Shadow Paging Is Feasible

Licentiate's Thesis

Tatu Ylönen

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelytekniikan laitos

Helsinki University of Technology
Faculty of Information Technology
Department of Computer Science

Otaniemi 1994

HELSINKI UNIVERSITY OF TECHNOLOGY		ABSTRACT OF THE LICENTIATE'S THESIS	
Author:	Tatu Ylönen		
Name of the thesis:	Shadow Paging Is Feasible		
Date:	April 27, 1994	**Number of pages:**	166
Department:	Faculty of Information Technology	**Professorship:**	Tik-76
Supervisor:	Professor Eljas Soisalon-Soininen		
Instructor:			

Many areas of the modern society have become dependent on large computer systems. Database systems are used to manage the data stored in the computers. Their central functions include concurrent access to the data, enforcing security and integrity, and physically organizing data so that it can both be used efficiently and maintained reliably in the presence of software, hardware, and other failures.

Transactions are the foundation of database systems. A transaction is an atomic unit of work. That is, the work of a transaction is either performed entirely, or not at all. Transactions are the basis for all work on fault tolerance and concurrency control in databases.

Shadow paging is one method for implementing fault tolerance and concurrent access. However, it has been considered unsuitable for large multi-user systems. The problems have included inability to do fine-granularity locking, B-tree locking, two-phase commit, incremental dumping, media recovery, clustering, and most of all, bad performance.

This thesis presents solutions the abovementioned problems, removing the primary obstacles in the use of shadow paging. It is also shown that shadow paging can be used to take transaction-consistent snapshots very efficiently, which may permit certain applications that have not been possible with existing database systems.

Keywords: databases, crash recovery, concurrency control, shadow paging, fine-granularity locking, B-tree management, snapshots, on-the-fly incremental dumping, clustering, media recovery, two-phase commit

TEKNILLINEN KORKEAKOULU		LISENSIAATINTYÖN TIIVISTELMÄ	
Tekijä:	Tatu Ylönen		
Työn nimi:	Varjosivutus on käyttökelpoinen		
Päivämäärä:	27. huhtikuuta 1994	**Sivuja:**	166
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-76
Työn valvoja:	Professori Eljas Soisalon-Soininen		
Työn ohjaaja:			

Monet yhteiskunnan alat ovat tulleet riippuvaisiksi laajoista atk-järjestelmistä. Tietokantoja käytetään järjestelmissä olevan tiedon tallennukseen, sen eheyden ja turvallisuuden takaamiseen, ja tiedon organisointiin fyysisesti niin, että sitä voidaan käsitellä tehokkasti, ja toisaalta sen säilyminen voidaan taata myös vikatilanteissa.

Tapahtuma on tietokannan käytön perusyksikkö. Tapahtuma on yhtenä kokonaisuutena tehtävä operaatio, eli se tehdään aina kokonaan tai ei lainkaan. Tapahtumien eheyden takaaminen on perustana tietokantojen vikasietoisuudelle ja yhtäaikaiselle käytölle.

Varjosivutus on yksi tapa vikasietoisuuden ja rinnakkaisen käytön toteuttamiseen. Sitä on kuitenkin pidetty huonona useista syistä. Sillä ei ole pystytty toteuttamaan tietuetason lukitusta, tehokasta hakemistojen käsittelyä, hajautettuja tietokantoja, tehokasta varmuuskopiointia, vikasietoisuutta laitevikojen osalta eikä samalla kertaa käytetyn tiedon pitämista yhdessä paikassa. Tärkein syy on ollut se, että varjosivutuksen suorituskyky on ollut paljon muita vaihtoehtoja huonompi.

Tässä työssä esitetään ratkaisut mainittuihin ongelmiin, jolloin varjosivutuksesta tulee varteenotettava vaihtoehto tietokantojen vikasietoisuuden ja yhtäaikaisen käytön toteutuksessa. Lisäksi varjosivutuksella on mahdollista ottaa näennäinen kopio tietokannasta erittäin tehokkaasti, mikä saattaa mahdollistaa sellaisia sovelluksia, jotka eivät ole nykyisillä tietokannoilla olleet lainkaan mahdollisia.

Avainsanat: tietokannat, vikasietoisuus, rinnakkaisuus, varjosivutus, tietuetason lukitus, hakemistojen lukitus, näennäiskopiot, varmuuskopiointi, ryvästys, hajautetut tietokannat

Acknowledgements

First and foremost, I thank my professor, Eljas Soisalon-Soininen, for his encouragement and support. His long-term experience with databases has provided me with many important insights, references, and ideas, and he had an important role in securing the funding for this project.

I thank all those who have participated in building the Shadows database system prototype: Johannes Helander, Sampo Kellomäki, Tero Kivinen, Kenneth Oksanen, and Heikki Suonsivu. This work would not have been possible without their work, ideas, and discussions. I particularly want to acknowledge the contribution of Johannes Helander and Tero Kivinen to media recovery, the contribution Tero Kivinen, Heikki Suonsivu, and Tomi Männistö to various details of snapshot implementation, the contribution of Tero Kivinen to some of the semaphore locking protocols with fine-granularity locking, and the contribution of Eljas Soisalon-Soininen to my understanding of early releasing of locks and its relation to strict two-phase locking.

Heikki Suonsivu and I built the first shadow paging prototype using some of these ideas in Spring 1992 [109]. This prototype eventually led to the Shadows project and this thesis. I thank Heikki for the the work we did together.

The SSP/SDB project carried out by Tero Kivinen and Heikki Suonsivu for New Generation Software (NGS) Oy, Inc. in late 1992 and early 1993 also contributed to the development of these ideas. Heikki did the first implementation of snapshots during that project, and Tero's work made me understand what was needed to implement recovery for indexes.

Kenneth Oksanen has thrown many intriguing questions and ideas at me. Some of these may turn out to be useful extensions and optimizations of the ideas presented in this work. He has also helped me with mathematical analyses of several aspects of the system, and has taken over much of the administrative work of the Shadows project from me. For all this I am deeply indebted.

The work on fine-granularity locking with B-trees was highly influenced by experience gained during my Master's project. I thank Eljas Soisalon-Soininen who supervised that work and introduced me to the research on the field.

I thank Per-Åke Larson for his patience, comments, and suggestions on these ideas and this thesis. I thank Jim Gray for his encouraging comments and insightful criticism in the early phases of the project. I also thank Antoni Wolski, Jari Veijalainen, Jarmo Ruuth, and Heikki Tuuri for their comments and questions.

I thank the members of the Steering Committee of the Shadows project, Kari Hiekkanen (NGS), Maaret Karttunen (NTC), Heikki Saikkonen (HUT), Matti Sihto (TEKES), Eljas Soisalon-Soininen (HUT), Artturi Tarjanne (Solid), and Matti Tikkanen (NTC), for their support and trust in this work.

This research was funded by TEKES and the Academy of Finland. Sun Microsystems donated a SPARCstation 10/402 for the project. Nokia Telecommunications, New Generation Software (NGS) Oy, and Solid Information Systems each gave support for the project. I thank all of the abovementioned organizations for their support.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Database Management

A database management system (DBMS) is a set of computer programs used to manage (access, update, and organize) a collection of data [46]. The collection of data is called the *database*. The primary goal of a DBMS is to provide an environment that is both convenient and efficient to use in retrieving information from and storing information into the database.

Database systems are designed to manage large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition, a database system must provide for the safety of the information stored in the database, despite system crashes or attempts at unauthorized access. The system must also coordinate concurrent use of the data by multiple users to avoid anomalous results. Figure 1.1 gives an overview of the setting.

1.1 The Need for General-Purpose Database Management Systems

General-purpose database systems were designed to answer to a number of compelling application demands:

- **Data redundancy and inconsistency**. Since files and programs used to store and access data are created over a long period of time by a large number of programmers, the same piece of information may get duplicated in several places. This redundancy results in higher storage

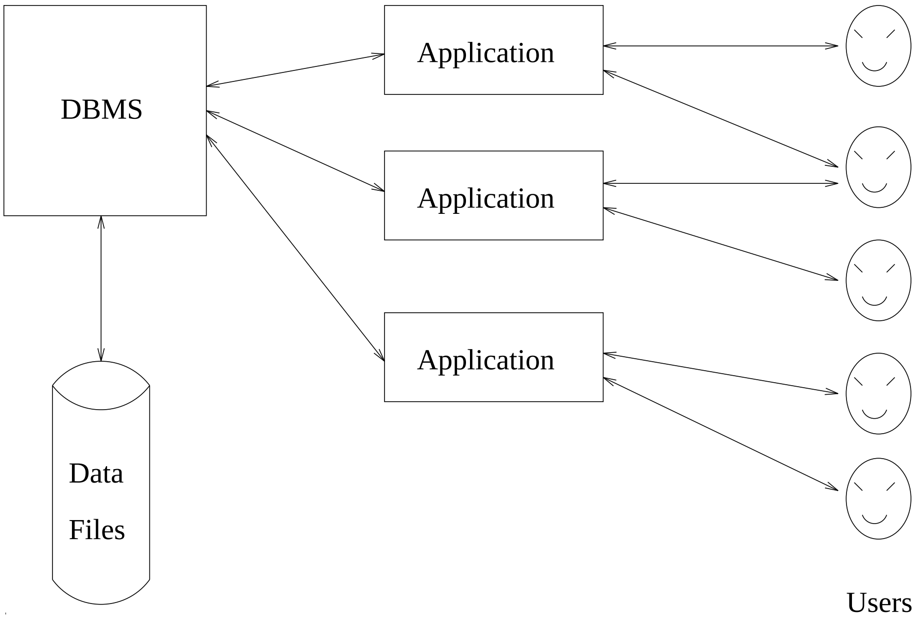

Figure 1.1: A database management system.

and access cost as well as potential for data inconsistency (that is, the various copies may no longer agree).

- **Difficulty in accessing data.** Sometimes there is a need to process unusual queries for which it is not possible to prepare application programs in advance. A general mechanism for formulating such queries is needed. This is not practical if each piece of data is managed by a separate set of programs, each with its own representation and access methods for the data.

- **Data isolation.** Since data is scattered in a number of files, and files may be in different formats, it is difficult to write new application programs to perform queries which require combining information from several files.

- **Multiple users.** Most large systems must support many, often hundreds or thousands, concurrent users with a response time of at most a few seconds. Since some programs need to update several pieces of information, inconsistencies and anomalies can result if the order of updates is not carefully controlled.

- **Security**. Not every user of a system is allowed to see all of the data.
- **Data integrity**. The data values stored in the database must satisfy certain consistency constraints. The constraints often change over the lifetime of the system. It would be very difficult to change the constraints if the change would have to be propagated to every application program in the system.
- **Maintenance**. New application programs are written and old ones are modified all the time by a large number of programmers. Many of these programs need to access and modify a common repository of data. If each program was to access the data in its own way, many programmers would write code to perform very similar functions, and most likely some of the implementations would eventually be incompatible with the others. A single sufficiently general database management system is needed to keep the system maintainable and to help eliminate duplicate work.

These reasons, among others, have lead to the development of general-purpose database management systems. These systems provide a suitable abstraction for all data stored in the system, and provide a set of facilities that allow performing all of the required operations on the data in a consistent manner.

1.2 Data Abstraction

There are several types of general-purpose database systems for different application areas. Different applications have different needs, and the systems differ in the abstractions and facilities that they provide.

The *relational data model* is the most widely used abstraction of data. All data and relationships among data are represented by a collection of tables, each of which has a number of columns with unique names. There are many commercial database systems based on the relational model, including Oracle, DB2, RDB, Informix, Ingres, Sybase, and many others.

The *network data model* represents data by records (in the Pascal or PL/1 sense), and relationships by links between records. This model is used in some older database management systems.

The *hierarchial data model* resembles the network model, but the relationships among data are limited to trees (hierarchies) instead of arbitrary graphs. The best known commercial system using this model is IMS.

Object-oriented data models are used in many unconventional database applications, such as engineering databases, CAD, and modelling [7]. The database contains a number of objects and their relationships. There are several different object oriented data models; for more information see [13].

An important issue in data abstraction is the separation of the *logical data* contained in the database from the *physical representation* used by the database. The physical representation is not visible to application programs, and is only known to the database management system. Only the logical contents of the database are visible to applications.

The logical structure of the database is described by a *database schema*. The schema describes the structure of the data in the database as it is seen by applications. Additional information may be given to allow the database management system to optimize the physical representation of the data for the expected types of queries.)

Additionally, many database systems support *views*, which is an abstraction of the schema. A view looks like a logical database schema to an application, and maps all application requests to the actual database schema. Typically views are used to provide access to a subset of the entire database; however, arbitrary computations are allowed in creating a view.

1.3 Requirements for a Database Management System

A general-purpose database system must satisfy a large number of application requirements.

- **Access to the data**. The system must provide sufficient query facilities so that the required data can be located and retrieved efficiently.

- **Modification of the data**. The system must provide sufficient facilities to update existing data, to add new data, and to remove old data.

- **Concurrency control**. The system must coordinate the activities of multiple concurrent users and ensure that concurrent use does not result in conconsistencies or anomalies.

- **Fault recovery**. All computer systems (like any mechanical systems) are subject to failure. There are a variety of causes of such failure, including disk crashes, power failures, component failures, earthquakes, and acts of war. The database system must protect the data against failures. This will be discussed in more detail in Section 2.

- **Security enforcement**. The system must provide facilities for controlling which users can access and modify which data.
- **Integrity enforcement**. It must be possible to specify integrity constraints for data stored in the database, and the database management system must enforce the constraints.
- **Physical storage of data**. The database management system interacts with the operating system to control the physical devices and files used for storing the data.

1.4 Transactions and Atomicity

A *transaction* is an abstraction which provides an easy-to-use basis for concurrency control and recovery. A transaction has the following properties.

- **Atomicity**. A transaction is either executed entirely or not at all. There is no possibility of the results of a partially executed transaction ever being visible in the database.
- **Consistency**. Assuming the database satisfies all integrity constraints before a transaction is executed, it will satisfy them after the transaction has executed (or otherwise the transaction will be rolled back and the effect is as if it had never been started).
- **Isolation**. Two transactions running in parallel have the illusion that there is no concurrency. It appears that the system runs one transaction at a time.
- **Durability**. Once the database system has reported to an application that a transaction has been executed, there is no way that the results of the transaction could disappear from the database except by executing another transaction that explicitly undoes the effects of the earlier transaction.

Chapter 2

Fault-Tolerant Computing

Reliable systems have become extremely important in the modern society as more and more functions are automated. Banks, credit card companies, insurance companies, warehouses, nuclear power plants, aerospace applications, military applications, industrial applications, vehicle control, and life support equipment are just some examples.

Many systems are critical for normal operation of the society. For example, the entire economy would very quickly collapse if bank and credit card computer systems ceased to operate. In other applications, such as airplane control, nuclear weapons systems, industrial control, and hospital equipment, failure of the embedded computer system may lead to immediate loss of human life and large scale environmental catastrophes.

Large systems are designed to be realiable and fault tolerant. Some component of the system will inevitably eventually fail. People go crazy, computers malfunction, turbines break, or an earthquake destroys the factory. Large systems are designed to tolerate component failures, and minimize the effects of such failures.

Reliability and safety must be considered at all levels of system design. Redundancy, failure detection, damage control, and repair are some of the general methods used to achieve these goals.

Most large systems contain embedded computers, often hundreds or thousands of them. Computers have central roles in data storage, communications, and control. The computer system is often the most critical part of a large system, causing gross unavailability of service if it fails. Building reliable computer systems has become extremely important.

This section will discuss the basic terminology of fault-tolerant computing, introduce some of the basic methodology, and relate the field of databases to building reliable computer systems in general.

2.1 Failures, Reliability, Failfast, Masking, Modularity

A system can be viewed as a module. Modules are composed of submodules, which in turn are composed of submodules. Each module has an ideal *specified behavior* and an *observed behavior* [33, p. 98]. A *failure* occurs when the observed behavior deviates from the specified behavior. A failure occurs because of an *error*, or a defect, in the module. The cause of the error is a *fault*. The time between the occurrence of the error and the resulting failure is the *error latency*. When the error causes a failure, it becomes *effective*; before that, the error is *latent*.

For example, a programmer's mistake is a fault. It creates a latent error in the software. When the erroneous code is executed with certain data values, it causes a failure and the error becomes effective. As a second example, a cosmic ray (fault) may discharge a memory cell, causing a memory error. When the memory is read, it produces the wrong answer (memory failure), and the error becomes effective.

The observed module behavior alternates between *service accomplishment*, when the module acts as specified, and *service interruption*, when the behavior of the module deviates from the specified behavior.

Module reliability measures the time from an initial instant to the next failure event. Reliability is statistically quantified as *mean-time-to-failure* ($MTTF$); service interruption is statistically quantified as *mean-time-to-repair* ($MTTR$).

Module availability measures the ratio of service accomplishment to elapsed time. Module unavailability is statistically quantified as

$$\frac{MTTR}{MTTF + MTTR}.$$

Module reliability can be improved by reducing failures, and failures can be avoided by *valid construction* and *error correction*. Validation can remove errors during the construction process, thus ensuring that the constructed module conforms to the specified module. Since physical components fail during operation, validation alone cannot ensure high reliability or high availability.

Error correction reduces failures by using redundancy to tolerate faults. *Latent error processing* tries to detect and repair latent errors before they become effective. Preventive maintenance is an example of latent error processing. *Effective error processing* tries to correct the error after it becomes effective.

Effective error processing can either *mask* the error or *recover* from the error. Masking uses redundant information to deliver the correct service and to construct a correct new state. Error correcting codes (ECC) used for electronic, magnetic, and optical storage are examples of masking. Error recovery denies the requested service and sets the module to an error-free state.

Faults can be *hard* or *soft*. A module with a hard fault will not function properly until it is *repaired*. A module with a soft fault appears to be repaired after the failure, and will not exhibit increased probability of failing again after the failure. Soft faults are also known as *transient* or *intermittent* faults. For example, timing faults (that is, failing to respond within the specified time constraints) are often soft.

A module is *failfast* if it stops execution when it detects a fault (*failstop* is sometimes used to mean the same thing). Failfast behavior is important because latent errors after a fault can cause the system to fail again later. Failing fast minimizes latent errors.

Modules are built *recursively*. That is, the system is a module composed of modules, which in turn are composed of modules, and so on down to the elementary particles. The goal is to start with ordinary hardware, organize it into failfast hardware and software modules, and build up a system that has no faults and, accordingly, is highly available. This goal can be approached with the controlled use of redundancy and with techniques that allow a module to mask the failures of its component modules.

2.2 Faults and Outages in Practice

In the early days of computing computers used to fail every day. However, nowadays it is common for systems (workstations, disks, processors) to have mean-time-to-failure ($MTTF$) ratings of 100 000 hours (about ten years) or more. Systems consisting of hundreds or thousands of such components can offer MTTF of one month if nothing special is done, or 100 years if great care is taken.

Empirical studies on the causes of service outages indicate that very few outages are caused by hardware faults [33, p. 100]. Fault tolerance masks most hardware faults. If a fault-tolerant system fails due to a hardware error, there is probably also a software error (the software should have masked the hardware fault), or an operator error (the operator did not initiate repair), or a maintenance error (all the standby spares were broken and had not been repaired), or an environmental failure (the machine room was on fire).

Service outages can be traced to a few broad categories of causes:

- **Environment**. Facilities failures such as the machine room, cooling, external power, communication lines, weather, earthquakes, fires, floods, acts of war, and sabotage.
- **Operations**. All the procedures and activities of system administration, configuration, and system operation.
- **Maintenance**. All the procedures and activities performed to maintain and repair the hardware and facilities. This does not include software maintenance.
- **Hardware**. The physical devices, exclusive of environmental support.
- **Software**. All the programs in the system.
- **Something else**. Examples include labor disputes (strikes) and shutdowns due to administrative decisions (e.g. stock exchange shutdown at panic).

Empirical studies indicate that a large fraction of outages is caused by reasons other than hardware or software [33, p. 102]. This emphasizes the importance of considering all aspects of availability when designing fault-tolerant systems.

Hardware designers have developed simple and effective ways of making arbitrarily reliable hardware, and software designers have developed ways to mask most of the remaining hardware faults. Due to the hardware getting very cheap, these techniques are becoming standard.

There is a clear trend toward using software to mask hardware, environmental, operations, and maintenance faults. As all the other faults are masked, the software remains. Additionally, millions of new lines of code are being added as new features are added and more functions are automated.

Studies indicate that production programs engineered using the best techniques available (structured programming, walk-throughs, careful code inspections, extensive quality assurance, alpha and beta testing) have two or three bugs per thousand lines of code. Using this rule of thumb, a few million lines of code will have several thousand bugs.

Writing better software is possible but extremely expensive. For example, the space shuttle software costs USD 5 000 per line of code, and it still contains gross errors [33, p. 116]. In practice no-one has enough money to build perfect programs unless someone first builds a better programmer.

Few people believe that design bugs can be totally eliminated. Good specifications, good design methodology, good tools, good management, and good designers are all essential to high-quality software, but even after these improvements there will still be a residue of problems.

The relative proportion of software faults has been rapidly increasing in the recent years [33, p. 104]. Techniques for dealing with software faults are getting extremely important.

- **N-version programming** is a method where the module is constructed by several independent groups of designers. All of these modules are installed in the actual system, and they take a majority vote for each answer. The hope is that the design diversity should mask many failures. However, in practice the modules tend to have related failures, this method is very expensive to implement since at least three implementations of the system must be built, and it is difficult to repair a module once it has failed.

- **Transactions** with ACID properties provide a way of writing the program as a sequence of state transitions with consistency checks. At the end of each transaction, if the consistency checks are not met, the transaction is aborted and restarted. Rerunning the transaction the next time should usually work.

2.3 Transactions and Software Fault Tolerance

The theory behind using transactions for software fault tolerance is that most software failures are *Heisenbugs*, that is, transient software errors that only appear occasionally and are related to timing or overload. Heisenbugs are contrasted to *Bohrbugs* which have deterministic behavior and occur every time the program is executed with the given data [33, p. 117]

Studies indicate the ratio of hard software faults (Bohrbugs) to soft faults

(Heisenbugs) is of the order of 1:100. Masking all transient faults thus improves MTTF by a factor 100.

Many existing large software systems have data structure repair programs that traverse data structures, looking for inconsistencies, and heuristically repair any inconsistencies they find. In effect, these programs try to mask latent errors left behind by some Heisenbug. Even though this may sound as a very questionable approach, sometimes leading to new problems due to erroneous corrections, this has been reported to improve system mean times to failure by an order of magnitude.

The alternative approach of restarting the system at the first sign of trouble makes things much worse: the system will be crashing all the time. Transactions make a desired compromise: with their ACID properties, they are perfect for masking the effects of Heisenbugs.

- **Atomicity**. The effects of individual transactions can be discarded by rolling back the transaction, providing a fine granularity of failure. There is no need to restart the entire system.

- **Consistency**. Since the previous system state was a result of executing some transaction it is known to be a consistent system state satisfying all state invariants. If a Heisenbug causes the transaction to end up in an inconsistent state, the system will automatically abort the transaction, repairing the effects of the Heisenbug.

- **Isolation**. Each program is isolated from the concurrent activity of others and, consequently, from the failure of others.

- **Durability**. No committed work is lost by recovering from later faults.

Heisenbugs are thus detected and repaired automatically without disturbing the rest of the system. The failed operation can be restarted, and with all likelihood it will succeed on the second try. Even if a transaction contains a Bohrbug, the correct distributed system state will be reconstructured by the transaction undo, and only that operation will fail.

The programming style of failfast software designs is called *defensive programming*. Every software module should check all its inputs and raise an exception if the inputs are incorrect. The result should be checked after every subroutine call. When a module detects an error, it can either correct the problem, or return an error to the higher level, eventually causing a transaction abort or some other coarse form of recovery.

Failfast programming is very useful when using *process pairs* or *system pairs*, where the other process or system acts as a hot standby and takes over should the primary process or system fail.

If Heisenbugs are the dominant form of software faults, then failfast plus transactions plus system pairs result in software fault tolerance [33, p. 119]. Geographically remote system pairs tolerate not just Heisenbugs, but environmental faults, operator faults, maintenance faults, and hardware faults as well.

2.4 Real, Unprotected, and Protected Actions

It is useful to distinguish between three kinds of actions that a system may perform [33, p. 163].

- **Real actions**. These actions affect the real, physical world in a way that may be hard or impossible to reverse. Drilling a hole is one example; firing a missile is another.

- **Unprotected actions**. These actions lack all of the ACID properties except consistency (and even that assuming the software controlling the action is correct). Unprotected actions are not atomic, and their effects cannot be depended upon, nor can several unprotected actions be executed safely in parallel on the grounds that they would execute correctly when run separately. Almost anything can fail in an unprotected action. An example is a single disk write.

- **Protected actions**. These actions do not externalize their results before they are completely done. Their updates are controlled, and are rolled back if anything goes wrong before normal end, and once they have reached their normal end, there will be no unilateral rollback. Protected actions have the ACID properties.

  Protected actions can be built from unprotected actions by carefully controlling the order of execution and adding new actions to implement the ACID properties.

In general it is very difficult to turn unprotected actions into protected actions. Assumptions and simplifications need to be made about the kinds of failures that can occur while executing an unprotected action.

Without additional assumptions an unprotected action may execute fully, partially, not at all, or it may fail catastrophically and perform arbitrary actions. Arbitrary actions (e.g., polymorphing the CPU into a nuclear bomb and exploding it) cannot be handled by any mechanisms. Instead, the actions that are used in implementing protected actions must have well-defined failure modes. This involves both assumptions about the behavior of the action in the worst case, and techniques for explicitly limiting the possible effects the action can have if it fails.

If an action is to be used in constructing a protected action, it must have well-defined semantics. All of the possible end results of executing the action must be specified (with perhaps a reservation "with a probability of once every million years it will do something else", which causes the construction of the atomic action to be invalid with a some probability). Typically this is done by specifying the correct end result of the action and the state variables which may have undefined values if the action is not completed. An important question is also whether the enclosing action can determine reliably whether the action was performed successfully.

All unprotected actions have a finite probability of failing. Since many unprotected actions can fail at the same time, it is not possible to build infinitely reliable protected actions. All protected actions have a finite probability of failing to implement the ACID properties. In practice this probability can be made smaller and smaller by adding more and more redundancy and fault tolerance to the system.

In real systems, some actions can never be made protected since real systems deal with irreversible real actions. A real system must contain some unprotected components. However, complexity and fault tolerance are much more manageable when protected actions are used.

2.5 Spheres of Control

Spheres of control [20, 33] are a framework for describing and implementing protected actions. Atomicity is achieved by enclosing a set of actions inside a sphere of control, which as a whole is executed atomically. Spheres of control can be created dynamically; that is, it is possible to enclose one or more spheres of control inside a larger sphere of control at any time to make their execution collectively atomic.

The spheres of control framework can be used to describe dependencies both before they occur (the related actions are enclosed in a sphere of control before they are executed) and after they occur. Controlling dependencies afterwards effectively involves creating a new sphere of control which encloses all affected actions. For example, suppose some action finds a problem during its processing. One can then trace back in history which action created the invalid data. One then traces forward in time to find everyone who has used data produced by the affected actions. All of these actions are enclosed in a dynamically created sphere of control, which allows committing or aborting them as a whole.

Spheres of control are primarily a theoretical framework for describing dependencies. It is not directly used in applications; because of its generality (especially tracing back in history) its implementation would be very inefficient. It is primarily a notation for expressing the dependencies of control that are relevant for the application.

2.6 Database System as Part of a Reliable Computer System

Database systems implement atomic transactions for manipulating data. This is closely related to the notion of protected actions: the individual data manipulation operations are unprotected actions, and transactions turn them into protected actions.

Many large systems (e.g. telephone exchanges) have very complicated internal data structures, have many conccurrent users, are typically distributed, and must be highly reliable. The use of a well-defined database system for managing the shared data in such a system makes the complexity of the system more manageable.

Most large computer systems require reliable processing of data. For example, large banks have thousands of terminals, ATMs, and hundreds of branches. All of these access and update the collective information concurrently. If the computer system maintaining the balances of accounts becomes unavailable or looses data, the whole bank must soon stop operation. It is thus extremely important for the bank and its customers that its computer system works reliably. This is a very typical example where databases are used. The same problem can be seen on a smaller scale with warehouses, retail chains, accounting, marketing, and many other fields of business. Typically large systems have more strict requirements for availability and reliability, whereas many small systems can tolerate some amount of unavailability or data loss, but require low cost.

Transactions and database management are the key technology in building large data-intensive computer systems. On the whole, database and transaction technology has become one of immense importance in the modern society. The current role of computers in the society is, to a large extent, possible because of database technology. Most often database technology is hidden deep inside the system.

Chapter 3

Transaction Models

There are several different abstractions for transactions [33, pp. 159–235]. Different abstractions are appropriate in different applications. Some of the transaction models are widely used in applications; some are currently only ideas on paper.

3.1 Flat Transactions

A flat transaction is a collection of operations (queries, updates), which as a whole is executed with all the ACID properties.

Flat transactions are the "basic" kind of transactions. They are supported by virtually all database management systems, and in many systems they are the only kind of transactions supported. A typical example of a flat transaction is the Debit/Credit transaction.[1]

Flat transactions are well motivated by their simplicity and usability in many applications. However, there are applications for which flat transactions are not sufficient, and attempts have been made for extending the transaction concept. Such extensions often result in weakening some of the properties of the basic transaction, and thus lead to increased complexity in writing application programs. Care must be taken in selecting the most appropriate transaction model for each application.

[1] The Debit/Credit transaction [5] changes the balance of an account, the balance of the corresponding branch, and the balance of the teller. All of these updates must be done atomically as a whole.

3.2 Flat Transactions with Savepoints

Flat transactions can be extended by permitting the application to create *savepoints* [32] and optionally roll back to a previous savepoint.

A savepoint is a saved state of the transaction (usually created by a SAVE WORK statement). The application is given a handle to the savepoint. The application is permitted to create any number of savepoints. The application can later use the ROLLBACK WORK statement to restore a previous saved state (also invalidating all savepoints that were made after the savepoint that was rolled back to).

Savepoints are useful in implementing SQL statement-level atomicity, and in applications where it is desirable to roll back only part of the transaction when an operation fails.

3.3 Distributed Transactions

A transaction must often update data on several nodes on a network (Figure 3.1). Even if the transaction, from the application's point of view, is a simple flat transaction, in a distributed database the data may be located on several nodes, causing the system to implement the simple transaction as a distributed transaction.

A distributed transaction must guarantee global consistency. This means that the transaction must be locally consistent on each node, and additionally must be globally serializable.

A distributed transaction commits atomically as a whole. Either the subtransactions on different hosts all commit, or none of them commits.

Two-phase commit [33, pp. 562–573] is usually used for implementing distributed transactions. The idea is that when the global transaction wants to commit, it asks all of its subtransactions to *prepare* for commitment. Each node then saves the state of the subtransaction in stable storage so that it can be restored even after a crash. When all nodes have successfully prepared for commitment, the coordinator decides to commit, records the decision in stable storage, and asks all participating nodes to execute the commitment. The transaction is aborted if any node cannot successfully prepare for the commit.

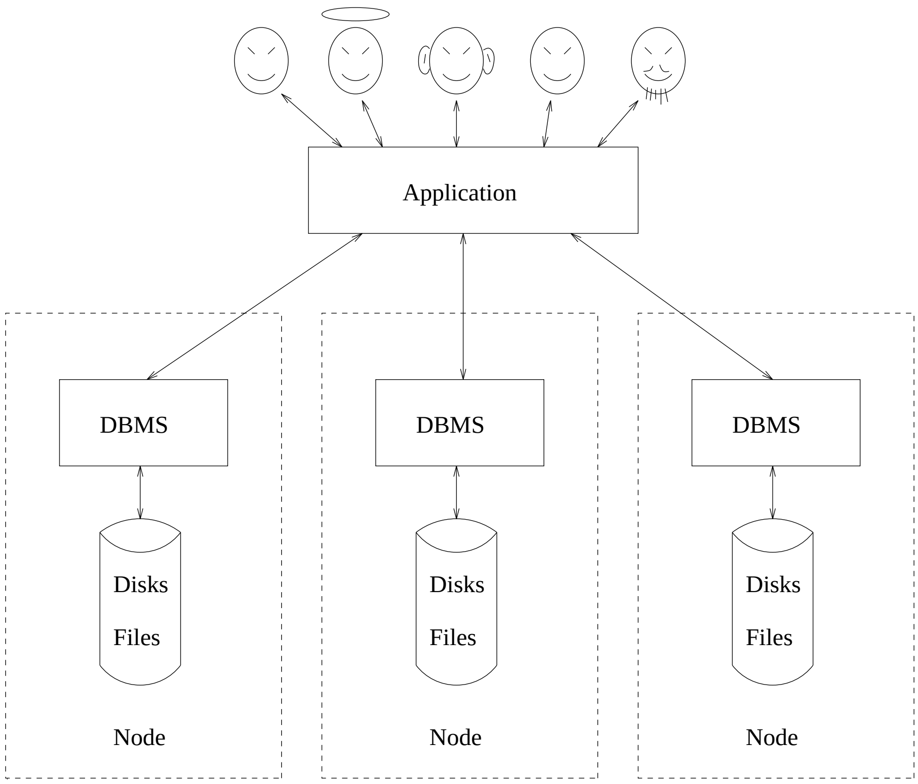

Figure 3.1: A distributed database management system.

3.4 Nested Transactions

A nested transaction [74] is a tree of transactions, the subtrees of which are either nested or flat transactions. Transactions at the leaf level are flat transactions. The distance from the root to the leaves can be different for different parts of the tree. The transaction at the root of the tree is called the *top-level transaction*; the others are called subtransactions. A transaction's predecessor in the tree is called a *parent*; a subtransaction at the next lower level is called a *child*.

A subtransaction can either commit or roll back. Its commit will not take effect, though, unless the parent transaction commits. Any subtransaction can finally commit only if the top-level transaction commits.

The rollback of a transaction anywhere in the tree causes all its subtransactions to roll back. For this reason, subtransactions have only A, C, and I, but not D of ACID.

All changes made by a subtransaction become visible to the parent transaction upon the subtransaction's commit. Objects held by a parent transaction can be accessible to its subtransactions. Changes made by a subtransaction are not visible to its siblings, in case they execute concurrently.

In the original nested transaction model [74] only the leaf-level transactions could do actual work (access the database or send messages). Higher-level transactions only organize the control flow and determine when to invoke each subtransaction.

In a nested transaction, all locks acquired by a subtransaction are *counter-inherited* by (transferred to) the parent transaction when the subtransaction commits. In other words, the objects that have been locked by the subtransaction are kept locked after its commit, but the parent transaction is made the owner of the locks. Conversely, a parent transaction can give locks on objects to a subtransaction (locks are *inherited* by the subtransaction) at the moment the subtransaction starts. This can be done selectively, or all of the parent's locks can be given to the subtransaction. If a subtransaction aborts, all locks that were newly acquired by that subtransaction are dropped, but those that were given to it by the parent are returned to the parent.

Nested transactions are currently supported by relatively few commercial systems. However, internally many systems use similar mechanisms to implement things like the atomicity of SQL update statements. Recently, some systems (most notably the Camelot system [27] and its commercial version Transarc Encina) have used nested transactions for implementing transactional programming languages.

Nested transactions are often used in connection with distributed systems and distributed transactions.

3.5 Other Transaction Models

3.5.1 Chained Transactions

The idea of chained transactions is that rather than taking (volatile) savepoints, the application program commits what it has done so far, thereby waiving its rights to do a rollback; at the same time, however, it is required to stay inside the transaction. In particular, it wants to keep locks on database objects and to keep its SQL cursors open and in their existing positions.

It is possible to use chained transactions to commit one transaction, release all database objects that are no longer needed, and pass on the processing context that is still required to the next transaction that is implicitly started.

3.5.2 Multi-Level Transactions

Multi-level transactions [61] are like nested transactions, but some subtransactions are allowed to permanently commit their changes to the database (also called *pre-commit*). This gives up the possibility of unilateral rollback of the updates. However, *compensating transactions* are used to semantically reverse what the committed subtransaction has done in case the parent decides (or is forced) to roll back. Since these commit-compensation dependencies are enforced at all levels of nesting, it is guaranteed that all updates can be revoked, even if the top-level transaction fails and a whole number of subtransactions have committed before.

Multi-level transactions are very useful because they allow committing (and releasing) data before the top-level transaction has executed. However, they assume the existence of compensating transactions. This places restrictions on the organization of subtransactions [33, pp. 203–210]. Multi-level transactions provide all the ACID properties for the top-level transaction.

3.5.3 Long-Lived Transactions

All of the transaction models presented so far are problematic for very large update transactions (for example, updating all of a bank's million accounts at the same time) [33]. If the entire operation is performed as a flat transaction, very much work will be lost if the transaction aborts.

One approach for long-lived transactions is splitting the operation into a number of *mini-batches*, each of which executes a part of the whole operation and saves information about what has been done so far (as well as any other relevant context information) in the database. If the system crashes, it is possible to pick up the context from the database and continue the operation.

3.5.4 Sagas

Sagas [29] are an attempt towards providing system support for the execution of mini-batch-like sequences of transactions. Sagas are an extension of chained transactions: each subtransaction has a compensating transaction, making the entire chain atomic, and the term saga refers to the entire chain.

3.5.5 Cooperating Transactions

The idea of cooperating transactions [75] is that a transaction may request an object from some other transaction (knowing that it is only preliminary), use the object for reading, and then return the object to the original transaction when needed. Cooperative transactions have been suggested for applications like computer-aided design (CAD).

Chapter 4

Concurrency Control

The purpose of concurrency control in a database system is to coordinate the activities of multiple concurrent users [12, 33, 46]. It provides the isolation property of transactions (and has an important role in implementing the other properties).

4.1 Serializability

A set of transactions is said to be executed *serially* if they are executed one transaction at a time in any order [12, 33]. However, normally many transactions are executed in parallel, and each transaction consists of several operations.

The execution of a set of transactions is said to be *serializable* if the result of the execution (the state of the database, as seen by the transactions themselves and by future transactions which may come later) is equal to some serial execution of the transactions.

Serializability is equivalent to the isolation property of transactions, and it is used as the correctness criterion for concurrency control. All valid concurrency control methods must ensure that only serializable executions are possible.

There are three anomalies which can appear if the concurrency control method does not guarantee serializability.

- A *lost update* happens when two transactions read a value, then both compute a new value, and write the value. One of the updates is not reflected in the final value.

- A *dirty read* happens when a transaction reads an object previously written by another transaction, and then the other transaction makes further changes to the object (either by explicit changes or by rolling back). The version read by the transaction may be inconsistent, because it is not the final (committed) value of the object.

- *Unrepeatable read* is when a transaction reads an object twice, without changing it in the meanwhile, and gets two different answers. This may happen if another transaction modifies the object between the reads. This anomality is also called the *phantom problem*.

It can be shown that these three forms of inconsistency are the only possible anomalies caused by concurrency. If they can be prevented, the transaction will appear to run in isolation [33, p. 381].

4.2 Recoverability

A concurrency control method must guarantee that the recovery method can undo the effects of aborting transactions without violating durability. In particular, no transaction can commit before all the data that it has read has been committed. If this requirement were not satisfied, the transaction creating the data could abort, but then it would no longer be possible to undo the effects of the aborting transaction, because they would already be part of the output of a committed transaction, and removing the effects at that point would violate durability.

A concurrency control method that satisfies this requirement is called *recoverable* [12, pp. 6–7]. All practical concurrency control methods must satisfy this requirement.

4.3 Avoiding Cascading Aborts

If a concurrency control method allows reading data that has not yet been committed, it must keep track of which transactions depend on which transactions, and prevent transactions from committing before all transactions they depend on have committed. Consequently, if a transaction is aborted, all transactions that depend on it must be aborted as well. This phenomenon is called *cascading abort*.

A concurrency control method can be recoverable and allow cascading aborts. However, in many practical systems it is desirable to avoid cascading aborts. A method which does not permit other transactions to read uncommitted data *avoids cascading aborts* [12, pp. 6–9], and there is no need to keep track of dependencies between transactions.

4.4 Strict Executions

Many recovery methods use *before images* [33, p. 162] for rolling back aborted transactions. If multiple transactions are allowed to write to the same data item before the others have either committed or aborted, it is possible that before image logging cannot be used for recovery: when the transactions all abort, the final value will be the before image of the last transaction that aborted, which may not be the original value of the data item.

These problems can be avoided by requiring that the execution of a write operation be delayed until all transactions that have previously written the same data item have either committed or aborted.

Executions that delay both reads and writes of a data item until all transactions that have written that data item have either committed or aborted are called *strict* [12, pp. 9–11].

Strict executions are always recoverable and avoid cascading aborts. Recoverability is a fundamental requirement which makes recovery possible. Avoiding cascading aborts and strictness are pragmatic requirements imposed by many recovery methods (but not all) for performance reasons. The recovery method described in this thesis purposely neither avoids cascading aborts nor is strict.

4.5 Degrees of Isolation

Full isolation is not appropriate for some practical applications due to performance reasons. Many database systems support several degrees of isolation [33, pp. 397–403].

Degree 0. The transaction will not overwrite another transaction's data if the other transaction runs at degree 1 or higher.

Degree 1. The transaction has no lost updates.

Degree 2. The transaction has no lost updates and no dirty reads.

Degree 3. The transaction has no lost updates and has repeatable reads (which also implies no dirty reads). This is "true" isolation.

The SQL standard requires all systems to support Degree 3 isolation (full serializability). However, many systems use Degree 2 isolation to get better performance. This is called *cursor stability*. Level 1 is called *browse access*. It permits the application to scan through a table without disturbing other transactions; however, it may get inconsistent results.

4.6 Two-Phase Locking

Concurrency control is usually implemented using *locks*. Locks are typically identified by a name (which may be derived from the page number, record identifier, key value, or some other property of the object being locked). Locks can be acquired in *shared mode*, which means that several transactions can hold shared locks on the same object, or in *exclusive mode*, which means that no other transaction can hold any lock on the same object. Typically a transaction must obtain a shared lock before it can read an object, and an exclusive lock before it can modify the object.

Locks must be acquired and released in a controlled manner to guarantee serializability. Which locks are acquired and released, and when, is controlled by the *locking protocol*.

The most common locking protocol is *two-phase locking*. The idea is to divide the transaction into two phases: the *growing phase* and the *shrinking phase*. New locks can be obtained only in the growing phase, and locks can be released only in the shrinking phase. Together with obtaining a shared lock before reads and an exclusive lock before updates this guarantees serializability [12].

In practical systems it is often not possible to know which objects the transaction is going to access before commit until the transaction has actually requested to commit. This means that the shrinking phase cannot begin until after the transaction has requested to commit. This variant of two-phase locking is called *strict two-phase locking* [12].

4.7 Deadlocks

If a transaction tries to obtain a lock which is held by another transaction in a conflicting mode, the transaction will have to wait until the other lock has been released.

Suppose transaction T1 requests and gets an exclusive lock on object O1. T2 then requests and gets an exclusive lock on O2. T1 then requests a lock on O2; since there is a conflicting lock, the request blocks. Then, transaction T2 requests a lock on O1; it blocks as well. Both transactions are now blocked waiting for each other. This situation is called *deadlock*.

A deadlock occurs when there is a set of transactions waiting on each other. A deadlock situation can be detected by constructing a directed graph called the *waits-for graph*. The graph has all active transactions as its vertices; there is an edge in the graph for each lock wait from the waiting transaction to the owner of the lock being waited on. There is a deadlock if this graph has a cycle.

A deadlock is typically resolved by aborting one of the transactions participating in the deadlock (that is, one of the transactions forming the cycle).

4.8 Predicate Locks

Locking is described in the previous sections as referring to individual objects. In practice things are not this simple: for example, what are the objects accessed in a query in a relational database? The transaction reads the database structure, table headers, and the individual records. Locking the table headers prevents all insertions and deletions to the table, which is clearly unacceptable in high-performance systems. If just the records are locked, other transactions can insert or delete records, and those records might show up in later queries by the same transaction as phantoms.

Locks on individual records can be used to protect against lost updates and dirty reads. However, they are inadequate for preventing phantoms. Locking table headers, on the other hand, severely limits concurrency.

There is a solution called *predicate locks* [28]. The idea is to lock an arbitrary subset of the database (as specified by a predicate). When a query accesses a set of records, it obtains a predicate lock on all the records that are relevant to the query. This predicate lock prevents phantoms.

Predicate locking requires testing for predicate satisfiability, which is NP-complete. Direct use of predicate locking is computationally not feasible. However, it is possible to devise more limited systems that offer most of the benefits of full predicate locks, yet permit efficient implementation.

4.9 Granular Locks

Phantoms raise the issue of *locking granularity* [33, p. 406–411]. Possible units of locking are databases, files, file subsets, records, fields within records, and so on. The choice of the lock granule is a trade-off between concurrency and locking overhead. *Coarse-granularity locking* is appropriate for large transactions, whereas *fine-granularity locking* is appropriate for small transactions. Additionally, some transactions need to lock subsets of all objects (possibly including objects not (yet) in the database).

Predicate locks permit locking at multiple granularities, as well as locking arbitrary subsets. The computational complexity of predicate locking can be overcome by choosing a (possibly large) fixed set of predicates, and forming a partial order where a "larger" predicate implies all "smaller" predicates. A lock on a predicate implies a lock on all smaller predicates.

There is are two special lock modes, *intention shared* and *intention exclusive*. An intention shared lock is incompatible with exclusive locks, and an intention exclusive lock is incompatible with shared and exclusive locks. The intention lock modes are compatible with each other.

All lock requests are constrained so that before a transaction can obtain a lock on a lower level predicate, it must obtain an appropriate intention lock on all larger predicates.

This locking protocol is called *granular locking* or *DAG locking*. When the predicates can be ordered in a tree (instead of an arbitrary partial order) this is called *tree locking*.

Many systems use special locking for commutative operations. These operations can be performed in an arbitrary order with the same result. Such operations are also reversible. A typical example is adding a number to the value of a field. A lock for this kind of operations is called an *update lock*, or generally *extended lock modes*.

Table 4.1 shows typical lock modes and their relationships. A transaction must obtain an intention lock (IS, IX, or SIX) on the higher level node in the locking hierarchy before it is allowed to obtain a concrete lock (S, SIX, U, or X) on a lower level node. A concrete lock in one mode on a high-level node implies the same lock mode on all lower level nodes covered by the higher-level node. Lock mode IS permits the transaction to obtain shared (IS and S) locks on lower level nodes; mode IX permits the transaction to obtain any lock modes on lower level nodes. Lock mode SIX implies a shared lock on all lower level nodes covered by the higher-level node, and permits the transaction to obtain locks on lower level nodes.

Compatibility matrix for granular locks							
	Granted mode						
Requested mode	None	IS	IX	S	SIX	Update	X
IS	+	+	+	+	+	-	-
IX	+	+	+	-	-	-	-
S	+	+	-	+	-	-	-
SIX	+	+	-	-	-	-	-
U	+	-	-	+	-	-	-
X	+	-	-	-	-	-	-

Table 4.1: Compatibility matrix for granular lock requests [33, p. 408].

The database system usually does not know how many objects a transaction is going to access when the transaction starts. Therefore, it does not have enough information to select the most appropriate locking granularity for the transaction. This is usually solved by having the transaction first use fine-granularity locking, and a coarse-granularity lock is obtained after the transaction has made a certain number of modifications. This is called *lock escalation.*

Chapter 5

Recovery

Recovery provides the atomicity and durability properties of transactions in the presence of failures. It must guarantee that every transaction is either executed entirely, or not at all. Additionally it must guarantee that once commitment has been acknowledged to the application, the transaction will remain permanent.

One should be specific about the circumstances under which a transaction is to be atomic and durable. Achieving these properties under the assumption that the system is functioning normally is fairly easy. Getting the properties in the presence of system crashes (without any hardware components being damaged) requires more effort, and the difficulty increases as more types of failures (disk crashes, fire in the machine room, etc.) are considered. In most of the following discussion, atomicity and durability are guaranteed in the presence of system crashes and the loss of at most one hardware component. Techniques for higher levels of fault tolerance (such as loss of a machine room) are discussed in Section 5.4.3. Techniques against situations where a failure goes undetected (that is, the failfast property of components fails), particularly N-plexing, are discussed in [33].

5.1 Types of Storage

Computers have several types of storage that have different recovery characteristics. *Volatile storage* (e.g., main memory) loses its contents on power-down or system restart. *Nonvolatile storage* keeps its contents during power failures and normal system restarts. However, it is still susceptible to failures. *Stable storage* is an ideal storage type which keeps its contents forever and has the ACID properties. Stable storage can be approximated by storing the data on several devices with independent failure modes, and using additional control information to synchronize updates.

Normal crash recovery deals with loss of volatile storage. Media recovery deals with loss of nonvolatile storage. In the end the entire database system can be seen as an approximation of stable storage for the data. Most systems are designed to maintain this approximation in the presence of at most one component failure. Failure to maintain the approximation of stable storage results in loss of the ACID properties.

5.2 Log-Based Recovery

Logs are the best-known technique for recovery. The general idea is that as modifications are made to the database, the modification (e.g., old value and new value of the object) is stored in the log. The log can then be used to redo the updates or to undo the updates.

There are several variations of logging [12, 35, 102].

Before-image logging (also called *undo logging*) means that the old value of a data item (along with control information) is written to the log before the value of the data item is changed. All data modified by a transaction must be written to disk before the transaction can commit, because there is no way to redo the transaction if the system crashes after commit but before the data has been written to disk.

After-image logging (or *redo logging*) means that the new values of data items are written to the log. New values are not written to data pages until the transaction has committed, and thus there is no need to undo them at recovery time.

Write-ahead logging (or *undo/redo logging*) records both the old and the new value of a data item in the log. This permits updates to be written to the database while the transaction is active, but does not require flushing changes to disk at transaction commit (only the log needs to be flushed).

Most log-based systems modify data in-place; that is, they modify the contents of existing data pages and overwrite the old contents.

Shadow paging is an example of a recovery method which does not update data in-place. Instead, a new page is allocated whenever one is modified, and existing pages are never overwritten as long as they are valid. Shadow paging is discussed in more detail in Section 6.

5.3 Write-Ahead Logging

Write-ahead logging is widely used in commercial databases [66], and has been found to provide the best overall performance [3, 4, 42, 43, 89]. It supports fine-granularity locking [66], extended lock modes [66], partial rollbacks [66], on-the-fly incremental dumping [69], B-tree locking [53, 54, 65, 67], transient versioning for efficient execution of read-only transactions [71], remote backup management [70, 72] and nested transactions [92].

The basic write-ahead logging algorithm works as follows [12, pp. 180–195]. T_i identifies the transaction, x names a data item, and v is the value to be written.

Write(T_i, x, v)

1. If the page containing x is not in the cache, fetch it.
2. Append [T_i, x, $value(x)$, v] to the log.
3. Write v into the space occupied by x.

Read(T_i, x)

1. If the page containing x is not in the cache, fetch it.
2. Return the value of x.

Commit(T_i)

1. Write a commit record for T_i to the log and flush the log.
2. Acknowledge commitment to the scheduler.

Abort(T_i)

1. For each data item x updated by T_i:
   - if the page containing x is not in the cache, fetch it;
   - copy the before image of x (as recorded in the log) into the space occupied by x.
2. Write an abort record for T_i to the log.
3. Acknowledge the abortion to the scheduler.

Recovery after a crash works by using the before images to undo any transactions for which there is no commit record, and by using the after images to redo any updates which had not been flushed to disk from the cache. Log sequence numbers (LSNs) [12, 33, 66] are used to determine which pages on disk already contain new values.

Checkpointing is used to limit how far the log needs to be scanned. The beginning and the end of a checkpoint are recorded in the log. All dirty pages are flushed to disk during the checkpoint; however, transaction activity is not quiesced. The recovery system then knows that all updates made before the beginning of the last checkpoint are already reflected in the physical database on disk. The checkpoint log records also contain a list of active transactions at that time; that information can be used to determine which transactions were still active when the system crashed without needing to scan further in the log. *Fuzzy checkpointing* can be used to reduce disk traffic during a checkpoint; the idea is to flush those pages that have not been written to disk since the beginning of the previous checkpoint, and at restart scan the log until the beginning of the second to last checkpoint.

Logical logging (or *operation logging*) is commonly used with write-ahead logging. The idea is to log the operation that was performed (for example, insert record r in relation R). Logical logging reduces log size, and is more flexible than physical logging. It allows restructuring the physical database so that data may get stored on a completely different physical page after recovery. An example of this is B-tree reorganization during insertion [67].

Record level locking (that is, locking individual records which may be smaller than a page) requires care because page-level LSNs can no longer be used to determine the state of a page if the system crashes during recovery. This can be solved using *compensation log records*, which are written to the log during the undo phase of recovery, or by including a LSN in each record [12, 66].

5.4 Media Recovery

Media recovery deals with recovering from destruction of the physical media (disk) used to store the database. Since hardware failures (disk crashes etc.) occur relatively frequently (about once a year), the system must be able to recover from such failures.

Backups are one form of media recovery. A backup alone cannot guarantee the durability of transactions, since transactions executed after the backup are not reflected in the backup.

A backup together with log entries for transactions executed after the dump can be used to reconstruct the state of the database at the time of the failure. All industrial-strength database systems must support dumping and restoring the database, dumping preferably not disturbing normal transaction processing. Very large databases cannot reasonably be dumped in full very often, and incremental dumps (that is, dumps of only that data which has changed since the previous dump) are important in such environments [36, 69, 84, 91, 94].

Redundancy can be used to tolerate failures without disturbing transaction processing. *Mirroring* [33, 37, 78] is a technique where the same data is stored on two (or more) disks, and if one disk fails, the other can be used to retrieve the data and repair the failed disk. Mirroring also improves read performance at the expense of write performance. Doubly distorted mirrors [78] provide improved write performance at the expense of disk space and special hardware.

RAID (Redundant Array of Inexpensive Disks) [37, 80, 99, 100] also permits recovery from disk failures. The idea is to store the data on just one disk, but for every few disks there is a parity disk which contains the exclusive-or of the data in the corresponding blocks on the other disks. If one disk fails, its data can be reconstructed by computing the exclusive-or of the data on the other disks. This permits recovery after a disk crash, and even normal operation at reduced performance. Different RAID implementations store the parity information differently, and have radically different performance profiles. RAID does not improve read performance and causes extra overhead for writes. However, its disk space overhead is much less than that of mirroring (about 10–20% compared to 100% for mirroring).

5.4.1 Recovering from a Bad Block

The most common kind of a disk failure is the destruction of the contents of a single disk block. Bad blocks appear fairly often (about once a year) on disks during normal usage. Data in a disk block may also get destroyed due to a software error which writes garbage to the disk (in this case the most difficult problem is detecting that the block has been corrupted).

Many systems support automatic recovery from this kind of failures [66]. The log and backup copies can be used to reconstruct the state of the block. Mirroring and RAID systems can easily recover from this kind of failure without disturbing normal operation.

In many applications it is unacceptable to crash the database system or make it unavailable for users because of the failure of a single block.

5.4.2 Recovering from a Disk Crash

The contents of an entire disk are sometimes lost due to a head crash, component failure, or some other similar problem. Typical disks have mean times to failure of a few years. Most database applications thus need to consider this possibility.

One approach to recovering from a disk crash is to restore the latest backup dump of the database, and redo transactions that have been executed after the dump from the log.

Mirroring and RAID systems can recover from this kind of a failure. A new disk is substituted for the failed one, and the system brings the new disk up to date using the intact ones. Normal operation can continue during recovery; however, not all practical systems support normal transaction processing during recovery from a disk crash ([37] discusses the problems involved). While the system is repairing the damage, it cannot sustain another disk crash; however, an additional disk crash during repair is highly unlikely if the disks have independent failure modes.

5.4.3 Recovering from Site Loss

A rather extreme catastrophe is the loss of an entire site (computer room and facilities) due to fire, earthquake, acts of war, sabotage, labor disputes, or some other cause. Basically all lesser failures can be treated as a site loss if the other methods fail.

Geographically remote system pairs with independent connections to user terminals can be used to tolerate site crashes [33]. Several algorithms exist for maintaining a hot standby at a remote site [33, 44, 70, 72, 83]. Should the primary site fail for any reason, the backup site can immediately take over and continue serving users without any interruption of service.

In the event that all of these methods fail, the only possibility is to restore the entire database from the backup copies, compromising the durability of transactions. In many low-end systems this is the only kind of media recovery supported. In high-end and high-availability systems more sophisticated methods are necessary.

5.5 Classification and Performance Issues

Different recovery methods have different properties with respect to cache management, checkpointing, and atomicity of disk updates [35].

A recovery method has *atomic* updates if any set of modified pages can be propagated as a unit, such that either all or none of the updates become part of the materialized database. Conversely, a method is not atomic if multiple updates cannot be done atomically. Most update-in-place methods are not atomic in this sense, because they can modify multiple pages, and it is possible that some of the pages have been written to disk while some have not. Shadow paging (Section 6) is an example of a method which has atomic updates.

Some recovery methods permit the cache to write dirty pages to disk at any time (strictly speaking, only when the page is not being modified by the system). This is called the *steal policy* (the cache is allowed to write the page to disk and remove it from the cache). Some methods, particularly redo-only logging, do not permit modified data to be written to disk before transaction commit.

Methods which do not permit the steal policy have problems in handling large transactions. There may not be enough cache memory in the system to hold all updates of a transaction. Thus, such systems must either limit the maximum transaction size, or implement some kind of overflow mechanisms (which are difficult to make efficient).

Undo-only logging requires that all modified data must be written to disk before the transaction can commit. This is called the *force policy* (updates are forced to disk at commit).

Systems that use the force policy must perform many more writes than systems that do not require the force policy, especially if the database contains many hotspots. Moreover, if update-in-place is used, the writes go to arbitrary fixed locations on the disk, resulting in many time-consuming seeks and rotational delays, which greatly lengthens the time required for processing a commit. Force policy in connection with shadow paging is discussed in Sections 6.5 and 17.

5.6 Clustering

Clustering [33] is an issue not directly related to recovery which can greatly affect database performance. Clustering means that data which is frequently accessed together is stored in nearby locations on disk.

Clustering is relevant in several situations. In sequential scans it is very important that the data to be read is stored contiguously on disk to avoid seeks. Large objects (*blobs*) are also usually accessed sequentially. Complex objects [15, 16] often involve accessing several objects together. Such objects should be stored on the same or on nearby pages.

Many join[1] algorithms depend on being able to efficiently read a record from one table, and then records from another table that have the same key value in some field. This is typically done using a clustering index. For this purpose it is important to be able to cluster all records with the same key value into the same physical block.

Clustering is not relevant for small random accesses to a large database. Such accesses will almost always require a full seek regardless of how the data is physically organized.

[1] Join is a relational algebra operation which merges data in two or more tables [46]. It can be seen as a selection on the cartesian product of the tables.

Chapter 6

Shadow Paging

Shadow paging is an alternative to logging in the implementation of recovery. The idea is that the data structures used for implementing the logical database only refer to *logical pages*, and a *page table* is used to map each logical page number to a *physical page number*. When a page is modified, the original physical page is not modified; instead, a new physical page is allocated, and when the transaction commits, the page table is updated atomically to reflect the new locations of modified pages.

There are several alternatives for implementing atomic page table updates. The original shadow page algorithm [60] used two page tables (the *current* and the *shadow page table*), and a bit which tells which page table is current. System R [32] used a similar approach with logs on page table pages to implement multiple concurrent transactions. Reuter [88] avoided the page table entirely by using two physical pages with timestamps for each logical page. Lampson and Sturgis [49, 64] used *intentions lists* as a redo mechanism (the list of changes is first recorded in stable storage, and only then is the page table modified). Kent [42, 43] implemented the page table as a tree-like structure, the modified parts of which were rebuilt using fresh pages, and a *page table pointer* was set to point to the new page table at commit. Each of these alternatives will be discussed in more detail in the following sections.

There have been several studies on the relative performance of shadow paging algorithms compared to log-based algorithms [3, 4, 32, 42, 43]. Shadow paging has consistently been found to have inferior performance. Additionally, shadow paging has had several other problems for which no solutions have been presented: shadow paging does not easily support fine-granularity locking [66], large read-only transactions, media recovery, two-phase commit [66] (though [49] presents an algorithm for two-phase commit), or partial rollbacks [66] (though System R had them [32]). Additionally, shadow paging destroys clustering of data [12, 32, 33], although techniques have also been presented to avoid this [43, 60, 88].

Generally, very little research has been done on shadow paging since about 1985. It is very widely believed that shadow paging will never be a usable recovery mechanism. Newer books on database systems hardly mention it. Virtually all commercial systems use logs, with the exception of some object-oriented databases, and even there the experiences have been discouraging [30].

6.1 The Original Shadow Page Algorithm

Lorie [60] presented the original shadow page algorithm. In many textbooks this algorithm is the only description of shadow paging.

The idea is that there are two page tables on disk in fixed locations. One of the page tables contains the current database state (possibly containing updates of uncommitted transactions). The other page table (called the *shadow page table*) contains an earlier transaction-consistent database state. Additionally, there is a *master record*, which tells which of the page tables describes the current database. (Lorie [60] also describes the use of two free block bitmaps, and his description includes multiple segments, each of which has a separate page table.)

Read operations work by mapping the requested page number to a physical page number using the current page table, and reading that physical page.

Write operations work by allocating a new page, copying the new contents to the new page, and changing the mapping in the current page table to point to the new page.

The original scheme [60] does not support durable commitment for transactions. Instead, a *checkpoint* is used to make permanent the changes made by transactions executed so far. A checkpoint works by quiescing transactions, writing to disk all changes made so far (including the current page table), and toggling the bit in the master record which indicates which page table is current. The old current page table is then copied to the other page table, and the old current page table becomes the shadow page table.

The original scheme does not discuss aborting individual transactions.

(It is of course possible to return to the previous checkpoint, but several transactions may have been executed after the last checkpoint.)

Recovery after a crash works by copying the shadow page table (and free block bitmap) to the current page table. No other processing is needed. The state of the database returns to that of the last checkpoint.

If durability of transactions and aborting individual transactions need to be supported, only one transaction can be performed between checkpoints. System R [32] solved this using logs together with shadowing.

6.2 Intentions Lists

Lampson and Sturgis [48, 49] described a method for implementing atomic transactions. Their idea is to store all the changes made by a transaction in an *intentions list*. The intentions list is saved in stable storage in the first phase of commit (they use two-phase commit), and executed in the second phase of commit. If the transaction is aborted, the intentions list is never executed. If the transaction crashes before the intentions list has been executed, it will be re-executed during recovery. Intentions lists can be seen as a variation of redo-only logging.

Intentions lists can be used either with shadowing or without shadowing. When used with shadowing, they contain changes to the page table mapping. Only one copy of the page table is needed, since changes are made atomic by the redo mechanism. When used without shadowing, the intentions list must contain all of the new data to be written.

Intentions lists can be used to implement multiple concurrent transactions, since locking guarantees that the intentions lists of different transactions refer to different sets of pages. Durable commitment and abortion of individual transactions are easily supported.

Menasće and Landes [64] use the intentions list method (they also discuss not having a page table and first writing the page to a different area on disk) with two-phase commit. The page table itself is shadowed (that is, existing page table pages are not modified; instead, a new page table page is allocated whenever a modification is made). However, updates to the page table on disk are done only at checkpoints [60], not for every transaction. On-the-fly dumping of the database is implemented using the *initial version method* described in [91].

Agrawal and DeWitt [4] have also used the intentions list paradigm. They use an *incremental page table*, which is similar to an intentions list, but is used as a differential to the global page table while the transaction is active.

6.3 Page Table in Shadowed Storage

Kent [42, 43] has made several improvements to the previous shadow paging schemes. First, the page table is structured as a tree, and existing pages are never modified. Instead, a new physical page is allocated for the modified page table page, and higher level page table pages are modified the same way to contain the new address of the lower level page table page. There is a *page table pointer* in a fixed location in stable storage which is used to record the current location of the highest level page table page.

Kent uses a per-transaction incremental page table to hold the changes made to the page table by a transaction. The incremental page table is basically a list of mappings $\langle L, P_{old}, P_{new} \rangle$, where L is the logical page number, P_{old} is the old mapping of the page, and P_{new} is the new mapping of the page.

Read operations first check from the incremental page table if the transaction has a local copy of the page. If available, the local copy is used. Otherwise, the page table is used to map the logical page number to a physical page, and that page number is used.

Write operations first check if the transaction already has a local copy of the page. If so, that copy is used. Otherwise, the system allocates a new physical page, copies the old contents of the logical page to that page, modifies the copy, and adds an entry for the page to its incremental page table.

Page-level two-phase locking on logical page numbers is used for concurrency control. Reads lock the page in shared mode before doing anything else, and writes lock the page in exclusive mode. Shared locks are released when the transaction requests to commit, and exclusive locks when the transaction has actually committed.

If a transaction needs to be aborted, all that has to be done is to free the new page numbers in its incremental page table and to release all locks held by the transaction.

Kent uses *commit batching* (or *group commit* [33, pp. 509–510]) to improve the performance of commits. When a transaction requests to be committed, it is put on a list of transactions waiting to be committed. A separate process takes all transactions on the list a few times a second. It merges their incremental page tables (there can be no conflicts between the transactions since only pages that are exclusively locked can have entries in the incremental page table). The changes of all those transactions are then made to the global page table, allocating new physical pages for all page table pages that are changed. All modified pages are then flushed to disk, and the address of the new page table is written to the master pointer. The old physical pages (and the old versions of the modified page table pages) are freed.

Kent's algorithm supports multiple concurrent transactions without any logging. It supports full durability and isolation for all transactions. No garbage collection is needed. Kent does not discuss things like on-the-fly dumping or media recovery; however, it will be shown in Sections 11.6.2 and 14 that both of these can be supported efficiently.

6.4 Clustering with Shadow Paging

Several alternatives have been proposed for supporting clustering with shadow paging [43, 60, 88] (strictly speaking, [88] is not a shadow paging system in the same sense as the others since it has no page table). The basic idea in all of them is to keep the logical-to-physical mapping approximately linear.

Lorie [60] has proposed that disk cylinders should be used as clusters, and that neighboring logical pages should, if at all possible, be allocated from the same cylinder. Some space should be reserved on each cylinder so that new copies of pages can be allocated on the same cylinder.

Reuter [88] proposed in his TWIST algorithm allocating two physical pages for each logical page. A timestamp is used to determine which of the physical pages contains the current value of the page. The two pages are always adjacent pages on disk, and thus it is very cheap to read both of them at the same time. No page table is needed in this scheme since the mapping from logical to physical pages is fixed.

Kent [42, 43] uses a clustering mechanism similar to that of Lorie [60]. Kent's algorithm makes the transaction wait if there no suitable physical page is available for allocation.

6.5 Performance Results

Several studies have shown that shadow paging performs much worse than the best log-based approaches, particularly write-ahead logging.

Agrawal and DeWitt [3, 4] compared write-ahead logging to shadow paging. Their shadow paging version was based on the Lorie [60] algorithm, and used intentions lists [49] to implement concurrent transactions. They assumed that the clustering algorithm [60] was being used. Logging was found to be the overall winner, especially for small update transactions, although shadow paging performed reasonably well for large transactions with a sequential access pattern. The primary overhead with shadow paging turned out to be reading and updating the shadow page table.

Kent [42, 43] compared write-ahead logging with his version of shadow paging (Section 6.3). He found logging to perform better for small update transactions, but for large transactions shadow paging sometimes performed better. The primary overhead with shadow paging was page table I/O.

Many papers mention lack of clustering as a major performance problem with shadows [32, 66]. This is strange in the light that several clustering algorithms have been presented for shadow paging (though System R [32] did not use any of them).

Force policy (that is, the fact that all modified pages must be flushed to disk at transaction commit) has been mentioned as one problem with shadow paging [30, 66]. This is a real problem, which clearly increases the time needed to commit a transaction. It is, however, unclear how much it really increases the steady-state average disk I/O [106].

Part II

The New Shadow Paging Algorithms

Chapter 7

Introduction

This part of the thesis presents the new shadow paging algorithms developed in this work. They are based on the variant of shadow paging developed by Kent[1] [43].

Sections 8 and 9 describe a novel method for supporting fine-granularity locking with shadow paging [105, 108]. Section 10 describes how to use the fine-granularity locking framework for B-trees. Section 11 describes how shadow paging can be used to take transaction-consistent snapshots of the database very efficiently [107]. Section 11.6.1 describes how to execute read-only transactions without any locking using snapshots, and Section 11.6.2 describes how to use snapshots to implement on-the-fly multi-level incremental dumping without disturbing normal transaction processing. Section 12 describes a write optimization method which significantly improves the performance of shadow paging [106]. The clustering problem is analyzed in Section 13 and a solution is presented which provides reasonable performance for most applications. Section 14 describes how to support media recovery efficiently with shadow paging. Finally, Section 15 describes how to do two-phase commit so that shadow paging can be used in distributed databases [105].

[1]Actually, many of the ideas described in [43] were reinvented independently, and only later did the earlier work become known to the author.

7.1 The Variant of Shadow Paging Used in This Work

The variant of shadow paging used in this work differs significantly from most of the earlier algorithms in the literature [32, 48, 49, 60, 88], but is similar to Kent's algorithm [42, 43] described in Section 6.3. This section describes the page-oriented version of the algorithm (Kent assumes page-level locking) and establishes some terminology.

The database consists of a number of disk blocks organized as pages. Each page can hold a fixed number of bytes, and is identified by a number from which its address on disk can be computed. These pages will be called *physical* because they have a direct representation on disk.

The levels of the database system above the transaction manager also see the database as a collection of numbered pages. These will be called *logical* pages to distinguish them from physical pages. High level data structures, such as those used to implement tables or indexes, only refer to logical pages.

The mapping between logical and physical pages is maintained using a *page table*. Conceptually it is an array of physical page numbers indexed by the logical page number. There is always a valid page table in nonvolatile storage on disk.

The page table is implemented as a tree-like structure (Figure 7.1). Leaf pages of the tree contain a fixed number of pointers to physical data pages. Each page above the leaf pages contains the same number of pointers to leaf pages, and so on. All levels of the tree are identical. When moving down the tree, the page on each level is indexed by $(L/N^{level-1}) \bmod N$, where L is the logical page number, N is the number of page table entries on each page, and *level* is the level on the page table on which the page table is, leaf level being 1, the one above that 2, and so on.

The page table is in shadowed storage (that is, existing pages are never modified, and new pages are allocated if a page needs to be modified). There is a *page table pointer* in a fixed location in stable storage. It contains the address of the root of the page table on disk. When a transaction modifies the database, it constructs a new page table (without modifying any of the existing page table pages), writes the new page table to disk, and commits by atomically writing the address of the new page table to the page table pointer. The new page table is partially shared with the old page table; only those pages which are modified are rewritten.

Each page table entry contains two physical page numbers (Section 14

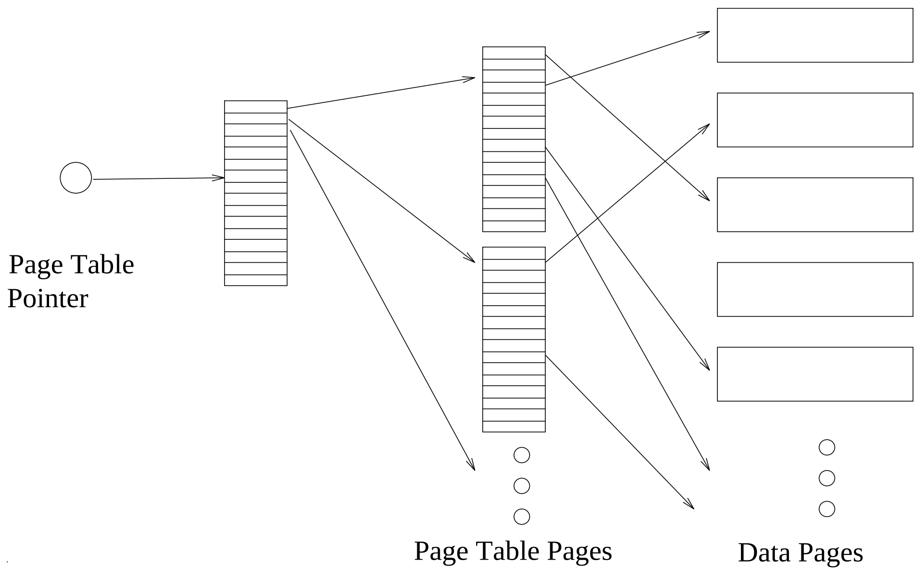

Figure 7.1: The shadow paging file structure.

explains why there are two physical page numbers), a timestamp of the last modification of the page table entry (Section 11), the size of the page (Section 13), plus flags and information for free space management.

The size of a page table entry is 16 bytes. This allows 40 bits for each physical page number, 40 bits for the timestamp, one byte for page size and flags and free space information. With 512 byte physical pages, 40 bits for the address allows 512 terabyte maximum database size. The timestamp is a commit batch sequence number. At ten batches per second it wraps around after 3 500 years. Page size is stored as $\log_2 S - 9$ in 4 bits. 4 bits are available for flags and allocation information.

Typical page sizes are from 512 bytes to a few kB for frequently updated relational data, and up to 1 MB for blobs and multimedia data. The size of the page table is about 0.01 % to 3 % of the size of the database. In many applications it fits entirely in main memory.

Concurrent transactions are implemented by having a per-transaction incremental page table which is used to store the changes made by the transaction. The incremental page table is conceptually a list of tuples $\langle L, P_{old}, P_{new}\rangle$, where L is the logical page number, P_{old} is the old physical page number, and P_{new} is the new physical page number. P_{old} corresponds to the global version of the page and P_{new} corresponds to the local version of the page.

Two-phase locking on logical pages is used for concurrency control (it will be seen in Sections 8 and 10 that with fine-granularity locking this is somewhat different, but the basic idea remains the same). To read a page, the transaction first acquires a shared lock on the logical page. It then looks for a corresponding entry in its incremental page table, and if not found, maps the page using the global page table. To write a page, the transaction obtains an exclusive lock on the logical page. It then checks if it already has an entry for the page in its incremental page table. If so, it uses the already-existing local version of the page. If not, it creates a copy of the page, and adds a corresponding entry to its incremental page table.

Since a transaction must have an exclusive lock before it can modify a page, only one transaction can have a local version of any given logical page, and there can be no conflicts between the incremental page tables of different transactions. This means that it is possible to install the modifications of several transactions into the global page table at once, reducing the overhead due to constructing new page tables. This is implemented by queuing all transactions which have requested to be committed, and having a separate process periodically (a few times a second or as soon as the previous commit batch has been completed) take all transactions in the queue and install their modifications (local versions) to the global page table. This is called *commit batching* [33, 43]. The basic idea is to combine many small transactions into a larger page table level transaction.

No garbage collection is needed with this method. After a transaction has committed, it can free the "old" versions of the pages (including page table pages) that it has modified. If a transaction needs to be aborted, all that has to be done is to free the "new" versions in its incremental page table; no modifications need to be made to the global database.

The free list of physical pages can either be extracted from the page table on disk at startup time, or be computed and written to disk at every commit batch.

7.2 The Impact of Technological Development

The cost of main memory has dropped dramatically since the first half of 1980's, when most performance comparisons between shadow paging and logging were made. It is now reasonable to assume that in most applications the entire page table of even a fairly large database will be available in the cache most of the time. Even for very large databases, those parts of the page table which are used frequently will remain in the cache. This has removed the most serious performance problem with shadow paging.

Chapter 8

Fine-Granularity Locking Supporting Extended Lock Modes and Early Releasing of Locks

It is nontrivial to do fine-granularity locking with shadow paging. The primary problem is that if the modifications were made to the actual data pages, and two transactions modified the same page, it would not be possible to commit or abort the transactions individually (they would have to be committed or aborted together since there would be no way to commit one and undo the other). If, on the other hand, each transaction created its own copy of the page, their changes would somehow have be merged at commit time, requiring extra bookkeeping and overhead. The only feasible approach thus seems to be to store the changes made by each transaction separately in a per-transaction data structure, and install the changes to the data pages only after the transaction has requested to be committed [105, 108].

The approach presented here is based on storing the changes separately in main memory. The changes do not refer to physical pages; instead, logical record identifiers are used, and the operation to be performed is stored instead of actual values.

Two-phase locking is used for concurrency control. The locks refer only to logical identifiers, and it is possible that the object moves to a different page while the transaction is active. The locking guarantees that the parts of the logical database which the transaction has modified will not be touched

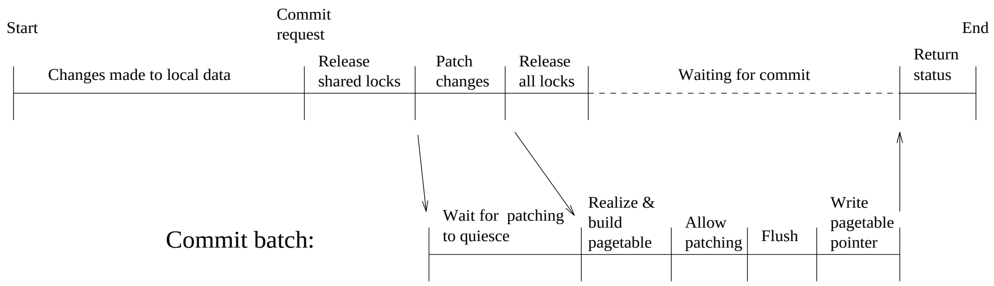

Figure 8.1: The lifetime of a transaction.

by other transactions.

Very large transactions are handled by escalating the locks to table level, and switching to use immediate page-level updates for that transaction and table. This is discussed in Section 8.6.

8.1 The Lifetime of a Transaction

For most of its duration a transaction is in the "active" state; all modifications are made in this state. In the active state, the transaction may at any time request to be aborted or to be committed. After the transaction has requested to be committed, it can no longer issue any other requests to the database. For some time the fate of the transaction is undetermined; then the system either aborts the transaction or commits it. In either case the result is then reported to higher levels of the system. (See Figure 8.1.)

In the active phase a transaction must protect all of its actions using appropriate locking. An object must be locked in shared mode before reading and in exclusive mode before updating.

With shadow paging and fine-granularity locking, it is normally not permissible to modify the actual cache page containing the data being modified. Instead, it is necessary to store the changes separately in a suitable per-transaction data structure. Only if the transaction holds a lock covering the entire page, can it allocate a local copy of the original data page and modify the page.

8.2 Commit Processing and Combining Transactions

When a transaction requests to be committed, all of its shared locks can be released immediately because it is known that the transaction will not do any more reads or updates.

At some point before a transaction can become permanent, all changes made by it must be patched to the global database. To avoid anomalies it is essential that all changes patched to a page are made by transactions that end up in the same commit batch. The easiest way to maintain this constraint is to guarantee that all transactions that have started to patch their changes end up in the same batch as all other transactions that have started to patch their changes but have not yet been included in some commit batch.

This essentially means that to start a commit batch, it is necessary to prevent further transactions from starting to patch their changes, to wait until all transactions currently patching have finished patching, and to include all transactions which have patched their changes in the batch. Patching can continue after the changes made in the commit batch have been entered into the page table; however, there is no need to wait until the page table or data pages have been flushed to disk. This means that transactions that come later can see uncommitted data; however, it is guaranteed that none of those transactions can commit without the earlier batch committing as well.

The changes made to the database by a transaction patching its changes are not visible to active transactions since the parts of the logical database affected by patching operations are exclusively locked by the transaction doing the patching. The physical database may, however, change anywhere; for example, a patching operation can cause a B-tree reorganization, but it will not affect the logical database. Patching operations themselves must use latches[1] to maintain consistency both against other patching operations and against active transactions. It is important to understand that locks are on logical database objects whereas latches are on pages, and the mapping can be far from one-to-one. Since only latches are used during patching, it is possible that not all of the updates made by a transaction are patched

[1]A latch is a low-level lock on a physical page [33]. They are used to protect the physical consistency of updates. A latch on a page locks the page in main memory. A shared latch allows reading the page, and an exclusive latch allows updating the page. Latches are often referred to as the FIX-USE-UNFIX protocol.

atomically. Other patching operations can thus see partial updates unless explicitly protected using latches. However, from the point of view of active transactions the patching happens atomically since the affected portions of the logical database are exclusively locked.

After a transaction has patched all of its changes, it must wait until its commit batch has been completed or the transaction has been aborted. It will be seen in Section 8.4 that it is possible to release all locks, including exclusive locks, before entering the wait. This means that locks do not need to be held during the wait and the processing of the commit batch (since patching typically only involves operations in the cache, it is very fast, whereas the commit batch must do real disk I/O, resulting in lengthy waits). Early releasing of locks is possible because of the ordering restrictions on transaction commits: it is not possible that a transaction which has read uncommitted data would commit without the transaction which modified the data also committing.

8.3 Aborting Transactions

Aborting transactions which have not yet begun patching is trivial, because those transactions have not yet made any modifications to the global database. It is thus sufficient to free all modification data and all local pages of the transaction.

If a transaction has to be aborted after it has released its exclusive locks, cascading aborts must be considered. All transactions which have accessed any data modified by the transaction must be aborted as well. This either requires keeping a graph of dependencies between transactions, or aborting all transactions which had not yet requested to be committed when the transaction released its locks.

A transaction can also get aborted after it has started patching but before it has released its locks. In this case it would have to undo its partial patching modifications. This would require code similar to undo recovery in log-based databases, and is clearly unwarranted for handling a few exceptional error conditions. Aborting all transactions that would go to the same or a later batch and throwing away any patched copies of pages solves this problem.

To summarize, if any transaction, for any reason, needs to be aborted after it has begun patching its changes, all transactions in the same or a later batch must be aborted. This is a very rare event and should only be caused by exceptional error conditions such as a system crash (in which case the cascading abort is implicit), a resource shortage (such as disk full), or a hardware or software failure (such as a disk crash). None of these happens under normal operation.

8.4 Early Releasing of Locks

The purpose of the early releasing of locks optimization [21, 22, 23, 57, 105] is to reduce the time that locks are held. It turns out to be possible to release exclusive locks immediately after the changes of the transaction have been made globally visible. This is based on the following properties of the system:

1. no transaction which has requested to commit after transaction A released its exclusive locks can commit before transaction A, and
2. no transaction which has requested to commit after transaction A released its exclusive locks can commit without transaction A also committing.

Informal proof of the first property. A transaction releases its locks after it has finished patching. The commit thread takes into a batch those transactions which have started to patch before a certain point of time. Since A has finished patching, it certainly has started to patch, and thus will be included in the next batch to start. Since B cannot be included in any batch earlier than the next batch to start, it cannot possibly commit before A.

Informal proof of the second property. As described in Section 8.3, if a transaction aborts after it has started patching, all active and uncommitted transactions in the system will be aborted. Since B is in the same or a later batch than A, and the transactions in a batch either all commit or all abort, B is not yet committed when A aborts. Since all uncommitted transactions are aborted if any transaction is aborted after it has started patching, B is also aborted if A aborts.

The locking protocol is not strict since exclusive locks are released before commitment. However, locks are only released after patching. Other transactions may modify the uncommitted data before the transaction which originally modified the data is included in a commit batch. All of the transactions which modify the data while it is uncommitted end up in the same commit batch, and are either all committed or all aborted (if a transaction uses page-level updates, it will create a copy of the page and not overwrite the uncommitted data except if it ends up in the same commit batch as the original transaction). On the other hand, after the page has been included in a commit batch, further modifications to the page will create a new copy of the data, effectively accessing a different version of the data (this is a very restricted form of multi-version concurrency control). Non-strictness of the locking protocol is thus not a problem with this method.

This locking protocol implies cascading aborts. Their handling was described in the previous section. Cascading aborts do not cause any problems in this case.

8.5 Relaxing Durability

If full persistence is not required by the application, it is possible to reduce transaction commit times by reporting success immediately when the transaction requests to commit. The transaction will then become permanent when the next commit batch completes (usually within a few tenths of a second). All the things that could cause the transaction to be aborted during that time are very exceptional, and can be treated as a crash. All other properties of ACID are maintained. The decision whether to relax persistence can be made on a per-transaction basis.

8.6 Very Large Transactions

The fine-granularity locking scheme described here does not work well for very large transactions, both because of the locking overhead, and because the data structures needed to hold the modifications of the transaction grow excessively large. Instead, large transactions are handled by switching to page-level updates after a transaction has made a certain number of modifications. This is used together with lock escalation. The idea is to make exclusive copies of the modified pages, so that there is no need to store the changes separately, and data can be flushed to disk from the cache.

Switching to page-level updates for large transactions will usually not reduce concurrency but will instead reduce the locking overhead. Without switching to page-level updates, there might not be enough main memory to store the changes of a very large transaction, and installing the fine-granularity changes of a large transaction at commit time would intolerably lengthen the time needed for processing the commit batch and would delay other transaction processing.

8.7 Extended Lock Modes

Extended lock modes [33, 99] for commutative operations (such as increment and decrement) can easily be supported with this scheme. The commutative operation is recorded in the fine-granularity changes of the transaction, and the actual update to the data is done at patching time.

8.8 Read Operations

Read operations return data from the disk filtered by changes in main memory. There are four levels where data may originate: the disk-based data structure, global pages into which changes have been patched but which have not yet been committed to disk, local copies in the transaction's own incremental page table, and fine-granularity changes maintained by the particular fine-granularity type in main memory. The first three levels are handled by the fine-granularity locking framework described here (a particular page will come from one of these sources); filtering the page-level data by the changes in main memory must be done by the fine-granularity level. Note that locking guarantees that only changes made by the transaction itself need to be considered.

8.9 Interaction with Write Optimizations

The write optimizations cause add some complexity to the cache and page table levels (cf. Section 12). The basic idea is to have *virtual page numbers* (Section 12.2) which refer to pages in the cache for which physical page numbers have not yet been assigned. The pages are then *realized* at commit time, which means that actual physical page numbers are allocated for them. The physical page numbers are then stored in the page table. (The cache is free to allocate physical page numbers at any time and map requests to the real physical addresses, but this is not visible to higher levels of the system.)

The implementation of concurrent transactions does not need to be aware whether the page numbers returned to it by the lower levels are virtual pages or real physical pages. In connection with fine-granularity locking the term *physical page number* will be used to refer to either type of a page number. The code to implement concurrent transactions and fine-granularity locking is completely identical whether or not the write optimizations are implemented.

8.10 Interaction with Media Recovery

Two physical page numbers are stored in page table entries for the implementation of media recovery (Section 14). However, if the physical page numbers shown to the implementation of concurrent transactions are virtual, the implementation of concurrent transactions does not need to know about media recovery. It never sees the page numbers stored in the page table, and it never knows how the data is stored physically.

8.11 Interaction with Snapshots

Determining when to free pages in the presence of snapshots (Section 11) causes extra work and complexity at the page table level. However, the implementation of concurrent transactions does not need to know about snapshots. A snapshot is taken by copying the page table pointer, and is not in any way affected by uncommitted transactions.

Chapter 9

Implementing Fine-Granularity Locking

The ideas described in the previous section have been implemented in the **Shadows** database system prototype being built at Helsinki University of Technology, Finland. This section describes the structure of that implementation to the degree that is needed for understanding the techniques.

9.1 Overview

Concurrent transactions have been implemented using C++ classes `PageTable`, `PageReference`, `BaseTransaction`, `BaseTransactionDatabase`, `Transaction`, `TransactionDatabase`, and a number of classes each defining one type of fine-granularity locking (Figure 9.1).

`PageTable` Implements the page table and atomic updates to the page table. It also implements write optimizations [106], snapshots [107], and page table caching (Section 16.1). This class is not described in detail here; however, the relevant interfaces are listed below. Levels above the page table never see physical page numbers which have been entered into the page table.

allocate_logical_page_number Allocates a logical page number.

free_logical_page_number Frees a logical page number (only used for freeing pages previously allocated in the same transaction; other pages are freed by changing their mapping to **nil**).

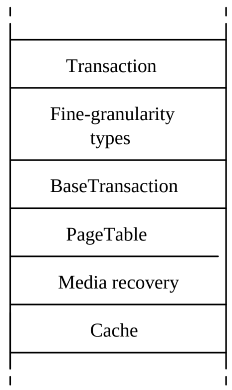

Figure 9.1: Layers of the Shadows system around the implementation of concurrent transactions and fine-granularity locking.

begin_update Begins a page table update.

update_mapping Tells the page table level the new mapping of a logical page. The page table level records the mapping in its private data structures.

commit_update Commits the page table update. This works in two phases. In the first phase this realizes the new physical pages and constructs a new page table. The new page table is then made visible to read_logical_page(), and a callback mechanism is used to inform the higher level that the new pages are visible through the page table. In phase two this flushes the new pages and the new page table pages to disk and updates the page table pointer. This then frees the old data and page table pages (provided that there are no remaining references to them from snapshots; cf. Section 11.2).

read_logical_page Translates the given logical page number to a physical page number, reads the page, and returns a read-only reference to it.

read_physical_page Reads a physical page and returns a read-only reference to it. Since physical page numbers become invalid when the page is entered into the page table, it is illegal to use the physical page number during that time (however, it is legal to have a reference to the page during that time).

read_physical_page_no_wait Reads a physical page if it is already in the cache; otherwise returns an error.

read_logical_page_no_wait Reads a logical page if it is already in the cache; otherwise returns an error.

allocate_physical_page Allocates a new physical page. Returns the number of the page and an updatable reference to the page.

allocate_physical_copy_of_logical_page Allocates a new physical page, copies the contents of an existing logical page to that page, and returns the new page number and a reference to it. If there are any references to the physical page corresponding to the logical page, this will wait until all references have been released. Most of the time this function can be implemented by "renaming" the page in the cache, and no data needs to be copied.

allocate_physical_copy_of_physical_page Allocates a new physical page, copies the contents of the specified physical page to it, and returns the new page number and a reference to the new page. If there are any references to the old page, this will wait until all references to the page have been released. Most of the time this function can be implemented by "renaming" the page in the cache, and no data needs to be copied.

read_physical_page_for_update Returns an updatable reference to the given physical page. It is illegal to read a physical page which has been entered into the page table, and it is illegal to have an updatable reference while it is being entered.

free_physical_page Frees the specified physical page. It is illegal to free a page which has been entered into the page table.

`PageReference` Acts as a reference to a page in the cache. The page is latched (in shared mode if the reference is read-only, and in exclusive mode if the reference is updatable). The method release() is used to release the latch and the reference when no longer needed. ptr() returns a `const` pointer to the data of the page, and updatable_ptr() returns a non-`const` pointer to the data (and raises a fatal exception if the reference is read-only).

`BaseTransaction` Implements concurrent transactions and the framework for building fine-granularity locking types. Most of the rest of this section is about this class.

`BaseTransactionDatabase` A descriptor for a database (or a snapshot) complementing the `BaseTransaction` class. This contains the data common to all active transactions in a database (or snapshot).

`Transaction` Implements concurrent transactions with fine-granularity locking. Mostly a wrapper for all supported fine-granularity types. New fine-granularity locking types are added to this class. Most requests are forwarded to the appropriate fine-granularity type; commit_transaction() and abort_transaction() are forwarded to `BaseTransaction` (there is one `BaseTransaction` object in every `Transaction` object).

`TransactionDatabase` A descriptor for a database (or a snapshot) complementing the `Transaction` class. This contains any per-database data which may be needed by fine-granularity types.

Fine-granularity types implement the various types of fine-granularity locking. For example, `FGBTree` implements fine-granularity operations for B-trees. All fine-granularity types implement the following operations: patch_changes() patches the changes made by the transaction to the global database, and abort_changes() frees all changes made by the transaction. Additionally, each fine-granularity type implements type-specific operations, such as insert() and delete() for B-trees. The implementation of B-tree management in this context is described in Section 10.

9.2 The BaseTransaction Class

`BaseTransaction` (together with `BaseTransactionDatabase`) implements most of fine-granularity locking. An overview will be presented in this section; Section 9.3 describes the data structures, Section 9.4 describes the locking protocols used, and Section 9.5 describes the implementation of each operation.

The `BaseTransaction` object offers the following interfaces for implementing fine-granularity types.

read_page Reads the specified logical page and returns a read-only reference to it.

read_page_for_update Reads the specified logical page and returns an updatable reference to it. If this is used while the transaction is active, locking must be used to guarantee that no other transaction can access any data on the same page.

allocate_page Allocates a new logical page number and returns an updatable reference to the page.

free_page Frees the specified logical page.

lock Performs the specified locking operation. Returns when the lock has been granted, a deadlock has been detected, or the timeout expires.

Additionally, it has commit_transaction() and abort_transaction() interfaces. It also expects the `Transaction` class to implement two interfaces which are called by `BaseTransaction`: patch_changes_callback() is expected to call the patch_changes() interface of each fine-granularity type, and abort_changes_callback() is expected to call the abort_changes() interface of each fine-granularity type.

The interfaces for implementing fine-granularity types can be used both while the transaction is active and from within the patch_changes_callback() function. However, they are implemented differently in the two cases. Also, if anything else than read_page() is used while the transaction is active, it is important to make sure that no other transaction is going to access the page (neither in active mode nor during patching). The only reason for allowing calls to modification operations while the transaction is active is to allow switching to page-level updates for large transactions.

9.3 Data Structures

A `BaseTransactionDatabase` object contains a list of all active transactions, a lock manager, a list of transactions waiting to be included in the next commit batch, an incremental page table for patched changes to go in the next commit batch, and two read/write locks called `patch_lock` and `stability_lock`.

A `BaseTransaction` object contains a pointer to the corresponding database object, per-transaction data for the lock manager, the state of the transaction (active, committed, etc.), and a per-transaction incremental page table for storing page-level modifications made by the transaction.

The incremental page tables contain $\langle L, P\rangle$ pairs (L is a logical page number and P is the corresponding new physical page number), and are implemented as hash tables. They support shared/exclusive-style locking of the mapping for an arbitrary L (whether in the table or not; the locking may be of a coarser granularity than a single mapping). The global incremental page table in `BaseTransactionDatabase` is called `global_remap`, and the local incremental page table in `BaseTransaction` is called `local_remap`.

To access a logical page, a transaction first looks for a mapping for the page in `local_remap`. If not found, it looks for a mapping in `global_remap`. If not found, it uses the page table. `local_remap` and `global_remap` thus act as filters for the database state the transaction sees.

9.4 Locking Protocols

There are several potential problems which must be avoided using locking (in this section, locking refers to semaphores[1] and other low-level locks).

1. Old physical page numbers become invalid when the pages are entered into the page table. The `PageTable` module will inform the commit batch via a callback when the pages have become visible through the page table. After the callback returns, the old physical page numbers are invalid and the pages can only be accessed through the page table.

2. Patching of changes must be atomic with respect to commit batches (that is, the changes of a transaction must all be included in the same batch).

3. If two transactions simultaneously attempt to patch a page for which there is no entry in `global_remap`, both might create a copy of the page. Similarly, a page number retrieved from `global_remap` might become invalid before being used (for example, due to being freed by another transaction).

4. Latching must be implemented in such a way that a shared latch on a page means that no-one else can get an exclusive latch on the same logical page (but a different copy). This is important when using latch coupling in the implementation of fine-granularity types.

`patch_lock` is a lock which can be obtained in either shared or exclusive mode. A transaction locks it in shared mode before it begins patching its

[1] In this thesis, a semaphore means any low-level lock, such as a Mutex, SpinLock, critical section, or whatever. They are distinct from transaction-level locks, no deadlock detection is done for semaphores, and they are distinct from latches (as used in this paper) in that they do not imply a reference to a page.

changes, and releases it after it has finished patching. A commit batch locks it in exclusive mode before determining which transactions to include in the batch. (Locking `patch_lock` in exclusive mode effectively quiesces patching activity.)

`stability_lock` is a lock which can be obtained in shared or exclusive mode. It is used to protect the validity of old page numbers on `global_remap` (problem 1 above). It is used in such a way that `stability_lock` is locked in exclusive mode only when `patch_lock` is being held in exclusive mode. Thus, either `patch_lock` or `stability_lock` must be held in shared mode while using `global_remap` to guarantee that the page numbers do not "disappear" while they are being used.

The first problem is thus solved by holding either `patch_lock` or `stability_lock` in shared mode while using page numbers on `global_remap`. The commit batch will be holding both locks in exclusive mode when it clears `global_remap` and invalidates the page numbers.

The second problem is solved by quiescing patching before making changes to the page table. This is done by having transactions lock `patch_lock` in shared mode while they are patching their changes, and having the commit batch lock it in exclusive mode when it wishes to quiesce patching.

The third problem is solved by locking the mapping for the affected page in `global_remap`. The mapping is locked in shared mode for read operations, and in exclusive mode for operations which may change the mapping.

The fourth problem is solved by having allocate_physical_copy_of_logical_page() and allocate_physical_copy_of_physical_page() wait until all references to the page have been released.

Locks are always acquired in the following order to avoid deadlocks: `patch_lock` first, `stability_lock` then, and `global_remap` mapping lock last. It is permissible to not lock some of these, but none of the former may be requested while any of the "latter" locks are being held by the thread.

There is no need to lock entries in `local_remap`, because a single transaction can do only one action at a time.

There is no need to protect against two transactions creating a local page for a single logical page. This should never happen, because transaction-level locking should only allow one transaction at a time to do page-level modifications on any given page. However, it is desirable to trap this condition, because it is an indicator of locking bugs in the implementation of fine-granularity types.

It is sometimes necessary to hold locks while reading data. It is desirable to implement these reads in such a way that the read is first attempted using read_physical_page_no_wait() (or its page table equivalent), and if unsuccessful, the locks are released, the page is read into memory, and the operation is restarted. This has not been described in the next section in order to keep the pseudocode comprehensible.

9.5 Implementation

This section describes the implementation of each of the operations performed by `BaseTransaction`. The operations are described in pseudocode. Error handling and handling pages of different sizes are not included because they are not essential for understanding the algorithms and would unnecessarily complicate the pseudocode.

In practice one could first attempt reads using the no_wait versions of the functions, and if the page is not in the cache, release all locks, read the page into the cache, and then retry the operation. This is not shown in the pseudocode.

9.5.1 commit_transaction

commit_transaction() works as follows.

Release all shared locks held by the transaction.
Lock `patch_lock` in shared mode.
Merge `local_remap` to `global_remap`.
Call patch_changes_callback() to patch changes.
Release all locks held by the transaction.
Add the transaction to the commit list.
Wake up the commit thread if it is sleeping.
Unlock `patch_lock`.
Sleep until the commit thread has either committed or aborted
  the transaction.
return status.

It is important to notice that `patch_lock` is already locked in shared mode when patch_changes_callback() is called, and is thus held when any of the other functions are called while patching.

The relaxing persistence optimization (Section 8.5) is not included in the pseudocode for clarity.

9.5.2 The Commit Thread

The commit thread works as follows. The code for handling failures and for terminating the commit thread is not included for clarity.

loop forever
  Sleep until work to do.
  Lock `patch_lock` in exclusive mode.
  Take all transactions from the commit list.
  begin_update().
  for all mappings $\langle L, P \rangle$ in `global_remap` **do**
    update_mapping(L, P).
  commit_update().
  Mark the transactions as committed and wake them up.

During the call to commit_update(), the page table module will call a callback function provided by the `BaseTransaction` class after the changes have been made visible through the page table but have not yet been written to disk. The callback function works as follows.

  Lock `stability_lock` in exclusive mode.
  Clear all mappings in `global_remap`.
  Unlock `stability_lock`.
  Unlock `patch_lock`.

9.5.3 abort_transaction

abort_transaction() works roughly as follows. The actual implementation is a bit tricky, because it must handle abortions at different stages of processing, and must handle aborting all transactions. It must also prevent new transactions from starting while aborting all transactions.

  Release all locks held by the transaction.
  Call abort_changes_callback() to free fine-granularity changes.
  Free new pages in `local_remap`.
  if the transaction had already started to patch its changes **then**
    Abort all active transactions in the same or any later batch.
    Free new pages in `global_remap`.

9.5.4 read_page

During patching read_page(L) works as follows. `patch_lock` is held in shared mode when this code is executed.

Lock mapping for L in `global_remap` in shared mode.
if $\langle L, P\rangle$ for L in `global_remap`
  then R = read_physical_page(P).
  else R = read_logical_page(L).
Unlock mapping for L in `global_remap`.
return R.

While the transaction is active read_page() works as follows.

if $\langle L, P\rangle$ for L in `local_remap` **then**
  R = read_physical_page(P).
  return R.
Lock `stability_lock` in shared mode.
Lock mapping for L in `global_remap` in shared mode.
if $\langle L, P\rangle$ for L in `global_remap` **then**
  R = read_physical_page(P).
  Unlock mapping for L in `global_remap`.
  Unlock `stability_lock`.
  return R.
R = read_logical_page(L).
Unlock mapping for L in `global_remap`.
Unlock `stability_lock`.
return R.

9.5.5 allocate_page

During patching allocate_page() works as follows. `patch_lock` is held in shared mode when this code is executed.

L = allocate_logical_page_number().
P, R = allocate_physical_page().
Lock mapping for L in `global_remap` in exclusive mode.
Add $\langle L, P\rangle$ to `global_remap`.
Unlock mapping for L in `global_remap`.
return L, R.

When the transaction is active (and using page-level updates) allocate_page() works as follows.

L = allocate_logical_page_number().
P, R = allocate_physical_page().
Add $\langle L, P\rangle$ to `local_remap`.
return L, R.

9.5.6 free_page

During patching free_page(L) works as follows. `patch_lock` is held in shared mode when this code is executed.

Lock mapping for L in `global_remap` in exclusive mode.
if $\langle L, P\rangle$ for L in `global_remap` **then**
  free_physical_page(P).
  Change mapping for L in `global_remap` to be $\langle L, \mathbf{nil}\rangle$.
else Add $\langle L, \mathbf{nil}\rangle$ to `global_remap`.
Unlock mapping for L in `global_remap`.

When the transaction is active (and using page-level updates) free_page() works as follows.

if $\langle L, P\rangle$ for L in `local_remap` **then**
  free_physical_page(P).
  Change mapping for L in `local_remap` to be $\langle L, \mathbf{nil}\rangle$.
else Add $\langle L, \mathbf{nil}\rangle$ to `local_remap`.

9.5.7 read_page_for_update

During patching read_page_for_update(L) works as follows. `patch_lock` is held in shared mode when this code is executed.

Lock mapping for L in `global_remap` in exclusive mode.
if $\langle L, P\rangle$ for L in `global_remap` **then**
  R = read_physical_page_for_update(P).
  Unlock mapping for L in `global_remap`.
  return R.
P_2, R_2 = allocate_physical_copy_of_logical_page(L).
Add $\langle L, P_2\rangle$ to `global_remap`.
Unlock mapping for L in `global_remap`.
return R_2.

While the transaction is active (and using page-level updates) read_page_for_update() works as follows.

if $\langle L, P\rangle$ for L in `local_remap` **then**
  R = read_physical_page_for_update(P).
  return R.
Lock `stability_lock` in shared mode.
Lock mapping for L in `global_remap` in exclusive mode.
if $\langle L, P\rangle$ for L in `global_remap` **then**
  P_2, R_2 = allocate_physical_copy_of_physical_page(P).
  Unlock mapping for L in `global_remap`.
  Unlock `stability_lock`.
  Add $\langle L, P_2\rangle$ to `local_remap`.
  return R_2.
P_2, R_2 = allocate_physical_copy_of_logical_page(L).
Unlock mapping for L in `global_remap`.
Unlock `stability_lock`.
Add mapping $\langle L, P_2\rangle$ to `local_remap`.
return R_2.

Chapter 10

B-Tree Index Management

B-trees are a widely used access method in databases. This section describes how to use the fine-granularity locking framework of Section 8 with B-trees. The solution is nicely structured, with recovery implemented by shadow paging as described earlier, fine-granularity updates implemented using the data structures and algorithms described in this section, and the transaction-level locking protocol implemented as the highest level layer above the others. The overall structure of the solution is shown in Figure 10.1; each layer builds on the services provided by the layers beneath it.

10.1 Background

B-trees were introduced in 1972 by Bayer and McCreight [9]. A more theoretical analysis was presented in [8]. The B-tree received lots of interest, and many papers were published in the next few years. Knuth [45, pp. 471–479] gives a good presentation of the early work, including the B^*-tree (nodes at least $\frac{2}{3}$ full) and B^+-tree (all data is in leaf nodes) variants (although the name B^+-tree came to use later; [18] discusses some of the confusion about the names).

Bayer and Schkolnick [10] presented in 1977 some schemes for concurrent operations on B-trees. Guibas and Sedgewick [34] presented the top-down approach (see also [73]). Lehman and Yao [52] presented B^{link}-trees, where much of the locking is avoided by the addition of an extra link in each node. Lanin and Shasha [51] presented an improved version of B^{link}-trees. The idea of a separate process for rebalancing the tree was introduced by Sagiv [93], and generalized by Nurmi and Soisalon-Soininen [76, 77] (see also [41]).

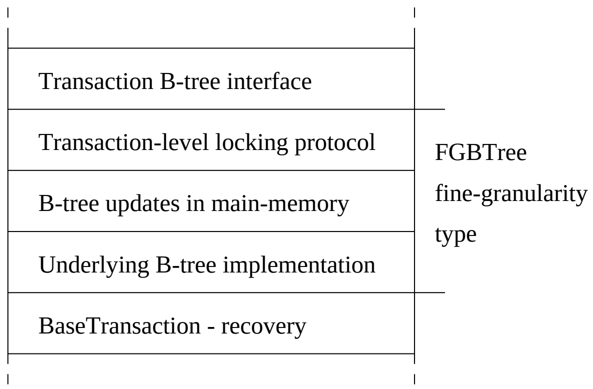

Figure 10.1: Layers of the Shadows system around the implementation of B-tree management.

Eiter, Shrefl and Stumptner [24] compare the operation of different B-tree concurrency control algorithms.

Mathematical analyses of the dynamics of B-trees have been presented e.g. by Zhang and Hsu [110], Baeza-Yates [6], Johnson and Shasha [39], and Matsliasch [63]. Srinivasan and Carey [97] have simulated B-tree concurrency control algorithms and found the B$^{\text{link}}$-tree [52] to perform best.

Solutions for building indexes for very large tables without preventing concurrent updates to the table have been considered in [68, 98]. Batching B-tree updates has been studied in [19, 50, 81, 82, 103, 104].

Most work on concurrent use of B-trees has ignored transaction recovery and isolation issues. Transactions have been assumed to consist of just a single B-tree operation. Recent work [33, 53, 54, 58, 59, 65, 67] has also considered recovery and isolation issues in log-based databases.

10.2 Operations on B-Trees

Operations supported by B-trees are described below. In the terminology used here, the B-tree contains arbitary records, part or all of the record being the B-tree key. The record may be a database record, or a $\langle key, RID\rangle$ pair for a secondary index. All records are assumed to have a unique B-tree key.

read_unique(key) Returns the record having the specified key, or an error if the specified record does not exist.

read_next(key) Returns the record with the smallest key greater than the specified value, or an error if there are no such records. One should note that this operation is not considered in many papers on B-trees. However, this operation is necessary in practical databases (e.g., for the implementation of scans).

insert(record) Adds a new record to the index. Returns an error if the specified record already is in the index.

delete(key) Deletes the record identified by the key, or returns an error if it does not exist.

update(key, field, operation) Performs a commutative update on the specified field of the record with the specified key value. Returns an error if the record does not exist. Note that even though the same operation could be performed by querying the old value of the record, deleting the record, and inserting it with the new contents, this operation permits locking optimizations which may be very important in hotspots.

write(key, field, value) Writes a new value to the specified field of the record with the specified key value. Returns an error if the record does not exist. Note that even though the same operation could be performed by querying the old value of the record, deleting the record, and inserting it with the new contents, this operation may permit optimizations.

One should note that general purpose database systems (particularly, the SQL standard [101, 38]) require that transactions can consist of an arbitrary number of queries or updates. It must thus be possible to group these basic operations (including range queries) arbitrarily into atomic transactions. Recovery mechanisms must guarantee the atomicity and durability of arbitrary transactions consisting of these operations.

10.3 Locking Protocols for Ordered Lists

There are some general locking algorithms for ordered lists of values [33, pp. 411–414]. The B-tree is a special case of an ordered list.

Ordered lists have the following operations: **read unique** (return a record, X, given its key), **read next** (return the next record, Y, after the record X), **read previous** (return the previous record W, before the record X), **insert** (insert record X between W and Y), and **delete** (delete record Y). Additionally, we can consider the **update** and **write** operations above. (Note that in this section only one operation will be assumed to be in progress at any time; having several of these operations in progress simultaneously will be considered in Section 10.8.)

The primary purpose of locking is to prevent phantoms (that is, records that seem to appear or disappear mysteriously, causing inconsistent results). This is equivalent to providing repeatable reads.

Phantom records arise in the following cases [33, p. 411]. If transaction T performs a **read unique** of record X, and it is not found, T must prevent others from inserting phantom record X until T commits. If T is at record W and does a **read next** to get record Y, then W and Y cannot change; in addition, no one may insert a new record (phantom record X) between W and Y, until T commits. A similar situation occurs with **read previous**. If T **inserts** record X, other transaction should **delete** X until T commits. If T **deletes** record Y (between X and Z), no other transaction should **insert** a phantom Y, until T commits. In addition, no other transaction should notice that the original Y is missing and that Z is now immediately after X, until T commits.

10.3.1 Key-Range Locking

The idea of key-range locking [33, pp. 411–412] is that the range of possible key values is divided to a set of fixed key ranges [R, S). A lock on the key starting the range is used to represent the whole range. The key ranges are thought of as predicate locks covering the records in the range. Multiple granularities of locks can be supported by using the granular locking protocol with ranges of different granularity.

A **read unique** locks the range containing the key in shared mode (requests a shared lock on the starting key of the range). A **read next** from W to Y locks all key ranges in the range from W to Y in shared mode. The **insert** of X and the **delete** of Y both lock the key range containing X or Y, respectively, in exclusive mode.

10.3.2 Dynamic Key-Range Locking

Static key-range locking works, but is not adaptive with respect to skewed key distribution. *Next-key locking* and *previous-key locking* are varieties of key-range locking that have a unique range between any two consecutive records in the file. In previous-key locking, the transaction requests a lock on the key value of X to lock the key range [X,Y). In next-key locking the range is (X,Y], and Y represents the range.

Dynamic key ranges appear and disappear as keys are inserted and deleted. This makes the locking protocol more complicated than static key-range locking [33, pp. 412–413]. The operation of previous-key locking is shown below; next-key locking works analogously.

read unique If the read unique operation finds the record X, it gets a shared lock for T on the key. That lock prevents anyone else from updating or deleting the record until T commits. Actually, since the shared lock on X is a key-range lock on [X,Y), it also prevents inserts in that key range.

If the record is not found, the read operation must prevent a phantom insertion of X by another transaction prior to T's commit. To do this, the read locks the current key range that would hold X. This key range is named by the key immediately before the phantom X. Suppose the value X falls between keys with values W and Y. Then the range is [W,Y), represented by W. A shared lock is requested on W to prevent others from inserting a phantom X.

read next Consider the case that transaction T is currently reading the record X and requests the next record Y in the ordered list. T already hold a lock on X, and thereby holds a lock on the range [X,Y). All T needs, then, is to request a shared lock on key Y. **Read previous** can be implemented correspondingly.

insert Inserting a new record in a key range splits the range in two. Inserting X into [W,Y) creates two key ranges [W,X) and [X,Y). First, the old key range [W,Y) must be locked in exclusive mode to ensure that it is not locked by another transaction. Then, the key range [X,Y) should be locked in exclusive mode. There is a subtle reason for this second lock: if another transaction reads the [X,Y) range, the access mechanism will see X in the list and attempt to get a key-range lock on [X,Y) rather than on [W,Y). T's lock on [X,Y) causes this second transaction to wait for T to commit. The record is inserted when both locks have been obtained.

delete The protocol for deleting a record is analogous to that for inserting a record. Delete merges two key ranges. To delete key Y from the sequence X,Y,Z, first lock Y (key range [Y,Z)) in exclusive mode, then lock key X (which is old key range [X,Y), soon to be key range [X,Z)) in exclusive mode. When these two locks are granted, perform the delete.

There is one troublesome point [33, p. 413]: if a key-range lock has to wait to be granted, then when it is granted, the key range may have disappeared. Consider, for example, the **insert** operation. Suppose that after the insert, a second transaction tries to read the record previous to Y. In that case it will wait for the [X,Y) key-range lock, which is currently held by the insert transaction. If the insert transaction aborts, the insertion will be undone, and the key-range will return to [W,Y). To deal with this, any transaction that waits for a key-range lock should revalidate the key range when the lock is granted. If the key range has changed, the transaction should release the lock and request a lock for the new key range. (Alternatively, it can simply restart the operation and it is likely succeed this time; however, this approach may require additional precautions to prevent starvation in hotspots.)

10.3.3 Separate Lock Modes for the Range

The next-key and previous-key protocols always lock the associated range, even if only the specific key value should be locked. This somewhat reduces the available concurrency.

Concurrency can be improved by having two separate lock modes for each key: one for the key itself, and one for the range that the key implies [59]. This can be implemented without modifications to the lock manager, e.g., by adding an indicator to the lock name specifying whether it is a lock for the key itself or for the corresponding range.

10.4 Secondary Indexes

Most of this section only considers unique indexes. However, similar algorithms can be used for secondary indexes as well. A secondary index contains records in the format $\langle key, RID\rangle$[1], where the key need not be unique. The

[1] RID is Record Identifier, a unique idenfier for the record in the database. The record identifier may contain a page number and in indentifier inside page, the value of the

update and **write** operations are not possible for secondary indexes.

A secondary index can be implemented most easily by concatenating the key and the record identifier to form the B-tree key. All queries are transformed into range queries.

All of the algorithms in this section can be used unchanged for secondary indexes when they are implemented this way. A B-tree with prefix omission [17] should probably be chosen as the underlying B-tree.

10.5 Data-Only Locking

When records are not stored directly in the index, and especially if there are many secondary indexes, it is beneficial to lock the actual data records instead of locking the key values [67]. This eliminates the need use DAG locking when records are accessed using several paths.

Data-only locking can be used with either next-key or previous-key locking. It can be seen as an optimization of the protocols, and can be implemented with a small modification to the locking functions. It cannot be used directly when there are separate lock modes for the ranges (when there are several secondary indexes, there are several ranges corresponding to each key; however, one could imagine using data-only locking for locks on keys, and ordinary locking for ranges).

10.6 Cursor Stability Locking

Cursor stability locking means that the record currently under cursor is locked for the transaction; however, shared locks are released as soon as the cursor moves away from the record. Exclusive locks are held until transaction commit.

Cursor stability can be supported trivially by releasing the shared lock on the key previously under the cursor when the cursor moves. For simplicity, cursor stability locking is not shown in the pseudocode.

---

primary key of the record, or some other data that is needed to identify the record in the database.

10.7 Lock Escalation

It is desirable to switch to a coarse granularity of locking after a transaction has done very many reads or modifications, both to reduce locking overhead and to avoid excessively long commit times as the modifications are merged to the index on disk.

Without predicate locking, B-trees do not easily support other locking granularities than record level locks, fixed key ranges, and index-level locks.

If a transaction has done excessively many read operations, it will obtain a shared coarse-granularity lock. Updates by the transaction will still be done using fine-granularity locking.

Lock escalation for updates is slightly more complicated, since this involves both escalating the locks and starting to use page-level updates for all or part of the index (cf. Section 8.6).

When a transaction has done a certain number of modifications, its locks are escalated to a coarser granularity. However, the fine-granularity locking framework of Section 8 requires that only a single transaction may perform page-level modifications on a page. Thus, it must be possible to determine which pages of the index are protected by the coarse-granularity lock, even after arbirarily many B-tree reorganizations. This is trivial for the lock on the whole index (all pages of the index are covered by the lock), but difficult for intermediate level locks (e.g., on fixed key ranges). One solution is to have a separate B-tree for each key range.

Once a coarse-granularity lock has been obtained, all changes to the physical pages covered by that lock can be patched to the B-tree on disk. All further updates to the key range covered by the lock are made directly to the B-tree.

Transactions should record the modes in which they hold coarse-granularity locks and only call the lock manager if they do not already have the lock.

10.8 Operations Required by the Implementation of Locking Procotols

The implementation of locking protocols requires the following operations from the B-tree management component: **read unique**, **read next**, **read previous**, **insert**, **delete**, **update**, and **write**. These operations do not perform any transaction-level locking. Their effects become immediately visible to other transactions. Several of these operations can be active concurrently, and they are performed atomically. Latches are used for synchronization.

However, there is a problem that needs to be considered. The implementation of the locking protocol uses the **read next** (or **read previous**) in several places to determine the key to be locked. Also, the **read unique** operation behaves differently depending on whether the key was found. It is possible that another transaction performs an **insert** or **delete** that changes the situation for the other operation before it has obtained transaction-level locks.

The abovementioned locking protocols may, in the worst case, have to perform a **read unique** followed by a **read next** (or **read previous**), as a single atomic operation. In the B^+-tree, this means having at most two nodes latched. This can be implemented fairly easily, but the B-tree interface must support this. Similar synchronization is needed for the main-memory data structures. A separate **release interval** call is used to release the interval when it has been protected by transaction-level locks.

One has to be careful to avoid potential latch deadlocks. There is no problem if one uses next-key locking and only supports the **read next** operation, or uses previous-key locking and only supports **read previous**. Supporting both directions may require requesting the second latch in conditional mode, and releasing everything and retrying if the latch is not granted. See Section 10.12.

10.9 Implementation of Fine-Granularity Updates

As was discussed in Section 8, fine-granularity locking with shadow paging requires storing the updates in main memory until the transaction has committed. The view shown to the locking protocol, on the other hand, contains both the permanent database and any uncommitted changes.

Uncommitted changes are kept in a main memory data structure. There is one such data structure for each B-tree; the same data structure is shared by all transactions. (However, when snapshots are introduced in Section 11, each snapshots has its own data structures.)

The data structure must have the following properties:

- Efficient access by key value.
- Efficient listing of all entries owned by a transaction.

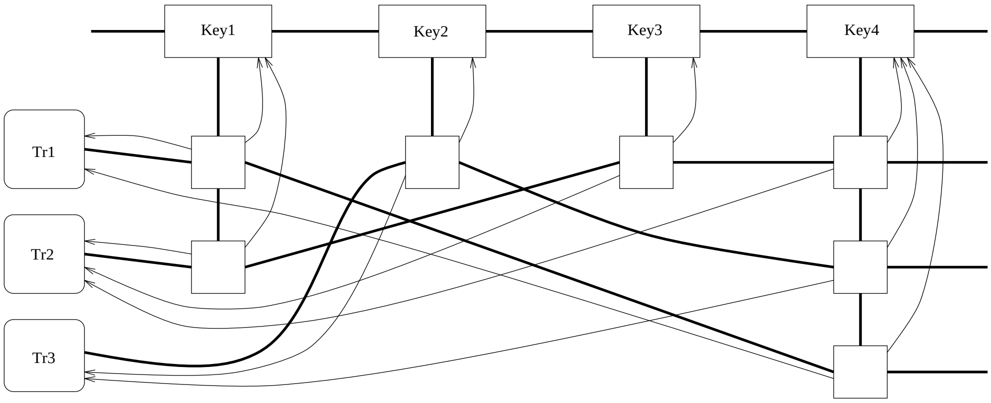

Figure 10.2: Data structure for storing fine-granularity changes for B-trees.

- Efficient insertion of new entries.
- Efficient deletion of all entries owned by a transaction.
- Efficient access to the next/previous key in key value order.
- Several transactions may have an entry for the same key value.
- The data structure is accessed concurrently.

The basic data structure used to store the changes is shown in Figure 10.2. Boxes on the left are per-transaction B-tree handles, boxes on the top are key values, and boxes in the middle are individual updates. The updates are stored in a chain attached to the key value.

There are three kinds of lists connecting the objects (shown as thick lines in Figure 10.2). One list connects all key value nodes in key value order. It is used to find the key value node by key value and to find the next/previous key node. A skip list [85, 86, 87] provides efficient implementations for searches, deletions, insertions, and for finding the next key, and it is efficient and easy to implement. Another alternative would be a balanced tree, but they may be more complicated and potentially less efficient.

Another list connects all updates made by a transaction. It is necessary to be able to traverse through the list. It need not be ordered. The whole list is released at once. A simple linked list is sufficient.

There is a list for each key value containing the updates made to the record identified by that key value. New entries are appended to the list, and it is released all at once. A simple linked list (with a tail pointer) is fine. This list will be very short most of the time, typically a single entry. However, there may be several entries if the transaction modifies the same record several times, or if there are several commutative updates to the same record by different transactions.

There is a global object (per snapshot) describing each B-tree in the system. That object contains the key list.

Each transaction has a per-transaction handle object for each B-tree that it accesses. These objects correspond to the boxes on the left in Figure 10.2.

Each B-tree update object (the boxes in the middle in Figure 10.2) contains the following information.

`key` Back-pointer to the key object containing this update.

`type` Type of the operation: INSERT, DELETE, UPDATE (commutatively updating the value of a field), or WRITE (overwriting the value of a field).

`data` Data needed for performing the operation. For INSERT this is the new record; for UPDATE this identifies the field and the operation to be performed; for WRITE this identifies the field and gives the new contents of the field. No data is needed for DELETE.

10.10 Concurrent Access to Main-Memory Data Structures

The list of key values is accessed concurrently by many transactions, and there are concurrent insertions, deletions and queries (including queries for the next key). Algorithms for concurrent use of skip lists have been presented in [85]; concurrent algorithms for tree structures can be found in [26, 25, 47, 62, 76].

The list of updates made by a transaction is not updated concurrently. No locking is needed for it.

The list of updates for a key value might be accessed by several transactions; however, the concurrency requirements are not high. The best way to protect the list may be to share semaphores with the list of key values (one has to be careful to avoid race conditions in deletions and insertions). The exact implementation depends on the data structures and synchronization protocols used.

10.11 Underlying B-Tree

It is important to notice that the implementation of recovery and transaction-level fine-granularity locking does not constrain the implementation of the underlying B-tree. Almost any variant of B-trees can be used, with latches used for what is conventionally called locking in the context of B-trees. This separation significantly simplifies the implementation and permits one to choose the most appropriate B-tree variant. The current implementation uses B^{link}-trees, but this is by no means the only possibility.

The following operations must be implemented by the underlying B-tree: **read unique** reads a record with a specific key value, **read next** returns the record with the next higher key value, **read previous** returns record with the next smaller key value, **insert** inserts a new record, **delete** deletes a record, **update** commutatively updates the contents of a field, and **write** writes a new value to a field.

All read operations leave the relevant leaf node latched until **release interval** is called. It is permissible to follow a **read unique** by one call to either **read next** or **read previous**, always the same direction for the same B-tree (cf. Section 10.12). This permits the **read unique** and the following **read next** or **read previous** to be performed as a single atomic operation.

The underlying B-tree is implemented using the services provided by the BaseTransaction class (Section 9.2). This means that page-level updates are used in the underlying B-tree implementation, and its functions can only be called while patching fine-granularity changes to the global database, or if the transaction is using page-level updates.

One should note that no special handling is needed for B-tree structure modification operations (node splits or merges). Structure modifications are only done during patching, or when a transaction is doing page-level updates. The fine-granularity locking framework will guarantee that the structure modification operation is done atomically. Since locks and changes in main memory only use logical identifiers (no references to specific pages), and there is no log with references to specific pages, no-one cares about which page each record resides on. Normal latching is used to guarantee the consistency of individual B-tree operations.

10.12 Supporting Both Read Next and Read Previous

No deadlock detection is used for latches, and deadlock prevention must therefore be used. This is most easily done by acquiring latches always in the same order.

If only the **read next** operation is supported and next-key locking is used, all accesses are in the same direction. The situation is similar if only **read previous** is supported and previous-key locking is used.

To support both **read next** and **read previous**, either operation will need to acquire latches in the "wrong" direction. To prevent deadlock, one can request the latch in "don't wait" mode, and back off if the latch cannot be taken. There are several possible recovery actions:

- Release all latches, and restart the entire operation. This is simple, but may lead to starvation in hotspots.

- Release some or all latches, and reacquire them in the proper order. After the latches have been acquired, validate that they are still the correct latches (the B-tree leaf might have been split in the meanwhile). If they are no longer valid, restart the operation.

For many systems it is sufficient to support only one of **read next** and **read previous**, and choose the locking protocol accordingly. This is the approach taken in [65].

10.13 Pseudocode for the Locking Protocols

Pseudocode is given below for the implementation of the next-key locking protocol.

For simplicity, lock escalation is not shown in the pseudocode (cf. Section 10.7). Also, releasing latches when a lock request blocks is not shown. **Read previous** is not shown.

10.13.1 Read Unique

Read unique(X):
  status,record = read $\geq$ X
  if (status == found **and** record == X)

```
    lock shared X
    release interval
    RETURN record
  if (status == found)
    lock shared record
  else
    lock shared "end of index"
  release interval
  return "not found"
```

10.13.2 Read Next

```
Read next(X):
  status,record = read > X
  if (status == found)
    lock shared record
    release interval
    return record
  lock shared "end of index"
  release interval
  return "not found"
```

10.13.3 Insert

```
Insert(X):
  status,Y = read > X
  if (status == found)
    lock exclusive Y
  else
    lock exclusive "end of index"
  lock exclusive X
  release interval
  insert X      // may fail
```

10.13.4 Delete

```
Delete(Y):
```

```
  lock exclusive Y
  status,Z = read > Y
  if (status == found)
    lock exclusive Z
  else
    lock exclusive "end of index"
  release interval
  delete Y        // may fail
```

10.13.5 Update

```
Update(X, field, operation):
  status,record = read ≥ X
  if (status == found and record == X)
    lock exclusive X
    release interval
    return update X, field, operation
  if (status == found)
    lock shared record
  else
    lock shared "end of index"
  release interval
  return "not found"
```

10.14 Pseudocode for the Fine-Granularity Update Level

This section shows the implementation of the operations used by the locking protocols. These in turn use operations provided by the main-memory data structures and the disk-based B-tree.

10.14.1 Read CC

Note: CC is either $>$ or $\geq$.

```
Read CC:
  status1,record1 = read CC from btree
  status2,data = read CC from main memory
```

```
  select smaller, or combine if equal
  release larger
  note which is locked
  return smaller
```

10.14.2 Release Interval

```
Release interval:
  release noted latches
```

10.14.3 Insert

```
Insert(X):
  status,Y = read ≥ X from btree
  if (status == found and Y == X)
    release btree
    return "already exists"
  status = insert X in main memory        // may fail if exists
  release btree
  return status
```

10.14.4 Delete

```
Delete(X):
  status,Y = read ≥ X from btree
  if (status == not found or Y != X)
    // not found in B-tree
    status,Y = read ≥ X from main memory
    if (status == not found or Y != X)
      release main memory
      release btree
      return "not found"
    release main memory
  delete X in main memory
  release btree
```

10.14.5 Update

```
Update(X):
  status,Y = read ≥ X from btree
  if (status == not found or Y != X)
    // not found in btree
    status,Y = read ≥ X from main memory
    if (status == not found or Y != X)
      release main memory
      release btree
      return "not found"
    release main memory
  update in main memory
  release btree
```

10.15 Potential Optimizations

All of the existing locking protocols for B-trees make the following assumptions about the underlying system:

- The effects of insertions, deletions, and updates become visible immediately.
- It is not possible for a transaction to see a key which has been deleted but whose deletion has not yet been committed.
- Other transactions see keys which have been inserted but not yet committed.
- Updates become visible to other transactions immediately, even while uncommitted.

In most log-based systems without multi-version concurrency control, it is not feasible to have a radically different set of assumptions. These assumptions are easily satisfied with shadow paging as well (as was done in this chapter). However, it is also possible to make a different set of assumptions:

- The effects of insertions, deletions, and updates are kept on a separate data structure until the transaction commits. The system can choose, individually for each update or transaction, whether it wants the transaction to see the current uncommitted state or the last committed state, or something in between.

- A transaction can see any deleted (but uncommitted) keys if it so desires.
- Transactions can choose not to see uncommitted insertions.
- Transactions can choose to see or not see uncommitted updates.

The first set of assumptions means that the order of the two transactions becomes fixed when their locks first conflict. The second transaction will have to wait until the first transaction has completed.

However, with the second set of assumptions, when the first conflict is detected, it is possible to read either the old or the new value of the record. This permits the scheduler to decide a serialization order for the transactions. Whenever these transactions conflict, the value to read is chosen according to the selected serialization order. Blocking is necessary only in the case that a new conflict would create a circle in the serialization graph.

Another possibility for optimization is storing the locks themselves in the B-tree data structures. This would avoid calling the lock manager (which keeps data structures that essentially parallel those of the B-tree module, except that the B-tree module does not ordinarily need to keep entries for keys that have only been read), and would save some memory space. It would also permit separate modes for the range locks fairly easily and efficiently.

Both of these optimizations are beyond the scope of this thesis.

Chapter 11

Snapshots, Read-Only Transactions, and On-The-Fly Multi-Level Incremental Dumping

11.1 Introduction

A snapshot is a transaction-consistent[1] copy of the database [1]. Shadow paging allows taking transaction-consistent snapshots of the entire database by saving the address of the page table and preventing freeing of pages referenced from the snapshot [105, 107].

There is not much previous work on snapshots with shadow paging. In System R [32] shadows were used for checkpointing, but their approach does not allow more general snapshots. Other work on snapshots [1, 40, 56] involves actually copying the data or using the log to access old versions; in those approaches the creation, deletion, refreshing, or accessing of snapshots is very expensive. Object-level versions are commonly used in object-oriented databases, and a snapshot-like interpretation for them has been discussed in [14]. The approach presented here is, to the author's best knowledge, completely new.

The problems in supporting snapshots efficiently include

[1]Transaction-consistent means that all transactions are either fully reflected in the copy, or not at all. No effects of uncommitted or aborted transactions will be visible in a transaction-consistent copy.

Figure 11.1: Snapshots represent consistent database states as of some time in the past.

1. determining when a page can be freed (i.e., when is the last reference to the page removed),
2. determining which locations to check for modified pages when dropping a snapshot without scanning the entire page table,
3. how to support modification of snapshots, and
4. how to make snapshots permanent.

11.2 Freeing Physical Pages

The primary problem with snapshots is determining when a page can be freed. It turns out that the properties of shadow paging allow a very efficient implementation. From within a single snapshot a physical page can be referenced only once (multiple references to the same physical page from several logical pages are not allowed). It is also not possible to move a physical page from one logical location in the database to another logical location. This means that if a page, which is referenced from one snapshot, is also referenced from some other snapshot, that reference must be from the same logical location. For data pages this means the same logical page; for page table pages this means the page table pages mapping the same logical pages.

All active snapshots in the system form a chain in chronological order (Figure 11.1). The immediate predecessor of a snapshot in this chain will be called the parent of the snapshot, and the immediate successor will be called a child of the snapshot. Only the current database state can be modified.

If a snapshot has a reference to a page, the page must either also occur in the parent snapshot, or it must have been created during the time interval between taking the parent snapshot and the current snapshot. Thus, to determine whether a page can be freed, it is only necessary to check whether the timestamp of the page was newer than the timestamp of the previous snapshot.

This algorithm is used to free the old versions of data and page table pages after a commit batch has committed successfully. The modified pages of an aborted transaction can always be freed without any checking, because they have not been made visible to other transactions.

11.3 Dropping Snapshots

Snapshots should be dropped as soon as there are no references to them in order to release their disk space. It is possible to scan all pages in the snapshot and use the timestamps to determine whether each page can be freed, but in practice this would be too costly. More efficient methods are possible.

11.3.1 Change Sets

A page can differ in a snapshot and its parent (note that the current database state is also seen as a snapshot in this context) only if it has been modified during the interval between taking the snapshots. It is possible to record in the chronological chain of snapshots the set of logical pages that have changed between two consecutive snapshots. The change sets can be used to efficiently determine which pages need to be freed. The method directly gives the set of pages that need to be freed as the intersection of the left and the right change sets of the snapshot being dropped.

There are three things to consider here: modifying the current database, taking a new snapshot, and dropping a snapshot. The snapshots are assumed to form a chain in chronological order (Figure 11.1). Associated with each link is a set of logical page numbers that had been modified (created, deleted, or updated) between taking the two snapshots. For simplicity of description, an imaginary snapshot is assumed at infinity in the past and at infinity in the future. The sets associated with the imaginary links from the infinite past and the infinite future contain all possible logical page numbers.

Whenever any modification is made to the current database, the number of the logical page number which was modified is added to the left change set of the current database state.

A new snapshot can be added in the chain immediately on the left side of the current database state. Effectively the link between the previous newest snapshot and the current state becomes the link between the previous newest snapshot and the new snapshot, and it retains its change set. A new link is created between the new snapshot and the current state; the change set of this link is initially empty.

Pages may need to be freed when dropping snapshots. A page was inherited from the parent snapshot if it was not changed between the snapshots (it is not in the change set associated with the link). Correspondingly, it has been inherited to the child if it has not been changed between the snapshot and the child. The page has no other references if it is inherited neither from the parent nor to the child. Thus, the pages that can be freed are those in the intersection of the left and the right change sets of the snapshot, and any page table pages used for mapping those pages.

There are some implementation problems in using this approach directly. One issue is maintaining the change set. If it is in main memory, it can grow rather big. If the snapshots are permanent (Section 11.5), the change sets must also be permanent and thus saved to disk at every commit batch (which is slow if the change set is big; however, some kind of an incremental approach might be workable). There are also update problems: modifications to the change set must be written to disk before the page table address is written; however, those changes should become visible only after the page table address has been written. Additionally, the change set can become very large in some applications, and it must be processed efficiently. For example, two change sets need to be merged when dropping snapshots. Putting all the complications together, it may be rather difficult to use change sets in an actual system.

11.3.2 Timestamps in Page Table Entries

A suprisingly simple implementation of the change sets is to store the timestamp of the last modification in every page table entry. The timestamp is the sequence number of the last commit batch which has modified the entry. Higher-level page table pages also contain a timestamp in each entry, indicating the maximum value of any timestamp in the subtree pointed to by that entry.

The sequence number of the last transaction included in each snapshot is stored with the snapshot. It is possible to reconstruct the change sets by traversing the page table (but this is not necessary in practice): the change set includes those logical pages that have changed after the timestamp of the previous snapshot. The timestamp of the imaginary "infinite past" snapshot is negative infinity, and the timestamp of the "infinite future" snapshot is positive infinity.

A changed page can be freed if its old timestamp was newer than the timestamp of the previous snapshot. There are no problems with page table pages: they also have timestamps (either in the higher-level page table entries pointing to them or in the page table pointer), and it can thus be directly determined whether the page table page is newer than the newest snapshot.

Taking a new snapshot is trivial; the current timestamp is just recorded in the new snapshot and the new snapshot is added to the chronological chain.

Dropping a snapshot is a little more complicated. All those pages should be freed that are newer than the timestamp of the previous snapshot, and older than the corresponding page in the following snapshot. These pages can be found by traversing the page tables of the dropped and the following snapshot in parallel and comparing the timestamps; however, since the timestamps in higher-level page table pages contain the maximum of any timestamp in the corresponding subtree, only those subtrees need to be traversed which contain changed pages. If the database is large and there are relatively few changes (or the changes are localized in some parts of the database), only a small fraction of the entire page table needs to be traversed. The pseudocode for this operation is shown in Figure 11.2.

11.4 Modifying Snapshots

It is possible to modify a snapshot. The snapshot behaves like a copy-on-write[2] copy of the database. From the user's point of view, each snapshot is an independent copy of the database. It is possible to make modifications to the copy, and these modifications will not affect other copies. No synchronization of transactions is required between two copies, and the copies diverge as more modifications are made.

Such modifiable snapshots are called *versions of the database*. They are different from ordinary object-level versions in that they are global, and object-level versioning may be used together with database versions if desired. A similar concept has been used in [14] for version identification in the context of object-oriented databases; however, their implementation is entirely different.

The versions and snapshots of a database form a tree (Figure 11.3) as opposed to the linear chain of snapshots (Figure 11.1). Only snapshots with

[2]Copy-on-write means that a page is not copied immediately. Instead, the same page is shared by all versions, and a private copy is made when the page is first modified.

```
drop_unreferenced_pages(pt_left, pt_drop, pt_right):
  drop_traverse(pt_drop.root, pt_drop.timestamp,
      pt_right, pt_left.timestamp, 0,
      pt_drop.depth)
drop_traverse(subtree, subtree_timestamp, pt_right,
      left_timestamp, L, level):
  if subtree = nil
    then return
  if subtree_timestamp ≤ left_timestamp
    then return
  if subtree_timestamp ≥ timestamp_on_level(pt_right, L, level)
    then return
  if level > 0 then
    R = reference to page subtree
    for i = 0 ... N − 1 do
      E = R.entries[i]
      LL = L + i N^(level−1)
      drop_traverse(E.physical, E.timestamp, pt_right,
          LL, level - 1)
      Release R.
  Free the physical page subtree.
```

Figure 11.2: Pseudocode for dropping a snapshot using timestamps to determine which pages to free. timestamp_on_level() returns the timestamp of the page table entry corresponding to the specified page on the specified level of the page table (if the requested level is higher than the depth of the page table, it returns the timestamp of the page table pointer). Its implementation would be incremental in practice, exploiting information from the previous call.

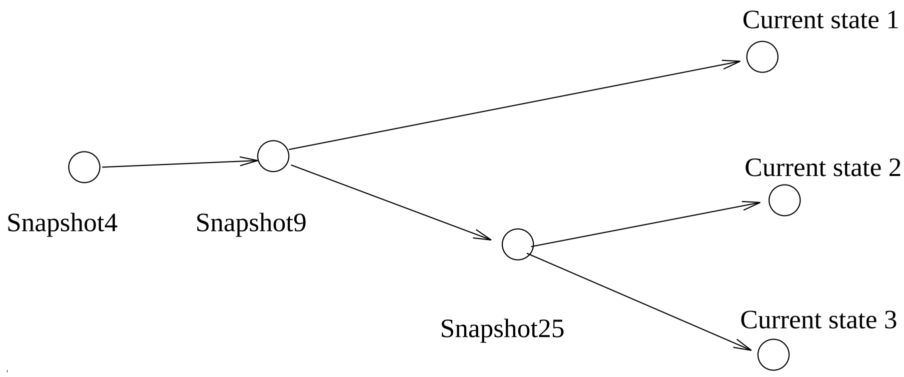

Figure 11.3: Versions of a database form a tree. Implicit snapshots are retained of all forking points.

no children can be modified; if a snapshot with children is to be modified, a new snapshot can be taken from it and that snapshot modified.

The tree can be seen as a collection of chains of snapshots, where the older ends of the chains have nodes in common. All the algorithms in Sections 11.2 and 11.3 can be used.

It is not permissible to drop a snapshot which has more than one child (a forking point in the tree). If dropping such snapshots were allowed, the assumption that other references must be either from immediate parent or immediate child would no longer hold, and the algorithms described in this paper would not work. A forking point can be dropped when it no longer has more than one child and has no other references.

11.4.1 The Logical Free List

The free list of logical page numbers is independent in each snapshot, and can be very large. Also, the list must be copied when taking a snapshot, and freed when releasing a snapshot. However, in most cases the snapshot will not be modified or only a small fraction of it will be modified. If garbage collection is available, the problem is trivial (just copy the pointer and never modify the nodes). However, most database systems use explicit memory management, and different mechanisms must be used to avoid the linear copy and free times for a normal list.

One possible solution is to use reference counting. Four operations are done on the logical free list: **get** (returns a number from the list), **put** (adds

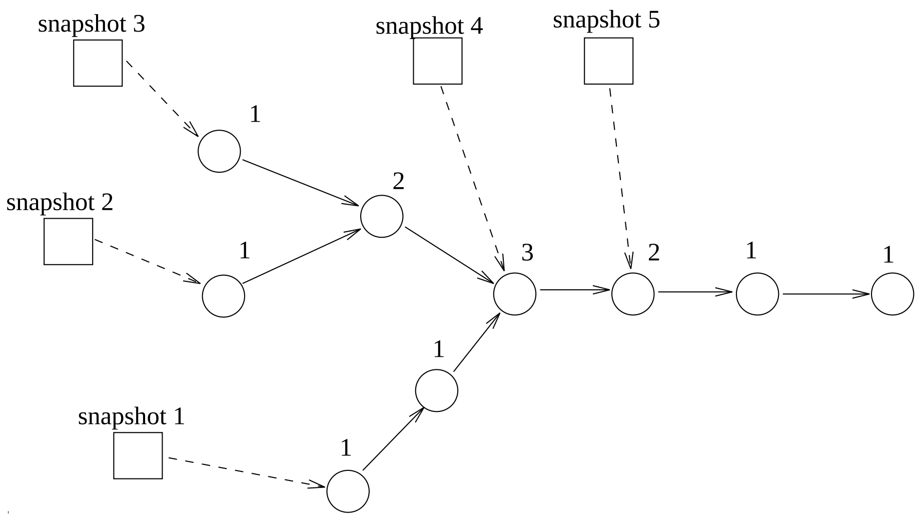

Figure 11.4: The logical free list data structure and its reference counts in the presence of multiple snapshots.

a number to the list), **copy** (returns a copy of the list), and **free** (frees the entire list). All these operations can be implemented with O(1) time complexity, except for **free** which has a linear worst case.

Each node has a reference count which indicates how many direct pointers there are to the node. A pointer may be either from another node or from a snapshot. The data structure is partially shared between copies (Figure 11.4). A node can be freed when its reference count becomes zero.

Get decrements the reference count of the current node and moves the current node pointer to point to the next node. If the reference count of the old current node reaches zero, it is freed; otherwise the reference count of the new current node is increased (note that it is not increased if the previous node was freed). The page number from the old current node is returned.

Put creates a new node containing the given page number. The next pointer of the node will be set to point to the old current node, and the new node will be made the current node. The reference count of the new node is initialized to one.

Copy increments the reference count of the current node. The returned new list is actually the same as the old list (but they have independent current pointers).

Free decrements the reference count of the current node. If the reference count reaches zero, the node is freed and the next node is made the current node. This is continued until the reference count is not zero after decrementing, or the end of the list is reached.

11.4.2 Other Main Memory Data Structures

There may also be other main memory data structures associated with a snapshot. They may require special handling to allow modification of snapshots. Small data structures can simply be copied; larger data structures must be handled similarly to the logical free list. A notable exception is data structures used to implement fine-granularity types. They are always initialized to be empty when creating a snapshot, since they only contain uncommitted changes which are not included in a snapshot.

11.5 Permanent Snapshots and Versions

There are two basic approaches to make several snapshots or versions permanent in the database. In the first approach each snapshot or version is an independent database, and a single transaction can modify only one version without resorting to distributed transaction processing algorithms. Each version has its own page table pointer and transactions can be committed independently to each version. A *master pointer* is used to contain the addresses of the page table pointers; both the master pointer and the page table pointers are in stable storage. This file structure is illustrated in Figure 11.5. If there are very many permanent snapshots, the master pointer may have extension pages which are stored in normal shadowed storage. Creation and deletion of versions are atomic operations which are asynchronous with respect to transaction processing. The rest of this section primarily considers this approach.

In the second approach the versions are more like objects in a database, and it is possible to access multiple versions in a single transaction (this corresponds to the definition of the database version concept in [14]). There is only one atomically updatable page table pointer, but it may contain several page table addresses. Creation and deletion of versions are normal operations that are done inside transactions. (It is also possible to use a combination of these two approaches: there can be a master pointer which points to a number of page table pointers, each of which contains one or more page table addresses.)

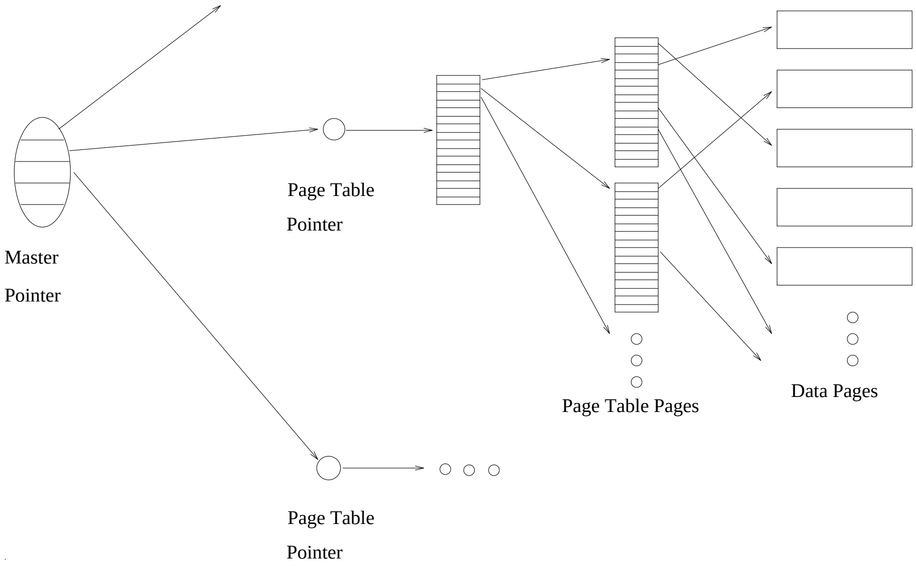

Figure 11.5: File structure with a master pointer and multiple versions.

All snapshots are initially temporary. They become permanent when a permanent reference is created, usually by giving the snapshot a name. Using the first approach, all of its pages must be forced to disk (if the snapshot has been modified after it was taken, some of the modifications may not be on disk since no flushing is needed when executing transactions to a temporary snapshot), and a page table pointer is allocated for the snapshot. When all pages and the page table pointer have been flushed to disk, the address of the page table pointer is atomically inserted to the master pointer. With the second approach, a reference is created when the address of the page table is stored to the page table as part of commit processing, and there is no separate page table pointer.

It is important to guarantee that the hierarchical relationships of permanent snapshots is retained over crashes. In particular, if two permanent snapshots have a common ancestor, that ancestor must implicitly be permanent.

It is possible that a snapshot is on the path between two snapshots which must be permanent, there is an application reference to the snapshot, but the snapshot itself need not be permanent. Such snapshots should no longer exist after recovery. This can be implemented either by checking for such snapshots at recovery time or by storing only the truly permanent hierarchy on disk. The former alternative requires a little extra recovery code; the latter alternative requires maintaining two overlapping snapshot hierarchies (the temporary hierharchy in main memory and the permanent hierarchy on disk). Both alternatives are feasible.

To make a permanent snapshot temporary, the pointer to the page table pointer must be removed from the master pointer (or the page table pointer if using the second approach). After that, the snapshot is no longer permanent. In many cases the snapshot is freed immediately after making it temporary (only temporary snapshots can be freed). The system must ensure that snapshots which are required to maintain the permanent snapshot hierarchy (forking points) are not made temporary, but this need not be visible to application programs.

11.6 Applications

11.6.1 Read-Only Transactions

Any read-only transaction can be implemented by taking a snapshot of the entire database, reading from the snapshot, and releasing the snapshot at the end of the transaction. No locking or any interaction with other executing transactions is needed, yet the read-only transaction sees a consistent database state. The read-only transaction is effectively serialized to the moment when the snapshot was taken. (This is one way of implementing multi-version concurrency control [11, 12]; read-only transactions in that context have been discussed e.g. in [2].)

This solves the problems of large or long-duration read-only transactions that are difficult to solve in conventional log-based databases. Since the transaction reads from a snapshot, no synchronization is needed with other transactions and thus there will be no conflicts with update transactions. Transient versioning [71] can used for the same purpose in log-based databases.

11.6.2 On-The-Fly Multi-Level Incremental Dumping

One use of snapshots is to implement dumping of the database while transaction processing is active. The backup process can be seen as a large read-only transaction, and can be implemented by taking a snapshot at the start of the dump, reading the logical database using the snapshot, and dropping the snapshot at the end of the dump. Only the logical database is dumped; unused free pages or page table pages are not dumped.

Incremental dumping (that is, dumping of the changes made after the previous dump) can be implemented using timestamps in page table entries (cf. Section 11.3.2). The timestamp of the snapshot is recorded whenever a dump is taken. When taking an incremental dump, the timestamps of all page table entries are compared against the timestamp of the previous dump, and only pages modified after the previous dump are dumped. Pages which have a null page number in their page table entry have been deleted; those pages are naturally not dumped but instead the fact that the page has been freed since the previous dump is recorded.

Multi-level incremental dumping allows several levels of dumps, e.g., a monthly, weekly, and a daily dump. The idea is to dump the changes made since the last higher-level dump. This can be implemented by recording the timestamp of the last dump of each level, and dumping only pages that have been modified after the desired timestamp.

Since the dump only reads the page table (which is usually in main memory) and the pages which are actually dumped, incremental dumping is very efficient. It is also possible to read directly from the disk while taking a dump, bypassing the normal disk cache. This may be desirable to avoid overhead and contention of the cache [69]. Since the physical pages referenced from the snapshot do not change, reading directly from the disk is trivial and does not change the algorithms in any way.

Many algorithms have been presented for on-the-fly dumping of log-based databases [36, 44, 69, 84, 91, 94]. Most of the algorithms are based on taking a fuzzy dump of the database and dumping the log records of transactions which were active during the dump. [84] describes an algorithm which can be used to directly take a consistent dump using locking (and correspondingly, possibly causing some transactions to be aborted; it should also be noted that [84] uses the term "incremental" in a different sense).

Some of the earlier algorithms can support incremental dumping by having a bit in the data pages which is set whenever the page is modified [36]. In [69] the bit was stored in the space map pages of the database. Both of these methods can be extended to incremental dumping by storing as many bits as there are dump levels. Log sequence numbers (LSNs) can be used in a similar way to do incremental dumping. Most of the algorithms require scanning the entire database even for incremental dumping. The algorithms in [69] are use bits in the free map pages to determine which pages need to be dumped, and thus do not require a full scan.

11.7 Modifiable Snapshots

Multiple permanent versions could be useful in a number of applications. Examples include large design databases, where several design directions could be considered simultaneously. Another application could be asking "what if" questions in a large database without affecting the original database. In large knowledge bases, multiple versions could be used to implement new search algorithms. In certain fault-tolerant computing environments it might be possible to see both the file system and the state of the program as a shadow paging database. The possibilities for interesting applications are numerous and largely unexplored. It has not been possible to support multiple independently updatable database versions efficiently with existing databases.

Chapter 12

Write Optimizations

In most log-based database systems, the mapping between logical and physical pages is fixed. A fixed mapping has the advantage that maintaining a page table is not required, and it allows efficient clustering of related logical pages. On the other hand, in systems with a fixed mapping, each modified data page must eventually be written back to its original location. Since modifications are usually scattered throughout the entire database, each modification will typically require a seek to the desired location on disk. This need is not removed by delaying or batching writes (except in hot spots), since the database is usually very large compared to the number of pages modified, and thus the density of writes is still low.

Shadow paging allows the database system to choose where to write a modified page. The system can choose to write all modified pages in a contiguous area on disk, or just near each other. This allows many pages to be written with a single seek. Alternatively, the system can use a "write anywhere" strategy[1].

A similar idea is used in log-based file systems [79, 90, 95], where long writes are used to improve write performance.

12.1 Potential Speedup from Sequential Writes

A typical low-cost disk currently has about 15 ms average access time (t_a) and 4 MB/s sustained data transfer rate (R). The time required to transfer

[1] "Write anywhere" is a strategy supported by some I/O subsystems. The idea is that the I/O subsystem is given a block, it will write it anywhere on the disk, and return the address of the block.

S	512	1024	2048	4096	8192
F_∞	124	62	32	16	9

Table 12.1: Limit of potential speedup F as $N \to \infty$ for different page sizes S.

a page of data $t_x = \frac{S}{R}$, where S is the size of the page in bytes. If a seek is required, the time to read or write S bytes is $t_a + \frac{S}{R}$. If many adjacent pages are read or written, no physical seek is needed except for the first page. Similarly, if one page is read or written, the next is skipped, and the next written, no seek is needed for the latter page, but extra transfer (rotational latency) time must be counted for the skipped page.[2]

When a seek is required for each page, the time required to transfer N pages of size S is $N(t_a + \frac{S}{R})$. When a seek is required for the first page only, the time is $t_a + N\frac{S}{R}$. The maximum speedup factor F that can be achieved by doing the writes sequentially is

$$F = \frac{N(t_a + \frac{S}{R})}{t_a + N\frac{S}{R}}. \tag{12.1}$$

As $N \to \infty$, the limit on speedup

$$F_\infty = \lim_{N\to\infty} F = 1 + t_a \frac{R}{S}. \tag{12.2}$$

Figure 12.1 shows the speedup factor F for various page sizes and write lengths for a typical low-cost disk ($t_a = 15\,\mathrm{ms}$ and $R = 4\,\mathrm{MB/s}$). Table 12.1 shows the limit of potential speedup when $N \to \infty$.

The maximum speedup can be achieved if a sufficiently large contiguous area can be found in the database. This is often not the case in practice, and the part of the database with the highest density of free pages should be used for writing. The pages which are be skipped between writes must be counted in the transfer time. This can be represented by a skip factor s,

[2]There are practical problems in writing to nearby sectors efficiently. In theory this should be very efficient with track-buffered disk drives. The SCSI Write Cache feature provides everything that is needed. The trouble is that most SCSI disks either do not support Write Cache at all, or support it only enough that sales literature can claim that it is supported (e.g., IBM 0662 drives are able to cache one write only, which renders the write cache completely unusable).

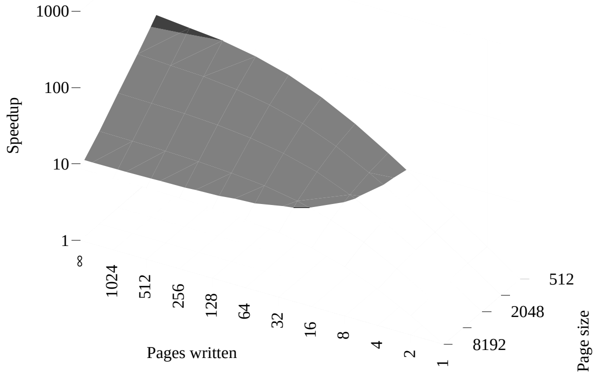

Figure 12.1: Speedup factor F for different page sizes S and transfer lengths N.

which is the amount by which the transfer time must be multiplied. If the area is contiguous, the skip factor is 1. If 25 percent of disk space is free and it is evenly distributed (the worst case), $s = 4$. Even now, the maximum speedup would be by a factor of 31 for 512 byte pages and by a factor of 4 for 4096 byte pages.

12.2 Making Writes Sequential

The idea in shadow paging is to allocate a new physical page whenever a page is modified. The new page number is kept in the incremental page table until the transaction is committed.

Allocation of physical page numbers can be delayed by introducing the concept of *virtual page numbers* (note that these have nothing to do with logical page numbers). A virtual page number is an identifier for a page known by the transaction and the cache manager. The transaction may access a page using the virtual page number, and the cache will return the appropriate page. Virtual pages do not usually have a physical page associated with them; they only exist in the cache.

A virtual page number is allocated whenever a page is first modified. Typically this is done by specifying an existing page number to the cache manager. The cache manager will then allocate a unique virtual page number and rename the page in its memory to use the virtual number. The old physical page is effectively removed from the cache. Unless snapshots (Section 11) are used, no-one will ever access the old page except in the case that the current transaction is aborted.

Virtual page numbers cannot be stored in the page table, as they are only temporary identifiers in volatile storage. Before a transaction can commit, actual physical pages must be allocated for its virtual pages (this is called the *realization* of the virtual pages). Since many transactions are usually committed as a batch, a large number of modifications can be combined, and all virtual pages of the committing transactions can be realized at the same time. The system can now allocate contiguous disk pages and write them sequentially.

Modification of page table pages is also done using virtual pages. When a commit batch is being processed, virtual pages are created for all page table pages that are going to be modified. The virtual page table pages are then realized together with modified data pages, and the actual physical page numbers are stored in the page table pages. Finally, both modified data and page table pages are flushed to disk, and when they have all been written, the page table pointer is updated atomically to point to the new page table. It is possible to allocate the new physical pages on multiple disks and use striping[3].

The cache is allowed to allocate physical pages for virtual pages even before realization. This is desirable if there are large update transactions which create very many virtual pages. In such cases the cache can make long enough writes to achieve most of the maximum speedup. Transactions will still access those pages using the virtual page number, and the new physical page numbers will be internal to the cache until the virtual pages are realized (for such pages, no pages are allocated during realization as they already have a physical page). The virtual page number can be released after the transaction holding the number has terminated.

When mirroring is used for media recovery (Section 14.1), two physical pages are allocated for each virtual page. The physical pages are required to be on two separate disks (on disks with independent failure modes). It is possible to allocate two contiguous regions, one on each disk, or to use more disks, whichever is more convenient.

It should be noted that this optimization makes conventional clustering

[3]Striping that a large write is done using multiple disks in parallel. Each disk will write only a small portion of the data. I/O bandwidth is multiplied by the number of disks.

between logical pages impossible, as no attempt is made to keep the logical-to-physical mapping linear. An alternate approach to clustering is presented in Section 13.

12.3 Disk Space Fragmentation

In order to be able to do sequential writes, the system needs to find fairly large contiguous regions. Contiguous regions are also important for large pages when multiple page sizes are used. Pages are, on the other hand, freed in fairly random order. Without countermeasures, this quickly leads into severe free space fragmentation, and it becomes impossible to find contiguous free areas.

There are several possible approaches to prevent fragmentation.

- The allocator can try to avoid using large free regions; it can instead try to fill the smaller slots first.
- It is possible to divide the disk space into smaller segments, always write sequentially to an empty segment, and aggressively free fragmented segments by copying their data to the output. This is very similar to what is done in log-structured file systems [79, 90].
- One can reserve different parts of the disk for different page sizes. This helps in finding space for the larger pages.

Dealing with fragmentation is still under research, and new ideas are being investigated.

Chapter 13

Clustering

The write optimizations in Section 12 prevent conventional clustering of related logical pages. Because it is desirable to be able to write a page to a new location without constraints, earlier clustering algorithms for shadow paging which attempted to keep the logical-to-physical mapping approximately linear [43, 60, 88] are not applicable.

There are three important applications of clustering:

- Sequential scans
- Reading/writing large objects
- Certain types of joins

13.1 Sequential Scans and Large Objects

Clustering problems are at worst when reading large multi-megabyte objects or scanning a large table. The time required to read an object (or table) of S bytes in N_{parts} parts is

$$t(S, N_{parts}) = \lceil \frac{S}{N_{psz}} \rceil t_x + N_{parts} t_a.$$

It is assumed that in systems with a fixed mapping the object can always be stored as a contiguous region ($N_{parts} = 1$), although this optimal case is not always possible in practice due to free space fragmentation. It is also assumed that each page of the object in a shadow paging system is stored in a separate location ($N_{parts} = \lceil \frac{S}{N_{psz}} \rceil$), which is the worst case.

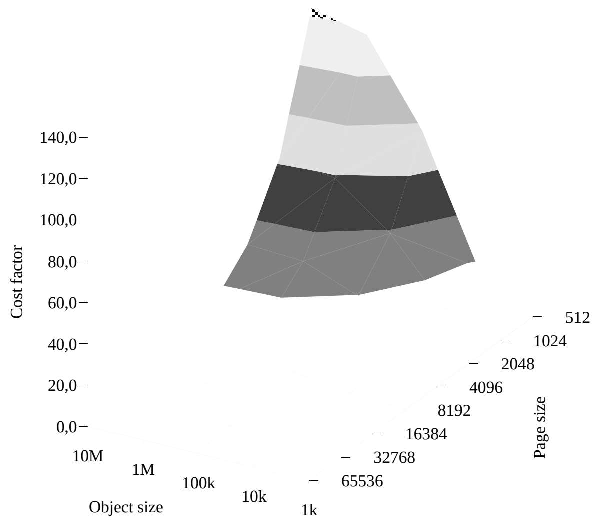

Figure 13.1: Worst-case fragmentation cost factor C for different page sizes N_{psz} and object sizes S.

The worst-case fragmentation cost factor C can be defined as

$$C = \frac{t(S, \lceil \frac{S}{N_{psz}} \rceil)}{t(S, 1)}. \qquad (13.1)$$

This expresses the worst-case overhead due to fragmentation, with the value 1 meaning no overhead, value 2 meaning that I/O time is doubled, etc. Values for the fragmentation cost factor are shown in Figure 13.1. In practice the cost factors are smaller, partly because it becomes impossible due to free space fragmentation to find sufficiently large contiguous free areas to store the object in one piece, partly because there might be locality in a shadow paging system due to correlation of update times, and partly because it is often more efficient to store a large object divided on several disks instead of just one contiguous region on one disk.

It can be seen that fragmentation can be helped by increasing the page size. On the other hand, as seen from Table 12.1, the write optimizations become less effective when the page size increases. Also, since the entire page needs to be written to a new location if any part of it changes, transfer time increases with page size.

The solution is to use variable-size pages (multiples of the basic block size). The size of each logical page can be stored in the page table where it is easily accessible whenever needed. The basic block size should be chosen as small as possible to make small updates as fast as possible where clustering is not needed, and larger pages should be used for storing large objects or tables which are often accessed sequentially. Variable-size pages allow tuning the sequential read versus small update performance tradeoff on a per-object basis to best suit the needs of each application.

Page size is optimal when the average cost of an access is minimal. To express this formally, let $p_r(S)$ be the probability that a particular access is a read of size S and $p_w(S)$ be the probability that the access is a write of size S. The probability that the access is a read is $p_r = \sum_{S=0}^{\infty} p_r(S)$, and the probability that it is a write is $p_w = \sum_{S=0}^{\infty} p_w(S)$. By definition, $p_r + p_w = 1$.

The cost of a read in terms of I/O time is

$$c_r(S, N_{psz}) = \lceil \frac{S}{N_{psz}} \rceil (t_a + \frac{N_{psz}}{R}).$$

Writes are cheaper due to batching of writes. The cost of a write (ignoring the initial seek as it is divided between many writes) is

$$c_w(S, N_{psz}) = \lceil \frac{S}{N_{psz}} \rceil s \frac{N_{psz}}{R}.$$

The average cost of an arbitrary I/O operation is

$$\begin{aligned} c_{io}(N_{psz}) \quad = \quad & \sum_{S=0}^{\infty} p_r(S) c_r(S, N_{psz}) + \\ & \sum_{S=0}^{\infty} p_w(S) c_w(S, N_{psz}). \end{aligned} \tag{13.2}$$

The optimal value for the page size is the value at which $c_{io}(N_{psz})$ is minimum.

13.2 Joins

It is fairly common to store two or more tables using the same clustering index. This permits the tables to be joined very efficiently on the attribute on which they are clustered.

Due to the dynamic nature of most index structures, such as a B-tree, leaf nodes which contain adjacent key values cannot easily be stored in nearby locations on disk. The B-tree nodes, on the other hand, might consist of several physical blocks.

Shadow paging permits using a clustering index the same way as log-based databases. Each B-tree node is a page. Data from several tables will be clustered within each page. Different index pages will not be clustered with respect to each other. The situation is essentially identical to that in log-based databases.

Chapter 14

Media Recovery

Log-based database systems can use a dump plus log entries to reconstruct the database state before crash [33]. However, since shadow paging does not have a log, this option cannot be used for media recovery.

14.1 Mirroring

Mirroring [33, 78] is a technique where the same data is stored on two disks, and if one disk fails the other can still be used. Mirroring also has the advantage that reads can use either copy of the data, reducing read latency. However, writes must be made to both copies of the data, increasing write latency. Additionally, disk space requirements are doubled. Doubly distorted mirrors [78] are an optimization which reduces the cost of writes.

Mirroring works very nicely with shadow paging. All references to physical pages are made to contain two page numbers. When a physical page is allocated, two pages on different disks are allocated. Both pages are written and contain the same data. When read, either page can be read. Two page numbers are stored in page table entries, higher-level page table entries, in the page table pointer, and everywhere else where a physical page number is stored. The page table pointer is stored on two disks.

Mirroring is used together with the write optimizations of Section 12. While the transaction is active, it refers to the page using a virtual page number. When it commits, it realizes its virtual pages. The system then allocates two physical pages on different disks for each virtual page. The result is typically two nearly contiguous regions on two disks. It is not fixed which pairs of disks contain each others data – it is sufficient to pick any two disks with enough free space (things such as the average load or the current position of the read/write head can be used for selecting the two disks). The two nearly contiguous regions are then written. (Alternatively, a write anywhere strategy could be used for each individual page.)

Mirroring with shadow paging gets all the benefits of doubly distorted mirrors and is simple to implement. It works very nicely with the write optimizations. It is flexible, and does not require disks of the same size. It is even possible to do recovery after a disk crash without a spare part: the page copies that were on the damaged disk are relocated to other disks in the system. The only constraint is that the two copies of the page cannot be on the same disk or on disks with common failure modes.

The implentation of mirroring is simplified by the fact that any data that is being written is not part of the permanent database. Should the system fail in the middle of a write, it does not matter if just one of the copies got written since the page is not part of the database anyway. Without shadowing, however, the system must deal with situations where the two copies of a mirrored page contain inconsistent data.

14.2 Recovery Procedures

Recovery from a single bad block can be done by reading the data from the other copy of the page, and virtualizing the page. The page will be realized and its new address updated to the page table as part of the next commit batch. Note that it is sufficient to virtualize the failed side of the page; the other copy need not be moved. However, supporting this may not be worthwhile.

If permanent snapshots are supported, the page must be updated synchronously in all snapshots referring to it. The snapshots should still share the page after it has been moved.

Recovery from a disk crash can be done by traversing the page table, and processing each page which has a copy on the failed disk as was described above for a single page.

The exact details of recovery (e.g., how to recover the page in the presence of snapshots, and how to continue normal operation during recovery) are still under research.

14.3 RAID

Conventional RAID [37, 80, 99, 100] does not work well with shadow paging. The problem is that all pages are written to newly allocated locations. This means that to update the parity block, the old value of the data page must probably be read from the disk so that it can be removed from the parity block. Alternatively all the pages using that parity block would have to be available. All visible implementation strategies lead to unacceptable overheads.

It appears to be possible to use a novel completely dynamic RAID algorithm resembling the mirroring algorithm above. Tero Kivinen is writing about the RAID implementation [in preparation].

Chapter 15

Two-Phase Commit

Lampson and Sturgis [48, 49] have presented a method for doing two-phase commit with shadow paging using the intentions list paradigm. The method presented here resembles their method, but is more general.

Two-phase commit is a common method for implementing transactions in distributed databases. Two-phase commit requires that the changes made by a transaction can be saved so that they survive a possible crash but can still be undone.

The basic idea is to store the incremental page table on disk in the first phase of commit, and leave a pointer to the incremental page table in the page table pointer. If a crash occurs, the incremental page table can be recovered from the copy on disk and the coordinator queried about the fate of the transaction. In the second phase of commit, the transaction is committed normally and the incremental page table on disk is freed (atomically, in the same commit batch).

Fine-granularity locking requires special consideration with two-phase commit. It must be possible to store on disk the fine-granularity locking state of the transaction. This is implemented by having each fine-granularity type implement a method which generates a linear (byte array) representation of the fine-granularity changes of that type made by the transaction. The representation must be sufficient that the fine-granularity modifications and locks held by the transaction can be restored using the representation. The `Transaction` class provides a method which returns the combined state of all fine-granularity types used by the transaction. It also has a method which can be used to restore the state of the transaction from the linear representation; it splits the representation into parts supported by each fine-granularity type and passes the parts on to individual fine-granularity types.

15.1 First Phase of Commit

When a node is requested to do the first phase of commit, all shared locks are released immediately. The incremental page table of the transaction, the linearized fine-granularity changes, and any modified pages are written to disk. The saved information must be sufficient to reconstruct the state of the transaction after a crash, and to identify the coordinator of the transaction and the transaction within the coordinator. During processing of the next commit batch, a pointer to the saved information on disk will be added to the page table pointer (if there are very many transactions doing two-phase commit, it may be necessary to use extension pages to store all the pointers). When the page table pointer has been stored on disk, the system reports "ready" to the coordinator.

15.2 Second Phase

The second phase is executed like any transaction commit. The pointer to the saved information is removed from the page table pointer during the commit and the disk space is freed.

If the coordinator decides to abort the transaction, the pointer to the saved information is removed from the page table pointer during the next commit batch and the disk space is freed. Otherwise the abort is done normally.

15.3 Crash Recovery

During crash recovery, the system must read the saved information of partially processed global transactions from disk. The pages mentioned in saved incremental page tables must be counted as used when extracting the free list from the page table. The state of global transactions is restored from the saved information (including the incremental page table, fine-granularity modifications, and exclusive locks). Normal transaction processing can begin when the exclusive locks held by recovered global transactions have been restored. The system then queries the coordinator of each global transaction about the fate of the transaction.

The coordinator of a global transaction must keep enough information to answer queries about the fate of the transaction, e.g. by keeping a list of global transactions that have been partially (phase one) committed but that have not yet been reported fully committed by all other machines. The list of active global transactions can be kept in the page table pointer, with extension additional pages if necessary.

Chapter 16

Miscellaneous Optimizations

16.1 Page Table Translation Lookaside Buffer

Translating a page number through the page table costs several hundred instructions assuming a buffer cache lookup can be done in about a hundred instructions. The total overhead for a TPC-B[1] style transaction can be several thousand instructions.

The translation cost can be reduced dramatically by using a specialized data structure to cache recently used mappings. One possibility is to use an array of mappings $\langle L, P\rangle$ as a hash table (L is a logical page number and P is the corresponding physical page number). The logical page number is hashed into the table using the lower bits of the page number (if the size of the array is a power of two, this can be done with a single bitwise-and instruction). When a translation from a logical to a physical page number is needed, the appropriate slot of the table is checked first, and if present, the mapping there is used. Otherwise the translation is done using the page table, and the mapping is copied to the hash table. Page table modifications also update mappings in the hash table.

This optimization almost eliminates the translation cost for most accesses. Higher hit rates for the lookaside buffer can be achieved by using an N-way set-associative architecture [96].

[1]TPC-B [31] is a benchmark defined by the Transaction Processing Council. It simulates a large bank and uses small Debet/Credit type transactions.

16.2 Storing Write Hotspots in the Page Table Pointer

Many databases contain small write-intensive hotspots, such as counters for unique identifiers. It is possible to avoid frequent writes of the data pages containing those hotspots by relocating the hotspots in the page table pointer. The page table pointer is written anyway during every commit batch, and any data stored there gets written to disk for free. The available space is limited, but the savings can be considerable in certain types of applications.

Hotspots can probably be detected automatically. For example, the cache can collect statistics on which pages are modified very frequently. When a frequently modified page has been identified, more detailed statistics are collected on higher levels to identify the hotspot object(s) on the page.

Relocation to the page table pointer can be implemnted by storing the identifiers of objects stored there in a hash table, and whenever an object is accessed, first checking from the hash table if that object is stored in the page table pointer instead of the location indicated by the object identifier.

Part III

Conclusion

Chapter 17

Conclusion and Further Research

Solutions have been presented to all the major problems with shadow paging that have been mentioned in the literature. With these extensions, shadow paging seems to satisfy all the requirements for an industrial-strength database system.

It is clear that shadow paging has many desirable properties. Recovery is simple, transaction rollback is fast, the write optimizations contribute to reasonable performance, mirroring performs extremely well, no logs are needed which simplifies both the implementation and administration, and snapshots allow read-only transactions to be run without interference with other transactions and allow efficient on-the-fly multi-level incremental dumping.

However, there are still many open questions.

- The force policy is used for buffer management. That is, all pages modified by a transaction must be written to disk before the transaction can commit. It is not clear how much this really affects throughput. Even in log-based systems each modified page must eventually be written to disk, and since the density of writes is typically low compared to the size of the database, the total I/O may not be affected very much by the force policy, except in hot-spots. The write optimizations may offset the extra writes. Jim Gray has suggested using battery-backed-up memory [personal communication], but the issue has not yet been looked into.

- The time needed to commit a transaction is longer than in log-based databases. The reason is that more data needs to be written to disk (force policy, page table writes, and page table pointer writes). Waiting for the next commit batch also causes a short delay. The difference may be some tenths of a second.

- Update transactions of greatly varying sizes may intolerably slow down commit batches. In general, processing of very large transactions is somewhat awkward and may sometimes require locking of the entire table or index. These are probably not serious problems in most environments, but there are situations where the current solutions may not work very well.

- The implementation of fine-granularity transactions somewhat complicates normal transaction processing because changes need to be kept separately. This also causes some overhead (although in main memory databases "shadow updating" [55] has been reported to perform quite well). Preliminary benchmarks using B-trees and TPC-B type transactions suggest that the overhead is not significant.

- Even though solutions have been presented for clustering, there are some types of applications (a mixture of random updates and frequent sequential reads) where the performance may not be very good.

- Media recovery is quite different from log-based systems since there is no log which could be used for media recovery. Mirroring works very nicely with shadow paging (it is possible to get all the benefits of doubly distorted mirrors [78] using normal disk controllers and with less overhead). RAID is quite different from conventional RAID subsystems, but can be implemented efficiently.

- Two-phase commit requires one commit batch to do phase one and another to do phase two. The overhead is not very high, but may still be too much in some applications. The severity of the problem is not yet known.

- The write optimizations rely on finding sufficiently large contigous regions of free space. However, pages are freed in random order. This leads to fragmentation of disk space unless special countermeasures are taken. One approach is to use suitable heuristics in the allocator; another is to move pages to combine small free regions into larger regions.

Fragmentation is not a very serious problem if the disk drives used support track buffering; however, eliminating it becomes very important when skipping over a few sectors is expensive. Disk space allocation is still under research.

- The log is sometimes used for other purposes besides recovery (e.g., auditing). If needed, it is possible to maintain a log for auditing purposes in normal shadowed storage. No special recovery mechanisms will be needed for the log. Also, writes to the log will be included in the sequential write generated by the write optimizations, and the performance overhead should be very small.

All of these issues are still more or less open. Better solutions are likely to be found for many of them. The final question, whether the good aspects of shadow paging (e.g. write optimizations, snapshots) offset the problems, is still open and probably will not be answered until the ideas have been fully tried out in practice.

There are some important questions that have not been addressed in this thesis. These include nested transactions, partial rollbacks, and maintaining a remote hot standby. These issues will be addressed in future research.

The ideas presented in this thesis are being implemented in the **Shadows** database system prototype being built at Helsinki University of Technology, Finland. It is too early to do real performance evaluations; however, preliminary results have been promising.

Bibliography

[1] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Very Large Data Bases*, pages 86–91, 1980.

[2] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *ACM SIGMOD*, pages 408–417, 1989.

[3] R. Agrawal and M. J. Carey. The performance of concurrency control and recovery algorithms for transaction-oriented database systems. *IEEE Database Engineering*, 4:118–127, 1985.

[4] R. Agrawal and D. J. DeWitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, 1985.

[5] Anon. et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.

[6] R. A. Baeza-Yates. Expected behaviour of B^+-trees under random insertions. *Acta Informatica*, 26:439–471, 1989.

[7] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *Computing Surveys*, 23(3):269–317, 1991.

[8] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[9] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[10] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[11] P. A. Bernstein and N. Goodman. Multiversion concurrency control – theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[13] R. G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[14] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System*, pages 447–462. Morgan Kaufmann, 1992.

[15] E. E. Chang and R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *ACM SIGMOD*, pages 348–357, 1989.

[16] J. R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented darabases. In *ACM SIGMOD*, pages 22–31, 1991.

[17] Y. Choueka, A. S. Fraenkel, and S. T. Klein. Compression of concordances in full-text retrieval systems. In *ACM SIGIR*, pages 597–613, 1988.

[18] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[19] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *ACM SIGIR*, pages 405–411, 1990.

[20] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[21] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD*, pages 1–8, 1984.

[22] M. H. Eich. Main memory database recovery. In *Fall Join Computer Conference*, pages 1226–1232, 1986.

[23] M. H. Eich. A classification and comparison of main memory database recovery techniques. In *Data Engineering*, pages 332–339, 1987.

[24] T. Eiter, M. Schrefi, and M. Stumptner. Sperrverfahren für B-bäume im vergleich. *Informatik Spektrum*, 14:183–200, 1991.

[25] C. S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[26] C. S. Ellis. Concurrent search and insertion in avl trees. *IEEE Transactions on Computers*, 29:811–817, 1980.

[27] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[28] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Treiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–6633, 1976.

[29] H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD*, pages 249–259, 1987.

[30] J. Gray, 1993. Personal communication.

[31] J. Gray. *The Benchmark Handbook.* Morgan Kaufmann, second edition, 1993.

[32] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *Computing Surveys*, 13(2):223–243, 1981.

[33] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[34] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[35] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4):287–317, 1983.

[36] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, 1986.

[37] R. Y. Hou and Y. N. Patt. Comparing rebuild algorithms for mirrored and RAID5 disk arrays. In *ACM SIGMOD*, pages 317–326, 1993.

[38] ISO and ANSI SQL3 Working Draft – February 5, 1993.

[39] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *ACM PODS*, pages 235–246, 1989.

[40] B. Kähler and O. Risnes. Extending logging for database snapshot refresh. In *Very Large Data Bases*, pages 389–398, 1987.

[41] A. M. Keller and G. Wiederhold. Concurrent use of B-trees with variable-lenght entries. *ACM SIGMOD Record*, 17(2):89–90, 1988.

[42] J. Kent, H. Garcia-Molina, and J. Chung. An experimental evalution of crash recovery mechanisms. In *ACM PODS*, pages 113–121, 1985.

[43] J. M. Kent. *Performance and Implementation Issues in Database Crash Recovery.* PhD thesis, Princeton University, 1985.

[44] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[45] D. Knuth. *The Art of Computer Programming*, volume 3, chapter 6.2.4 Multiway Trees, pages 471–479. Addison-Wesley, 1973.

[46] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1986.

[47] H. T. Kung and P. L. Lehman. A concurrent database manipulation problem: Binary search trees. *ACM Transactions on Database Systems*, 3:339–353, 1980.

[48] B. W. Lampson. *Distributed Systems – Architecture and Implementation*, chapter 11. Atomic Transactions, pages 246–265. Springer-Verlag, 1983.

[49] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, April 1979.

[50] S. D. Lang, J. R. Driscoll, and J. H. Jou. Batch insertion for tree structured file organizations – improving differential database representation. *Information Systems*, 11(2):167–175, 1986.

[51] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Fall Joint Computer Conference*, pages 380–389, 1986.

[52] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[53] F. E. Levine and C. Mohan. Method for concurrent record access, insertion, deletion and alteration using an index tree. United States Patent 4,914,569, IBM, April 1990.

[54] F. E. Levine and C. Mohan. Method and apparatus for concurrent modification of an index tree in a transaction processing system utilizing selective indication of structural modification operations. United States Patent 5,123,104, IBM, June 1992.

[55] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Data Engineering*, pages 117–124, 1993.

[56] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *ACM SIGMOD*, pages 53–60, 1986.

[57] C.-C. Liu and T. Minoura. Effect of update merging on reliable storage performance. In *Data Engineering*, pages 208–213, 1986.

[58] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *ACM SIGMOD*, pages 351–360, 1992.

[59] D. B. Lomet. Key range locking strategies for improved concurrency. In *Very Large Data Bases*, pages 655–664, 1993.

[60] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, 1977.

[61] N. A. Lynch. Multilevel atomicity – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, 1983.

[62] U. Manber and R. E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, 1984.

[63] G. Matsliach. Performance analysis of file organizations that use multi-bucket data leaves with partial expansions. In *ACM PODS*, pages 164–180, 1991.

[64] D. A. Menasće and O. E. Landes. On the design of a reliable storage component for distributed database management systems. In *Very Large Data Bases*, pages 365–375, 1980.

[65] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Very Large Data Bases*, pages 392–405, 1990.

[66] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[67] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management methof using write-ahead logging. In *ACM SIGMOD*, pages 371–380, 1992.

[68] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. Technical Report RJ 8016, Data Base Technology Institute, IBM Almaden Reserch Center, March 1991.

[69] C. Mohan and I. Narang. An efficient and flexible method for archiving a data base. In *ACM SIGMOD*, pages 139–146, 1993.

[70] C. Mohan, R. Obermark, and K. Treiber. Concurrently applying redo records to backup database in a log sequence using single queue server per queue at a time. United States Patent 5,170,480, IBM, December 1992.

[71] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *ACM SIGMOD*, pages 124–133, 1992.

[72] C. Mohan, K. Treiber, and R. Obermark. Algorithms for the management of remote backup data bases for disaster recovery. In *Data Engineering*, pages 511–518, 1993.

[73] Y. Mond and Y. Raz. Concurrency control in B^+-trees databases using preparatory operations. In *Very Large Data Bases*, pages 331–334, 1985.

[74] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981. Available as MIT/LCS/TR-260.

[75] M. H. Nodine and S. B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Very Large Data Bases*, pages 83–94, 1990.

[76] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *ACM PODS*, pages 192–198, 1991.

[77] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *ACM PODS*, pages 170–176, 1987.

[78] C. U. Orji and J. A. Solworth. Doubly distorted mirrors. In *ACM SIGMOD*, pages 307–316, 1993.

[79] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, 1989.

[80] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD*, pages 109–116, 1988.

[81] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Concurrent group-update algorithms. Technical Report 1992/TKO-B87, Helsinki University of Technology, Finland, 1992.

[82] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Batch updates and concurrency in B-trees. Technical Report 1993/TKO-B106, Helsinki University of Technology, Finland, 1993.

[83] C. A. Polyzois and H. Garcia-Molina. Evaluation of remote backup algorithms for transaction processing systems. In *ACM SIGMOD*, pages 246–255, 1992.

[84] C. Pu. On-the-fly, incremental, consistent reading of entire databases. In *Very Large Data Bases*, pages 369–375, 1985.

[85] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989. Revised June 1990.

[86] W. Pugh. A skip list cookbook. Technical Report UMIACS-TR-89-72.1 (CS-TR-2286.1), Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989. Revised June 1990.

[87] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[88] A. Reuter. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Transactions on Software Engineering*, SE-6(4):348–356, 1980.

[89] A. Reuter. Performance analysis of recovery techniques. *ACM Transactions on Database Systems*, 9(4):526–559, 1984.

[90] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles*, pages 1–15, 1991.

[91] D. J. Rosenkrantz. Dynamic database dumping. In *ACM SIGMOD*, pages 3–8, 1978.

[92] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Very Large Data Bases*, pages 337–346, 1989.

[93] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 33:275–296, 1986.

[94] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Data Engineering*, pages 452–462, 1989.

[95] M. Seltzer and M. Stonebraker. Transaction support in read optimized and write optimized file systems. In *Very Large Data Bases*, pages 174–185, 1990.

[96] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, 1982.

[97] V. Srinivasan and M. J. Carey. Performance of B-tree concurrency control algorithms. In *ACM SIGMOD*, pages 416–425, 1991.

[98] V. Srinivasan and M. J. Carey. Performance of on-line index construction algorithms. In *EDBT*, pages 293–309, 1992.

[99] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of XPRS. In *Very Large Data Bases*, pages 318–330, 1988.

[100] M. Stonebraker and G. A. Schloss. Distributed RAID – a new multiple copy algorithm. In *Data Engineering*, pages 430–437, 1990.

[101] R. F. van der Lans. *The SQL Standard – A Complete Reference*. Academic Service and Prentice Hall, 1989.

[102] J. S. M. Verhofstad. Recovery techniques for database systems. *Computing Surveys*, 10(2):167–195, 1978.

[103] T. Ylönen. An algorithm for full-text indexing. Technical Report 1992/TKO-B75, Helsinki University of Technology, Finland, 1992.

[104] T. Ylönen. An algorithm for full-text indexing. Master's thesis, Laboratory of Information Processing Science, Helsinki University of Technology, Finland, 1992.

[105] T. Ylönen. Concurrent shadow paging: A new direction for database research. Technical Report 1992/TKO-B86, Helsinki University of Technology, Finland, 1992.

[106] T. Ylönen. Write optimizations and clustering in concurrent shadow paging. Technical Report 1993/TKO-B99, Helsinki University of Technology, Finland, 1993.

[107] T. Ylönen, T. Kivinen, H. Suonsivu, and T. Männistö. Concurrent shadow paging: Snapshots, read-only transactions, and on-the-fly multi-level incremental dumping. Technical Report 1993/TKO-B104, Helsinki University of Technology, Finland, 1993.

[108] T. Ylönen, E. Soisalon-Soininen, and T. Kivinen. Concurrent shadow paging: Fine-granularity locking with support for extended lock modes

and early releasing of locks. Technical Report 1993/TKO-B112, Helsinki University of Technology, 1993.

[109] T. Ylönen and H. Suonsivu. A multi-user shadow paging transaction manager on Mach 3.0. In J. Helander and H. Saikkonen, editors, *Distributed Operating Systems and Mach: Proceedings of the Spring 1992 Graduate Seminar on Distributed Systems*. Helsinki University of Technology, Finland, 1993. Available as report 1993/TKO-C61.

[110] B. Zhang and M. Hsu. Unsafe operations in B-trees. *Acta Informatica*, 26:421–438, 1989.

Index