SYSTEM SUPPORT FOR SOFTWARE FAULT TOLERANCE IN HIGHLY AVAILABLE DATABASE MANAGEMENT SYSTEMS

by

Mark Paul Sullivan

Memorandum No. UCB/ERL M93/5

13 January 1993

COVER PAGE

SYSTEM SUPPORT FOR SOFTWARE FAULT TOLERANCE IN HIGHLY AVAILABLE DATABASE MANAGEMENT SYSTEMS

by

Mark Paul Sullivan

Memorandum No. UCB/ERL M93/5

13 January 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

System Support for Software Fault Tolerance in Highly Available Database Management Systems

by

Mark Paul Sullivan

Abstract

System Support for Software Fault Tolerance in Highly Available Database Management Systems

by

Mark Paul Sullivan

Doctor of Philosophy in Computer Science
University of California at Berkeley
Professor Michael Stonebraker, Chair

Today, software errors are the leading cause of outages in fault tolerant systems. System availability can be improved despite software errors by fast error detection and recovery techniques that minimize total downtime following an outage. This dissertation analyzes software errors in three commercial systems and describes the implementation and evaluation of several techniques for error detection and fast recovery in a database management system (DBMS).

The software error study examines errors reported by customers in three IBM systems programs: the MVS operating system, the IMS DBMS, and the DB2 DBMS. The study classifies errors by the type of coding mistake and the circumstances in the customer's environment that caused the error to arise. It observes a higher availability impact from addressing errors, such as uninitialized pointers, than software errors as a whole. It also details the frequencies and types of addressing errors and characterizes the damage they do.

The error detection work evaluates the use of hardware write protection both to detect addressing-related errors quickly and to limit the damage that can occur after a software error. System calls added to the operating system allow the DBMS to *guard* (write-protect) some of its internal data structures. Guarding DBMS data provides quick detection of corrupted pointers and similar software errors. Data structures can be guarded as long as correct software is given a means to temporarily unprotect the data structures before updates. The dissertation analyzes the effects of three different update models on performance, software complexity, and error protection.

To improve DBMS recovery time, previous work on the POSTGRES DBMS has suggested using a storage system based on no-overwrite techniques instead of write-ahead log processing. The dissertation describes modifications to the storage system that improve its performance in environments with high update rates. Analysis shows that, with these modifications and some non-volatile RAM, the I/O requirements of POSTGRES running a TP1 benchmark will be the same as those of a conventional system, despite the POSTGRES

force-at-commit buffer management policy. The dissertation also presents an extension to POSTGRES to support the fast recovery of communication links between the DBMS and its clients.

Finally, the dissertation adds to the fast recovery capabilities of POSTGRES with two techniques for maintaining B-tree index consistency without log processing. One technique is similar to shadow paging, but improves performance by integrating shadow meta-data with index meta-data. The other technique uses a two-phase page reorganization scheme to reduce the space overhead caused by shadow paging. Measurements of a prototype implementation and estimates of the effect of the algorithms on large trees show that they will have limited impact on data manager performance.

Acknowledgements

My readers, Mike Stonebraker, Ram Chillarege, Arie Segev, and John Ousterhout, had many insightful suggestions. Reading and digesting a dissertation as long as this one was a lot of work for all of them, and I appreciate their thoroughness. The work presented in this dissertation was supported in part by National Science Foundation grants MIP-8715235 and IRI-9107455.

My research advisor, Mike Stonebraker, led me to this thesis topic. His enthusiasm for database management systems and computer science research is in large part responsible for my decision to remain in graduate school after I finished my Master's degree. I have enjoyed working with him and hope I have picked up some of his savvy about how to build systems and how to identify productive research areas.

I was fortunate to be able to work at IBM Research as a graduate student. Ram Chillarege arranged for me to work with IBM and gain access to the raw software error data analyzed in Chapter Two. I have learned a lot from Ram both about how to conduct research and how to increase the impact of my work on people building real systems. From IBM, Al Garrigan, Dave Ruth, Hakan Markor, Janice Crawford, and Chris Byrne helped Ram and I to understand, gather, and present the error data presented in Chapter Two of the dissertation.

I would like to thank the XPRS group for its contributions to my research. John Ousterhout, Dave Patterson and Randy Katz gave me comments on both the presentation and early direction of the guarding work. During the XPRS retreats, discussions with Dave Lomet and Marc Weiser were especially helpful in shaping my research direction. The other graduate students in the XPRS group, especially John Hartman, Ann Drapeau, Pete Chen, Ken Shirriff, Mendel Rosenblum, and Ethan Miller, provided critical feedback.

In completing the dissertation, I have come to rely on the advice and encouragement of the Berkeley computer science community. Ramon Caceres, Chris Black, Ethan Munson, Vance Maverick, David Bacon, Lu Pan, Srinivasan Keshav, Diane Hernek, and Marti Hearst all gave both suggestions and moral support. Stuart Sechrest and Peter Danzig took me under their wings when I first arrived at Berkeley. I would not be going into computer science research now if Doug Terry had not encouraged my work when I was completing my Master's degree. Mark Noworolski, Scott Leubking and my officemate-in-law, Keith Bostic, have always had ideas for improving my work, many of which were perfectly reasonable. Jim Mott-Smith went out of his way to fix the obscure bugs in Sprite that only

seemed to arise when I ran POSTGRES on the Sprite machines. Carol Paxson's help was critical in the high-speed chase portion of the signature gathering process.

Without the work that many people have put into POSTGRES, my research would have been impossible. I would like to thank Jeff Meredith, Joe Hellerstein, Cim Taylor, Curt Kolovson, Anant Jhingran, Paul Aoki, Spyros Potamianos, Jolly Chen, Sunita Sarawagi, Greg Kemnitz, John Forrest and especially Wei Hong for their energetic discussions and their perseverance in long study sessions. Claire Mosher and Chandra Ghosh helped keep the frightening UC bureaucracy at bay.

Three people from the graduate student community have been especially helpful. Mike Olson and I developed the B-tree consistency support presented in Chapter Five and he implemented the techniques in POSTGRES. Mike's insight into POSTGRES and software design have been an important influence on the rest of my research as well. Mary Baker worked with me on fast recovery issues in POSTGRES and contributed suggestions to the client/server protocols in Chapter Four. Her advice on organization and presentation of ideas has been valuable at many points during the research that led to this dissertation. I have benefited from my officemate Margo Seltzer's clear thinking and experience in many ways while I have been at Berkeley. Many of the insights and design decisions that appear in this dissertation crystallized during long discussions with her. I hope that I am fortunate enough to work with people of this caliber when I leave Berkeley.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Failures and Data Availability

Commercial computer users expect their systems to be both highly reliable and highly available. Given a system's service specification, the system is *reliable* if does not deviate from the specification when it performs its services. The system is *available* if it is prepared to perform the services when legitimate users requests them. A fault tolerant system is one that is designed to provide high availability and reliability in spite of failures in hardware or software components of the system. Once a fault tolerant system is in production, it maintains high reliability through error detection, halting an operation rather than providing an incorrect result. Fault tolerant systems achieve high availability by recovering transient state quickly after an error is detected, minimizing down time to increase overall availability.

Traditionally, fault tolerant systems have focused on detecting and masking hardware (material) faults through hardware redundancy [41]. In today's fault tolerant systems, however, software failures, rather than hardware failures, are the largest cause of system outage [29]. Figure 1.1 compares outage distributions in three years of a five year study of Tandem Corporation's highly available systems. In the figure, outages are classified by the nature of the failure that caused the outage. Software outages are caused by failures of the operating system, database management system, or application software. Hardware outages are caused by double failures of hardware components, including microcode. Errors made by the people who manage and maintain the system are separated into operator and maintenance errors, since the system's owners controlled day-to-day operations while Tandem was responsible for routine maintenance. Environment failures include fires, floods, and power outages of greater than one hour.

Tandem's studies found that outages shifted over time from a fairly even mix across all sources to a distribution dominated by software failures. From 1985 to 1989, software went from causing 33% of outages to 62%. By 1989, the second and third largest contributors,

operations and hardware, were at fault only 15% and 7% of the time, respectively.

For Tandem, the trend is not due to worsening software quality, but to success in curtailing outages caused by hardware and maintenance failures. Overall, Tandem's systems have gradually become more reliable; the mean time between system failures has risen from 8 years to 21 years. The reliability of the hardware components from which the systems are built has increased. Hardware redundancy techniques have gone a long way in detecting and masking faults when those hardware components do wear out. The increasingly reliable hardware also needs less maintenance. When maintenance is required, many of the maintenance tasks have been automated in order to limit the errors that the maintenance engineers can make. The rate of operator errors has remained constant, but it should soon improve for some of the same reasons that maintenance error rates improved. Operator interfaces are becoming less complex, hence, operators are less likely to make mistakes. Over time, more of the tasks currently done by operators will be automated as well, which removes the opportunity for operator errors. Thus, while progress in these areas has had a noticeable impact, the growing dominance of software outages is making continued advances in non-software fault tolerance less and less important.

A second study from Tandem indicates another software-related limit to system fault tolerance [28]. Even when software does not cause the original outage, it often determines the duration of the outage. Once an outage of any sort occurs, the system must reestablish software state lost at the time of the failure. While the system is reinitializing, it is unavailable to its users. A thorough approach to improving system availability must also address software restart time.

This dissertation focuses on part of the software fault tolerance problem: improving the reliability and availability of the database management system (DBMS). The integrity and availability of data managed by a DBMS is usually an important feature of the environments in which fault tolerant systems are used. In Tandem's outage study, the DBMS accounted for about a third of the software failures (the remainder being divided between operating system, communication software, and other applications). While we focus on the DBMS, much of the work is applicable to other systems programs.

Before presenting the approach to software fault tolerance taken in the dissertation, this chapter introduces a model of errors and describes some existing software fault tolerance techniques. The model and some of the terms defined in the first section below will be used throughout the dissertation. A review of the software fault tolerance literature is in the section following the description of the error model. The final section below outlines the remainder of the dissertation.

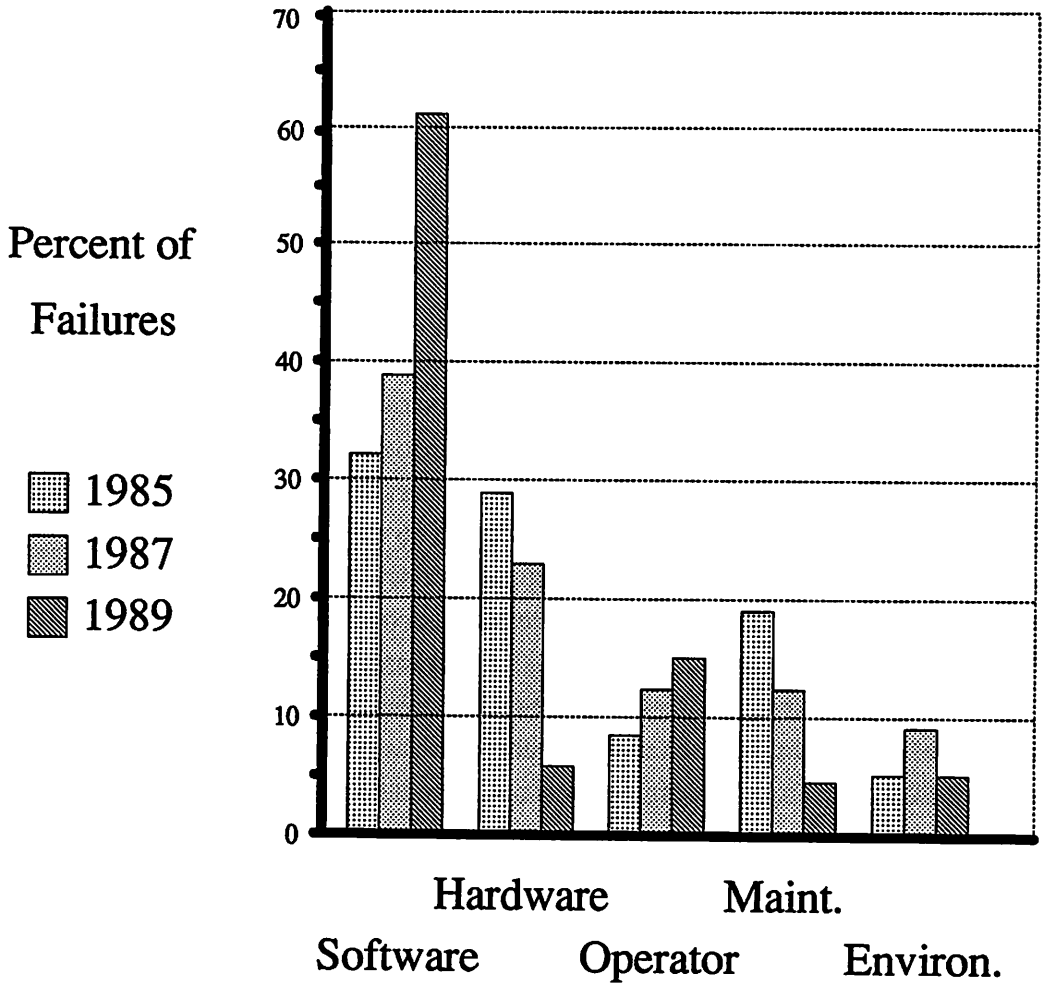

Figure 1.1: Causes of Outages in Tandem Systems. The chart represents the results of three years of a five year study. Outages are classified by the nature of the component that failed. The graph shows a dramatic shift to software as the primary cause of system outage. The bars for a given year do not sum to 100% because the causes of some outages could not be identified.

1.2 A Model of Software Errors

The software error model used in this dissertation highlights one of the significant differences between hardware and software failure modes, **error propagation**. Using redundancy, hardware components can detect their own errors and often recover without disturbing the system. Software errors, on the other hand, sometimes cause damage that is not detected immediately. The damaged system can initiate a sequence of additional software errors as it executes, eventually causing the system to corrupt permanent data or fail. Error propagation complicates software failure modes, making the code difficult to reason about, test, and debug. Reproducing propagation-related failures during debugging is difficult since error propagation can be timing dependent.

To explore software fault tolerance techniques in the DBMS, we propose a model that distinguishes between software errors based on the ways in which they propagate damage to other parts of the system. The model breaks software errors into three classes: control errors, addressing errors, and data errors. **Control errors** include programmer mistakes such as deadlock in which the point of control (the program counter) is lost or the program makes an illegal state transition. The only corruption that occurs is to the variables representing the current state of the program. Control errors can propagate only when the broken module communicates with other parts of the system. **Addressing errors** corrupt values that the faulty routine did not intend to operate on. An uninitialized pointer would be an addressing error, for example. Propagation from addressing errors is the most difficult to control since, from the standpoint of the module whose data has been corrupted, the error is "random": it happens at a time when the module designers do not expect to communicate with the faulty module. **Data errors** corrupt the values computed by the faulty routine. A data error causes the program to miscalculate or misreport a result. Like control errors, data errors can propagate only to modules related to the routine with the error. Unlike many addressing errors, the source of the corruption in a data or control error can be tracked during debugging by examining the code that is known to use the corrupted data.

In future database management systems, the impact of the cross-module error propagation caused by addressing errors may increase because of two trends in DBMS design: data manager extensibility and main memory resident databases. Extensible DBMSs include extended relational systems [67], object-oriented systems [7], and DBMS toolkits [15]. An extensible DBMS lets users or database administrators add access methods, operators, and data types to manage complex objects. Moving functionality from DBMS clients to the DBMS itself improves application performance but could worsen system failure behavior. Extensibility allows different object managers with varying degrees of trustworthiness to run together in the data manager. Every time one user on the system tries to use a new object manager or combine existing ones in a different way, there is a risk of uncovering

latent errors. Because of addressing errors, this risk is not confined to the person using the new feature; it affects the reliability and availability achieved by all concurrent users of the database.

System designers have realized for some time that DBMS performance would improve dramatically if the database resided entirely in main memory instead of residing primarily on disk (e.g. [21]). Years ago, main memory capacity was the factor limiting the appeal of main memory DBMSs. In high-end systems today, however, main memories large enough to hold many databases are available, and memory prices are dropping. Commercial systems still do not use main memory DBMSs, probably because system designers believe that data stored main memory is more likely to be corrupted by errors than data stored on disk. Corruption due to hardware and power failures can be eliminated if existing redundancy techniques based on those discussed in [41] are applied to large main memories. Operator and maintenance errors could harm data on disk as easily as data in memory. This leaves software errors as the largest remaining reliability difference between disk-resident databases memory-resident ones. In a main memory DBMS, the danger of error propagation makes addressing errors one of the most important differences in the risk to data in main memory and on disk.

1.3 Existing Approaches to Software Fault Tolerance

Current strategies for reducing the impact of software errors on systems fall into two classes: fault prevention and fault tolerance. System designers would obviously prefer not to have software errors at all than to invent techniques for tolerating them. Some software errors are prevented through modular design, exhaustive testing, and formal software verification. A survey of error prevention techniques is presented in [55]. Although most software designs incorporate one or more of these techniques, the complexity and size of concurrent systems programs such as the operating system and database management system make error prevention alone insufficient for achieving high system reliability and availability.

Since fault prevention alone is not effective, software fault tolerance techniques are used to detect and mask errors when they occur in the system. Like hardware fault tolerance, software fault tolerance is usually based on redundancy. Because software errors are usually design errors, rather than material failures, redundancy-based techniques have limited effectiveness in software. Redundant hardware components can be expected to fail independently, but software design errors often do not cause failure independently in each redundant components. Most redundant software schemes only mask software errors triggered by hardware transients and unusual events, such as interrupts, that might arrive at the redundant components at different times.

Systems that tolerate software faults usually employ either **spatial redundancy, temporal redundancy**, or a hybrid of the two. Spatial redundancy uses concurrent instances of the program running on separate processors in the hope that an error that strikes in one instance will not occur in any of the others. In temporal redundancy, the system tries to clean up any system state damaged by the error and retry the failed operation. Wulf [78] makes the distinction between spatial and temporal redundancy in a paper on reliability in the Hydra system.

N-version programming [4] is a famous spatial redundancy technique designed as a software analog of the triple modular redundancy (TMR) techniques commonly used for hardware fault tolerance. In N-Version programming, there are several versions of a program each of which is designed and implemented by a different team of programmers. The N versions run simultaneously, comparing results and voting to resolve conflicts. In theory, the independent programs will fail independently. In practice, multiple version failures are caused by errors in common tools, errors in program specification, errors in the voting mechanism, and commonalities introduced during bug fixes [75]. Furthermore, experimental work [42][64] has indicated that even independent programmers often make the same mistakes. Not surprisingly, different programmers find the same tasks difficult to code correctly. For example, different programmers often forget to check for the same boundary conditions.

Most database management systems rely on temporal redundancy to recover from software errors. Most of recovery techniques surveyed in Haerder and Reuter [33] restore the database to a transaction-consistent state in the hopes that the error does not occur. The database management system's clients then reinitiate any work aborted as a result of the failure. In [60], Randell describes a temporal redundancy method called recovery blocks. At the end of a block of code, an acceptance test is run. If the test fails, the operation is retried using an "alternate" routine. Ideally, this is a reimplementation of the routine that is simpler, but perhaps less efficient, than the original routine. Recovery blocks require fewer hardware resources than N-version programs, but may be ineffective for the same reasons as N-version programs.

Process pairing [8] is a hybrid between spatial and temporal redundancy in which an identical version of the program runs as a backup to the primary one. The primary and backup run as separate processes on different processors. In addition to masking unrepeatable software errors, process pairs reduce the availability impact of hardware errors since the primary and backup run on different processors. If a hardware error causes the processor running the primary process to fail, the backup process will take over the clients of the primary. Because only one team of programmers is required, a process pair is considerably cheaper than an N-version program. Auragen [14] used a similar scheme. Another spatial/temporal redundancy hybrid method uses redundant data in the

same address space to reconstruct data structures damaged by errors [73]. When an error is detected during an operation on the data structure, the structure is rebuilt using the redundant data and the operation is retried.

A system can only tolerate software errors if these errors are detected in the first place. The most common approach to error detection in systems programs is to lace the program with additional code that checks for errors. Sometimes these include data structure consistency checkers that pass over program data and examine it for internal consistency. By detecting errors quickly, even systems without redundant components limit the chance that minor errors will propagate into worse ones.

Unfortunately, checking for errors is expensive. No published figures are available regarding the cost of error checking in the DBMS, but run time checks for array bounds overruns in Fortran programs can double program execution time [31]. Furthermore, the checkers themselves can have software errors. Error checking is not usually done systematically. The checking code has to be maintained as the software it checks is maintained. Implementing and testing error checkers increases development cost.

1.4 Organization of This Dissertation

The dissertation makes three contributions towards the goal of improving software fault tolerance in database management systems. First, it assembles and analyzes a body of information about software errors that will be useful to software availability and reliability researchers. Second, it describes the implementation and evaluation of a mechanism for detecting addressing errors that can be used in conjunction with existing ad-hoc consistency checkers. Finally, it extends the DBMS fast recovery techniques of the POSTGRES storage system [66] in order to improve availability.

Chapter Two examines error data collected after software failures at IBM customer sites in order to improve system designers' understandings of the ways in which software causes outage. The chapter presents the results of two software error studies in the MVS operating system and the IMS and DB2 database management systems and compares these results to those of earlier software error studies.[1] Chapter Two shows that 40-55% of the errors reported in these three systems were control errors, while addressing and data errors were 25-30% and 10-15%, respectively (others could not be classified according to the model). In addition to the control/addressing/data error breakdown, Chapter Two provides finer grain classes that include more detail about exactly how the programmer made the error. The MVS study gives some specific information about the error propagation caused by addressing errors. For example, these errors are more likely than other software errors

[1] MVS, IMS and DB2 are trademarks of IBM Corporation.

to have high impact on the availability experienced by customers. Addressing errors in MVS tend to be small and often corrupt data very near the data structure that the software intended to operate on. This and other data presented in Chapter Two can be used to provide a larger picture of software failures in high-end commercial systems that, we hope, will be useful to others studying fault tolerance and software testing outside of the context of the dissertation.

Chapter Three focuses on the use of hardware write protection both to detect addressing-related errors quickly and to limit the damage that can occur after a software error. System calls added to the Sprite operating system allow the DBMS to *guard* (write-protect) some of its internal data structures. Guarding DBMS data provides quick detection of corrupted pointers and array bounds overruns, a common source of software error propagation. Data structures can be guarded as long as correct software is given a means to temporarily unprotect the data structures before updates. The dissertation analyzes the effects of three different update models on performance, software complexity, and error protection. Measurements of a DBMS that uses guarding to protect its buffer pool show two to eleven percent performance degradation in a debit/credit benchmark run against a main-memory database. Guarding has a two to three percent impact on a conventional disk database, and read-only data structures can be guarded without any affect on DBMS performance.

To lessen the availability impact of errors once they are detected, the DBMS must restart quickly after such errors are detected. Chapter Four develops an approach to fast recovery centered on the POSTGRES storage system [66]. The original POSTGRES storage system was designed to restore consistency of the disk database quickly, but did not consider fast restoration of non-disk state such as network connections to clients. Chapter Four describes extensions to POSTGRES required for fast reconnection of the DBMS and its client processes. The chapter also describes a set of optimizations that reduce the impact of the storage system on everyday performance, making fast recovery more practical for databases with high transaction rates. Finally, Chapter Four presents an analysis of the I/O impact of the POSTGRES storage system on a TP1 debit/credit workload. This analysis shows that the optimized storage system does the same amount of I/O as a conventional DBMS when a sufficient amount of non-volatile RAM is available.

Chapter Five also widens the applicability of the POSTGRES fast recovery techniques by extending the POSTGRES storage system to handle index data structures. While the POSTGRES storage system recovery strategies are effective for restoring the consistency of heap (unkeyed) relation without log processing, different strategies must be taken for maintaining the consistency of more complex disk data structures such as indices. The two algorithms described in Chapter Five allow POSTGRES to recover B-tree, R-tree, and hash indices without a write-ahead log. One algorithm is similar to shadow paging, but improves performance by integrating shadow meta-data with index meta-data. The

other algorithm uses a two-phase page reorganization scheme to reduce the space overhead caused by shadow paging. Although designed for the POSTGRES storage system, these algorithms would also be useful in a conventional storage system as support for logical logging. Using these techniques, POSTGRES B-tree lookup operations are slower than a conventional system's by 3-5% under most workloads. In a few cases, POSTGRES lookups also require an extra disk I/O. On the other hand, the system can begin running transactions immediately on recovery without first restoring the consistency of the database.

The sixth chapter concludes and describes some avenues for future research. Because the dissertation has four very distinct sections, the literature review for each chapter will be included in the chapter. Together, these chapters attack three problems of interest to fault tolerant system designers: they describe the character of software errors, improve error detection, and widen the applicability of some existing fast recovery techniques.

Chapter 2

A Survey of Software Errors in Systems Programs

2.1 Introduction

Any technique for improving system reliability and availability has underlying it a model of system failure. A given technique is successful only if real systems fail in ways covered by the model. The introduction described a model of system failure based on three kinds of software errors that propagate errors in different ways. This model guided our approach to maintaining high availability in POSTGRES and motivated some of the techniques described in Chapters Three, Four, and Five. In this chapter, we present an analysis of errors discovered in three commercial systems programs. The analysis helps to clarify the control/addressing/data error model, hence, the reliability and availability impact of the techniques described in the dissertation.

The chapter describes two studies of software errors identified in the MVS operating system and the IMS and DB2 database management systems. The data available for the studies comes from an internal IBM database of error reports. Each report was filed by a customer service representative when the software failed at a customer site in the field. The IBM programmers who repair a fault amend the error report with further details about the fix. The studies only considered errors for which fixes were eventually found.

We classified the IBM error data in several different ways, each considering the cause of an error from a slightly different perspective. Chapter Two concentrates on two of these classifications: error type and error trigger. The **error type** provides insight into the programming mistakes that cause software failures at customer sites. A better understanding of programming mistakes will help programmers, recovery system designers, and software tool designers to improve code quality. The **error trigger** illustrates the circumstances

under which latent errors arise at customer sites. Since software testing is supposed to uncover these latent errors before the code is shipped to customers, the trigger data should help show how testing strategies can be improved. The chapter also includes statistics on **failure symptoms** that characterize the way the system failed when it executed the faulty code.

Because both the original data and the classification process are prone to error, studying several different programs was important. Each program provides a fairly independent error sample; the programmers and the people who wrote bug reports were different for each one. MVS is not an ideal source of error data, since it is an operating system not a database management system. However, many of the resource management issues in DBMSs and OSs are the same. DBMS and OS programs also have similar size, are written in similar systems programming languages, and have the same kinds of concurrency, availability, and performance requirements. Given the available data, MVS seemed a good choice for an additional source of error information.

A second reason that MVS was chosen as a source of error data is that MVS maintenance programmers noted the existence of addressing errors in a standard way. In MVS, the damage caused by an addressing-related error is called an **overlay** by IBM field service personnel. Searching for error reports that use this term allowed us to collect a large sample of error reports that discuss addressing-related errors. These error reports could be compared to MVS error reports as a whole. Because the error detection mechanism described in Chapter Three only affects addressing errors, it was important to gather as much additional information as possible about the character of addressing errors.

The chapter is organized as follows. Section Two summarizes several related software error studies. Section Three describes the data used in the IBM studies and the classification systems used to characterize the data. Section Four presents the results of the studies, and Section Five summarizes the implications of these results for our system availability techniques. For additional details about the studies themselves, see [70], which compares addressing errors to errors overall in MVS, and [71], which focuses on control errors and discusses differences between operating system and database management system errors.

2.2 Previous Work

We would have liked to use a survey of data collected and analyzed by other researchers to evaluate the effectiveness of the POSTGRES error detection techniques, rather than gather our own data. Unfortunately, error studies are often difficult to adapt to purposes other than the ones that the original researchers had in mind. Several early error studies tried to show the importance of clear software specifications for improved code quality. Endres

[24] studied software errors found during internal testing of the DOS/VS operating system. His classification was oriented towards differentiating between high-level design faults and low-level programming faults. Glass [27] provides another high-level, specification-oriented picture of software errors discovered during the development process. Neither study gave much detail about what kind of coding errors caused the programs to fail, so neither is of much help to us.

Another important reason why existing surveys of software errors are not ideal for studying system availability is that they focus only on errors discovered during the system test and code development phases of program life cycles. The errors that actually affect availability are the ones discovered at customer sites, after development and testing are complete. Another early error study, [74], provides some of the same level of error analysis that our study provides, but on errors discovered during the testing and validation phases. Basili and Perricone study the relationship between software errors and complexity in Fortran programs [9]. Their study finds a predominance of errors in interfaces between modules, but the study also focuses on development and test phases. In [43], Knuth describes both design and coding errors uncovered in his TeX text processing program. The presentation includes some efforts at fault categorization, but is largely a collection of anecdotes. It is less applicable than the other studies since the program was written by one person, rather than a team of programmers, and it is a very different application from database manager. Like the other studies, it covers mostly program development and early test phases.

A few researchers have examined failures in system software at customer sites, but they provide little detail about the types of software errors that led to the failure. One example is Levendel's study of the software that manages the ESS5 telephone switch [48]. The study does not break errors into classes, but instead uses error data to estimate the effectiveness of some standard reliability metrics. These metrics use trends in bug-fix rates to guess how many more errors remain in a given piece of code. Managers can use this information to make decisions about release dates, but it is not the kind of information that can be used to evaluate potential error detection or recovery strategies.

Several studies used data from error logs to track failures at customer sites [16][38][56]. Error log records are generated automatically by the system after a program fails. Because the log entries are generated automatically, they give extremely high-level representations of the error. For example, the log entry might be a code indicating that the program tried to store into an invalid address. The error log does not include the semantic information about the error needed to determine what the programmer did wrong.

2.3 Gathering Software Error Data

The data available for our studies came from an IBM internal field service database called REmote Technical Assistance Information Network (RETAIN). RETAIN serves as a central database for hardware problems, software problems, bug fixes, and release information. When an IBM system fails, IBM service personnel use RETAIN to determine if the same failure has occurred at another site. If so, information stored in RETAIN identifies a tape containing a fix for the problem. If the problem has never occurred before, people must be assigned to track down and repair the software error that caused the failure. It is quite possible for the same error to occur at multiple sites. Although IBM fixes errors as soon as possible when they are detected, customers often delay installing the fixes until their systems have to be taken down for other reasons, such as maintenance. In these cases, the customer prefers to risk the occurrence of a known bug rather than suffer periodic additional outages to install fixes.

When a new software error has arisen in an IBM product, a customer service person files an **Authorized Program Analysis Report** (APAR) describing the fault in RETAIN. Every APAR identifies a few standard attributes associated with the faulty software, such as the type of machine running the software, the software release number, a symptom code describing the failure, and a severity rating. The service person filing the APAR also adds a text description of the error if any information is available. After the error is repaired, one of the programmers responsible for the repair writes a description of the fix and amends the initial problem description and severity rating.

An APAR does not contain standardized fields identifying the "cause" of a fault. Semantic information about the fault and the circumstances under which it arises is only contained in the APAR text. The text is oriented toward future RETAIN searches by IBM service personnel after the fault occurs at a different site. Often it contains more information about the effects of the fault than about the fault itself.

IBM saves an APAR for each distinct error that occurs in its software products, but the APAR does not include an accurate count of the frequency with which that error occurs. **Problem Reports**, or PMRs, are filed for each customer outage whether it is caused by a unique fault or not. Since PMRs include a field for the APAR associated with a given software problem, they could be used, in theory, to determine the frequency of observed faults. PMRs, however, are not retained by IBM for more than a few months. Also, the accuracy of some PMR-APAR associations is questionable. If an untraceable software error occurs, IBM service and the customer site will often agree to reboot the newest version of the software and hope for the best. If the fault was transient, the error will seem to go away even if the new software does not contain a fix. Earlier studies, such as [28], suggest that transient software faults are fairly frequent.

Some software errors are worse, from the customer's perspective, than others, so it would be a mistake for the error studies to give all APARs in RETAIN equal weight. APARS describing errors with little or no impact on availability were discarded in our studies. These included suggestions for user interface changes and errors which affected the presentation but not the content of program results (e.g. garbage characters are printed to the terminal after the prompt). Errors with especially high impact were singled out to be examined in more detail. RETAIN does not identify high impact errors directly, but several standard APAR attributes can be used to estimate the impact of the error described.

Severity Code is supposed to indicate how badly inconvenienced the customer was by the outage. It is also used do indicate the priority of the bug to the people who assign maintenance programmers to fix it. Severity one APARs have the worst affect on availability. The customer has stated that work at his or her site cannot progress until the fault is fixed. Severity two errors have customer impact, but have lower priority to the maintenance teams because the customer has found a circumvention or temporary solution to the fault. Severity three and four APARS correspond to lesser damage and can range from annoyance to look and feel or interface problems.

HIPER The HIghly PERvasive error flag is assigned by the change team that fixes the faulty code. HIPER software errors are those considered likely to affect many customer sites — not just the one that first discovered the error. Flagging an error as HIPER provides a message to branch offices to encourage their customers to upgrade with this fix.

IPL errors destroy the operating system's recovery mechanism and require it to initiate an Initial Program Load (IPL) or "reboot." An IPL is clearly a high impact event since it can cause an outage of at least 15 minutes. This metric is probably the most objective of the impact measurements since there is little room for data inaccuracy. While labeling an error HIPER or severity one is a judgement call, the occurrence of IPL is difficult to mistake. Note that IPL is an effective impact estimator for MVS, but in the DBMS error study there were no errors that cause the operating system to IPL. DB2/IMS errors in which the DBMS failed and had to restart should be counted as high impact, but this information was not always included in the APAR.

Using these impact estimators, RETAIN's APARs can be broken into three groups. Low impact APARs with severity ratings of three and four were discarded from the study. Severity two APARs were serious enough to be considered in the study, but not labeled as high impact. Errors flagged as HIPER, IPL, or severity one are considered **high impact** errors. When error distributions are presented later in the chapter, high impact errors will be singled out and presented separately.

The MVS study uses error data from the MVS Operating System for the period 1986-1989, representing several thousand machine years of execution. It only includes errors in the operating system and some of the low-level software products that are bundled with it. The IMS and DB2 APARs were drawn from those recorded against those two database management systems in the years 1987-1990. The second study took errors from a later period because it was conducted a year later and because DB2 was not mature enough in 1986 to have a large APAR base.

2.3.1 Sampling from RETAIN

If it were possible to classify APARs using software, each of the APARs in RETAIN associated with MVS, IMS and DB2 could be classified in order to find the complete distribution of errors for those products. RETAIN provides some help in this regard. It allows users to identify subsets of APARs using simple keyword searches on the keyed fields (e.g. HIPER, severity). Keyword searches allow us to report customer impact statistics based on the entire population of APARs associated with each product.

The error type and triggering event, unfortunately, are too complex to identify without reading the APAR text and extracting fault information from the change team's problem description. Classifying the thousands of available APARs to get this information would be beyond the resources available for this study. Therefore, we sampled from the population of available APARs in order to restrict the number of APARS to be read.

For the MVS study, we constructed two sets of APARs — the **regular** sample and the **overlay** sample. To gather the regular sample we drew 150 APARs from the population of all severity one or two APARs from 1986-1989 filed against MVS. To derive the overlay sample, we could not just take the subset of MVS APARs that involved overlay errors since the MVS sample itself was so small. Instead, we searched the text parts of the APAR for strings containing words such as "overlay" and "overlaid." From this restricted set of APARs, we drew APARs that were potential overlays. IBM software engineers use the term overlay to mean "stored on top of" data currently in memory, so occasionally the overlay is legitimate behavior unrelated to the error described. Further reading allowed us to weed out APARs in which the overlay was not caused by broken software, leaving 91 overlay APARs. For the DBMS study, we randomly sampled 201 of IMS's severity one and two APARs and 222 of DB2's.

The MVS regular sample is not taken in the straightforward way because of a sampling error in the initial phases of the first study. We had first planned to examine only severity one APARs. Later, we realized that severity two errors had a high enough customer impact that it would be a mistake to ignore them in the study. To overcome this problem, we pulled a second independent random sample from the population of severity two APARs.

We then combined the results from the severity one and two samples in the proportion they are represented in the population. We used boot-strapping [22] to combine the samples rather than a simple weighted average. Boot-strapping is a common statistical technique that does not build in any assumptions about the distribution of the parent population as would a weighted average.

2.3.2 Characterizing Software Defects

The error studies approach the "cause" of an error from both the standpoint of a programmer/recovery-manager and from the standpoint of a system test designer. **Error type** is the low level programming mistake that led to the software failure. The **error trigger** classification was meant to give insight into the software testing process. Both IBM and its customers test software thoroughly before the customer relies heavily enough on the software for its failures to have an impact. When an error arises at a customer site, some aspect of the customer's execution environment must have caused the defective code to be executed, even though the same code was never executed during system test. The error trigger classification distinguishes the different kinds of events that cause errors that remained dormant during testing to surface at the customer site. Better understanding of these triggering events should improve the testing process.

To identify error type and error trigger classes, we made several passes through the sample looking for commonalities in the errors. Once some general categories were chosen, we read each APAR more carefully, placing it into one of the possible categories for error type and one category of error trigger. Each of the APARs in the samples was associated with only one error type and error trigger even though the same APAR occasionally mentioned several related faults in the software. After classifying the APARs we found several categories with one or two APARs in them, which we merged into larger, more general classes. Several of the one and two APAR categories were grouped together into an "Other" category when they could not reasonably be grouped together with APARS of a more meaningful error type.

Error Types

A few programming errors caused most of the errors in the programs we studied. These were the error types defined during the study of MVS:

Allocation Management : One module deallocates a region of memory while the region is still in use. After the region is reallocated, the original module continues to use it in its original capacity. The few errors in which the memory region allocated was too small for the data to be stored in it were counted as allocation management errors as

well. Arguably, these should have been placed in the Copying Overrun class instead, but there were not enough of these to make much difference to the study results.

Copying Overrun : The program copies bytes past the end of a buffer.

Data Error : An arithmetic miscalculation or other error in the code makes it produce or read the wrong data.

Pointer Management : A variable containing the address of data was corrupted. For example, a linked list is terminated by setting the last chain pointer to NIL when it should have been set to the head element in the list.

Statement Logic : Statements were executed in the wrong order or were omitted. For example, a routine returns too early under some circumstances. Forgetting to check a routine's return code is also a statement logic error.

Synchronization : An error occurred in locking code or synchronization between threads of control.

Type Mismatch : A field is added to a message format or a structure, but not all of the code using the structure is modified to reflect the change. Type mismatch errors also occur when the meaning of a bit in a bit field is redefined.

Undefined State : The system goes into a state that the designers had not anticipated. For example, the program may have no code to handle an end-of-session message which arrives before the session is completely initialized.

Uninitialized Variable : A variable containing either a pointer or data is used before it is initialized.

Other : Several error categories which had few members were combined into a single category called Other.

Unknown : The error report described the effects of the error, but not adequately enough for us to classify it.

During the DBMS study, we added three error types to the set used to classify MVS. The additional error types represent a refinement to the classification system based on the data in the second study. Errors from each of these classes were present in MVS, but uncommon, so they fell into the Other class in the original MVS study.

Interface Error : A module's interface is defined incorrectly or used incorrectly by a client.

Memory Leak : The program does not deallocate memory it has allocated.

Wrong Algorithm : The program works, but uses the wrong algorithm to do the task at hand. Usually these were performance-related problems.

Error Triggering Events

This classification describes the circumstances which allowed a latent error to surface in the customer environment. For every error in the sample, we assigned one of the following trigger events:

Workload : Often software failures occur under limit conditions. Users can submit requests with unusual parameters (e.g., please process zero records). The hardware configuration may be unique (e.g., system is run with a faster disk than was available during testing). Workload or system configuration could be unique. (e.g., too little memory for network message buffering).

Bug Fixes : An error was introduced when an earlier error was fixed. The fix could be in error in a way that is triggered only in the customer environment, or the fix could uncover other latent bugs in related parts of the code.

Client Code : A few errors occurred when errors were propagated from application code running in protected mode. In order for these to appear in the APARs that we sampled, the code for recovering from the propagated error would have had to contain a fault.

Recovery or Exception Handling : Recovery code is notoriously difficult to debug and difficult to test completely. The DBMS data distinguishes full DBMS recovery (using the log) from cleanup after transient errors (exception handling).

Timing : Timing triggers are an important special case of workload triggers in which an unanticipated sequence of events directly causes an error. An error that only occurs when the program is interrupted at an inopportune moment would be a timing-triggered error.

Unknown : The triggering event could not be determined from the available data.

Failure Symptom Codes

When an APAR is opened, a symptom code is recorded describing one of the external effects of the fault. This field is often used by customer service personnel to search for an existing fix when an error is first discovered. They focus on symptoms because symptoms are usually the best information available about a fault when it first occurs.

The symptom code of an APAR was not assigned as part of our APAR studies; we simply used and analyzed data already present in RETAIN. Also, a single failure may have many symptoms. Maintenance programmers decide which is the most interesting one to record in the APAR symptom code field. "Interesting" failure symptoms for the maintenance programmer may not be interesting for fault tolerance research. For example, the unusual error message that the system printed to the screen before it went into an infinite loop might be recorded as the failure symptom, rather than the infinite loop itself.

Failure symptoms fall into these classes:

ABEND : An abnormal program termination occurred. The currently running application program failed and must be restarted.

Address Error : The system fails after trying to use a bad address.

Endless Wait : Processes wait for an event that will never occur.

Incorrect Output : The system produces incorrect output without detecting the failure.

Infinite Loop : The system goes into an infinite loop.

Error Message : The system cannot perform the requested function but prints an error message on the screen and performs local recovery rather than ABENDing

2.4 Results

We describe the results of the two IBM studies together in the following section, comparing MVS, IMS, and DB2 wherever possible. The results section is divided into four subsections, based on the different APAR categorization schemes defined in Section 2.3. The largest of these four subsections discusses error type, the categorization based on types of programmer mistakes. The error type subsection gives breakdowns of control, addressing, and data errors in order to provide a better understanding of the error propagation model given in Chapter One. It also gives finer-grain description of programmer errors based on the error types defined in Subsection 2.3.2. The second subsection compares the number of high impact errors in the DB2, IMS, MVS overall, and MVS overlay-only APAR samples. The next subsection, which describes error triggering events, will be of most interest to system test suite designers. However, it is also of interest in recovery system design because it indicates the frequency of repeatable software errors. The fourth subsection gives the failure symptoms that describe the system behavior after the error occurred.

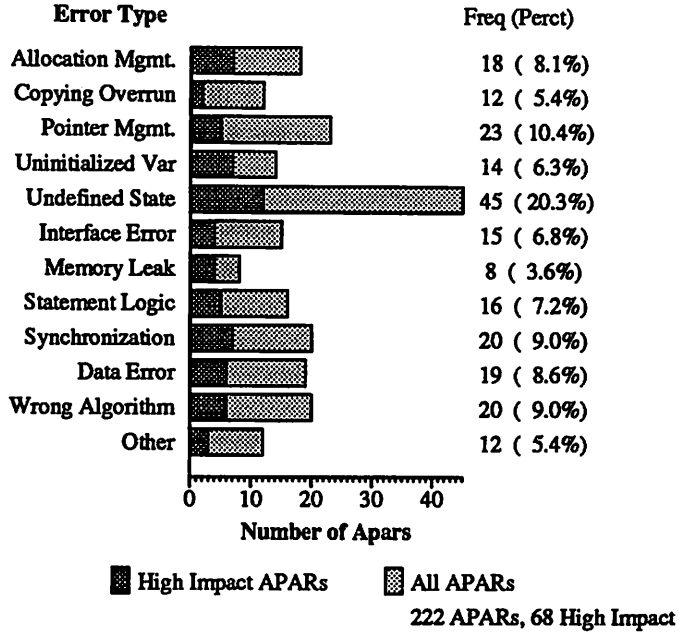

Figure 2.1: DB2 Error Type Distribution.

2.4.1 Error Type Distributions

Figures 2.1 and 2.2 summarize the error type distributions for each database management system. Figure 2.3 shows a breakdown of error types from the regular sample in the MVS study. Each figure shows two distributions: one for availability-related APARs as a whole, and one for high impact APARs. The high impact distribution is superimposed on the overall distribution since the high impact APARs are a subset of the overall APAR sample. Each bar in the figure represents one of the error types defined in Section 2.3.2. The length of the bar shows the number of errors represented in the APAR sample which were caused by that type of error.

In both DBMS products, undefined state, a control error, was the largest error type. In IMS, undefined state errors accounted for 40% of the whole and 29% of the high impact errors. The next largest class was pointer management, an addressing error, which accounted for 11% of the APARs sampled. In DB2, undefined state accounted for 20% of APARs and 18% of the high impact ones. DB2's next highest class overall was again pointer management errors with 10%. Undefined state was an important source of errors in MVS, but it did not dominate the error type distribution as much as in IMS and DB2 (17% of the whole and 25% of the high impact errors). The pointer management class in MVS was 12% of errors, about the same as it was in the two DBMSs.

The remainder of this subsection explores the error type data in greater detail. First, we

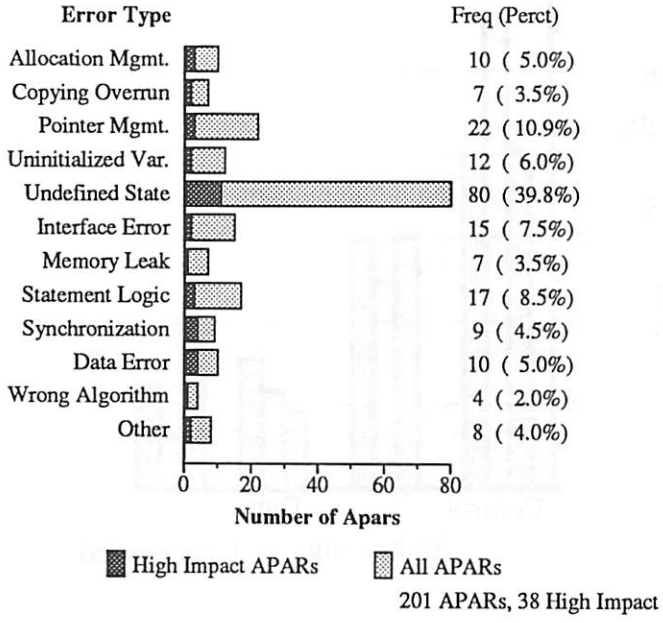

Figure 2.2: IMS Error Type Distribution.

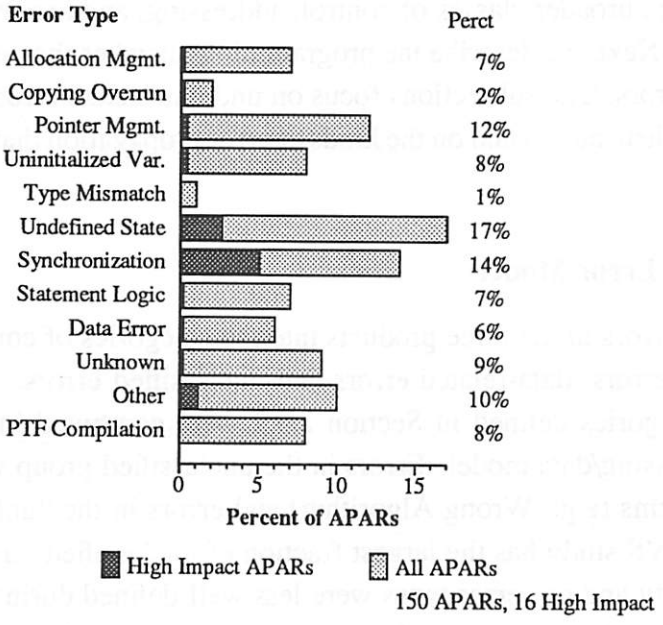

Figure 2.3: MVS Regular Sample Error Type Distribution.

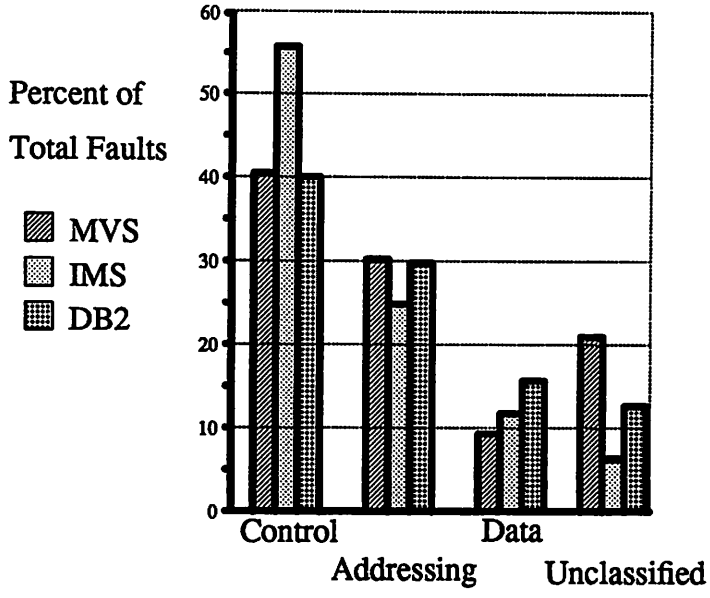

Figure 2.4: Control/Addressing/Data Error Breakdown DB2, IMS, and MVS Systems.

combine error types into the broader classes of control, addressing, and data error used in the model in Chapter One. Next, we describe the programming mistakes that led to control errors and to addressing errors. The subsections focus on undefined state errors since they dominate the control error distribution and on the kinds of error propagation that result from addressing errors.

Control/Addressing/Data Error Model

Figure 2.4 groups the errors in the three products into the categories of control-related errors, addressing-related errors, data-related errors and unclassified errors. To produce Figure 2.4, error type categories defined in Section 2.3.2 were combined into the categories of the control/addressing/data model. Errors in the unclassified group were largely performance-related problems (e.g. Wrong Algorithm) and errors in the "unknown" and "other" categories. The MVS study has the largest fraction of unclassified APARs in part because it was the first study and our error types were less well-defined during that study. The Y-axis in this chart shows the *percentage* of errors from each product's sample that fall into each class, not the absolute number of APARs. MVS, in this chart, is the MVS regular sample.

In all three products, control errors make up the most significant fraction of errors and

addressing errors, the second most significant. Control errors are more common than the other two, composing 40% to 55% of the total for each product. A much larger fraction of the IMS errors are control-related than errors in the other two products. In part, this is because IMS was late in the product life cycle during the time covered by the study. Few new features are added to IMS any more, so most of the changes to the code occur during maintenance. From the data, it seems that maintenance programmers have difficulty understanding all of the implications of a given change to the control flow of the program. DB2 has more data errors than the other two products. Many of these errors were mistakes in calculating the cost of a prospective query plan during the planning stage of query execution.

Because this data comes from errors discovered once the software had been released to customers, there are two possible causes for the error distributions in Figure 2.4. Possibly, the distributions represent the frequency with which each kind of programming mistake occurs. Programmers may simply be more likely to make control errors than data errors. A more likely explanation of the figure, however, is that some errors, such as data errors, are detected relatively easily during program development and test by standard debugging techniques. Hence, the distribution in the figure is skewed towards the errors that are hardest to detect during normal development and test. As will be shown below, control errors often occur during error handling. If the error condition is difficult to generate during system test, the error handling code might not be fully tested. Incomplete testing may prevent some addressing errors from being uncovered early, as well. Addressing errors sometimes cause corruption of storage that is near a data structure managed by faulty code. The order in which data structures are allocated may determine which one is damaged by the error. Because testing cannot cover all allocation orders, the error may never occur during development and test.

In Chapter One, we suggested that addressing-related errors were the most dangerous error class in terms of error propagation. An addressing error can corrupt data unrelated to the module in which the error occurs, hence can be difficult to find and remove. Addressing-related errors, including copy overruns, allocation management, pointer management problems and uninitialized pointers, make up 25 to 30 percent of the APARs filed against IMS, MVS, and DB2. This is consistent with several other studies of software errors in operating systems summarized in Figure 2.5. The published studies in the figure are from DOS/VS [24] and MVS (one from [76] and two from [56]). The Unpublished I study was a survey of errors reported in the 4.1/4.2 releases of BSD, a UNIX-like operating system [69]. [1] The Unpublished II operating system error study was conducted internally at a company that would not allow the release of its name. Control and data

[1]UNIX is currently a trademark of UNIX Systems Laboratory in the US and other countries, however, pending litigation calls this point into question.

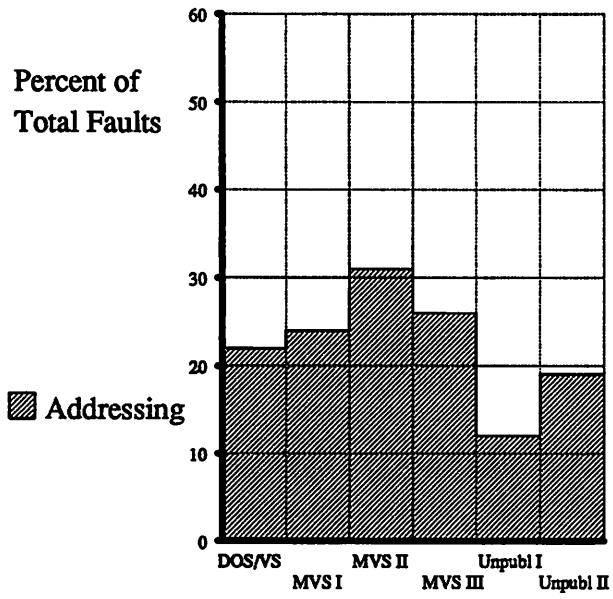

Figure 2.5: Summary of Addressing Error Percentages in Previous Work.

errors are not pictured because the studies in the figure did not categorize errors in a way that mapped to the control/addressing/data error model. Each study identified some errors as addressing-related, however, which allows some comparison between these studies and our own.

The BSD study showed many fewer addressing-related errors than the other studies. Most of the errors in the BSD study were synchronization or configuration problems related to device drivers and network protocols. The error report information available did not distinguish between errors discovered in test phase and production-use phase and many of the device driver problems would probably have been discovered during testing in a commercial enterprise with a large, in-house quality assurance group. If we could consider only post-test-phase software errors in BSD, the fraction of addressing errors might be closer to that seen in the other studies.

Together, the available information on programmer mistakes suggests that at least twenty to thirty percent of the faults that cause systems to fail involve addressing errors. Thirty percent may not be an upper bound since these studies usually only report addressing errors when they are the *primary* cause of a software failure. Even in the APAR data, an error report describing a control or data error will occasionally mention that the system failed with an address trap, indicating that secondary addressing errors occurred but were considered too unimportant to describe in the APAR.

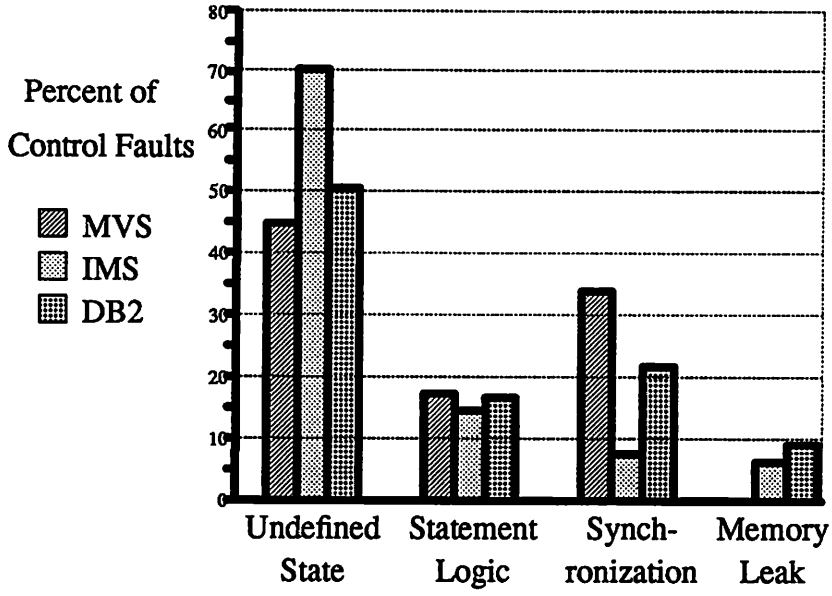

Figure 2.6: Distribution of the Most Common Control Errors.

The next four subsections describe control and addressing errors in more detail. The first subsection lists the major causes of control errors. The second details the dominant control error, undefined state. The third subsection gives the distribution of addressing error types along with some examples, and the fourth describes some additional information on addressing errors gathered in the MVS overlay study.

Characterizing Control Errors

Figure 2.6 shows the distributions of the most common of the control-related error types for each of the IBM products studied. Each bar in the figure represents one of the error types defined in Section 2.3.2. The MVS bars represent error type distributions in the MVS regular sample, not the overlay sample. The MVS sample has no memory leak errors because memory leak was not selected as an error type until the DBMS error study. There were memory leak errors in MVS, but so few that we did not identify it as a separate error type during the study. Memory leak counts as a control error because these errors eventually cause the system to be reinitialized in order to allow reallocation the memory lost in the leak.

For each of the products, undefined state is the most common control error. In DB2 and IMS, synchronization-related errors are fairly common. The DB2 synchronization errors usually occur when DB2 is used interactively, and they are often related to cleanup

after errors. Clean up after the user cancels a command from the keyboard caused some synchronization problems in DB2, also. MVS synchronization errors were usually related to communication protocols, although some of the highest impact ones were errors in interrupt handlers. Because the majority of control errors in the DBMS are caused by undefined state, the next subsection describes these errors in some detail.

DBMS Undefined State Errors

An undefined state error occurs when an event in the program execution environment arises which the program has not anticipated. The program either has no code to handle the event or misinterprets the event and makes a faulty state transition as a response. The MVS study showed that undefined state errors were common, but did not provide details about what caused them. In general, the undefined state errors involved concurrency. For example, a process takes a page fault, then an interrupt for an I/O completion, and never completely initializes the page table of the faulted page.

In the DBMS study, we kept more systematic notes about how undefined states arose in the program. This turned out to be important since undefined state was even more common in the two DBMS products than they were in the operating system. These errors represent 20% of all DB2 errors sampled from RETAIN and 40% of all IMS errors. In both systems, undefined state errors had a slightly lower impact than the average error.

For IMS, about a third of the undefined state errors occurred when the program lost track of its current state. In IMS, current state for network connections, database recovery, and log management is represented by a collection of flags. Sometimes the program changes state without updating the flags correctly, or checks the wrong combination of flags to determine the current state. Many of these APARs had to do with error handling. An error would occur causing the program to change state, but flags representing the current state would not be reset. The program made the wrong response to subsequent events because it was mistaken about its current state.

Another third of the IMS undefined state errors were "missing case" problems in which a programmer forgot about a state or an external event that could arise during execution. Some of these were classic boundary conditions. For example, the programmer writes a routine comparing one element to each of the elements in a list and does not consider that the list could have zero elements. Many others arose after unanticipated error conditions. For example, a higher level and a lower level routine each expect the other to handle authorization failures. When the higher level routine sees an authorization failure, it fails since it expects the error to have been handled at a lower level.

Most of the remaining undefined state errors in IMS came from incomplete protocol specifications or implementations. The protocol might not be complete because it does not

consider some states that arise. For example, after an error condition, some kinds of log records do not make sense. A log record specifying changes to sessions does not make sense if there is no longer a current session. Sometimes the implementation omitted states that were defined in the protocol. A bug fix occasionally prevented a portion of the protocol implementation from being executed.

In DB2, the same kinds of behavior were observed but in somewhat different proportions. The missing case problems were much more common in DB2 than in IMS. Nearly half of the undefined state errors were due to unhandled error conditions or forgotten states arising from boundary conditions. Additional DB2 undefined state problems resulted when data structure consistency checkers were called at the wrong time. Sometimes the error checks detected inconsistencies that were not going to cause the software to fail. About fifteen percent of undefined state errors in DB2 were false alarms due to data structure consistency checkers.

As one would expect, about two thirds of the undefined state errors in each database manager happened because the programmer omitted logic from the program rather than because the programmer did something incorrectly. Therefore, undefined state problems generally arose not from mishandled events but from forgotten events.

Characterizing Addressing Errors

Figure 2.7 shows the distributions of the most common of the addressing-related error types for each of the IBM products studied. The figure shows pointer management, allocation management, and copy overrun errors for the IMS sample, the DB2 sample, and the MVS regular sample. As in the control error figures, the length of the bar tells the percent of all control errors that fall into the type associated with the bar. The miscellaneous errors in this case were largely uninitialized pointer errors (in particular, the large number of miscellaneous addressing errors in MVS were often uninitialized pointer errors).

Among these three common types of addressing-related faults, pointer management problems were the largest classification, accounting for 35-40% of the addressing faults. A fairly common type of pointer management error was mis-termination of a linked list data structure. Another common pointer error arose when two different kinds of pointers could be stored in the same location (i.e. as in PASCAL or C union types). The programmer would mistake a pointer of one type for a pointer of another type. A third common pointer management subclass were "register reuse" errors. The language in which IMS, DB2, and MVS are written allowed programmers to explicitly control register use, if necessary. This explicit control allowed for mistakes in which two variables were assigned to the same register, allowing the second value stored to overwrite the first. If this was a pointer value, an overlay often followed.

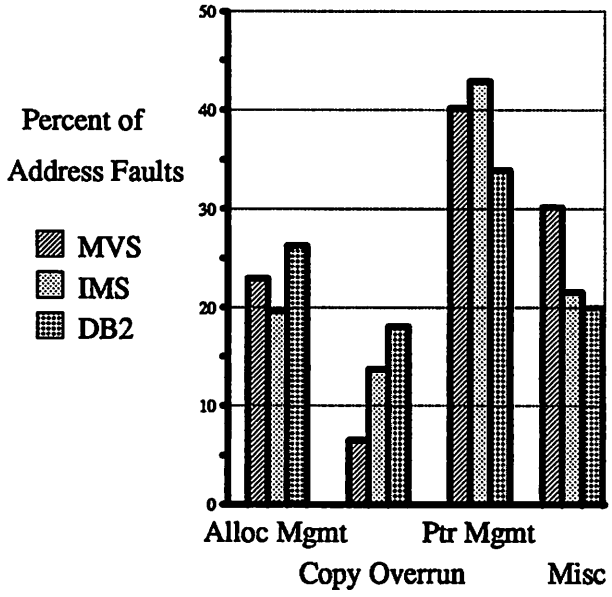

Figure 2.7: Distribution of the Most Common Addressing Errors.

Using the MVS Overlay Sample to Understand Addressing Errors

Because MVS error reports gave additional textual clues about addressing-related errors, the MVS overlay error sample was constructed containing 91 overlay-only error reports. The overlay sample shows that some overlay errors eventually follow after non-overlay error types have occurred. For example, a synchronization error sometimes allowed unsynchronized access to pointer data structures. The APAR describing the synchronization error, then, mentioned that MVS used the corrupted pointers at the time of the failure. Figure 2.8 gives the breakdown of error types for this sample. Each bar in the figure represents an error type defined in Section 2.3.2. As in the previous figures, the high impact APAR distribution is super-imposed over the overall error distribution.

Since most of the MVS operating system's tasks involve managing a system of control blocks and buffers connected by pointers, one might expect that these pointers would account for most of the overlay errors in MVS. In fact, pointer management errors and uninitialized pointers were important, but accounted for only 18% of the overlay APARs studied and 27% of the high impact overlay APARs.

Together, copying overruns (miscopying data into buffers) and allocation management errors (deallocating storage incorrectly) accounted for 39% of the total overlay APARs and 34% of the high impact ones. Although allocation management and copying overrun have about the same number of APARs filed against them, copying overruns have lower impact.

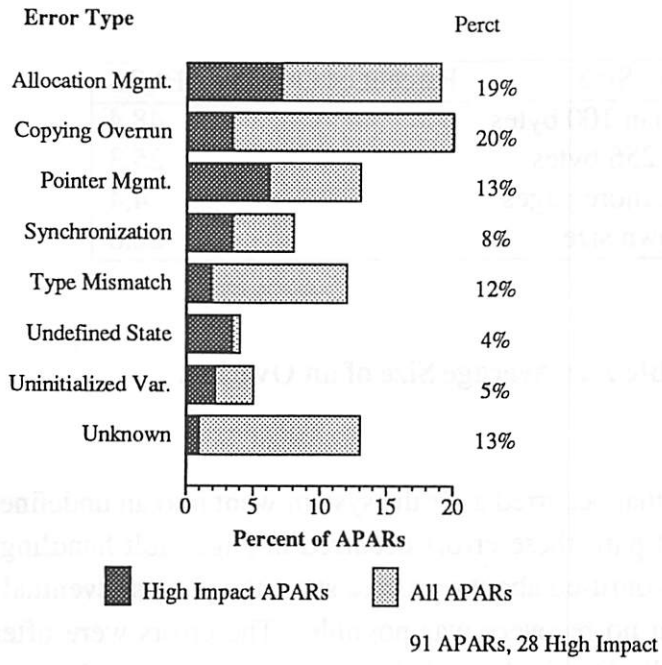

Figure 2.8: MVS Overlay Sample Error Type Distribution.

Many of these errors appeared in the terminal I/O handling code or in code for displaying messages on the console. Copying overruns were often caused by overflows or underflows of the counter used to determine how many bytes to copy. Many other copying overruns were "off-by-one" errors. In network-management code and terminal I/O handlers, buffers are processed slightly and passed from one routine to another. If the offset to the beginning of valid data or the count of valid bytes is corrupted, copying overruns occur. Most copying overruns involved only a few bytes. The few overruns which had high impact, however, caused massive corruption of memory.

One would expect some overlays to be caused by unsynchronized access to storage. In the APARs we studied, however, more overlay errors came from memory allocation mistakes than from mistakes in acquiring and releasing locks. Even when the complexity of the programming task involves synchronization, the error itself involved garbage collection. For example, a process can request a software interrupt and then free a region of memory before the interrupt is scheduled. If the interrupt tries to use this freed memory, an overlay occurs. In this case, synchronization is correct since the interrupt is not scheduled while the original process is using the memory region. Garbage collection is not correct, since the region is freed before the operating system has finished with it. When unsynchronized access to memory did occur, usually too few levels of interrupts had been masked. In these cases, unmasked interrupts allowed concurrent access to linked list data structures.

Overlay Size	Percent of Overlay APARS
Less than 100 bytes	48.4
100 to 256 bytes	25.3
One or more pages	4.4
Unknown size	22.0

Table 2.1: Average Size of an Overlay.

The few overlay errors that occurred after the system went into an undefined state were fairly severe. For the most part, these errors occurred in page fault handling. When the page fault handler became confused about a process state, the process eventually corrupted so much of the system that no recovery was possible. The errors were often extremely complex. The reports usually listed a long chain of separate events and propagations that had to occur before the failure happened.

The overlay sample allowed us to collect two additional pieces of information about how addressing errors propagate: the overlay's size and its distance from the correct destination address. Table 2.1 shows the average size of an overlay in bytes. Note that most overlays are small: nearly half are less than 100 bytes. Table 2.2 gives a rough "distance" between the overlaid data and the area that should have been written. For example, a copying overrun error corrupts data immediately following the buffer that the operating system is supposed to be using, hence, has distance "Following data structure." An example of the distance type "Within data structure" is a type mismatch error in which the operating system overlays a field of the same structure it intends to update.

Summarizing the size and distance tables, we find that most of the overlays are small with a vast majority of them close to their intended destination. In the cases in which both source and destination of the overlay could be determined, only about a fourth of the overlays were "wild stores" that overwrote distant, unrelated areas of storage.

This subsection has described an APAR categorization based on error type. The error type category has been used to show what kinds of programmer mistakes cause the system to fail at customer sites. The other important APAR categorization schemes based on error trigger and failure symptoms are described in sections 2.4.3 and 2.4.4, respectively. Before beginning the trigger and symptom subsections, we compare the customer impact of the APARs filed against MVS, IMS, and DB2. Estimating the impacts of the MVS errors is especially important because it allows us to compare the impact of the overlay and regular sample.

Overlay Distance	Percent of Overlay APARS
Following data struct	30.8
Anywhere in storage	18.7
Within data struct	26.4
Unknown	24.2

Table 2.2: Distance From Intended Write Address.

2.4.2 Comparing Products by Impact

Table 2.3 compares the fraction of APARS that have high impact in MVS, IMS, and DB2. The rows show the differences between the products in Severity one errors (errors identified by the customer as high impact), HIPER errors (error identified by maintenance programmers as highly pervasive) and high impact errors overall. For the MVS overlay and regular samples, the table lists the fraction of errors that cause the system to IPL (reboot).

Comparing the high impact error percentages in the MVS overlay and MVS regular sample shows that overlay errors have higher availability impact than non-overlay errors. Table 2.3 lists 30.8 percent of the overlay errors as high impact. When overlay and non-overlay errors are considered together in the regular sample, the high impact APAR total drops to 18 percent. Overlay errors were three times more likely to be flagged as HIPER or IPL than MVS errors overall.

The high impact of overlay-related errors is almost certainly because of error propagation. The potential for error propagation is one factor field service personnel consider when they flag APARS as HIPER. The higher HIPER rate in overlay errors was one reason for the higher impact of overlay APARs. Also, propagated errors lessen the effectiveness of system recovery mechanisms, hence, force the system to IPL after an error.

The table also indicates that DB2 has higher impact errors than MVS and IMS by all three impact metrics. DB2 is still fairly early in its product life cycle, and software defect rates have been shown to go down over time. Perhaps the impact of DB2's APARs will go down over time as well.

Several other reasons for the high HIPER and Severity ratings in DB2 have been suggested to us by the product developers. Different people assign HIPER and severity ratings for IMS, MVS and DB2. The service people assigned to DB2 may be more willing to take the customer's side than the service people in the older products. Also, MVS and IMS customers know exactly what these products should do; if the applications that use these products continue to work well, the customer is satisfied. System test can anticipate the

	Percent of APARs			
Impact Metric	MVS Regular	MVS Overlay	IMS	DB2
IPL (reboot)	6.3	19.8	NA	NA
HIPER	5.2	18.7	12.5	21.0
Severity 1	12.6	17.6	9.5	16.0
Overall	18.0	30.8	19.0	30.0

Table 2.3: Operating System and DBMS Error Impacts. The same APAR could fall into each high impact category: IPL, HIPER, and Severity 1. Thus, the Overall high impact errors figure is less than the sum of the figures in the other three rows.

workload for these products fairly well. On the other hand, DB2 customers are writing many new applications. System test probably has a harder time anticipating the way these new applications will use the DBMS. The fact that high impact DB2 errors are often triggered by unusual workloads and boundary conditions supports this suggestion.

2.4.3 Error Triggering Events

This section characterizes the events that make latent faults surface in code that has passed through system test. Most software faults that affect availability at customer sites have remained latent in the code for some time. Often, the program has been executed successfully for months at many other sites before it fails for one customer. The *trigger* is meant to capture the condition that causes defective code to be executed. By determining triggering events for the APARs examined in the two studies, we hoped to help quality assurance engineers retarget future testing efforts as well as focus efforts in building recovery systems.

Figures 2.9, 2.10, and 2.11 summarize the triggering events found in DB2, IMS, and MVS. The bars in this case are the error trigger events defined in Section 2.3.2. Again, the bar length shows the number of APARs from the sample associated with the event represented by the bar. As in the figures for error types, the high impact distributions are super-imposed on top of the overall trigger event distributions.

Conventional wisdom says that software failures at customer sites are usually timing-related. Because it is impossible to test all possible interleavings of events before the software is released, failures are assumed to involve untested interleavings of events that occur after months or years of use in the field. Our data does not support this hypothesis.

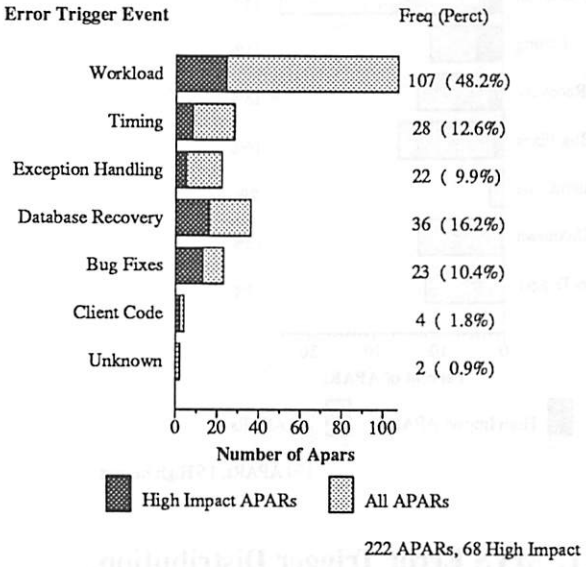

Figure 2.9: DB2 Error Trigger Distribution.

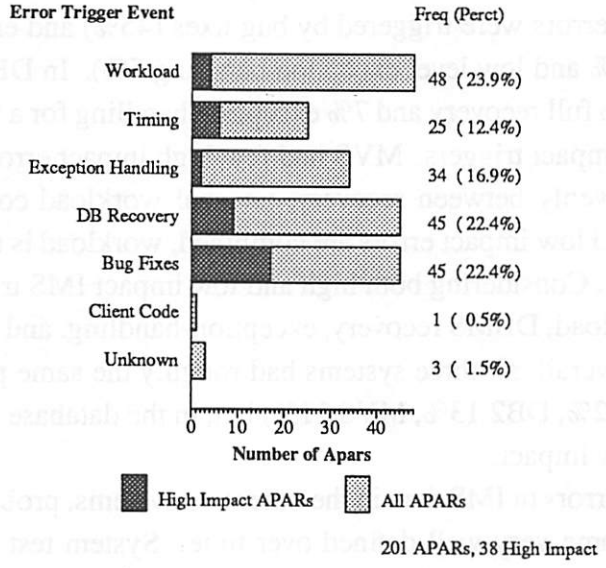

Figure 2.10: IMS Error Trigger Distribution.

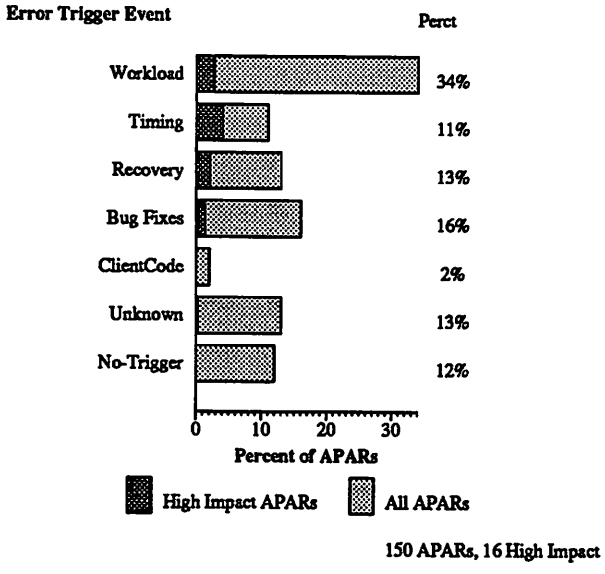

Figure 2.11: MVS Error Trigger Distribution.

Timing directly triggers a relatively small percentage of errors in each of the APAR samples we examined. The dominant trigger for most errors is unusual workload conditions. Most failures recorded in our APAR samples occurred when customers used new features, new hardware configurations, or used old features in a new way.

In IMS, most high impact errors were triggered by bug fixes (45%) and error handling (both full DBMS recovery 24% and low level exception handling 5%). In DB2, workload (35%) and error handling (24% full recovery and 7% exception handling for a total of 31%) were the most common high impact triggers. MVS had few high impact errors. The ones we saw were divided fairly evenly between recovery, unusual workload conditions and unusual timing. When high and low impact errors are combined, workload is the dominant trigger type for DB2 and MVS. Considering both high and low impact IMS triggers, many triggering events such as workload, DBMS recovery, exception-handling, and bug fixes are more common than timing. Overall, all three systems had roughly the same proportion of timing-triggered errors (IMS 12%, DB2 13%, MVS 11%) but, in the database manager, the timing-triggered errors had low impact.

Workload triggered fewer errors in IMS than in the other two systems, probably because the workload in IMS has become very well-defined over time. System test for IMS can anticipate most error conditions and much of the product's workload, so unusual boundary conditions do not arise as often. DB2, on the other hand, has a more broadly-defined workload (ad hoc queries), which is more difficult to cover during test. Hence, a substantially higher fraction of its errors are detected in the field by untested workload conditions.

Bug fix errors in IMS have much higher impact than they do in the other systems, but that probably comes from the product's age rather than from its testing procedures. Because IMS is late in its product life cycle, little if any new functionality is added to the system. The higher impact of maintenance-related APARs may just reflect the fact that most of the activity on IMS is maintenance-related than in the other two systems.

The text of the MVS APARs often indicated that code reuse was involved in the errors triggered by unusual workload conditions. Programmers often use the services provided by an old module rather than write new ones with slightly different functionality. Over time, some modules are used for things the original designer never considered. While this increases productivity, it also lessens the effectiveness of the original module-level testing. The tests run on the old module by the original programmer do not stress aspects of the module used by newer clients. The high level tests run by quality assurance do not stress the differences between the services the module was designed to provide and the service for which it is eventually used. Code reuse may also have caused reliability problems in the two DBMS products, but it was not as apparent in the APARs for these products.

The fact that unusual workload conditions accounted for such a high percentage of the triggering events in the three products was surprising. Boundary conditions are the type of error that one would expect testing to detect most easily. In fact, many unanticipated boundary conditions continue to arise after the software is released. What this data indicates is that inadvertently "testing" new features in a production environment is a common cause of outage. From this fact, we can draw two conclusions. First, test designers should *not* be focusing on new ways to uncover timing-related errors, but should focus instead on better ways to find untested boundary conditions. Second, errors described in the APAR database are very likely to be repeatable. If the boundary condition arises repeatedly, the system is likely to fail in the same way repeatedly. Redundancy-based recovery strategies, such as N-version programming [4] and process pairs [8], are unlikely to help much against this kind of error.

Control Errors and Recovery-Related Triggers

In both DB2 and IMS, failures triggered by faults in error handling or DBMS recovery code are likely to be related to undefined state. Compare the error type distribution for all sampled DB2 APARs to the sub-population of errors triggered by error handling (Figures 2.1 and 2.12). The distribution shifts from 20% undefined state errors to 36%. In IMS, the shift is from 40% undefined state errors to 54% in the sub-population defined by the error-handling trigger (compare Figures 2.2 and 2.13). The shift shows that undefined state errors are more likely to arise during recovery than other errors.

Unanticipated error conditions are implicated in a significant fraction of undefined state

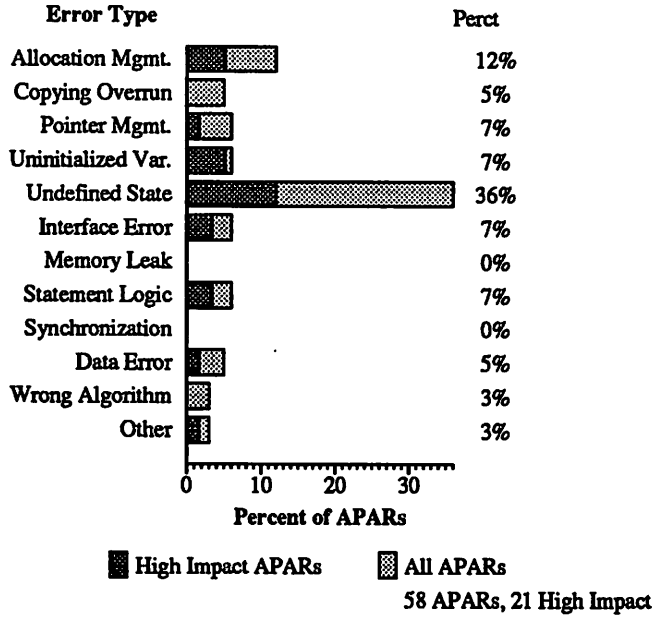

Figure 2.12: Error Type Distribution for Error-Handling-Triggered in DB2.

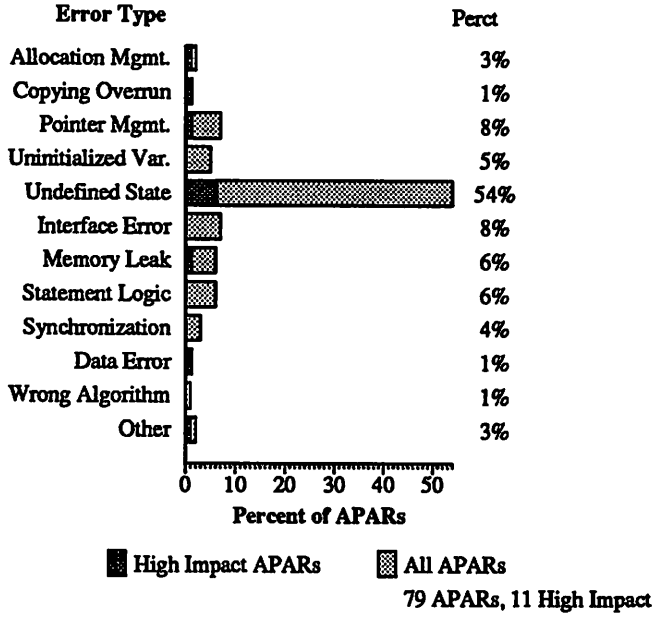

Figure 2.13: Error Type Distribution for Error-Handling-Triggered in IMS.

errors. Sometimes unanticipated error conditions directly caused the undefined state (i.e. the error condition itself was not handled correctly). In IMS, error conditions also played a part in the problem of maintaining state variables. For example, when an error condition caused the program to change state, the condition itself was handled correctly, but the state management variables were not reset.

When the database manager goes through full recovery from disk, it must construct some consistent state from the current contents of the database. The recovery protocol must anticipate all possible error states that the database is left in. In general, the logging protocols that record changes to the data in the database work correctly, but error states occur at the boundary of operating system owned resources and DBMS records of those resources. For example, the protocol for restoring the database from the log might work correctly, while maintaining the consistency of the operating system directories owned by the database manager does not.

2.4.4 Failure Symptoms

Figures 2.14 and 2.15 summarize the symptoms of the failures that occurred when code containing errors was executed. Remember that symptom is an attribute assigned by the programmer fixing the broken software. The assignment is made primarily to assist others who come across similar problems in finding the fix, i.e. the primary goal is to assign a *unique* symptom, not the symptom of the failure most relevant to an availability study. For example, if the operating system prints an unusual error message and then takes an address fault, the error message, not the address fault is the "symptom" of the failure.

In spite of these problems with the symptom data, some interesting observations can be made about it. Figure 2.14 shows that only 39 percent of overlay errors are detected as addressing violations. One could imagine that addressing errors such as pointer management errors always make the system take an addressing fault and fail without propagating the error. Even if this 39 percent figure is understated by the way symptom codes are assigned, the low number of addressing faults suggests that the subsystem damaged by an overlay uses the corrupted data before failing. Unfortunately, guessing whether or not propagation occurs is necessary since APARs usually do not say anything about the chain of propagated errors.

As expected, overlay errors are more likely to cause addressing faults than non-overlay errors. The most common non-overlay error types, undefined state and synchronization, often appear in network and device management protocols and usually cause processes to wait for events that never happen. Non-overlay errors are also more likely to cause incorrect output than overlay errors. Incorrect output failures include jobs lost from the printer queue or garbage characters written into console messages. None of the errors classified in the

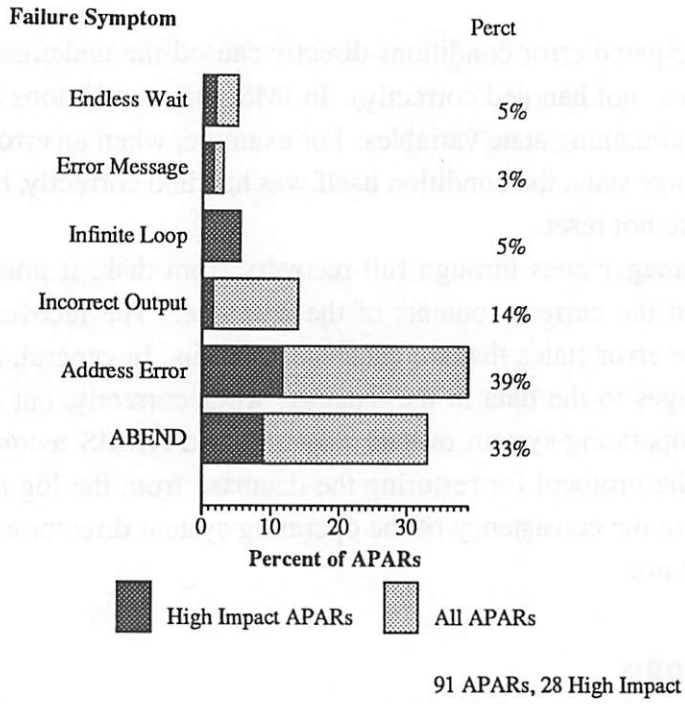

Figure 2.14: MVS Overlay Sample Failure Symptoms.

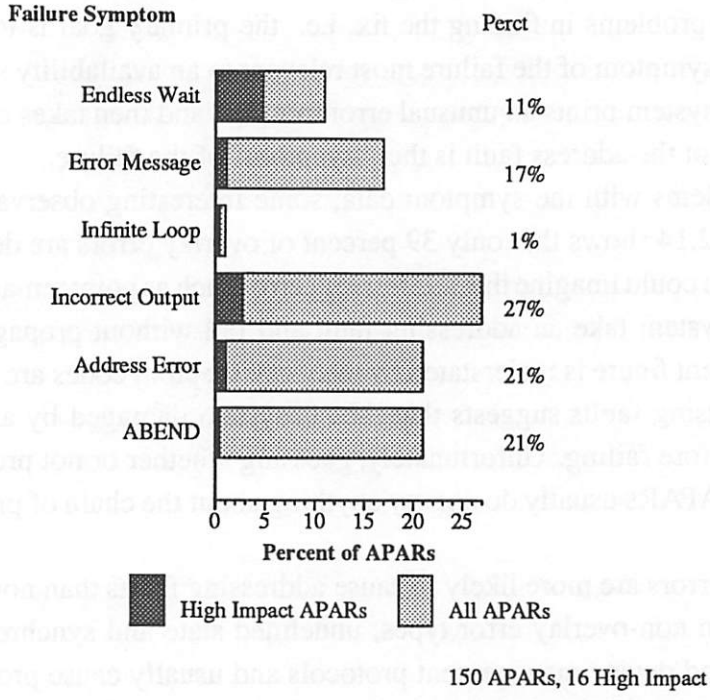

Figure 2.15: MVS Regular Sample Failure Symptoms.

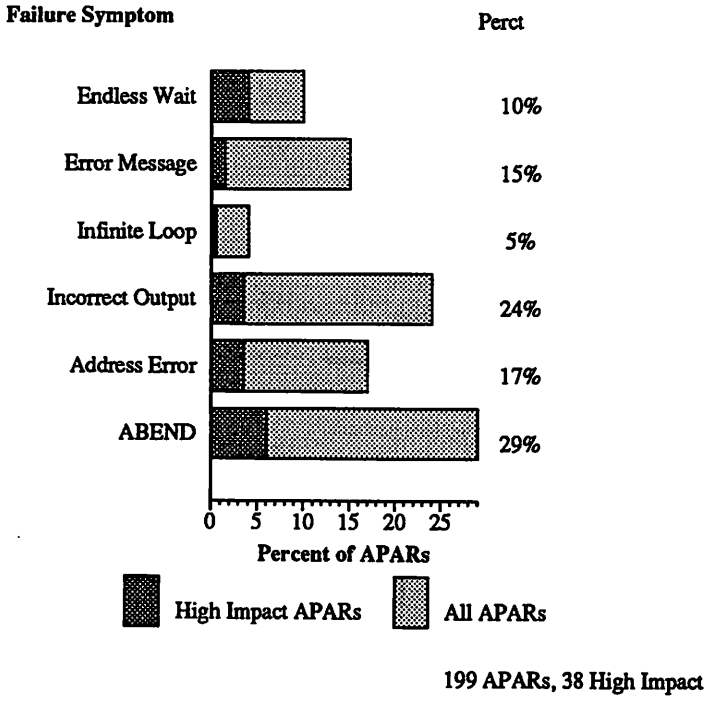

Figure 2.16: IMS Failure Symptoms.

study caused failures which corrupted user data.

IMS (Figure 2.16) and MVS have similar distributions of failure symptoms. More of IMS's software faults result in ABENDs (abnormal program termination) than MVS's and IMS takes slightly fewer address faults (as a percentage of all failures) than the operating system. Remember that IMS had more control errors and fewer addressing errors than the other two programs, so it is not surprising that fewer of its errors are detected by hardware addressing violations.

DB2 has the lowest percentage of errors that result in addressing faults and the largest that result in ABENDs. It has fewer Endless Wait and Infinite Loop failures than the other programs, in part because it has a timeout mechanism that turns some kinds of deadlock errors into ABENDs. The Performance failures in DB2 usually occur when the wrong access path is taken to the data — a problem that cannot arise in MVS or IMS since access to data is less flexible than in relational database managers.

2.5 Summary

Chapter Two has gathered together data from several sources to develop a picture of software faults and the ways they cause system unavailability and unreliability. The bulk of the chapter summarizes and analyzes data gathered from four years of software

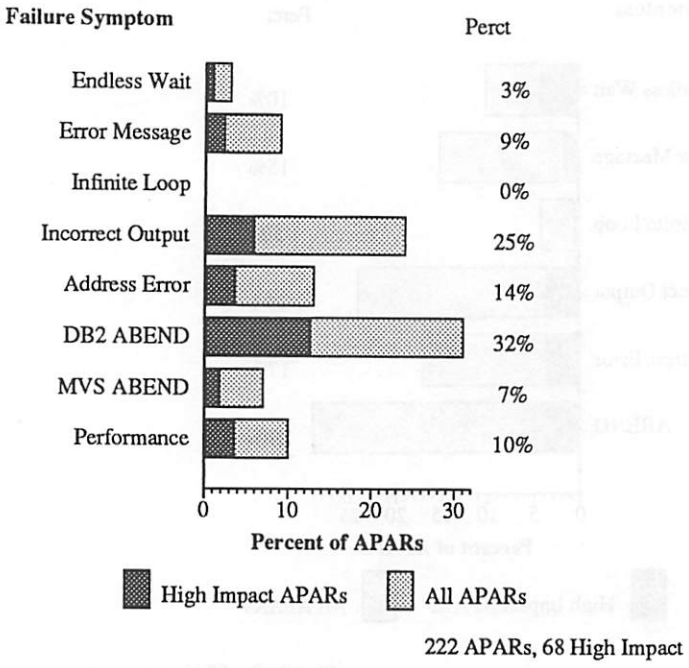

Figure 2.17: DB2 Failure Symptoms.

faults discovered in IBM systems programs at customer sites. The data comes from defects reported in the MVS operating system, IMS database management system, and DB2 database management system. It has been sampled from RETAIN, IBM's field error database, which represents several thousand machine hours of product use at customer sites.

Each error in the MVS, IMS, and DB2 surveys was classified by error type, error trigger, impact, and failure symptom. Together, these classifications provide several different perspectives on the "cause" of the software fault. Most importantly, the error type corresponds to a low level programming error that causes outage. This characterization should be the most useful in recovery system design. The error trigger describes the circumstance that allowed the error to surface in the field and characterizes potential areas for enhancement in system test.

In Chapter One, we highlighted the importance of addressing errors and error propagation. The two studies presented in Chapter Two have illustrated several important characteristics of addressing errors and the ways in which they propagate damage to other modules in the system:

1. The ranking of control errors, addressing errors, and data errors was the same across all three products. About half of all errors were control errors, 25-30 percent were addressing errors, and 5-10 percent were data errors. The remainder could not be classified using the model, usually because they affected system performance but

neither corrupted data nor propagated errors.

2. Addressing-related "overlay" errors have a much higher impact on customer availability than regular errors in MVS. These errors are more likely to damage the MVS recovery mechanisms than other errors. IBM programmers view them as higher risk than other errors to the customer base if left unrepaired. Also, customers viewing the failures caused by errors are more likely to rank errors involving overlay as high impact than the average MVS error.

3. Our data shows that most overlays are small (on the order of a few bytes) and about 75% occur near the address that the software was supposed to write. "Wild pointers" that could damage any module in memory were only about 25% of addressing errors.

These observations about the character of software errors will be used to motivate and evaluate the techniques in Chapters Three, Four, and Five. The remainder of the dissertation looks at ways to detect addressing errors, ways to limit the propagation that they can cause, and ways to recover quickly after such an error is detected.

The chapter presented additional information that is unrelated to propagation and addressing errors, but information that other researchers should find useful. For example, the error trigger classification showed that untested boundary conditions in the software trigger a majority of failures. This suggests that many of the software errors surveyed were repeatable, in contrast to the Tandem errors reported in [28]. Recovery and timing-triggered failures are few but tend to have a high impact when they do occur. This information should help guide the design of tools to help software testing. Chapter Two also shows that control errors are dominated by the undefined state error type. These errors are often related to error handling, and usually involve omitted code rather than state transitions which are handled incorrectly. Such an observation suggests that tools to improve a programmer system designer's understanding of the states the program can go into, especially after errors, will improve reliability. We hope that these and other observations from this chapter will some day assist the designers of system test suites, software development tools, reliability evaluation techniques, and recovery mechanisms.

Chapter 3

Using Write-Protected Data Structures in POSTGRES

3.1 Introduction

Chapter Three focuses on the error detection problem, describing and evaluating techniques for detecting addressing errors. Chapter Two showed that addressing errors are an important class of software error. Addressing errors are implicated in twenty to thirty percent of all software outages, and these errors have higher customer impact than other errors. Also, the introduction of the dissertation explained that addressing errors were the most dangerous source of error propagation; control and data errors usually do not affect data belonging to parts of the system unrelated to the faulty code.

In order to detect addressing errors in the DBMS, we have modified POSTGRES to use the hardware that supports virtual memory to protect some data structures from propagated errors. Several system calls were added to the Sprite operating system [58] to allow the DBMS to **guard** (write protect) parts of its address space. The DBMS uses these services to protect data in its buffer pool. To provide read-write data with protection against errors, the DBMS must support an *update model* that allows correct software to modify protected data, but prevents accidental updates by incorrect software. Different update models will make different tradeoffs regarding software complexity, performance, and the kind of error protection offered.

We have experimented with three models for updating guarded data structures: *Expose Page*, *Deferred Write*, and *Expose Segment*. A single DBMS can use different update models in different program modules, if necessary. The Expose Page model is the simplest one. The DBMS must recognize that it is about to update a protected record, unprotect the page containing the record, and reprotect the page after it is updated. In the Deferred

Write model, the DBMS copies a record it intends to update into unprotected memory and updates the copy. At the end of transaction, a system call recopies the updated record into protected memory. Finally, the Expose Segment model lets the DBMS make a system call to unprotect all guarded data at once. After the update, a second system call reprotects the guarded data.

In all three models, guarding DBMS data allows the hardware to detect illegal attempts to write to protected pages. Systems could use guarding support to improve error detection both during development and in production systems. As a debugging tool, guarding can help find software errors earlier in the development cycle. After product release, guarding lessens the impact of addressing-related errors by detecting errors at the time propagation occurs rather than after the damaged data is used. Because guarding detects a class of errors not well-covered by data consistency checkers, it complements existing fault tolerance techniques. For multi-process DBMS architectures, guarding can prevent one DBMS process' errors from corrupting data structures used by the other processes — improving overall DBMS availability. In an extensible data manager, guarding is a compromise between running application code in a separate process and running it as a full fledged part of the DBMS. Much of the protection of the separate address space model is retained at a cost much closer to the single-address space model.

This chapter is divided into five sections. The remainder of the first section describes relevant features of the POSTGRES DBMS and Sprite operating system test beds on which we have implemented guarding. The second section presents previous work related to guarding. This chapter's third section details the update models and describes their implementations. The fourth section shows some performance results and evaluates the reliability effects of guarding based on the statistics about system software errors presented in Chapter Two. A fifth section gives some conclusions.

3.1.1 System Assumptions

The discussion that follows requires some understanding of the POSTGRES process architecture depicted in Figure 3.1. The POSTGRES DBMS consists of several cooperating server processes. Each DBMS server process has its own private address space, but all of them share a single common memory region. The shared region contains a lock table, buffer pool, and other in-memory data structures used by all of the server processes. DBMS application programs run in separate address spaces and communicate with the DBMS using message passing.

POSTGRES has an unconventional storage system [66], but the results of this chapter should still be applicable to more traditional DBMS designs. The POSTGRES storage system has a "no overwrite" policy in which data records are not updated directly. An

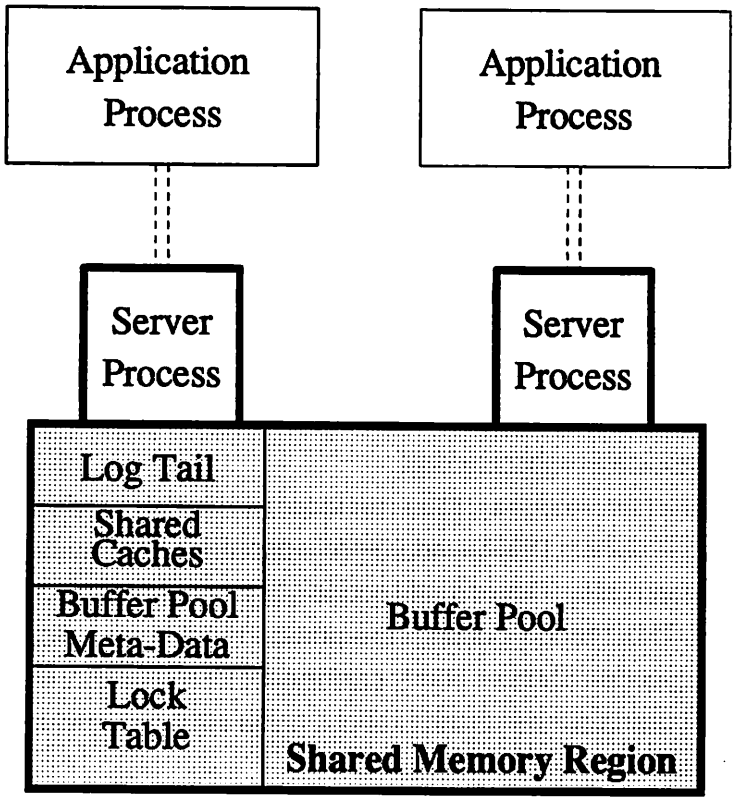

Figure 3.1: POSTGRES Process Architecture. Both server processes can address the shared memory region containing the buffer pool. Conversations between server processes and applications use a message passing interface.

"update" marks the current version of the record as invalid and inserts a new version of the record into the relation. Out-of-date records are removed (or archived) by a background garbage collector process. Guarding is implemented below the level of the POSTGRES storage system and does not take advantage of its no-overwrite property.

POSTGRES is extensible, so code implementing user-defined operators, access methods, and data types can be added to the DBMS. Most extension code will access the database through routines in the core POSTGRES modules. Generally, the core POSTGRES routines, not the extension code, must implement the POSTGRES support for guarding. Some extensions, however, such as user-defined access methods, have their own page formats. These extensions have to know about and use guarding directly. For example, B-tree access methods had to be modified to unprotect pages before adding or deleting keys.

The Sprite operating system, which we modified to support guarding, is a UNIX-like distributed operating system being developed at Berkeley. We chose Sprite as a test bed because the source code was available and well-documented. A DECstation 3100 served as a hardware platform for the guarding experiments. [1] It uses a software-loaded, hash-based Translation Lookaside Buffer (TLB). The guarding implementation does not rely on any DECstation 3100 hardware characteristics. However, the cost of updating TLB entries is hardware-specific and will be reflected in the cost of guarding.

3.2 Previous Work Related to Guarded Data Structures

Now that the guarding mechanisms have been described, we can compare them to similar mechanisms used by other systems. An alternative to protecting shared data structures with guarding is to keep those data structures in one address space and the clients of the data structures in another. In order to make such an architecture practical, a fast cross-address-space procedure call mechanism like that of the Taos operating system [11] is required. The Taos Lightweight Remote Procedure Call (LRPC) is optimized for RPC-style communication in which only a few parameters are passed between caller and called routine. The Service Request Block (SRB) mechanism in the MVS/XA[35] operating system is similar to LRPC. An SRB is a high priority thread of control which can be created in a remote address space. Both LRPC and SRB use a fast path through the scheduler and some shared memory to reduce overhead.

Guarding provides the same kinds of protection against non-malicious damage as does an address space boundary. However, access to read-only records is faster than would be possible in a separate address space implementation. Since database workloads often require the DBMS to scan through large amounts of data before selecting some for update,

[1] DECstation is a trademark of Digital Equipment Corporation.

faster read performance is a distinct advantage.

Tandem's process pair mechanism [8] also relies on multiple address spaces to prevent propagation of software errors. The Tandem data manager has a primary and "hot spare" process executing at the same time on different machines. The primary executes all transactions and sends checkpoint messages to the spare. If the primary fails, the spare can reconstruct the data manager's state from the checkpoint messages. While errors might propagate within the primary, they are less likely to propagate to the spare.

While process pair prevents the same kinds of errors as guarding does, it is much more expensive. Keeping the spare up to date requires resources for sending and processing checkpoint messages. Worse, the implementation of the checkpoint protocol is non-trivial. Modifications to the DBMS may affect the checkpoint protocol, making them expensive to implement and test. Finally, the model does not help detect errors. The primary and spare both have large, unprotected buffer pools. An undetected pointer error can damage a buffer without making the primary turn over control to the spare. The corrupted buffer will eventually corrupt permanent data.

The 801 System [17] uses page protection bits to provide operating system support for DBMS locking and logging, rather than using page protection to increase fault tolerance. A data manager running on the 801 does not set locks explicitly. Memory management hardware detects a read or a write to an unlocked buffer and the DBMS traps to the operating system. The operating system then sets locks and implements physical logging of 128 byte subpages. To support fine-grain locking, the 801 memory management unit provides write-protection at subpage granularity. The same hardware would support subpage granularity guarding.

Unlike a system using guarded data structures, the 801 treats any attempt to write to one of its buffers as legitimate. By moving responsibility for locking from the DBMS to the operating system, the 801 is losing information available to the DBMS about which data is updated erroneously. If a bad pointer causes a write to an unlocked buffer, the 801 locks the buffer and logs it normally. Under the same circumstances, a guarded system would immediately halt the transaction.

Implementing protected operations such as locking in the operating system is one alternative to guarding. However, installing the DBMS code in the operating system makes the operating system vulnerable to errors in the installed code. Guarding gives the DBMS implementor more freedom to decide what code is reliable enough to have access to protected data. More debugging support is available for user programs than for the operating system, so implementing protected subsystems in the DBMS is more practical than implementing them in the operating system.

Guarding provides some of the same protections as a protected subsystem mechanism without requiring any special hardware or restricting the designer's choice of programming

environment. Existing protected subsystem mechanisms often rely on special memory management hardware [62], [79], or type-safe languages [45]. Guarding can be implemented on conventional hardware and used with common systems programming languages. Of course, guarding is designed to protect against accidental damage not malicious damage. Existing protected subsystem mechanisms were designed to protect against both.

We chose to implement the virtual memory support required for guarding by modifying the operating system. It would also be possible to support guarding using the Mach external pager [80]. Implementing guarding directly in the operating system should make guarding more efficient.

3.3 Models for Updating Protected Data

3.3.1 Overview of Page Guarding Strategies

The basic idea in page guarding is that the DBMS write-protects its own data in order to detect accidental updates to that data. Clearly, any attempted update to read-only data is illegitimate, so write-protecting such data will prevent all errors from corrupting it. When data can be legitimately updated, the guarding implementation must allow the DBMS to disable guarding and overwrite the protected data. POSTGRES can use guarding to protect either its buffer pool or all of the shared memory region shown in Figure 3.1. The different models presented in this section allow the DBMS to enable and disable write protection in different ways. Each model will make different tradeoffs in terms of the kinds of errors it protects against and its performance impact. Before going into the model tradeoffs and implementation details, we present two examples that outline the models and show how guarding would work in practice in an extensible DBMS.

A Simple Example

The basic guarding models will all be described in the subsections that follow in terms of this simple example. The example assumes that the DBMS has only guarded the DBMS buffer pool. In the example, the DBMS runs a simple Postquel query such as:

```
replace (emp.salary = emp.salary * 1.1)
    where emp.name = "Mike Stonebraker"
```

which gives Mike Stonebraker a ten percent raise. To execute this query in the simplest case, the DBMS scans the employee relation examining the "name" field of each record for "Mike Stonebraker." In POSTGRES, records are stored on the disk in database pages and buffered in a main memory buffer pool. To examine the records on a given page,

the DBMS executor asks a buffer pool manager to determine if the page is currently buffered. If it is not buffered, the buffer pool manager reads the page into the buffer pool, replacing an existing page if necessary. When Mike Stonebraker's employee record has been located, the executor calculates the new salary value using the record and calls a lower-level "replace" operation. The replace operation installs this new salary value into the record. In POSTGRES, replacing a value in a record is done logically rather than physically by creating a new version of the employee record containing the new salary value.

Each of the guarding models has a different effect on the implementation of the POSTGRES replace operation. In the first guarding model, *Expose Page*, two system calls called *UnguardPage* and *GuardPage* are used to change write access to protected data. These allow the DBMS to change protection at the finest granularity supported by the underlying processor architecture. To change the salary in Mike's employee record, the page containing the record is unprotected at the beginning of the replace operation using *UnguardPage* and protected again at the end of the replace operation *GuardPage*. The *Expose Segment* model looks to the DBMS much like the Expose Page model, but the underlying implementation is different. Because the implementation is different, the protection/performance tradeoffs are different also. Details will be presented later in the chapter. In the Expose Segment model of guarding, *ExposeData* and *HideData* are used to obtain and remove write access to protected data instead of UnguardPage and GuardPage.

The remaining model, *Deferred Write*, does not change the buffer pool protection during the replace operation, but instead defers the protection change until the end of transaction. In this model, the POSTGRES replace operation creates a temporary version of Mike's updated employee record in a scratch area of the DBMS address space and links a pointer to the temporary version into a list of deferred updates. At the end of the transaction, the DBMS passes through the linked list installing each of the updates into protected memory with a single system call, *InstallData*. The buffer pool data structures are modified during the replace operation, so that if the transaction rereferences Mike's employee record, it sees the updated temporary version rather than the out-of-date protected version. Again, the implementation details and advantages of this technique are described in the sections that follow.

What Can Guarding Strategies Achieve in an Extensible DBMS?

The query in Figure 3.2 helps illustrate why guarding should be both inexpensive and effective in an extensible database management system. The hypothetical database in the example is a mixture of relational data and non-relational molecule data, designed for commercial pharmaceuticals research. The query uses a relational operator and a molecule-

```
append available_markers (id = molecule.id,
    expire_date = molecule.patent_date+''15 years'',
    etc.)

where
   (molecule.patent_date < ''January 1990'')
       AND
   (molecule.has_benzene_ring == TRUE)
       AND
   (similarity(penicillin,molecule.structure) > 0.90)
       AND
   (similarity(root-beer,molecule.structure) < 0.05)
     ...
```

Figure 3.2: Example of Extensible DBMS Query. The figure shows a query against a database that has been extended to handle molecule data. The function *similarity* is a hypothetical graph matching function that determines how similar two molecules are and returns a value between 0 and 1.

oriented extension operator called *similarity*. When a record is selected by the query, a conventional relational update is used to save or update the resulting records.

The DBMS query can be divided logically into two phases: a *qualification phase* in which operators determine which database data to update, and an *update phase* in which the selected records are modified or created. In its qualification phase, the DBMS passes over the data, applying a combination of extension and relational operations. During qualification, the DBMS does not need permission to write to the database data that it is examining. During the update phase, this permission is needed, but the DBMS applies a different, and possibly more trustworthy set of functions and/or operators. In the example, the update operations are fairly unsophisticated integer operations while the qualifications are extension operations.

Guarding support allows the DBMS to explicitly identify its qualification phase, telling the operating system through a set of system calls that any operator writing to the database at this time is in error. The qualification could still have bugs; it could, for example, qualify the wrong record. It could also corrupt a value in unprotected memory which is later, in

the update phase, used to generate a value stored in protected memory. However, these are much more benign errors from the standpoint of error propagation than addressing errors that "randomly" corrupt records in the buffer pool during qualification. For one thing, both of the errors mentioned can be undone if the transaction aborts since the transaction system logged the errors before allowing the updates. If a stray pointer corrupts the buffer pool, on the other hand, it does so without logging the change. Also, if these errors involve DBMS extensions, data unrelated to the extensions is unlikely to be corrupted by the error. Uncontained addressing errors can affect entirely unrelated data.

The remainder of this section discusses three different models that the DBMS could use to support the guarded data abstraction. Each subsection that follows describes one of the three update models.

3.3.2 The Expose Page Update Model

In the *Expose Page* update model, a DBMS process unguards a record before writing to it and reguards the record after the write. Because write-protection is enforced in hardware at page granularity, unguarding one record also unguards all of the records on the same page. The page granularity of guarding does not imply page granularity for transaction locks, since transaction locks are enforced by software.

Managing protected data in the buffer pool using this model is straightforward. When the data manager updates, inserts, or deletes a record on a buffer page, it unprotects the page with a system call. While the page is unprotected, data in the record can be changed or additional records can be allocated on the page. The *UnguardPage* system call clears a write-protection bit in the page table entry (PTE) associated with the page containing the data. UnguardPage also clears protection in the hardware TLB entry associated with the page. The *GuardPage* system call restores the protection bits in the page table and TLB entry.

After the DBMS has updated a record, it does not necessarily have to reguard the record immediately. If the DBMS delays reprotecting the data, subsequent updates to the same record do not pay the costs of turning page protection on and off. Unfortunately, the longer the page remains unguarded, the less protection is offered. Delaying the reguard operation also increases the opportunity for the DBMS to "forget" to reguard the page. Our implementation unguards one record at a time, reguarding each record before updating the next. If two POSTGRES processes unguard the same page at the same time, the last one to reguard the page issues the actual GuardPage call.

In the Sprite shared memory implementation, unguarding a page for one DBMS process unguards it for all of the others as well. Sprite uses a single software page table for each shared memory segment. When UnguardPage clears the protection bits for a page, all

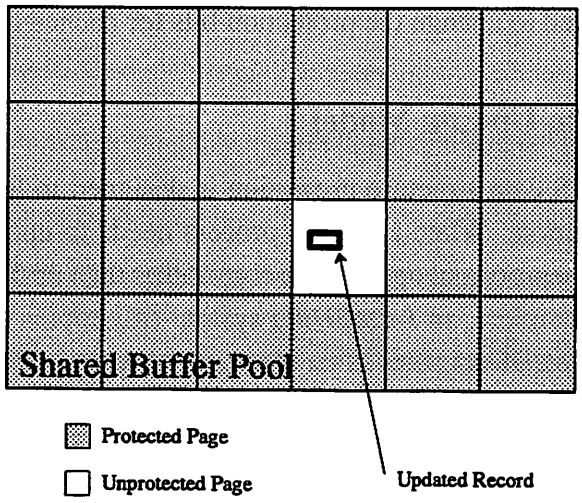

Figure 3.3: Expose Page Update Model. The smallest granule of hardware write protection containing the record of interest is unprotected before the record is updated. For most architectures, this unit is a page.

POSTGRES processes can write to the unprotected page. Thus, while one process updates the page, faulty code executed by another process can corrupt it.

A *GuardedRead* system call helps reduce the vulnerability of buffer pool pages by allowing them to remain protected during an I/O operation. The DBMS uses the GuardedRead system call in place of the normal *read* system call to load pages from disk into the buffer pool. In the absence of an explicit GuardedRead call, POSTGRES would have to unprotect the page before issuing the read. The page would remain unprotected for all DBMS processes until the read completed and the issuer reprotected the page. In GuardedRead, the operating system turns off page protection briefly while data is copied from system buffers into the user address space, rather than leaving it off during the entire I/O.

Expose Page is best for detecting pointer errors affecting pages containing infrequently updated records. "Hot" pages containing frequently updated records will be unprotected much of the time, so they will receive less benefit from guarding than cold pages. The major costs associated with Expose Page are an increased number of system calls and the additional TLB operations required to change page protections. If guarding were implemented on a processor with a virtually-addressed cache, changing page protection status from read-write to read-only would require the page to be flushed from the cache. Virtually-addressed caches store protection bits in the processor cache with the cached data. The protection bits can only be changed by reloading the cache line from memory. Hence,

a cache flush is normally required to change the protection bits for cached data.

3.3.3 The Deferred Write Update Model

The second model of DBMS data structure protection is designed to leave the record guarded until the end of transaction. When a DBMS process needs to update a record, it copies the record into writable memory and updates the copy rather than updating the record in place. After the update is complete, an *InstallData* system call copies the new record value into the protected page. InstallData takes as an argument an array of <source address, destination address, length> triples, so several records can be installed with a single system call.

InstallData combines an UnguardPage operation and a GuardPage operation into a single system call, so the user-level process never modifies protected memory directly. In InstallData, the operating system changes the TLB entry for the page containing the protected version of the record, copies the new version of the record into the page, and reprotects the page. Unlike the Expose Page model, *Deferred Write* does not modify the page table entry, just the entry in the TLB. As Section 3.2.2 has explained, processes can share page table entries, so modifying the page table entry disables protection for all of the DBMS processes that share the page. Because processes do not share TLB entries, the protected page is not vulnerable to errors in other POSTGRES processes during the install operation.

The reason that InstallData does not have to modify page table entries is that only the operating system ever has write access to the protected data. Page table entries are used to create TLB entries; the protection bits in the page table entry determine the protection bits in the corresponding TLB entry. Modifying protection in only the TLB entry allows access to a page until a TLB flush occurs or the entry is replaced in the TLB. When the page is referenced again, a new TLB entry is constructed and the page becomes protected again. To mask protection faults in this case, InstallData sets a copy-in-progress bit in the process control block before copying a record into a protected page. If a protection fault occurs due to a reconstructed TLB entry, the fault handler will use the copy-in-progress bit to detect that fault was spurious. It then unprotects the TLB entry and allows the write to proceed. Because the operating system copies records into protected pages in a tight loop, TLB entries will rarely be replaced and the extra protection fault will occur too infrequently to affect performance. The copy-in-progress bit is cleared and the TLB entry is reprotected before the DBMS process returns from the InstallData system call.

As in the Expose Page model, Deferred Write offers the DBMS programmer some latitude in deciding when to install the new version of the record into shared memory. The updated record could be reinstalled immediately after the update. It could also be installed

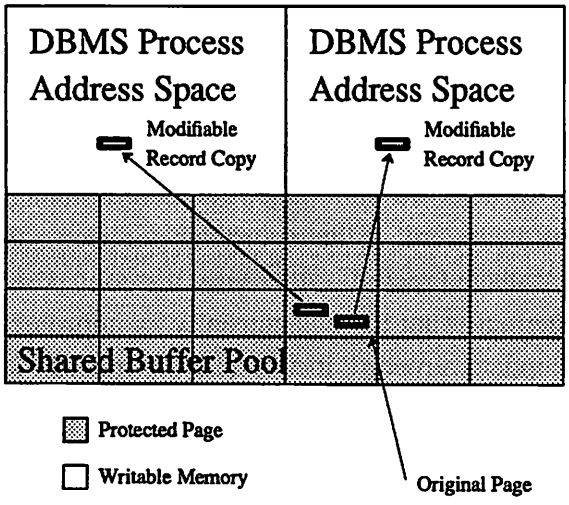

Figure 3.4: Deferred Write Update Model. A record is copied to writable memory before it is updated. Later, it will be copied back into protected memory using an *InstallData* system call.

after several updates or at transaction commit time. In our implementation of the Deferred Write model, guarded records are installed at transaction commit time.

Deferred Write is designed to work with record-level locking. Records from the same page may be updated concurrently by different DBMS server processes as is shown in Figure 3.4. When a DBMS server process copies a record to its private memory, it locks the record but not the page containing the record. While the latest version of the record is in one process's private memory, that process holds a transaction-duration lock on the data. The update is installed at transaction commit time before the lock is released.

Although updates to *data* on a page can be deferred until the end of a transaction, record-level locking requires undeferred updates to the page header whenever a new record is created on a page. A counter in the page header describes the amount of free space on a page. The DBMS must decrement this counter when a new record is added. When record-level locking is used, concurrent transactions are allowed to create records on the same page. Thus, changes to the free space counter must be immediately visible to all DBMS server processes. When allocating records on the page, the DBMS can use the InstallData system call to update the free space counter, but cannot defer the update until the end of the transaction.

Before making an InstallData system call, the DBMS must check that the destination page is still present in the buffer pool. In long-running transactions, the disk page from

which an updated record was taken could have been evicted from the buffer pool. If a record must be installed in a page that is no longer in the buffer pool, the DBMS reads the page back into memory before installing the data.

Some modifications to the POSTGRES record manager were required to support Deferred Write. If the DBMS asks for a record on a page, the record manager has to see if there is already a writable copy of the record. If the record has not been copied, the record manager returns a pointer to the protected record. Otherwise, the copy is returned. A hash table tells the record manager whether or not there is currently an unprotected copy of the record. If the DBMS decides to update a record, it first tells the record manager to make sure the record is writable. The request to make a record writable is logically at the same place the DBMS would lock the data. Hence, the existence of copies did not cause radical changes to the DBMS software.

While Deferred Write has a higher impact on software architecture than Expose Page, it provides more protection to guarded records than the Expose Page model does. Deferred Write updates protected records during a system call, so the DBMS can never store into a buffer pool page without issuing an InstallData system call. Addressing errors are unlikely to cause the DBMS to "accidentally" call InstallData. They can still damage the writable copy of a record before it is installed into the buffer pool. They can also damage the meta-data that tells where the record will be installed in the buffer pool, causing it to be installed in the wrong place.

Combining Deferred Write with a little additional error checking reduces error risk further. The DBMS currently checks that the update to be installed by an InstallData does not cross record boundaries before issuing the system call. Deferred Write also allows the DBMS to check for addressing errors that corrupt storage nearby the record modified. When the modifiable copy of a record is created, the DBMS can put known bit patterns before and after the copy. Some addressing errors which occur near the record can be detected by looking for corruption of these known bit patterns. In a conventional system and in Expose Page, these "nearby" addressing errors would be undetectable.

With Deferred Write model of guarding, corrupting the record directly (as in data errors) or installing the update to the wrong place on the page are the most likely ways of corrupting the protected data. At some additional cost, even these errors could be detected. The DBMS could checksum the record and its associated meta-data when the record is modified. By recalculating the checksum before installing the record into the buffer pool, the DBMS would be able to detect some of these additional addressing errors.

Deferred Write has an additional advantage over both Expose Page and conventional DBMS transaction management. When bad software corrupts data, often the damage is not detected immediately. By the time the DBMS notices the error, it cannot tell how much data has been affected; the faulty code that halted the system could have caused a large

cluster of undetected errors. With guarding and Deferred Write, however, the DBMS knows that protected data cannot be corrupted until the InstallData system call at the end of the transaction. If a transaction detects that it has corrupted some of its data, it simply throws away all uninstalled data. Any undetected damage to data records caused by the transaction will be thrown away as well. When record-level locking is used, the free space counter on a page can be modified during the transaction, but only a limited portion of the DBMS ever changes the free space counter. Thus, a limited amount of error checking ensures that data in the buffer pool is not damaged by the failing transaction, even if the extent of propagated damage is unknown.

A conventional DBMS aborts the current transaction when an error is detected and hopes that abort processing removes the effects of undetected errors. Aborting the transaction will remove the damage only if the erring software accurately recorded its updates in the log. Some errors, like those caused by corrupted pointers, corrupt data without logging the before-image of the record into the log. The most practical way for a conventional DBMS to get the same guarantee as the Deferred Write update model is to invalidate the entire buffer pool after detecting an error.

Page Remapping Techniques for Large Objects

Deferred Write is similar in some respects to the shadow paging technique used in System R [51]. Shadow paging is a no-overwrite transaction management technique in which a new block on the disk is allocated for every page modified by a transaction. When the page is evicted from memory or forced to disk, it goes to the new location. The update is committed by remapping the new page into the original page's position in its home relation. Shadow paging has fallen into disfavor as a recovery management technique because it prevents relations from being allocated on disk in keyed order. Thus, scans of the relations lose the performance advantage of sequential disk reads. While shadow paging and Deferred Write are superficially similar, shadow paging was not used in conjunction with write protection in System R and did not provide the error detection benefits of Deferred Write. Also, unlike shadow paging, Deferred Write does not affect the allocation of the database pages on the disk, hence does not hurt sequential read performance.

An in-memory variation of shadow paging could be used in conjunction with guarding to limit copying costs for large objects. For small record sizes, the cost of copying a record to a writable location and copying it back may not be significant, but as the record size rises so do the copy costs. When objects are large, remapping the DBMS buffer pool meta-data can reduce copy costs.

Instead of copying a large, possibly multi-page object to writable memory, a region of (shared) protected memory is unprotected and the pages containing the object are copied

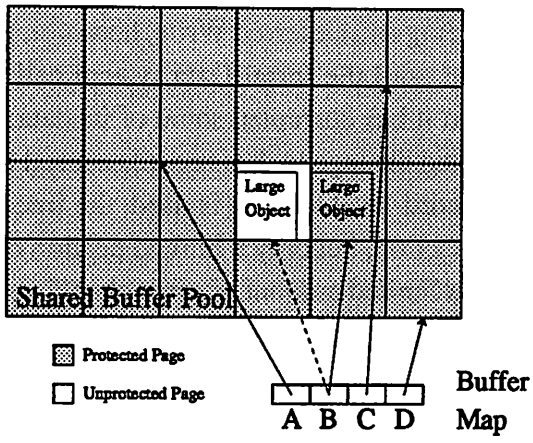

Figure 3.5: Remapping to Avoid Copies in Deferred Write. Page B contains a large object. Instead of updating the object in private memory, an unused page of the buffer pool is unprotected and the object is copied there. After the update is complete, the new version of page B is protected, the buffer map is changed, and the old version of page B is freed.

there (See Figure 3.5). Because it is unprotected, the copy of the object can be updated in place. To commit the updates to the object, the DBMS reprotects the page and changes the buffer map, which associates disk blocks with their location in the buffer pool. The pages that contained the original version of the object are now freed for use in further updates. For higher performance, the original version's pages could be unprotected in the same system call that protects the new version's pages. The freed pages will then be already unprotected when they are needed for the next update.

The remapping variation of Deferred Write is only cost effective when the object updated is large relative to the size of a database page. In normal Deferred Write, updating a protected object requires the DBMS to copy the object twice. The first copy occurs when the original version of the object is copied into unprotected memory. Second, after the object is updated, the new version is copied back into protected memory. The remapping variation of Deferred Write incurs two costs in place of the second copy. First, there is a small cost to change the buffer pool meta-data after the update. Second and more important, the entire page containing the updated object must be copied into an unprotected page before the update occurs, rather than just the object. If the object being modified is small, the cost of the single page-sized copy is larger than the cost of copying the object twice.

3.3.4 The Expose Segment Update Model

The *Expose Segment* update model is similar to the Expose Page model, however, protection is added to or removed from all guarded pages at once. When the DBMS makes an *ExposeData* system call, all protected data becomes visible. A second system call, *HideData* returns the protection to all exposed data.

Expose Segment provides less protection than the other two models since nothing is protected from the routines which update critical data structures. The reason for using the Expose Segment model is that it simplifies the management of guarded data in some modules. Using the Expose Segment model, a DBMS programmer can unprotect data for a procedure and its descendants in the call tree without knowing exactly which protected pages will be written. For POSTGRES, we found the Expose Segment model to be convenient for small, fast, and trustworthy operations that needed access to data on several pages. For example, we used it to protect a shared memory hash table in the implementation of the lock table.

To further simplify programming in the Expose Segment model, we use a pre-processor to place calls to ExposeData and HideData in procedures. The DBMS programmer flags with a keyword any procedure which is to update protected data. The pre-processor adds ExposeData and HideData calls at the first line and before all return statements in the targeted procedures. The pre-processor eliminates a class of errors in which data is never hidden again after an ExposeData call. It also makes adding protection to new data structures very easy.

To implement the Expose Segment update model in Sprite, we modified the operating system routine that handles write-protect faults. The ExposeData system call sets a "trusted" bit in the DBMS process's control block indicating that the process has permission to update protected data, but no page table and TLB entries are changed. When the process tries to update protected data, it takes a "false" protection fault. The operating system fault handler distinguishes true and false protection faults by examining the trusted bit in the process control block. On a false protection fault, the operating system clears the protection bits from the page's TLB entry and the process proceeds with the update. When the data is hidden again, the trusted bit is cleared and the mappings for any guarded pages still in the TLB are returned to read-only status.

The simplest approach to restoring page protection during HideData would be to flush the TLB, but flushing and reloading the TLB is expensive. Our implementation maintains a small log in the process control block containing page numbers whose TLB entries have been unprotected. The HideData system call passes through the log and resets the protection bits in the TLB entries corresponding to the logged page numbers. If the log ever overflows, the entire TLB must be flushed to reprotect the exposed pages.

The Expose Segment model of guarded update is similar to a conventional protected subsystem. Other protected subsystems (the operating system kernel, for example) require more complicated mechanisms since they are expected to prevent malicious as well as accidental damage.

A slightly less safe version of Expose Segment can reduce the high system call overhead inherent in this model. If the DBMS needs to update a single protected page, the Expose Segment model forces it to enter the operating system three times. The DBMS process first makes an ExposeData system call. Second, it takes a false protection fault when it attempts to update the protected page. Finally, the DBMS process makes a HideData system call to restore protection to the page. The Deferred Write model requires only one system call and Expose Page requires two.

The ExposeData system call could be eliminated to improve performance. This system call is only necessary to inform the operating system that the DBMS process is placing itself in trusted mode; it sets the trusted bit in the DBMS process control block. The DBMS could put the trusted bit in its own address space if, at system initialization time, it identified the address of the trusted bit to the operating system. Now, instead of making a system call to expose the segment, the DBMS process would set the trusted bit in its own address space. When the operating system handles the false TLB protection fault later, it looks for the trusted bit in the reserved area instead of the process control block. The HideData system call is still necessary since it updates TLB entries to remove write permission on the protected data. This variation of Expose Segment is less safe since it is possible for the application to "accidentally" go into protected mode by corrupting the trusted bit.

3.4 Performance Impact of Guarded Data Structures

Because the DBMS and operating system have to do extra work during updates of guarded records, guarding will decrease DBMS performance for update-intensive workloads. The extra costs involved in guarding include the additional system calls and TLB operations required to change page protections. In the Deferred Write update model, additional processing is required to create and keep track of record copies. This section evaluates the performance of guarding in two ways. Section 3.4.1 presents some of the raw costs of accessing protected data in all three guarding models. Section 3.4.2 shows the impact of guarding on the overall performance of a DBMS running a debit/credit workload.

3.4.1 Performance of the Guarding System Calls

Table 3.1 shows the raw costs of the guarding system calls: UnguardData and GuardData from the Expose Page model, InstallData from the Deferred Write model, and ExposeData

System Calls	Elapsed Time ± Std Dev
UnguardData	62.2 μs ± 0.6
GuardData	63.0 μs ± 0.4
InstallData	74.5 μs ± 0.4
ExposeData	21.6 μs ± 0.4
HideData	21.2 μs ± 0.4

Table 3.1: Raw Costs of Guarding System Calls. These are the elapsed times in microseconds of the five different system calls added to the DECstation 3100 version of Sprite to support guarding. Each entry in the table is the mean of five measurements and a measurement is the mean of 10,000 system calls.

and HideData from the Expose Segment model. These measurements were taken on a DECstation 3100 version of the Sprite operating system augmented with guarding support. Each entry in the table gives the mean and standard deviation of five measurements. Each measurement is the mean of 10,000 system calls. In InstallData, only a single byte of protected data is modified in order to limit the effect of data copying overhead, which is not present in the other system calls.

The costs of ExposeData and HideData as shown in this test can largely be attributed to Sprite system call overhead. ExposeData simply sets a bit in the process control block and returns. HideData checks that no pages have been unprotected and clears the bit. UnguardData and GuardData are slower than ExposeData since they must operate on the DECstation 3100's Translation Lookaside Buffer. The measurements show that GuardData is slightly slower than UnguardData. The system calls are identical except for the bits that are loaded into the TLB, so if this difference is actually significant, it is a feature of the hardware not the software. InstallData is the slowest of these system calls, but it is much less expensive than UnguardData and GuardData combined. Since InstallData is logically a combination of these two operations, we can see that there is a performance advantage to combining the unguard and guard operations into a single system call.

The graph in Figure 3.6 shows the cost of updating a small record on a protected page in each of the models. The X axis in the figure is the number of bytes in the record and the Y axis is the elapsed time in microseconds. As in Table 3.1, each measurement is taken from the elapsed time of 10,000 operations, where an operation copies a record into guarded memory. Each data point on the graph is the mean of five measurements and the standard deviation for these measurements is always less then 2% of the mean. The

graph also includes curves showing the cost of a simple bcopy into unprotected memory and the cost of copying a record from one address space to another using UNIX pipes. Pipes are not the fastest possible interprocess communication mechanism, however, these measurements give a reasonable comparison between protecting data structures through guarding and protecting them by maintaining separate address spaces for a protected data structure and its clients.

All five curves in Figure 3.6 have the same slope, determined by the cost of copying the bytes in the record. The basic overhead for each of the four protection models shown differs significantly. The multi-process model has the highest overhead. This is probably dominated by context switch time. Expose Segment is the next most expensive, but it is still less than half the cost of the multi-process model. It is more expensive than the other guarding models because the DBMS must trap to the operating system three times per record modified in this model. A system call is required to expose the protected data and to hide it again. Then, the data manager faults to the operating system one more time when it first refers to the data. Expose Page is less expensive than Expose Segment because it only enters the operating system twice: once to unguard the data and once to guard it again. It is more expensive than Deferred Write, because Deferred Write enters the operating system only once.

3.4.2 Guarding in a DBMS with a Debit/Credit Workload

The microbenchmarks described in the previous section do not give a complete picture of the cost of guarding. In order to measure the impact of guarding on a full system, we compared several different versions of POSTGRES, each with a different protection strategy, using a workload based on the TP1 debit/credit benchmark [1]. In our version of this benchmark, two thousand transactions were run against a small database. Each transaction retrieves a tuple from an account relation, updates the account relation and two other smaller relations (branch and teller), and appends a record to a fourth relation (history). Account has 10,000 records and is 200 pages long. In this benchmark, Branch has one record and Teller ten, so each is only one page long.

We measured guarding under both a CPU-bound and a disk-bound workload. In the CPU-bound benchmark, POSTGRES operates on the benchmark database without forcing its updates to disk at commit time. The benchmark database is small in order to allow the DBMS to store the entire database in main memory. Because the database is small and updates are not forced to disk, the CPU-bound benchmark does no I/O operations at all and saturates the CPU. To make the benchmark disk-bound, we turned force-at-commit back on. The resulting I/O operations bring CPU utilization down to about 25 percent. Both benchmarks were run single-user on a DECstation 3100 implementation of the Sprite

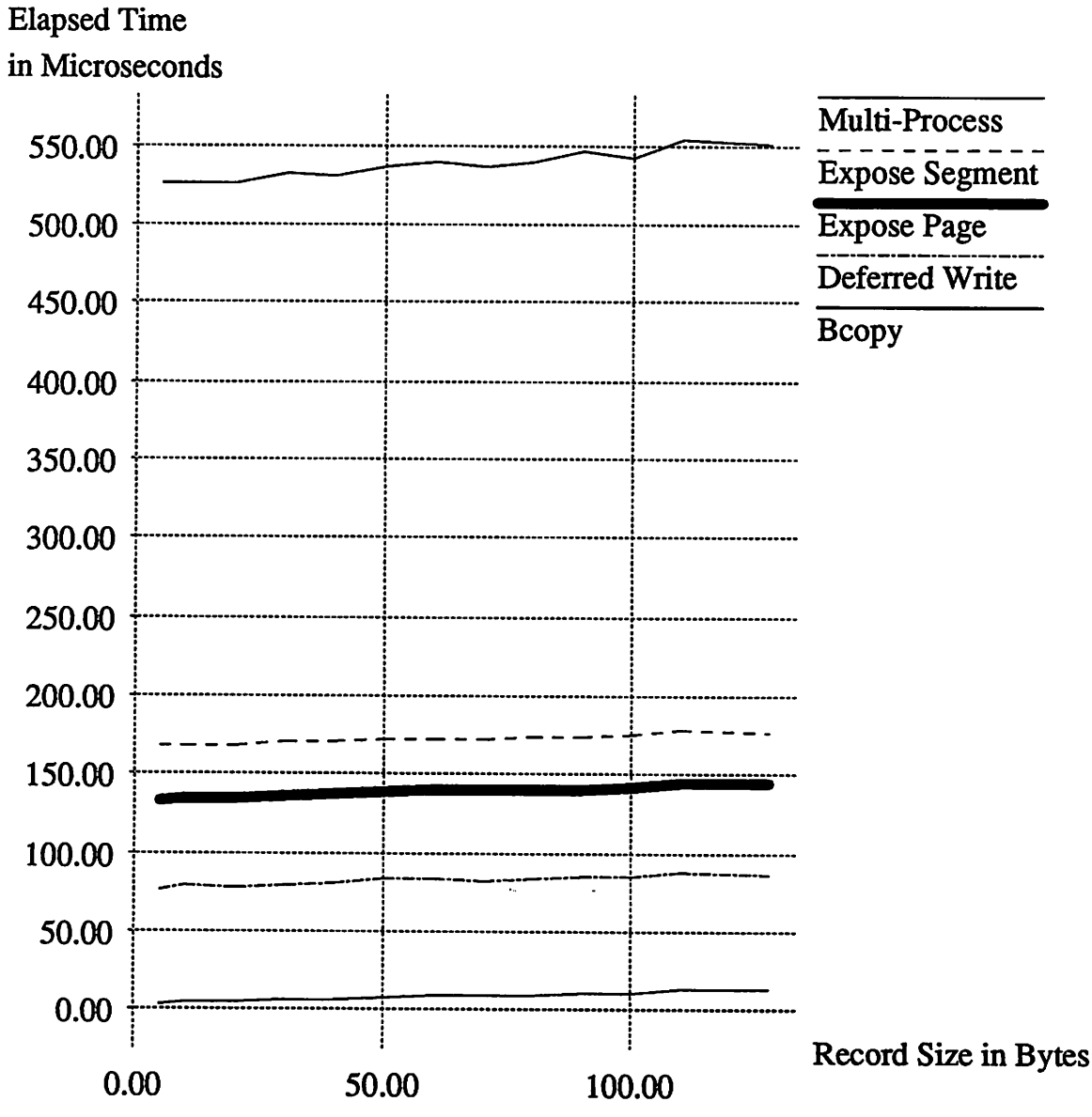

Figure 3.6: Costs of Updating Protected Records. This graph shows the cost of updating a protected record using the Expose Page, Deferred Write, and Expose Segment models of guarding. They are compared to a multi-process protection mechanism in which a data structure is protected from its client by placing it in a separate address space from the client. In the Multi-Process model, interprocess communication is through UNIX pipes. The graph also shows the cost of unprotected access to the record through a simple bcopy.

operating system.

We compared six different versions of POSTGRES to a normal version with no guarding support. The *unprotected copy* version used the Deferred Write update model but did not protect the pages. Comparing the unprotected copy POSTGRES to normal POSTGRES shows the overhead in Deferred Write attributable to copy management, but not to write protection. Three POSTGRES versions each use a different one of the update models described in the paper. The *read-only queries* version was actually a modified version of the benchmark run with Expose Page guarding. This version is just a sanity-check to show that guarding does not impose any costs when records are not updated.

The last POSTGRES version, *full protection*, protects all of shared memory — including the lock table, some shared memory lookup tables, and the buffer pool. The *full protection* version uses the Expose Page update model to update data in the buffer pool and Expose Segment to update all other data structures.

Tables 3.2 and 3.3 compare the protection overhead for each of the six program versions. Each benchmark run of two thousand transactions was repeated five times to get an average elapsed time. If the standard deviation of the five elapsed times was greater than one percent of the average, the original five runs were discarded and all five runs were repeated. The tables present their results as the percent increase in the average elapsed time caused by the protection mechanism.

The two tables show that the least expensive model for updating guarded buffers is Expose Page. Expose Segment is slightly more expensive, again, probably because Expose Segment requires both system calls and a TLB fault to access protected data while Expose Page only requires system calls. In the disk-bound case, the costs of the different models are roughly the same. Since guarding does not affect disk accesses, it has a large impact only when there is high CPU utilization. As one would expect, the read-only transaction workload showed no additional expense due to guarding.

The software overhead required to manage record copies in Deferred write is apparently significant. The Deferred Write model has about the same cost as Expose Segment, even though InstallData is the cheapest guarding system call. Comparing the unprotected copy DBMS to the Deferred Write DBMS shows that much of the expense is related to copy management. From profile data, we have seen that nearly all of the copy management costs come from allocating, freeing, and searching for record copies in the copy hash table. Because records are small in the benchmark, physical copying does not affect performance.

The full protection version of the DBMS is much slower than the versions that only protected the buffer pool. This version requires a guarded-memory update whenever the process sets a lock or pins a buffer in the buffer pool. Since pins and locks are acquired more often than buffers are updated, the cost is higher.

The measurements in this section illustrate the costs of guarding in a system that uses

Update Model	Protection Overhead
Expose Page Guarding	7%
Read-only Queries	0%
Expose Segment Guarding	10%
Full Shared Memory Protection	87%
Deferred Write Guarding	11%
Copy costs only	6%

Table 3.2: Performance Impact of Guarding a CPU-Bound Version of POSTGRES. The CPU-Bound case was constructed by running a debit/credit benchmark on a database that was small enough to fit in memory. Without guarding, the DBMS ran about 10 transactions per second.

Update Model	Protection Overhead
Expose Page Guarding	2%
Read-only Queries	0%
Expose Segment Guarding	3%
Full Shared Memory Protection	5%
Deferred Write Guarding	3%
Copy costs only	2%

Table 3.3: Performance Impact of Guarding an IO-Bound Version of POSTGRES. The IO-Bound case was constructed by running the same debit/credit benchmark on the same small database, but forcing updates to disk on commit. The CPU utilization in this case is 25%.

memory management hardware available today. While these costs are not exorbitant, they will be too much for some high performance systems. The next subsection discusses ways in which changes to memory management units can reduce the costs of guarding so that even high performance systems can write protect data.

3.4.3 Reducing Guarding Costs Through Architectural Support

One of the advantages of the current guarding implementation is that it uses conventional memory management hardware, making it a practical tool for existing systems. However, if virtual memory management hardware were redesigned, the performance impact of guarding could be significantly reduced. A large part of the cost of our guarding implementation is the trap to the operating system required to change read/write access to protected data structures. The UnguardData, GuardData, and InstallData system calls also must copy arguments from user space to kernel space. Modifying the operating system and the virtual memory management hardware to allow unprivileged processes to protect and unprotect parts of their address spaces could bring the cost of guarding down by as much as forty percent (assuming system calls are 22 microseconds and argument passing takes about 5 microseconds).

Protection violations are detected by the address translation mechanism in the memory management unit of the processor. Usually, a bit indicating whether a page is writable is stored with the virtual-to-physical address mapping for the page. When the virtual address is translated to a physical address, the protection bit is checked to make sure that the address being stored into is writable. We have been calling the hardware that manages this mapping the Translation Lookaside Buffer (TLB), but implementations of the mapping hardware vary widely. The VAX has two levels of hardware tables mapping virtual address to physical address [49]. In the DECstation 3100 [40], this mapping is a hardware hash table entry. In a machine with a virtually-addressed cache such as the SPARC II, the information is stored with the cache line. Cheng [18] describes some of the expenses involved in managing protection changes in such an environment.

Usually, the same (supervisor-mode) instructions are used to change the TLB's virtual-to-physical address mapping as are used to change the protection bits. Unprivileged processes cannot execute these instruction since allowing unprivileged processes to change virtual-to-physical address mappings would be a security hole. If unprivileged access were allowed, any process could allow itself to address any part of physical memory. Modification of protection bits can be a security hole as well in UNIX systems since code segments are shared between processes. If a malicious user unprotected a shared code segment and modified the code, he or she could make other processes executing that shared code take actions unintended by the owners of those processes.

To allow unprivileged processes to guard and unguard data in their own address spaces quickly, the processor instruction set should include a separate, unprivileged instruction to store a protect/unprotect bit into a TLB entry. The TLB entry would have to have an additional bit and/or mode that allowed the operating system to protect some TLB entries from modification (e.g. code segments).

Even with hardware support, the operating system would have to cooperate with user processes in order to implement user-level guarding operations. TLBs can be flushed at any time by the operating system, for example, after a context switch operation. When the operating system reinitializes a TLB entry, it will do so using the protection information stored in the process page table. If a user process unguards a record using the new instruction, takes a context switch, and then accesses the unguarded data, it will fault; the operating system will have reguarded it after the context switch. Therefore, a user-level guard/unguard operation must not only physically change the protection of the data, but also save the new page status in a way that allows the operating system to determine that status during a TLB reload.

One can imagine many implementations of user-level unguard operations. For example, in the POSTGRES guarded buffer pool experiments, most of the buffer pool was guarded most of the time. Records were unguarded temporarily during updates, but then reguarded immediately, and only one page per DBMS process was ever unprotected at a time. An effective implementation for POSTGRES would be a system call with which the user program specifies a buffer containing a list of currently unprotected pages. The user-level unguard routine would keep the list up to date. After a protection fault on a guarded page, the operating system could check this buffer for the virtual address (or virtual page number) of a temporarily unprotected page. The protection fault will only occur if the TLB entry is lost between the unguard operation and the modification of the unguarded record.

3.5 Reliability Impact of Guarded Data Structures

The control/addressing/data error model presented in the introduction was designed to break errors into classes differentiated by their effects on guarded data. In order for guarding to detect errors, failing software must try to update protected data illegally. If broken software always managed to unguard data structures before corrupting them, guarding would not detect errors effectively. Guarding would also have no impact if software failures simply cause the program to halt without ever overwriting any data. From the error model and the data in Chapter Two, we can estimate how much impact guarding will have on software reliability.

Data errors would corrupt guarded data or cause the program to produce invalid results

in spite of the guarding protection, but, fortunately, these errors were uncommon. Data errors occur when the software calculates and stores the wrong data value. Guarding will not protect against these errors; the faulty DBMS code will simply turn off the protection and corrupt the data. The data in Chapter Two shows, however, that the assert statements and other standard debugging and antibugging techniques used in current systems do an excellent job of detecting data errors, limiting this risk to guarded data.

Control errors are also unaffected by guarding, but because they do not corrupt data, not because they turn off guarding. Control errors corrupt transient program state or cause deadlock, but do not directly overwrite anything. After a control error, the system only needs to reinitialize transient state and begin accepting transactions again. The secondary effects of the error sometimes involve addressing failures, however. For example, some control errors in the MVS study had "address trap" failure symptoms, meaning that the control error was detected by the system when the code tried to access unaddressable memory. While guarding will not detect control errors, it will limit the possibility of error propagation after a control error occurs.

Guarding will be most likely to detect addressing errors, such as uninitialized pointers. The studies in Chapter Two indicate that addressing errors make up twenty to thirty percent of recorded software errors. According to Chapter Two, however, addressing errors tend not to be the "wild pointer" errors that randomly corrupt data arbitrarily far away from the data that the failing module was using. When we could tell from the APAR which data structure was corrupted, 75% of the time the data structure was very near the data that the programmer intended to update. Guarding is unlikely to detect these addressing errors. "Wild pointers" represented only a quarter of the addressing-related errors; hence, the errors most likely to be detected by guarding make up make up about 5 to 7.5 percent of all software errors.

While guarding will not detect most software errors, reducing the number of software outages by even five percent will be extremely helpful in many environments. Chapter Two also showed that addressing errors have the highest impact on the customer, either because they caused the most serious outages or were the most difficult for the system to recover from. Moreover, even when the resulting outage is minor, addressing errors represent some of the most difficult software errors to find and fix. By the time the damage has been detected, the module containing the error is no longer executing. Anecdotal evidence from the development of POSTGRES and other systems suggests that much more than five percent of the system development effort goes into finding and repairing addressing-related faults.

3.6 Summary

This chapter describes modifications to the operating system and database manager which are designed to limit software error propagation in the DBMS. Write-protecting the data manager's buffer pool allows early hardware detection of addressing-related software errors. Guarding reduces the complexity of software failure by preventing errors from propagating to protected data structures. Guarding techniques can also improve recovery speed since limiting potential error propagation decreases the amount of work required at recovery time. While any DBMS could use these techniques, they are especially important to a extensible DBMS such as POSTGRES. With a guarded system, one person using (or developing) new access methods or data types has smaller impact on the availability and reliability achieved by his or her peers.

It is difficult to quantify the reliability improvements that will result from using guarding in commercial systems. Chapter Two showed that 25-30% of software errors in several existing systems are addressing-related. Only 25% of those were "wild pointers" that damaged parts of the system unrelated to the component with the error, though. This implies that guarding will eliminate about 5-7% of software errors. However, some of these software errors were among the most difficult to detect by ordinary means, so a 5-7% reduction in software errors may result in a much larger reduction in the engineering effort required to produce a reliable software system. These errors are also of higher than average customer impact, so the reliability increase perceived by the customer will probably be more than 5-7% as well.

In general, the performance impact of guarding is comparable to the impact of other software techniques for detecting software errors, such as data structure verifiers or array bounds checks. Guarding can be implemented efficiently by taking advantage of processors with software-loaded TLBs. For read-only workloads, guarding provides the DBMS with additional protection at no extra cost. For update-intensive workloads, experiments have shown that the additional CPU demand caused by guarding is only a few percent when small records are updated. Page remapping techniques could be used as a method for reducing copy cost for large records.

In deciding whether or not to guard data structures, system designers face a tradeoff between potential reliability and availability improvement and a small but measurable performance loss. For some systems, no reliability gain will be worth any loss in performance. Others may be willing to accept the small performance loss in order to achieve any reliability improvement. Still other systems may want the option of switching from guarded to normal operations at different points in the system lifetime or for different customers.

Over time, trends in system cost will probably tilt the performance/protection tradeoff in the favor of guarding. As processors become faster, the additional processing demands

caused by guarding will become less of a concern. The big potential risk to the long-term usefulness of guarding techniques is that the cost of changing page protection might not scale with processor performance. However, hardware designers have been made aware of the need for fast protection changes in other applications such as distributed shared memory [2], so, hopefully, they will consider this issue in future processor architectures. Meanwhile, the need for guarding will almost certainly increase over time. Falling memory prices are increasing the sizes of disk caches like the DBMS buffer pool. Some data in the cache will remain unused for long periods of time. It is essential that bad writes into this data, however infrequent, be caught at the time of the error rather than the first time the data is used. It is also essential for fast recovery that these gigantic caches not be reloaded from the disk after software failures. Finally, as non-volatile RAM becomes less expensive, it will be more likely to be more frequently used as stable storage by applications such as database management. Non-volatile RAM will never be as resistant to failure as disk storage without some protection from addressing errors.

Chapter 4

Fast Recovery in the POSTGRES DBMS

4.1 Introduction

A fast, simple recovery mechanism is critical to highly available data management in fault tolerant systems. As Chapter One pointed out, faster recovery leads directly to higher availability. Long software restart times lengthen the outages that occur after any kind of failure, and longer outages decrease system availability. Section 2.4.3 of Chapter Two illustrated the reliability risk due to recovery system software. Many software outages caused by control errors were related to recovery and error handling code. The data indicates that recovery systems are hard to implement correctly and hard to maintain. Testing recovery systems is also difficult since it requires test suite designers to anticipate failure conditions that will arise in the field. This is a daunting task in a large software system.

Traditionally, fault tolerant systems have tried to mask failures and avoid recovery rather than improve recovery speeds. For example, Tandem [8], Stratus [77], Auragen [14], Harp [50], XRF [36] and HA-NFS [12] all maintain a primary and one or more backup systems in order to avoid recovering when the primary fails. When a failure occurs, operation switches over to the backup system rather than delay users while the primary recovers. Unfortunately, the protocol for keeping backups up to date is expensive and its correctness is very difficult to verify. Also, even if the protocol works correctly, there is no guarantee that software errors will not propagate from the primary to the backup.

Another common approach to masking failures is to provide a multi-level software recovery mechanism. The Integrity-S2 [39] operating system attempts to correct internal data structures when it finds errors in them. If two failures occur within a few minutes, then the system assumes the correction did not work and goes through a full recovery. MVS [3] uses a multi-level recovery scheme in which different portions of the system can fail and recover independently. Another two-tiered recovery mechanism [6] implemented

in the Sprite operating system uses a reserved area of memory to hold backup copies of state associated with the distributed file system and distributed applications. In the event of control errors and most addressing errors, the backup state can be used for quick regeneration of operating system and application program state without disk operations or communication with remote sites. When power outages, hardware errors, or software errors corrupt the reserved memory, the normal, slow recovery path is used.

The POSTGRES approach to maintaining high availability is to improve the speed of system recovery after errors are detected. Failure is not masked, as is the case with hardware, but a fast recovery mechanism still improves availability by eliminating long outages after failures. The approach requires little to be done during recovery that is not done during a normal system restart, so the recovery system may be easier to debug and test than conventional multi-level recovery mechanisms. In contrast, most database management systems use write-ahead log (WAL) recovery techniques (surveyed in [33]). In WAL, all of the updates applied to the database are written to a log. The log is processed during system restart to ensure that no committed updates are lost and no aborted updates remain. After the WAL survey was published, ARIES [53] took many steps to improve the concurrency and restart performance of the basic write-ahead logging techniques, but increased the complexity of the recovery system software. Even in ARIES, database recovery time is proportional to the number of log records that must be processed during recovery. To significantly improve recovery times, log processing must be eliminated.

The work in this dissertation takes as its starting point the 1987 POSTGRES storage system, which uses no-overwrite techniques to combine support for historical data with support for transaction management [66]. The details of the no-overwrite storage system are left to Section 4.2, but, briefly, the storage system works by creating a new version of any tuple updated by the DBMS rather than updating the tuple in place. If the DBMS fails and the updating transaction aborts, the previous version of the tuple remains and can be used for recovery. Falling back to the previous version does not involve log processing, so the storage system requires little work at restart time.

While the designers of commercial database systems desire the faster recovery that is possible without write-ahead log processing, this community has not applied the POSTGRES storage system ideas to commercial DBMSs. The two most likely reasons for this involve recovery from media failures and performance considerations. POSTGRES assumes that the I/O subsystem handles media recovery, hence, it depends on either mirrored disks or RAID (Redundant Array of Inexpensive Disks [59]) disk subsystems. Traditionally, write-ahead logs have been used in media recovery for non-mirrored disks. Because RAID storage systems are now commercially available, this is becoming less of a problem. A more important reason that the POSTGRES storage system ideas are not widely used is that the original design does not perform as well as traditional storage systems when

the database must support a very high update rate. The data structures used to implement the no-overwrite transaction support in POSTGRES made retrieving tuples from such a database expensive. Also, POSTGRES must use a force-at-commit buffer management policy: all buffers containing tuples updated by the transaction must be written to disk before transaction commit. Most database management systems do not use this policy because it causes the DBMS to do much more disk I/O than would be necessary with a write-ahead logging policy. To increase the usefulness of POSTGRES' fast recovery techniques in applications such as banking and stock trading in which both high update rates and fast recovery are important, the performance impact of the storage system must be reduced.

Chapter Four of this dissertation makes four contributions to fast recovery in the database management system. The first two increase the applicability of the POSTGRES storage system in environments with high update rates, allowing these environments to take advantage of POSTGRES' fast recovery. First, the chapter suggests several changes to the way tuples are stored in POSTGRES. By changing the way that tuples are stored, we speed access to the data in the database. Section 4.2 describes the original POSTGRES storage system and the new optimizations. Second, Section 4.3 uses an analytic model to evaluate the I/O impact of the storage system on a RAID. It shows how non-volatile RAM and modern file systems such as the log-structured file system (LFS) [61] can eliminate the additional I/O costs associated with POSTGRES' no-overwrite techniques. Together, the techniques described in Section 4.2 and the analysis of Section 4.3 should increase the applicability of no-overwrite transaction support to applications with high update rates.

Chapter Four also considers recovery of several kinds of DBMS state that the original POSTGRES storage system ignored. When the DBMS recovers from a failure, it must reestablish four kinds of context lost during the failure:

(1) Disk Database Context: The database on the disk must be made transaction-consistent.

(2) Disk Cache: After a failure, the DBMS must reload frequently-accessed database pages into main memory.

(3) Session Context: Network connections between the DBMS server and its clients are lost during the failure. Reconnecting a client to the server means reauthenticating the client, reinitiating the network protocol, and determining if any messages were in transit at the time of the failure. In some systems, human intervention is even required to restart application programs after the DBMS server fails.

(4) Current Transaction Context: The transactions executing at the time of the failure had some transient state associated with them — for example, the query plan structures,

the lock table, and the temporary relations holding intermediate state. This state must be reinitialized.

The POSTGRES storage system addresses item (1) from the list. Sections 4.4 and 4.5 describe methods of recovering disk cache and session context, items (2) and (3) from the list, which were ignored in the original storage system. Regeneration of the current transaction context, item (4) in the list above, is left as future work. The issue of regenerating transaction context is not important when the DBMS only executes short transactions. In this case, the fastest, simplest way of recovering lost transaction context is to reexecute the aborted transactions. Strategies for reestablishing the transaction context of long-running transactions are outlined in the final chapter of the dissertation.

4.2 A No-Overwrite Storage System

The POSTGRES storage system differs from most other DBMS storage systems in that user data is not updated in place. Instead, POSTGRES creates a new version of the tuple and updates the new version. When a tuple is logically deleted, it is actually marked invalid and left physically in place. Instead of write-ahead log processing, POSTGRES recovers from failures by falling back to the previous version of the data. If the transaction is aborted, the DBMS detects and ignores any changes to the database made by the transaction. Even if the transaction commits, the updated tuple versions remain accessible to users as historical data. Because the new version of the data is physically located in the data pages, all data pages written by the transaction must be written to stable storage or non-volatile memory before the transaction commits. In [33], this policy for managing data pages is called **force-at-commit**.

The subsections that follow describe the POSTGRES Storage System and several enhancements to it. The first four subsections describe the most important issues affecting the performance and cost of the transaction system: (a) storage of tuple versions, (b) reclamation of space in data pages, (c) the run-time detection of invalid updates, and (d) access to historical data. The third of these subsections discusses the actual recovery mechanism. Much of the current section summarizes design points of the original POSTGRES Storage System design and is included here for completeness. Some changes have been made to improve recovery speed, to simplify parts of the storage system, and to improve performance. In other places, we describe details of the storage system that were omitted in [66]. The differences between the original POSTGRES storage system and the version modified for the dissertation will be identified as they arise. The version implemented for this dissertation is referred to as the "modified" version in the text.

4.2.1 Saving Versions Using Tuple Differences

In order for the POSTGRES no-overwrite storage system to make more efficient use of space, consecutive versions of the same tuple are stored as a sequence of **tuple difference records** rather than a sequence of full tuples. When a tuple is initially inserted into a relation, an **anchor point record** is constructed representing the full tuple. Subsequent updates are represented as difference records containing only the fields of the new tuple version that differ from the previous version. The difference records are chained together so that starting at the anchor point and following the chain will allow POSTGRES to reconstruct any version of the tuple.

The original POSTGRES storage system used a difference record management scheme based on **forward difference chains**. In forward differencing, the anchor point is the oldest available version of the tuple. The difference chain goes from the oldest available version to the newest one, hence, queries referring to the current version of the tuple must pass through the entire difference chain to construct the tuple (see Figure 4.1). As records are updated, the difference chain will grow and references to current data will become increasingly expensive.

The modified POSTGRES storage system used in this dissertation improves access to current data using **backward difference chains**. The anchor point in this case is the most recent version of the tuple. When an update occurs, a link is constructed from the newly-generated version to the current version in the difference chain. Because the current version of the tuple is readily available, scans and updates of the current database are fast.

Unlike the anchor point in a forward difference chain, the anchor point in a backward difference chain can contain fields from several different tuple versions. For example, if a transaction updates one attribute value in a four attribute tuple, as in Figure 4.2, the most recent version of the tuple contains fields from two difference records. Therefore, the anchor point is structured as an array with an element for each of the tuple's attributes. Each element points to the most recent value for the given attribute. Because of its array-style anchor point, backward differencing uses more space than forward differencing. Forward differencing simply used the oldest available tuple difference record as its anchor point.

It should be clear from Figures 4.1 and 4.2 that, while both forward differencing and backward differencing are logically no-overwrite techniques, the data on stable storage is physically overwritten after each update. Data is transferred between main memory and disk in page-sized units. When a new difference record is added to a page, the entire page is rewritten to stable storage. We assume that database pages are written to disk atomically except in the case of a media failure. We assume that this as well as other media failures is detectable. On devices (and file systems) in which page writes cannot be guaranteed to be atomic, POSTGRES or the operating system would have to checksum each page in

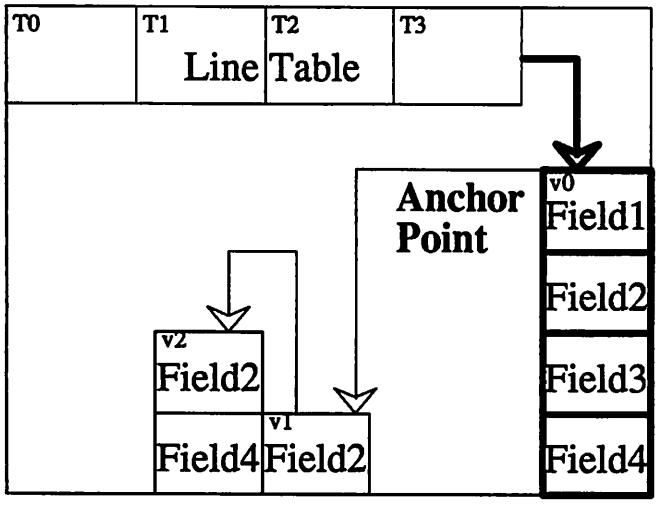

Figure 4.1: Forward Difference Chain. This data page contains four tuples, only one of which, T3, is shown. The line table entry points to the anchor point (in bold). The forward difference chain connects the records representing versions v0, v1, and v2. To construct the current version of a tuple, the DBMS starts with v0 and follows the difference chain.

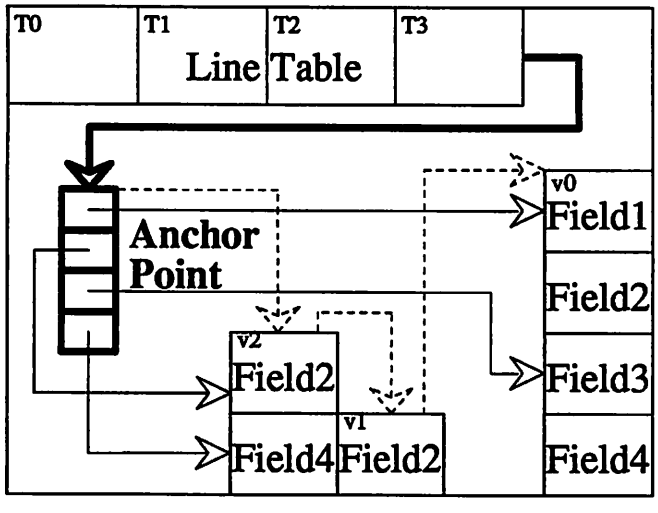

Figure 4.2: Backward Difference Chain. In this figure, T3 uses an array-style anchor point (in bold, as above) and a backward difference chain. The difference chain is shown as dotted arrows connecting the records associated with versions v2, v1, and v0. The anchor point array points to the youngest member of the chain and to the most up-to-date values of T3's fields. Since only Field2 and Field4 have been updated, two of these field values come from v0 and two come from v2.

software and examine the checksum every time the page is read from disk.

POSTGRES indices are described in depth in Chapter Five, but one detail regarding them is important to this section. Records in a POSTGRES index point to the line table entry on the data page rather than an individual record. Because the line table entry points to the anchor point, the index can be used to find any version of the tuple. Thus, the no-overwrite policy does not force the DBMS to update index records every time data records are updated.

4.2.2 Garbage Collection and Archiving

Tuple difference chains reduce the amount of space taken up by historical data, but the no-overwrite policy will eventually cause the database to run out of disk space without an additional strategy for reclaiming storage space. The original POSTGRES storage system allowed space to be reclaimed in three ways. First, any tuple versions created by transactions that later aborted can be garbage collected and removed from the database at any time. Second, historical data can be moved to a cheaper storage medium such as optical disk, freeing up space on the faster medium. Third, historical data older than a user-defined threshold can be destroyed. This section will, for ease of presentation, address garbage collection and the archiving/destruction of historical data as separate functions. The section occasionally refers to the garbage collector and the archiver as separate entities when, in fact, they are implemented in a single program called the **vacuum cleaner**.

In its garbage collection capacity, the vacuum cleaner examines each page of each relation in the database, reorganizing the page to eliminate tuple versions created by aborted transactions. A page is reorganized by first allocating a temporary page in memory, then copying all historical and current tuple versions to the new page. The copying is necessary because the invalid tuple versions created by aborted transactions are interspersed with valid tuple versions on the page. After the new page has been constructed, the DBMS buffer pool meta-data is modified so that the new page replaces the old one.

During garbage collection, the layout of the data page changes, but the contents of valid tuples on the page do not change. Therefore, garbage collection does not conflict with transactions' two-phase read and write locks on the page's tuples. If it did conflict, the garbage collector would have to lock tuples during garbage collection, reducing overall concurrency and allowing the garbage collector to deadlock with existing transactions. High concurrency during garbage collection is important since the most frequently updated relations have both the highest concurrency requirements and consume the most space if not vacuumed frequently.

While two-phase locks are not required, some coordination between the garbage collector and transactions in the DBMS is necessary because the DBMS process can have

pointers into the old version of the page. The DBMS must detect that garbage collection has occurred and revalidate these pointers before the old page is reallocated. When garbage collection completes, the garbage collector stores a pointer to the new version of the page in the buffer header structure associated with the old page. Whenever the DBMS re-examines a tuple, it checks to see if there is a new version of the page. If there is a new version, the backend process reassembles the tuple using pointers to the difference records in the new page and unpins the old version of the page. When the last pin on the page is released, the buffer containing the old version can be reallocated. The garbage collector must also hold the latch (semaphore) associated with the page while it copies tuple versions from the old to the new page. The DBMS normally uses this latch during updates to synchronize allocation of space on the page, so holding the latch prevents updates during garbage collection. Until garbage collection has completed, the DBMS does not know how much space is available on the page so no space can be allocated for the new tuple version created by an update.

When archiving, the vacuum cleaner chooses a time value ARCH-DELAY seconds before the current time and declares that to be the **archive start time**. The archiver selects all tuple versions committed before the start time and copies them to the archive or destroys them. To ensure that it copies the correct tuples, the archiver uses the POSTGRES historical data (or **time query**) facility to look up archivable tuples (described in Section 4.2.4). Since the time query only returns data that was valid at the archive start time, uncommitted updates are never copied to the archive. The current POSTGRES implementation stages archived tuple versions to a magnetic disk write buffer before writing them to the archive, since access to the archive media (tape or write-once optical disk) is typically an order of magnitude slower than access to disk.

After the data is archived, the archiver deletes historical tuple versions from the magnetic disk relation. It will usually also have to construct a new tuple difference record representing the oldest available tuple version. Because consecutive tuple versions share many attribute values, the oldest available tuple version probably incorporates attribute values from difference records that the archiver has deleted. The new tuple difference record retains these shared attributes in the non-archived version of the relation. Details of the archive and its cache are described in [57]. The archive indexing strategies are addressed in [44]. Unlike the garbage collector, the archiver must use two-phase locking to guarantee that no transactions are using historical data when it is moved to the archive.

Constructing Overflow Pages to Support the No-Overwrite Policy

In the no-overwrite storage system, the policy for managing **page overflow** has a significant impact on DBMS performance. Because of the no-overwrite policy, repeated updates to tuples on a data page eventually fill the page. The space reclamation strategies

described above will not always prevent pages from filling, especially in a high-update-rate environment. Since high-performance commercial database management systems can run at rates of hundreds of transactions per second, pages fill up too rapidly. Minimum-sized tuples in the original storage system are about 64 bytes, even when differencing is used. Thus, approximately 127 updates fill an 8K page containing one tuple.

The original storage system simply extends the tuple difference chain to a new page when a transaction tries to update tuples on a full page. In the high-update-rate environment, this strategy causes performance to degrade rapidly, especially when the DBMS uses forward difference chains. Whenever a tuple is accessed or updated, each page in the multi-page forward difference chain has to be accessed. "Hot" tuples that receive frequent updates will form the longest multi-page tuple chains. Therefore, the tuples that are used the most often will have the greatest access cost.

Backward difference chains improve the access to multi-page tuple chains in some important cases, but at greater storage overhead. Only the pages containing current attribute values need to be examined if backward difference anchor point arrays are allowed to point across page boundaries. In the case in which the same attribute field is updated repeatedly, only two pages are accessed: the one containing the anchor point and the one containing the most recent difference record. However, the anchor point array entries must be larger if they can point across page boundaries. Only two bytes per pointer are required if the chains are contained within a page, while six bytes (a four-byte page number and a two-byte offset) are required to point to a difference record on another page. Also, updates require both the anchor point and difference record page to be updated. In forward differencing, only the page containing the difference record is modified during an update.

The modified POSTGRES storage system uses an alternative strategy to limit the performance impact of multi-page tuple difference chains, a strategy based on page reorganization. If a transaction updates a tuple on a full page, the DBMS creates an **overflow page** and moves some of the tuple difference records from the original page to the overflow page using a technique detailed below. Managing overflow pages recoverably is more complex than the original POSTGRES storage system strategy, but in the common case it allows access to the current database to take place without examining more than one page per tuple.

There are two possible strategies for creating overflow pages. The simplest strategy would be to construct a new anchor point for each tuple from the original page on the new overflow page. If all of the current version's attribute fields are assembled on the overflow page, the current tuple version can always be constructed from a single page. The header of the new page would have a pointer back to the original page in order to allow access to historical data from the current version of the data. Unfortunately, the DBMS may have many indices referring to records on the original page. Each index would have to be updated in order for indexed access to the data to remain fast. If there are many records on the page

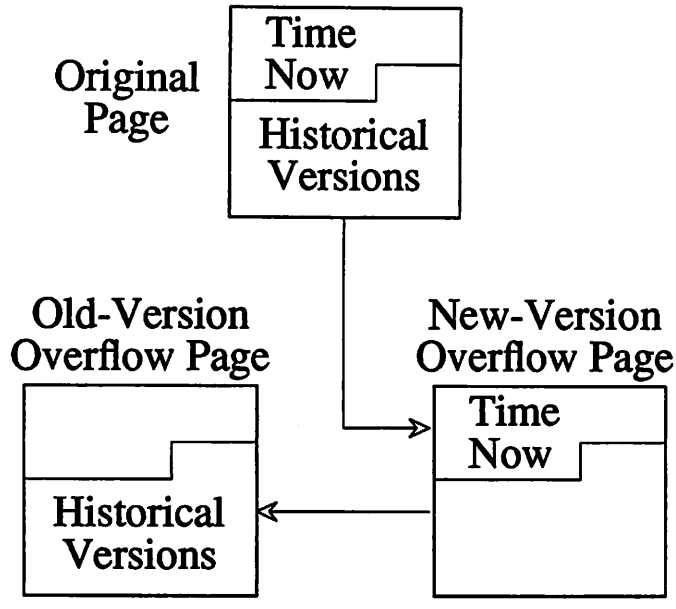

Figure 4.3: Creating an Overflow Page. When the original page overflows, it is split into two pages. The old page contains historical tuple versions and the new page contains the current versions. While the old page is written asynchronously to stable storage, the new page is mapped to a temporary location on disk. Once the old page has been written to stable storage successfully, the new page is allowed to overwrite the original page in the database and the temporary location can be reused.

and many indexes on the relation, creating an overflow page would require updates to many other pages and again have significant performance impact.

A better strategy is to move the *older* versions of the tuples on the original page to the overflow page, as shown in Figure 4.3. That way, index entries still point to the same page and that page still contains the most recent version of the tuple. Overflow pages are chained together so that any historical version of the data can be reached by a multi-page scan.

Creating an overflow page containing historical tuple versions in a way that prevents information from being lost in a crash is tricky. To create an overflow page, the DBMS creates two new pages: the new-version overflow page and the old-version overflow page. The new-version overflow page contains the most recent version of each tuple on the original page. The old-version contains the historical versions of tuples on the original page. Once the old-version page has been saved in stable storage, the new-version page can be used in place of the original. Until the old-version page has been successfully written to stable

storage, the original page contains the only stable version of the historical tuples on the page. If the new-version page were allowed to replace the original page before the old-version page was stable, a crash could destroy the historical tuple versions. The DBMS logically replaces the original page with the new-version page by modifying the buffer pool meta-data. Buffer pool cache meta-data tells which buffer in main memory is associated with a given page of the database. If a page is ever written to disk, the buffer pool meta-data tells where it should be written.

Creating overflow pages is not very expensive if a small amount of non-volatile RAM is available, but, if disk is used for stable storage, overflow causes an extra disk write. In the original storage system, overflow causes the new page to be written to stable storage (to commit the new tuple version) and the original page to be written to stable storage (to link the previous version to the new one). When non-volatile RAM is available, the modified version of the POSTGRES storage system creates its new-version overflow page and old-version overflow page, then blocks while the old-version overflow page is copied to stable storage. After the old-version page has been copied, the DBMS replaces the original page with the new page. When the transaction causing the overflow commits, the new-version page is written to stable storage. As in the original POSTGRES storage system, two pages on stable storage are updated.

When disk is used for stable storage, the new scheme cannot block the DBMS while the old-version overflow page is written to stable storage. Disk latency is too long for such a strategy to be efficient. Instead, the DBMS writes the old-version page to disk asynchronously. As above, the tuples on the original page must remain intact until the old-version overflow page is written to disk. To allow the DBMS to commit transactions before the write of the old-version page has been confirmed, the new-version page is mapped to a temporary location on disk. The temporary page is chained to the original page and the old-version is chained to the new-version as is shown in Figure 4.3. On a commit, the original page and the new-version page must both be written to disk. The original page must be written in order to preserve its pointer to the temporary location of the new-version page. Thus, three pages are written to disk on an overflow instead of two.

In summary, the no-overwrite storage system must have some policy for creating overflow pages. The original storage system's policy of allowing tuple difference chains to span several pages forces the DBMS to examine more than one page during the update of a single tuple. Even if the vacuum cleaner runs hourly, these chains of pages could run to tens of pages for highly-updated tuples in a high performance DBMS. The modified storage system puts historical data on a new page instead of the newly-created data, so that access to the current database remains fast even if the vacuum cleaner runs infrequently. This strategy will result in an extra disk write per overflow, however, if no non-volatile memory is available for stable storage.

4.2.3 Recovering the Database After Failures

The DBMS recovery system must mask any inconsistencies in the database resulting from a DBMS failure. In POSTGRES, these inconsistencies take the form of tuple versions that were created by transactions that later aborted. In a conventional system, data pages can contain two kinds of inconsistencies. First, tuples may have been updated in place by transactions that were aborted. Second, tuples updated by committed transactions may not have been written to stable storage before the failure. Both kinds of storage system require some recovery actions to ensure that transactions starting after system restart never use this inconsistent data.

After a failure, a conventional log-based DBMS makes the entire database consistent before allowing users to access the data. The log in a conventional DBMS contains a sequence of records representing updates to the database and records telling which transactions have committed. At recovery time, the DBMS reads the log to find out which transactions have committed, then examines the data pages affected by each log record to make sure that committed updates have been applied and that aborted ones have not. Recovery, in conventional systems, is usually I/O bound due to the many data pages that have to be read. The cost will be proportional to the length of the log.

The subsection that follows describes the techniques used to detect and ignore invalid tuple versions in POSTGRES. Because the DBMS can detect invalid tuples on use, it does not have to remove inconsistencies in the database at system restart time. We discuss the cost of POSTGRES database recovery after describing the technique for detecting invalid tuple versions.

Transaction Status File

When a POSTGRES transaction begins, a slot is reserved for the transaction in the **transaction status file** maintained by the DBMS. A transaction identifier, or **XID**, is a pointer to this transaction status file slot. The status file records the current state of both current and past POSTGRES transactions. In the original storage system, the transaction can be in one of three states — committed, aborted, and in-progress — while the modified storage system only requires committed and aborted states. The in-progress state was used for synchronization between the POSTGRES vacuum cleaner described in Section 4.2.2 and current transactions. The modified storage system uses more conventional synchronization techniques, so it can use one-bit rather than two-bit slots to encode each state.

Queries of historical data require the DBMS to maintain a second file called **commit time file** in the original POSTGRES storage system. When a transaction commits, it stores the current time in the commit time file. Time queries use this commit time to determine when data written by the transaction became valid. Note that the commit time must be

written before the transaction is committed so that each committed transaction has a valid commit time. The commit time file is decomposed into slots in the same way as the transaction status file, although each slot is four bytes wide instead of one bit wide. This allows the DBMS to use the same XID to look up a transaction's commit time and current state.

POSTGRES backend processes actually reserve blocks of XIDs instead of allocating them individually. The DBMS maintains on stable storage the next available XID, **next-XID**, which indicates the first XID that can be allocated to a transaction after a system failure. When a POSTGRES backend runs out of XIDs, it updates next-XID on stable storage to reserve the next available block. As transactions are initiated by clients, the backend process assigns XIDs from its block consecutively. Because the block is owned by a single process, the backends do not need to coordinate the allocation of a new XID; they only need to ensure that one of them at a time is allocating new XID blocks. Larger XID blocks lessen the overhead of XID allocation, but increase the number of unallocated XIDs that will have to be discarded during a failure.

Identifying the Updating Transaction

At run time, the POSTGRES storage system must detect and ignore updates to the database made by transactions that were later aborted. The storage system stores XIDs in tuple difference records to identify the transaction that created, updated, or deleted a given tuple version. By mapping the XIDs to slots in the transaction status file, the DBMS can determine whether or not these transactions have committed. Because tuples are locked using a conventional two-phase locking scheme [25], a transaction will block if it encounters tuples created or written by other in-progress transactions. Therefore, any uncommitted tuple updates that the transaction encounters are invalid.

The modified POSTGRES storage system stores an XID in the anchor point of the tuple and in each tuple difference record. The anchor point of a tuple stores the tuple's **inserterXid**, the XID of the transaction that inserted the tuple into the database. The XID in each tuple difference record identifies the transaction that updated or deleted the tuple version represented by the difference record. If no transaction has attempted to update or delete a tuple version, the XID field in the difference record contains an invalid transaction identifier.

In the original storage system, additional XIDs were stored, but these turn out not to be necessary. Each difference record kept its own minXid and maxXid. The minXid identified the transaction that created the tuple version and the maxXid told which transaction updated or deleted the tuple version. Clearly, the maxXid of one difference record is the same as the minXid of the following one, so these two fields could be merged into a single XID field in

the modified storage system.

The modified storage system also maintains in each anchor point a field called the **commandID** indicating the DBMS command, or query language statement, that last modified the tuple. Each query language statement is a separate command. When a command changes a tuple, the change is not visible until the next command. So, for example, a record inserted into a relation during a query will not be visible in the database until after the query that inserted it completes. Each POSTGRES process maintains a command counter for its currently executing transaction. The DBMS stores the current command counter value in the tuple's commandID field when the tuple is created and modifies the commandID field every time the tuple is updated. A transaction ignores a given tuple version if that transaction was the one to modify the tuple and the current command matches the tuple's commandID.

Instead of associating a single commandID with the entire tuple, the original storage system associated a maxCommand and a minCommand with each tuple version in the same way as maxXid and minXid. The modified storage system uses a single commandID field for the tuple because the command is only ever relevant to the last tuple in the difference chain. The command field is only used when a transaction has updated a tuple already and is examining the tuple again. Since the current command cannot see its own updates, it cannot have created more than one element in the tuple version chain. Therefore, the only tuple version that could possibly have been created by the current command is the most recent tuple version. The maxCommand and minCommand become a single field because the tuple cannot be created and deleted in the same command.

Detecting Invalid Tuple Versions

A conventional database management system uses write ahead log processing to remove an aborted transaction's updates after recovery. POSTGRES detects and ignores these invalid updates whenever the updated tuple is used after the failure. Because the DBMS maintains previous versions of every tuple updated (using the difference record chain described above), ignoring an invalid update simply means using whichever of the previous versions was valid at the time that the aborted transaction began its update.

An invalid update can be:

(A) An invalid insert. A transaction creates a new tuple, inserts it into the database, then aborts due to a failure. If a later transaction examines this tuple, it must ignore all tuple difference records associated with the transaction.

(B) An invalid delete. A transaction could delete an existing tuple from the database and abort. If a later transaction examines the tuple, it must ignore the delete.

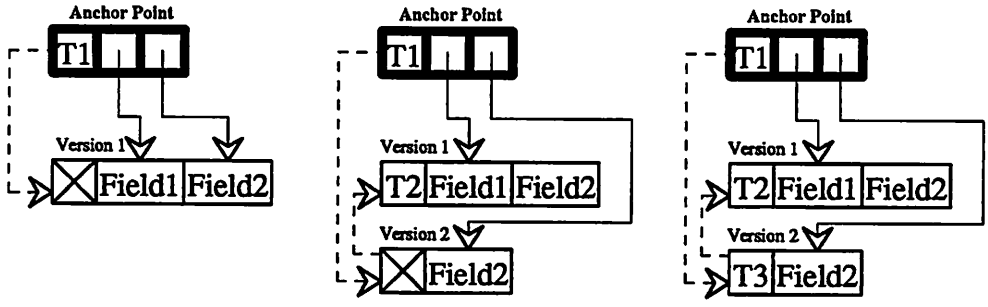

Figure 4.4: Tuple Qualification. This figure shows the same tuple at three instances in time. On the left, the tuple is inserted by transaction T1. T1 writes its XID in the inserterXid slot in the anchor point. In the center, Field2 is replaced by transaction T2. T2 writes its XID in the XID slot of the first version of the tuple. Finally, on the right, transaction T3 deletes the entire tuple.

(C) **An invalid replace.** A transaction could replace a field in an existing tuple, creating a new tuple version. If the transaction aborts, later transactions must use a previous tuple version in the difference record chain, the one that was valid when the aborted transaction made its update.

An invalid update may create more than one tuple version. For example, a transaction may insert a tuple into the database and then update it, creating a tuple with two difference records in it. If the transaction then aborts, both difference records are invalid. Only versions at the end of the tuple difference chain can be invalid since two-phase locking prevents one transaction from updating another's uncommitted tuple versions.

To find invalid updates, we must check for each of the three cases in the list above. Each check requires us to consider an XID associated with the tuple. Note that the checks described below require only one of these three XIDs to be looked up in the transaction status file. Checking for invalid tuples is a common operation and examining the transaction status file is relatively expensive, so reducing the number of lookups to one per tuple gives a performance advantage.

We only check for case (A), invalid inserts, if there is exactly one tuple difference record in the chain. To check for case (A), we examine the inserterXid associated with the tuple header. If that XID is associated with an aborted transaction, the tuple is invalid. If there are several tuple difference records in the chain, we treat it like case (C), even if the same transaction has initiated all of the updates.

To check for case (B), we examine the XID associated with the last difference record in

the tuple's difference record chain. If that XID is NULL, no transaction has attempted to delete the tuple. If the XID is valid and maps to an ABORTED transaction status file bit, then a transaction attempted to delete the tuple and aborted. In this case, we fall back to the last tuple version created by a transaction other than the aborted one.

We only check for case (C) if no transaction has attempted to delete the tuple. To see how to detect an invalid replace operation one must remember how a replace operation is implemented in the POSTGRES storage system. To replace a field in an existing tuple, a transaction creates a new tuple difference record containing the replaced fields and a NULL XID field. The transaction stores its own XID in the XID field of the current difference record in the tuple (effectively "deleting" this difference record) and links the new version to the front of the difference record chain. Thus, a later transaction checks if the replace transaction has committed by examining the XID of the second difference record in the chain. This XID is mapped to the COMMITTED/ABORTED state of the replace transaction using the transaction status file. If the transaction has committed, the last difference record describes the current tuple version. If this transaction has aborted, the last difference record created by an earlier transaction is the valid one.

If the last transaction to update or delete the tuple has aborted, we have to search through the tuple difference chain to locate the last valid version. The DBMS searches through the chain until it finds a difference record with an XID field different from the XID of the aborted transaction. If none is found and if the inserterXid is also equal to the invalid XID, the entire difference chain was inserted by a single aborted transaction and is invalid. If a new XID value is found, the difference record *following* the one containing that XID is the last valid version. Obviously, if the inserterXid is the first XID not equal to the aborted transaction's XID, the initial tuple version is the valid one.

Recovery Costs in the POSTGRES Storage System

Three factors contribute to the costs of recovery in POSTGRES. First, the system must be reinitialized after a failure. While no log processing is required, the DBMS must do some work to initialize the storage system. Second, the DBMS must check for invalid tuple versions on use. Third, overflow pages occasionally result in an extra I/O to find the current version of a tuple. These costs are addressed one at a time in the paragraphs that follow.

At restart, the modified POSTGRES storage system simply allocates a new XID block for each backend process and reinitializes its in-memory data structures. New XID blocks must be allocated after a failure because the DBMS cannot tell which XIDs from the old blocks had been allocated at the time of the failure. Because of efficiency concerns, transactions do not stably record the fact that an individual XID has been allocated (only XID blocks). The original storage system also needed to scan the tail of the transaction

status file, converting the state of each in-progress transaction to aborted in order to show that the transactions in-progress at the time of the failure have aborted.

Although tuple validation is required for every tuple examined by a transaction, validation is not very expensive. Profiles of the debit/credit benchmark used in Chapter Three showed that validation consumed about 1.5% of the DBMS' CPU time; 1.3% came from mapping the XID to a transaction status file slot.

The profile does not include the cost of reading transaction status file blocks from the disk. If the transaction status file is too large to store in memory, additional disk reads will be required to validate tuples. Notice from the previous subsection that at most one XID per tuple is ever looked up in the transaction status file, but one disk read per tuple scanned would still make the storage system prohibitively expensive.

Fortunately, the vacuum cleaner can be used to compact the transaction status file, keeping the file small enough to be cached in main memory. To implement compaction, the vacuum cleaner must record the XID of the oldest in-progress transaction at the time the vacuum cleaner begins its sweep of the database. After the sweep is over, the database contains no invalid tuple versions with updater XIDs smaller than this oldest in-progress transaction's XID. Now, if this oldest-unresolved XID is recorded, it can be used to validate tuple versions. The transaction status file need not be consulted for XIDs smaller than the oldest-unresolved XID; these transactions have definitely committed. The status file could even be truncated at the oldest-unresolved XID in order to save disk space.

Because transaction status can be represented with a single bit, relatively small amounts of memory are required for the status file cache. A DBMS that executes 128 transactions per second consumes only 512 KBytes of status file in nine hours. Thus, even at high transaction rates, the garbage collector can easily ensure that the status file lookups never go to disk by running every few hours. Extremely long running transactions, however, can prevent the status file from being compacted and affect the performance of the entire system.

Finally, when the DBMS fails during the creation of an overflow page, the DBMS must read two pages in order to find the most recent version of the tuples on the page. Figure 4.3 showed how POSTGRES created temporary pages to prevent historical tuple versions from being destroyed. If the temporary page exists, it must be read into memory the first time the page is accessed after a failure, requiring two I/Os to find the tuples on the page instead of one.

POSTGRES requires much less I/O to recover its data than a conventional write-ahead logging system. The conventional system must read each page referred to by a log record during recovery. Many of the data pages read in during recovery will be replaced in memory before the data on them is used by new transactions. Thus, these I/Os would never have happened if the system had not failed. POSTGRES only recovers a page when the data on

the page has been accessed by a current transaction. At that point, the page must be read into memory anyway. In the normal case, the current and previous version of a tuple reside on the same page. Even if the current version of the tuple is invalid, no extra I/O is required to access the data.

4.2.4 Validating Tuples During Historical Queries

When users query historical data, POSTGRES examines transaction commit times to ensure that tuples were valid during the time period of interest. To determine the commit time of a tuple version, the DBMS maps difference record XIDs to commit times using the commit time file. If the current version of the tuple is in the time period of interest, the DBMS must also check that the version was not written by an aborted transaction, using the transaction status file as described above. Status file lookups are necessary because it is possible for an aborted transaction to have a valid commit time. A failure might have occurred between the time that the DBMS updated the commit time file and the time it updated the status file, effectively aborting the transaction. Historical queries must use two-phase locking in order to prevent the archiver from removing tuples from magnetic disk while the query is in progress.

In order to improve the performance of POSTGRES time queries, the original storage system copied the commit time into tuples during garbage collection. Thus, the commit time file did not need to be searched for queries of data older than the last garbage collector run. POSTGRES also maintains a cache of commit times to allow time queries to proceed without constantly accessing the disk to read transaction commit times. However, this cache must be 32 times as large as the status cache since POSTGRES represents commit time using four byte quantities. If not enough memory is available for the cache, then time queries will have to access the commit time file on disk during validation.

4.3 Performance Impact of Force-at-Commit Policy

Commercial database management systems do not use a force-at-commit policy for managing data pages because this policy has poor performance on conventional disk-based stable storage. If several data pages are forced to different locations on the disk, commit is delayed while the disk arm seeks to each location. At commit time, a write-ahead logging storage system only writes log records synchronously; data pages can be written asynchronously when they are ejected from the DBMS disk cache. By placing the log on a separate device from the rest of the database, the conventional DBMS does not have to pay for any disk seeks at commit time.

Modern system architectures and file organizations have a large impact on the performance of POSTGRES' force-at-commit policy. This section compares the I/O performance of POSTGRES to that of a conventional DBMS that uses a write-ahead log. In order to separate the expense of the POSTGRES historical data feature from the expense of fast recovery, we will also consider two versions of POSTGRES: one with and one without the historical data feature. The analysis considers: (a) conventional disk subsystems, (b) non-volatile RAM (NVRAM) stable storage, (c) RAID parallel disk subsystems [59], and (d) Log-Structured File Systems (LFS) [61]. This analysis is based on the analysis in [66] which did not consider RAID, LFS, archiving costs, or the impact of large disk caches. The analysis in this section shows that on a system with a sufficient amount of non-volatile RAM and a log-structured file system, POSTGRES (with history disabled) performs about the same as a conventional system, despite the force-at-commit policy. With history enabled, POSTGRES performs at least thirty percent more I/O than a conventional DBMS.

4.3.1 Benchmark

For the comparison, we use an analytic model based on the TP1 debit/credit benchmark [1]. A transaction in the TP1 benchmark randomly accesses two "hot" relations (Branch and Teller), and one "cold" relation (Account). Each of these is first read then written. Finally, the transaction appends to a History relation, and writes any necessary log records. In the subsection that follows, we describe first the parts of TP1 transaction execution that the conventional system and both versions of POSTGRES execute in the same way. Then, we describe the differences between the three DBMS versions when executing this benchmark.

Assume that there is enough main memory available to cache all of the two hot relations, but not all of the cold one. Thus, in steady state, the DBMS must read one Account page from the disk and write one (different) Account page to the disk on every transaction. History relation tuples contain 50 bytes of data and a tuple header. In POSTGRES, the header is 60 bytes so 74 history tuples can fit on a single 8K page. Therefore, a History block must be written to disk every 74 transactions, on average. A conventional system will maintain less information in its tuple header. If the header is 10 bytes, then a history page is filled every 136 transactions.

In the POSTGRES storage system, the four data blocks updated by TP1 and the transaction status file block must be forced to stable storage after every transaction. The version of POSTGRES in which history is disabled never creates overflow pages. Instead, it garbage collects historical data on a given page whenever the page fills. For the analysis, we will also assume that the version of POSTGRES with history disabled does not record commit times. The commit times are only required by historical queries. Since the history-disabled version of POSTGRES is not preserving historical data, there is no reason for the DBMS

to maintain commit times.

In a conventional file system, a TP1 transaction constructs log records containing the before- and after- image of the updated tuples. At commit time, these log records are forced to stable storage in a single write. We assume that the log records required to describe 20 TP1 transactions fill a log page. This corresponds to about 400 bytes of log record per transaction.

Conventional systems typically do not keep the write-ahead log on the same disk as the database in order to avoid disk seeks at commit time. Since the DBMS always appends to the log, storing it on a separate device from the database means that the disk head is always near the tail of the log and log writes are sequential I/Os. To make the comparison fair, POSTGRES is also allowed one disk to use for sequential writes. Unfortunately, POSTGRES does not have a data structure like the log with a strictly sequential access pattern. If transactions commit in roughly the same order that they are initiated, however, the transaction status file and transaction commit time file will be accessed nearly sequentially. For the analysis, we assume that the version of POSTGRES that has disabled historical queries stores the transaction status file on a separate device. The version of POSTGRES with history support will store the commit time file on the separate device.

To simplify the presentation, we will call one sequential I/O two sevenths, 0.29, of a random I/O. The number is taken from a Fujitsu Eagle drive that has an average seek time of 30ms, average rotational latency of 8ms, and transfer speed of 4ms per 8K page. Thus, the average sequential I/O takes 12ms and the average random one takes 42ms.

Historical Data and Archiving Costs

The version of POSTGRES that preserves historical data pays additional costs to maintain this data. The system must create overflow pages as described in Section 4.2.2 to store the historical tuple versions until they are archived. The system must maintain a commit time file so users can query the historical data as described in Section 4.2.4. Finally, the historical data must eventually be copied to an archive device in order to leave room on the disk for data that is generated by new transactions.

To record the historical data, overflow pages must be created every time a page fills. The rate at which overflow pages are generated depends on how much free space is reserved on each page for updates. If each database page contains a single tuple, 127 updates to a TP1 Account, Branch, or Teller tuple can occur before the page fills. If thirty percent of the page is reserved for historical data, 51 tuples can be stored on each page and a page is filled every 38 updates. For the analysis, we assume that 30 percent of a page is left free when the page is initialized. TP1 replaces three tuples per transaction and a page is filled on average every 38 updates, so the DBMS must write to an overflow page every 3/38, or

0.08, transactions, on average.

In some environments, the cost of migrating data from the magnetic disk onto the archive device could be ignored. In these environments, there is a slow period, perhaps at night, when historical data can be moved from magnetic disk to the archive device. For this analysis, however, we assume that there is no slow period or the slow period comes infrequently enough that storage will need to be reclaimed during operation. This is quite reasonable when a page fills every 38 updates and the DBMS has a sustained, high transaction rate. If the DBMS runs at 128 transactions per second, it creates in an hour about 36,864 overflow pages and historical data consumes 288 MBytes of disk space. Therefore, the analysis assumes that the cost of a transaction must include the cost of archiving the historical data generated by the transaction.

While we must account for archiving costs, the analysis only considers the cost of archiving overflow pages, not the cost of examining and archiving historical tuples on current pages of the database. Overflow pages and the commit time file grow as a function of the transaction rate, so it is relatively easy to determine how much of their costs to account to each transaction. The vacuum cleaner also examines all of the non-overflow pages for historical data. However, this cost depends on the size of the database and the rate at which the vacuum cleaner runs; it is independent of the transaction rate. In order to simplify the analysis, we will assume that the vacuum cleaner runs infrequently enough relative to the transaction rate that archiving costs will be dominated by the commit time file and overflow page cost.

As stated in section 4.2.2, archived data is not written directly to the archive device in POSTGRES. Instead, the pages are accumulated in a write buffer on magnetic disk. When the buffer fills, it is reread from disk and the data is finally written to the archive. Thus, to preserve the historical tuple versions on a single overflow page, the DBMS must:

(a) Create the overflow page and write it to the disk in the current database,

(b) During vacuum cleaning, read the page from disk to find the archivable data on the page.

(c) The vacuum cleaner writes the page to the archive write buffer.

(d) The vacuum cleaner deletes the tuples from the overflow page in the current database and rewrites it.

(e) The vacuum cleaner rereads the write buffer from the disk and pushes it to the archive.

The operations must be done in this order to prevent the archived data from being lost in a failure.

We showed above that each overflow page generated causes several I/Os when all of the archiving costs are taken into consideration. Each overflow page results in one random read (b), one sequential read (e), and three random writes (a,c,d) in the current implementation of POSTGRES. Thus, the total historical data cost is 3*0.08 or 0.24 random writes per transaction and 0.08*1.29 or 0.1 random reads per transaction, plus one update to the commit time file per transaction.

This section ignores some additional costs related to the archive device manager described in Section 4.2.2. We assume that the archive device itself is not a bottleneck. Currently, the optical disk archive used by POSTGRES runs at 1/40 of the speed of a magnetic disk. POSTGRES does enough disk I/O that the archive device is not a bottleneck at present. Also, the 1987 POSTGRES storage system design assumes that new indices are constructed for the data once it is moved to the archive. The cost of creating and maintaining these indices is ignored in the analysis.

4.3.2 Conventional Disk Subsystem

We see the following costs in a conventional disk subsystem:

The conventional DBMS and both versions of POSTGRES each do one random read to get the page containing the transaction's account record.

The conventional DBMS writes one account page to disk to make room in its cache for the new account page. Every 136 transactions, it fills a history relation block that must eventually be written to disk. The cost of these History relation updates is 1/136, rounded to 0.007.

Each version of POSTGRES writes the four pages that were updated by the transaction: account, teller, branch, and history. The force-at-commit policy requires these pages to be written to stable storage at transaction commit.

The conventional DBMS writes the page containing its log records to disk sequentially at a cost of 0.29 random I/Os.

The history-disabled version of POSTGRES writes the transaction status file sequentially at a cost of 0.29 random I/Os.

The history-enabled version of POSTGRES writes the transaction status file and the transaction commit time file. Together, these cost 1.29 random I/Os since one of these files will be written sequentially. As shown in the previous section, it also does 0.1 random reads and 0.24 random writes per transaction on average for overflow pages.

Conventional Disk System	Read	Write	Total
POSTGRES (history-enabled)	1 + .1	4 + .24 + 1.29	6.63
POSTGRES (history-disabled)	1	4 + .29	5.29
Write-Ahead Log	1	1 + .007 + .29	2.30

Table 4.1: Summary of I/O Traffic in a Conventional Disk Subsystem. POSTGRES was not designed to be run without non-volatile RAM to use as stable storage. The conventional system is able to make much more effective use of the cache because of its write-ahead log.

These I/Os are summarized in Table 4.1.

The analysis shows that, in a conventional disk storage system, the POSTGRES no-overwrite policy is much more expensive than write-ahead logging, whether historical data is retained or not. There are two important reasons why the conventional system outperforms POSTGRES in this environment. First, the conventional system can take better advantage of caching than POSTGRES to mask disk writes to the branch, teller, and history relations. The conventional system uses the log to make updates to these relations recoverable so dirty blocks from these three relations do not need to be written to disk so frequently. Second, the history-enabled version of POSTGRES records additional information that conventional systems do not: commit times and overflow pages. This result is different from the one in [66] because the benchmark used in that analysis never rereferenced pages once they were written. Hence, the conventional system could not use the disk cache to absorb writes.

4.3.3 Group Commit

Most high performance DBMSs use a mechanism called **group commit** to reduce the cost of transaction commit. In group commit, the DBMS batches several transactions and commits them at the same time. Group commit improves performance of a conventional DBMS because the log records from all transactions in the group can be written to disk together in a single I/O operation. Instead of having one log write per transaction, there is 1/G where G is the commit group size. Group commit does not decrease the number of random I/Os done by the conventional system on a benchmark like TP1, because the transactions usually update account records on different pages.

POSTGRES receives some benefit from group commit also. Many transactions can share the same write to the status file and commit time file. All of the transactions in the group will usually append to the same History relation page, as well. Some of the updates

Group Commit, Group Size 20	Read	Write	Total
POSTGRES (history-enabled)	1.1	2.4 +0.014 + .24 + 0.05	3.80
POSTGRES (history-disabled)	1	2.4 + 0.014	3.41
Write-Ahead Log	1	1 + 0.007 + 0.014	2.02

Table 4.2: Group Commit in a Conventional Disk Subsystem. POSTGRES benefits more than the conventional system from group commit, since some of the many random I/Os are eliminated. The four POSTGRES force-at-commit I/Os for the TP1 relations become 2.4 I/Os because some pages in the relations are rereferenced by consecutive transactions in the group. The table shows the I/O traffic when the group size is 20.

to branch and teller will fall on the same pages. In a POSTGRES TP1 database with 1,000 branches and 10,000 tellers, the Branch relation has 17 pages and the Teller relation has 169. This figure considers the overhead of POSTGRES page headers, tuple headers, and assumes an average of 20 percent of each non-overflow page contains free space or historical data. Assuming that each TP1 transaction chooses a record to update at random, the expected number of pages can be calculated for any group size. At group size 20, about 5% of the Teller page writes fall onto dirty pages as do 60% of the Branch writes. Thus, transactions, on average, write .95 and .40 percent of a Teller or Branch page, respectively, for a total of 1.35 random I/Os. At group size 20, POSTGRES will write the History relation once per group or 0.05 times per transaction. The total number of random I/Os for the four relations is 2.4. The transaction status file and log are written sequentially once per group for a cost of 0.05 * 2/7 or 0.014. The history-enabled version of POSTGRES writes the commit time file once per group at a cost of 0.05 random I/Os per transaction.

4.3.4 Non-Volatile RAM

The original POSTGRES storage system was designed to use non-volatile RAM to reduce the number of random I/Os required at commit time. POSTGRES would use NVRAM, presumably in combination with guarding, as stable storage so data could be stored recoverably without writes to disk. NVRAM changes the costs of the three systems to the following:

Again, each DBMS does one random read to get the page containing the transaction's account record. The conventional DBMS must write a dirty account page to disk

Non-Volatile RAM	Read	Write	Total (P=1)
POSTGRES (history-enabled)	1.1	1.014 + 2*(1-P) + 0.24	2.35
POSTGRES (history-disabled)	1	1.014 + 2*(1-P)	2.01
Write-Ahead Log	1	1.007 + 0.014	2.02

Table 4.3: Summary of I/O traffic When NVRAM is Available. The number of random I/Os required by POSTGRES depends on the amount of NVRAM available. If all of the branch and teller relations can be cached, POSTGRES with the history feature enabled is about seventeen percent slower than the other two systems. POSTGRES with history disabled is slightly faster than the conventional system in this environment because it does not have to write log pages.

every transaction in order to make room in the cache for the new page. POSTGRES will have to do the same, although it is making room in NVRAM for the account record to be written at commit time.

POSTGRES will be able to store the tail of the History relation in NVRAM. As before, the conventional system fills a history relation block every 136 transactions at a cost of 0.007 random I/Os per transaction. In POSTGRES, History relation blocks are filled every 74 transactions at a cost of 0.014 random I/Os per transaction because POSTGRES has larger tuple headers. POSTGRES will use NVRAM to mask writes to the history relation until a page has filled.

When enough NVRAM can be made available, POSTGRES can buffer TP1's two hot relations in NVRAM also. The branch and teller relations together take about 1.5 MBytes, in POSTGRES. Let P be the fraction of the two hot relations that can be stored in NVRAM.

When NVRAM is available, the conventional DBMS only writes log records to disk when a log page has filled. We assumed that this would take 20 transactions, so the logging cost is 0.05 sequential or 0.014 random I/Os per transaction.

If NVRAM were available, POSTGRES would certainly keep the tails of the status file and the commit time file there. Every 64K transactions, a status file block fills and must be written to disk. In the historical-query version of POSTGRES, a commit

time file block fills every 2K transactions. These numbers are small enough that we will omit them from the analysis.

The history-enabled version of POSTGRES still must write overflow pages to disk every 38 transactions.

Table 4.3 summarizes disk activity required for each storage system when NVRAM is available for stable storage. The POSTGRES costs are parameterized by P, the fraction of the hot relations that can be buffered in NVRAM. In POSTGRES, a TP1 database with 1,000 branches and 10,000 tellers could be buffered in about 1.5 MBytes of NVRAM. This figure considers the overhead of POSTGRES page headers, tuple headers, and assumes an average of 20 percent of each non-overflow page contains free space or historical data.

POSTGRES and the conventional system have comparable speeds if enough NVRAM is available for POSTGRES to cache the hot relations. The conventional system cannot take much advantage of NVRAM; the only improvement it sees due to NVRAM is fewer log writes. POSTGRES can use NVRAM to absorb disk writes in the same way the conventional system used the volatile RAM cache. The NVRAM also masks the cost of maintaining a commit time file for the history-enabled version of POSTGRES.

4.3.5 RAID Disk Subsystems

Next, we consider the cost of running POSTGRES on a RAID disk subsystems[59]. RAIDs are parallel disk subsystems that use parity to provide media recovery at lower costs than standard techniques such as disk mirroring. A RAID is divided into **stripes** of N-1 data blocks and one parity block, each on a different disk. If one disk fails, each block on the failed disk can be reconstructed using the parity block and the N-2 other data blocks from its stripe. Unfortunately, maintaining parity blocks worsens random write performance significantly. When a data block is randomly written, the I/O subsystem must (a) read the parity block, (b) reread the data block from disk so its original value can be determined (c) compute a new parity block from the old parity block, old data block, and new data block, and (d) write the parity block out again. Thus, each random write causes two additional random reads and an additional random write. The additional reads can be eliminated for random I/Os if the I/O subsystem has enough physical memory available for caching parity blocks and the original values of the data blocks. Since the DBMS is already delaying writes as long as possible, such caching is unlikely to be very effective, especially when NVRAM is available.

Because of parity blocks, RAID quadruples the number of random I/Os required by every transaction in either storage system. Therefore, RAID increases the amount of I/O that takes place when insufficient NVRAM is available to buffer the hot relations. The

RAID + NVRAM	P=1.0	P=0.875	P=0.5
POSTGRES (history-enabled)	6.12	7.12	10.12
POSTGRES (history-disabled)	5.06	6.06	9.06
Write-Ahead Log	5.04	5.04	5.04
Conventional disk + NVRAM	**P=1.0**	**P=0.875**	**P=0.5**
POSTGRES (history-enabled)	2.35	2.60	3.35
POSTGRES (history-disabled)	2.01	2.26	3.01
Write-Ahead Log	2.02	2.02	2.02

Table 4.4: Comparison of Random I/Os in RAID and a Conventional Disk Subsystem. Reading and writing RAID parity blocks increases the penalty for insufficient NVRAM to buffer random writes in POSTGRES. P is the fraction of the branch and teller relations that can be buffered in NVRAM. The upper part of the table shows the affect of limited NVRAM when the database resides on a RAID. The lower part of the table shows the effect of NVRAM when a conventional disk subsystem is used. The P=1.0 column in the lower table is the same as the right column of Table 4.3.

2*(1-P) random writes from Table 4.3 become 8*(1-P) random I/Os when parity blocks are considered. This becomes one extra write per transaction on average when P is 0.875 and four extra writes per transaction when P is 0.5.

4.3.6 RAID and the Log-Structured File System

Finally, the Log-Structured File System (LFS) described in [61] can be used to eliminate the random writes required by the DBMS and to reduce the cost of maintaining parity on a RAID. LFS organizes the disk as a collection of half-megabyte **segments**. One of these segments is the current segment, or tail of the log. When an updated file block is forced to disk in LFS, the file system appends the block to the current segment rather than seeking to the block's original location on disk and writing it there. The file system meta-data is updated in memory and logged to the current segment also, so future reads can find the newer version of the block. Eventually, LFS garbage collects old segments, throwing away out-of-date blocks. The blocks that are not out-of-date ("live" blocks) are coalesced into a new segment and rewritten.

LFS improves DBMS performance on a RAID because it turns random writes into sequential writes. When enough NVRAM is available to allow the system to buffer a large

LFS/RAID/NVRAM	P=1.0	P=0.875	P=0.5
POSTGRES (history-enabled)	2.12	2.27	2.72
POSTGRES (history-disabled)	1.61	1.86	2.61
Write-Ahead Log	1.62	1.62	1.62
Conventional disk + NVRAM	**P=1.0**	**P=0.875**	**P=0.5**
POSTGRES (history-enabled)	2.62	2.87	3.62
POSTGRES (history-disabled)	2.01	2.26	3.01
Write-Ahead Log	2.02	2.02	2.02

Table 4.5: Comparison of I/Os in LFS RAID and a Non-LFS Conventional Disk Subsystem. LFS sequentializes I/O and eliminates the I/Os associated with calculating parity block changes for the blocks updated by a TP1 transaction. As in the previous tables, P is the fraction of the branch and teller relations that can be buffered in NVRAM. Ten percent of each segment garbage collected by LFS is assumed to be live data. Again, POSTGRES can outperform a conventional DBMS in this environment because it writes fewer log pages and pays little penalty for non-sequential write behavior.

amount of data, the write traffic for many transactions can be batched together into a large sequential write. If the data is written to disk in full stripes, the stripe's parity block can be computed from the N-1 other blocks in the stripe. This eliminates the cost of reading parity blocks and amortizes the cost of writing a parity block over N-1 blocks of user data.

While LFS turns random writes into sequential writes, garbage collection increases the number of blocks that must be read and written by the TP1 transaction. Garbage collection cost depends on how much live data is contained in the garbage collected segment. If F is the fraction of live data on a garbage collected segment, the TP1 transaction must read one block and rewrite F blocks for every block of free space it reclaims. Therefore, each random write from Table 4.3 becomes roughly (2+F) sequential I/Os, or 2/7 * (2+F) random I/Os when LFS is used. Table 4.5 shows the bottom line: when LFS is used and enough NVRAM is available, the POSTGRES storage system is as fast or faster than a conventional storage system. LFS reduces the cost of constructing parity blocks and eliminates the disk seeking that force-at-commit causes in non-LFS file systems.

Not addressed here is the fact that LFS randomizes the layout of pages on disk, so sequential reads during queries are effectively impossible. This problem is discussed in [63] and database reorganization strategies to minimize this effect is a subject of continuing

research. Measurements presented in [63] comparing sequential Account file reads after four hours of TP1 transactions on LFS and a conventional file system show the LFS read to be about 1/3 slower than the conventional file system read.

4.3.7 Summary

In summary, large amounts of NVRAM are crucial to the performance of POSTGRES for update-intensive applications such as TP1. A WAL-based system is able to buffer updated pages in volatile memory and use the write-ahead log to guarantee the durability of updates. POSTGRES can only buffer updated pages in NVRAM. Therefore, the performance of POSTGRES is comparable to that of a WAL-based DBMS if the heavily-updated parts of the DBMS can be cached in NVRAM. When POSTGRES is used with a RAID, the penalty for insufficient NVRAM increases by about four times; four random I/Os are required for every random I/O required on a conventional disk system. Using a log-structured file system changes the way in which RAID parity blocks are calculated, hence eliminates this penalty. Thus, even on a RAID I/O device, POSTGRES performs well when enough non-volatile RAM is available. This section also indicates that, while a DBMS can use the fast recovery features of POSTGRES without losing performance, the historical data feature reduces performance by about seventeen percent in a high-update-rate environment.

The analysis in this section also drives home the importance of techniques like page guarding to both conventional systems and to POSTGRES. Using NVRAM as stable storage only makes sense if data stored there is safe from errors. Because of the increasing importance of software errors, systems can only assume that data in NVRAM is safe from errors if precautions such as guarding are taken.

Finally, we have assumed in this section that the archive device itself is not a bottleneck. Current POSTGRES measurements [57] show that data can be archived to optical disk at about a fortieth the rate that it can be stored on magnetic disk. Given the current archiver implementation, archive data is not generated quickly enough for the archive to limit performance. However, POSTGRES archives all information that a conventional DBMS would store in its log. Performance of a conventional high performance DBMS is usually limited by log device speeds. Hence, it is conceivable that a redesign of the storage system would make archiving a bottleneck.

4.4 Guarding the Disk Cache

Large main memory disk caches help DBMS performance significantly, but make the outage that occurs after a software failure more noticeable to customers. After a software failure, the disk cache (DBMS buffer pool) is usually discarded because the extent of the

damage caused by the error is unknown. Rather than risk propagating corrupted data into the permanent database, the DBMS reinitializes the disk cache using the clean versions of the cached pages on disk. The recovery cost of demand-paging the database into main memory is:

disk-seek-time * effective-cache-size / page-size.

Ignoring the effect of disk arm contention with currently executing transaction, recovering the disk cache takes about 4 minutes if the disk seek time is 30ms, the effective cache size is 64Mbyte and the page size is 8 KBytes.

Chapter Two, however, showed that the most common types of errors are not the ones most likely to damage data in the buffer pool. Most errors are control errors which do not affect the guarded buffer pool. If the buffer pool is guarded to prevent corruption by addressing errors, the DBMS can reuse the old buffer pool after a failure. Reliability is only affected if errors have propagated to buffer pool pages, but not to pages stored on disk (or in stable memory). This section describes the situations under which additional reliability risk does occur. We must consider four separate cases.

First, an error could corrupt the values that are being inserted into the database. For example, a data error could cause ten dollars to be deducted from a bank account instead of one dollar. If the transaction is allowed to commit, these errors will become unrecoverable whether the buffer pool is guarded or not. Because of transaction durability, all updated tuples become permanent at transaction commit time. In a conventional system, the corrupted values are written to the log; in POSTGRES, the corrupted values are written into data pages and forced to stable storage. Thus, recovering from a guarded buffer pool does not increase reliability risk due to this first class of errors.

Second, an error could corrupt data on the same page as a tuple updated by the DBMS. In POSTGRES, this corrupted page will be written to stable storage at the end of transaction. In a conventional DBMS, the corrupted page will remain in the buffer pool until it is replaced or until the next checkpoint. If the DBMS fails before the page would have been written to stable storage in a conventional system, recovering the buffer pool from disk would clear the damaged page, hence guarding reduces reliability in this case. In POSTGRES, the damaged page is written to disk at transaction commit, so reloading the buffer pool provides no benefit. Presumably, the DBMS *would* reload any pages that were unprotected at the time of the failure. This class of errors argues strongly for the deferred write model of guarding, which is unlikely to affect unmodified records on a page containing modified ones.

Third, an error could corrupt a page that is not updated by any transactions at all. The data from Chapter Two shows that it is unusual for "random" pages in memory to be

damaged by errors. When they do occur, such errors are also the ones that are most likely to be detected by guarding.

The fourth error case to consider is corruption of the buffer map. Buffers are identified by a mapping between <relation ID, blockNumber> and the buffer. Even if the page is not physically corrupted by an error, corrupting the mapping will effectively corrupt the data. By saving <relation ID, blocknumber> pair in the header of each data page, this kind of error can be detected on use.

In summary, recovering without reloading the buffer pool will improve availability at some risk to DBMS reliability. Given the available data on software errors and the lack of available techniques for measuring software reliability, it is hard to quantify the increase in risk. Case two, corrupting data near updated tuples, and case three, random corruption of the buffer pool are the only ways the recoverable cache can decrease reliability. The exact increase in risk depends on how effective guarding is at preventing errors and how long errors remain undetected after they occur. The data in Chapter Two is not conclusive, but it indicates that the risk to guarded data in the buffer pool is small, especially if the deferred write model of guarding is used.

4.5 Recovering Session Context

In order for a DBMS client program to submit queries to the POSTGRES backend (or server) process, it must establish a communication **session** with several kinds of state that can be lost in a failure. Reestablishing sessions between clients and the server is slow for four reasons. First, recovery is client-driven. The clients must detect through timeouts that the DBMS server has failed before any recovery actions can begin. Second, restoring sessions requires messages to be exchanged between client and server processes, hence transmission delays are incorporated into the recovery time. Third, when the server has many clients, all of them try to reconnect at the same time and contend for server resources. Finally, if a client is awaiting confirmation of a transaction commit, it must query the database to determine whether or not the commit occurred before the system crash. If the transaction did not commit, the client must resubmit it. When a transaction is short enough (e.g. debit/credit workload), the entire transaction can be contained in a single message so every client needs to find out if its last transaction committed before submitting a new one.

This section describes techniques developed in the course of this dissertation for reducing the impact of these problems. In the modified version of POSTGRES, recovery is server-driven. It allows sessions to be created and stored so clients do not have to run the reconnect protocol before new queries are submitted to the DBMS after a failure. The session recovery mechanism also integrates the POSTGRES storage system and the communication protocol

in order to determine quickly whether or not a clients' last transaction succeeded. Finally, the recovery mechanism takes advantage of guarded memory to limit the number of clients that need to communicate with the server during recovery from software errors.

4.5.1 Communication Architecture of POSTGRES

The original version of POSTGRES had a backend-per-client software architecture; one backend process was created for each DBMS client requesting service from the DBMS. In the original version of POSTGRES, the DBMS was considered available again when the DBMS server was ready to accept new connections from clients. Little work had been done to help clients determine how to reestablish state lost in the failure.

Partly in order to support fast recovery, the architecture was changed so that all clients connect to and share a *pool* of DBMS backend processes. When a message arrives from one of the clients, it is queued in shared memory. Every time a backend process becomes idle, it chooses a session with pending work and does the work. Once a client's session is assigned to a given backend process, the backend continues working with the client until the end of a transaction. This simplifies the implementation substantially since backend processes are not multi-threaded and POSTGRES has a great deal of per-transaction state. In order to simplify the protocols described below, we assume that a client does not submit more than one transaction in a single message.

To simplify the description of the recovery mechanism that follows, we break communication between client and server into five phases based on the status of the client's outstanding transaction: unsubmitted, submitted, queued, executing, and committed/unconfirmed. These phases are summarized in Figure 4.5. When a client has no outstanding transaction, the communication protocol is in the **unsubmitted** phase. The second phase, **submitted**, takes place while the message initiating the transaction is in transit between the client and the server. The **queued** phase fills the time between the arrival of the message and the assignment of the transaction to a backend process for execution. The **executing** phase is next and may involve additional message traffic between the client and server. After the DBMS commits the transaction, the protocol begins the **committed/unconfirmed** phase which lasts until the client receives confirmation of the commit from the server. Because all executing transactions are aborted anyway after a failure, unconfirmed aborts are effectively the same as executing transactions. When transactions abort, the executing phase simply continues until the abort confirmation arrives at the client. After the commit/abort status of the transaction is confirmed, the unsubmitted phase begins again.

Before sending transactions to the DBMS server, the client application must authenticate itself and initialize a session. The original version of POSTGRES used a communication protocol implemented using operating-system-supplied virtual circuits (TCP/IP [20]). For

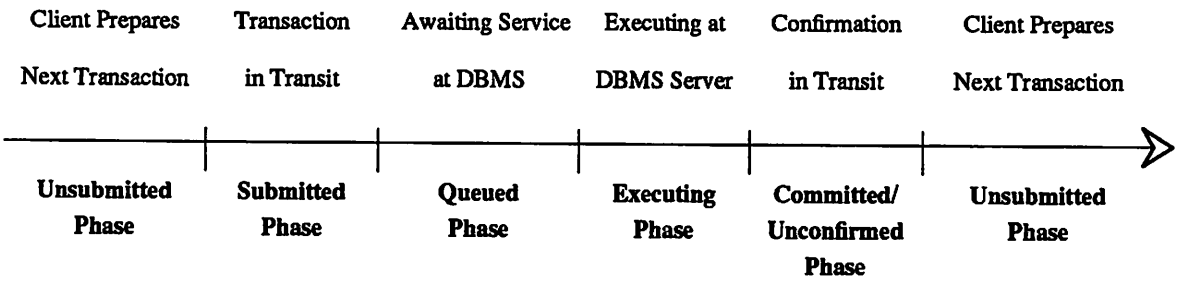

Figure 4.5: Phases of the Client/Server Communication Protocol. The *unsubmitted* phase ends when the client sends a message containing a transaction to the DBMS server. The *submitted* phase ends when the server accepts the messages and queues its contents for service. The *queued* phase ends when a server process is available to execute the transaction. The *executing* phase ends when the server commits the transaction. The *committed/unconfirmed* phase ends when the client receives confirmation of the transaction's commit. The executing phase may contain other client/server communication if the transaction requires more than one message and this phase leads directly to the next unsubmitted phase if the transaction is aborted.

the work described in the current Section, POSTGRES was modified to use a reliable datagram protocol built on top of the unreliable datagrams provided by the operating system (UDP [20]). Reimplementing parts of the network protocol at user level gave POSTGRES control over the system state used in interprocess communication. Because this state is managed by POSTGRES instead of the operating system, it can be saved at session establishment time and restored after a failure. As in TCP/IP connections, reliable datagram sessions are established in a three-message exchange between the client and the server in which sequence numbers are established and the client is authenticated. The dissertation considers only server recovery, hence, the section that follows contains no provisions for saving and restoring state present only at the client.

4.5.2 Recovery Mechanism for POSTGRES Sessions

To reestablish communication with a client after a failure, the server must restore four kinds of session state:

(1) **Authentication information:** When a client has been authenticated, the server generates an authentication token. The client must send the token with every subsequent message to prove it has been authenticated.

(2) **Peer address:** The client and server must each record the other's network address.

(3) **DBMS context:** In addition to the communication-related context, clients have some database-related context that is maintained with the session. For example, the client states the name of the database it is operating on when it establishes a session.

(4) **Sequence numbers:** A sequence number is recorded for the next incoming and outgoing network packet in order to detect lost and duplicated packets.

The first three items are generated at the beginning of the session and not modified again until the session is closed. Sequence numbers change every time a message is sent or received and the server's sequence numbers must agree with the sequence numbers maintained at the client. Saving the sequence numbers of a session that is actively being used is two expensive to be practical, however, an established, but unused, session can be described by a small structure containing the first three kinds of state plus the initial sequence numbers for the session.

In order to have sessions that are ready to use at recovery time, POSTGRES allows clients to create backup sessions and save the server side of the backup session on stable storage. After a failure, the client and server can begin to use the backup sessions immediately without going through the normal session establishment protocol. When a client initially

connects to the server, it establishes several sessions simultaneously, using a single three-way message exchange. Each of these sessions has a unique authentication token, but all share the same peer address and DBMS context. One of the sessions established is designated the **active** session and used for the initial communication between client and server. The other sessions are linked into an ordered list and saved on stable storage. Backup sessions are always activated in the order assigned them when they were created.

After a failure, a backup session can be activated in one of two ways. First, either the client or the server can activate a session simply by sending a message using that session. The client can also ask the server to activate a backup session automatically if the primary session has failed. To request an automatic activation, the client appends a backup session's session ID and authentication token to every request it sends to the server. If the primary session has been lost in a failure, the DBMS acts as if it received the message using the backup session. Eventually, new sessions can be established to replace the ones destroyed during the failure, but the database is available while the backup sessions are being replaced.

The automatic activation mechanism is designed to help avoid additional communication when a client submits a new transaction after a server failure. Without such a mechanism, the server would reject the first message each client sent after a failure and force the client to resend the message using one of the backup sessions. This mechanism just piggybacks the information that would be resent onto the first message, making the message eight bytes longer but avoiding a retransmission after a failure. Note that only the message that initiates a transaction can specify a backup session. Once the transaction begins, the client must do extra work to handle transaction aborts as described below anyway, so the extra message traffic cannot be saved.

4.5.3 Restarting Transactions Lost During Failure

Because all communication between client and server is associated with a transaction, the recovery action required to restore data that was in transit at the time of the failure is fairly straightforward. If a given client session was in the submitted or queued phase, the outstanding transaction must be resubmitted. If the transaction was executing at the time of the failure, it has been aborted. An aborted transaction can be resubmitted unless the transaction is complex enough that higher level abort recovery procedures are required. For this section, we will assume that if an aborted transaction cannot be resubmitted then fast recovery is impossible. If the transaction was in the unsubmitted, then the client simply continues normally. If the client was in the committed/unconfirmed phase, it can continue without resubmitting the transaction as soon as it confirms that the transaction has committed. Thus, to recover the data in-transit at the time of the failure, the client must only determine whether or not to resubmit the last transaction.

To determine which phase the communication protocol was in at the time of the failure, POSTGRES uses the transaction identifiers (XIDs) discussed in Section 4.2. In addition to the four items of session state described above, each POSTGRES session is allocated an XID. The initial XID is sent to the client as part of the session establishment protocol. Every time the server confirms a transaction commit, a new XID is allocated and sent to the client in the confirmation message. The client saves the current XID of the session to be used in recovery if the server ever fails.

After a failure, the server sends a **recover message** to each client, telling the client that a failure has taken place. After receiving a recover message, the client assumes that the last transaction was either lost or aborted and resubmits it using a new session. The client also sends the both the XID and the session ID used by the transaction the first time it was submitted. These two items will be used to determine if the transaction was committed but unconfirmed when the server failed.

After receiving the resubmitted transaction, the DBMS server looks up the XID submitted by the client in the transaction status file. If the status file shows that the transaction has committed, the transaction was committed but unconfirmed at the time of the failure. The server resends a confirmation message in this case and does not reexecute the transaction. If the lookup returns "aborted," the transaction was in one of the other states when the server failed. The DBMS then assigns the transaction a new XID, the one associated with the current session, and executes it. The transaction cannot reuse the old XID since uncommitted tuple versions with that XID may have been created before the failure.

If the server fails again before completing the resubmitted transaction, the client will resubmit the transaction again using the next available backup session. As before, the client must send the XID used when the transaction was originally submitted and the session ID of the initial session over which the transaction was submitted. Since the sessions are ordered, the server will realize that it has received the second resubmission of a transaction (the original session ID and the current session ID will differ by two). This time, the server must check *two* XIDs when it receives the resubmission. Either the original submission of the transaction or the first resubmission may have resulted in transaction commit. The XID for the session used in the first resubmission is determined from the backup session structure stored on stable storage. Again, if either of them committed, a confirmation is sent to the client. If neither did, the transaction is reexecuted using the XID associated with the current session.

If the server fails more than two times without completing a transaction, the same procedure is followed until the client runs out of backup sessions. Each time the server fails, the client resubmits the transaction using a new backup session. Because sessions are ordered and the client sent the session ID of the first session used to submit the transaction, we can find all XIDs that might have been associated with the transaction. The DBMS

checks the XID for each session between the initial one and the current one, sending a confirmation message if one of them is committed. The session structures on stable storage are used to find the XID associated with each of the intermediate sessions. Once the transaction is executed and the client receives a confirmation of the commit, it will send a new transaction (not a recovery message). When the server receives the new transaction, the old sessions can be garbage collected from stable storage.

Using these techniques, the server still must send a recover message to every client at recovery time, and every client that has an outstanding transaction must resubmit that transaction. If guarded memory is available, however, the server can recover with reduced message traffic after software failures. Guarded memory buffers are used to store the messages containing queued transactions. By also maintaining a guarded memory list of clients that have acknowledged their commit confirmation message, the server can avoid sending messages to most clients in the unsubmitted state as well. At recovery time, the server sends recover messages to some clients in the resubmit state and all clients in the executing and committed/unconfirmed state. Only clients in the executing and committed/unconfirmed state ever resubmit transactions. Fewer messages from the server and fewer clients requesting recovery actions will help the system scale to larger numbers of clients.

4.6 Summary

Fast recovery techniques such as those discussed in this chapter are an important component of the fault tolerant system. The error detection mechanisms normally used in fault tolerant systems and the new error detection mechanism presented in Chapter Three halt the system when an error is detected. This makes the system more reliable, prevents it from producing incorrect results, but also makes the system less available to its users. In addition to detecting its errors, the system must minimize the length of time that it takes before beginning to accept new transactions. Also, the mechanisms used to limit downtime must be simple enough that they do not reduce reliability as they increase availability.

Because processing the write-ahead log consumes the bulk of the recovery time in a conventional system, the key fast recovery feature in POSTGRES is the 1987 storage system design, which allows systems to restart without log processing. This chapter builds on the original storage system design by providing enhancements that improve storage system performance on transaction processing workloads. The enhancements include backward differencing of tuple versions, shorter tuple difference chains, a shortened transaction status file, and a faster strategy for system restart. We also provide more details to data page garbage collection than were considered in the original design.

This chapter does a thorough analysis of the impact of the POSTGRES force-at-commit buffer management policy on TP1 performance. The analysis shows that the optimized version of the POSTGRES storage system does the same amount of I/O as a conventional storage system when a sufficient amount of non-volatile RAM is available and the POSTGRES historical data feature is disabled. For TP1, about 1.5 MBytes of NVRAM is required for performance comparable to a WAL DBMS. When a RAID disk subsystem is used, POSTGRES still performs as well as a conventional system as long as the log-structured file system (LFS) is used. When the POSTGRES historical data feature is enabled, the analysis shows that POSTGRES performs about seventeen percent more I/O operations.

Finally, this chapter extends POSTGRES fast recovery support by with mechanisms for recovering the state required for communication between clients and the DBMS server. Saving client/server connections in stable storage allows the client to begin submitting transactions to the server immediately after the server recovers from a failure, without first going through a connection reestablishment protocol. The chapter also discusses the effects of using the guarded memory facility introduced in Chapter Three to reduce the need to reload the disk cache after a failure.

Technology trends are making the fast recovery benefits of POSTGRES more practical in many environments, particularly high end data processing systems. Increasing CPU speeds are reducing the already small performance impact of POSTGRES garbage collection and run-time checks. Hopefully, the performance impact of guarded memory will be reduced in faster processors as well. The costs related to force-at-commit can be controlled if enough NVRAM are made available to the DBMS. NVRAM prices are dropping and are currently about four to six times the cost of volatile RAM [5]. As cost effective, high performance systems become easier to build with new generations of hardware, customers will be more willing to trade limited amounts of transaction performance for high availability.

Chapter 5

Supporting Indices in the POSTGRES Storage System

5.1 Introduction

Both the original version of POSTGRES and the extended one presented in Chapter Four addressed ways that no-overwrite strategies in the management of heap (unkeyed) relations could improve DBMS availability. Chapter Five considers the effects of no-overwrite recovery strategies on DBMS index data structures, an issue omitted from the original POSTGRES storage system. In this chapter as in the previous one, the goal is to support fast DBMS recovery and reduce down time after failures. By recovering without relying on a write-ahead log, the database becomes available immediately after the DBMS is restarted. If the failure causes inconsistencies in the index data structures, these are detected and repaired as they are encountered. From an availability standpoint, this is a better strategy than checking for and repairing all inconsistencies at DBMS restart time.

Most database management systems treat indices and heap relations in different ways because indices have higher concurrency requirements than heap relations and have more complex structure. For example, a high performance DBMS often uses two-phase locking only on the heap relations and short-term locks on B-tree index pages. In two-phase locking, data updated by a transaction remains locked until the transaction commits. Non-two-phase locking improves concurrency in indices because many unrelated index keys are accessed using the same internal pages of the index. If one transaction modifies a shared internal page, two-phase locks would prevent other transactions from using the page until the first transaction committed. Non-two-phase locking complicates recovery, however, because one transaction, A, can insert a key using a shared page modified by another transaction, B. If A commits, it must also commit the shared page in order to commit the inserted key.

If B aborts, it must not undo any modifications to the shared page or it might also remove access to the key inserted by transaction A.

The POSTGRES storage system techniques described in Chapter Four will not provide recovery when these non-two-phase locks are used. The POSTGRES storage system associates a transaction identifier (XID) with any update to the database. When the data is examined, the XID is mapped to a status bit to determine whether or not the transaction has committed. Because XIDs are allocated to transactions, one transaction cannot commit changes that depend on updates made by other transactions.

A second problem for index management in the POSTGRES storage system is that inserting a single key into an index sometimes requires several pages to be updated. For example, in a B-tree index, adding a key to a leaf page can cause the leaf page to **split**, which in turn causes the leaf's parent to be updated. The page split modifies the contents of several pages and changes the inter-page pointers that maintain index structure. Failing after some but not all of the updated pages have been written to stable storage leaves the index structurally inconsistent. In a conventional DBMS which uses a write-ahead log (WAL) protocol for recovery, the atomicity of index updates is guaranteed by log processing at recovery time (e.g. [54]). In these systems, the log records describing structural changes to the index are written to stable storage before the updated index pages. During recovery, the structural changes are redone and the inconsistent pointers are repaired before new transactions are allowed to update the index. Because POSTGRES has no log, it requires other solutions.

In [52], the DBMS maintains consistency of B-tree indices by adding extra synchronous disk writes and by controlling page write order. For example, if a new index page P is created in a page split, P must be forced to stable storage synchronously before any page of the index that contains a pointer to P. POSTGRES index management assumes that synchronous writes to a single file are *unordered* for two reasons. First, using several synchronous writes per page split would significantly worsen page split performance. Controlling write order in a single multi-page synchronous write is not allowed in UNIX-based operating systems and would worsen the performance of disk scheduling algorithms even if it were allowed. A second and more important reason not to depend on write ordering for index management is that it will not work for some common kinds of indices. Section 5.3.6 describes an example from the B^{link}-trees used in POSTGRES. No write order exists that will leave this data structure consistent during the entire page split [1]. In file systems that support efficient transactional updates to files, such as the version of the log-structured file system described

[1] When the chapter refers to "conventional" B-trees, it assumes that write-ahead logging is used for recovery, not ordered writes. Commercial systems sometimes use the ordered write model despite its problems. Customers also sometimes use non-recoverable indices, preferring to rebuild the indices from scratch when the indices are corrupted to suffering the performance penalties of the ordered-write model.

in [63], solutions based on control of write order will perform well and will be simpler than the techniques described in this chapter.

This chapter presents two general techniques for maintaining index consistency without using write-ahead logging. In both techniques, the DBMS detects on first use any inconsistencies in the index caused by interrupted updates. When an inconsistency in the index is discovered, consistency is restored by reexecuting incomplete page split or merge operations. Although we have implemented them only for B^{link}-trees, the same techniques can be used for R-trees [32], extensible hash indices [26], and other B-tree variants such as B*-trees [19].

One of the two techniques uses a no-overwrite strategy which is similar to shadow paging [51]. The before-image of a page to be split is left intact on stable storage until the two half-pages resulting from the split have been written out. Although recovery mechanisms based on shadow paging have been abandoned in commercial systems because of the performance problems experienced by System R [30], they are a practical mechanism for managing indices. Shadow paging makes sequentially-ordered pages in the file non-sequential on the disk. While non-sequential ordering ruins the performance of clustered relation scans, it is not an issue for index files. The shadowing technique, however, requires the index to store pointers to the locations of before-images of its pages. These additional pointers cause the shadow page B-tree to use more disk space than a conventional B-tree.

The second technique, page reorganization, eliminates that space overhead, but performs poorly when the same index page splits many times during the same transaction. The page reorganization scheme ensures that keys moved from one page to another in a split are always available on either the source or destination page. A hybrid between the two algorithms could preserve the best features of each at a cost of greater software complexity. The hybrid would use different algorithms for splitting pages near the root and near the leaf of the B-tree. Using the shadowing technique at the leaf nodes where page splits are most common would maintain high performance during page splits. Using page reorganization near the root would reduce space overhead.

The index management techniques used in POSTGRES can even improve the performance and reliability of a conventional write-ahead log storage system. In these systems, B-tree index implementations record structural changes to the index in the log. The keys involved in page splits and merges must be physically copied into the log in order to guarantee the structural integrity of the index. Using POSTGRES indices would allow the system to log the keys inserted and deleted from the index, but not the keys involved in structural changes. Combining POSTGRES index management with conventional write-ahead logging would have both performance and software fault tolerance benefits.

This remainder of this chapter is divided into five parts. The first one lists some assumptions used throughout the chapter. The second section describes the new index

management techniques. A third section discusses the implications of the techniques for a conventional storage system based on write-ahead logging. The fourth section evaluates the performance impact of these techniques and the fifth section gives conclusions.

5.2 Assumptions

An index allows the DBMS to improve access to tuples in a base relation. Entries in the index are <V, TID> pairs where V is a key value and the TID (tuple identifier) is a pointer (page number, offset) to a tuple in the base relation. The index implementation must support an insert operation that adds entries to the index, a delete operation that removes entries, and a lookup operation that returns the TID associated with a given key. B-trees often allow a GetNext and GetPrev operation which returns the <V, TID> pair following or preceding the last key looked up. In POSTGRES, these operations are implemented as options to the normal lookup operation.

The algorithms described in this chapter require each key managed by the index to be unique. Since indices are sometimes built using attributes that can have duplicate values, the DBMS must convert each user-visible key value V into a pair <V, OID> before it is entered into the index. The OID is the unique object identifier associated with the object referred to by the index entry. Because the OIDs are unique, the keys inserted into the index are unique. This conversion adds four bytes to the size of every key. Note that the Lehman-Yao concurrency control algorithms used in most B-tree implementations make the same assumption. Therefore, these four bytes of overhead are not an overhead we associate with the shadow or page reorganization B-trees in the analysis of this chapter.

In POSTGRES, all pages that are modified by a transaction must be written to stable storage before the transaction commits. For the purposes of this paper, when the DBMS syncs its pages, all modified pages are written to disk. They are written to disk in an order chosen by the operating system, not the DBMS. When a crash occurs during a sync operation, any subset of the synced pages may have been written to disk. We assume that single-page disk writes are atomic. The sync system call is assumed either to block the DBMS or to notify the DBMS when all the page writes have been completed. The sync operation corresponds to the limited control over page write order that the UNIX operating system gives its users. UNIX allows groups of pages to be written to disk together, but does not allow the application to control the write order of the pages within a group. Also, it is possible for one transaction to be updating data in a page at the time that another transaction is syncing the page.

To make the index recoverable without log processing, the DBMS must ensure that currently valid keys are visible and invalid keys are invisible to index lookup operations.

The POSTGRES storage system can detect and ignore records pointed to by invalid keys, so recovery only needs to ensure that valid keys are not lost.

In POSTGRES indices, there are two possible sources of inconsistencies: *inter-page* and *intra-page* inconsistencies. Inter-page inconsistencies occur when a pointer to page B is stored in page A. A failure could occur after A has been written to stable storage but before B has been. An intra-page inconsistency happens if a page is written to stable storage while the DBMS is adding a key to the page or deleting a key from it. Concurrency control prevents two processes from *modifying* a page at the same time. However, for performance reasons, POSTGRES does not reacquire a lock on the page when it forces the page to stable storage. If one process is modifying the page while another commits, the page will be inconsistent on stable storage. After a crash, the DBMS must detect the inconsistency and repair it.

5.3 Support for POSTGRES Indices

This section describes two algorithms for implementing indices in the POSTGRES storage system. We will describe both in terms of B^{link}-trees, but R-trees [32] can be managed using the same algorithms. Techniques analogous to those discussed for B^{link}-trees can be used with extensible hashing [26]. The application of our techniques to hashing is discussed briefly in [72].

This section describes the basic B-tree data structure, then the modifications to that data structure required for the POSTGRES shadow and page reorganization algorithms. Separate sections highlight the parts of the algorithms required to support B^{link}-trees, delete operations, and short term locking.

5.3.1 Traditional B-Tree Data Structure

In a traditional B-tree [10], each page of the tree contains an array of *<key,data>* pairs and a header that describes space allocation on the page (see Figure 5.1). The order of the keys on the page is recorded by a **line table**. Each entry of the line table contains an offset to the beginning of a *<key,data>* pair in the page. If a new key is added to a page, the line table entries are reordered, not the *<key,data>* elements stored on the page. On an *internal page*, the data element associated with a key is a pointer to a child page. On a *leaf page*, the data element associated with a key is a tuple identifier (TID) — a pointer to a data page and a line table entry on that page.

Comer [19] describes B-tree data structures in some detail, but several details of the insert and delete operations are important enough for our algorithms to summarize here. In the simplest B-tree, a split occurs when the amount of free space in a page goes below a

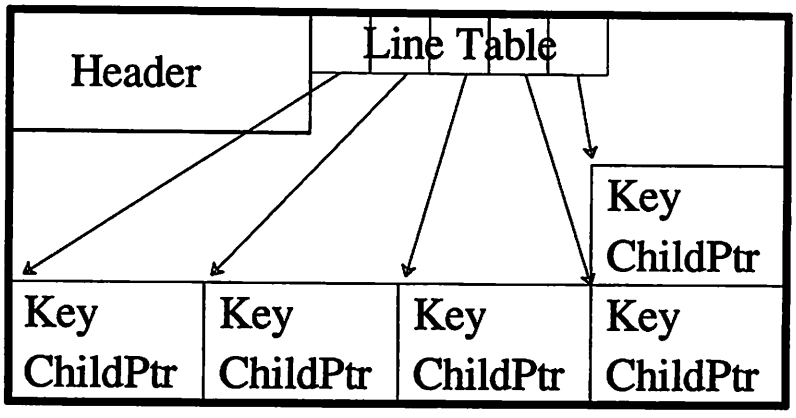

Figure 5.1: Conventional B-Tree Page.

threshold. To split a page, one new page is allocated. Half of the *<key,data>* pairs from the old page are inserted into the new one and deleted from the old. A *<key,data>* pair representing the new page is added to the split page's parent. When the last key is removed from a page, the page is freed.

Some variations of the B-tree data structure use a *merge* operation to rebalance two neighbor pages if inserts or deletes cause one page to have many more keys than its neighbor. Merge moves keys from the heavy page to the light one and adjusts the key value on the parent page to reflect the change. Simple variations on the basic POSTGRES page split algorithms will support page merges. These variations are described in Section 5.3.5 after the basic algorithms have been presented.

5.3.2 Sync Tokens and Synchronous Writes

The POSTGRES index management algorithms need to be able to determine whether two B-tree pages linked by pointers were written out during the same sync operation. To record this information, POSTGRES maintains a global **sync counter** that counts sync operations in which the B-tree underwent structural changes. After every sync operation in which an index split occurred, the DBMS increments the global sync counter. A **maximum sync counter** guaranteed to be larger than the global sync counter is maintained on stable storage. If the current global sync counter approaches the maximum, a new maximum must be chosen and written to stable storage. After a crash, the maximum sync counter is used to reinitialize the global sync counter.

A **sync token** is the value of the global sync counter at one point in time. Sync tokens are saved on index pages to detect inter-page inconsistencies. The value of the maximum

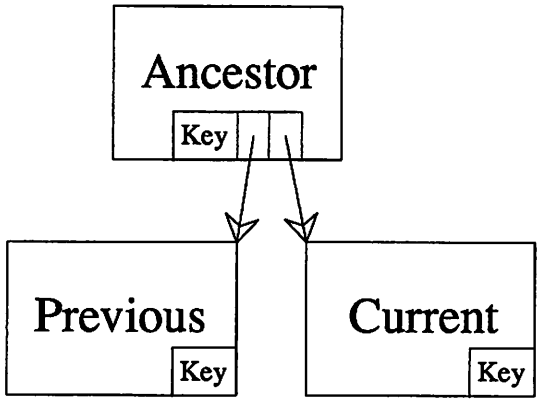

Figure 5.2: Shadowing Page Strategy. Keys on internal pages of the tree contain a prevPtr and a childPtr. The childPtr points to the most up-to-date version of the page (current). Because current might be on volatile storage, prevPtr points to the most recent version of the page that has definitely been written to stable storage.

sync counter at the time of the most recent system crash is called the **last crash sync token.** If the DBMS shuts down cleanly, the global sync counter and last crash sync token are written to stable storage.

5.3.3 Technique One: Shadow Page Indices

In POSTGRES shadow B-trees, every key on an internal page contains a pointer to the current and previous version of the child page associated with the key. Instead of an array of *<key,childPtr>* pairs on the page, the shadow B-tree page is an array of *<key,childPtr,prevPtr>* triples (see Figure 5.2). The previous page associated with a key is a page containing the key value which is guaranteed to be on stable storage. The current page pointed to by childPtr is the most up-to-date version of the page, which may be stored in volatile memory. If the system crashes and the current page is lost in the crash, the previous page will be used to construct a new current page in a manner described below.

Page Split Algorithm for Shadow B-trees

When splitting a page P in the shadow B-tree, two new pages are allocated — call them P_a and P_b. Half of the keys from P are copied to P_a and half to P_b. During the split, the keys on P are neither modified nor overwritten. When P_a and P_b are initialized, the value

of the global sync counter is recorded in a **syncToken** field in each page's header.

After the split, P's parent page, A, must be updated. Page A initially contains a key K1 which points to P. The traditional B-tree split algorithm calls for a new key, K2, containing a pointer to P_b, to be added to A. In the shadow paging algorithm, A is updated in the following manner:

(1) The new key K2 is allocated on A. K2's childPtr field contains the page number of page P_b.

(2a) If P's sync token is different from the current global sync counter, P must have been written to stable storage already. In this case, the prevPtrs for both K2 and K1 are set to point to P, and P is added to an in-memory to-be-freed list. After the next sync operation, P will be added to the index freelist (see Figure 5.3).

(2b) If P's sync token is the same as the current global sync counter, the prevPtr for K1 must be reused since P is not yet on stable storage. K1's prevPtr is assigned to K2's, and P is freed immediately. This situation only occurs if two splits occur at the same key between sync operations (see Figure 5.4).

(3) K2 is inserted into page A's line table.

(4) K1 is modified so that its childPtr field contains the page number of P_a instead of P.

If adding K2 to the page A causes A to split, the same algorithm is followed unless A is the B-tree root page. If the root page splits, a new root page is created containing two *<key,data>* pairs pointing to the two halves of the old root. The first page of the index is a meta-data page containing a pointer to the current root of the tree. Like internal page keys, the root pointer must contain a previous and current page pointer.

In order to prevent an intra-page inconsistency, we must be careful when adding K2 to the line table. The line table entries are intra-page pointers, offsets within the page, which point to key values. The line table is ordered, so the line table entry following K1's offset is selected to hold K2's offset. The line table is extended by first copying the last entry in the line table one element beyond the line table, then incrementing the **nKeys** field of the page header. Next, all of the line table entries between K1's and the last one are copied one entry to the right of their current position. Finally, K2's offset is saved in the entry after K1's.

Adding elements to the line table in this manner limits the kind of intra-page inconsistency that can occur. Even if one transaction forces a B-tree page to stable storage while another is adding a line table entry, we guarantee that the only possible intra-page inconsistency is a duplicate entry in the line table. The subsections below explain how these are detected and removed.

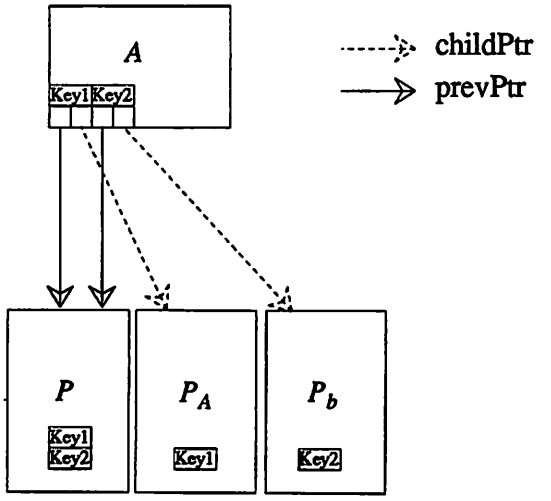

Figure 5.3: Shadowing Page Split. Page P has split. After using the syncToken to verify that P is on stable storage, the original prevPtr value for Key1 on page A is discarded. P becomes the previous page for both Key1 and the new Key2.

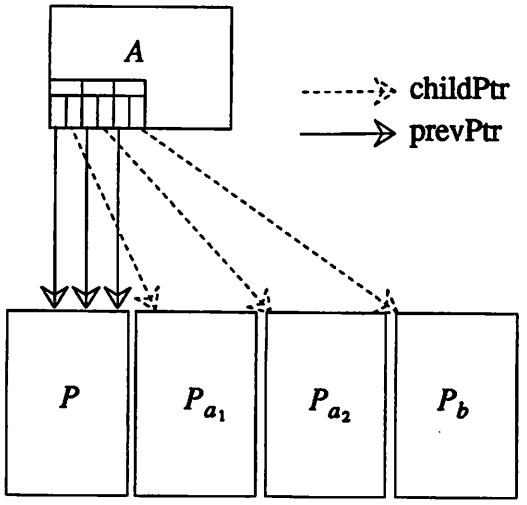

Figure 5.4: Two Page Splits During the Same Transaction. First P split then P_a split in the same transaction. P_{a_1},P_{a_2}, and P_b all share the same previous version since any key on any one of these pages that existed before the failure is recorded stably on page P.

Detecting Inconsistencies in the Index

Section 5.2 pointed out that, in POSTGRES B-trees, only two kinds of inconsistencies could potentially arise after a failure: inter-page and intra-page inconsistencies. Intra-page inconsistencies occur when a duplicate line remains in the page as described in the previous subsection. A crash during a B-tree update can cause an inter-page inconsistency only if the parent, A, is written to stable storage before the crash, but not the child. In that case, A points to an uninitialized page or a page that has been reused. If A was not written, then the new child page is inaccessible, but the parent-child link is consistent. Reclamation of pages that become inaccessible in a crash is discussed in a subsection below.

The key whose insert originally caused an interrupted page splits may or may not have been lost in the failure, but, because of the POSTGRES force-at-commit policy, that key will not make the index inconsistent. If the key is present, it is certainly uncommitted. The transaction that caused the interrupted page split must have been aborted by the crash. POSTGRES transactions force all writes to disk at commit time, so the split could not have been interrupted if the transaction had committed. If an uncommitted key is on a leaf page, it points to an invalid heap record (or no heap record) and POSTGRES will ignore it as explained in Chapter Four. The *committed* keys in the subtree rooted at any B-tree internal page are the same whether the split occurs or not. Thus, the failure effectively causes one or more spontaneous page splits, but does not affect the committed contents of the index.

POSTGRES detects both inter-page and intra-page inconsistencies in the index during the course of normal index operations. When descending from A to P during a key lookup, insert, or delete, the DBMS determines from A the minimum and maximum key values that should be on P before stepping from A to P. At P, the minimum and maximum key values actually present on the page are compared to the expected key range. If the key ranges are the same, the parent-child link is consistent and the search can continue. If the key ranges differ or if the page is zeroed, the DBMS has detected an inter-page inconsistency.

The DBMS detects an intra-page inconsistencies by checking whether or not adjacent entries in the line table contain the same offset value. Intra-page inconsistencies only need to be detected and repaired when a key is added to or deleted from a page. The duplicate entry will not cause key lookups to fail, so it can be ignored during key lookups.

Repairing Inconsistencies in the Index

As soon as a broken inter-page pointer link is discovered, the DBMS completes the work lost in the interrupted page split operation. The prevPtr shows the page that existed before the split. To reinitialize the out-of-date child page, the DBMS uses the keys on the parent page to determine the range of keys that were on the missing page. These keys are copied directly to the child page from the page pointed to by prevPtr. The sync token on

the child page is initialized to the current global sync counter. After the child page has been reinitialized, the B-tree search can continue using the new child page. Note that it is possible that both halves of the page split were lost in the crash. If that is the case, the loss of each is detected and repaired independently.

If the root page is split and the new version of the root is lost, the prevChild page is copied directly to the child page. If no root page existed before the failure (i.e. all keys inserted into the tree were lost), the root has no prevChild page and is initialized to an empty page.

The DBMS repairs an intra-page inconsistency by deleting the duplicate entry. The DBMS copies line table entries left until the duplicate is the last entry in the line table, then, decrements nKeys in the page header.

Free Space Management

During normal operation, a linked list of pointers to the pages freed from an index is kept on an in-memory freelist associated with that index. Because the freelist is in volatile storage, it does not survive system failures and must eventually be regenerated. As discussed in Chapter Four, POSTGRES heap relations require a garbage collector as part of the storage system's archiving feature [66]. Adding index freelist regeneration to its current archiving tasks does not make garbage collection much more expensive. While the freelist is being regenerated, new pages can always be allocated by extending the index file as long as the file system does not run out of disk space. We assume that crashes are infrequent enough and disk space is plentiful enough that the index file can be extended while the freelist is being regenerated.

The volatile memory freelist is only lost if the system fails. When the DBMS is shut down cleanly, the index freelist is written to disk. Index meta-data records the number of entries in the freelist and a pointer to the list on disk. When the DBMS is restarted, the freelist from disk is used to initialize a new in-memory freelist. Before any of the pages from the freelist are used in new page splits, the meta-data pointer to the freelist on disk is invalidated. The list has to be invalidated on disk since all pages on the disk freelist will become free again after a failure. If pages are taken from the in-memory freelist in the mean time and allocated to page splits, these pages could be reallocated when the DBMS restarts.

The freelist in POSTGRES indices also must record information about the contents of the free page in order to ensure that broken parent-child pointer links in the shadow B-tree will be detected. To show what information is necessary, we first review how the freelist is used in a shadow B-tree page split. First, two new pages, P_a and P_b, are allocated from the free list. Next, half the keys from the original page P are copied to each of the

newly-allocated pages. Then, a reference to P is added to a to-be-freed temporary freelist, whose contents are added to the true freelist only after the page split has been committed. Finally, P's parent A is updated so it contains pointers to P_a and P_b. Remember that when the DBMS later descends from A to P_a during the search for a B-tree key, the DBMS compares the range of keys on P_a to the range of keys that A indicated would be on that page. If the child page contains a different key range, an inter-page inconsistency has been detected; page A was been written to stable storage before a failure, but not P_a.

This technique for detecting inter-page inconsistencies restricts the way the DBMS can allocate pages from the freelist to hold the new child pages P_a and P_b. At the time of the page split, the page allocated to P_a from the freelist contains whatever keys were on that page at the time it was deallocated. Inter-page inconsistencies will not be detected unless the keys contained on the freelist page allocated to P_a in the page split are not legal contents of page P_a. If the freelist page and P_a contain the same key range, the DBMS will be unable to determine if P_a was written out to stable storage before the system failed. In order for the inter-page inconsistency to be detected, the freelist must record the key ranges of the pages in the list. When a page P is deallocated during a page split, the first and last key value on P must be recorded in the freelist along with the usual pointer to the deallocated page. This allows the DBMS to check that the page is not reallocated to hold the same key range.

5.3.4 Technique Two: Page Reorganization Indices

The B-tree modifications described above add four bytes to each key on an internal page (for a prevPtr). If keys are small, the extra four bytes will reduce B-tree fanout and increase the height of the tree. Increasing the height of the tree increases the average cost of data access.

The page reorganization algorithm reduces this loss of fanout by eliminating the prevPtr from the *<key,data>* pairs in a B-tree page. In this algorithm, however, splitting page P does not reclaim space on the page immediately. During the split, the DBMS copies half the keys on P to a new page and reorganizes P according to the algorithm described below (see Figure 5.5). After reorganization, P's original keys are intact on the page, so space has been made available on the new peer but not the original page P. If the DBMS ever fails after P is written to stable storage but before P's new peer is, no keys are lost. The reorganized page P can still be used for recovery. Once a sync operation successfully writes both the reorganized P and its new peer to stable storage, the space on page P containing the duplicated keys is reclaimed. If the DBMS must add keys to the original page P *before* the next sync operation, it initiates an extra sync operation and blocks until the sync completes. Once the sync operation is done, the space containing the duplicate

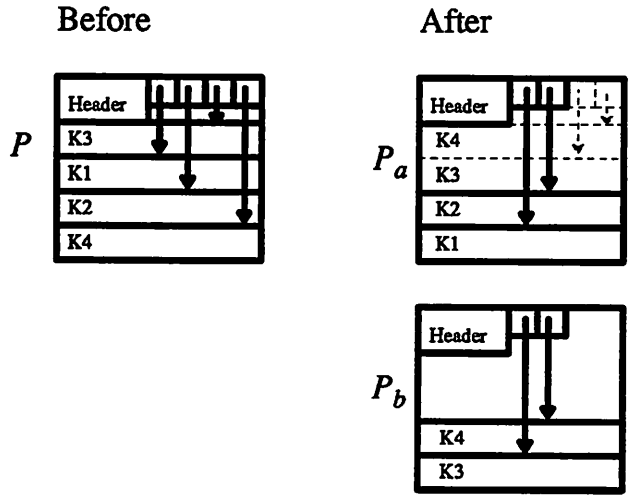

Figure 5.5: Page Split For Page Reorganization B-Trees. After the split, the reorganized page P_a is mapped on top of the old page P on disk. Keys K3 and K4 are saved in the free space region of P_a. If all of the split pages are successfully written to stable storage, the area containing K3, K4 and the corresponding line table entries becomes free space.

keys on P can be reclaimed and the DBMS can add a new key to the page.

The page reorganization algorithm adds the fields **prevNKeys** and **newPage** to the page header. If the prevNKeys field on a page is non-zero, the page still contains backup keys to be used in recovery. If prevNKeys is zero, the page is safe for update. Below, we describe a split of page P into P_a and P_b. P_a is the reorganized page. P_b is the page that will contain the new key that caused the split. Note that P_a may be either the left or the right child after the split. The newPage pointer in the reorganized page (P_a) points to P_b; newPage in P_b is null.

A split of page P proceeds as follows:

(1) Two new pages are allocated. P_a is allocated in memory only; it is not backed up on the disk. P_b is allocated normally.

(2) Half of P's keys are copied to P_a and half to P_b, just as in a normal split. The prevNKeys field on P_b is initialized to zero. On P_a, it is initialized with the number of keys on the original page P.

(3) The keys from P_b are now copied to the free space area of P_a. These keys are not *allocated* on the page, just copied into the page's free space region. A line table for

the keys is set up just beyond the line table for P_a. P_a is guaranteed to have space enough for P_b's keys and line table because all of this information was stored on the original page P.

(4) The sync tokens of P_a and P_b are initialized using the global sync counter.

(5) P_a is remapped (in the in-memory buffer pool meta-data) to P's location on disk.

(6) The new key whose insertion caused the split is added to P_b. P's parent page is now updated to reflect the split.

Detecting and Repairing Inconsistencies

POSTGRES uses the same technique for detecting inter-page inconsistencies in the page reorganization B-trees as it did in the shadow page B-trees. When the DBMS is searching for a key, it steps from parent page to child page. At each step, the DBMS checks that the key range on the child is consistent with the key range indicated by the parent. Intra-page inconsistencies are detected and repaired in the same way in both types of B-trees.

Repairing inter-page inconsistencies is slightly more complex in the page reorganization B-tree, however. In the shadow B-trees, inter-page inconsistencies could occur only if the parent page was written to stable storage before either of the new child pages created in the page split. In the page reorganization B-trees, the children are not symmetric so five different kinds of inconsistencies can occur:

(a) only P_a is written to disk (replacing P),

(b) only P_a and P_b are written (P_b is inaccessible from the parent),

(c) only the parent and P_a are written,

(d) only the parent and P_b are written,

(e) only the parent is written.

If only P_b is written, the tree is not inconsistent (but page P_b is lost). Note that each of these inconsistencies will be detected by the same kind of range check used in the shadow B-tree.

As was the case in shadow B-trees, the inconsistencies are repaired as soon as they are detected. In cases (a) and (b), the tree becomes consistent by regenerating P (assigning prevNKeys to nKeys reallocates the duplicate keys). In case (c), P_b is regenerated by copying the duplicate keys saved on P_a. In case (d), P_a is regenerated by removing the keys that are represented on P_b. In case (e), the split is repeated to generate both P_a and P_b.

Every time a key is added to or deleted from a page, the DBMS must check whether or not the free space on the page needs to be reclaimed. If the prevNKeys field is zero, there are no extra keys stored in free space. Otherwise, the sync token on the page must be checked. There are three cases:

(1) If the sync token is the same as the global sync counter, no sync operation has occurred since the page was initialized, so the duplicate keys on the page are still required for recovery. The DBMS must block for a sync operation before the key can be added to the page.

(2) If the sync token is greater than or equal to the **last crash sync token** but different from the global sync counter, the new key can be added normally. A sync operation has definitely committed P_a and P_b, and the keys on P_a will no longer be needed for recovery.

(3) If the page sync token is less than the **last crash sync token**, we cannot immediately tell if the split was committed successfully. The DBMS has crashed since this page was written. If the page's sibling from the last split was lost in the crash, the backup keys on this page are still needed for recovery.

In the last case, the newPage pointer is used to find the sibling. If the sibling exists and has the same sync token as the current page (or a larger one), the sibling does not need to be recovered; the parent and both halves of page P made it to stable storage after the split. If the sibling is zero or has an older sync token, the sibling is out of date and must be recovered. After a new key is inserted, the prevNKeys field should be zeroed so we do not check for inconsistencies again until the next page split.

5.3.5 Delete, Merge, and Rebalance Operations

In a conventional storage system, deleting a record from the database forces the DBMS to delete all index keys that refer to that record from the database as well. If the transaction that deleted the record aborts, the DBMS must reinsert the record and all of the index keys that referred to it. As Chapter Four explained, POSTGRES is not a conventional system. When a record is logically deleted, it remains physically in place but is marked invalid. When the DBMS encounters an index key that points to a logically-deleted record, it is ignored. Eventually, a vacuum cleaner process deletes the record and its related index keys.

This strategy means that the index recovery algorithms used by POSTGRES do not need to consider the problem of reinserting index keys after a failure. The vacuum cleaner only physically deletes index keys when the transaction that logically deleted them has definitely committed. If the DBMS halts without completing a given index key delete

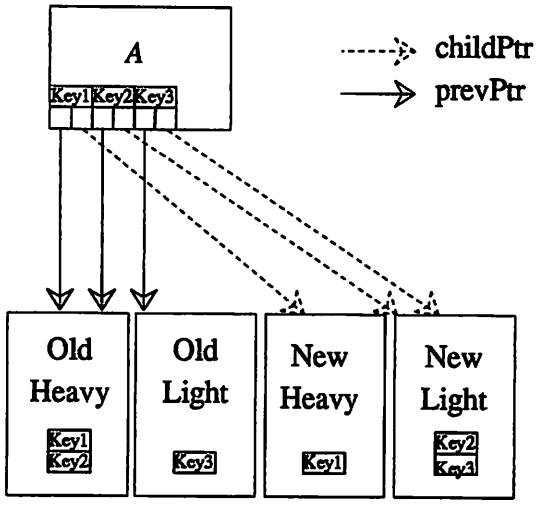

Figure 5.6: A Merge Operation on a Balanced Shadow B-Tree. Some keys, including Key2, have been moved from the heavy page to the light page in order to even the sizes of the two pages. On the ancestor page, A, a dummy key has been added to represent the keys moved from heavy to light.

operation, the vacuum cleaner will eventually encounter the key again after DBMS restart and delete it. Therefore, the only recovery-related problem that needs to be considered in delete operations is ensuring that no structural inconsistencies in the index occur as a result of failed delete operations.

For the simplest kinds of B-trees, deletes have less potential for causing inconsistencies than inserts. Delete operations remove inter-page pointers from pages rather than store them on pages. Thus, deletes never leave pointers to allocated but uninitialized pages as occurred in page splits. In the simplest B-trees, a page is ready to deallocate when the last key on that page is deleted. When a page P is empty, Ps key on the parent page is deleted. As was the case with prevPages in the shadow algorithm, P cannot actually be deallocated until the parent has been written to stable storage in the next sync operation. Delaying P's deallocation ensures that it will not be reallocated while pointers to the page still exist in valid parts of the index on stable storage. Intra-page (line table) inconsistencies resulting from interrupted deletes look exactly like interrupted inserts (duplicate entries remain in the line table), and are handled in the same way.

In general, the merge operations required by balanced B-trees (B*-trees) can be handled by the recovery algorithms in the same way as page splits. Page reorganization can treat merge operations exactly like splits. When the merge operation moves keys from the heavy

page to the light page to balance the two, it leaves the two peers in exactly the same state as two page reorganization peers: the heavy page is treated as the original peer in the split, and the light page is treated as the "new" peer created by the split. The "new" peer in this case initially contains a few keys, but the recovery mechanism will not need to be aware of this.

For shadowing, merge operations must be done a little more carefully since the new light page effectively has two prevPages, the original light page and the original heavy page. The merge proceeds as usual, keys are moved from the heavy page to the light page, however, instead of modifying the key for the light page on the ancestor, we add a new key. The new key represents those keys moved from the heavy to the light page during the merge operation. Its child page is the new version of the light page; its prevPage is the old version of the heavy page. After the new pages are written to stable storage, this dummy key and the light key can be merged in order to reclaim space on the ancestor page. See Figure 5.6 for an example.

The first five subsections of section 5.3 described shadow and page reorganization algorithms for managing basic B-tree operations without a write-ahead log. However, POSTGRES and many commercial systems use a slightly more complex variation on the basic B-tree called a B^{link}-tree. These data structures have additional pointers between pages to achieve better performance. Section 5.3.6 explains how these structures work and shows the changes required to support them without a write-ahead log. Section 5.3.7 discusses conventional index concurrency control algorithms and the ways in which they can be modified to support the POSTGRES index recovery techniques.

5.3.6 Secondary Paths to Leaf Pages: B^{link}-tree

In B^{link}-tree indices, the performance of indexed scans is improved with a doubly-linked **peer pointer** chain between leaf pages with consecutive keys (see Figure 5.7). The peer pointers allow scans to move from leaf page to leaf page without reading additional internal pages. Key inserts still traverse the path from root to leaf. When a page is split, the left neighbor (or right and left, in the shadow page algorithm) of the page must be re-linked so that the peer pointer path is consistent.

B^{link}-trees have more complicated failure modes than simple B-trees. There are two paths to any given leaf page; a key on the leaf page may be reached by either the peer pointer or the root-to-leaf path. Techniques like those described above could be used to correct inter-page inconsistencies in either path, but, in the worst-case failure mode, the two paths could become inconsistent with one another. For example, in Figure 5.8, the root-to-leaf path contains the post-split version of a given page (in bold), while the old peer pointer path contains the pre-split version of the page.

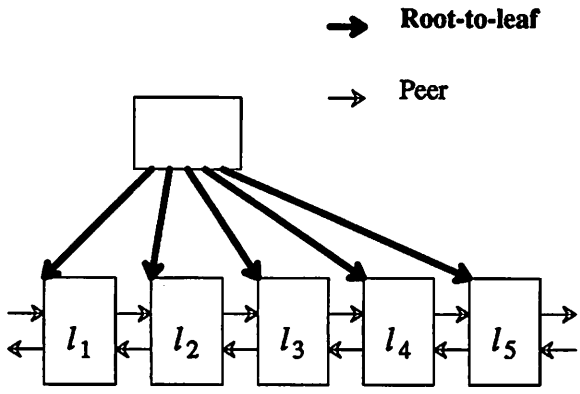

Figure 5.7: Normal B^{link}-Tree. Leaf nodes l_i are connected to one another by peer pointers. The path from parent to child is referred to as the root-to-leaf-path.

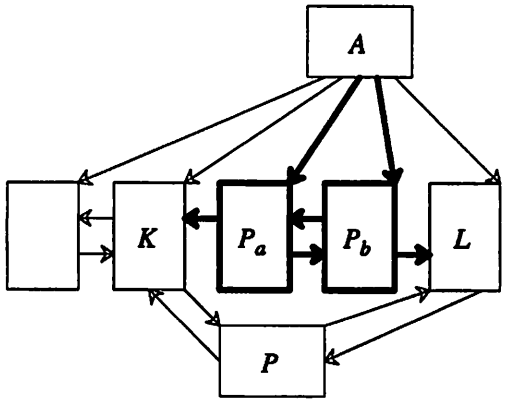

Figure 5.8: Worst-Case Inconsistent B^{link}-Tree. Page P has split and A, P_a, and P_b were written to stable storage before the system crashed. P's peers, K and L, were not. Thus, the tree has a peer pointer path consistent with the tree before the split and a root-to-leaf path consistent with the tree after the split.

Even this worst-case failure does not actually corrupt the index unless a key is added to or deleted from one of the duplicate pages created by the failure. The transaction whose incomplete split created the duplicate paths did not commit (otherwise both paths would have been successfully written to disk). Until the first insert/delete after the failure, the duplicate pages contain the same set of valid keys.

Detecting Inconsistencies in the Index

During a B^{link}-tree scan, the peer pointer path is checked for inter-page inconsistencies. Unfortunately, the key ranges used to detect inconsistencies in the root-to-leaf path cannot be used for the peer pointer path. On the peer pointer path, a page does not know its peer's key range and cannot record it accurately unless each page is updated when keys are added to its peer.

To detect inconsistent peer pointer paths, we use two additional sync token fields which must be included in the page header — one associated with each peer pointer. If P1 and P2 are peer pages, P1's pointer to P2 and P2's pointer to P1 must have the same sync token associated with them. When the peer pointers are reconciled during the split, the sync tokens for the peer pointers on the neighbor pages must be reset also.

Comparing two peers' sync tokens during path traversal will detect any inconsistency in the path. If a link is broken by a crash during update, the sync tokens on adjacent pages will not agree. An inconsistent link is repaired by following the root-to-leaf path to the correct peer. If the root-to-leaf path is broken, it is repaired using one of the repair algorithms described above.

Even sync tokens do not detect the existence of two completely separate pointer paths as occurs in Figure 5.8. In this case, the peer pointer path is internally consistent (and the sync tokens match), but the peer pointer path is not consistent with the root-to-leaf path. Whenever a key is inserted into a page P, we must ensure that P is linked into the most recent peer pointer path.

When inserting a key into page P, the DBMS first checks that P's sync token is greater than the last crash sync token. If so, we know the page is part of a consistent peer pointer path. The path only becomes inconsistent during a system failure. Otherwise, the DBMS must follow the peer pointer path in both directions from the leaf page targeted for insert. The search stops when a page with a different sync token is discovered (page sync token *not* peer pointer sync token). If the peer pointer path is consistent until this point, the leaf page inserted into is reachable along the peer pointer path. Once this is done, we reinitialize the sync token on the page. This will prevent the DBMS for rechecking the path on subsequent insertions. Because we are inserting a key into the page, the page will be written to stable storage anyway. Thus, the reinitialized sync token will reach stable storage at the end of

transaction without causing any extra I/O.

In the worst case, searching this path is the most expensive part of this algorithm. If many page splits occur at the same time, the resulting pages have the same sync token. An insert into one of these pages, will cause each of them to be read. Even in this case, insert performance is affected only for the first key inserted into each page in the path after a failure; key lookup is not affected at all, even after failures. Insert performance after a crash could be improved in this worst case with a small LRU cache of sync tokens. When a sync token is verified (by searching the peer pointer path during an insert and finding no inconsistencies), the token should be added to the cache. On an insert, the cache is checked before verifying the peer path. A cache of size one would handle the worst-case, which occurs when a large index is created in a single transaction. In this case, each page in the index has the same sync token.

5.4 Concurrency Control

The POSTGRES B^{link}-tree implementation uses a concurrency control algorithm based on the one designed by Lehman and Yao [47]. In Lehman-Yao, readers and writers must descend the tree from root to leaf to find the page containing a given key. Writers ascend again as splits or deletes propagate up from the leaf. When descending, locks are not coupled; readers always release one lock before acquiring the next. When ascending, locks are coupled; the lock on a child page is released only after the lock on the correct parent page is acquired. As pointed out in [46], this algorithm is deadlock-free, since lock coupling is only used when traversing the tree in one direction. Lock coupling in both directions allows deadlock when a reader holding a lock on ancestor, A, tries to acquire a lock on child page P at the same time a writer holding a lock on P (during a page split) tries to lock A.

Complexity arises in Lehman-Yao from the fact that a reader descending from A to P may find that P has split during the period when the reader was not holding any locks. When descending, the reader saves a pointer to a child page P, releases the lock on the parent, and acquires the lock on P. It is possible for the reader to be descheduled by the operating system right before it acquires the lock on P. Other processes could split P before the reader is rescheduled. In the original Lehman-Yao scheme, the page split operation could move the key sought by the reader from P to a neighbor page. Pages are never deallocated in Lehman-Yao B-trees and a page split always moves the higher-valued keys to a new page, leaving the lower-valued ones in place. Thus, if the reader finds that the key it is searching for is no longer in P, the reader moves horizontally in the tree (again, without coupling) until it finds the key. In the unlikely event that there have been many page splits during the descent, the reader may traverse many pages.

POSTGRES B-trees, especially the shadow B-trees, must account for page deallocation. Because POSTGRES pages can be deallocated after a split, the DBMS must ensure that, when a page is deallocated, no descheduled reader will reawaken and try to examine the deallocated page. Our algorithm calls on the reader to pin the buffer containing the child page in memory before releasing the parent lock. The allocator knows not to reallocate pages in buffers with a pin count greater than one. The reader may unpin the buffer as soon as the child's lock is released. This solution does not add synchronization overhead since the buffer must be pinned in memory before use anyway. Lanin and Shasha [46] discuss two more complex techniques for solving this problem in the case of pages recycled after the last key is deleted.

Also unlike Lehman-Yao, the reader process in POSTGRES shadow B-trees must find out which pages were produced in the split of the child page P. Lehman-Yao guarantees that P itself is one of the pages that results from the split. The reader process can start its horizontal movement from the original page P and be guaranteed to find any key that was on P at the time of the split. For POSTGRES page reorganization B-trees, this is still true. For shadow B-trees, the page P was replaced with two new pages. To allow the reader to find these pages, we add a **page replacement pointer** to the B-tree page header. The page replacement pointer on the original page is set to point to the new left page. Whenever a process visits a page with a non-null page replacement pointer, it traverses the link to the new left child. This is analogous to the horizontal movement described above, required when the key of interest was on the high half of a split page. Note that the page replacement pointer is only of interest when the page is pinned in memory by a current reader. It does not ever need to be written to disk and does not need to survive failures.

The original Lehman-Yao locking algorithm also assumed that peer pointers were unidirectional; each page only had a pointer to its right peer. This restriction means that rightward scans are faster than leftward scans. In order to eliminate the restriction, we introduce a new locking protocol to ensure that peer pointers are adjusted correctly. The POSTGRES protocol relies on a new type of lock called a *split lock* that allows us to distinguish page splits from reads and writes. Split locks conflict only with split locks. Only the process holding a split lock can split a page or add keys to a page. Other processes, however, may adjust peer pointers on the page without holding the split lock.

The protocol will be described in detail below, but the description will be easier to follow if it is clear how bidirectional peer pointers can give rise to deadlock. When a DBMS process splits a page, it first acquires a lock on the page to ensure that, during the split, no other processes add keys, delete keys, or concurrently attempt to split the page. When the split is complete, the process must adjust the peer pointers so that the new pages resulting from the split are accessible from the original page's neighbors. To adjust the neighbor page's peer pointers, each neighbor page must be locked. This situation is a case

of lock coupling. The DBMS process is holding a lock on the page being split and acquiring a lock on its neighbor. If two adjacent pages are split concurrently, a deadlock can occur as each process holds its own page and tries to acquire the neighbor. In the unidirectional pointer case, processes never lock couple in opposite directions so deadlock never occurs. Deadlock is possible in the bidirectional case because two processes are lock coupling in opposite directions.

POSTGRES uses normal (write) locks on pages in combination with the new split locks in order to avoid deadlock when two processes lock couple in opposite directions. When a DBMS process inserts a key into a page, it first acquires a write lock on the page to prevent other processes from inserting keys at the same time. If the process finds that a page must be split, it releases the write lock, acquires a split lock, and reacquires the write lock. Then, if the split is still necessary (someone else could have gotten the write lock and split the page after the process released the write lock), the process splits the page. Finally, the write lock on the original page is released and peer pointers on neighboring pages are updated. Updating a neighboring peer pointers requires a write lock on the neighbor page, but not a split lock on the neighbor page. The split lock on the original page is released once the neighbor's peer pointers have been updated.

Deadlocks are impossible since processes acquire the split lock before the write lock, and acquire only one such pair in the tree at a time. Because split locks and write locks do not conflict, processes can hold a split lock on one page and acquire a (write) lock on a peer without causing deadlock.

Concurrent access can make inter-page links temporarily inconsistent, so our algorithm must distinguish between true inconsistencies and false inconsistencies that arise during concurrent updates. When a link token inconsistency is discovered, the two inconsistent tokens are compared to the last crash sync token. If one or both of the inconsistent tokens is more recent than the last crash sync token, then the inconsistency was a transient one caused by concurrent access. If both are older than the last crash sync token, the inconsistency could not have been caused by a concurrent update.

5.5 Using Shadow Indices in Logical Logging

Thus far, we have discussed our index management techniques in terms of the POSTGRES storage system, however, the same techniques can be used to support logical logging in a conventional WAL-based storage system. Conventional index management schemes, such as the one used by ARIES/IM [54], require all modifications to the index to be written into the log. If a tuple is updated, the DBMS logs all keys inserted into indices as a result of the update. If an index insert results in a page split, all keys moved from one page to

another in the split must be logged as inserts into the destination page. Deletes from the original page in a split are logged simply as changes to the line table (i.e. an abbreviated log record is constructed that tells which key range was moved from the page in the split). As stated in the introduction to this chapter, conventional systems use the logged information to restore the index to consistency after a failure.

A logical logging scheme does not save index changes in the log. When a tuple is updated, changes to the tuple are logged but not the keys inserted into or deleted from the index. Instead, the logged tuple attributes serve as implicit log records for the indices on those attributes. During recovery, the index keys affected by an update can be derived from the logged attribute values. If the logged change is undone or redone, the DBMS deletes or inserts keys into the indices as necessary. The DBMS must detect and ignore reinsertion or redeletion of the same *<key,data>* pair.

The difference between the logical log and the conventional log is that the logical log contains only the keys inserted into or deleted from the index. It does not log keys that move around within the index due to page split and merge operations. While the logical log allows the system to determine which keys have been inserted into or deleted from the index, it does not maintain the structural integrity of the index. Some other technique, such as the ones described in this chapter, must be used to maintain index consistency during page splits. A conventional system using the POSTGRES index consistency techniques would not need to sync the index after every transaction. In the POSTGRES storage system, the DBMS had to sync the index after every transaction in order to make the keys inserted or deleted by that transaction permanent. Syncing the logical log makes inserts and deletes to the index permanent when logical logging is used. Log processing will restore any keys lost during the failure.

Logical logging has some performance and disk space advantages over conventional index management. The conventional log is longer than the logical log, since conventional logs store many *<key,data>* pairs after a split or a merge. Because the conventional system must log all keys moved from the original page to the new peer page, each page split adds at least half a page to the log (8 KBytes and 4 KBytes are typical page sizes). The longer log means more data needs to be written to disk on commit, and more log pages need to be read from the disk during recovery. The conventional log log takes up more space on disk as well.

More importantly, logical logging has some fault tolerance advantages over conventional B-tree management. Little special case code is required for recovery. The same insert and delete operations used for normal execution are also used for recovery. Specialized recovery code includes only the code to repeat the incomplete page split after an inconsistency is detected. Also, because logical logging stores a high level representation of index operations, systems using it are less likely to propagate damage caused by software errors

into the log. If, for example, an internal index page is corrupted by a software error, conventional physical logging techniques can copy the corrupted keys into the log. During recovery, the corrupted keys will be restored to the index. Logical logging *never copies information from the index into the log*. If software corrupts an index, the index can be recovered using a backup version (or checkpoint) and the log.

When comparing System R to ARIES, Mohan and Levine [54] suggest four reasons why the write-ahead logging techniques used by ARIES are superior to the shadow-based logging approach used in System R[30]. These four objections do not apply to logical logging using shadow B-trees:

(1) Deadlocks During Undo: The usual response to a deadlock is to abort one of the deadlocked transactions. Since abort requires an undo, the potential for deadlocks during undo means only one aborting transaction can be active at a time. The lock coupling strategy described in Section 5.4 prevents processes from deadlocking during index operations. Therefore, concurrent aborts can execute concurrent shadow B-tree operations.

(2) Concurrency Overhead During Recovery: If several processes are used for recovery in System R, concurrency overhead is incurred during logical undo and redo operations. ARIES requires no concurrency control for the index during recovery because recovery operations can be applied to each page independently. Parallel recovery of shadow B-trees will have to use concurrency control just as System R did, but the locks involved are short-term locks, not two-phase locks. Recent simulation results indicate that when short term locks are used concurrency control overhead will not limit recovery performance. In [65], the concurrency control scheme from [47] was simulated on a DBMS running a100% insert workload with enough main memory buffering for 75% of the B-tree. The simulations showed that the workload was I/O-bound even at high degrees of multi-programming. If concurrency overhead had significantly affected performance, the simulation workload could not have been I/O bound, especially with such a large amount of buffer space available.

(3) I/O Overhead During Recovery: System R and shadow B-trees require more I/O operations during recovery than ARIES because logical undo operations must traverse the path from the root to leaf for every operation undone or redone. ARIES' page-oriented recovery can usually undo or redo an operation with a single read and write of a leaf page. The additional I/O required by the shadowing scheme, however, is small. The root and the upper pages of the B-tree index must be loaded into memory as the first few operations are processed. Unless memory is scarce, these pages will remain in memory during the rest of log processing. Page-oriented recovery may not require these pages to be brought in during recovery, but the pages will have to be brought in before any useful work is done with the index after recovery. Also, operational logging will actually reduce the number of disk reads required to process the log since the log itself is much more compact.

(4) B-tree Consistency After Failures: DBMS failures can leave indices inconsistent

unless the file system uses shadow paging. Mohan and Levine's objections to maintaining index consistency with shadow pages are based on the poor performance of shadow paging in System R.

Because System R used shadow paging in the file system, it had to use the technique to support recovery on both indices and data files. For data files, shadow paging reduced the performance of sequential scans dramatically. Shadow paging makes sequentially-ordered pages in the file non-sequential on the disk. The techniques also force an extra lookup (through the page map) for direct access to file pages. The consistency maintenance techniques described in this paper allow either no shadowing at all (page reorganization algorithm), or shadowing limited to index files only. In indices, the sequential order of the pages on the disk is unimportant for performance. As shown in the next section, our shadowing-based algorithm does have an impact on performance, but not as pronounced as the impact of shadow paging on System R's data files.

In summary, even in a DBMS that relies on conventional write-ahead logging instead of the POSTGRES storage system, the index recovery techniques from Chapter Five can be helpful. Using our index recovery techniques in conjunction with logical logging reduces the amount of information stored in the log, giving both performance and fault tolerance advantages over more conventional index management. While a similar logical logging scheme caused performance problems in System R, the POSTGRES techniques have been designed to avoid these problems.

5.6 Performance Measurements

The index management techniques described in this chapter increase the cost of indexed access to the data in the database in several ways. First, shadow B-trees have larger space requirements than conventional B-trees. The prevPtrs stored in the shadow B-tree keys make the keys bigger so fewer keys fit on a page. Thus, shadow B-trees will eventually become higher than conventional B-trees with the same number of keys. Higher B-trees mean more pages will have to be accessed to get to the indexed data. In order to illustrate this cost, Section 5.6.1 presents a comparison of shadow and normal B-tree heights. Second, the DBMS must check for inter-page inconsistencies as it descends from page to page in the tree (the key-range checks are described in Subsection 5.3.3). To quantify the cost of checking for inter-page inconsistencies, we have implemented both techniques and measured the implementations. Section 5.6.2 presents these measurements. Third, several special cases cause POSTGRES B-trees to do an extra disk read or write either during recovery or during a page split. Section 5.6.3 enumerates these cases and estimates their impact on performance.

5.6.1 Modelling The Effect of Increased Tree Heights

One performance concern regarding POSTGRES B^{link}-tree indices is that the additional space overhead they incur will increase the height of the tree, thus driving up access costs. In order to quantify this cost, we calculated the index capacity at fixed heights for normal, page reorganization, and shadow B^{link}-trees. As expected, normal trees add levels the most slowly, and shadow trees add levels the most quickly. Page reorganization trees grow at nearly the same rate as normal trees, so we have omitted them from the analysis that follows for the sake of brevity.

Figure 5.9 illustrates the differences in height between normal B-trees and POSTGRES shadow page B-trees for different tree sizes. The curves in Figure 5.9 labelled "Normal 4-Byte" and "Shadow 4-Byte" show the heights of normal and shadow B^{link}-trees storing four-byte keys. The curves labelled "Normal 20-Byte" and "Shadow 20-Byte" show the storage capacity vs. height tradeoffs for trees with twenty-byte keys. Note that the X axis in Figure 5.9 is logarithmic. The shaded regions highlight the tree sizes at which shadow trees have greater height than normal trees. For all regions of the X axis which do not have values in the shaded areas, shadow and normal trees have the same height. The trees modelled have 8-KByte pages. Normal B^{link}-trees have 6-byte internal page keys while shadow B^{link}-trees have 10-byte internal page keys because of the 4-byte prevPtr. The page header in a normal tree is 16 bytes while the shadow tree header is 36 bytes because of sync tokens and the replacement pointer.

The growth rate used in the calculations is pessimistic for the shadowing strategy, since tree height is calculated assuming that keys are inserted in worst case order (ascending values). Ascending order leaves the maximum amount of unused free space in the index and forces the tree to grow at the fastest rate. If the trees grew more slowly, the curves would have the same relationship to one another (the shaded regions would have the same area), but the steps would occur at larger tree sizes. We used a page size of 8 KBytes in the analysis, since this is the default in POSTGRES.

The figure shows that prevPtr overhead in shadow trees has lower impact as key size increases. At height three, the difference in capacity between the trees storing twenty-byte keys is much smaller than the difference between those storing four-byte keys. The space consumed by prevPtrs in internal pages causes a reduction in fanout, which eventually causes greater tree height at smaller capacities. The reduction in fanout caused by shadowing is a function of the ratio of overhead to key size. Larger keys have proportionally less overhead, hence, show a proportionally smaller reduction in fanout.

In practice, the space overhead for shadow index prevPtrs will usually not affect tree height, even when key size is small. Small trees have few levels of internal pages, so prevPtr overhead is negligible. The heights of B^{link}-trees with several levels will coincide for most

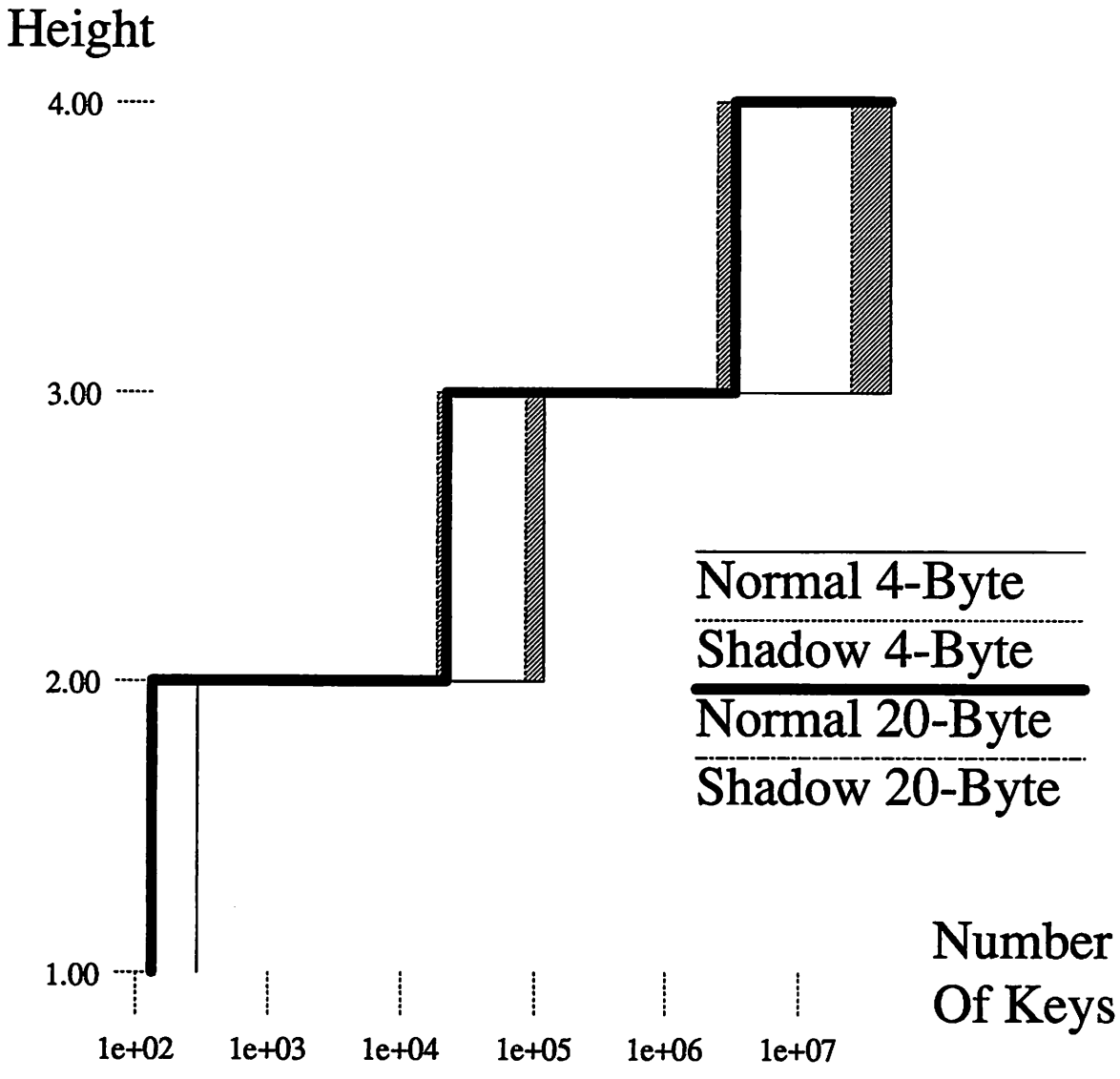

Figure 5.9: Height of Tree for Different Size B-Trees.

tree sizes, so the height impact of shadowing will still be minimal. If an intermediate-height shadow tree becomes stable at one of the non-coincident values, running a reorganization utility will redistribute free space and reduce the height of the index to the same level as a normal tree. Significant height differences that could not be masked through reorganization would arise only if keys were small and if the tree had many levels. However, even with the worst-case insertion order, a B^{link}-tree of either type storing four-byte keys would exceed the 2 GByte maximum size of a Unix file before it reached five levels.

5.6.2 Measurements of the POSTGRES B^{link}-Tree Implementation

To measure the performance of the shadow and page reorganization index implementations, we ran two tests against each type of index. The first test built indices of three different sizes using four-byte keys. As in the calculations of the previous subsection, these measurements give worst case performance; keys were added in ascending order in order to give the largest number of page splits and greatest tree height. The second test retrieved 8,000 random keys from each index created in the insertion test. Keys were uniformly distributed throughout the range represented in the index. Measurements were made on a DECstation 5000/200 running Ultrix 4.0 and POSTGRES.

The times shown in Table 5.1 are the mean elapsed times of ten repetitions of each test. The standard deviation of each set of measurements was less than 2.5% of the mean. Each entry in the table includes, in parentheses, a normalized time for that test. The normalized time is calculated by dividing the elapsed time for the test by the elapsed time of the conventional B-tree. For example, a shadow B-tree with a normalized read time of 1.02 is two percent slower than a conventional B-tree on the same workload. Only time spent in the B^{link}-tree access method, and in the routines that it calls, is reported in the table. This includes the cost of reading and writing index pages from and to the operating system cache, but does not include the cost of committing transactions. Commit cost will depend on the logging scheme chosen.

The results show that the shadow algorithm is within three percent of the cost of ordinary B^{link}-trees for insertions. The higher cost is due to the added expense of verifying inter-page links in traversing the tree. For reads, the shadow tree percentages are about three and a half percent worse than ordinary B^{link}-trees. These measurements only show the CPU costs of the algorithm; they do not account for extra I/O that would be necessary if the shadow tree is higher than the normal B-tree. In each of the cases shown here, the heights of the shadow B-tree and normalized B-tree are the same. For the ranges at which the shadow tree is higher than the normal tree, each shadow lookup would pay an additional I/O.

Costs for the page reorganization algorithm are similar. Reads are between three and four percent more expensive than for the normal tree. Page reorganization insertions, however,

Operation B-tree Type	Size of Index in Keys 10,000	 20,000	 40,000
Inserts			
Normal	12.065 s (1.000)	24.269 s (1.000)	51.307 s (1.000)
Page Reorg	12.584 s (1.043)	25.191 s (1.038)	53.718 s (1.047)
Shadow	12.318 s (1.021)	24.924 s (1.027)	52.282 s (1.019)
8,000 Lookups			
Normal	9.122 s (1.000)	12.492 s (1.000)	19.536 s (1.000)
Page Reorg	9.441 s (1.035)	12.879 s (1.031)	20.259 s (1.037)
Shadow	9.368 s (1.027)	12.892 s (1.032)	20.200 s (1.034)

Table 5.1: Insert/Lookup Performance Comparison.

are more expensive, between three and five percent higher than the cost for insertions into an ordinary B^{link}-tree. Extra work must be done to order data on old pages during splits in page reorganization. As noted elsewhere in this chapter, page reorganization is best suited to environments with low insertion rates.

The overall cost of using either index management strategy is likely to be very small for many workloads, since the DBMS spends little of its time in the index access methods. For example, in the Wisconsin benchmark [13], POSTGRES spends only 3.6 percent of its time in the indexed access methods. The debit/credit benchmark used int Chapter Three spends only 16 percent of its time in the index access methods. Even 4.7 percent of this, our worst performance degradation, is smaller than the measurement error in the benchmark.

5.6.3 Estimating Additional I/O Costs During Recovery

The POSTGRES index management techniques have several workload-dependent I/O costs that were not measured in the dissertation. In the normal case, a POSTGRES B-tree page split and a conventional B-tree page split each require three pages to be written to disk: the parent and each child. A page reorganization B-tree, however, will force a synchronous page write if the same index page splits twice during the same transaction. If keys are four bytes long and pages are 8 KBytes, inserting 292 keys in the worst-case order during a single transaction could cause this additional synchronous write.

The other workload-related I/O cost occurs the first time keys are inserted into some page reorganization B-tree or B^{link}-tree pages after a failure. In both trees, inserting a key into a page P sometimes requires the DBMS to read additional pages to determine whether the page split that created P was committed. For example, when a page split occurs in a page reorganization B-tree, the duplicate keys on the reorganized page cannot be overwritten unless the peer page has definitely been written to stable storage. The first time a key is inserted into the page after the split has been committed, the page is marked (prevNKeys is cleared) so later key insertions do not have to consider the state of the peer. When no failure has occurred since the split, comparing the sync token on the page to the global sync counter provides this information without examining the peer. If the first key insertion to a reorganized page occurs after a failure, however, the peer must be read and examined to ensure that it has the same token as the reorganized page (or a larger token). When the first key insertion to a B^{link}-tree page occurs after a failure, the DBMS must check that the page is linked into the peer pointer path as explained in Section 5.3.6. Again, extra work must be done if no key has been inserted into a page since the split that created the page.

Without workload measurements, it is difficult to determine exactly how often each of these situations will occur in practice. If we assume that key values are drawn from a

uniform distribution, however, we can estimate the number of pages that are untouched at the time of a crash. To estimate the number of pages for which extra I/Os were required, we simulated the construction of 8,000 B-trees each with a randomly-selected size averaging 40,000 ± 500 random-valued 4-byte keys. These are two-level trees with about 128 pages. On average 0.05 pages were untouched since their last page split. Hence, the additional I/O was rarely required. Simulating 1,000 larger B-trees with 1,000,00 4-byte keys each, we found an average of 17 untouched pages that would have to be examined on recovery. However, one of these pages would only be encountered every 1,000 key insertions, so the extra recovery work would still have a limited effect on performance.

5.7 Summary

The POSTGRES DBMS relies on a no-overwrite storage system to avoid log processing during recovery. By avoiding log processing, POSTGRES recovers from failures quickly and eliminates a great deal of the complex recovery code found in most data managers. Unfortunately, concurrency requirements and inter-page pointers make the POSTGRES storage system techniques difficult to apply to index data structures such as B^{link}-trees.

In this chapter, we have presented two techniques for managing indices without using either write-ahead log processing or the usual no-overwrite techniques of the POSTGRES storage system. The first technique is based on shadow paging; the second on page reorganization during splits. Both algorithms use redundant information in index pages to detect inconsistencies caused by system failures as they are encountered. Inconsistencies are removed by repeating the interrupted page split or merge operations. The two techniques will also be useful in WAL-based data managers that want to avoid physical logging during page splits.

Measurements of a prototype implementation suggest that the algorithms will have little overall effect on data manager performance. Performance measurements show that key inserts and lookups will only be three to five percent slower when the tree is entirely in main memory. Estimates of the effect of the algorithm on tree height show that key lookups in shadow-page B^{link}-trees will read one more page from the disk than lookups in conventional B^{link}-trees under some workloads.

The height estimates and performance measurements also indicate that a hybrid between the two algorithms could reduce costs while preserving the best features of each algorithm. Using shadow paging near the leaf pages would eliminate the cost of page reorganization splits in the part of the tree in which splits are most common. Using page reorganization nearer the root would reduce space overhead caused by prevPtrs in internal pages and significantly increase fanout.

Chapter 6

Conclusions

The days when users simply accepted that computer systems could go down for hours or even minutes are rapidly drawing to a close. In the future, fault tolerance will no longer be a specialty service required only by military systems, hospitals, banks and stock exchanges. Trends in the prices of non-volatile RAM (NVRAM) and hardware reliability have reduced the costs of the hardware components of fault tolerant systems. The advances in operator interfaces and maintenance of fault tolerant systems will probably enter mainstream systems soon as well. This will lead to widely-available, reasonably-priced conventional systems that mask most hardware errors and power outages. The tools used to administer these systems will prevent many operator and maintenance errors.

However, in order for modern systems to remain reliable and available for long periods of time, they must run reliable system software. More careful software engineering will help some, but software will always be complex enough that software failures will occur. In the face of these failures, the fault tolerant system must be able to halt rather than produce incorrect results. Once halted, the system must recover quickly, hopefully without interrupting the users. Regeneration of lost program state must be fast, both to mask failures from users of the system and to eliminate the temptation for system designers to build complex, unreliable recovery systems.

This dissertation has examined the software fault tolerance problem from the standpoint of database management systems. It has addressed three problems faced by the designers of fault tolerant software. First, it presented and analyzed data from software errors uncovered in commercial systems in order to help characterize software errors. Second, it described and evaluated a technique for detecting addressing errors and controlling the error propagation that they cause. Finally, it extended the POSTGRES fast recovery feature to improve every day performance in high-update-rate environments and to handle fast recovery of communication state and index data structures.

Using data from commercial systems programs, Chapter Two assessed some of the root

causes of software outage. We proposed a model of errors based on different kinds of error propagation: control, addressing, and data errors. Studies of the MVS operating system, IMS database manager, and DB2 database manager showed that the distribution of these three kinds of errors was similar over the three systems. Control errors were about half of all errors, addressing errors 25–30%, data errors 10–15%, and the rest miscellaneous. Chapter Two showed that programs lost their point of control largely due to forgotten error conditions or unanticipated program events. Addressing errors often had to do with memory management, not necessarily with bad pointers. Addressing errors had higher than average customer impact, probably because error propagation made them difficult to diagnose and correct. The data presented in Chapter Two showed that most addressing errors were small and affected working data structures rather than data structures far away from the point of control. Finally, repeatable errors were relatively common. This fact combined with the difficulty of designing primary/backup communication protocols bodes ill for the most common redundancy-based software fault tolerance techniques.

Chapter Three proposed and evaluated several models of page guarding, a technique that uses conventional virtual memory hardware to limit propagation of addressing errors. The models differed in the manner in which the DBMS specified legitimate updates to the data, offering different protection/cost tradeoffs. An implementation on the DECstation 3100 showed seven to eleven percent impact for protecting the buffer pool in an update-intensive main-memory database, but only two to three percent impact for the same database when disk I/Os were considered. While Chapter Two indicated that the kinds of "wild pointer" errors that would be most easily detected by guarding were uncommon, these errors are among the hardest to find and fix using conventional debugging techniques. More important than their error detection ability, the guarding techniques help eliminate the set of errors that affect data cached in main memory differently than data written to disk. In the guarded version of POSTGRES, the primary reliability difference between data on disk and data cached in main memory is that the data structures used to manage the two resources are different, hence, are subject to different software errors.

Chapters Four and Five attacked the system availability problem by extending the POSTGRES DBMS fast recovery features in several ways. In the original POSTGRES storage system design, the DBMS was optimized for fast restart rather than fast commit in order to improve system availability. Chapter Four described enhancements to the POSTGRES storage system that reduce its cost in a high-update-rate environment. These enhancements include backward differencing and a new strategy for handling overflow pages that together make access to the current database fast even when the database contains a great deal of historical data. Performance analysis in Chapter Four suggests that with these enhancements, POSTGRES does the same amount of I/O as a conventional DBMS if (1) a sufficient amount of non-volatile RAM is available and (2) the log-structured file system

(LFS) is used, and (3) the POSTGRES historical data feature is disabled. If historical data is enabled, the analysis shows that POSTGRES does about seventeen percent more I/O. Chapter Four also showed how changes to the use of the transaction status file can eliminate all examination of this file during system restart. Because the database remains unavailable until clients are actually connected to the DBMS, Chapter Four added to POSTGRES techniques for quickly recovering communication between clients and the DBMS server. Chapter Five extended the POSTGRES storage system to handle index data structures without a write-ahead log. It described two index management techniques, one based on shadow pages and one based on page reorganization.

Overall, the POSTGRES fault tolerance strategy has been to anticipate technology shifts — faster processors, non-volatile RAM — and assume that new hardware can be used to mask the performance impact of simpler recovery strategies and additional error detection. Non-volatile RAM makes the POSTGRES storage system possible by softening the performance impact of force-at-commit buffer management. Faster processors mean that additional processing costs associated with guarding and POSTGRES on-demand database recovery will be little noticed by customers. Faster processors will also mean that using the same routines for recovery and for normal processing will have limited effects on performance. This has important reliability implications in, for example, the index management code, since the code used at recovery time is continually tested during normal processing instead of just at recovery time.

6.1 Future Work

6.1.1 Providing Availability for Long-Running Queries

The recovery model discussed in this dissertation considered the DBMS to be available if new transactions could be initiated against the data. It did not consider the cost of discarding work done by uncommitted transactions. In a high-update-rate, short-transaction environment, the current POSTGRES model works well. Forcing the clients to simply resubmit failed transactions is a worthwhile complexity/availability tradeoff.

When the DBMS is used for long-running complex queries, however, restarting the query after a failure may be unacceptable. Complex queries can run for minutes or hours, even in a high performance system. If the DBMS fails frequently relative to query execution time, users may not be able to make any progress on their work even though the database is "available" in that users can submit new queries at a moment's notice.

To provide high availability for long-running queries, POSTGRES would have to checkpoint intermediate state such as the current state of the query plan and temporary relations. Current commercial systems use savepoints to limit the rollback of long-running trans-

actions, but savepoints only record *updates* made by the long running transaction. The complex query checkpoint mechanism would record intermediate state of read-only transactions and record some DBMS data structures in addition to database changes. Such a mechanism would require a tunable parameter to set the frequency with which checkpoints are taken. An additional open question in the design of such a system is determining how to restore the two-phase locks associated with the query.

6.1.2 Fast Recovery in a Main Memory Database Manager

An important disadvantage of the POSTGRES Storage System is its reliance on a force-at-commit strategy for managing buffers. RAID, LFS, and NVRAM minimize this disadvantage, but still the cost of using magnetic disk as stable storage is a significant cost in today's systems. Obviously, database management systems designed to reside in main memory, rather than disk, would eliminate concerns related to force-at-commit [21]. POSTGRES can use NVRAM to lessen its commit costs, but it is still designed for a disk database. For example, care is taken that previous and current tuple versions reside on the same disk page to reduce the I/Os required during recovery and on index scans. As NVRAM prices approach those of conventional main memory, the idea of maintaining a main memory large enough to safely store an entire database becomes more and more practical.

Such a system could maintain high reliability and availability using variations on the page guarding and POSTGRES fast recovery techniques. The database itself would be organized probably as a single append-only log to facilitate page guarding; only the tail of the log would ever be unguarded. Indexing strategies might be changed since structures such as B-Trees were designed for speedy access to data on disk. The garbage collection strategies would be closer to those of the log-structured file system than to the ones described in this dissertation. The storage system would be unlike a conventional write-ahead log in that the log contains actual data values, not just undo/redo information for recovery. A fast main memory database management system would require some kind of checkpointing mechanism in order to provide media recovery.

6.1.3 Automatic Code and Error Check Generation

Much of the control error problem in IMS and DB2 had to do with programmers "missing a case" — not considering an error condition or timing condition that might arise. Software engineering tools that track where error conditions are handled would be helpful. This is especially true during program maintenance. The change team that repairs a software error discovered in the field may not always understand how the change affects the rest of the

program control flow. Regression testing alone does not seem to show whether all error conditions that were handled previously are still handled after a bug fix. In older programs such as IMS, a significant fraction of software errors come from program maintenance. Software engineering tools that helped show how small modifications to the code affect program control flow would be helpful.

DB2 had a small number of false error detections that occurred when the program changed, but the assert statements designed to detect bad internal state did not. Software engineers would help alleviate this problem by designing tools to (a) generate assert statements, or (b) flag assert statements that are affected when code is changed. Solution (a) requires less work for programmers, but, on the surface, seems more error prone. Programmers are supposed to think about assert statements. If assert statements are generated automatically, incorrect data structures can generate incorrect assert statements.

6.1.4 High Level Languages

Throughout this dissertation, we have assumed that the current generation of low-level systems languages will remain popular among system designers. While these languages will probably never go away, it is conceivable that fault tolerant system designers will switch over to languages with more debugging and anti-bugging features than the ones used to construct POSTGRES and the systems studied in Chapter Two. One important area of future work is to examine the error characteristics of languages such as C++ [23], Hermes [68], and Modula-3 [34] with higher degrees of type safety than current languages. Many of the addressing-related errors catalogued in Chapter Two involved errors in memory management, unsafe pointer operations, and errors in type coercion (union type problems) that these languages are designed to prevent. To our knowledge, no detailed error studies of systems programs written in these languages exist. It would be interesting to find out whether such languages have additional classes of errors not found in conventional programming languages.

The programming language Ada [37] has a built-in exception handling facility. We have seen that many errors in systems programs result from mishandled error conditions. Since many large Ada programs exist now, a study of error reports in this language – especially in users' exception handling code – would be interesting. Such a study would also be useful to designers of software engineering tools that help programmers write code to handle errors.

Bibliography

[1] Anon. et al. A Measure of Transaction Processing Power. Technical Report 85.1, Tandem Corporation, January 1985.

[2] A. Appel and K. Li. Virtual Memory Primitives for User Programs. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] M. Auslander, D. Larkin, and A. Scherr. Evolution of MVS. *IBM Journal of Research and Development*, 25(5), September 1981.

[4] A. Avizienis. The N-Version Approach to Fault Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12), December 1985.

[5] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[6] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. *Proceedings of the Summer USENIX Conference*, June 1992.

[7] J. Bannerjee, W. Kim, H. Kim, and H. Korth. Semantics and Implementation of Scheme Evolution in Object-Oriented Databases. *Proceedings of the SIGMOD Conference*, pages 311–322, December 1987.

[8] J. Bartlett. A NonStop Kernel. *Proceedings of the 8th Symposium on Operating System Principles*, 1981.

[9] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1), January 1984.

[10] R. Bayer and C. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[11] B. Bershad, T. Anderson, L. Lazowska, and H. Levy. Lightweight Remote Procedure Call. *Proceedings of the 12th Symposium on Operating System Principles*, pages 102–122, December 1987.

[12] A. Bhide, E. Elnozahy, and S. Morgan. Implicit Replication in a Network File Server. *IEEE Workshop on Management of Replicated Data*, November 1990.

[13] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking Database Systems, a Systematic Approach. *Proceedings of the Very Large Data Bases Conference*, November 1983.

[14] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[15] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, and E. Shekita. The Architecture of the Exodus Extensible DBMS. *Proceedings of the IEEE International Workhop on Object-Oriented Systems*, September 1986.

[16] X. Castillo and D. P. Siewiorek. Workload, Performance and Reliability of Digital Computing Systems. *Digest 11th International Symposium on Fault-Tolerant Computing*, 1981.

[17] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[18] R. Cheng. Virtual Address Cache in UNIX. *Proceedings of the Summer USENIX Conference*, 1987.

[19] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(4), 1979.

[20] D. Comer. *Internetworking with TCP/IP*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[21] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. *Proceedings of the SIGMOD Conference*, June 1984.

[22] B. Efron and R. Tibshirani. Bootstrap Methods for Standard Errors, Confidence Intervals, and otherMeasures of Statistical Accuracy. *Statistical Science*, 1(1):54–77, 1986.

[23] M. Ellis and B. Barnestroup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[24] A. Endres. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.

[25] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[26] R. Fagin, J. Nieverrgelt, N. Pippenger, and H. Strong. Extensible Hashing — A Fast Access Method for Dynamic Hashing. *ACM Transactions on Database Systems*, 4(3):315–334, September 1979.

[27] R. Glass. Persistent Software Errors. *IEEE Transactions on Software Engineering*, SE-7(3), March 1981.

[28] J. Gray. Why do computers fail and what can be done about it? *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[29] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[30] 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. *ACM Computing Surveys*, 13(2), June 1981.

[31] R. Gupta. A Fresh Look at Optimizing Array Bounds Checking. *Proceedings of ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[32] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the SIGMOD Conference*, pages 47–57, 1984.

[33] T. Haerder and A. Reuter. Principles of Transaction-Oriented Recovery. *ACM Computing Surveys*, 15(4), 1983.

[34] S. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[35] IBM. *MVS/Extended Architecture Overview*, publication number gc28-1348.

[36] IBM Corporation. *MS/VS Extended Recovery Facility (XRF): Technical Reference*, 1987.

[37] J. Ichbiah, J. Heliard, O. Roubine, J. Barnes, B. Krieg-Bruckner, and B. Wichmann. Preliminary Ada Reference Manual. *SIGPLAN Notices*, 14(6), June 1979.

[38] R. Iyer and D. Rossetti. Effect of System Workload on Operating System Reliability: A Study on IBM 3081. *IEEE Transactions on Software Engineering*, SE-11(12), December 1985.

[39] D. Jewett. Integrity-S2 – A Fault-tolerant UNIX Platform. *Digest 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[40] Gerry Kane. *R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[41] W. Kim. Highly Available Systems for Database Applications. *ACM Computing Surveys*, 16(1), March 1984.

[42] J. Knight, N. Levenson, and L. St.Jean. A Large Scale Experiment in N-Version Programming. *Digest 15th International Symposium on Fault-Tolerant Computing*, 1985.

[43] D. Knuth. The Errors of TeX. *Software: Practice & Experience*, 19(7), July 1989.

[44] C. Kolovson. *Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems*. PhD thesis, University of California, Berkeley, EECS Department, Computer Science Division, 1990. UCB/ERL TR M90/105.

[45] B. Lampson and D. Redell. Experiencs with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[46] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. *Proceedings Fall Joint Computer Conference*, pages 380–389, 1986.

[47] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, 6(4), December 1981.

[48] Y. Levendel. Defects and Reliability Analysis of Large Software Systems: Field Experience. *Digest 19th International Symposium on Fault-Tolerant Computing*, June 1989.

[49] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3), March 1982.

[50] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.

[51] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, March 1977.

[52] D. Menasces and O. Landes. Dynamic Crash Recovery of Balanced Trees. *Proceedings on Reliability in Distributed Software and Database Systems*, pages 131–137, July 1981.

[53] 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), March 1992.

[54] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write Ahead Logging. Technical Report RJ 6846, IBM, 1989.

[55] D. Morgan and D. Taylor. A Survey of Methods for Achieving Reliable Software. *IEEE Computer*, 10(2), February 1977.

[56] S. Mourad and D. Andrews. On the Reliability of the IBM MVS/XA Operating System. *IEEE Transactions on Software Engineering*, SE-13(10):1135–1139, October 1987.

[57] M. Olson. Extending the POSTGRES Database System to Manage Tertiary Storage. Master's thesis, University of California, Berkeley, EECS Department, Computer Science Division, May 1992.

[58] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[59] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the SIGMOD Conference*, June 1988.

[60] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[61] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *Proceedings of the 13th Symposium on Operating System Principles*, pages 1–15, October 1991.

[62] M. Schroeder and J. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[63] M. Seltzer. *File System Performance and Transaction Support.* PhD thesis, University of California, Berkeley, EECS Department, Computer Science Division, 1992.

[64] T. Shimeall and N. Leveson. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. *IEEE Transactions on Software Engineering*, SE-17(2), February 1991.

[65] V. Srinivasan and M. Carey. Performance of B-Tree Concurrency Control Algorithms. *Proceedings of the SIGMOD Conference*, pages 416–425, June 1991.

[66] M. Stonebraker. The POSTGRES Storage System. *Proceedings of the Very Large Data Bases Conference*, pages 289–300, September 1987.

[67] M. Stonebraker and L. Rowe. The Design of POSTGRES. *Proceedings of the SIGMOD Conference*, June 1986.

[68] R. Strom, D. Bacon, A. Goldberg, A. Lowry, D. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing.* Series in Innovative Technology. Prentice Hall, Inc., 1991. ISBN 0-13-389537-8.

[69] M. Sullivan. Software Errors Reported in 4.1 and 4.2 BSD UNIX. Unpublished notes from a survey of the BSD error report database, 1990.

[70] M. Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability — A Study of Field Failures in Operating Systems. *Digest 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[71] M. Sullivan and R. Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. *Digest 22nd International Symposium on Fault-Tolerant Computing*, July 1992.

[72] M. Sullivan and M. Olson. An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System. Technical Report M91-98, University of California, Berkeley, 1991.

[73] D. Taylor, D. Morgan, and J. Black. Redundancy in Data Structures: Improving Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-6(5), May 1980.

[74] T. Thayer, M. Lipow, and E. Nelson. *Software Reliability*. TRW and North-Holland Publishing Company, 1978.

[75] K. Tso and A. Avizienis. Community Error Recovery in N-Version Software: A Design Study with Experimentation. *Digest 17th International Symposium on Fault-Tolerant Computing*, 1987.

[76] P. Velardi and R. Iyer. A Study of Software Failures and Recovery in the MVS Operating System. *IEEE Transactions on Computers*, C-33(6):564–568, June 1984.

[77] S. Webber and J. Beirne. The Stratus Architecture. *Digest 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[78] W. Wulf. Reliable Hardware/Software Architecture. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[79] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[80] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. *Proceedings of the 11th Symposium on Operating System Principles*, pages 63–76, December 1987.