A study of online construction of fragment replicas

Fernanda Torres Pizzorno

June 30, 2005

Preface

In this report, I present the "hovedoppgave" of my 5th year in the "Siv. ing. i Datateknikk" course at NTNU. The project entitled "*A study of online construction of fragment replicas*", was elaborated by Dr. Ing. Øystein Torbjørnsen (Fast Search and Transfer, Trondheim) and Prof. Svein-Olaf Hvasshovd (NTNU, Trondheim). It was carried out at the "Institutt for Datateknikk og Informasjonsvitenskap", "Fakultet for Informasjonteknologi, Matematikk og Elektronikk" at NTNU, under joined supervision of Prof. Svein-Olaf Hvasshovd and Dr. Ing. Øystein Torbjørnsen.

I want to thank Prof. Svein-Olaf Hvasshovd and Dr. Ing. Øystein Torbjørnsen for their kind support during the whole process.

Trondheim, the 26th of june 2004

Fernanda Torres Pizzorno

Project text: A node failure in a multi-node shared nothing DBMS reduces the system's fault-tolerance level. It is therefore essential, for a DBMS providing high-availability based on a shared nothing platform, to be able to efficiently create a new fragment replica after a node failure. This project should address different methods for creating a new consistent fragment replica using a catch-up production method. The project should evaluate the completion time of the methods, under different conditions, on each of the nodes involved in the process. If possible a simulation of the network speed's effect on the methods should be performed.

Summary

High availability in database systems is achieved using data replication and online repair. On a system containing 2 replicas of each fragment, the loss of a fragment replica due to a node crash makes the system more vulnerable. In such a situation, only one replica of the fragments contained in the crashed node will be available until a new replica is generated. In this study we have investigated different methods of regenerating a new fragment replica that is up to date with the transactions that have happened during the process of regenerating it. The objective is to determine which method performs the best in terms of completion time at each of the nodes involved, in different conditions.

We have investigated three different methods for sending the data from the node containing the primary fragment replica to the node being repaired, and one method for catching-up with the transactions executed at the node containing the primary fragment replica during the repair process. These methods assume that the access method used by the DB system is B-trees. The methods differ by the volume of data sent over the network, and by the work (and time) needed to prepare the data prior to sending. They consist respectively in sending the entire B-tree, sending the leaves of the B-tree only, and sending the data only; the latter has two alternatives on the node being repaired, depending on whether the data is being inserted into a new B-tree, or whether the B-tree is being regenerated from the leaf-level and up.

This study shows that the choice of recovery method should be made considering the network configuration that will be used. For common network configurations like 100Mbits or lower, it is interesting to use methods that minimize the volume of data transfered. For higher network bandwidth, it is more important to minimize the amount of work done at the nodes.

Contents

List of Figures

List of Tables

Introduction

ClustRa is a telecom database management system designed to provide high availability, high throughput and real-time response. It uses a 2-safe replication scheme over two sites with independent failure modes, a novel declustering stategy, early detection of failures with fast takeover, on-line self-repair, and on-line maintenance [7], to achieve availability class 5.

The takeover and online self-repair capabilities, in particular, play a crucial role in the high availability of this DBMS. Takeover allows the system to mask a node failure, while online self-repair automatically reestablishes the fault tolerance level of the system after a node failure.

The node recory algorithm used by ClustRa varies depending on the level of corruption of the node. The recovery can either be done from main memory, from disk or from the neighbor node. During the recovery of a fragment replica, operations may have been executed to the fragment in the time interval during which the replica was unavailable. In addition to performing a recovery of the replica, failed node must also catch-up with these remaining operations [6]. Figure 1 illustrates a case where recovery from the neighbor is performed.

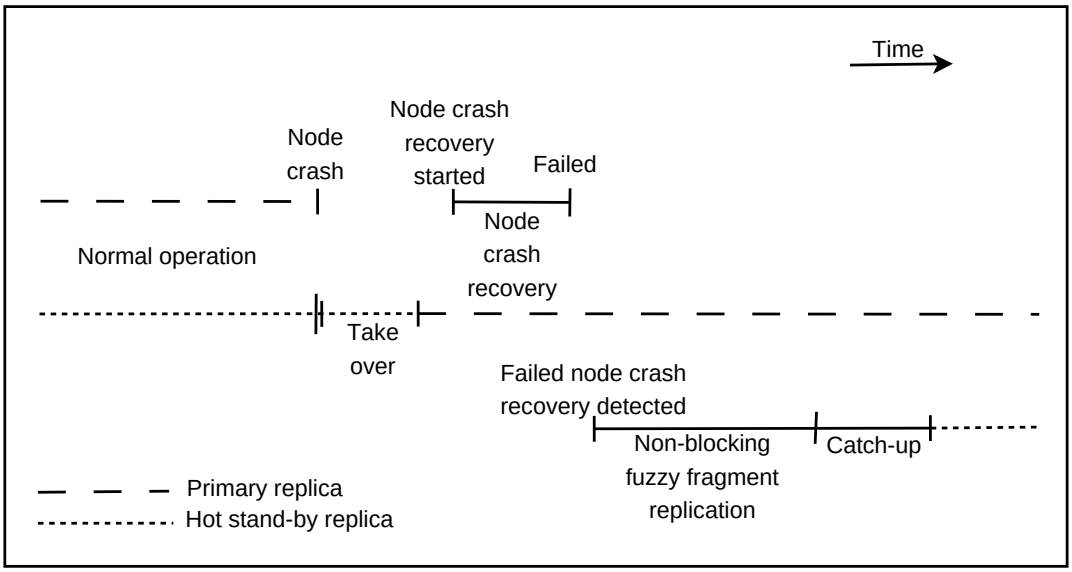

Figure 1: *In the node crash recovery process, failure of the original mode during recovery triggers the online self-repair on a spare node [6].*

The study presented in this report examines three different methods for reproducing lost replicas after a node failure, in ClustRa or other similar DBMS. The objective is to determine which of the methods studied has the best performance in terms of completion time at both nodes, for regenerating a new fragment replica and catching-up with the operations executed during the recovery process. The methods are evaluated under different conditions in terms of fill level of the B-tree, number of operations executed on the fragment per second, size of the database and network bandwidth. One of the methods is used in ClustRa, and a motivation for the project was to examine whether other recovery methods could be implemented preferably.

The report is organized as follows. In Chapter 1, we give an overview of the harware and DBMS architecture of ClustRa. All methods studied use B-trees as access method, because that is the access method used by ClustRa. We present that type of access method, and some of its variants that are relevant to the project, in Chapter 2, together with a short introduction to transaction logging. The alternative recovery methods investigated are discussed in Chapter 3, and our implementation of these methods is presented in Chapter 4. Chapter 5 provides a detailled description of the experiments and a discussion of their results. The main conclusions of and the prospects to the project are finally discussed in Chapter 6.

Chapter 1

ClustRa

In this chapter we give a short overview of a database system that uses recovery methods similar to those investigated in this project. The recovery methods that we address could be potential candidate implementations for such a system. The particular DBMS that we present here is ClustRa.

1.1 Overview

ClustRa [7] is a database system designed to provide high availability, high throughput and real-time response. To achieve high troughput and real-time response, ClustRa is a memory-based database with neighbour write-ahead logging. To achieve high availability ClustRa uses data replication and allocation of primary and hot stand-by replicas in nodes with different failure modes, early detection of failures with fast take-over, and on-line self-repair and maintenance.

The ClustRa database system is based on a shared nothing architectures. It consists of a collection of interconnected *nodes* that are functionally identical and act as peers, without any node being singled out for a particular task. The fact that each of these nodes has its own disk and large main memory and neither disk nor memory are shared between the nodes makes it possible for a node to fail or be replaced without affecting the other nodes.

Nodes are grouped into *sites*, which are collections of nodes with correlated failure probabilities. Sites are failure-independent with respect to environmental and operational maintenance and each site contains a replica of the database. Figure 1.1 illustrates the hardware architecture of ClustRa.

Each table may be distributed over multiple nodes by horizontal fragmentation according to a hash or range partitioning algorithm. ClustRa uses an asymetric replictation scheme, where there is one primary replica, but there might be several hot stand-by replicas. Each node in the system may be primary for some fragments

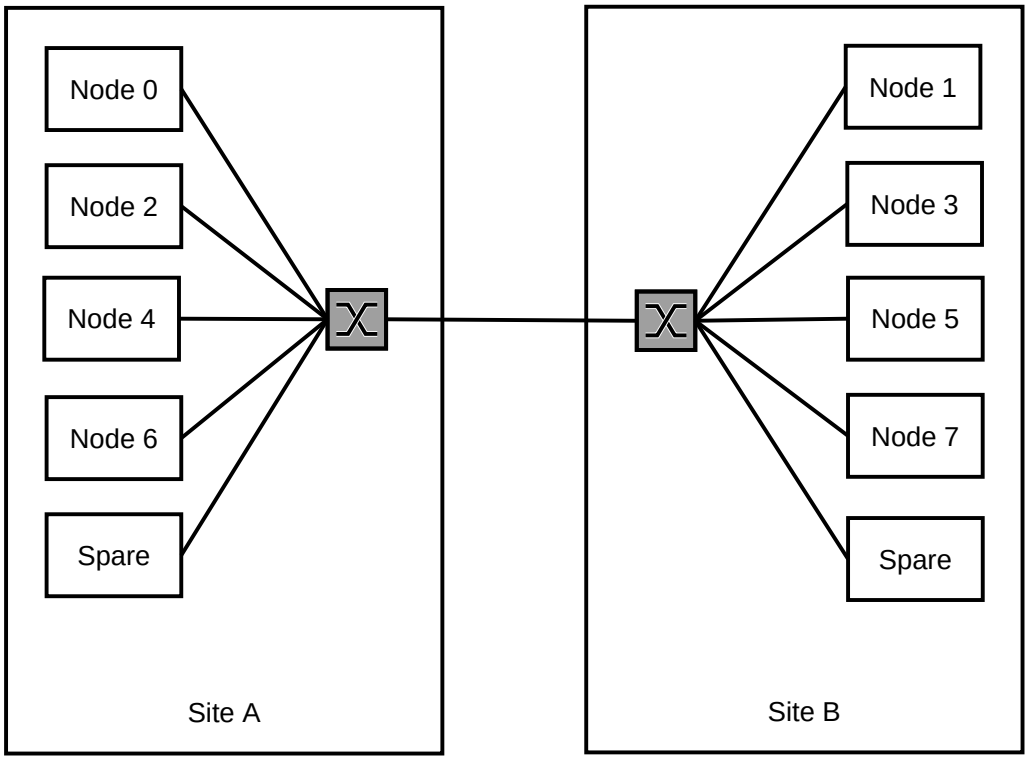

Figure 1.1: *Example of ClustRa's hardware architecture with two sites, A and B, containing five nodes each, one of the nodes on each site being a spare node. The network connection is done using a switch.*

at the same time as it is hot stand-by for others. This facilitates load balancing both during normal processing and during node failure, where takeover must take place. Figure 1.2 illustrates the data distribution.

All of the nodes in the system play the same role and run the same software. The only difference between them is the data they store. Each node runs the following services:

- *transaction controller*: service that is responsible for handling the connections to the clients and managing the transactions they run;
- *database kernel*: service that is the data manager, responsible for storing both the log and the database;
- *update channel*: service that is responsible for reading the local log and shipping log records from primary fragment replicas to hot stand-by replicas;
- *node supervisor*: service in charge of collecting information about the availability of different services and of providing the nodes with information about changes.

During the execution of a transaction, the transaction controller at the node that receives it becomes the primary controller. A hot stand-by transaction controller,

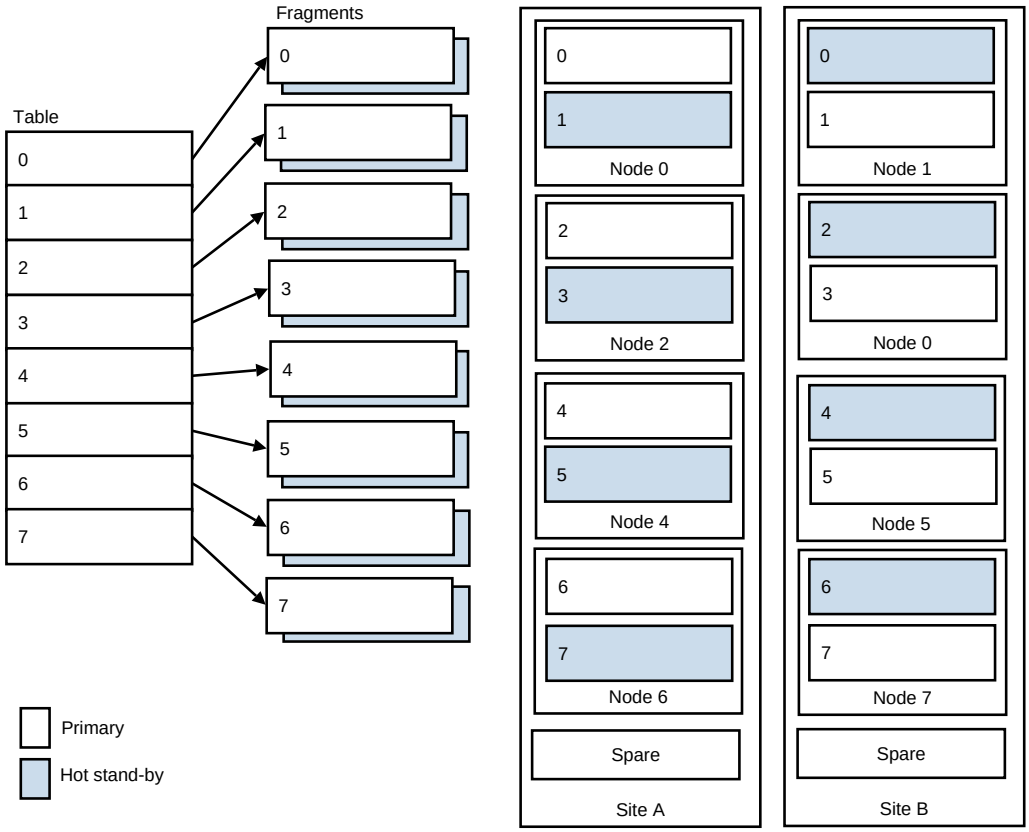

Figure 1.2: *Fragmentation of a table in 8 fragments and distribution in two sites with five nodes each, where one node on each site is a spare node.*

belonging to a site different from the site of the primary controller, is assigned. The hot stand-by transaction controller will be responsible for taking over if the primary controller fails. The transaction controller is responsible for *(i)* finding the nodes involved in a transaction; *(ii)* sending *'start transaction'* and the operation to execute to the nodes containing the primary replica of the records involved in the operation; *(iii)* sending *'start transaction'* and the number of log records that will be produced by the transaction to the nodes containing the hot stand-by replica(s) of the records involved in the operation; *(iv)* sending an early answer to the client and *(v)* coordinating the transaction by sending *'start transaction'*, *'commit'* and eventual *'roll-back'* to the nodes that take part in the transaction. The database kernel at the node containing the primary replica of a record will always be the one responsible for executing a request on that record. To keep its hot stand-by replicas consistent, the update channel at the same node will be responsible for reading the log records produced by the transaction and sending them to the hot stand-by replica(s). The database kernel at the node(s) containing the hot stand-by replicas will be responsible for redoing these log records. Figure 1.3 illustrates the execution of a transaction.

ClustRa combines two logging methods: a logical, and a physiological log. The logical log is used to keep the hot stand-by replicas consistent and to provide transaction durability, while the physiological log is used as a node-internal log.

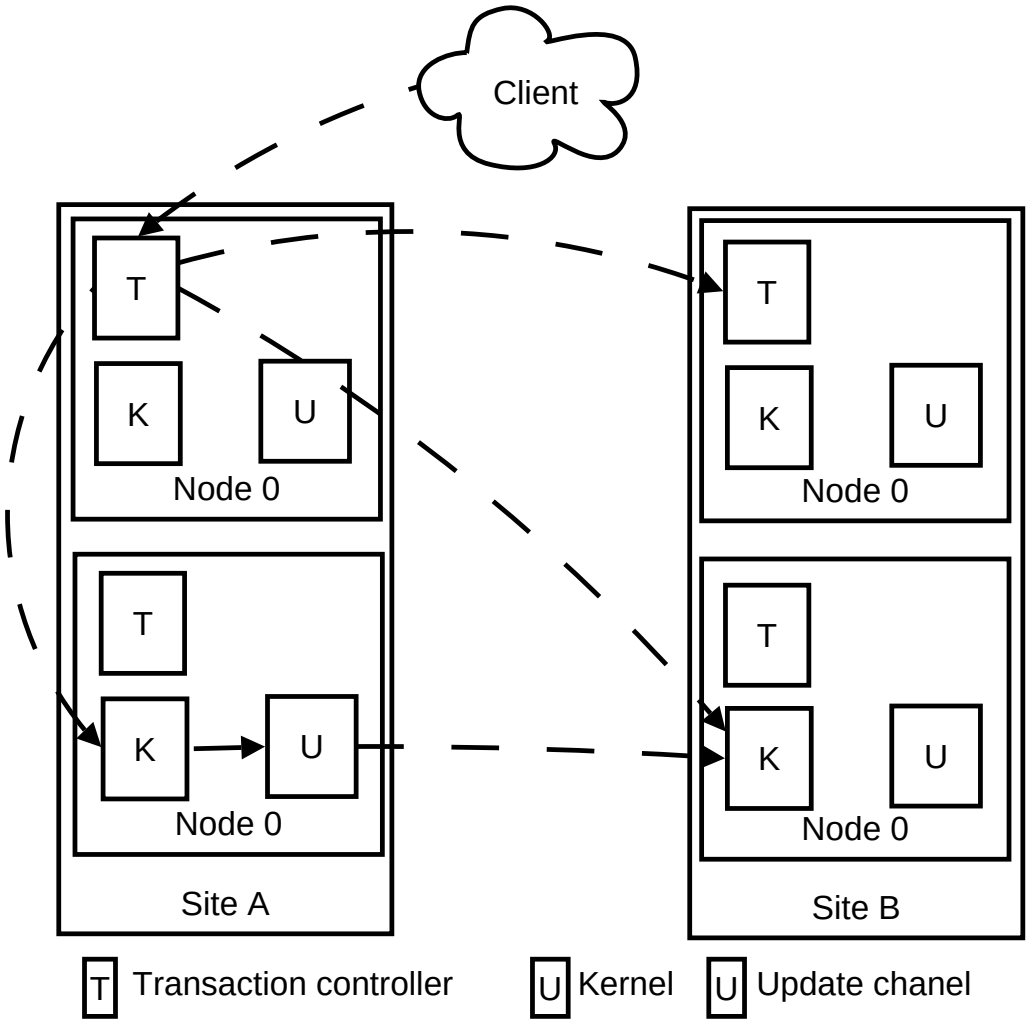

Figure 1.3: *Transaction execution at a ClustRa system with two sites. Node 0 is the node receiving the request from the clients and becomes the transaction controller and node1 is assigned as hot stand-by controller. Node2 and node3 contain the primary and hot stand-by fragment replicas of all records involved in the transaction.*

The node-internal physiological log is used for storing node-internal operations for access methods, free block management, and file directory. These operations are implemented transactionally. It is disk-based and is not shipped to any other node.

The distributed logical log is used for record operations. Each log record is identified by the primary key and a *log sequence number*, and contains both redo and undo information. Thus, it can be applied to any replica of a fragment. The distributed log applies a *neighbour write-ahead logging* [6], which is a main-memory logging technique where transaction commit does not force log to disc; instead, a log record must be written to two nodes in different sites before an update operation is allowed to be reflected on disk, and before the transaction commits. This log shipping is also used for data replication.

ClustRa supports parallel, online, non-blocking scaleup. The scaling is performed

by gradually redistributing the database so that all the available nodes are taken into operation. ClustRa performs scaleup by redistributing a table at a time. Within a table, a few fragments are refragmented in parallel to avoid overload of the sender and receiver nodes. Scaleup involves subfragmenting a table, thus the number of fragments increases as part of a scaleup. This strategy allows, in addition to keeping the additional load on the system low, to minimize the extra storage needed as part of the subfragmentation. Only fragments under redistribution need an extra replica.

When the subfragmentation of a fragment is completed, the subfragmented replica takes over as the hot stand-by replica for the fragment, and the old hot stand-by replica is deleted. The storage for the old replica is then freed. The new hot stand-by fragment replica then takes the role of primary replica for the fragment. Another subfragmented replica is then produced, based on the current replica. When this replication is finished, the new subfragmented replica takes the role of the hot stand-by and the old not-subfragmented replica is deleted. The subfragmentation of a fragment is run as a transactions with savepoints per replication.

Scaledown is performed through parallel replication, in a similar way as for scaleup. Parallel scaledown requires that multiple fragments are merged. The same techniques as for scaleup are used in respect to minimizing additional storage needed [3].

1.2 Failure detection and recovery

An I-am-alive protocol is applied to discover failed services or nodes. The I-am-alive protocol between the nodes is organized as a circle. The node supervisor at each node periodically polls all processes at the node to detect their state. Each node sends and receives I-am-alive messages from both its neighbours in the circle. If consecutive I-am-alive messages are missing from a node, the *virtual node set protocol* is activated. The node supervisor sends a build node set message to all known nodes, who respond with their services. If a node has not responded within a certain number of resends, it is assumed to be down, and a new node set is distributed.

A primary fragment replica crash causes a fragment to be unavailable. A node crash or a controlled node stop similarly causes the fragments with their primary replicas at the failed node to be unavailable. These fragments will continue to be unavailable until the hot stand-by replicas of the fragment change their roles to primary. Before a hot stand-by replica can become available, it must redo the hot stand-by log records that arrived before the new node set, and roll back in-flight transactions being active at a crashed node. This process of changing the role from hot stand-by to primary is called *take-over*.

After a take-over the failed node is tried restarted. If the node failure was caused by a transient error, the node will restart and a take-back can take place. If the node does not restart, the recovery of the failed node is started. The recovery method

used by ClustRa will vary depending on the degree of corruption of the node.

When only parts of the main memory are corrupted and the database buffer and main memory log are intact, a recovery from main memory is done. It involves just undoing the eventual ongoing node-internal transactions and undoing the record operations that were not reflected on any other node before the crash occurred.

When the contents of the main memory is garbled, a recovery from disk is done. It involves performing a *'redo'* followed by an *'undo'* recovery from the stable node-internal log, before a redo recovery is executed. The redo recovery is based on the distributed log shipped from the nodes with primary fragments for those stored at the recovering node.

When the disk(s) is/are corrupted or the node will not restart, a recovery from the neighbour is done. It involves reloading the fragments stored in the node being repaired from the primary fragment replicas stored in other nodes.

Operations can be executed during the period during which the recovering node is unavailable. The recovering node will then have to *catch-up* with these operations. This is done by redoing the distributed transaction log. After the recovering node has caught-up, it forces a takeback, which is the opposite of a takeover. The original primaries and hot stand-bys get their original roles and the system state is restored.

1.2.1 Recovery from main memory

ClustRa uses checksums on the buffer data structure, the log access structured, and the log structure to decide whether a recovery from main memory can be done. The recovery from main memory will be done if these checksums are found to be in order.

If during a main memory-based recovery a block is found to be corrupted upon access, a partial recovery is performed. The block is read from disk, the node internal log records regarding that block are redone, and the distributed log records regarding that block are also redone.

1.2.2 Recovery from disk

A recovery based on main memory is performed when the main memory is gargled. In such a recovery, memory structures are rebuilt based on the stable node internal log.

The recovery starts from the penultimate checkpoint log record of the node internal log. The transactions stored on this checkpoing log record are retrieved, and the log records starting by the checkpoint LSN are redone. The transactions that are still active when the end of the node interal log is achieved, and can be rolled forward, are rolled forward. Then the node internal log is scanned backwards and the transactions that are still active are undone. [9]

1.2.3 Recovery from a neighbour

A recovery based on the neighbour happens when either the contents of the disk are garbled, or the failed node fails to restart. In such a recovery, the complete database is reloaded from other nodes.

The recovery in this case is based on the data at the repairing node. During the process the repairing node will be responsible for sending the records it containd to the node being repaired. This is done without locking any records, so that they will be available for online transactions, and in a fuzzy manner because the fragment replica is not in a transaction-consistent state. The node being repaired will reconstruct the access method, by receiving the data and inserting it.

In the following chapters we will evaluate different node recovery strategies for the case where the recovery is done from the neighbour.

Chapter 2

B-trees and Transaction Log

In this chapter we give a short presentation of B-trees and transaction log. We have included this introduction to B-trees because it is the access method that we shall use for the implementation of the recovery methods. We have chosen to also include an introduction to transaction logging because we refer to it when we discuss the catch-up process later in paragraph 3.2.

2.1 B-trees

As presented in [4], B-trees are balanced search trees. As in binary search trees, in a B-tree the branch taken at a node depends on the outcome of a comparison. A binary search tree contains only one key per node, so the comparison between the key stored in the node and the query key gives the branch to follow. While in a B-tree a node can contain several keys, thus several comparisons might be needed to chose the correct branch to follow.

Each node of a B-tree of order d contains at most $2n$ keys and $2n + 1$ pointers, and at least d keys and $d + 1$ pointers. Keys and pointers are organized so that, if a pointer is on the left of a key, it points to data that is smaller than the key, and a pointer in the left of a key points to data that is bigger or equal to the key.

The beauty of a B-tree lies in that the methods for inserting and deleting records always leave the tree balanced. The insertion of a new key requires two steps: (i) search from the root to locate the proper leaf for insertion; (ii) insert the key in the node found. To keep the tree balanced, whenever a key needs to be inserted in a node that is already full, a split occurs, the node is divided in two, of the $2d+1$ keys, the smaller ds are placed in one node, and the larger ds are placed in another node, the remaining value is promoted to the parent node where it serves as a separator. If the parent node is also full, then the same splitting process will happen again. The deletion in a B-tree is done in the opposite way of an insertion. Figure 2.2 illustrates the insertion of a new record into a B-tree causing a page split.

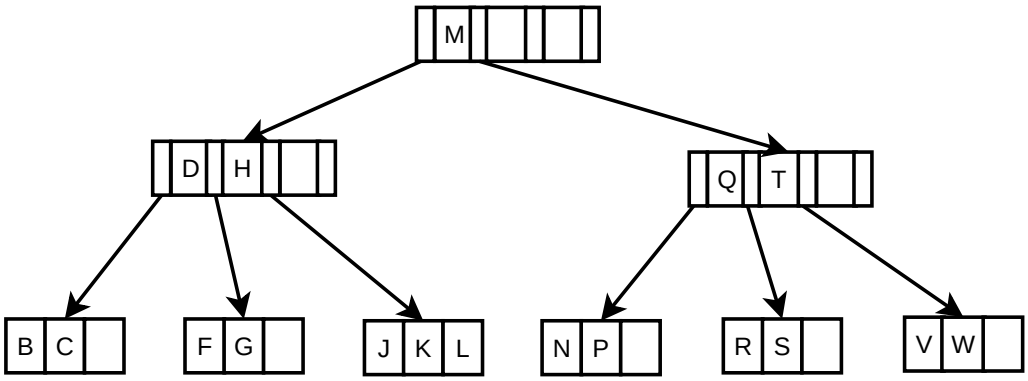

Figure 2.1: *Sketch of a B-tree structure. Each page has a maximum of 3 indexes and 4 pointers. The pointer in the left (respectively, right) side of an index points to pages containing keys that are smaller (respectively, larger) than the index. The insertion and deletion algorithms ensure that the tree remains balanced.*

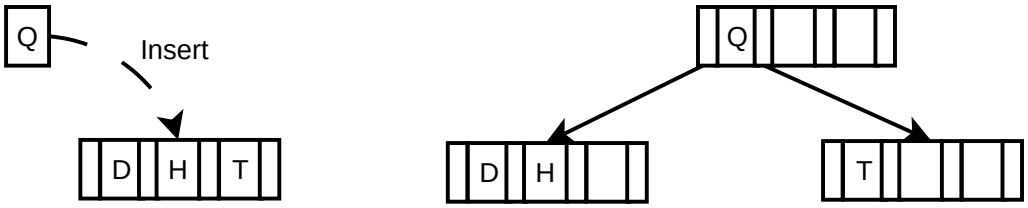

Figure 2.2: *Sketch of the insertion of a new record into a B-tree causing a page split.*

The implementation of the fragment replication methods, presented in chapter 4 and used for the experiments presented in chapter 5, was done using Berkeley DB's implementation of B-trees, and the database(s) used for measuring the performace of these methods were generated using Berkeley DB. Berkeley DB implements a variation of B-trees called B^+-tree. In a B^+-tree all the data resides in the leaves. The upper levels, consist only of an index to enable rapid location of the record. Leaf nodes are usually linked together left-to-right to allow easy sequential processing. Figure 2.3 illustrates the structure of a B^+-tree.

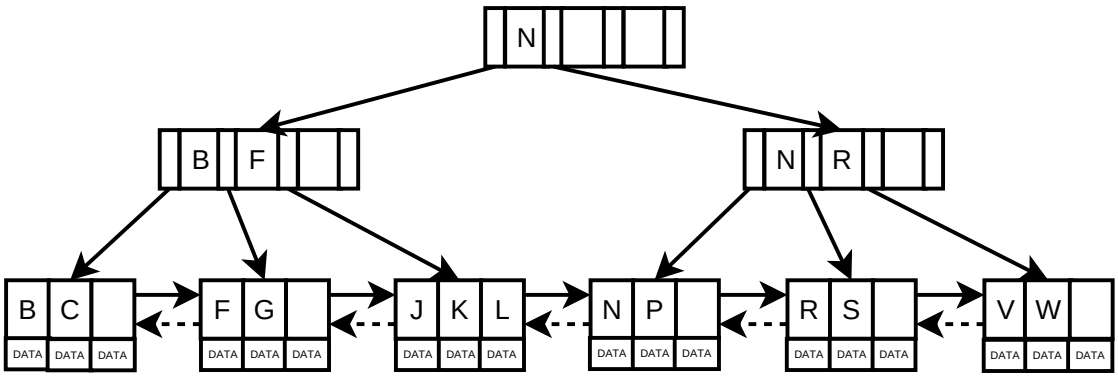

Figure 2.3: *Sketch of the structure of a B^+-tree. The leaf nodes are linked left-to-right. The dotted lines indicates that in some implementations, as the one in Berkeley DB, these nodes are also linked right-to-left.*

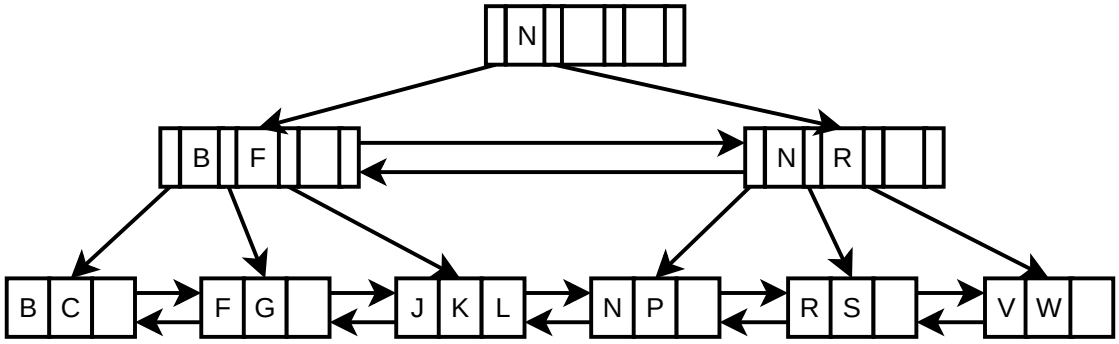

Figure 2.4: *Sketch of the structure used by ClustRa. Nodes in all levels of the tree are linked both left-to-right and right-to-left.*

Clustra implements another variant of B-trees. As in B^+-tree all the data resides on the leaves. But on ClustRa's implementation nodes at a given level whether at leaf level or not, are linked together both left-to-right and right-to-left. ClustRa's implementation of B-trees is illustrated in figure 2.4.

2.2 Transaction log

The transaction manager [1] ensures the A, C and D of the ACID properties. It provides *(i)* atomicity by undoing aborted transactions, redoing commited ones, and coordinating commitment with other transaction managers for distributed transactions; *(ii)* consistency by aborting any transactions that fail to pass the resource manager consistency test at commit; and *(iii)* durability by, as part of the commit processing, forcing all log records of commited transactions to: either durable memory when using WAL protocol, or two copies of the log records in main memory at nodes with independent failure modes when using nWAL protocol.

During normal processing, the transaction manager simply gathers information that will be needed in case of failure. As a transaction progresses, its update operantions generate a sequence of log records, which is called the transaction's log. It is stored in a common log table. Each log record has a unique key, called Log Sequence Number (LSN).

The transaction manager can undo the whole transaction by undoing each of its individual actions. This is done by reading the transaction log and invoking each of the log record's UNDO operation.

During a system restart after failure, the transaction manager must ensure that committed transactions are preserved. The transactions manager scans the log forward and applies the log record's REDO operation. After the REDO has been applied, there may be some transactions that generated log records but did not commit. These uncommitted transactions must be rolled back. This is done by applying the UNDO operation of log records for these transactions in backward di-

rection, starting by the most recent operation. The result is a consistent state where all committed transactions have been redone and all uncommitted transactions have been undone.

Undo work should always be done by reading the log backward; the most recent log record for should be the first one to be redone. In contrast *redo* should be applied in forward direction, starting by the first log record for a transaction.

Each log record contains a standard header and a type-dependent body that describes the operation that generated it. The header carries the log sequence number, transaction identifier and some other fields. The log body contains whatever information is needed by the UNDO and the REDO operations.

2.2.1 Physical logging or value logging

The simplest technique of writing log records and the corresponding UNDO-REDO programs for durable data places the old and new object states in the log record. UNDO and REDO are then trivial: the object needs only to be reset from the appropriate value to the appropriate value contained in the log records.

Value logging is a good design if the object state is small. But if the object state is large and the change is small, then the log record is often compressed so that it contains only the changed portion(s) of the object, see [1].

2.2.2 Logical logging or operation logging

Logical logging records the name of an UNDO-REDO funcion and its parameters, rather than the object values themselves. For example, a single log record like

```
<insert op, table name, record value>
```

can be used to record the insertion of a certain record into a certain table. This insertion may cause disk space to be allocated, other records to move within that page, or even a complex index update like a B-tree split. Thus, a single logical log can correspond to several physical log records.

Clearly, logical logging is the best approach; the log records are small, and the UNDO-REDO operations are mirrors of one another. However, logical logging assumes that each action is atomic and that in each failure situation the system state will be action-consistent: each logical action will have been completely done or completely undone. In some failure cases, actions will be partially completed, and the state will not be action consistent. The UNDO operation will then be requested to UNDO stated produced by such partially completed actions, see [1].

2.2.3 Physiological logging

Physiological logging is a compromise between logical and physical logging. The term physiological logging derives from the use of logical logging to represent transformation of physical objects. The physical part of the design refers to the fact that each log record applies to a particular physical page or a particular communication session. The state transformation of a physical object can be represented physically or logically.

Complex actions generate a sequence of physiological log records. In addition to the log record describing the table insert, there would be additional log records describing the insert to the table indeces, the updates performed by triggers, and so on.

For example, let us consider a logical log record like in the form:

```
<insert op, table name = A, record value = r>
```

If the table for example had two indeces, the corresponding physioligical log record would be:

```
<insert op, base filename = A, page number = 508, record value = r>
<insert op, index1 filename = B, page number = 72, index1 record value = keyB of r = s>
<insert op, index2 filename = C, page number = 94, index2 record value = keyC of r = t>
```

Physiological logging has many of the benefits of logical logging. The UNDO and REDO operations are often similar to other operations. When compared with physical logging, it has small log records, see [1].

2.2.4 Compensation log records

Log records generated during UNDO are often called *compensation log records*. A compensation log record will be produced for each log record produced by a transactions. The REDO information of the compensation log record will be the UNDO information of the log record it compensates and respectively the UNDO information of the compensation log record is the REDO information of the log record it is a compensation for.

Pages frequently carry a monotonically increasing sequence number used to record the page version, enforce the write-ahead protocol, and provide idempotence. This sequence number is usually just the LSN of the most recent update that transformed the page and is called the *page LSN*. Without the use of compensation log records, the undo of an operation would not be shown in the page LSN, because the undo of an operation would produce a new log record and therefore not produce a new LSN.

Compensation logging simplifies the design of physiological logging. With that design, each update on a page or session is accompanied by a log record. The action is undone, the UNDO looks just like a new action that generates a new log record - a compensation log record, see [1].

Chapter 3

Node Crash Recovery Methods

In this chapter we present three different methods for recovering a failed node. As mentioned in paraghaph 1.2, the recovery method used by ClustRa varies according to the degree of corruption of a node. The methods discussed in this chapter are intended for the case where the node crash causes complete loss of data and the recovery is done from the neighbour nodes.

First, we present each of the three methods for fragment replication that we are evaluating. Then, we present the catch-up process. During the fragment recovery process, the repairing node continues to execute transactions. The catch-up process is used to keep the node being repaired up to date with these transactions.

Note that both the methods for fragment replication and catch-up that we are using are designed for database system using B-trees (or a variant of B-trees) as access methods. The catch-up process assumes that the logging sub-system of the database system uses compensation log records.

3.1 Methods for Fragment Replication

In this section we present each of the methods for fragment replication, and discuss different ways of implementing them. For each of the methods we give special attention to: *(i)* how the method behaves when the nodes operate with different page sizes, *(ii)* how soon the catch-up process can be started, and *(iii)* how to implement a startup protocol that allows the method to restart recovery if one of the nodes crashes during the recovery process.

3.1.1 Sending the Entire B-tree

This method consists in sending the entire B-tree from one node to another. In order to do so, each page must be read by the repairing node, and sent to the node

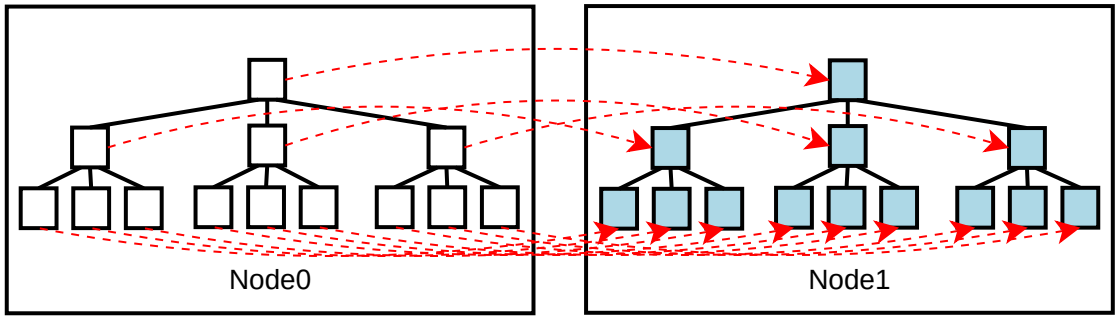

Figure 3.1: *Sending an entire B-tree from the repairing node to the node being repaired.*

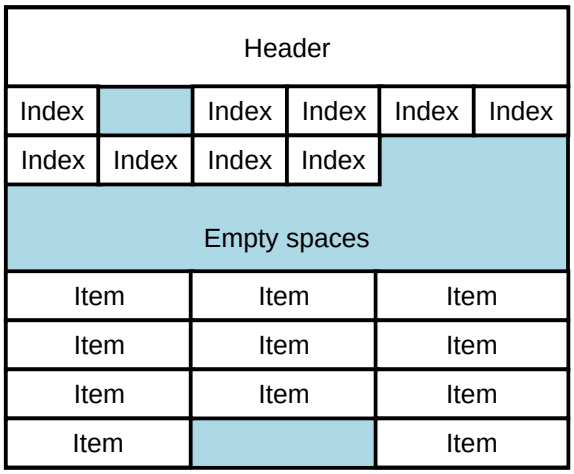

Figure 3.2: *Illustration of a typicall B-tree page. It contains a header with information about the page, pointers to the location of the items, which can be pointers to child pages, pointers to the location of the data, or the data itself.*

being repaired, where it will be stored. Figure 3.1 illustrates this method.

Data compression

A B-tree page, as illustrated in figure 3.2, may contain some empty space. To reduce the volume of data sent over the network, the empty space inside a B-tree page can be removed before the page is sent and inserted again when the page is received.

The fill level of a B-tree is garanteed to be at least 50%, or 67% for some variants [2]. However, in some implementation, B-tree page splits happen normally, but the pages are not merged when tuples are deleted. This can cause an even lower fill level than 50% (or 67%). In such cases, removing the empty spaces inside the B-tree page can reduce the amount of data transfered considerably.

Applying the method between nodes with different page sizes

This method can be used to replicate a fragment replica between nodes that operate with different page sizes. This may involve extra work. In the most simple case, the node being repaired operates with a larger page size than the repairing node. In such a case, the only necessary actions are: *(i)* to expand the pages as they are received so that they will have the same size as the node operates with, and *(ii)* to update the pointers to locations inside the page.

When the page size of the node being repaired is smaller than that of the repairing node, there are two alternatives: either the received page after the empty space has been removed is still larger than the page size the node is operating with, or it is smaller.

If the received page after removal of the empty spaces is still larger than the page size the node is operating with, the page will have to be split into two new pages. A B-tree page split will cause a new item to be added to the level above the one where the split happened. This can, if the page in the level above is full, cause another page split to happen. Thus, a page split can propagate all the way up to the B-tree root. In this method where pages of all levels of the B-tree are being received, such a page split would involve some extra work, depending on the order the pages are being received. For instance, if the page in which the new item should be inserted has not yet been received at the time the page split happens, the system will have to keep track of the item and insert it when the page is received.

If the received page after removal of the empty spaces is still smaller than the page size the node operates with, the following must be done: *(i)* the page must be expanded to the page size the node operates with, and *(ii)* all pointers to locations inside the page must be updated.

When can the catch-up process be started

During the period during which a new fragment replica is being copied over to the node being repaired, transactions will still be running at the repairing node. The outcome of these transactions have to be reflected in the new fragment replica being generated. This can be done by redoing the log records produced by these transactions at the node under repair. This process is called catch-up and is explained in more detail in paragraph 3.2.

We are interested in starting the catch-up process as soon as possible. The repairing node is able to start sending log records as soon as they are generated, but the node being repaired needs to have an access method available to be able to redo these log records.

For this method, where all pages of a B-tree are sent from one node to the other, whether an access method is available or not depends on the order in which the pages of the B-tree is sent.

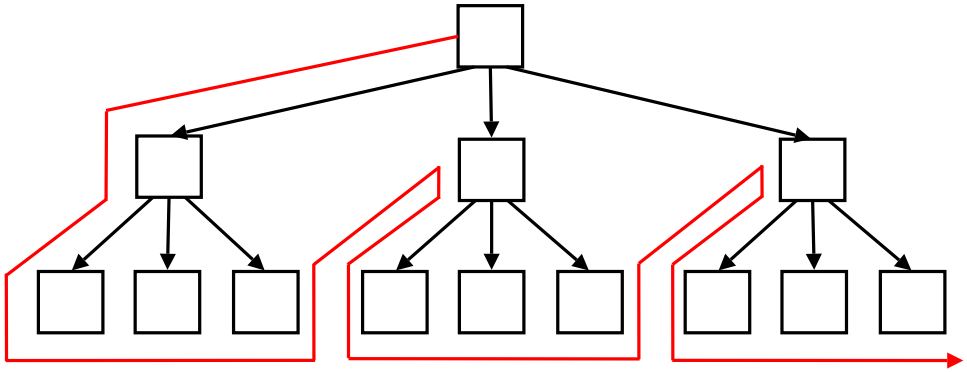

Figure 3.3: *Pre-order tree walk.*

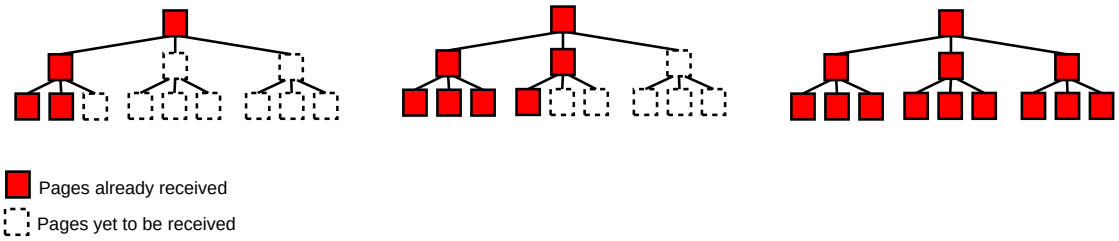

Figure 3.4: *Tree received in a pre-order walk order. This figure shows that the access method is, during the entire process, able to reach all pages that have been received.*

If the pages are sent in an order so that it is not possible to have access to all received records until all pages have been received, the log records have to be stored until all pages have been received. Only then, will it be possible to apply these log records. If the pages are sent in an order such that during the whole process there will be an access method available that allows access to all the received records, the log records can be applied as soon as they are received.

To ensure that an access method will be available and allow access to the received leaf pages, the B-tree pages must be sent in a special order. It is very important that all internal pages that are located in the path of a leaf page are received before the leaf page is received. Sending pages in the order shown in figure 3.3 would make this possible. This way of accessing a tree is often called pre-order tree walk [5]. Figure 3.4 illustrates how this order for sending the B-tree ensures access to the received leaf pages through the B-tree, at all times. This strategy makes it possible for the catch-up process to start applying log records as soon as they are received.

Node crash during recovery

A startup protocol is necessary to allow the replication process to restart in case of node failure. This protocol must include information about, the last page that was successfully written to disk by the node being repaired before the failure happened. This is necessary for the repairing node to know from which page to restart the process.

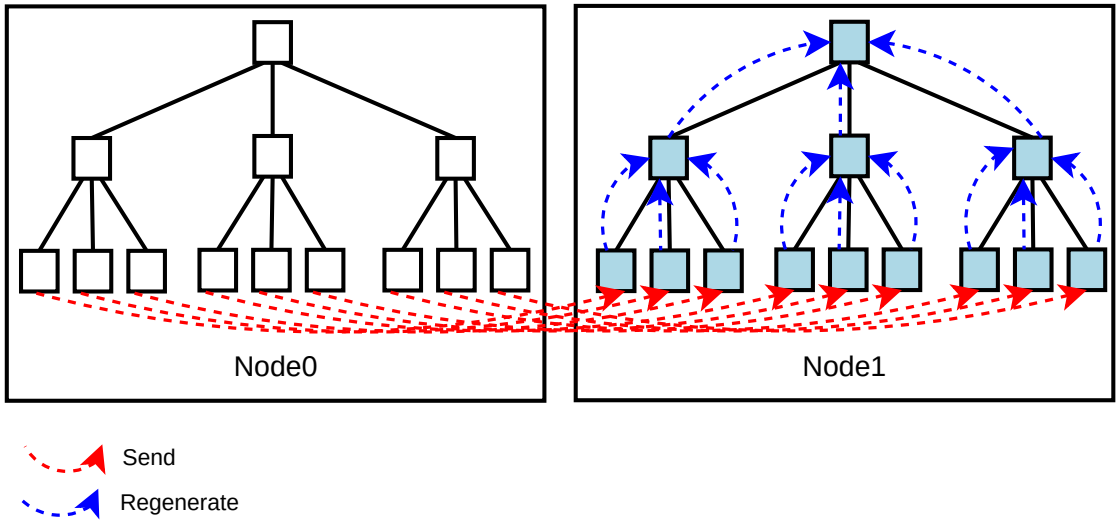

Figure 3.5: *Sending B-tree leaf pages from the repairing node to the recovering node and regenerating the internal pages of the tree.*

3.1.2 Sending leaf pages of the B-tree

This method is similar to that presented on paragraph 3.1.1, but here, instead of sending all pages of the B-tree, only the leaf pages are sent from one node to another. The leaf pages are the pages of the B-tree that contain the data, while the internal pages contain only indexes (or pointers to other pages) and can easily be reregenerated by the node being repaired. Figure 3.5 illustrates this method.

We have included this method as an alternative to the ones discussed in paragraph 3.1.1, because less pages are sent from the repairing node to the node being repaired. This reduces the volume of data sent over the network, but it requires extra work to be done at the node being repaired in order to regenerate the internal pages of the B-tree.

Data compression

As in paragraph 3.1.1, it is also possible here to remove the empty spaces inside the B-tree page before transferring it to the node being repaired. This is done to reduce the volume of data that is sent over the network.

Applying the method between nodes with different page sizes

Applying this method between nodes that operate with different page sizes may involve some extra work. This is similar to what was presented in paragraph 3.1.1. If the page size of the node being repaired is larger than that of the repairing node, then it will be necessary to do the same as explained in paragraph 3.1.1.

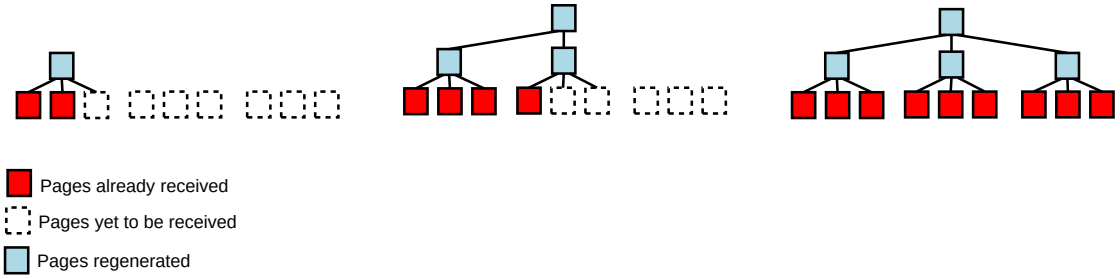

Figure 3.6: *Regenerating the B-tree as soon as the leaf pages are received.*

If the page size of the node being repaired is smaller than that of the repairing nodes, there are also the same two alternatives as in paragraph 3.1.1. The advantage in this case is that the internal pages of the B-tree are regenerated as the leaf pages are being received, splitting B-tree pages that are two big for the page size of the node does not cause as much extra work in this method as it does for the method presented in paragraph 3.1.1.

When can the catch-up process be started

As in paragraph 3.1.1, we are interested in starting the catch-up process as soon as possible. How early the log records can start being redone at the node being repaired depends on when the internal page of the B-tree are regenerated. If the internal pages are regenerated only after all of the leaf pages have been received, the log records received have to be stored until all leaf pages have been received and the internal pages have been regenerated. After all pages have been received, these log records can start being applied. If the internal pages start being regenerated as the leaf pages are being received, there will be an access method available during the entire process and the log records can start being applied. This is shown in figure 3.6.

Node crash during recovery

A startup protocol that initiates recovery after a node crash can be implemented in the same way as previously discussed in paragraph 3.1.1.

3.1.3 Extracting the data from the B-tree and sending only the data

This method differs from the ones presented in paragraphs 3.1.1 and 3.1.2, in that it only sends the data. In order to extract the data from the B-tree, each leaf page of the B-tree has to be read, and the tuples extracted from it. These tuples are then packed into a new data block. When the block reaches a specific size (the

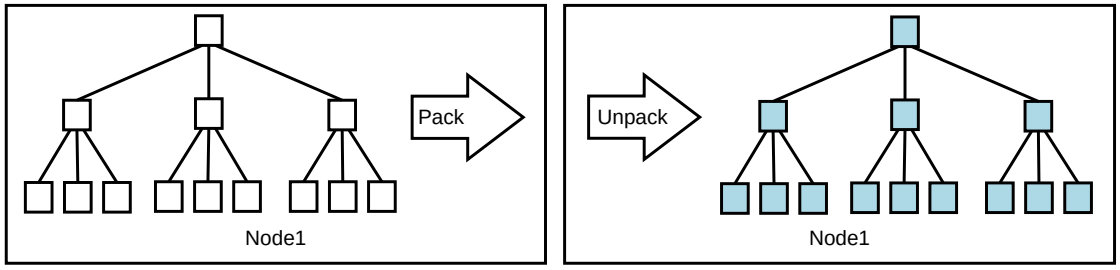

Figure 3.7: *Sending data from the repairing node to the node being repaired by extracting the tuples from each leaf page of the B-tree, packing the tuples into a new block, receiving the block, unpacking it and finally inserting into a B-tree when they are received.*

same size as a database page, for example), it is sent to the node being repaired, where the tuples are extracted and inserted into a B-tree. Figure 3.7 illustrates this method. It requires more work at both the sender and the receiver nodes than the ones presented earlier. Its advantage, though, is that the volume of data being sent over the network is smaller than with the other methods.

Applying the method between nodes with different page sizes

The LSN of the last log record applied to a page is often registered in the page. This is done to enable to database system to know whether a log record has already been applied to a page or not. Until now, we have been discussing methods where we send physical database pages from one node to another, and the LSN would be sent inside the page, so in the methods presented before, having the LSN stored by page has not been a problem.

To be able to use a method where database pages are not being sent, it is necessary to include a strategy for sending the LSN for the last transaction applied to each object. This can be done by either storing the LSN for each tuple instead of storing it for each page, or by having the data block that is sent contain page delimiters with their corresponding LSN, so that records will be in the same page and with the same LSN in both nodes.

If the LSN is stored for each tuple, the tuples are independent of the page, and there is no problem in applying the method for nodes that operate with different page sizes. On the other hand, if the LSN is stored for each page, the same issues concerning pages that are bigger than the page size of the node being repaired will be met, as for the method discussed in paragraph 3.1.2.

When can the catch-up process be started

As we have pointed out in paragraphs 3.1.1 and 3.1.2, it is important to have an access method available in order to be able to start the catch-up process before the

entire B-tree is received. As in paragraph 3.1.2, the same issues on whether an access method is available or not are present, depending on the implementation chosen for this method. If the internal pages of the B-tree are only regenerated after all the leaf pages have been regenerated, the catch-up process will have to wait to apply the log records until the internal pages have been regenerated. If the internal page are being regenerated as the data arrives, the log records can be applied as they are received.

The access method can be restored before all pages have been received using two alternative methods:

- The first alternative is to copy the tuple into leaf pages of a B-tree and regenerate it's internal pages from these leaf pages. Regenerating the leaf pages of the B-tree is possible because the records are being sent in increasing primary key order by the repairing node. As the leaf-pages are being generated, the internal structure of the B-tree can be rebuilt as explained in paragraph 3.1.2. This method will work independently of whether the LSN is stored within each page or within each tuple. As mentioned earlier, some extra work may be necessary of the LSN is stored on each page, though.

- The second alternative, which is the method used by ClustRa, consists in inserting the records one by one into a new B-tree, using the same insert procedure as used to insert a record under normal processing. This method only works if the LSN is stored for each tuple, because the rebuilding algorithm has no control over which page the record is being placed into.

Both alternative methods for rebuilding the B-tree allow the catch-up process to start as soon as the tuples start being received, because there will always be an access method available. The latter method, though, is more time consuming because (i) even though the tuples arrive in order, the position in which to insert a tuple in the B-tree has to be searched for each insert, and (ii) page splits shall happen.

Node crash during recovery

A startup protocol is necessary to allow the replication process to restart in case of node failure. This protocol must include information about the last key that was successfully written to disk by the node being repaired before the failure happend. This is necessary for the repairing node to know from which page to restart the process.

3.1.4 Summary on the respective qualities of the different methods.

The method presented in paragraph 3.1.1 is the most simple method presented. It requires the least work to be performed at both the repairing node and the node being repaired. The repairing node only needs to read each of the B-tree pages, and send them over to the node being repaired. The node being repaired only has to store the pages it receives. The entire B-tree structure is sent, in opposition to the other methods, and therefore no B-tree pages need to be regenerated. However, it is the method that sends the largest volume of data over the network, because compared to the method presented in paragraph 3.1.2 a larger number of pages are being sent, and compared to the method presented in paragraph 3.1.3 not only the data is being sent, but also structures internal to the B-tree, such as page headers, pointers, and internal pages.

The method presented in paragraph 3.1.2 is included as an alternative to the method presented in paragraph 3.1.1 because it reduces the volume of the data transfer. It increases the amount of work at the node being repaired because the internal pages of the B-tree have to be regenerated, but it has some advantages when the two nodes operate with different page sizes. We are interested in finding out whether the reduction in the amount of data sent over the network balances the time spent rebuilding the internal pages of the B-tree.

The method presented in paragragh 3.1.3 is the method sending the least amount of data, at the same time as it is the method that demands largest amount of work to be performed at the node being repaired. Its supremacy depends on whether the work involved in preparing a block to be sent and inserting the tuples into the database is less time consuming than sending a larger amount of data over the network, with less preparation, as in the other two methods.

3.2 Catch-up

During the process of recovering a node, the database system is still running and transactions are executed at the repairing node. The transaction log generated by these transactions can be used to make sure that their outcome are reflected at the node that is being repaired. To ensure this, the log records have to be sent to the node that is being repaired, where they must be redone.

Since the records are sent in a fuzzy manner, log records concerning tuples that have not yet been sent over to the node being repaired do not need to be sent, because the changes contained in these log records will be reflected at the records by the time they are sent. The transactions can affect any of the records with the same probability. Consequently, at the beginning of the transmission, when a small amount of data has been shipped to the node being repaired, the number of log records that need to be shipped is also small. As the amount of data that has

been shipped over increases, the number of log records that need to be shipped also increases.

A log record contains both redo and undo information, but for the purpose of keeping the node up to date with the changes that are happening on the neighbour node, it is sufficient to send the redo information. In the case of a transaction being rolled back, the redo data of the compensation log record will be shipped which is equivalent to having the undo data of the regular log record. By sending only the redo information on the log record, the amount of data sent over the network will be reduced. In the case of very large records this reduction can be close to 50% (redo and undo data have the same size and will be much bigger than the rest of the information in the log record).

Chapter 4

Implementation of the Node Crash Recovery Methods

In this chapter we present our implementation of the three different recovery methods and the catch-up method [1]. This is the implementation used for the experiments presented in chapter 5. Note that the different recovery methods for these experiments have been implemented assuming that *(i)* the node being repaired and the repairing node operate with the same page size, and *(ii)* that the nodes do not crash during recovery, so a startup protocol has not been implemented for any of the methods.

4.1 Program structure

Each of the recovery methods has been implemented using four threads at each of the nodes participating in the recovery process. Two of the threads are responsible for transferring data and log records between the nodes, while the other two are responsible for implementing one of the fragment replication methods presented in paragraph 3.1.

The implementation of the repairing node is illustrated in figure 4.1. Each of the boxes in the figure illustrates one of the threads running at the node. The threads *Prepare data block* and *Send data block* implement the different fragment replication methods, while *Prepare log block* and *Send log block* implement the catch-up method. *Send data block* and *Send log block* have the same function, they read a block and send it over the network. They communicate with *Prepare data block* and *Prepare log block* through a queue. *Prepare data block* and *Prepare log block* generate blocks to be sent and add them to a queue, and *Send data block* and *Send log block* read blocks from the queue and send them to the node being repaired.

[1] The source code for the implementation of the recovery methods including catch-up can be found in A

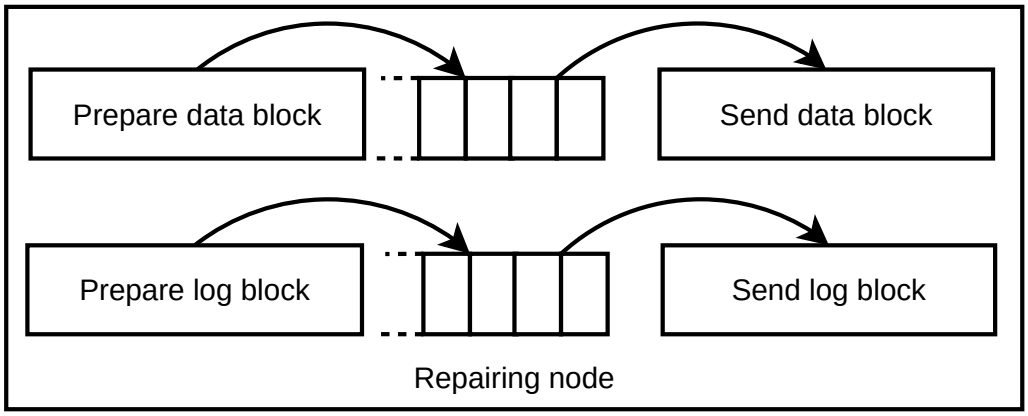

Figure 4.1: *Sketch of the implementation of the recovery methods at the repairing node (sender node), using four threads.*

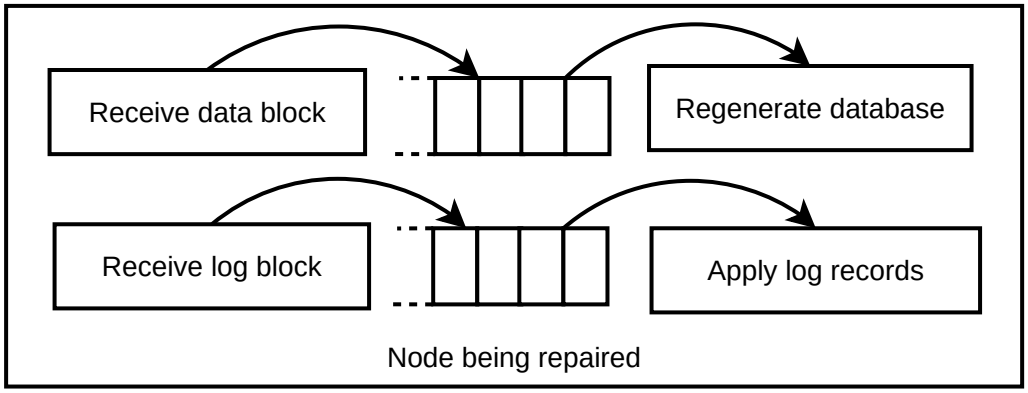

Figure 4.2: *Sketch of the implementation of the recovery methods at the node being repaired (receiver node), using four threads.*

Respectively, at the node being repaired, the thread *Receive data block* and *Regenerate database* implement the different fragment replication methods, while *Receive log block* and *Apply log records* implement the catch-up method. This is illustrated in figure 4.2. Similarly to what occurs at the repairing node, *Receive data block* and *Receive log block* receive blocks from the network and add them to a queue, while *Regenerate database* and *Apply log records* read the blocks from the queue and rebuild the access method, and apply the log records to the data that has been received.

In the following paragraphs we present the implementation of the threads *Prepare data block* and *Regenerate database* for each of the methods discussed in paragraph 3.1, and the implementation of *Prepare log record* and *Apply log record* for the catch-up method presented in paragraph 3.2.

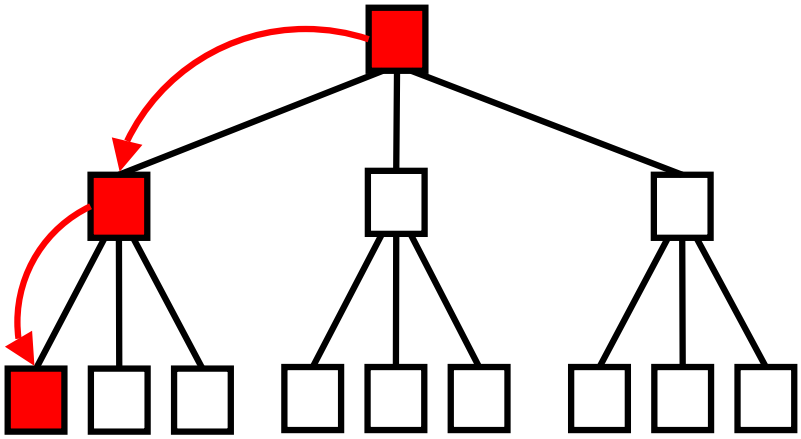

Figure 4.3: *The first page of the B-tree is found by navigating the left side of the tree.*

4.2 The *"Send all pages"* method

At the repairing node the B-tree is read in a pre-order tree walk order as previously discussed (see also figure 3.3). The empty spaces are removed from the page, which is then added to the queue in order to be sent to the node being repaired.

Once the pages have been received by the node being repaired, they are added to a queue. Each page that is read from the queue goes through the following procedure: *(i)* it is expanded back to its original size, by having its empty spaces added again; *(ii)* it receives a new page number (number that identifies the pages); and *(iii)* it has the link to it from its parent page updated.

The reason why the page has to be given a new page number and why the link from its parent page must be updated is related to the implementation of B-tree that we are using, namely Berkeley DB. In Berkeley DB's implementation of B-tree, the page number is used to calculate the offset from the start of the file to the position where the page can be found in the file, and the link between pages is done using the page number. Thus, if the pages are stored at the node being repaired in an order different from the one they had at the repairing node, a new page number corresponding to the position where they are stored must be assigned to them, and the links between the pages must be updated.

4.3 The "Send leaves" method

The repairing node starts by navigating the left side of the B-tree so as to find the first leaf page (see figure 4.3). Once the first leaf page is found, it *(i)* reads each of the leaf pages as shown in figure 4.4, *(ii)* removes the empty spaces inside the page, and *(iii)* adds the page to the queue of pages to be sent to the node being repaired.

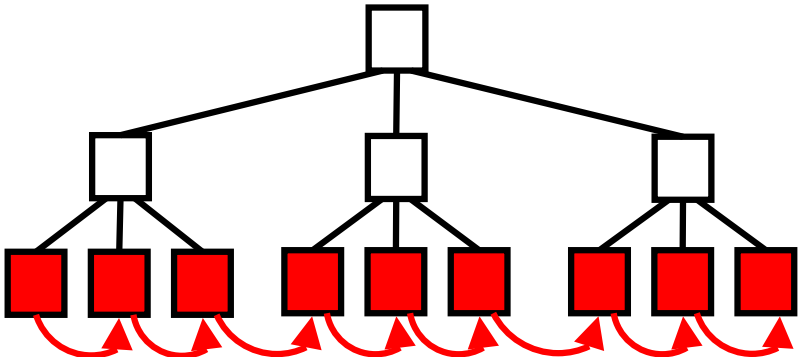

Figure 4.4: *Reading the leaf pages of the B-tree.*

Once the pages have been received by the node being repaired, they are added to a queue. Each page is then read from the queue and goes through the following preocedure: *(i)* it is expanded back the its orginal size, by having its empty spaces added again; *(ii)* it receives a new page number; and *(iii)* a link to the new page is inserted in a page in the upper level. If the page in the upper level has not been created yet or is full, a new page is created in the next-upper level of the tree; this page creation process can propagate all the way up to the root of the tree.

The pages must receive a new page number for the same reason as in paragraph 4.2.

4.4 The *"Send data"* method

As in paragraph 4.3, the node being repaired starts by navigating the left side of the B-tree to find the first leaf page. Once the first leaf page has been found, it *(i)* reads each leaf page as shown in figure 4.4, *(ii)* extracts each tuple from the page, *(iii)* adds the tuple to a new block containing only tuples, and *(iv)* once the block is full, it adds the block to the queue of blocks to be sent to the node being repaired. The size of the blocks that are sent between the nodes is the same as the page size the nodes operate with.

For this method we have implemented two different alternatives for how the blocks are treated by the node being repaired.

4.4.1 Alternative 1

Once the blocks have been received by the node being repaired, they are added to a queue. Each block is then read from the queue and undergoes the following treatmeant: *(i)* each tuple is extracted from the block; *(ii)* each tuple is added to a leaf page of the B-tree; and *(iii)* when a leaf page is full, a link to the new page is inserted in a page in the upper level, and a new leaf page is created for the next

tuples. If the page in the upper level has not been created yet or is full, a new page will be created in the upper level of the tree, and a link to it must be inserted in the page in the next-upper level; this step can propagate all the way up to the root of the tree.

4.4.2 Alternative 2

Once the blocks are received by the node being repaired, they are added to a queue. Each block is then read from the queue and undergoes the following scheme: *(i)* each tuple is extracted from the block; and *(ii)* each tuple is inserted to a new database using the insert method implemented by Berkeley DB.

4.5 Catch-up

Since there are no transactions taking place during the experiments, the catch-up process uses an artificial log file generated for testing. This log file contains a large number of log records with monotonically increasing LSN starting from LSN = 1. Since only the redo part of the log records is sent over to the node being repaired, the log file does not contain the undo part of the log records.

The repairing node has a timer that keeps track of for how long the repair process has been running. The last LSN that has been generated is calculated by using this timer and a constant indicating the number of log records generated by transactions per second.

In order to be able to know whether a tuple has already been sent to the node being repaired or not, each of the three fragment replication methods keeps a variable up to date with the primary key of the last tuple that has been sent. Since tuples are being sent in order, it is possible to check if a tuple has been sent or not by checking its primary key against the primary key of the last tuple that has been sent.

The repairing node loops through the following procedure: *(i)* if the current LSN is higher than the LSN of the last log records read, it reads a new log record; *(ii)* it checks if the tuple to which the log record applies has already been sent to the node being repaired; *(iii)* if the tuple has already been sent, the log record is added to the block of log records to be sent, and *(iv)* when the block, is full, it is added to the queue in order to be sent to the node being repaired, and a new block is started. Once the fragment replication part of the process is finished, the timer is stopped, and the last LSN to be a part of the recovery process can be calculated.

Once the blocks have been received by the node being repaired, they are added to a queue. Each block is then read from the queue and undergoes the following procedure: *(i)* each log record is extracted from the block; *(ii)* the tuple the log

record applies to is searched; and *(iii)* the log record is applied to the tuple.

The method presented in paragraph 4.4 has two implementations. For the second alternatice of that method, the node being repaired also has a special implementation for the catch-up process. It does the following: *(i)* each log record is extracted from the block, and *(ii)* the log record is applied by using the update method implemented by Berkeley DB.

In order to simplify the implementation, the methods work with records of fixed sizes and the log file contains only update operation to existing records; no insert or delete operations have been included in the log file. This allows to limit the complexity of the implementation, as insert and delete operations can cause split or join of pages in the B-tree.

Chapter 5

Experiments

In this chapter we describe the different conditions in which the experiments were carried out, and discuss their results. The experiments consisted in transferring a B-tree between two nodes, using the three different methods discussed in chapters 3 and 4. For the method "Send data only" two alternative implementation of the methods were tested, these alternatives were presented in paragraphs 4.4.1 and 4.4.2. In this chapter they will be denoted "Send data insert from bottom" and "Send data insert from top", respectively . Thus, four different protocols for transfering a B-tree between two nodes have in fact been examined.

5.1 General information about the experiments

The goal of the experiments was to evaluate the effect of different parameters on the performance of the various recovery methods. The following parameters where addressed: *(i)* the fill level of the B-tree; *(ii)* the number of log records generated per second; *(iii)* the size of the database in number of records; and *(iv)* the bandwidth of the network between the two nodes.

A reference configuration was chosen for each of the parameters, and the tests where performed varying one of the parameters at a time, all the other parameters being set to their reference value. Parameters *(i)*, *(ii)*, and *(iii)* where tested by running the implementation of the different methods as presented in chapter 4. Since we did not have the mandatory hardware available, parameter *(iv)* was tested using a simulation.

In this section we first describe the hardware platform and database used during the experiments. We then present the measures used to quantify the results of all experiments, and the reference configuration. In the last subsection, we present the simulation used for testing the effects of the network bandwidth on each of the methods.

5.1.1 Hardware platform

The experiments where run on two identical machines with the following configuration:

- **processor**: dual-CPU Athlon 1.4GHz
- **memory**: 1GB RAM
- **operating system**: Debian "Sarge" Linux with kernel 2.6.7-1-k7-smp
- **network bandwith**: 100Mbps.

The communication between the two machines was done using the TCP-IP protocol.

5.1.2 Database

The database used for the experiments was generated using Berkeley DB, an open source database system that implements B-trees, among other access methods. Using an open source database system enabled us to avoid implementing the access method, while still being provided full information about the internal structures of its B-tree implementation.

The B-tree contained 1.000.000 records. Each record contained two fields of fixed size, a primary key (4 bytes) and a string (96 bytes). The records where inserted in increasing order by primary key into the B-tree so as to obtain a high fill level. The fill level of the B-tree was approximately 100%.

5.1.3 Performance metrics

For each of the experiments we measured the following quantities:

- *Time to complete at the Repairing Node*: Total time at the node sending the data. We start counting when the first data block/page is sent, and stop when the last data block/page or log block is sent.
- *Time to complete at the Node Being Repaired*: Total time at the node receiving the data. We start counting when the first block/page is received and stop when the last data block/page or log block is processsed.

5.1.4 Reference configuration

The default configuration defined for the experiments was the following:

- *Database*: The standard database is the database presented in paragraph 5.1.2.
- *Log records per second*: The standard number of log records per second was set to 2000.
- *Fill level*: The standard fill level of the database was set to the fill level of the standard database presented on paragraph 5.1.2.
- *Network bandwidth*: The standard network bandwidth was set to 100Mbps.

5.1.5 Simulation model

Since we did not have the equipment necessary to run the implementations with different network configurations, a simulation of the different recovery methods was built in order to test the effects of the network bandwidth on the different methods. The simulation is not meant as an exact model of the system, it is meant as a simplification allowing to get insight into the behavior of a real system with different network configurations.

Figure 5.1 illustrates the model of the simulation. In this sketch, *Data block* provides the system with data blocks. For "Send all pages" and "Send leaf pages" methods, this simulates reading the pages from disk and removing the empty spaces, while for both alternatives of the "Send data" method, it simulates reading the tuples and packing them into a new block. *Log record* is responsible for generating the 2.000 log records per second, while *Pack log record* is responsible for checking whether the log record should be pack into a block and shipped or not. *Network* simulates the network connection between the two nodes and *Process data block* and *Process log block* simulate the part of the system that regenerates the B-tree from the data received, and the part of the system resposible for applying the log records received respectively. This simulation model was implemented using a descrete event simulator called JSim[1].

The model is controlled by the following parameter:

- *Data block*: average arrival rate for data blocks.
- *Log records*: arrival rate for log records.
- *Pack log records*: average time used to add each of the log records to the block.
- *Network*: average time used to send a packet over the network.

[1]The source code of the simulation can be found on Apendix B

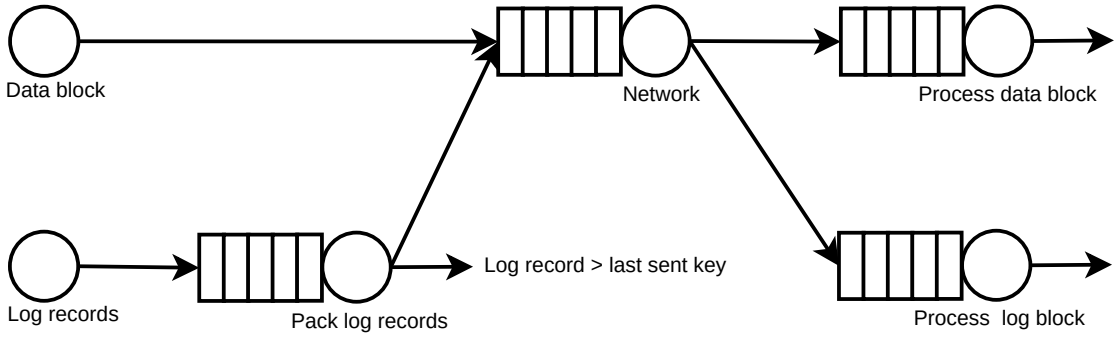

Figure 5.1: *Simulation model for the four methods studied.*

- *Process data block*: average time used to process a data block.
- *Process log block*: average time used to process a block containing log records.

The values for parameters *Data block*, *Log records*, *Pack log records*, *Process data blocks*, and *Process log blocks* were obtained from runs of the methods' implementations as the average values of the times measured. The parameter *Network* could not be measured and was estimated analytically. Indeed, assuming that there is only one network between two nodes, the service time (*Network*) for sending a message between two nodes can be calculated as follows [8]:

Let:

- *MessageSize* = size, in bytes, of the message exchanged between the node
- *MSS* = maximum segment size in bytes
- *TCPOvhd* = overhead, in bytes, of the TCP protocol
- *IPOvhd* = overhead, in bytes, of the IP protocol
- *FrameOvhd* = overhead, in bytes, of the frames
- *Overhead* = total overhead ($TCP + IP + frame$), in bytes, for all frames necessary to carry a message on the network
- *Bandwidth* = bandwidth, in Mbps, of the network
- *NDatagrams* = number of IP datagrams transmitted over the network to carry a message

Each datagram carries MSS bytes worth of message data. The number of datagrams needed to transmit a message over the network is

$$NDatagrams = \left\lceil \frac{MessageSize}{MSS} \right\rceil . \tag{5.1}$$

In the case of TCP/IP the total protocol overhead involved on transmitting a mesasge over the network is given by

$$Overhead = NDatagrams * (TCPOvhd + IPOvhd + FrameOvhd). \quad (5.2)$$

Finally, the service time at the network for a message is equal to the total number of bits needed to transmit a message (including the overhead) divided by the bandwidth in bps. Hence,

$$Network = \frac{(MessageSize + Overhead) * 8}{10^6 * Bandwidth}. \quad (5.3)$$

The overhead involved in transmitting a message is given in the following table:

Protocol	Parameter	Overhead(bytes)
TCP	TCPOvhd	20
IP version 4	IPOvhd	20
IP version 6		40
ATM		5
Ethernet	FrameOvhd	18

5.2 Experiments and results

In this paragraph we first present the results for the reference configuration defined in paragraph 5.1.4. Then we present the results obtained when varying each of the parameters at a time, with respect to the reference configuration.

5.2.1 Reference configuration

We have measured the two performance metrics for each alternative of the methods 10 time with the reference configuration. To compare the performance of the methods, we have plotted, for each of the methods, the relative variations of the two perfomance metrics as a function of the run index (see Fig. 5.2). The dispersion of the results over the different runs is small, which shows that the average over the 10 runs of the measurements for a given performance metrics and for a given method is a satisfying estimate of the performance of the method with respect to that particular metrics.

The results for the mean behavior of each method and their variants with respect to the two performance metrics are shown in table 5.1.

Figures 5.3.a and 5.3.b show the total time to complete at the repairing node, and at the node being repaired, respectively. At the repairing node, the method "Send data insert from bottom" apears to be the fastest, followed by "Send data insert from top". At the node being repaired, the method "Send data insert from bottom" apears to be the fastest, followed by "Send leaf pages" and "Send all pages", while "Send data insert from top" apears to be the slowest.

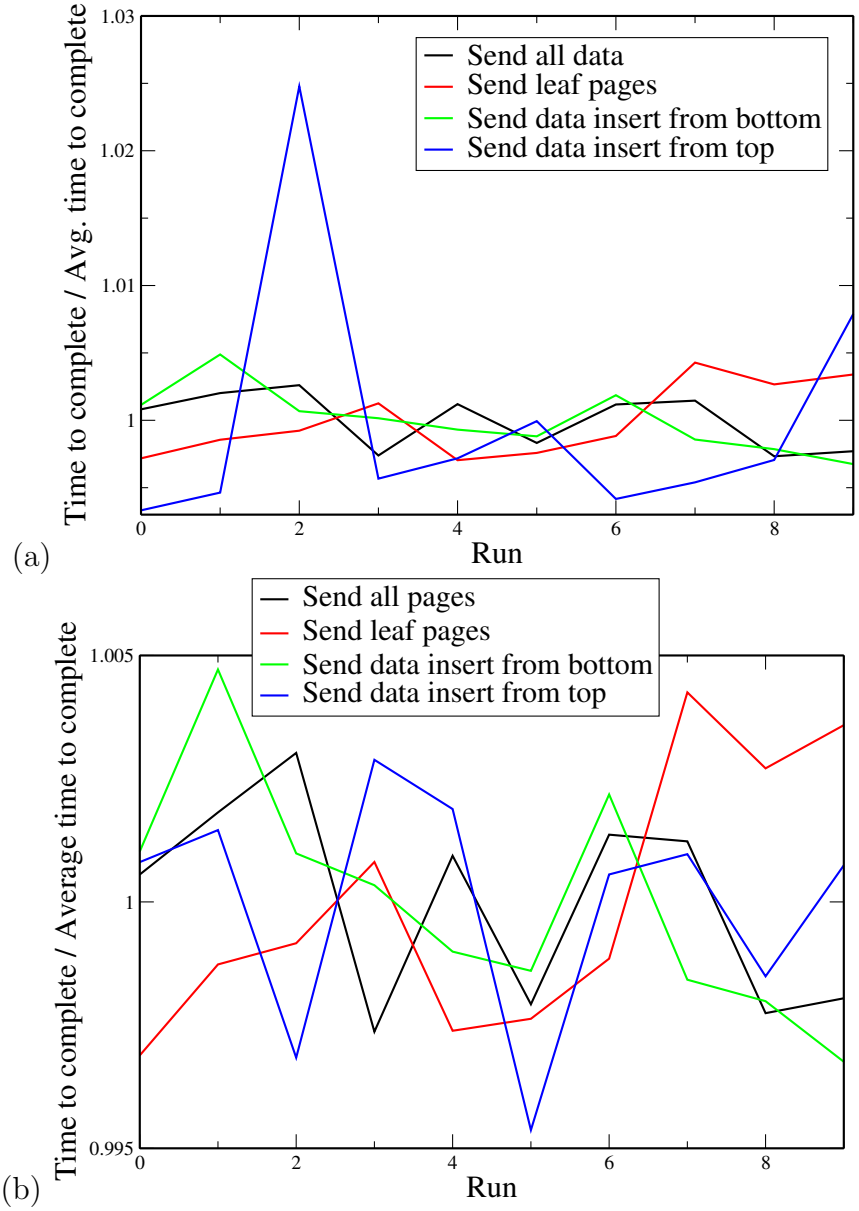

Figure 5.2: *Relative variations, as a function of the run index, of the performance metrics, measured for the different methods and their alternatives: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	9865117.1	9858224.9	9312353.7	9434051.6
Time to complete at node being repaired (μs)	9814547.1	9808255.2	9265612.1	11739039

Table 5.1: *Results averaged over 10 runs of the 4 different recovery methods.*

”Send data insert from top” and ”Send data insert from bottom” run the exact same algorithm at the node being repaired. The small difference in their performance at the repairing node might be due to ”Send data only insert from top” being much slower at the node being repaired, which might result in the buffers at this node

being filled up, and consequently the other node being idle until there is space again in the buffers.

5.2.2 Varying the B-tree's fill level

We have run each of the methods using a database with fill level of aproximately 50% to investigate what effect the variation of the fill level would have on the performance at both nodes. The results for the mean behavior of each method and their variants are shown in table 5.2, with respect to the two performance metrics.

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	13201006	9772824	9305144	9462078
Time to complete at node being repaired (μs)	23794430	9653825	9253099	11750465

Table 5.2: *Results averaged over 10 runs of the 4 different recovery methods when running the experiments with a database with fill level of aproximately 50%.*

Figure 5.4 shows a comparison between the results obtained with the reference configuration (fille level of approximately 100%) and the results obtained for a database with a fill level of aproximately 50%. "Send leaf pages", "Send data insert from top", and "Send data insert from bottom" where not very much affected by the variation of the fill level, while the performance of "Send all pages" decreased considerably with the lower fill level. This decrease in performance might have been due to the number of internal pages of the B-tree dramatically increasing and the node being repaired not being able to keep all internal pages in memory. This would cause a bigger number of disk accesses which would have a bad influence on the performance of the node. This did not happen on the other nodes because the internal pages are being regenerated with 100% fill level, so the number of internal pages for the other methods are not as high. More experiments would be necessary to invertigates what causes this behaviour.

5.2.3 Varying the number of log records generated per second

Each method has been run with 0 and 4000 log records per second and ivestigated how the performance metrics are modified. The results for the mean behavior of each method and their variants are shown in table 5.3 and 5.4, for 0 and 4000 log records per second respectively.

A comparison between the results obtained with 0, 2000 (reference configuration), and 4000 log records per second is shown in figure 5.5. "Send data only insert from bottom" appears to be the fastest method at both nodes. The time to complete at both nodes for the methods "Send leaf pages", "Send data only insert from top", and "Send data only nsert from bottom" appears to increase linearly as the

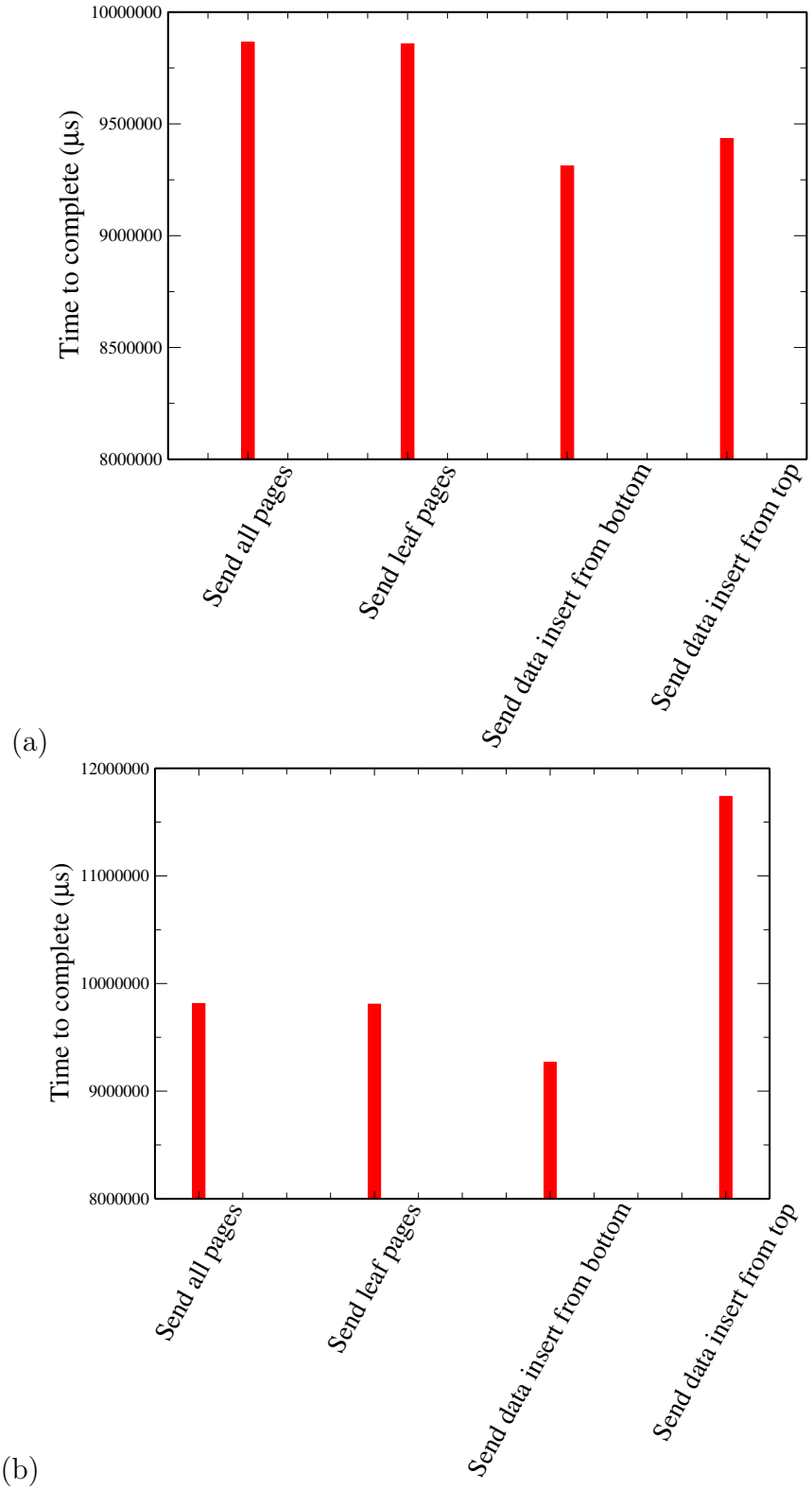

Figure 5.3: *Results averaged over ten runs of the 4 different recovery methods with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

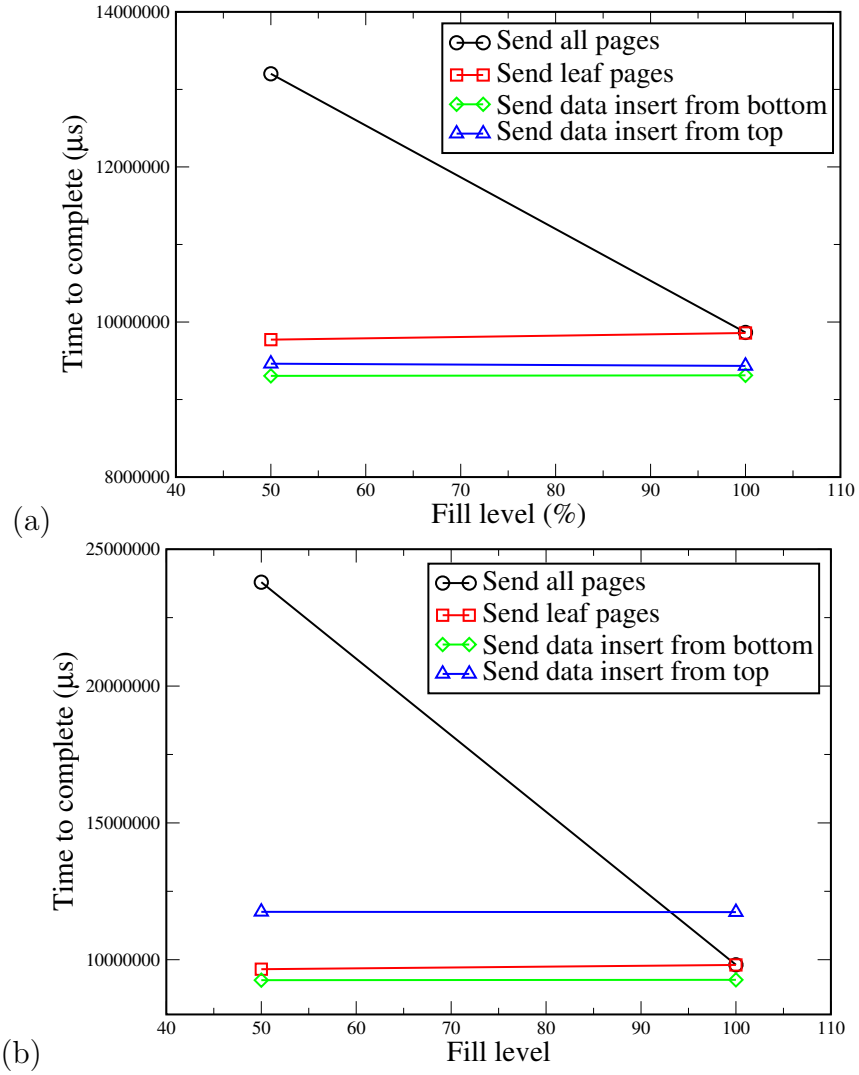

Figure 5.4: *Results averaged over 10 runs of the 4 different recovery methods when running the experiments with databases with 50% and 100% fill levels with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	9632766	9727580	9239989	9070043
Time to complete at node being repaired (μs)	9579257	9557672	9013507	11439680

Table 5.3: *Results averaged over the 10 of the 4 different recovery methods when running the experiments with 0 log records being generated per second.*

number of log records per second increases. The time to complete at both nodes for "Send all pages" presents a behaviour different from that of the other methods. More experiments would be necessary to invertigate the reasons for that behaviour.

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	11458393	10033647	9411427	9973364
Time to complete at node being repaired (μs)	20150361	9965961	9367019	12325112

Table 5.4: *Results averaged over the 10 of the 4 different recovery methods when running the experiments with 4.000 log records being generated per second.*

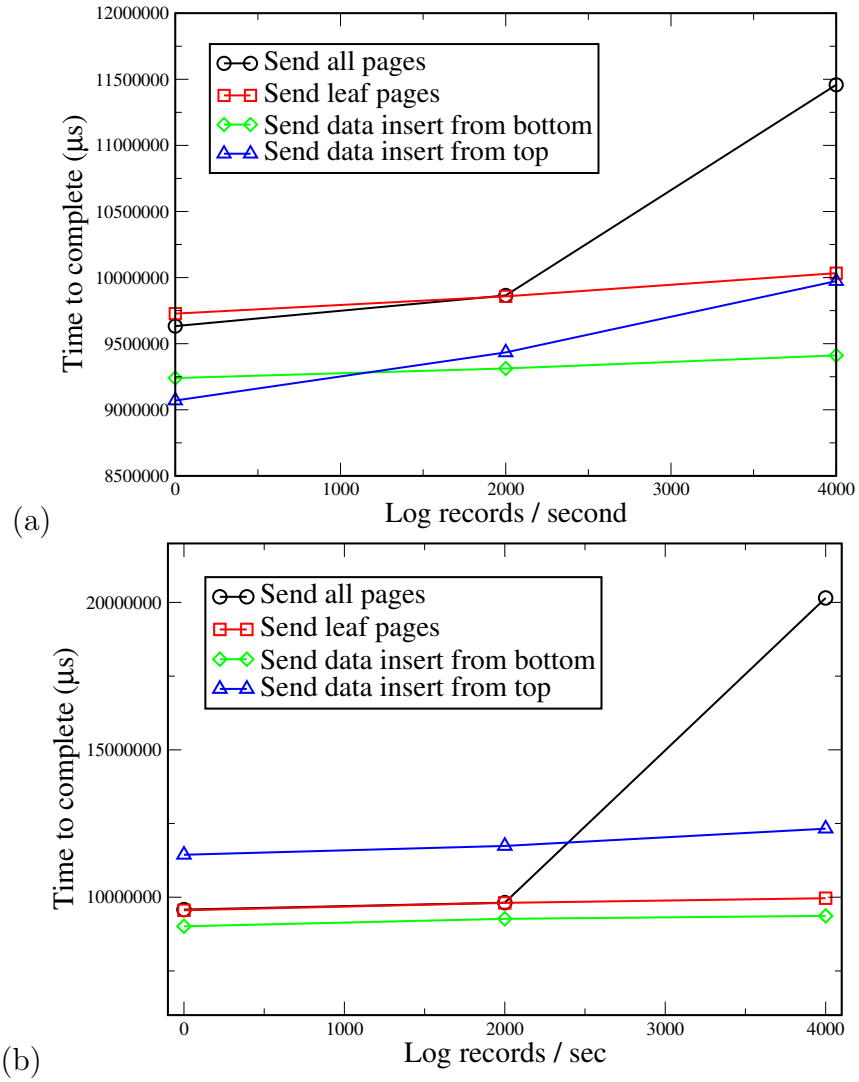

Figure 5.5: *Results averaged over 10 runs of the 4 different recovery methods when running the experiments with 0, 2000, and 4000 log records being generated per second with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

5.2.4 Varying the size of the database

We have also run each method with databases containing 500.000 and 1.500.000 tuples of equal size (those used in the reference configuration, see paragraph 5.1.2). The results for the mean behavior of each method and their variants are shown in

table 5.5 and 5.6.

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	4911645	4912763	4628378	5913510
Time to complete at node being repaired (μs)	5003681	4930106	4642916	4668942

Table 5.5: *Results averaged over the 10 runs of the 4 different recovery methods when running the experiments with a database containing 500.000 records.*

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (μs)	27665897	14703969	13863829	18271793
Time to complete at node being repaired (μs)	17873712	14778307	14055106	14283359

Table 5.6: *Results averaged over the 10 runs of the 4 different recovery methods when running the experiments with a database containing 1.500.000 records.*

Figures 5.6a. and 5.6b. show a comparison between the results obtained with databases containing 500.000, 1.000.000, and 1.500.000 fixed size tuples. At the recovering node, it appears that "Send data insert from bottom", and "Send data insert from top" are the fastest methods, followed by "Send leaf pages", while "Send all pages" appears to be the slowest of the methods. At the node being repaired, "Send data insert from bottom" appears to be the fastest method, followed by "Send leaf pages". The time to complete at both nodes for the methods "Send leaf pages", "Send data only insert from top", and "Send data only nsert from bottom" appear to increase linearly as the size of the database increases. The time to complete at both nodes for "Send all pages" presents a different behaviour than that of the other methods. More experiments would be necessary to invertigates what are the causes for this behaviour.

5.2.5 Varying the network's bandwidth

Prior to running the simulation, the first step was to find values for the various parameters of the simulation model. We used the implementation of the methods to repeatedly measure characteristic times for each of the method. After removing values corresponding to events that seemed to have suffered external influence, from other threads or processes running at the same time, the average values measured where set as model parameters for the simulation. These values are listed in table 5.7. Due to the subjective removing of inconsistent values, this mean values should not be taken as absolute values, they can be subject to errors in measurement, and/or misinterpretation[2].

We then used equation (5.3) to calculate the service time at the network for each of the methods, for bandwidths: 10Mbps, 100Mbps and 1Gbps.

[2]A graph with the different values measured for each parameter can be found on C

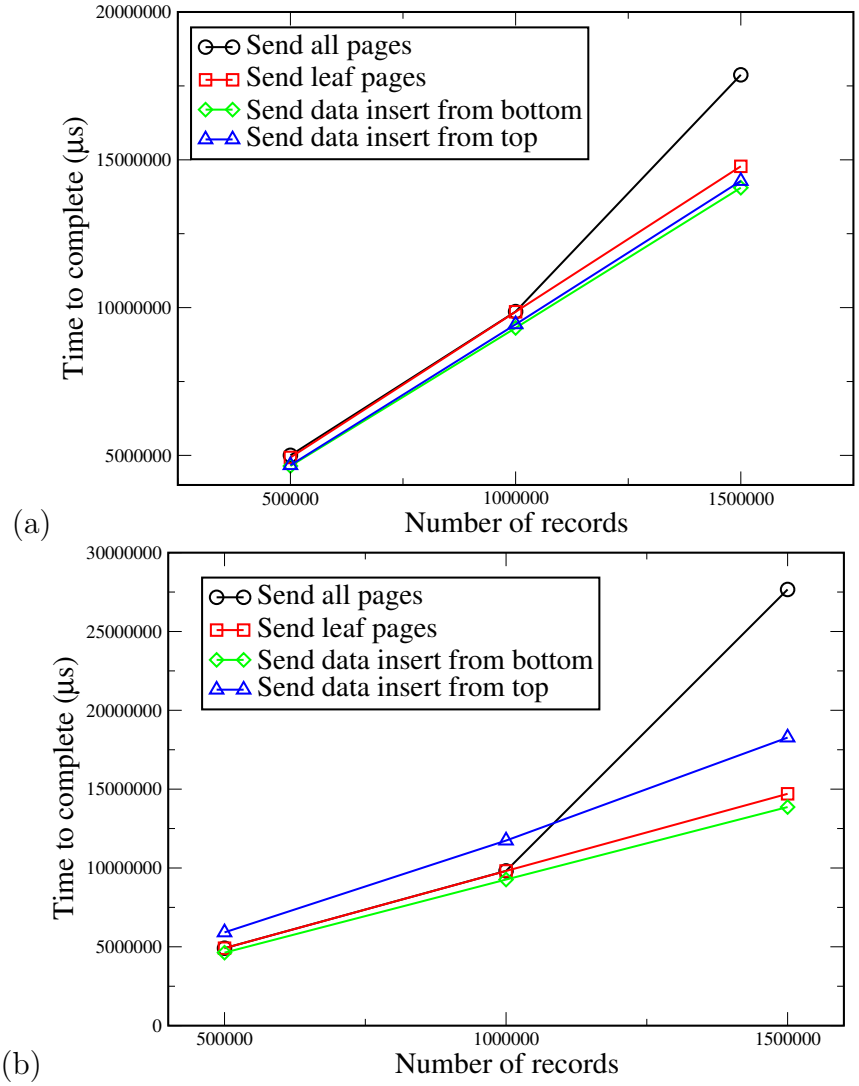

Figure 5.6: *Results averaged over ten runs of the 4 different recovery methods with database for 500.000, 1.000.000, and 2.000.000 fixed size tuples with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Prepare data	35	60	66	62
Prepare log	5	5	5	5
Process data	157	118	149	464
Process log	234	207	210	147
Number of data blocks	14119	14086	12989	12989
Average block size	7979	7979	8160	8160
MMS	1448	1448	1448	1448
Log record size	112	112	112	112

Table 5.7: *Values set to the different parameters of the simulation.*

Comparing these values to those obtained for the other parameters of the system we note that for network connections of 10Mbps and 100Mbps the values are somewhat larger than the average time for the other parameters. This could be

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
10Mbps	6661	6661	6806	6806
100Mbps log	666	666	680	680
1Gbps	66	66	68	68

Table 5.8: *Calculated service times for sending a message through the networks with bandwidths 10Mbps, 100Mbps and 1Gbps (µs).*

an indication that the bottleneck of the system for 10Mbps and 100Mbps be the network, while for 1Gbps this is not expected.

The service time for sending log blocks has to be calculated inside the simulation, because the number of log records that are sent from one node to the other depends on the completion time at the repairing node for each of the methods, in a given configuration.

After having all necessary parameters in place, we were able to run the simulation for each method. The results obtained with the simulation for network bandwidths of 10Mbps and 1Gbps are presented in tables 5.9 and 5.10.

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (µs)	104661759	104447466	98379867	98386726
Time to complete at node being repaired (µs)	104655235	104440901	98379867	98380364

Table 5.9: *Results from the simulation of the 4 different recovery methods when simulating a network bandwidth of 10Mbps.*

	Send all pages	Send leaf pages	Send data insert from bottom	Send data insert from top
Time to complete at repairing node (µs)	2496798	1928488	2192822	3133632
Time to complete at node being repaired (µs)	2499063	1929782	2195141	6286676

Table 5.10: *Results from the simulation of the 4 different recovery methods when simulating a network bandwidth of 1000Mbps.*

Figure 5.7 shows a comparison between the results with network bandwidths of 10Mbps, 100Mbps and 1Gbps, where the results for 10Mbps and 1Gbps have been obtained by using the simulation, while those for 100Mbps have been obtained by running the implementation of the methods. "Send data only insert from bottom" and "Send data only insert from top" appear to be the fastest methods at both nodes when using a 10Mbps and 100Mbps network bandwidth. As the network bandwidth increases to 1Gbps, "Send leaf pages" becomes the fastest method at both nodes. This indicates that the network might be the bottleneck for 10Mbps and 100Mbps, and that, as soon as the network is not the bottleneck, other methods that perform less work at both nodes, but transfer larger amount of data on the network, have better performance.

At 1Gbps we notice that, for the first time in these experiments, neither of the two alternatives of the "Send data only" method exibited the best performance.

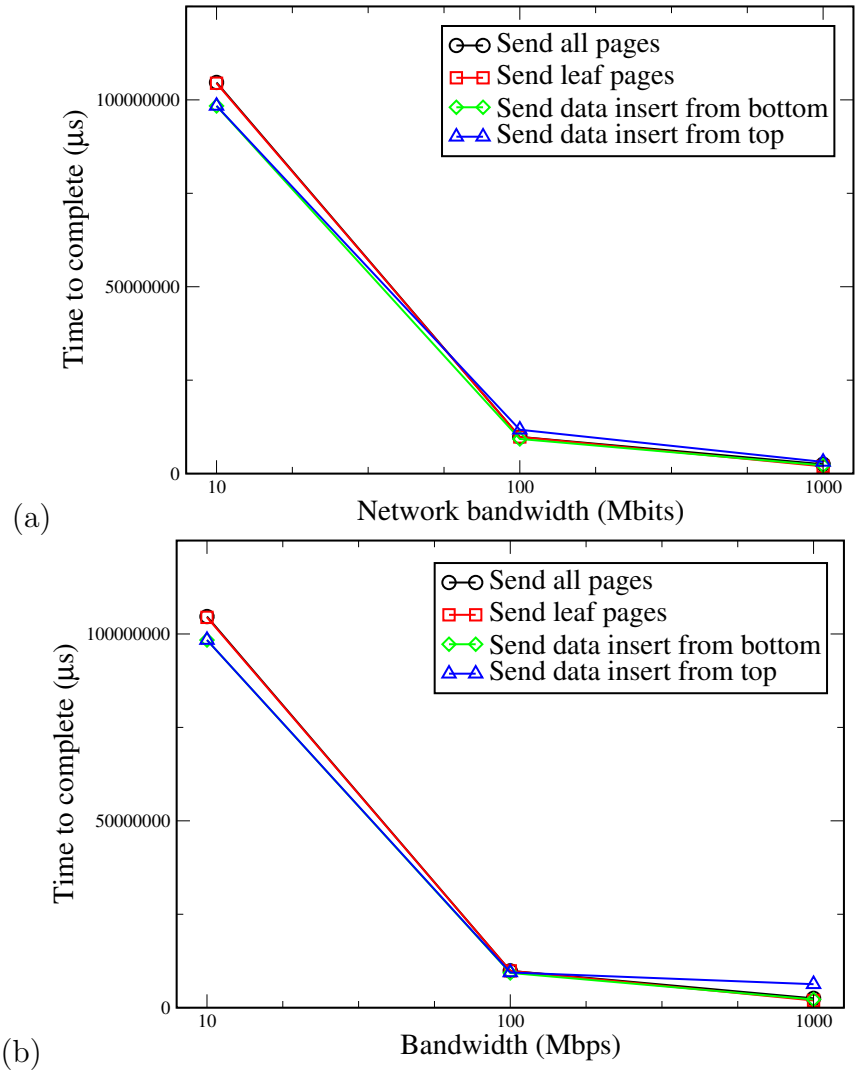

Figure 5.7: *Results averaged over 10 runs of the 4 different recovery methods with network bandwidths of 10Mbps, 100Mbps and 1Gbps with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

For that reason we decided to run several more experiments, with network bandwidths intermediate with respect to those studied earlier. The results obtained by running the simulation for each of the different methods with network bandwidths of 250Mbps, 500Mbps, 750Mbps and 1Gbps are shown in figure 5.8. It appears that as the network speed is being increased, the different methods at the node being repaired reach a point where their performance stops improving. That seems to influence the performance at the repairing node in a negative way. A possible explanation is that both the implementation of the methods and their simulation operate with a maximum queue length of 10 blocks; once the queues starts being full at the node being repaired, the repairing node has to stop processing until there is space again on the queues. More experiments varying the queue length would be necessary in order to further investigate this point.

If we consider again the parameters measured as input for the simulation, we can

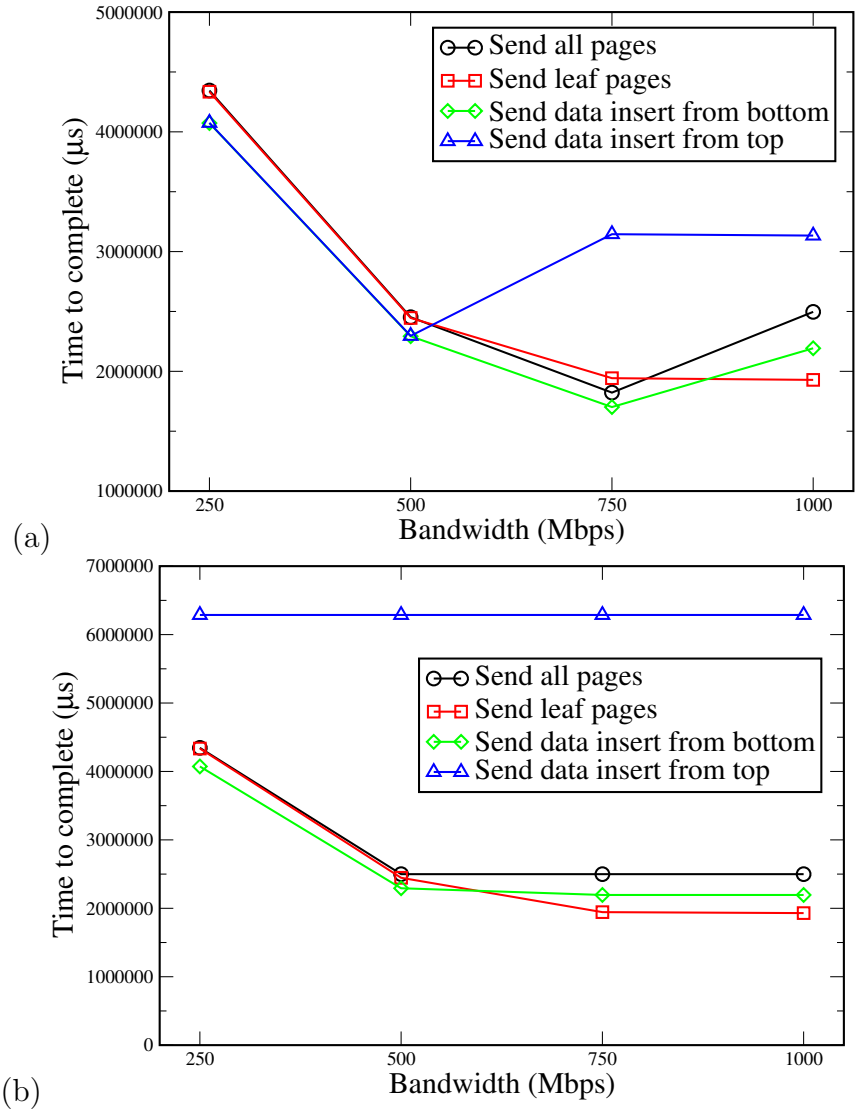

Figure 5.8: *Results averaged over 10 runs of the 4 different recovery methods with network bandwidth of 250Mbps, 500Mbps, 750Mbps and 1Gbps with respect to: (a) time to complete at repairing node, and (b) time to complete at the node being repaired.*

see that the *Process data* increases between the methods in the same order as the methods achieve saturation, this might be an indication that once the network is no longer the bottleneck of the system, the time it takes to process the data becomes the bottleneck.

Chapter 6

Conclusions and prospects

6.1 Conclusions

The objectives of this project were to evaluate the performances of different node crash recovery methods, in different configuration, in order to find out which method performs best under which circumstances. The main goal was to find out which method minimizes the completion time at both nodes involved in the recovery process, specially at the repairing node.

"Send all pages" can be considered as the method presenting the worse performance. It does not perform best in any of the different configurations examined in this project. With a 1Gbps network it presents a better performance than "Send data insert from top", but this is not sufficient for this method to be considered a valuable candidate.

"Send leaf pages" presents an intermediate performance in most of the configurations studied. It often performs worse at the repairing node than the two methods where only data is sent, but it performs often better at the node being repaired than "Send data insert from top". However, if the network reached a bandwidth of 1Gbps, this node presents the best performance at both nodes and therefore could be a good alternative.

"Send data insert from bottom" and "Send data insert from top" present the best performance at the repairing node in most of the configurations. At the node being repaired "Send data insert from bottom" performs better than "Send data insert from top", and could therefore be considered as the best method. Considering that the load imposed by transactions being executed during the recovery period is carried by the repairing node, the performances of the methods at the repairing node is more important than their performances at the node being repaired. Therefore, both methods can be considered good candidates.

It appears that the network characteristics should be taken into consideration when choosing a recovery method. For the most common network configurations

in use nowadays, i.e, 100Mbps or lower, methods that send less data through the network are recommended. Such are "Send data insert from top" and "Send data insert from bottom". However, as 1Gbits (or faster) network configurations will become more common with time, methods like "Send leaf pages" will probably become good alternatives.

One limitation to the study should be emphasized: in the experiments that we have presented, none of the nodes involved was subjected to any sort of external load. Having load applied at the repairing node, which is the conditions under which these recovery methods would run in a real system, may change the overal mutual rankings between the different methods.

6.2 Further work

Some prospects have arised from the results of this study:

1. Further experiments would be needed in order to determine why the performance of the "Send all pages" method behaved in a different manner from the other methods when the fill level decreased, the database size increased, or the number of log records per second increased. Is this behaviour due to the implementation of this method or is it characteristic of the method itself?

2. Applying a controlled load on the machine that runs as the repairing node (sender) would result in more complete experiments. How would this additional load affect the performances of the different methods and their mutual rankings?

3. It would be interesting to implement the methods keeping the database in memory and only storing it on disk in sequential order after all the database has been received and all log records have been applied, to see how this affects the performance of the methods.

Bibliography

[1] *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[2] Kjell Bratbergsengen. *Kompendium TDT4225 Lagring og behandling av store datamengder*. Tapir akademisk forlag (kompendieforlaget), 2003.

[3] Svein Erik Bratsberg, Svein-Olav Hvasshovd, and Øystein Torbjørnsen. Parallel solutions in ClustRa. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 20(2), June 1997.

[4] Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2), June 1979.

[5] Thomas H. Cormen and et al. *Introduction to algorithms*. McGraw-Hill Book Company, second edition edition, 2003.

[6] Svein-Olaf Hvasshovd. *Recovery in Parallel Database Systems*. Vieweg & Sohn, Braunshweig/Wiesbaden, 2nd edition, 1999.

[7] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *Proceedings of the 21st VLDB Conference*, pages 469–477.

[8] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services*. Prentice Hall PTR, 2002.

[9] Maitrayi Sabaratnam. *Improving Dependability in Database Management Systems*. PhD thesis, The Norwegian University of Science and Technology, 2001.

Appendices

Appendix A

Source code for the implementation of the methods

A.1 Main program

```
#include <stdio.h>
#include <stdlib.h>

#include "block_sender.h"
#include "block_receiver.h"

int main(int argc, char *argv[])
{
  int option = atoi(argv[1]);
  int alg = atoi(argv[2]);
  int i;

  if (option == 1) {
    getSenderSocket();
  } else {
    getReceiverSocket();
  }

  if (option == 1)
    switch (alg) {
      case 1:
        SendAllPages();
        break;
      case 2:
        SendLeaves();
        break;
      case 3:
        SendData();
        break;
    }
  else
    switch (alg) {
      case 1:
        ReceiveAllPages();
        break;
      case 2:
        ReceiveLeaves();
        break;
      case 3:
        ReceiveData();
        break;
    }

  if (option == 1) {
    closeSenderSocket();
  } else {
    closeReceiverSocket();
  }

  return EXIT_SUCCESS;
}
```

A.2 Sender side

```
#include "block_sender.h"

buffer_type buffer;
buffer_type log_buffer;

int send_data_socket;
int send_log_socket;

struct timeval start_time, end_time;

unsigned long last_sent_key;
char end_set = 0;


void getSenderSocket () {
  send_data_socket = getSocket();
  prepareToSend(send_data_socket, RECEIVER_IP, RECEIVER_PORT);
  send_log_socket = getSocket();
  prepareToSend(send_log_socket, RECEIVER_IP, RECEIVER_LOG_PORT);
}

void closeSenderSocket() {
  close(send_data_socket);
  close(send_log_socket);
}

unsigned long CalcLogCount(unsigned long time_spent) {
  unsigned long log_count;
  double log_records_pr_sec;
  log_records_pr_sec = LOG_RECORDS_PER_SEC;
  log_count = time_spent * (log_records_pr_sec / 1000000.0);
  return log_count;
}

void SetTimeStart() {
  gettimeofday(&start_time, 0);
}

void SetTimeEnd() {
  end_set = 1;
  gettimeofday(&end_time, 0);
}

unsigned long GetTimeSpent() {
  struct timeval current_time;
  unsigned long seconds, microseconds, time_spent;

  if (end_set == 0)
    gettimeofday(&current_time, 0);
  else
    current_time = end_time;

  seconds = (current_time.tv_sec - start_time.tv_sec);
  microseconds = (current_time.tv_usec - start_time.tv_usec);
  time_spent = (seconds * 1000000) + microseconds;
  return time_spent;
}

void *SendLog() {
  int res, old_state;
  u_int8_t *block;
  u_int32_t block_size;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);

  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);

  do {
```

```
    block_size = GetNextBlockFromBuffer(&log_buffer, block);
    res = sendPacket(send_log_socket, block, block_size);
  } while (res != sizeof(u_int8_t));

  pthread_exit(NULL);
}

void *PrepareLog() {
  LOGRECORD *lrecord;
  FILE *log;
  unsigned long next_lsn, lsn_key, time_spent;
  u_int8_t *block;
  u_int8_t end;
  db_indx_t block_offset;
  pthread_t sender;
  int old_state;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);
  pthread_create(&sender, NULL, SendLog, NULL);

  log = OpenLogReadOnly(LOG_FILE);

  lrecord = (LOGRECORD *) malloc(sizeof(LOGRECORD));
  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);
  bzero(block, MAX_PAGE_SIZE);

  block_offset = 0;
  next_lsn = 1;
  while (last_sent_key != 0) {
    time_spent = GetTimeSpent();
    if (CalcLogCount(time_spent) >= next_lsn) {
      lrecord = ReadLogRecord(log, next_lsn);
      next_lsn++;

      memcpy((void *)&lsn_key, (void *)lrecord->data_key, lrecord->len_key);
      if (lsn_key <= last_sent_key) {
        if ((MAX_PAGE_SIZE - block_offset) < sizeof(LOGRECORD)) {
          AddBlockToBuffer(&log_buffer, block, block_offset);
          block_offset = 0;
          bzero(block, MAX_PAGE_SIZE);
        }
        AddDataToBlock(block, &block_offset, (u_int8_t *) lrecord, sizeof(LOGRECORD));
      }
    }
  }
  if (block_offset != 0) {
    AddBlockToBuffer(&log_buffer, block, block_offset);
  }

  end = 0;
  AddBlockToBuffer(&log_buffer, &end, sizeof(u_int8_t));

  pthread_join(sender, NULL);

  pthread_exit(NULL);
}

void *SendPackets() {
  int res, old_state;
  u_int8_t *block;
  u_int32_t block_size;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);

  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);

  do {
    block_size = GetNextBlockFromBuffer(&buffer, block);
    res = sendPacket(send_data_socket, block, block_size);
  } while (res != sizeof(u_int8_t));
```

```
    pthread_exit(NULL);
}

void SendPagesPreFix(
        FILE *db,
        db_pgno_t pgno,
        db_indx_t page_size,
        unsigned long long *min_send_pages_time) {
148   u_int8_t *page, *sendPage;
    PAGE *header;
    db_indx_t pageSize, entry, offset;
    unsigned int record_size;
    BINTERNAL *internal;

    page = ReadPage(db, pgno, page_size);
    header = GetHeader(page);

    sendPage = (u_int8_t *) malloc(page_size);
158   memcpy(sendPage, page, page_size);
    pageSize = CompressPage(sendPage, page_size);
    AddBlockToBuffer(&buffer, sendPage, pageSize);
    free(sendPage);

    if (header->level != LEAFLEVEL) {
      for (entry=1; entry<=header->entries; entry++) {
        offset = GetOffset(page, entry);
        internal = GetDataInternal(page, offset, &record_size);
        SendPagesPreFix(db, internal->pgno, page_size, min_send_pages_time);
168     }
    } else {
      SetLastSentKey(page, &last_sent_key);
    }
    free(page);
}

void SendAllPages() {
    FILE *db;
    BTMETA *btMeta;
178   u_int8_t end;
    unsigned long totalTime, logCount;
    pthread_t sender, log_sender;

    SetTimeStart();

    db = OpenDatabaseReadOnly(DB_FILE);
    btMeta = GetMetaPage(db);
    InitBuffer(&buffer, btMeta->dbmeta.pagesize);
    InitBuffer(&log_buffer, btMeta->dbmeta.pagesize);
188   last_sent_key = 1;

    // send metapage
    AddBlockToBuffer(&buffer, (u_int8_t *) btMeta, DBMETASIZE);

    //start block sender
    pthread_create(&sender, NULL, SendPackets, NULL);
    // start log sender
    pthread_create(&log_sender, NULL, PrepareLog, NULL);

198   SendPagesPreFix(db, btMeta->root, btMeta->dbmeta.pagesize, NULL);

    // stop and wait for sender to finish
    end = 0;
    AddBlockToBuffer(&buffer, &end, sizeof(end));
    SetTimeEnd();
    last_sent_key = 0;

    pthread_join(sender, NULL);
    pthread_join(log_sender, NULL);
208
```

```
  TermBuffer(&buffer);
  TermBuffer(&log_buffer);

  totalTime = GetTimeSpent();
  logCount = CalcLogCount(totalTime);

  printf("Send all pages time: %lu logCount: %lu\n", totalTime, logCount);

  CloseDatabase(db);
}

void SendLeaves() {
  FILE *db;
  BTMETA *btMeta;
  u_int8_t *page, end;
  unsigned int pageSize;
  pthread_t sender, log_sender;
  db_indx_t pgno;
  PAGE *header;
  unsigned long totalTime, logCount;

  SetTimeStart();

  db = OpenDatabaseReadOnly(DB_FILE);
  btMeta = GetMetaPage(db);
  InitBuffer(&buffer, btMeta->dbmeta.pagesize);
  InitBuffer(&log_buffer, btMeta->dbmeta.pagesize);
  last_sent_key = 1;

  // send metapage
  AddBlockToBuffer(&buffer, (u_int8_t *) btMeta, DBMETASIZE);

  //start block sender
    pthread_create(&sender, NULL, SendPackets, NULL);
  // start log sender
  pthread_create(&log_sender, NULL, PrepareLog, NULL);

  pgno = FindFirstLeaf(db, (u_int8_t *)btMeta);

  while (pgno != 0) {
    page = ReadPage(db, pgno, btMeta->dbmeta.pagesize);
    SetLastSentKey(page, &last_sent_key);
    pageSize = CompressPage(page, btMeta->dbmeta.pagesize);
    AddBlockToBuffer(&buffer, page, pageSize);
    header = GetHeader(page);
    pgno = header->next_pgno;
  }

  // stop and wait for sender to finish
  end = 0;
  AddBlockToBuffer(&buffer, &end, sizeof(end));
  SetTimeEnd();
  last_sent_key = 0;

  pthread_join(sender, NULL);
  pthread_join(log_sender, NULL);

  TermBuffer(&buffer);
  TermBuffer(&log_buffer);

  totalTime = GetTimeSpent();
  logCount = CalcLogCount(totalTime);

  printf("Send leaf pages time: %lu logCount: %lu\n", totalTime, logCount);


    CloseDatabase(db);
}

void SendData() {
```

```
        FILE *db;
        BTMETA *btMeta;
        BKEYDATA *data, *key;
        u_int8_t *page, end, *block;
        unsigned int i, record_size_data, record_size_key;
        pthread_t sender, log_sender;
        db_indx_t pgno, offset, block_offset;
        unsigned long totalTime, logCount, blocks = 0;

288     SetTimeStart();

        db = OpenDatabaseReadOnly(DB_FILE);
        btMeta = GetMetaPage(db);
        InitBuffer(&buffer, btMeta->dbmeta.pagesize);
        InitBuffer(&log_buffer, btMeta->dbmeta.pagesize);
        last_sent_key = 1;

        // send metapage
        AddBlockToBuffer(&buffer, (u_int8_t *) btMeta, DBMETASIZE);
298
        //start block sender
        pthread_create(&sender, NULL, SendPackets, NULL);
        // start log sender
        pthread_create(&log_sender, NULL, PrepareLog, NULL);

        pgno = FindFirstLeaf(db, (u_int8_t *)btMeta);

        block = (u_int8_t *) malloc(btMeta->dbmeta.pagesize);
        bzero(block, btMeta->dbmeta.pagesize);
308     block_offset = 0;

        while (pgno != 0) {
          page = ReadPage(db, pgno, btMeta->dbmeta.pagesize);
          SetLastSentKey(page, &last_sent_key);
          for (i=1; i<=((PAGE *)page)->entries; i=i+2) {
            // two by two ... key missing
            offset = GetOffset(page, i);
            data = GetDataLeaf(page, offset, &record_size_data);
            offset = GetOffset(page, i+1);
318         key = GetDataLeaf(page, offset, &record_size_key);
            if (block_offset + (record_size_data + record_size_key )> btMeta->dbmeta.pagesize) {
              AddBlockToBuffer(&buffer, block, block_offset);
              blocks ++;
              bzero(block, btMeta->dbmeta.pagesize);
              block_offset = 0;
            }
            AddDataToBlock(block, &block_offset, (u_int8_t *)data, record_size_data);
            AddDataToBlock(block, &block_offset, (u_int8_t *)key, record_size_key);

328       }
          pgno = ((PAGE *)page)->next_pgno;
        }
        AddBlockToBuffer(&buffer, block, block_offset);
        blocks++;

        // stop and wait for sender to finish
        end = 0;
        AddBlockToBuffer(&buffer, &end, sizeof(end));
        SetTimeEnd();
338     last_sent_key = 0;

        pthread_join(sender, NULL);
        pthread_join(log_sender, NULL);

        TermBuffer(&buffer);
        TermBuffer(&log_buffer);

        totalTime = GetTimeSpent();
        logCount = CalcLogCount(totalTime);
348
```

```
    printf("Send data time: %lu logCount: %lu\n", totalTime, logCount);

    CloseDatabase(db);
}
```

A.3 Receiver side

```
#include "block_sender.h"

buffer_type buffer;
buffer_type log_buffer;

int recv_log_socket, send_log_to_socket;
int recv_data_socket, send_data_to_socket;

void getReceiverSocket() {
  recv_data_socket = getSocket();
  send_data_to_socket = prepareToListen(recv_data_socket, RECEIVER_PORT);
  recv_log_socket = getSocket();
  send_log_to_socket = prepareToListen(recv_log_socket, RECEIVER_LOG_PORT);
}

void closeReceiverSocket() {
  close(recv_data_socket);
  close(send_data_to_socket);
  close(recv_log_socket);
  close(send_log_to_socket);
}

void *ReceiveLog() {
  int old_state;
  u_int8_t *block;
  u_int32_t block_size;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);

  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);

    do {
    block_size = receivePacket(send_log_to_socket, block, MAX_PAGE_SIZE);
    AddBlockToBuffer(&log_buffer, block, block_size);
  } while (block_size != sizeof(u_int8_t));

  free(block);

    pthread_exit(NULL);
}

void *ProcessLog(void *db) {
  pthread_t receiver;
    u_int8_t *block;
  u_int32_t block_size, offset;
  LOGRECORD *lrecord;
  int old_state;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);
  pthread_create(&receiver, NULL, ReceiveLog, NULL);

  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);

  block_size = GetNextBlockFromBuffer(&log_buffer, block);
  while (block_size != sizeof(u_int8_t)) {
    offset = 0;
    // Process the log
    while (offset < block_size) {
      lrecord = (LOGRECORD *) GetDataFromBlockFixedSize(block, &offset, sizeof(LOGRECORD));
      //apply log
      FindAndSetData(
        (FILE *)db,
        lrecord->data_data,
        lrecord->len_data,
        lrecord->data_key,
        lrecord->len_key);
    }
    block_size = GetNextBlockFromBuffer(&log_buffer, block);
```

```
    }

  free(block);
    // wait for receiver to finish
    pthread_join(receiver, NULL);

  return NULL;

77 }

void *ReceivePackets() {
  int old_state;
  u_int8_t *block;
  u_int32_t block_size;

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);

  block = (u_int8_t *) malloc(MAX_PAGE_SIZE);
87

  do {
    block_size = receivePacket(send_data_to_socket, block, MAX_PAGE_SIZE);
    AddBlockToBuffer(&buffer, block, block_size);
  } while (block_size != sizeof(u_int8_t));// && block_size != 0);

  free(block);

    pthread_exit(NULL);
97 }

void ReceiveAllPages() {
  FILE *db;
  u_int8_t *btMeta;
  u_int8_t *page;
  unsigned long currentPage, totalTime = 0, totalData = 0;
  pthread_t receiver, log_receiver;
  u_int32_t block_size;

107  db = OpenDatabaseReadWrite(DB_REC);

  InitBuffer(&buffer, MAX_PAGE_SIZE);
  InitBuffer(&log_buffer, MAX_PAGE_SIZE);
  InitDatabase();

  //start block receiver
  pthread_create(&receiver, NULL, ReceivePackets, NULL);
  //start log receiver
  pthread_create(&log_receiver, NULL, ProcessLog, (void *) db);
117
  btMeta = (u_int8_t *) malloc(MAX_PAGE_SIZE);

  // get metapage
  block_size = GetNextBlockFromBuffer(&buffer, btMeta);

  SetTimeStart();

  SetMetaPage(db, btMeta);

127  page = (u_int8_t *) malloc(((BTMETA *)btMeta)->dbmeta.pagesize);

  for (currentPage=1; currentPage<=((BTMETA *)btMeta)->dbmeta.last_pgno; currentPage++) {
    bzero(page, ((BTMETA *)btMeta)->dbmeta.pagesize);
    block_size = GetNextBlockFromBuffer(&buffer, page);
    totalData += block_size;
    ExpandPage(page, block_size, ((BTMETA *)btMeta)->dbmeta.pagesize);
    AddPageToTree(db, page, 1, 1);
  }

137  // wait for receiver to finish
  pthread_join(receiver, NULL);
```

```
    pthread_join(log_receiver, NULL);

    TermBuffer(&buffer);
    TermBuffer(&log_buffer);

    CloseDatabase(db);

    SetTimeEnd();

    totalTime = GetTimeSpent();

    printf("Receive all pages time: %lu, data received: %lu\n", totalTime, totalData);

    free(page);
    //free(btMeta);
}

void ReceiveLeaves() {
    FILE *db;
    u_int8_t *btMeta;
    u_int8_t *page;
    unsigned long totalTime = 0, totalData = 0;
    pthread_t receiver, log_receiver;
    u_int32_t block_size;

    db = OpenDatabaseReadWrite(DB_REC);

    InitBuffer(&buffer, MAX_PAGE_SIZE);
    InitBuffer(&log_buffer, MAX_PAGE_SIZE);
    InitDatabase();

    //start block receiver
    pthread_create(&receiver, NULL, ReceivePackets, NULL);
    //start log receiver
    pthread_create(&log_receiver, NULL, ProcessLog, (void *) db);

    btMeta = (u_int8_t *) malloc(MAX_PAGE_SIZE);

    // get metapage
    block_size = GetNextBlockFromBuffer(&buffer, btMeta);

    SetTimeStart();

    ((BTMETA *)btMeta)->dbmeta.last_pgno = 0;
    SetMetaPage(db, btMeta);

    page = (u_int8_t *) malloc(((BTMETA *)btMeta)->dbmeta.pagesize);
    bzero(page, ((BTMETA *)btMeta)->dbmeta.pagesize);
    block_size = GetNextBlockFromBuffer(&buffer, page);

    while (block_size != sizeof(u_int8_t) && block_size != 0) {
      totalData += block_size;
      ExpandPage(page, block_size, ((BTMETA *)btMeta)->dbmeta.pagesize);
      //build tree
      AddPageToTree(db, page, 0, 1);
      bzero(page, ((BTMETA *)btMeta)->dbmeta.pagesize);
      block_size = GetNextBlockFromBuffer(&buffer, page);
    }

    // wait for receiver to finish
    pthread_join(receiver, NULL);
    pthread_join(log_receiver, NULL);

    TermBuffer(&buffer);
    TermBuffer(&log_buffer);

    CloseDatabase(db);

    SetTimeEnd();
    totalTime = GetTimeSpent();
```

```
    printf("Receive leaf pages time: %lu, data received: %lu\n", totalTime, totalData);

    free(page);
    //free(btMeta);
  }

  void ReceiveData() {
    FILE *db;
    u_int8_t *btMeta;
    u_int8_t *block, *page, *data, *key;
    PAGE *header;
    unsigned long totalTime = 0, totalData = 0;
    unsigned int record_size_data, record_size_key;
    pthread_t receiver, log_receiver;
    u_int32_t block_size;
    db_indx_t block_offset, offset;
    unsigned char end = 0;
    int count = 0;

    db = OpenDatabaseReadWrite(DB_REC);

    InitBuffer(&buffer, MAX_PAGE_SIZE);
    InitBuffer(&log_buffer, MAX_PAGE_SIZE);

    InitDatabase();

    //start block receiver
    pthread_create(&receiver, NULL, ReceivePackets, NULL);
    //start log receiver
    pthread_create(&log_receiver, NULL, ProcessLog, (void *) db);

    btMeta = (u_int8_t *) malloc(MAX_PAGE_SIZE);

    // get metapage
    block_size = GetNextBlockFromBuffer(&buffer, btMeta);

    SetTimeStart();

    SetMetaPage(db, btMeta);

    block = (u_int8_t *) malloc(((BTMETA *)btMeta)->dbmeta.pagesize);
    bzero(block, ((BTMETA *)btMeta)->dbmeta.pagesize);
    block_size = GetNextBlockFromBuffer(&buffer, block);
    count++;

    page = CreatePage(LEAFLEVEL);
    header = GetHeader(page);

    while (block_size != sizeof(u_int8_t) && block_size != 0) {
      // parse block
      block_offset = 0;
      totalData += block_size;
      while (block_offset < block_size) {
        key = (u_int8_t *)GetDataFromBlock(block, &block_offset, &record_size_key);
        data = (u_int8_t *)GetDataFromBlock(block, &block_offset, &record_size_data);
        if (CheckSpace(page, (record_size_data + record_size_key)) < 0) {
          AddPageToTree(db, page, 0, 0);
          bzero(page, ((BTMETA *)btMeta)->dbmeta.pagesize);
          page = CreatePage(LEAFLEVEL);
          header = GetHeader(page);
        }
        // set key
        offset = header->hf_offset;
        offset -= record_size_key;
        header->entries ++;
        SetOffset(page, header->entries, offset);
        SetDataLeaf(page, offset, record_size_key, key);
        //set data
        offset -=record_size_data;
```

```
        header->entries ++;
        SetOffset(page, header->entries, offset);
        SetDataLeaf(page, offset, record_size_data, data);
        header->hf_offset = offset;
      }
      bzero(block, ((BTMETA *)btMeta)->dbmeta.pagesize);
      block_size = GetNextBlockFromBuffer(&buffer, block);
      count ++;
287   }

      if (header->entries > 0)
        AddPageToTree(db, page, 0, 0);

      // wait for receiver to finish
      pthread_join(receiver, NULL);
      pthread_join(log_receiver, NULL);

      TermBuffer(&buffer);
297   TermBuffer(&log_buffer);

      CloseDatabase(db);

      SetTimeEnd();
      totalTime = GetTimeSpent();

      printf("Receive data time: %lu, data received: %lu\n", totalTime, totalData);

      free(block);
307   //free(btMeta);
      free(page);
    }
```

A.4 Libraries

```
/* Pages and blocks*/

#include "db_util.h"
#include "constants.h"

int compare (DB *db, const DBT *dbt1, const DBT *dbt2) {
  unsigned long key1, key2;

  memcpy((char *) &key1, (char *) dbt1->data, dbt1->size);
  memcpy((char *) &key2, (char *) dbt2->data, dbt2->size);
  return (key1 - key2);
}

u_int32_t prefix(DB *db, const DBT *dbt1, const DBT *dbt2) {
  return dbt1->size;
}

void InitDatabase() {
  root = current_leaf = NULL;
  current_page_no = 0;
}

FILE *OpenDatabaseReadOnly(char *path) {

  FILE *db = fopen(path, "r");

  if (db == NULL)
    printf("Error opening database readonly!\n");

  read_write_file = (pthread_mutex_t *) malloc (sizeof(pthread_mutex_t));
  pthread_mutex_init(read_write_file, NULL);

  return db;
}

FILE *OpenDatabaseReadWrite(char *path) {

  FILE *db;

  if (read_write_file == NULL)
    read_write_file = (pthread_mutex_t *) malloc (sizeof(pthread_mutex_t));
  else
    printf("read_write_file not null\n");

  pthread_mutex_init(read_write_file, NULL);

  pthread_mutex_lock(read_write_file);

  if ((db = fopen(path, "w+")) == NULL)
    printf("Error opening database read/write!\n");

  pthread_mutex_unlock(read_write_file);

  return db;
}

void CloseDatabase(FILE *db) {
  tree_item_type *current, *previous;
  for (current = root; current!=NULL; current = current->current_child)
    WritePage(
        db,
        ((PAGE *)(current->page))->pgno,
        meta_page->dbmeta.pagesize,
        current->page);

  if (root != NULL && meta_page != NULL) {
    meta_page->root = ((PAGE *)(root->page))->pgno;
    SetMetaPage(db, (u_int8_t *)meta_page);
```

```
    }

    if (root != NULL) {
      previous = root;
      for (
          current = previous->current_child;
          current!=NULL;
          current = current->current_child) {
        if (previous->page != NULL)
          free (previous->page);
        free (previous);
        previous = current;
      }
      free (previous);
    }

    root = NULL;
    current_leaf = NULL;

    if (meta_page != NULL)
      free(meta_page);

    pthread_mutex_lock(read_write_file);

    if (fclose(db) != 0)
      printf("error closing database\n");

    pthread_mutex_unlock(read_write_file);

    if (pthread_mutex_destroy(read_write_file) != 0)
      printf("error on destroy\n");
    free(read_write_file);
    read_write_file = NULL;

  }

  BTMETA *GetMetaPage(FILE *db) {
    BTMETA *metaPage;

    metaPage = (BTMETA *) malloc(DBMETASIZE);
    fseek(db, 0, SEEK_SET);
    fread(metaPage, DBMETASIZE, 1, db);
    return metaPage;
  }

  void SetMetaPage(FILE *db, u_int8_t *metaPage) {
     u_int8_t *filler;

    pthread_mutex_lock(read_write_file);

    fseek(db, 0, SEEK_SET);
    fwrite(metaPage, DBMETASIZE, 1, db);

    fseek(db, DBMETASIZE, SEEK_SET);
    filler = (u_int8_t *) malloc(((BTMETA *)metaPage)->dbmeta.pagesize - DBMETASIZE);
    bzero(filler , ((BTMETA *)metaPage)->dbmeta.pagesize - DBMETASIZE);
    fwrite(filler , ((BTMETA *)metaPage)->dbmeta.pagesize - DBMETASIZE, 1, db);

    pthread_mutex_unlock(read_write_file);

    meta_page = (BTMETA *)metaPage;
  }

  u_int8_t *CreatePage(u_int8_t level) {
    u_int8_t *page;
    PAGE *header;

    page = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
    bzero(page, meta_page->dbmeta.pagesize);
    header = GetHeader(page);
```

```
    current_page_no++;
    meta_page->dbmeta.last_pgno = current_page_no;

    //Initialize page Header
    header->pgno = current_page_no;
    header->hf_offset = meta_page->dbmeta.pagesize;
    header->entries = 0;
    header->type = (level == LEAFLEVEL) ? P_LBTREE : P_IBTREE;
    header->level = level;

    return page;
  }

  void UpdatePointer(tree_item_type *parent_page, db_pgno_t pgno, db_pgno_t new_pgno) {
    db_indx_t entry, offset;
    unsigned int record_size;
    BINTERNAL *data;
    PAGE *header;
    u_int8_t * page;

    page = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
    page = parent_page->page;

    header = GetHeader(page);

    offset = GetOffset(page, parent_page->current_entry);
    data = GetDataInternal(page, offset, &record_size);
    if (data->pgno == pgno && data->unused != 1) {
      data->pgno = new_pgno;
      data->unused = 1;
      SetDataInternal(page, offset, record_size, (u_int8_t *)data);
    }
    parent_page->current_entry ++;

  }

  int CheckSpace(u_int8_t *page, db_indx_t record_size) {
    PAGE *header;
    db_indx_t space;

    header = GetHeader(page);
    space = header->hf_offset;
    space -= (SIZEOF_PAGE + (sizeof(db_indx_t) * header->entries));
    return (space - record_size);
  }

  void AddPointer(FILE *db, u_int8_t *parent, u_int8_t *page, db_pgno_t new_pgno) {
    db_indx_t entry, offset, rsize;
    unsigned int record_size;
    BINTERNAL *data_internal;
    BKEYDATA *data_leaf;
    u_int8_t *new_parent = NULL, *old_parent = NULL; //, *parent_parent;
    tree_item_type *current = NULL;

    if (((PAGE *)page)->level == LEAFLEVEL) {
      offset = GetOffset(page, 1);
      data_leaf = GetDataLeaf(page, offset, &record_size);
      record_size = sizeof(BINTERNAL) + (record_size - sizeof(BKEYDATA));
      data_internal = (BINTERNAL *) malloc(record_size);
      data_internal->len = data_leaf->len;
      data_internal->type = ((PAGE *)page)->type;
      data_internal->nrecs = ((PAGE *)page)->entries;
      memcpy((void *) data_internal->data, (void *) data_leaf->data, data_leaf->len);
    } else {
      offset = GetOffset(page, 1);
      data_internal = GetDataInternal(page, offset, &record_size);
    }

    data_internal->pgno = new_pgno;
    rsize = record_size;
```

```
    if (CheckSpace(parent, rsize) <= 0) {
211   // Get current parent
      old_parent = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
      memcpy((void *) old_parent, parent, meta_page->dbmeta.pagesize);

      // Write parent to disk
      WritePage(db, ((PAGE *)parent)->pgno, meta_page->dbmeta.pagesize, parent);

      // Create new parent
      new_parent = CreatePage(((PAGE *)old_parent)->level);

221   //set new_parent to parent
      memcpy ((void *) parent, (void *) new_parent, meta_page->dbmeta.pagesize);

      // check that the parent has a parent
      current = root;
      while ( ((PAGE *)(current->page))->level > ((PAGE *)parent)->level)
        current = current->current_child;

      current->current_entry = 1;

231   }

    //update pointer in the parent
    ((PAGE *)parent)->entries++;
    offset = (((PAGE *)parent)->hf_offset);
    offset -= rsize;
    entry = ((PAGE *)parent)->entries;
    SetOffset(parent, entry, offset);
    SetDataInternal(parent, offset, record_size, (u_int8_t *)data_internal);
    ((PAGE *)parent)->hf_offset = offset;
241
    // if parent does not exist, create new root and set it as parent
    if (current != NULL) {
      if (current->parent == NULL) {
        current->parent = (tree_item_type *) malloc(sizeof(tree_item_type));
        current->parent->page = CreatePage(((PAGE *)parent)->level + 1);
        current->parent->parent = NULL;
        current->parent->current_child = root;
        current->parent->current_entry = 1;
        root = current->parent;
251     AddPointer(db, current->parent->page, old_parent, ((PAGE *)old_parent)->pgno);
      }
    }

    if (new_parent != NULL) {
      memcpy ((void *) new_parent, (void *) parent, meta_page->dbmeta.pagesize);
      AddPointer(db, current->parent->page, new_parent, ((PAGE *)new_parent)->pgno);
    }
    free (new_parent);
    free (old_parent);
261 }

  int AddPageToTreeCreateParent(FILE *db, u_int8_t *page, db_pgno_t new_pgno) {
    PAGE *page_header;
    u_int8_t *prev_page = NULL;
    u_int8_t *parent_page = NULL;

    //Get page header
    page_header = GetHeader(page);

271 if (root == NULL) {
      // Create root page
      root = (tree_item_type *) malloc(sizeof(tree_item_type));
      root->page = CreatePage(page_header->level + 1);
      root->parent = NULL;
      root->current_child = NULL;
      root->current_entry = 1;
      // Add place for the received leave
```

```
      current_leaf = (tree_item_type *) malloc(sizeof(tree_item_type));
      current_leaf->page = NULL;
281   current_leaf->parent = root;
      current_leaf->current_child = NULL;
      current_leaf->current_entry = 1;
      root->current_child = current_leaf;
    }

    // Update link to next in prev page and write to disk
    if (current_leaf->page != NULL) {
      prev_page = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
      memcpy((void *) prev_page, (void *) current_leaf->page, meta_page->dbmeta.pagesize);
291   ((PAGE *)prev_page)->next_pgno = new_pgno;
      WritePage(db, ((PAGE *)prev_page)->pgno, meta_page->dbmeta.pagesize, prev_page);
    } else {
      current_leaf->page = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
    }

    current_leaf->current_entry = 1;
    memcpy((void *) current_leaf->page, (void *) page, meta_page->dbmeta.pagesize);

    parent_page = (u_int8_t *) malloc(meta_page->dbmeta.pagesize);
301 memcpy(
          (void *) parent_page,
          (void *) ((tree_item_type *)current_leaf->parent)->page,
          meta_page->dbmeta.pagesize);

    page = current_leaf->page;
    page_header = GetHeader(page);
    page_header->pgno = new_pgno;
    meta_page->dbmeta.last_pgno = new_pgno;

311 page_header->prev_pgno = (prev_page == NULL) ? 0 : ((PAGE *)prev_page)->pgno;
    AddPointer(db, parent_page, page, new_pgno);

    memcpy(
          (void *) ((tree_item_type *)current_leaf->parent)->page,
          (void *) parent_page,
          meta_page->dbmeta.pagesize);

    free(prev_page);
    free(parent_page);
321
    return 0;
  }

  int AddPageToTreeUpdateParent(FILE *db, u_int8_t *page, db_pgno_t new_pgno) {
    PAGE *page_header, *prev_page_header; //, *parent_page_header;
    u_int8_t *prev_page, *parent_page;
    tree_item_type *parent, *current;
    int pagesize = ((BTMETA *)meta_page)->dbmeta.pagesize;

331 //Get current and prev header
    page_header = GetHeader(page);
    if (current_leaf != NULL) {
      prev_page = malloc(pagesize);
      memcpy(prev_page, current_leaf->page, pagesize);
      prev_page_header = GetHeader(prev_page);
    }

    if (root == NULL) {
      if (meta_page->root == page_header->pgno) {
341     root = (tree_item_type *) malloc(sizeof(tree_item_type));
        root->page = (u_int8_t *) malloc(pagesize);
        root->current_entry = 1;
        memcpy(root->page, page, pagesize);
        root->parent = NULL;
        root->current_child = NULL;
        current_leaf = root;
        current = root;
```

```
      } else
        return(-1);
    } else {
      if (prev_page_header->level != page_header->level) {
        current = root;
        parent = root;
        while (parent != NULL) {
          if (((PAGE *)parent->page)->level >= page_header->level) {
            current = parent;
          }
          parent = parent->current_child;
        }
        if (((PAGE *)current->page)->level == page_header->level) {
          WritePage(db, ((PAGE *)current->page)->pgno, pagesize, current->page);
          memcpy((void *) current->page, page, pagesize);
          current->current_entry = 1;
        }
        else {
          parent = current;
          current_leaf = (tree_item_type *) malloc(sizeof(tree_item_type));
          current_leaf->parent = parent;
          current_leaf->page = (u_int8_t *) malloc(pagesize);
          memcpy((void *) current_leaf->page, page, pagesize);
          current_leaf->current_child = NULL;
          current_leaf->current_entry = 1;
          parent->current_child = current_leaf;
          current = current_leaf;
        }
      } else {
        if (prev_page_header->level == page_header->level) {
          memcpy((void *)current_leaf->page, (void *)page, pagesize);
          current = current_leaf;
          current->current_entry = 1;
          page_header = GetHeader(current_leaf->page);

          if (prev_page_header != NULL) {
            prev_page_header->next_pgno = new_pgno;
            page_header->prev_pgno = prev_page_header->pgno;
          } else {
            page_header->prev_pgno = 0;
          }
          page_header->next_pgno = 0;
          WritePage(db, prev_page_header->pgno, pagesize, prev_page);
        }
      }
    }

    //update pointer in the parent and pgno
    page_header = GetHeader(current->page);
    parent = current->parent;
    if (parent != NULL) {
      parent_page = parent->page;
      UpdatePointer(parent, page_header->pgno, new_pgno);
    }
    page_header->pgno = new_pgno;
    return(0);
  }

  /* parent_received_created indicates whether the internal pages are being
     sent of should be generated. Use 0 for sent and 1 for generated.
     new_page_no used to decide whether as new pgno is needed for the page      */
  void AddPageToTree (
        FILE *db,
        u_int8_t *page,
        u_int8_t parent_received_created,
        u_int8_t new_page_no) {
    db_pgno_t new_pgno;

    //Get new pgno
    if (new_page_no != 0) {
```

```
      current_page_no++;
      new_pgno = current_page_no;
421   }
      else
        new_pgno = ((PAGE *)page)->pgno;

      //build tree or receive internal?
      if (parent_received_created == 0)
        AddPageToTreeCreateParent(db, page, new_pgno);
      else
        AddPageToTreeUpdateParent(db, page, new_pgno);
    }
431
    u_int8_t *ReadPage (FILE *db, db_pgno_t page_no, int page_size) {
        unsigned long int offset;
        u_int8_t *page, *found_page;
      u_int8_t found = 0;
      tree_item_type *current;

      current = root;
      while (current != NULL) {
        page = current->page;
441     if (((PAGE *)page)->pgno == page_no) {
          found_page = page;
          found = 1;
        }
        current = current->current_child;
      }

      page = (u_int8_t *) malloc(page_size);
      if (found != 1) {
        pthread_mutex_lock(read_write_file);
451     offset = page_no * page_size;
        if (fseek(db, offset, SEEK_SET) != 0)
        if (fread(page, page_size, 1, db) == 0)
          printf("\nERROR READING PAGE \n\n");
        pthread_mutex_unlock(read_write_file);
      } else {
        memcpy(page, found_page, page_size);
      }
            return page;
    }
461
    void WritePage (FILE *db, db_pgno_t page_no, int page_size, u_int8_t *page) {
        unsigned long int offset;

        offset = page_no * page_size;
      pthread_mutex_lock(read_write_file);
        if (fseek(db, offset, SEEK_SET) != 0)
        printf("Error on fseek\n");
        if (fwrite(page, page_size, 1, db) != 1)
        printf("Error writting page\n");
471   pthread_mutex_unlock(read_write_file);
      fflush(db);
    }

    PAGE *GetHeader (u_int8_t *page) {
        return (PAGE *)page;
    }

    unsigned int CompressPage (u_int8_t *page, int page_size) {
      PAGE *header;
481   unsigned int offset;

      header = GetHeader(page);
      offset = SIZEOF_PAGE + ((header->entries + 1) * sizeof(db_indx_t));
      memcpy (
            (void *)(page + offset),
            (void *) (page + header->hf_offset),
            (page_size - header->hf_offset));
```

```
    page_size = offset + (page_size - header->hf_offset);
    return page_size;
}


void ExpandPage(u_int8_t *page, int current_size, int page_size) {
    unsigned int offset, start;
    PAGE *header;
    u_int8_t *tmp_page = (u_int8_t *) malloc(page_size);

    header = GetHeader(page);
    start = SIZEOF_PAGE + ((header->entries + 1) * sizeof(db_indx_t));
    offset = page_size - (current_size - start);

    memcpy ((void *) tmp_page, (void *) page, start);
    memcpy (
          (void *) (tmp_page + header->hf_offset),
          (void *) (page + start),
          (current_size - start));
    memcpy ((void *) page, (void*) tmp_page, page_size);
}

db_indx_t GetOffset(u_int8_t *page, db_indx_t entry) {
          int position;
    db_indx_t *offset;

          position = SIZEOF_PAGE + ((entry - 1)* sizeof(db_indx_t));
    offset = (db_indx_t *)(page + position);
          return *offset;
}

void SetOffset(u_int8_t *page, db_indx_t entry, db_indx_t offset) {
    int position;

    position = SIZEOF_PAGE + ((entry - 1) * sizeof(db_indx_t));
    memcpy((db_indx_t *) (page + position), &offset, sizeof(db_indx_t));
          //*(page + position) = offset;
}

BINTERNAL *GetDataInternal (
          u_int8_t *page,
          db_indx_t offset,
          unsigned int *record_size) {
    BINTERNAL *data;

    data = (BINTERNAL *) (page + offset);
    *record_size = sizeof(BINTERNAL) + data->len - 1;
    //printf("length in getDataInternal: %i \n", data->len);
    return data;
}

void SetDataInternal (
          u_int8_t *page,
          db_indx_t offset,
          unsigned int record_size,
          u_int8_t *data) {
    memcpy((void *) (page + offset), (void *) data, record_size);
}

BKEYDATA *GetDataLeaf(
          u_int8_t *page,
          db_indx_t offset,
          unsigned int *record_size) {
    BKEYDATA *data;

    data = (BKEYDATA *) (page + offset);
    *record_size = sizeof(BKEYDATA) + data->len - 1;
    return data;
}
```

```
void SetDataLeaf (
        u_int8_t *page,
        db_indx_t offset,
        unsigned int record_size,
        u_int8_t *data) {
  memcpy((void *) (page + offset), (void *) data, record_size);
}

void AddDataToBlock (
        u_int8_t *block,
        db_indx_t *offset,
        u_int8_t *data,
        unsigned int size) {
  memcpy ((void *) (block + *offset), (void *) data, size);
  *offset += size;
}

BKEYDATA *GetDataFromBlock (
        u_int8_t *block,
        db_indx_t *offset,
        unsigned int *record_size) {
  BKEYDATA *data;

  data = (BKEYDATA *) (block + *offset);
  *record_size = sizeof(BKEYDATA) + data->len - 1;
  *offset += *record_size;
  return data;
}

u_int8_t *GetDataFromBlockFixedSize (
        u_int8_t *block,
        u_int32_t *offset,
        u_int32_t record_size) {
  u_int8_t *data;

  data = (u_int8_t *) (block + *offset);
  *offset += record_size;
  return data;
}
db_pgno_t FindFirstLeaf (FILE *db, u_int8_t *metaPage) {
  u_int8_t *page;
  PAGE *header;
  BINTERNAL *internal;
  db_indx_t offset;
  unsigned int *record_size;

  record_size = (unsigned int *) malloc(sizeof(unsigned int));
  page = ReadPage(db, ((BTMETA *)metaPage)->root, ((BTMETA *)metaPage)->dbmeta.pagesize);
  header = GetHeader(page);
  while (header->level != LEAFLEVEL) {
    offset = GetOffset(page, 1);
    internal = GetDataInternal(page, offset, record_size);
    page = ReadPage(db, internal->pgno, ((BTMETA *)metaPage)->dbmeta.pagesize);
    header = GetHeader(page);
  }

  return(header->pgno);
}

void SetLastSentKey(u_int8_t *page, unsigned long *last_sent_key) {
  PAGE *header;
  db_indx_t offset;
  unsigned int record_size;
  BKEYDATA *data;

  header = GetHeader(page);
  offset = GetOffset(page, header->entries - 1);
  data = GetDataLeaf(page, offset, &record_size);
  memcpy(last_sent_key, data->data, sizeof(unsigned long));
  //printf ("last sent key %i \n", *last_sent_key);
```

```
    }

    // find
    BINTERNAL *FindInternal (
            u_int8_t *page,
            db_indx_t start,
            db_indx_t end,
           DBT *looking_for) {
      BINTERNAL *mid, *next;
      db_indx_t mid_index, offset;
      unsigned int record_size;
      DBT mid_dbt, next_dbt;
      long long compare_res, compare_res_next;

      //Get object in the middle
      mid_index = (start + end) / 2;
      offset = GetOffset(page, mid_index);
      mid = GetDataInternal(page, offset, &record_size);

      // Copy object into DBT and compare
      mid_dbt.data = malloc(looking_for->size);
      if (mid->len == 0) {
        bzero(mid_dbt.data, 4);
        mid->len = 4;
      } else {
        memcpy(mid_dbt.data, mid->data, mid->len);
      }
      mid_dbt.size = mid->len;
      compare_res = compare(NULL, looking_for, &mid_dbt);

      // Check if found
      if (start == end) {
        return mid;
      }
      if (end == (start + 1)) {
        offset = GetOffset(page, end);
        next = GetDataInternal(page, offset, &record_size);

        next_dbt.data = malloc(looking_for->size);
        if (next->len == 0) {
          bzero(next_dbt.data, 4);
          next->len = 4;
        } else {
          memcpy(next_dbt.data, next->data, next->len);
        }
        next_dbt.size = next->len;
        compare_res_next = compare(NULL, looking_for, &next_dbt);
        if ((compare_res >= 0) && (compare_res_next < 0)) {
          return mid;
        } else {
          return next;
        }
      }

      // Search within the correct half
      if (compare_res <= 0)
        return FindInternal (page, start, mid_index, looking_for);
      else
        return FindInternal (page, mid_index, end, looking_for);
    }

    int *FindLeaf (u_int8_t *page, db_indx_t start, db_indx_t end, DBT *looking_for) {
      BKEYDATA *mid, *next;
      db_indx_t mid_index, offset;
      unsigned int record_size;
      DBT mid_dbt, next_dbt;
      long long compare_res, compare_res_next;

      //Get object in the middle
```

```
    mid_index = (start + end) / 2;
    if ((mid_index % 2) == 0) {
701   mid_index --;
    }

    offset = GetOffset(page, mid_index);
    mid = GetDataLeaf(page, offset, &record_size);

    // Copy object into DBT and compare
    mid_dbt.data = malloc(looking_for->size);
    if (mid->len == 0) {
      bzero(mid_dbt.data, 4);
711   mid->len = 4;
    } else {
      memcpy(mid_dbt.data, mid->data, mid->len);
    }
    mid_dbt.size = mid->len;
    compare_res = compare(NULL, looking_for, &mid_dbt);

    //printf("compare_res: %li\n", compare_res);

    // Check if found
721 if (start == end) {
      return mid_index;
    }

    if (end == (start + 2)) {
      offset = GetOffset(page, end);
      next = GetDataLeaf(page, offset, &record_size);

      next_dbt.data = malloc(looking_for->size);
      if (next->len == 0) {
731     bzero(next_dbt.data, 4);
        next->len = 4;
      } else {
        memcpy(next_dbt.data, next->data, next->len);
      }
      next_dbt.size = next->len;
      compare_res_next = compare(NULL, looking_for, &next_dbt);
      if ((compare_res >= 0) && (compare_res_next < 0)) {
        return mid_index;
      } else {
741     return end;
      }
    }

    // Search within the correct half
    if (compare_res <= 0)
      return FindLeaf (page, start, mid_index, looking_for);
    else
      return FindLeaf(page, mid_index, end, looking_for);
  }
751
  db_pgno_t Search(FILE *db, u_int8_t *page, DBT *looking_for) {
    PAGE *header;
    db_indx_t offset;
    BINTERNAL *data = NULL, *prev_data = NULL;
    unsigned int record_size, i;
    DBT current_data;
    unsigned long found;
    long long compare_res;
    int pagesize = meta_page->dbmeta.pagesize;
761 db_pgno_t pgno;

    current_data.data = malloc(looking_for->size);

    if (page != NULL) {
      header = GetHeader(page);
      if (header->level != LEAFLEVEL && header->entries > 0) {
```

```
        data = FindInternal(page, 1, header->entries, looking_for);

771     page = ReadPage(db, data->pgno, pagesize);
        pgno = Search(db, page, looking_for);

      } else {
        pgno = ((PAGE *)page)->pgno;
      }
    } else {
      pgno = 0;
    }
    return pgno;
781 }

  void FindAndSetData(
          FILE *db,
          u_int8_t data_data[MAXLENGTH_DATA],
          u_int16_t data_len,
          u_int8_t key_data[MAXLENGTH_KEY],
          u_int16_t key_len) {
    u_int8_t *current_page;
    PAGE *header;
791 DBT looking_for, current_data;
    unsigned int i, record_size;
    BKEYDATA *data;
    db_indx_t offset;
    unsigned long key1;
    db_pgno_t pgno;
    long long compare_res;
    int pagesize = meta_page->dbmeta.pagesize;

    if (root != NULL) {
801   looking_for.data = (char *) malloc(key_len);
      memcpy(looking_for.data, key_data, key_len);
      looking_for.size = key_len;

      current_data.data = (char *) malloc(key_len);

      memcpy((char *) &key1, (char *) looking_for.data, looking_for.size);
      //printf("created DBT %li\n", key1);

      current_page = (u_int8_t *) malloc(pagesize);
811   memcpy((void *) current_page, (void *)root->page, pagesize);

      pgno = Search(db, current_page, &looking_for);

      if (pgno != 0)
        current_page = ReadPage(db, pgno, pagesize);
      else
        current_page = NULL;

      if (current_page != NULL) {
821     header = GetHeader(current_page);
        if (header->level == LEAFLEVEL) {
          compare_res = 1;

          i = FindLeaf(current_page, 1, header->entries - 1, &looking_for);

          offset = GetOffset(current_page, i);
          data = GetDataLeaf(current_page, offset, &record_size);

          free(current_data.data);
831       current_data.data = (char *) malloc(data->len);
          memcpy(current_data.data, data->data, data->len);
          current_data.size = data->len;

          compare_res = compare(NULL, &looking_for, &current_data);

          i++;
          //set data
```

```
          if (compare(NULL, &looking_for, &current_data) == 0) {
            offset = GetOffset(current_page, i);
841         data = GetDataLeaf(current_page, offset, &record_size);
            memcpy((void *)data->data, (void *)data_data, data->len);
          }
          WritePage(db, header->pgno, pagesize, current_page);
        } else {
          printf("returned not leaf\n");
        }
      } else {
        printf("returned NULL\n");
      }
851   }
      free(looking_for.data);
      free(current_data.data);
    }

    /* Log */

    #include "log_util.h"


861 FILE *OpenLogReadOnly(char *path) {
      FILE *log = fopen(path, "r");
      if (log == NULL)
        printf("error opening log file\n");
        return log;
    }

    LOGRECORD *ReadLogRecord(FILE *log, int LSN) {
      LOGRECORD *log_record;
      unsigned long offset;
871
      log_record = (LOGRECORD *) malloc(sizeof(LOGRECORD));

      offset = LSN * sizeof(LOGRECORD);
      fseek(log, offset, SEEK_SET);

      offset = fread(log_record, sizeof(LOGRECORD), 1, log);
      return log_record;
    }

881 void CloseLog(FILE *log) {
      fclose(log);
    }


    /* Queues */
    #include "buffer_util.h"

    void InitBuffer(buffer_type *buffer, int size) {
      int i;
891
      pthread_cond_init(&(buffer->empty), NULL);
      pthread_cond_init(&(buffer->full), NULL);
      pthread_mutex_init(&(buffer->allow), NULL);

      buffer->free = BUFFER_SIZE;
      buffer->start = 0;
      buffer->end = 0;

      for (i=0; i<BUFFER_SIZE; i++) {
901     buffer->queue[i] = (u_int8_t *)malloc(size);
        buffer->block_size[i] = 0;
      }
    }

    void TermBuffer(buffer_type *buffer) {
      int i;
```

```
    pthread_cond_destroy(&(buffer->empty));
    pthread_cond_destroy(&(buffer->full));
    pthread_mutex_destroy(&(buffer->allow));

    buffer->free = 0;
    buffer->start = 0;
    buffer->end = 0;

    for (i=0; i<BUFFER_SIZE; i++) {
      free(buffer->queue[i]);
      buffer->block_size[i] = 0;
    }
}

void AddBlockToBuffer(buffer_type *buffer, u_int8_t *block, u_int32_t size) {
    pthread_mutex_lock(&(buffer->allow));
    if (buffer->free == 0) {
        pthread_cond_wait(&(buffer->full), &(buffer->allow));
    }
    buffer->free--;
      memcpy(buffer->queue[buffer->end], block, size);
    buffer->block_size[buffer->end] = size;
    buffer->end = (buffer->end + 1) % BUFFER_SIZE;
    pthread_cond_signal(&(buffer->empty));
    pthread_mutex_unlock(&(buffer->allow));
}

u_int32_t GetNextBlockFromBuffer(buffer_type *buffer, u_int8_t *block) {
  u_int32_t size;
  pthread_mutex_lock(&(buffer->allow));
  if (buffer->free == BUFFER_SIZE) {
    pthread_cond_wait(&(buffer->empty), &(buffer->allow));
  }
  size = buffer->block_size[buffer->start];
  memcpy(block, buffer->queue[buffer->start], size);
  buffer->start = (buffer->start + 1) % BUFFER_SIZE;
  buffer->free++;
  pthread_cond_signal(&(buffer->full));
  pthread_mutex_unlock(&(buffer->allow));
  return(size);
}

/* Network */

#include "net_util.h"

int getSocket() {
  return(socket(AF_INET, SOCK_STREAM, 0));
}

int prepareToListen(int socket, int port) {
  struct sockaddr_in any_addr, addr;
  int size, connfd;

  //~ printf("Prepare to listen %i\n", port);

  bzero(&any_addr, sizeof(any_addr));
  any_addr.sin_family = AF_INET;
  any_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  any_addr.sin_port = htons(port);

  if (bind(socket, (struct sockaddr*) &any_addr, sizeof(any_addr)) != 0)
    printf("Error in bind\n");
  if (listen(socket, 1) != 0)
    printf("Error in listen\n");

  size = sizeof(addr);
  connfd = accept(socket, (struct sockaddr*) &addr, &size);

  if (connfd < 0)
```

```
      printf("Error in accept\n");

981   return(connfd);
    }

    int prepareToSend(int socket, char *ip, int port) {
      struct sockaddr_in addr;
      struct hostent *h;

      //inet_pton(AF_INET, ip, &addr.sin_addr);
      if ((h = gethostbyname(ip)) != NULL) {
        addr.sin_family = h->h_addrtype;
991     memcpy((char *) &addr.sin_addr.s_addr, h->h_addr_list[0], h->h_length);
        addr.sin_port = htons(port);
      }

      while (connect(socket, (struct sockaddr*) &addr, sizeof(addr)) < 0) {
      //  printf(".");
      }

      return 0;
    }
1001
    int sendPacket(int socket, u_int8_t *packet, int size) {
      unsigned int sent_size, dummy, sent = 0;

      // Agree with the receiver on the number of bytes to send
      write(socket, &size, sizeof(int));

      while (sent < size) {
        sent_size = write(socket, (packet + sent), (size - sent));
        sent += sent_size;
1011  }
      return(sent);
    }

    int receivePacket(int socket, u_int8_t *packet, int size) {
      unsigned int recv_size, dummy, left_to_receive;

      // Agree with the sender on the number of bytes to send

      dummy = recv(socket, &recv_size, sizeof(unsigned int), MSG_WAITALL);
1021
      if (recv_size < size)
        size = recv_size;

      left_to_receive = size;
      while (left_to_receive > 0) {
        recv_size = read(socket, packet, left_to_receive);
        packet += recv_size;
        left_to_receive -= recv_size;
      }
1031
      return(size);
    }

    int closeConnection(int socket) {
      return(close(socket));
    }
```

Appendix B

Source code for the simulation

B.1 Main program

```
import cz.zcu.fav.kiv.jsim.*;

public class Simulation {

  public static void main(String[] args) {
    JSimSimulation simulation = null;
    QueueWithServer
        prepare_log_queue,
        net_queue_data,
        net_queue_log,
        process_log_queue,
        process_data_queue;
    NetServer net_server;
    PageLogServer process_log_server, process_data_server;
    LogBlocksServer prepare_log_server;
    PageSource page_source;
    LogSource log_source;

    double arrival_rate_page = 62.0;
    double arrival_rate_log = 500.0;

    double service_time_prepare_log = 5.0;
    double service_time_process_data = 464.0;
    double service_time_process_log = 147.0;

    int avg_block_size = 8160;
    int max_block_size_log = 8192;
    int count_data_blocks = 12989;
    int log_record_size = 112;

    double net_bandwidth = 1000;
    double max_message_size = 1448;

    boolean queues_are_empty = false;


    try
    {
      System.out.println("Initializing the simulation..");

      simulation = new JSimSimulation("Queueing Networks Simulation");

      prepare_log_queue = new QueueWithServer("Pack log records", simulation, null);
      net_queue_data = new QueueWithServer("Network queue", simulation, null);
      net_queue_log = new QueueWithServer("Network queue", simulation, null);
      process_data_queue = new QueueWithServer("Process data queue", simulation, null);
      process_log_queue = new QueueWithServer("Process log queue", simulation, null);

      page_source = new PageSource(
                "Page source",
                simulation,
                arrival_rate_page,
                net_queue_data,
                count_data_blocks,
                avg_block_size);
      log_source = new LogSource(
                "Log Source",
                simulation,
                arrival_rate_log,
                prepare_log_queue,
                page_source);

      prepare_log_server = new LogBlocksServer(
                "Prepare log server",
                simulation,
                service_time_prepare_log,
                (int)Math.floor( (double)max_block_size_log / (double)log_record_size),
                log_record_size,
```

```
                prepare_log_queue,
                net_queue_log,
                page_source);

        net_server = new NetServer(
                "Network server",
                simulation,
                net_bandwidth,
                max_message_size,
                net_queue_data,
                net_queue_log,
                process_data_queue,
                process_log_queue,
                page_source,
                prepare_log_server);

        process_data_server = new PageLogServer(
                "Process data server",
                simulation,
                service_time_process_data,
                process_data_queue,
                count_data_blocks,
                log_record_size,
                net_server);

        process_log_server = new PageLogServer(
                "Process log server",
                simulation,
                service_time_process_log,
                process_log_queue,
                count_data_blocks,
                log_record_size,
                net_server);

        prepare_log_queue.setServer(prepare_log_server);
        net_queue_data.setServer(net_server);
        net_queue_log.setServer(net_server);
        process_data_queue.setServer(process_data_server);
        process_log_queue.setServer(process_log_server);

        simulation.message("Activating the generators...");

        page_source.activate(0.0);
        log_source.activate(0.0);

        simulation.message("Running the simulation, please wait.");

        while ((!process_data_server.getEnd()) || (!queues_are_empty)) {
          if (!simulation.step())
            simulation.message("Step returned false ");
          else
            simulation.message("Current simulation time " + simulation.getCurrentTime());
          queues_are_empty =
                prepare_log_queue.empty() &&
                net_queue_data.empty() &&
                net_queue_log.empty() &&
                process_data_queue.empty() &&
                process_log_queue.empty();
        }

        simulation.message("\n\nSimulation interrupted at time " + simulation.getCurrentTime());
        simulation.message("\n\nTotal time at sender: " + net_server.getEndSenderTime());
        simulation.message("\n\nTotal time at receiver: "
                + (simulation.getCurrentTime() - net_server.getStartRecvTime()));
        simulation.message("\n\n");
      } catch (JSimException e) {
        e.printStackTrace();
        e.printComment(System.err);
      } finally {
        simulation.shutdown();
```

```
        }
      }
142 }
```

B.2 Sources

```
import cz.zcu.fav.kiv.jsim.*;

public class PageSource extends JSimProcess
{
  private double lambda;
  private QueueWithServer queue;
  private int count_blocks;
  private int avg_block_size;

  private boolean end;
  private double sent_pcent;


  // constructor
  public PageSource(
        String name,
        JSimSimulation sim,
        double l,
        QueueWithServer q,
        int parCountBlocks,
        int parBlockSize)
                throws
                        JSimSimulationAlreadyTerminatedException,
                        JSimInvalidParametersException,
                        JSimTooManyProcessesException
  {
    super(name, sim);
    lambda = l;
    queue = q;
    count_blocks = parCountBlocks;
    avg_block_size = parBlockSize;

    end = false;
    sent_pcent = 0.0;
  }

  protected void life()
  {
    JSimLink link;
    int count;

    try  {
      for (count = 1; count <= count_blocks; count ++) {
        link = new JSimLink(new Transaction(1, avg_block_size, count));

        link.into(queue);

        hold(20.0); // adding to queue

        if (queue.getServer().isIdle())
        {
          queue.getServer().activate(myParent.getCurrentTime());
        }
        message("added page " + count);
        hold(lambda);

        if (queue.cardinal() >= 10) {
          passivate();
        }

      }
    } catch (JSimException e) {
      e.printStackTrace();
      e.printComment(System.err);
    }
  }
```

```
  public void isEnd(long trans_count) {
    if (trans_count == count_blocks)
      end = true;
  }

  public boolean getEnd() {
    return end;
  }

  public void setPcent(long trans_count) {
    sent_pcent = (double) trans_count / count_blocks;
  }

  public double getPcent() {
    return sent_pcent;
  }
}

import cz.zcu.fav.kiv.jsim.*;

public class LogSource extends JSimProcess
{
  private double lambda;
  private QueueWithServer queue;
  private boolean end;
  PageSource page_source;

  // constructor
  public LogSource(
        String name,
        JSimSimulation sim,
        double l,
        QueueWithServer q,
        PageSource parPageSource)
                throws
                        JSimSimulationAlreadyTerminatedException,
                        JSimInvalidParametersException,
                        JSimTooManyProcessesException
  {
    super(name, sim);
    lambda = l;
    queue = q;
    page_source = parPageSource;
  }

  protected void life()
  {
    JSimLink       link;

    long count;

    try  {
      while (!page_source.getEnd()) {
        link = new JSimLink(new Transaction(0, 0, 0));
        link.into(queue);
        if (queue.getServer().isIdle())
        {
          queue.getServer().activate(myParent.getCurrentTime());
        }
        hold(lambda);
      }
    } catch (JSimException e) {
      e.printStackTrace();
      e.printComment(System.err);
    }
  }

}
```

B.3 Transactions

```
import cz.zcu.fav.kiv.jsim.*;

public class Transaction
{
  private int trans_type;
  private int block_size;
  private int trans_count;

  public Transaction(int tType, int bSize, int tCount) {
    trans_type = tType;
    block_size = bSize;
    trans_count = tCount;
  }

  public int getTransType() {
    return trans_type;
  }

  public int getBlockSize() {
    return block_size;
  }

  public void setBlockSize(int parBlockSize) {
    block_size = parBlockSize;
  }
  public int getTransCount() {
    return trans_count;
  }

}
```

B.4 Queues

```
import cz.zcu.fav.kiv.jsim.*;

public class QueueWithServer extends JSimHead
{
  private JSimProcess         server;

  public QueueWithServer(
        String name,
        JSimSimulation sim,
        JSimProcess s)
                throws
                        JSimInvalidParametersException,
                        JSimTooManyHeadsException
  {
    super(name, sim);
    server = s;
  }

  public JSimProcess getServer() {
    return server;
  }

  public void setServer(JSimProcess s) {
    server = s;
  }
}
```

B.5 Servers

```
import cz.zcu.fav.kiv.jsim.*;

public class PageLogServer extends JSimProcess
{
  private double service_time;
  private QueueWithServer queueIn;
  private QueueWithServer queueOut;
  private int max_trans_count;
  private int log_record_size;
  private NetServer net_server;

  private boolean end;

  public PageLogServer(
        String name,
        JSimSimulation sim,
        double parServiceTime,
        QueueWithServer parQueueIn,
        int parMaxTransCount,
        int parLogRecordSize,
        NetServer parNetServer)
                throws
                        JSimSimulationAlreadyTerminatedException,
                        JSimInvalidParametersException,
                        JSimTooManyProcessesException
  {
    super(name, sim);
    service_time = parServiceTime;
    queueIn = parQueueIn;
    max_trans_count = parMaxTransCount;
    log_record_size = parLogRecordSize;
    net_server = parNetServer;

    end = false;
  }

  protected void life()
  {
    Transaction   t;
    JSimLink      link;

    try
    {
      while (true)
      {
        if (queueIn.empty()) {
          passivate();
        } else {

          link = queueIn.first();

          t = (Transaction) link.getData();
          link.out();

          hold(20.0); // removing from queue

          if (net_server.isIdle()) {
            net_server.activate(myParent.getCurrentTime());
          }

          if (t.getTransType() == 1)
            hold(service_time);
          else
            hold(service_time * (t.getBlockSize() / log_record_size));

          message("processing transactions count " + t.getTransCount());

          if (t.getTransType() == 1 && t.getTransCount() == max_trans_count) {
```

```
            end = true;
          }

          link = null;

        }
      }
    } catch (JSimException e)  {
      e.printStackTrace();
      e.printComment(System.err);
    }
  }

  public boolean getEnd() {
    return end;
  }

}

import cz.zcu.fav.kiv.jsim.*;

public class LogBlocksServer extends JSimProcess
{
  private double service_time;
  private long max_log_records;
  private int log_record_size;
  private QueueWithServer queueIn;
  private QueueWithServer queueOut;
  private PageSource page_source;

  private int current_count;

  public LogBlocksServer(
        String name,
        JSimSimulation sim,
        double parServiceTime,
        long parMaxLogRecords,
        int parLogRecordSize,
        QueueWithServer parQueueIn,
        QueueWithServer parQueueOut,
        PageSource parPageSource)
                throws
                        JSimSimulationAlreadyTerminatedException,
                        JSimInvalidParametersException,
                        JSimTooManyProcessesException
  {
    super(name, sim);
    service_time = parServiceTime;
    max_log_records = parMaxLogRecords;
    log_record_size = parLogRecordSize;
    queueIn = parQueueIn;
    queueOut = parQueueOut;
    page_source = parPageSource;
    current_count = 0;
  }

  protected void life()
  {
    Transaction   t;
    JSimLink      link;

    try
    {
      while (true)
      {
        if (queueIn.empty()) {
          passivate();
        } else {

          hold(service_time);
```

```
            link = queueIn.first();

            t = (Transaction) link.getData();
            link.out();

            if (queueOut.cardinal() >= 10)
              passivate();

            if (current_count != max_log_records) {
              if (JSimSystem.uniform(0.0, 1.0) < page_source.getPcent())
                current_count ++;
              link = null;
            } else {
              t.setBlockSize(current_count * log_record_size);
              link.into(queueOut);

              hold(20.0); //adding to queue;

              if (queueOut.getServer().isIdle()) {
                queueOut.getServer().activate(myParent.getCurrentTime());
              }
              current_count = 0;
            }

          }
        }
      } catch (JSimException e)  {
        e.printStackTrace();
        e.printComment(System.err);
      }
    }

  }

  import cz.zcu.fav.kiv.jsim.*;
  import java.lang.Math;

  public class NetServer extends JSimProcess
  {
    private double bandwidth;
    private double max_msg_size;
    private QueueWithServer queueInData;
    private QueueWithServer queueInLog;
    private QueueWithServer queueOutData;
    private QueueWithServer queueOutLog;
    private PageSource page_source;
    private LogBlocksServer log_blocks_server;

    private double end_sender_time = 0;
    private double start_receiver_time = 0;

    public NetServer(
          String name,
          JSimSimulation sim,
          double parBandwidth,
          double parMaxMessageSize,
          QueueWithServer parQueueInData,
          QueueWithServer parQueueInLog,
          QueueWithServer parQueueOutData,
          QueueWithServer parQueueOutLog,
          PageSource parPageSource,
          LogBlocksServer parLogBlocksServer)
                  throws
                          JSimSimulationAlreadyTerminatedException,
                          JSimInvalidParametersException,
                          JSimTooManyProcessesException
    {
      super(name, sim);
      bandwidth = parBandwidth;
```

```
    max_msg_size = parMaxMessageSize;
    queueInData = parQueueInData;
    queueInLog = parQueueInLog;
    queueOutData = parQueueOutData;
    queueOutLog = parQueueOutLog;
    page_source = parPageSource;
    log_blocks_server = parLogBlocksServer;
  }

  protected void life()
  {
    Transaction t;
    JSimLink link;
    JSimHead head;
    long count;
    int queue;
    double service_time, overhead, ndatagrams;

    try
    {
      for (count=0;;count++)
      {
        if (((Math.IEEEremainder(count,2)) != 0 && !queueInData.empty()) ||
              queueInLog.empty()) {
          queue = 1;
        } else {
          message("\n\n*** queue 2 ***\n\n");
          queue = 0;
        }
        if (queueInData.empty() && queueInLog.empty()) {
          passivate();
        } else {

          if (queue == 1)
            link = queueInData.first();
          else
            link = queueInLog.first();

          hold(20.0); // removing from queue

          t = (Transaction) link.getData();
          link.out();

          ndatagrams = Math.ceil(t.getBlockSize() / max_msg_size);

          overhead = ndatagrams * (20 + 20 + 18);

          service_time = ((t.getBlockSize() + overhead) * 8.0) / bandwidth;

          hold(service_time);

          if(queue == 1 && page_source.isIdle()) {
            page_source.activate(myParent.getCurrentTime());
          }

          if(queue == 0 && log_blocks_server.isIdle()) {
            log_blocks_server.activate(myParent.getCurrentTime());
          }


          if (start_receiver_time == 0.0)
            start_receiver_time = myParent.getCurrentTime();

          if (t.getTransType() == 1) {
            page_source.setPcent(t.getTransCount());
            page_source.isEnd(t.getTransCount());
            if (page_source.getEnd())
              end_sender_time = myParent.getCurrentTime();
            link.into(queueOutData);
            if (queueOutData.getServer().isIdle()) {
```

```
                queueOutData.getServer().activate(myParent.getCurrentTime());
              }
            } else {
              link.into(queueOutLog);
              if (queueOutLog.getServer().isIdle()) {
284             queueOutLog.getServer().activate(myParent.getCurrentTime());
              }
            }

            hold(20.0); // adding to queue;

            if (queue == 1 && queueOutData.cardinal() >= 10) {
              passivate();
            }
            if (queue == 0 && queueOutLog.cardinal() >= 10) {
294           passivate();
            }


          }
        }
      } catch (JSimException e)  {
        e.printStackTrace();
        e.printComment(System.err);
      }
304   }

    public double getEndSenderTime() {
      return end_sender_time;
    }

    public double getStartRecvTime() {
      return start_receiver_time;
    }

314 }
```

Appendix C

Measured values for the simulation

C.1 Send all pages

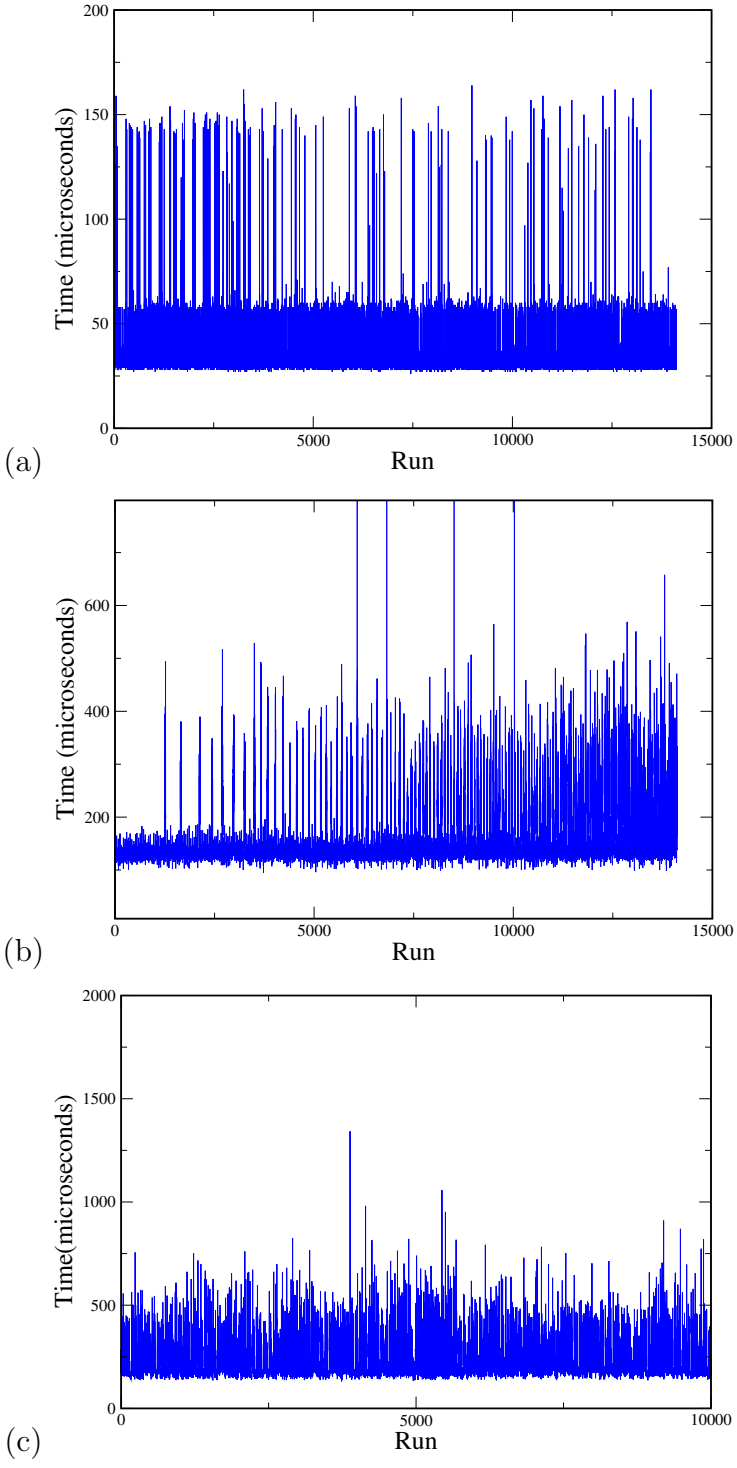

Figure C.1: *Measured values for (a) Data block, (b) Process data block, (c) each log record in Process log block.*

C.2 Send leaf pages

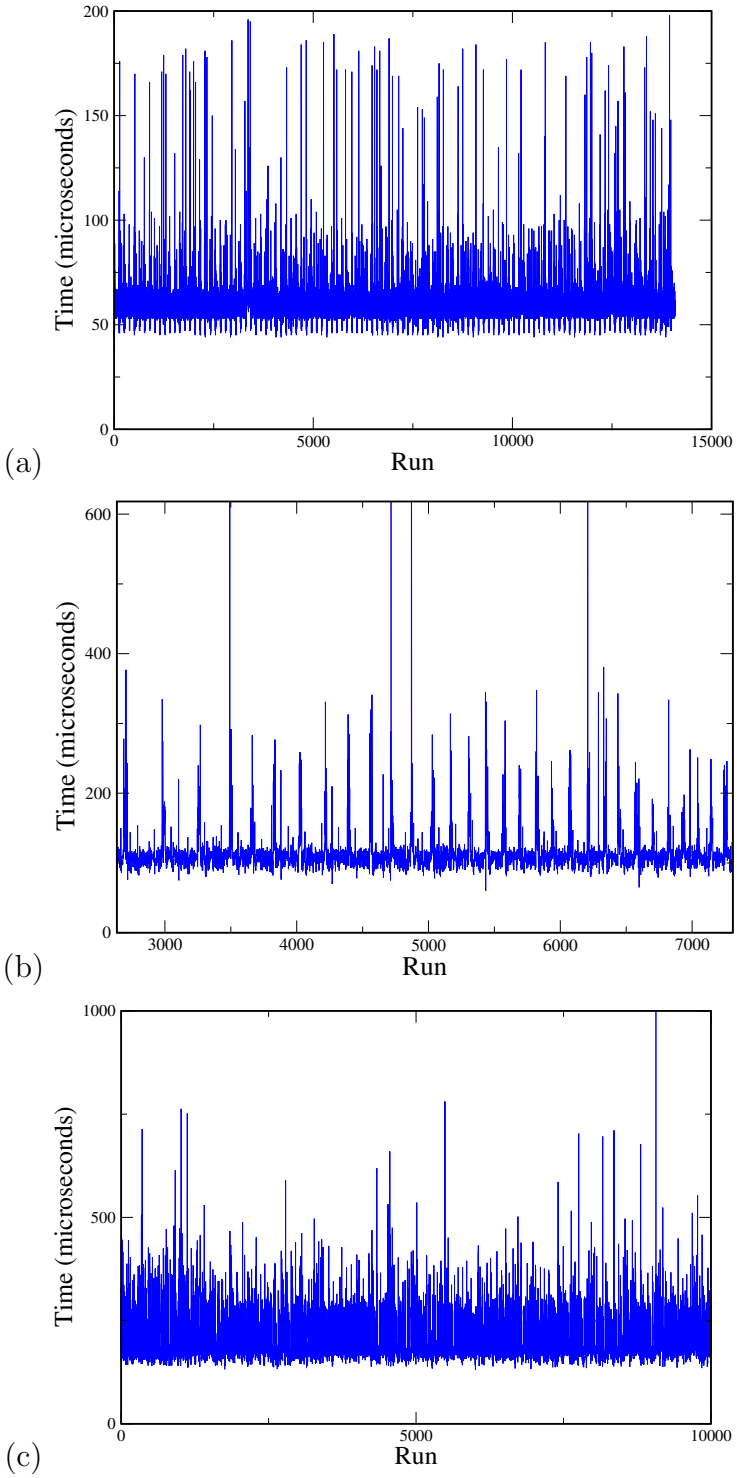

Figure C.2: *Measured values for (a) Data block, (b) Process data block, (c) each log record in Process log block.*

C.3 Send data insert from bottom

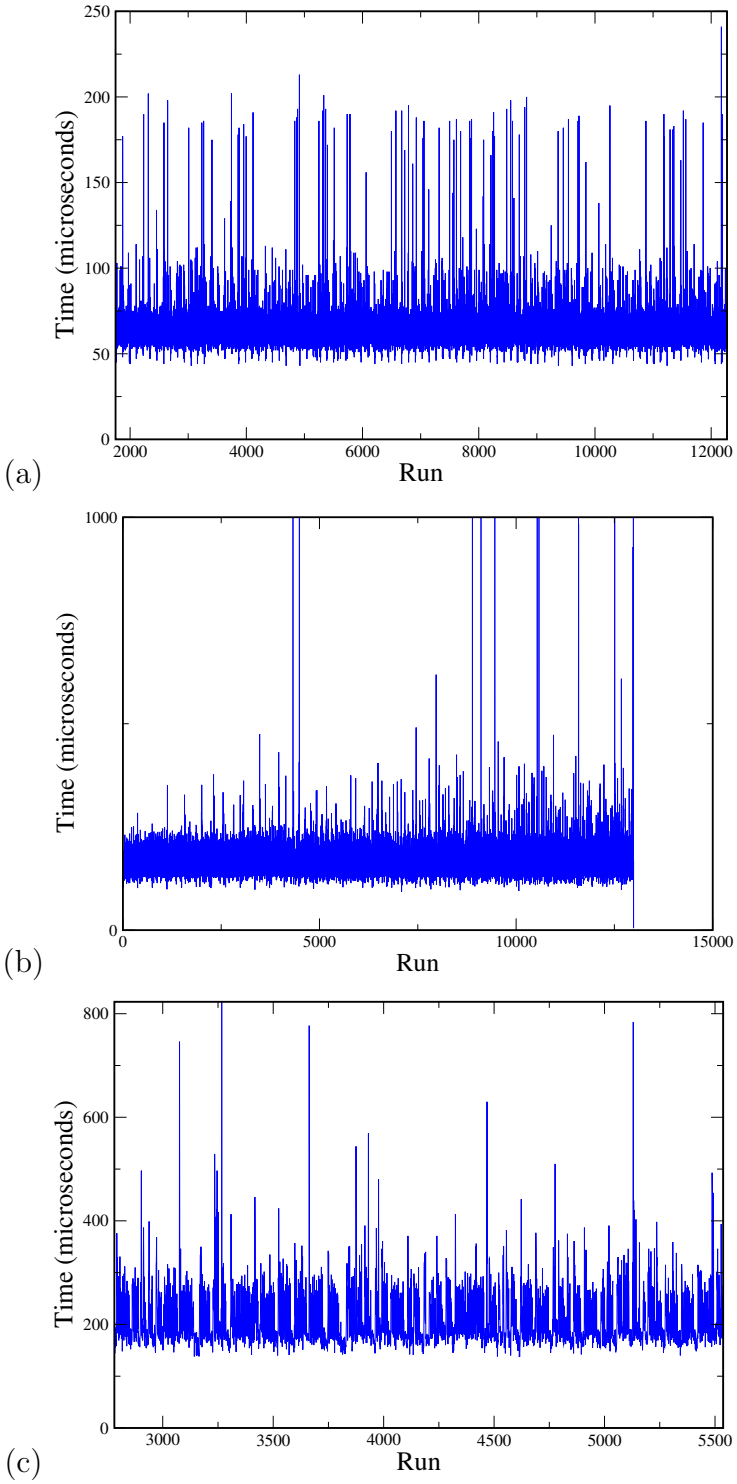

Figure C.3: *Measured values for (a) Data block, (b) Process data block, (c) each log record in Process log block.*

C.4 Send data insert from top

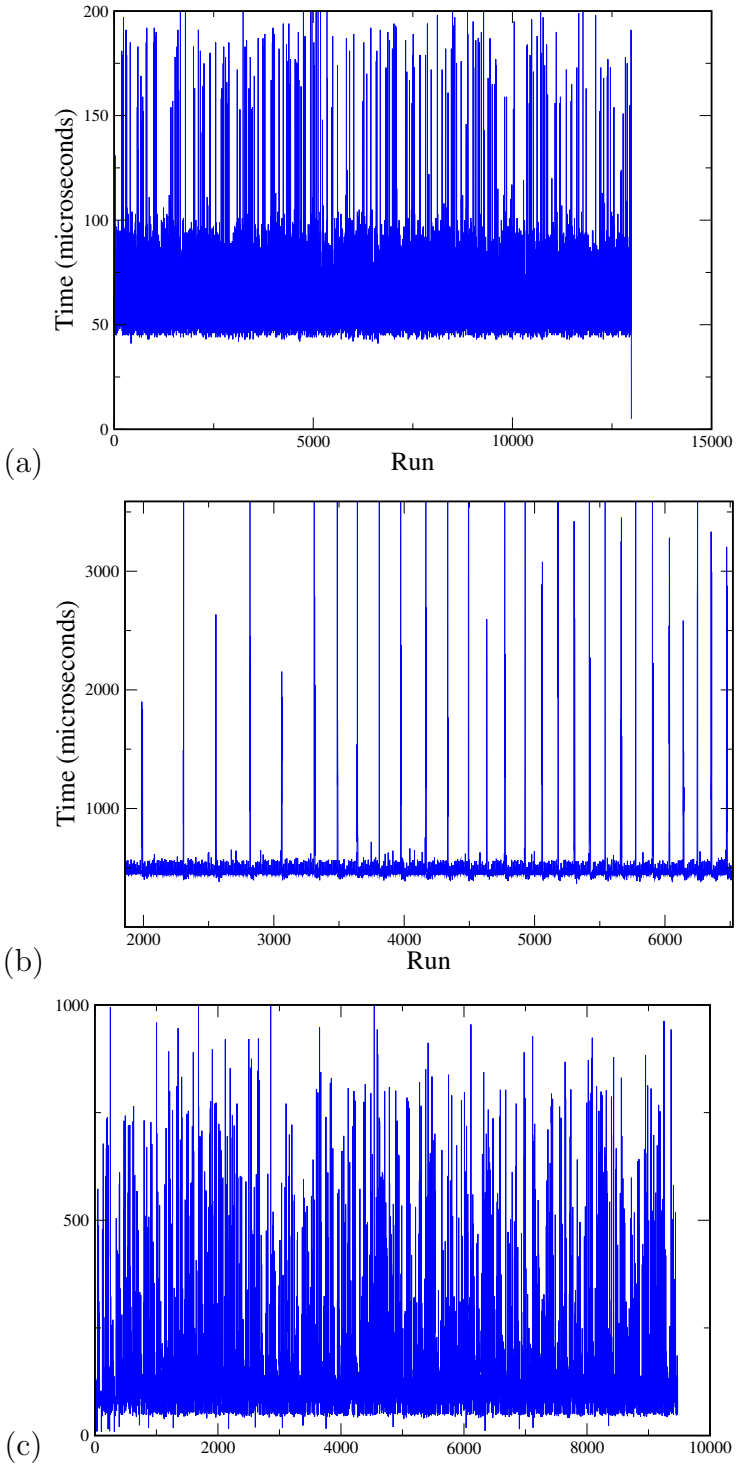

Figure C.4: *Measured values for (a) Data block, (b) Process data block, (c) each log record in Process log block.*

C.5 Pack log records

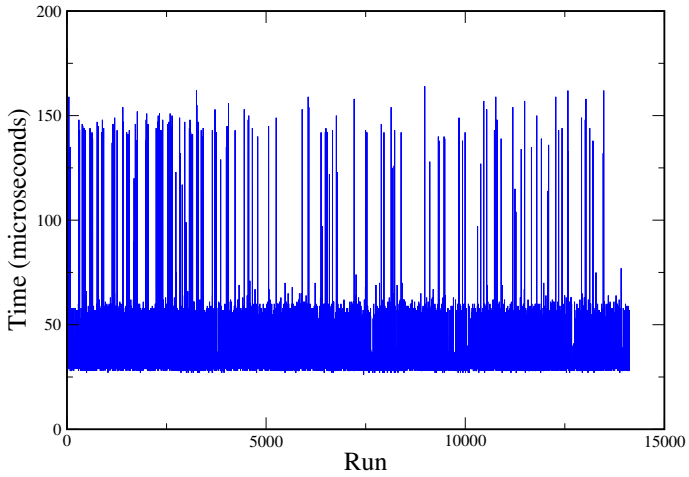

Figure C.5: *Measured values for Pack log records, used for all methods.*