Shared-Nothing Parallel DBMS. Research Report STF40 A93120, SINTEF DELAB, 1993.

[9] S.-O. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager. The ClustRa Telecom Database: High Availability, High Throughput and Real Time Response. In *Proceedings of the 21st VLDB Conference, Zürich*, 1995.

[10] J. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. *Proc. 15th. Ann. Int'l Symp. on Fault Tolerant Computing*, pages 2–11, 1985.

[11] I. Lee and R. Iyer. Faults, Symtoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. *Proc. FTCS-23*, pages 20–29, 1993.

[12] W. T. Ng and P. M. Chen. Integrating Reliable Memory in Databases. *Proc. of the 23rd VLDB Conference, 1997*, 1997.

[13] M. Sabaratnam, Ø. Torbjornsen, and S.-O. Hvasshovd. Evaluating the effectiveness of fault tolerance in replicated database management systems. In *Proc. 29th. Ann. Int'l Symp. on Fault Tolerant Computing*, 1999.

[14] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability- A Study of Field Failures in Operating Systems. *Proc. FTCS-21*, pages 2–9, 1991.

[15] M. Sullivan and R. Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. *Proc. FTCS-22*, pages 475–484, 1992.

[16] M. Sullivan and M. Stonebraker. Using Write Protected Datastructures To Improve Software Fault Tolerance in Highly Available Database Management Systems. *Proc. of the 17th VLDB Conference, 1991*, pages 171–180, 1991.