Multi-Site Declustering Strategies for Very High Database Service Availability

Øystein Torbjørnsen

Department of Computer Systems and Telematics
Faculty of Electrical Engineering and Computer Science
Norwegian Institute of Technology
University of Trondheim
Norway

January, 1995

Abstract

The thesis introduces the concept of multi-site declustering strategies with self repair for databases demanding very high service availability. Existing work on declustering strategies are centered around providing high performance and reliability inside a small geographical area (site). Applications demanding robustness against site failures like fire and power outages, can not use these methods. Such applications will often both replicate information inside one site and then replicate the site on another site and thus resulting in unnecessary high redundancy cost. Multi-site declustering provides robustness against site failures with only two replicas of data without compromising the performance and reliability.

Self repair is proposed for reducing the probability of double-failures causing data loss and reducing the need for rapid replacement of failed hardware. A prerequisite for multi-site declustering with self repair is fast, long-distance, communication networks like ATM.

The thesis shows how existing declustering strategies like Mirrored, Interleaved, Chained, and HypRa declustering can be used as multi-site declustering strategies. In addition a new strategy called Q-rot declustering is proposed. Compared with the others it gives larger flexibility with respect to repair strategy, number of sites, and usage pattern. To evaluate availability of systems using the methods a general evaluation model has been developed. Multi-site Chained declustering provides the best availability of the methods evaluated. Q-rot declustering has comparable availability but is significantly more flexible. The evaluation model provides insight and can be used to understand the declustering problem better and to develop new and improved multi-site declustering strategies. The model can also be used as a configuration tool by organizations wanting to deploy one of the declustering strategies.

Contents

Acknowledgements

I want to thank my advisor professor Kjell Bratbergsengen for giving me freedom in pursuing this work. Without being recruited to do research on parallel database systems I would not have been given the chance to participate in so many interesting research projects.

This work was initially started during a three years grant (ST.10.12.221192) from the Royal Norwegian Council for Scientific and Industrial Research (NTNF). In the years from 1992 to 1995 my employers Teleserve Transaction Technology A / S, SINTEF DELAB, and Telenor Research have kindly provided me the opportunity to finish my work. Without this help I could not have reached the goal. I want to thank Jet Propulsion Laboratory, Pasadena, USA, and Ed Upchurch for an interesting half year as guest researcher.

During the years I have worked on this thesis many people have helped me to reach my goal. My colleagues have always offered their help and provided me with a good environment for my work, although somewhat distracting. In particular I want to thank Petter Moe, Tore Sæter, and Svein-Olaf Hvasshovd for insightful discussions on parallel systems, databases, and fault-tolerance. Svein Erik Bratsberg has together with Tore and Svein-Olaf given valuable comments on the manuscript.

For distractions I want to thank Eirik and Nina for buns and tea, and Terje for “smalahove”. I also want to thank my parents waiting patiently for me to finish. Now my father has one less reason for calling.

Especially I want to thank all of you for never doubting that I would finish the thesis but instead encouraging me to continue.

Chapter 1

Motivation

Almost every home and office in the most developed countries have installed a telephone, and by lifting up the handset on his telephone the user is ready to operate the largest machine man ever have built. Since most of the current switching technology is based on computers we can also say that this is the biggest computer system ever made. From this perspective and based on the bad reputation computers have in the population it is strange that we trust the phone so much. In fact, most of the current businesses could not operate without telephones today and are making themselves yet more dependent on the telephone network. It has gone so far that we are carrying the phone with us wherever we go.

The phone system in some countries is now so reliable that when it fails, the users become first surprised, then helpless,[1] and major events are reported in the headlines of the news. A major outage in USA in 1991 caused an international airport to be closed down for hours during the outage.

The current telephone system mainly consists of digital switches interconnected with trunk lines. A signaling network, also connecting the switches together, is used to set up and shut down phone calls between two phones. In each switch there is a static table used to route a call from one input line to an output line. It is obvious that it is not possible to let this table contain all phone numbers in the country let alone the whole world. This would have required a huge number of entries and caused a tremendous update problem.

The solution has been to use physical addressing. The first digits in the phone number decides the switch the phone is attached to and is used to do the routing of a phone call. In each switch there is two tables: one for the phones connected directly to it and one for the reachable switches and their number prefix. Calls going to a phone connected to another switch are routed to this switch based of the first digits in the phone number. If the call is local it is connected to the line associated with the number. To further reduce the size requirements and the update problem, this scheme is used hierarchically.

This scheme is simple and robust. Each switch contains its own routing information and can connect to a target as long as none of the intermediate switches are down. By using alternative paths downtime can be further reduced. A user will also observe a system operating perfectly correct as long as he or she does not try to make a call that has to be

[1]The author recently lost the service for a week when a cable outside his home broke so he know how it feels.

Availability	$> 99.9996\%$
Response times	< 15 ms
Transaction rates	$> 1,000$ tps

Table 1.1: DBMS server requirements for SCP applications based on [HST$^+$91b].

routed through a failed switch.

Unfortunately there is some disadvantages with this scheme. The most notably is the inability to let the subscriber keep his phone number when he moves to another part of the country. Another problem is that there still is an update problem. Even though the tables are small and changes are seldomly needed, the changes must be applied to a large number of places each time a change occur and preferably atomically.

With the arrival of still faster, larger, and cheaper computer systems the telecom companies now see the opportunity to change this, both to their own and to the subscribers convenience. Instead of keeping static routing tables at each node, each time a call is set up the switches send a request to a computer connected to the signaling network. Upon receiving the request the computer, which is called a *Service Control Point* (SCP), translates the phone number into the physical routing the call should use and returns this to the requester. This mapping is even in use today but in a small scale.

In the telecom world this application is called *Universal Personal Telephone* (UPT). SCPs can be used for much more than UPT, for example charging and credit card verification. The common characteristics of these applications are that they are a part of the mission critical part of the telecom organizations and must provide an extremely high availability. While the existing network is able to operate when a few switches are down, a network based on a single SCP for establishing calls will be totally inoperable if the SCP fails.

As the reader already might have noticed the SCP providing the UPT functionality contains a database reflecting the current location of the subscribers. The application of a database management system (DBMS) therefore falls natural. The requirement to a DBMS that can be used in the operational part of the telephone network is radically different from normal business DBMSs. Table 1.1 lists three requirements given by the Norwegian Telecom. Note that these figures are for a complete system including software and hardware components. The response times and transaction rates given are not unreachable with current DBMS technology but the availability figure clearly is. The number given is equal to less that one hour accumulated outage over 30 years continuous operation! Commercially available fault-tolerant database servers claim an availability up to 99.99% which is 25 times worse than the requirement.

Telecom SCPs are not the only application for very high availability DBMSs. According to a Butler Cox report [Cox90] 25% of the companies asked say that they will not survive without computer service for more than one day. Relevant applications are medical information systems at hospitals, banking systems, airline reservation systems, manufacturing systems and point-of-sale (POS) systems. September 1993 a software failure in Oslo, Norway, almost caused the election to the Norwegian parliament (Stortinget) to be declared void and rescheduled later the fall with the consequential delays in the parliament's tasks.

This thesis will try to bring the state-of-the art closer to the availability requirement set by Norwegian Telecom. The work will both point out problems that must be solved, analyze alternative strategies and propose new ones. This is a theoretical work and there is not made any claims whether the goal is met or not, because there are too many uncertain parameters that only can be decided after the deployment of an actual implementation.

Chapter 2

Fault-tolerance terminology and technology

This chapter is a short introduction into the fault-tolerance terminology and technology which are required to read this thesis. The terminology is based on the work of Laprie [Lap85]. The chapter is ment as an introduction to the field for readers with a background in database systems. For a full introduction into the field of fault-tolerant computing the reader is directed to a textbook on the topic.

The chapter starts with a distinction between the terms faults, errors and failures. Measurements of service quality and the nature of faults are then discussed. The chapter ends with listing some methods that can be used to improve the availability of database servers.

2.1 Faults, errors and failures

A *module* is an entity with an external interface and a specified behavior. A module can for example be an integrated circuit (IC), a printed circuit board (PCB), a computer, a subroutine, a program, a printer, or a complete computer system with software and hardware. One module can be composed of several submodules, some possibly identical.

Unfortunately, sometimes a module does not behave according to the specifications. A *failure* is the event that a module does not meet its specification. An *error* is the internal state of a module that causes a failure. A *fault* is the event that causes the error. Example: a hard disk drive is a module. The disk drive is exposed to a mechanical shock which causes a disk block to become corrupted. The shock is the *fault*, the corrupted disk block the *error*, and when the computer later tries to read the block and the drive flags an error the computer observes a *failure*. Note that some errors do not need to cause a failure, e.g. if the disk block content never is read but instead overwritten with new data removing the corruption.

The user of a module should not only be satisfied with the specification of the module but should also know some details about how the module fails, how often it fails, etc. This information is covered by the module's *dependability*.

2.2 Dependability

Laprie [Lap85] defines dependability as:

> "Computer system **dependability** is the *quality of the delivered service* such that *reliance* can *justifiably* be placed on this service"

In general dependability is the collective term used to express the quality of all aspects of how much the user[1] can trust (or depend on) a system (or a module). Since there is many aspects which might be of interest it is of no practical use to quantify dependability into a single measure. Some proposed measures for dependability are:

Reliability: The probability that the module will meet its specification.

Availability: The fraction of time the module operates correctly.

Maintainability: The time it takes to bring the module into operation after a failure.

Safety: The ability of the module to fail without catastrophic consequences.

Security: The ability for the module to avoid intended misuse.

Performability: A measure of performance fulfillment.

Below, these aspects will be described more in depth. As we will see there is a partial overlap between the measures.

2.2.1 Reliability

Laprie [Lap85] defines *reliability* as:

> "a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant."

Service accomplishment is the system state where the system provides the specified service, while *service interruption* is the state where the delivered service is different from the specified service.

Mean Time To Failure (MTTF) is the reliability measure most often used. This is the average time from a module is set into service until it fails. An alternative measure is the probability of continuously service accomplishment over a given time interval.

[1]A *user* is not necessarily a human. A computer can be considered a user of a printer when it uses the services offered by the printer.

2.2.2 Maintainability

Laprie [Lap85] defines *maintainability* as:

> "a measure of the continuous service interruption, or equivalently, of the time to restoration."

The measure for maintainability is typically *Mean Time To Repair* (MTTR).

2.2.3 Availability

Laprie [Lap85] defines *availability* as:

> "a measure of the service accomplishment *with respect to the alternation* of accomplishment and interruption."

Gray and Reuter in [GR92] gives a less rigorous definition for *availability* as:

> "the fraction of time that a system performs requests correctly and within specified time constraints"

Based on this definition availability for a module can directly be computed from the reliability and maintainability measures for the module. Availability A is given by:

$$A = \frac{MTTF}{MTTF + MTTR}$$

Since A can be computed from the reliability and maintainability measures it is a redundant measure. On the other hand it represents a single measure incorporating the essential information from the other two and reflects the essential properties of a module.

Similarly, *unavailability* U is defined as the fraction of time the system does not provide its specified service. Thus

$$U = 1 - A = \frac{MTTR}{MTTF + MTTR}$$

Since availability for highly available systems will be close to 100% it can be more convenient to use the unavailability figure. An unavailability of 10^{-6} is much easier to understand than an availability of 0.999999 (99.9999%).

The definition for availability does not consider that the usage intensity might vary over time, e.g. that there is a higher request intensity during day than during night. An outage during a busy period would have worse effects that during hours with a low load. It should therefore be useful to have an alternative definition that reflects usage frequency, like:

System type	Unavailability (min / year) U	Availability (in percent) A	Availability class C
Unmanaged	50,000	90	1
Managed	5,000	99	2
Well-managed	500	99.9	3
Fault-tolerant	50	99.99	4
High-availability	5	99.999	5
Very-high-availability	0.5	99.9999	6
Ultra-availability	0.05	99.99999	7

Table 2.1: Classification of systems based on availability.

> "the fraction of requests that a system performs correctly and within specified time constraints"

Throughout the rest of this document the first definition will be used for availability and the term *success rate* for the last one.

Gray and Siewiorek [GS91] groups systems into seven availability classes, see Table 2.1. If A is availability the availability class C can be expressed as:

$$C = \lfloor \log_{10} \frac{1}{1 - A} \rfloor$$

Managed computers are typically class 2 and commercially available fault-tolerant computers are class 4. The requirement goal set in Chapter 1 equates to class 6 which is the same as delivered by telecom switches.

2.2.4 Safety

Catastrophic failures are events that can not be justified with the advantages the system gives (the cost is larger that the earnings). Example: engine trouble in an air-plane should not cause the air-plane to crash but instead allow the pilot to perform an emergency landing. Often catastrophes are associated with health threatening events but also significant money losses and loss of confidence qualify as catastrophic events. The common factor is that the consequences threaten the existence of the user either it is the air-plane passenger, a bank, or a software company.

Safety is by [Lap85] defined as:

> "a measure of continuous safeness, or equivalently, a measure of the time to catastrophic failure."

The quantitative measure for safety will therefore be *Mean Time To Catastrophic Failure* (MTTCF).

Based on this discussion it is clear that some failures might be catastrophic for one user but not for a second. A user not taking backups of his or her hard disk is likely to call a disk crash a catastrophe while one taking backups will not. The fact that the second user takes backups actually means that he expects disk crashes while the first user is accepting the higher probability for a catastrophe.

A module which fails gracefully avoiding catastrophic failures is called *fail safe* or *fail soft*.

2.2.5 Security

Security is divided into three terms, namely *confidentiality*, *integrity*, and *availability*. [Org89] defines confidentiality as “the property that information is not made available or disclosed to unauthorized individuals, entities, or processes”, integrity as “the property that data has not been altered or destroyed in an unauthorized manner”, and availability as “the property of being accessible and usable upon demand by an authorized entity”. In short it can be said that security ensures authorized use and prevents unintentional use. The reason to include availability into the security definition is to avoid that unauthorized entities can block authorized accesses.

There is no simple measure like a single number for system security. Department of Defense in USA has in [Dep85] described a means to evaluate the security level of computer systems and a classification scheme using the classes “A1”, “A2”, “B1”, “B2”, “B3”, “C1”, and “C2”.

In this work we will not discuss security any further

2.2.6 Performability

Performability can be defined as a measure of ability to provide a specified capacity. It reflects the fact that a module’s capacity may vary over time as submodules fail and are repaired. Even if a module gives correct results on the incoming requests, the results can come too late and not be of any use. See for example [Mey80, STR88] for treatments on performability.

One measure for performability is the probability of servicing a specified amount of work over a given time interval.

2.3 The nature of faults

2.3.1 Classification of faults

There are several ways of classifying faults, both based on cause and on the nature of the fault. Gray [Gra85] gives a classification of the causes for system outages (service unavailability). Based on Gray’s classification the following categories are suggested.

Hardware: Failures caused by hardware components.

Software: Failures caused by software.

Operations: Failures caused by the user operating the system. Gray's classification also included failures caused by system administration into this class but here it will be considered to be a part of the maintenance class.

Maintenance: Failures directly caused by repair and system maintenance, both for hardware and software, e.g. replacement of an operating module instead of a failed one causing the system to crash.

Environments: Failures caused by the environment, e.g. power outage, communication failure, sabotage, flood, fire.

Process: Outages caused by other reasons, e.g. strike and administration decisions.

A *disaster* is an event that makes a whole or major parts of a computer site unavailable, e.g. flood, earthquake, fire.

A fault is either soft or hard. A *soft fault* is a fault that apparently disappears after the failure and no repair is needed. A *hard fault* on the other hand does not disappear and the module must be repaired before becoming operational again. Soft faults are also called *intermittent* or *transient*.

Most modules are manufactured in multiple copies and some faults will typically occur on all these copies. These faults, which is called *design faults*, can be caused by mistakes in both the specification, design, and assembly / implementation of the module(s). *Physical faults* on the other hand affects only a single instance of the module and replacing with an identical new module will repair the problem. Even though the design fault is latent in every unit, it is not necessary that all the users will encounter the fault.

Physical faults and design faults differ greatly on how they are repaired. The physical fault is repaired by replacing the single failed unit. Design faults however are repaired through two steps. First, the fault is fixed so that all units manufactured after this date will not exhibit this fault. This obviously does not mean that the fault automaticly will be removed from modules manufactured before the fix. To fully repair a design fault, all existing units must be fixed (through bug fixes), also for those that have not encountered the failure but potentially will (preventive repair). A corrective strategy can also be chosen that allows one to delay the fix until the fault is encountered the first time or is causing serious problems for the user.

Since fixing a design fault means that the module has to be changed, the repair can result in introducing new design faults. Adams [Ada84] therefore warns against uncritical preventive repair.

Software and hardware faults differs in several aspects as we will see below.

2.3.2 Hardware faults

The failure rate of a hardware module typically varies over the lifetime as shown by Figure 2.1. In the infancy period the failure rate is high due to manufacturing defects. After a short time the module becomes mature and the failure rate stabilizes on a low level. When the module approaches the end of the lifetime the module is worn-out and the failure

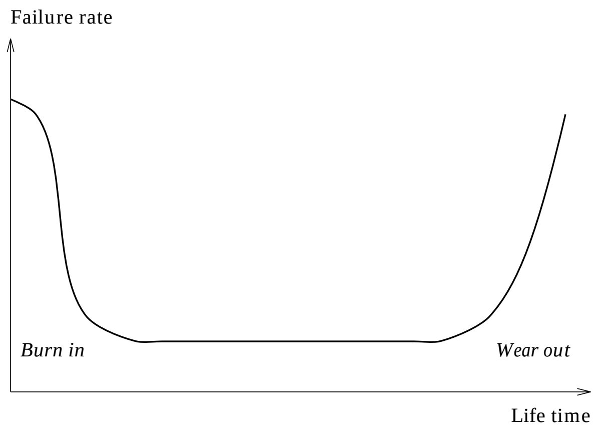


Figure 2.1: The bathtub curve. The failure rate depends on the module life time.

rate increases again. Hardware reliability can be significantly improved by removing the infancy and worn-out periods. The infancy period is removed by running the hardware through a burn-in in the factory before the customer receives it. Replacing the module when it becomes close to be worn out removes the second problem.

MTTF for a hardware module usually means operating time. Unfortunately, many vendors use MTTF to specify failure rate in the bottom of the bathtub curve and separately gives a *lifetime* for the module. See for example the data sheet for the Fujitsu M2622SA, M2623SA and M2624SA[2] hard disk drives which claims a MTBF[3] of 200,000 power-on hours (22.8 years) but a component life of 5 years [Fuj91].

According to [GR92] the ratio between hard and soft faults for printed circuit boards (PCBs) varies between 1:1 and 1:10 with an average of 1:5.

The operation environment has much influence on the hardware reliability. A rule of thumb is that a temperature reduction of 10 °C can increase the lifetime with a factor two [ES92]. Shocks, vibrations, and thermal changes causes mechanical stress on the hardware and reduces the lifetime and can directly cause a failure.

Hardware has not only increased in speed and capacity and reduced the cost over the time, but also the reliability has been significantly improved. This reliability improvement can mostly be contributed to improved design, a higher integration factor, and a lower component count [Gra90]. Also power consumption is reduced which gives lower temperatures and therefore longer lifetimes. Disk drive makers are claiming observed MTBFs of 500,000 hours [GR92] and Motorola sells processor boards with a 1,000,000 hours MTBF (114 years) [Mot91]. These figures are based on the failure rate of a high population of modules operating under ideal conditions (i.e. no accelerated testing).

[2]This way of using MTTF is not unique for Fujitsu, but are also used by many other manufacturers of hard disks.

[3]MTBF=Mean Time Between Failures. Valid for components that can be repaired. In this work we will consider MTTF and MTBF to be the same.

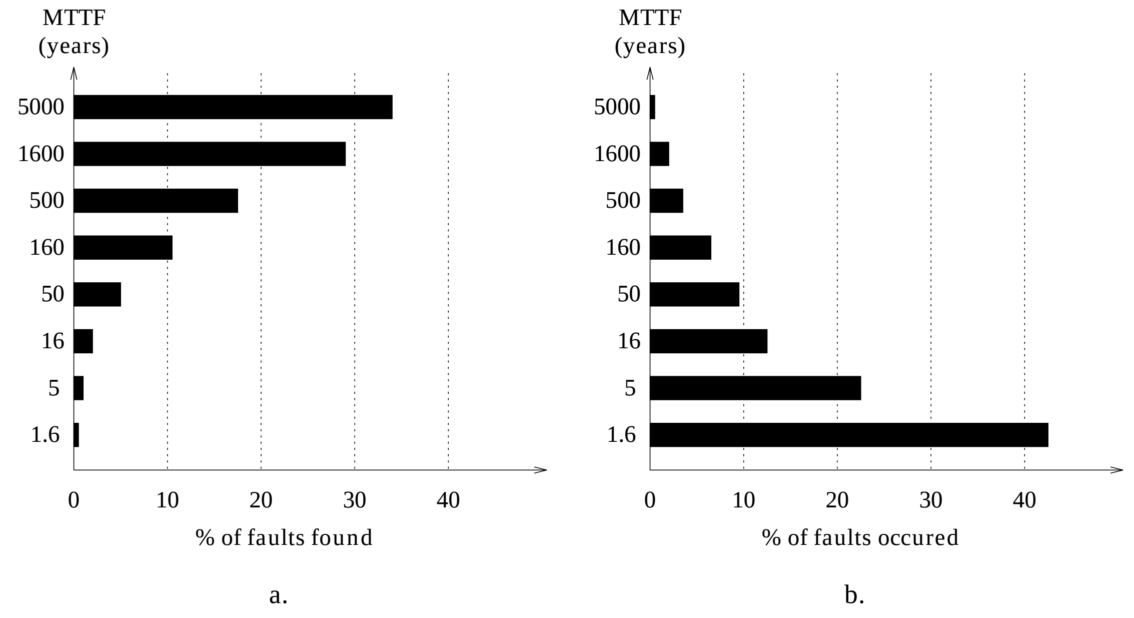


Figure 2.2: Software failure rates for an IBM product. The graphs show the distribution of faults found (a) and occurred (b) as a function of MTTF. Data based on [Ada84].

2.3.3 Software faults

Software faults differs significantly from hardware faults. Software is not exposed to physical faults, only design faults. This means that once fixed, the fault is removed forever. So, if the software is perfect and there are no faults in it, it will never fail! Unfortunately things are not so easy.

Virulent and benign faults

Adams presented in [Ada84] a study from IBM based on statistics on the failure rate of their system software products. The study showed that most failures only occurred a few times (Figure 2.2-a) and only a few bugs caused most of the failures (Figure 2.2-b). The figures show that 4% of the faults found contributed to 78% of faults occurred. (Even though this is only for a single product, the other products presented in the same work have a similar distribution.) Adams called this class of faults causing most of the failures for *virulent design errors*. The defects not being virulent are called *benign*.

The figure also shows that most faults have so long MTTF that each user will probably see only a few of them. Repairing all faults should therefore not be necessary. The study also showed that correcting a fault often resulted in new unknown faults.

Bohrbugs and Heisenbugs

Gray in [Gra85] classify software faults as either *Bohrbugs* or *Heisenbugs* which correspond to respectively hard and soft faults. Bohrbugs manifest themselves each time the code is executed while heisenbugs only on some occasions tend to cause an error. Bohrbugs are easy to single out and correct and are mostly removed during testing before the software leaves the manufacturer. Heisenbugs on the other hand are nondeterministic. They show up only under special conditions which are created by usage pattern, the environment, other software, and internal state. It should be noted that the latent bug is in the software itself, not directly caused by the environment.

An experiment done at Tandem showed that one out of 132 software faults in a production software product (a spooler) was a bohrbug [Gra85]. It is dangerous to make a conclusion on this single figure alone but it confirms the general experience that the majority of software faults in production code are heisenbugs.

One nice characteristic of a heisenbug is that it apparently has disappeared if the code is executed again a second time. The bug has not been removed but occurs only under very special conditions. This means that the user does not need to get his software fixed each time he or she encounters a failure but instead just reruns it. It can take days to get a new version of the software but only fractions of seconds to rerun it. Lee and Iyer [LI93] show that data collected by Tandem indicates that more than 80% of all failures are masked by process pairs. Due to underreporting of non-fatal faults this number can be as high as 96%.

The trend of improving reliability of hardware has not been followed up by software technology and a larger fraction of failures is caused by software [Gra90]. The reason for this is that software has grown more and more complex over the years. The improvement in the software development process has been consumed by this increase in complexity resulting in no overall improvement in software reliability.

2.4 Techniques for high availability

There are several ways to attack the problem of achieving high system availability. Fault-tolerance falls immediately into mind, but might not alone be sufficient. It can be possible to achieve the desired availability both faster and for less cost by using other methods. Faults are only one source for service interruption but also maintenance operations can cause downtime. This section devotes to methods for achieving high availability.

2.4.1 Fault-tolerance

Fault-tolerance is based on the fact that faults exist and that it is possible for the computer system to handle them without external intervention. By *masking* a fault, the fault does not cause a failure and the module does not become unavailable, i.e. causing service interruption. It should be noted that fault-tolerance is not a goal in itself, but rather a mean to achieve better dependability with respect to both availability and reliability.

Redundancy is the keyword to fault-tolerance, either as time or space redundancy. *Space redundancy* means that a redundant number of modules are used to process a task. On the other hand, when the same module does redundant work on a task by checking the result (e.g. by processing it multiple times), it is called *time redundancy*.

The redundancy can be used to both detect, mask, and recover from faults. Let two or more modules with the same specification perform the same task and compare the results. If the results differ, at least one of the modules has failed and the failure can be detected. By taking a majority vote (using three or more modules) it is also possible to mask the failure. The modules need not be identical (rather contrary) but should have the same external specification. Such different modules can be used to detect design faults. *Design diversity* [Ran75, Avi85] is a technique used to achieve this.

The redundancy does not have to have identical functionality as the module to be checked. A single *parity* bit can check for single bit errors in a 32 bit word and by increasing it to seven checksum bits, single bit errors can be corrected and double bit errors detected.

When an error has been detected and eventually masked, the failed module has to recover into a consistent state. In case of a transient failure, the failed module should be brought back into service either through recovery of its internal state or through a complete restart. If it is a hard fault, the component should be taken out of service and isolated from the rest of the system until it is repaired.

Even though a module is fault-tolerant, it can not handle all kinds of errors. The module is only prepared to handle a known set of errors and if an error is outside this set, the module will fail. Likewise, the fault-tolerance mechanism can itself fail and cause a failure.

2.4.2 Fault avoidance

It is said about energy, that "the best energy source is the energy that you save". This also applies to fault-tolerance. It is better to avoid faults at all instead of having to handle them. After they have occurred it is difficult both to detect and correct them.

High reliability: Through design, implementation, and testing the module should be built with reliability in mind. Quality assurance, highly skilled designers and programmers, and a systematic testing approach are keywords here. The modules should be designed with a high tolerance with respect to the environment (e.g. temperature, mechanical shock, power spikes, input) and high resilience.

Simplicity: Simpler designs contain less faults than complex ones. They contain less submodules that can fail and are easier to understand, build and test. Reducing the number of lines of code in a program will also reduce the number of bugs!

Burn in: A burn in increases the availability of a module since it removes the infancy problems of the module and "Monday products" are eliminated. Testing can be considered burn in for software.

Automatic maintenance: Human mistakes can not be avoided. Automatic maintenance procedures (e.g. automatic backup) can remove many of the faults human mistakes can cause.

"Friendly" user interfaces: All maintenance can not be done automatic. Making easy to use user interfaces reduces the number of human mistakes. Make them self-explaining, avoid cryptic codes, and validate input data.

Environment independence: The module should avoid relying too much on external entities like cooling, power, and maintenance. It is obviously impossible to avoid using these completely but the module should tolerate that they fail, e.g. by using *uninterruptible power supplies* to handle power outages.

Security: Malign attacks on a module can cause a failure and therefore reduce the availability.

Resource control: A module has a limited performance and is specified to handle a given load. Overloading can cause the system to reject all requests and therefore threatens the availability. The module should handle overflows gracefully by rejecting requests, either by priority or by random. Alternatively, the module can be dimensioned with a sufficient over-capacity to handle the peeks. This strategy is also attractive since a module operating close to its performance limit is more exposed to failures than a module with a normal load (greater chance for resource queuing, conflicts, and deadlocks).

Restarts: Over time, the chaos in a running program increases, e.g. heaps become fragmented, counters grow towards overflow, and the probability for a latent error increases. These situations are seldom encountered during testing. Regular preventive restarts could therefore reduce the number of failures.

2.4.3 Maintenance

The value of fault-tolerance is limited without repair. Since a hard fault should cause in isolating the faulty component, there is a reduction in the level of redundancy. The level of redundancy is limited and after some failures the redundancy is lost and failures can not be masked any more. Often a system is single fault-tolerant, i.e. only one faulty component can be handled. To avoid a second fault causing service interruption, faulty modules should be repaired or replaced.

Also non-fault-tolerant systems must be repaired. The system will be unavailable from the failure until it has been repaired. It is therefore important to do the repair as soon as possible.

Low MTTR: By reducing the *Mean Time To Repair* (MTTR) the probability for a double failure is reduced. Double failures will result in outage time and therefore reduced availability. This number is given by both the maintenance organizations ability to repair faults (see Section 2.4.6) and the systems maintainability.

Planned maintenance: Repair is done only at predefined time intervals. All failed modules are repaired. If the service intervals are long, the MTTR will be long.

Corrective maintenance: Repair is done as soon as possible after a failure has occurred. This reduces the MTTR compared to planned maintenance.

Preventive maintenance: Some faults can be avoided by replacing modules before they fail, e.g. by installing bug fixes before the bugs are encountered or replacing the color band of a printer before it is worn out. To be able to do preventive maintenance there must be means to find out that a component is close to the end of its lifetime, e.g. lifetime predictions or testing. Preventive maintenance is no alternative to repair.

2.4.4 On-Line maintenance

Traditionally computer systems have to be taken off-line for maintenance. This is undesirable for users that want 24 hours a day, 7 days a week service time, i.e. continuous available service. For these users the system is unavailable during off-line maintenance and if all or some of the maintenance can be done while the system is running, the availability will be improved.

On-line maintenance can be compared with doing maintenance on a car when it is in use. Clearly a car is not designed for this and several technical demanding mechanisms have to be added to achieve this.

Some relevant maintenance operations that might be done on-line are:

Diagnostics: Test of system components and detection of failed or marginal modules. Problems: how to test without affecting user tasks and how to locate the failure to the correct module.

Repair: Replacement of failed hardware modules. Problems: electrical and mechanical.

SW change: Upgrade or installation of new software. Problems: when should the new software take over for the old and how to handle incompatible data formats between old and new version.

System scaling: Increase or reduction of system capacity, e.g. transaction rates, number of users, data volume, and response delay. Problems: must be a scalable architecture, integration of new modules.

System reorganization Change of organization of data and applications, e.g. creation of database indices. Problems: enable access to data that is reorganized during operation.

Image copy Making a copy of the consistent system state (virtually a snapshot) for backup, history, or transfer purposes. Problems: how to create a consistent image.

2.4.5 Environment precautions

The environment is also a source for faults. Availability can be improved by being prepared to handle them.

Security: Physical security precautions hinder unauthorized personnel access to the system reducing the sabotage threat.

Air conditioning: Stable and cool room temperature reduces the failure rate and prolongs the lifetime of electronic components.

Power supply: The power supplied by electricity utilities has in addition to noticeable service interruptions also small irregularities like power spikes and bad sine waveforms. These are often not noticeable for humans but can cause problems for computer equipment. Power conditioners and uninterruptible power supplies (UPS) overcome most of these problems.

Multiple sites: A natural disaster can knock out a whole computer site (e.g. fire or earthquake). A second site located sufficiently far away will probably not be affected by the disaster and can take over the tasks of the destroyed site.

2.4.6 Maintenance organization

Maintenance organization performance (MOP) is a term expressing the quality of the organization doing maintenance on a system. MOP will have influence on the availability of the system. The following aspects of the maintenance organization should be considered:

Change management: All changes should be analyzed, planned and tested extensively before employed to the production system. Changes are a major source for problems. "If it ain't broken, don't fix it".

Trained personnel: Trained maintenance personnel is less likely to do mistakes resulting in failures during maintenance operations.

24 hour shift: If maintenance personnel is available for corrective maintenance both day and night, the MTTR is significantly reduced.

Spares availability: Repair of failed components must be postponed until spares are available. An on-site spare stock can reduce MTTR.

2.4.7 Disaster recovery

Even though it is possible to reduce the chances of a disaster knocking out a site, it can not completely be avoided. In the worst case, a disaster will be a catastrophe threatening the existence of the users. The owners should therefore be prepared on disasters and ready to recover the system in case of such an event. A short recovery-time reduces the unavailability and therefore improves the availability. The use of a second standby site will avoid most of the unavailability (only the time to switch systems), but the system will now be single fault-tolerant until the crashed site has been recovered. A fast recovery will reduce the chance of a double-fault. The following aspects should be considered:

Disaster plans: It should exist detailed plans on what to do in the case of a disaster. Thinking on what to do should not be delayed until it has happened.

Off-Line backups: Take backups regularly and keep an continuous incremental log. These should be kept on a place that not will be affected by the same disaster as the system, e.g. in a safe and / or on another location.

Standby system: Keep a system ready to take over the work of the failed system. This can for example be done through an agreements with another company in the same situation allowing you to run your software on each others systems in case of a disaster.

Chapter 3

DBMS servers and dependability

This chapter will discuss database technology seen from the viewpoint of highly dependable systems. Availability, safety and performability are the measures for dependability that will be focused upon.

3.1 DBMS dependability

C.J. Date in [Dat86] defines a *database system* as:

> "... a system whose overall purpose is to maintain information and to make it available on demand."

and later continues:

> "... a database system involves four major components: data, hardware, software, and users."

The term "users" includes both end-users, application programmers, and the database administrator (DBA). "Hardware" is all hardware components used from power supplies, disk drives, CPU, communication lines, and the user terminal. "Software" includes the *database management system* (DBMS), operating system, compilers, communication software, and application code.

This definition includes several aspects which are important seen from a dependability viewpoint.

- The phrase "maintain information" states that the information has to be kept in a correct state and protected against unintended changes. This statement ensures the correctness of the information.

- The phrase "make it available on demand" says that there is an availability requirement on the database system.

- By including the users into the definition, the dependability of a database system also includes user actions. A database system should therefore be robust against user mistakes.
- Since the application is a part of the database system, the dependability of the database system depends on the quality of the application code.

Based on Date's definition, database systems are a huge field to work in. This work has no ambitions to provide a solution to all the aspects of dependability for database systems but instead focus on the dependability aspects relating to the server part of a *client-server* database system architecture. This includes the DBMS software and the server operating system, communication software, hardware, and environment, but excludes the quality of the application software, the end-user actions, and all aspects of client computers software, hardware and communication.

The rest of this chapter will discuss client-server architectures and state-of-the-art database management system technology in general with a focus on dependability. Chapter 4 will look on hardware and operating system technology and describe an architecture that will be the basis for the rest of this work.

Going back to the definition for dependability given in section 2.2 on page 6, we see that "quality of the delivered service" is an essential component of the definition. Several of the measures given for dependability also incorporate the concept "service".

3.2 Transaction processing and complex query processing

On-line database processing can mainly be grouped into two classes *on-line transaction processing* (OLTP) and *on-line complex query processing* (OLCP). In general we can say that OLTP is database processing using *small* transactions and OLCP is using large transactions. Typically an OLTP transaction accesses few tuples in the database (<10) using random access while a complex query accesses many tuples (>100) without knowing the exact key value of the tuples. OLCP is often used for interactive batch analysis and sequential search for data in a database. The transactions are mostly reading the data and using set operation like join and union. OLTP transactions execute single tuple operations which often update the tuples. There is no clear border between OLTP and OLCP and there might be problems with classifying some queries. On the other hand, other queries are quite distinguishable. Table 3.1 lists the most typical differences.

OLTP systems and OLCP systems have very diverging requirements and design goals. OLTP systems focus on low request latency, predictable response times, and a high level of concurrency. OLCPs on the other hand is more concerned about high efficiency and high data throughput. This is confirmed by the fact that there are vendors specializing in these areas. A good example is Teradata [Ter85] which makes the DBC / 1012 database machine optimized for OLCP but with poor OLTP performance.

OLTP transactions are characterized by response times less than 10 seconds and therefore do not allow user interaction during the execution of the transaction. Complex queries on the other hand live for a long time and allow the user to interactively issue commands to the DBMS, e.g. through an interactive query language like SQL.

	OLTP	OLCP
Number of accessed tuples	few	many
Execution time	short	long
Time constraints	yes	no
Access pattern	random	sequential
Operation type	tuple	set
Access type	read / write	read only

Table 3.1: Characteristics of OLTP and OLCP transactions.

It is desirable to run both OLTP and OLCP concurrently on the same database with normal transaction ACID properties (atomicity, consistency, isolation, and durability) maintained. For current systems this involve a large threat against the availability for OLTP services. A conventional two-phase-locking strategy [BHG87] will block small OLTP transactions conflicting with complex queries that might live for minutes and hours. This violate response time constraints defined for the OLTP transactions and therefore result in an intermediate service interruption seen from issuers of the OLTP transactions.

The consequence of this conflict has been that large users employ two systems, one for OLTP and one for OLCP. The OLTP system maintains the current up-to-date data. The OLCP system is at regular intervals (e.g. each night) reloaded with a up-to-date dataset. Obviously this creates new problems since the OLCP does not have access to the most up-to-date data. One solution to this problem is a multi-version concurrency control strategy [BHG87] which ensures that read operations do not block update operations. Another solution proposed in [HST$^+$91b] is to provide a second replica of the data. When an OLCP transaction is issued one of the replicas are frozen in a transaction consistent state. The OLCP transaction accesses the frozen replica while OLTP transactions uses the other. When the OLCP transaction commits it checks for eventual conflicts which will cause it to abort and roll back. The updates done by committed OLTP transactions are then applied on the frozen replica. Note that a OLCP transaction only reading data can not result in conflicts and will therefore never roll back.

Many database maintenance operations have the same distinguished features as OLCP transactions. Inserting or deleting columns in a table is an example on a maintenance operation which will take a long time if the table is large. Most systems will not allow transactions to access the table during reorganization.

3.3 Client-server technology

Client-server technology is a hot topic in both the commercial world and as a research topic. The basis for this technology is two main components: servers and clients. The clients are

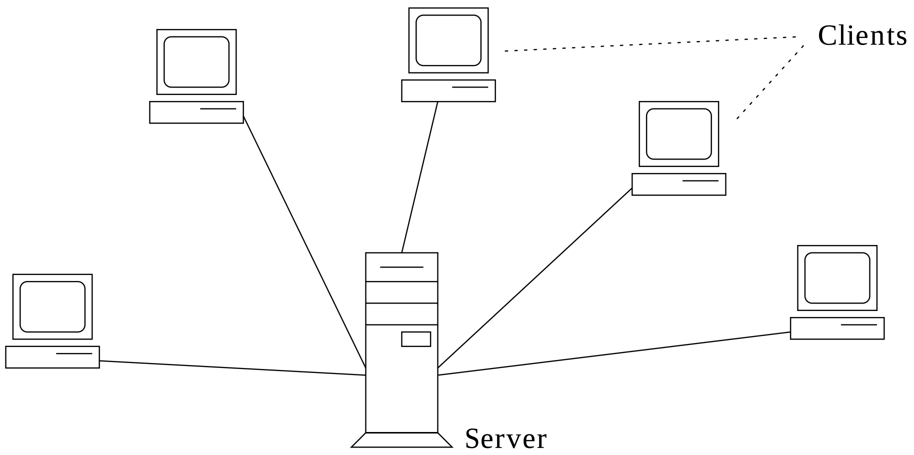


Figure 3.1: A client-server configuration.

connected to servers with communication lines. Figure 3.1 shows a simple (but typical) configuration with one server and multiple clients.

A server provides a predefined set of services to the clients. More complex configurations are also possible with both multiple servers and multiple clients connected to the same network. A client using a service provided by one server can also itself be a server for other clients. It is even possible for a server recursively to use itself as a server.

The term *server* or *client* should not be associated with a single computer but rather with software systems running on some hardware units. Since a single computer can both run multiple servers and clients, it should be possible to view these as separate components. It is also possible for services to migrate dynamically between different computers depending on load and availability.

There are several reasons to choose a client-server architecture: Specialization, organizational, mobility, heterogeneity, security, and not at least cost-efficiency.

Specialization: Some tasks require special hardware or software, e.g. a printer or a modem. A server can control the access to those units.

Organizational: The organizational structure of the users of the system can motivate the client-server architecture. By placing all user files on a central location, backups can be taken by a single person instead of spreading the responsibility to many people. Another example is that an organization providing the service to search for information in a database to users outside the organization but still maintain the ownership of the database.

Mobility: Some services are impossible or too expensive to implement with mobile computer equipment. By using a mobile client connecting to a central server (e.g. through telephone lines and a modem) the user can access a non-portable resource. The user might also often change client but need to access the same resource (e.g. his own files).

Heterogeneity: A fixed predefined protocol for client-server communication enables clients of various architectures to be used, both with respect to hardware and software. This protects investments in equipment and do not lock the organization to a single supplier.

Security: It is easier to protect a single site against physical attacks done by none-authorized persons.

Cost-efficiency: Instead of giving each user their own resource it might be more economical to let them share one or a few central resources. A Cray supercomputer is a *compute server* which a single user can neither afford nor utilize fully himself. The resource should instead be shared with many others. There are also cases where one big resource is more cost efficient than many small ones.

Implementors using client-server technology face the problem of how to split applications into a server part and a client part. Below follows a list of guidelines for client-server design.

1. The server functionality should be general and independent of application, i.e. standardized.
2. The client-server communication protocols should adhere to international standards, both on the transport level and the application level and support heterogeneous clients.
3. Communication should be insensitive to delays. The data volume sent should be low.
4. As much as possible of the processing should be done on the client side to achieve high server throughput and low latency.
5. Easy to use and / or program.

In general, some of these points are incompatible and compromises must be made.

3.3.1 Server semantics

In [Cri85, Cri90] Cristian divides the system specification of a server into three parts, the *standard semantics*, the *failure semantics*, and the *stochastic behavior*. The standard semantics is the functional requirements of the system, i.e. how the system is intended to behave. The failure semantics is the specification of how the server is expected to fail (if it ever fails) and which the users of the service must be prepeared to handle. If the server has a failure outside the failure semantics it is considered to be a *catastrophic event*. Figure 3.2 shows the relationship between Cristian's classification and possible outcomes from a server request. Either the server returns an expected answer, fails in an expected way, or fails in an unexpected way. The stochastic behavior expresses the *minimum* probability p_s that the system should respond according to the standard semantics and the *maximum* probability p_c for a catastrophic event.

Cristian classifies server failures as in the following list:

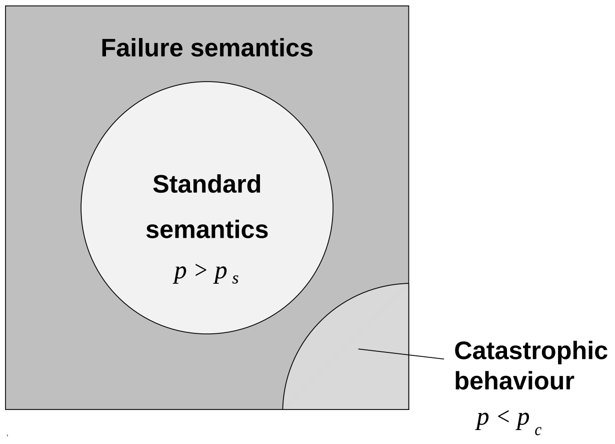


Figure 3.2: The space of possible outcomes from a server request.

Omission failure	The server does not respond to a request.
Crash failure	The server does not respond to a request and will not respond to any more requests.
Timing failure	The response from the server arrives too early or too late.
Performance failure	The response from the server arrives too late.
Response failure	Incorrect execution of a request.
Value failure	Incorrect response to a request.
State transition failure	The state transition caused by a request is incorrect.
Arbitrary failure	Any failure outside the standard semantics.

These classes are not disjunct since they partly overlap or is contained in each other. Value and state transition failures are both subclasses of response failures, performance failures are a subclass of timing failures, etc. Figure 3.3 shows how the sets of failures relates.

The failure semantics of a DBMS server is traditionally expected to be performance failures (replies on requests are coming too late) and omission failures (transactions are rolled back in the case of a failure or never responded to if the system is down). When a response is received from the DBMS it is assumed that the execution is according to the ACID rules with a correct database state transition and a correct answer.

The standard semantics of a DBMS server is not so clear since there are many alternative ways to distribute DBMS functionality and application code over the client and server. The question to ask is how this distribution can be done to achieve as good service availability as possible. The next sections try to answer this.

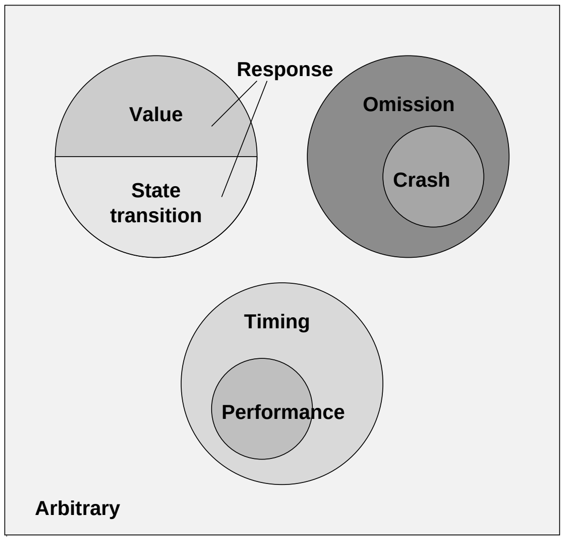


Figure 3.3: Failure classification.

3.3.2 Functionality of a DBMS server

Figure 3.4 shows four alternatives for distributing functionality between a client and a server. In alternative **A** the server only provides single tuple[1] or disk page access to the database including access methods, concurrency control, logging and recovery. The application interface, query compiler, query optimizer, and query executer is located in the client. A join between two relations is for example executed in the client by fetching single tuples or disk pages from the server. This means that the DBMS partly resides on the server, partly on the clients. The DBMS uses a communication protocol which can be proprietary for the DBMS. The application software executes only on the client together with presentation software like a terminal driver or a window system like MIT X11.

In alternative **B** all DBMS functionality is placed on the server. The interface to the server is the DBMS query language (e.g. SQL) which the application can access directly. This alternative clearly separates application and DBMS by assigning the DBMS to the server and the application to the client. While the interface between the application and the DBMS in alternative **A** could be a *call level interface* (CLI) where the DBMS is accessed through procedure or system calls, the interface for alternative **B** is the communication protocol used between the client and the server.

Alternative **C** allows the application to partly execute on the client, partly on the server. The interface between the application and the DBMS is a CLI at the server side. The communication between the client and the server is handled by the application.

Alternative **D** places the entire application onto the server leaving only a presentation interface on the client. The client is then no more than a terminal attached to the server

[1]We will throughout this work assume the relational database model. But, this does not exclude that the results also can be employed on other models like the hierarchical, network, or object oriented database models.

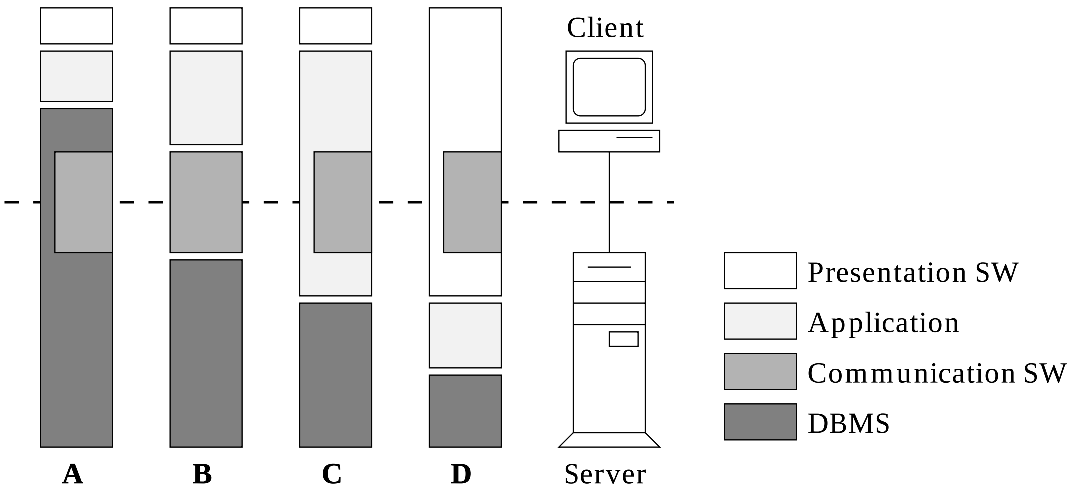


Figure 3.4: Alternative placement of DBMS and application in a client-server environment.

and the distribution is handled by the presentation software. This corresponds to the traditional mainframe solution.

Alternative **C** has been the first step towards real client-server processing. It has been driven by the application developers caused by lack of standards and available products. New standards like ISO SQL, ISO RDA, and ISO TP standards are pulling in the same direction toward solutions based on alternative **B**. Commercial DBMS vendors are starting to provide products supporting these standards. Several Object-Oriented DBMS' are providing functionality similar to alternative **A**.

There obviously are more alternatives than these four. Some are combinations of the four above. It is for example possible to imagine that the server only provides services for reading and writing disk pages without any file system or locking primitives. But, in this case we do not talk about a DBMS server and have no concurrency control between clients.

The list on page 23 contained guidelines for design of client-server applications. Applied to the four alternatives **A**, **B**, **C**, and **D** we get the following table:

Alternative	$+$	0	$-$
A	1, 4	5	2, 3
B	1, 2, 5	4	3
C	3		1, 2, 4, 5
D	2, 5		1, 3, 4

The table shows how well the four alternatives follow the guidelines. The "$+$" column lists the number of the guidelines well taken care of, "$-$" the guidelines which is ignored and "0" those partially taken in account.

3.3.3 TP-SQL service

Based on the design guidelines listed in the start of section a solution between **B** and **C** combining the best from both seems to represent the best client-server solution. This is no unique result since both database vendors and standardization bodies tend to have come to the same conclusion. The hybrid, which in the following will be called the *TP-SQL service*, is from the name based on SQL and is especially suited to OLTP. As mentioned before, OLTP consists of small transactions accessing few tuples without user interaction. This is not very well supported by a pure **B** solution since an SQL call must be issued for each tuple operation resulting in a large number of messages. A transaction accessing three tuples will therefore result in four messages (three tuple statements and one `COMMIT WORK` statement) sent from the the client to the server and four corresponding reply messages. Combining these four statements into a single message with a single request and reply greatly reduces the communication. This is easy to solve by allowing a request to contain a sequence of SQL commands. Regrettably this is not sufficient since some of the SQL commands in the same sequence can depend on results from earlier commands in the sequence, as for example to follow a relationship between two tuples. SQL as in the SQL 92 standard [Org92a] is not complete with this respect.

The **C** alternative on the other hand allows the server to run a part of the application code which can issue a sequence of SQL commands. Unfortunately this means that the server must provide application code execution and the service protocol must be defined and implemented by the application programmer.

The solution to this problem is to extend the SQL language to a complete programming language. Commercial vendors (e.g. Sybase with *Transact SQL*) have done this by extending SQL with a full procedural language which includes variables and control structures like loops, branches, procedures, and exceptions. This work has been followed up by ISO which in current proposals suggests to include this functionality into a future SQL standard (SQL-3 [Org92b]). This also opens up for precompiled SQL procedures greatly improving performance. Figure 3.5 shows an example of an SQL procedure for the TPC-B benchmark transaction [Gra91] written with the proposed SQL-3 standard.

Some readers might claim that user-written SQL procedures are a part of an application and that TP-SQL services should be classified as an alternative **C** system. On the other hand, there are some significant differences. The client-server protocol is not the responsibility of the application programmer, it adheres to standards and the procedural language is restricted to only access the database resources. As we will see later this makes up a significant difference for the reliability and security of the server.

3.3.4 Dependability aspects

A nice property of the client-server model is that it is also very well suited to fault-tolerance and dependable database systems.

As seen before, software is a significant cause for errors in computer systems. The number of errors in software also seem to be proportional to the number of lines of code and reducing the code volume should therefore reduce the total number of failures and therefore also increase the reliability. Another important observation is that a software error on the server can have much more severe consequences than an error on the client. If the error

```
MODULE tpc
  LANGUAGE SQL;
.
.
.
  PROCEDURE tpcB(:account  NUMERIC(9),
             :branch   NUMERIC(9),
             :teller   NUMERIC(9),
             :delta    NUMERIC(10),
             :balance  NUMERIC(10));
  BEGIN
    BEGIN WORK;
    UPDATE accounts
      SET    Abalance = Abalance + :delta
      WHERE  Aid = :account;
    SELECT Abalance INTO :balance
      FROM   accounts
      WHERE  Aid = :account;
    UPDATE tellers
      SET    Tbalance = Tbalance + :delta
      WHERE  Tid = :teller;
    UPDATE branches
      SET    Bbalance = Bbalance + :delta
      WHERE  Bid = :branch;
    INSERT INTO history (Tid, Bid, Aid, delta, time)
      VALUES (:teller, :branch, :account, :delta, CURRENT);
    COMMIT WORK;
  END;
.
.
.
END MODULE;
```

a)

```
EXEC SQL CALL tpc.tpcB(:account, :branch, :teller, :delta, :balance);
```

b)

Figure 3.5: SQL-3 procedure example. a) The TPC-B benchmark transaction written as an SQL-3 procedure. b) Invocation of the procedure from embedded SQL.

causes a fault on the client, only the users of that client is affected. If the server fails, *all* the users are affected. It should also be noted that system software like operating systems and database systems have a lower failure rate than application software. System software have often been in longer use and are better tested than applications.

Based on these assumptions we can make a simple analytic model for accumulated service unavailability caused by software errors. Given the following variables:

r_s	= Fraction of code of a transaction running on the server
r_c	= Fraction of code of a transaction running on the client
I_s	= Failure intensity of server code (failures / lines)
I_c	= Failure intensity of client code (failures / lines)
n_c	= Number of clients
l_c	= Load caused by one client (transactions / second)
s	= Code volume executed for each transaction (lines / transaction)
t_r	= Average repair time of a failure (seconds / failure)
U	= Unavailability due to software failures per time unit

and:

$V = l_c n_c s$	= Total lines of codes executed per time unit
$r_s V$	= Lines of code executed on the server
$r_c l_c s = (1 - r_s) l_c s$	= Lines of code executed per client
$f_s = r_s V I_s$	= Number of faults per time unit on the server
$f_c = (1 - r_s) l_c s I_c$	= Number of faults per time unit on each client
$n_c t_r$	= Average total time unavailable caused by a server failure
$n_c f_c$	= Accumulated number of client failures per time unit

We will here assume that a server failure causes all clients to be blocked for a time t_r, i.e. accumulated time unavailable for a server failure becomes $n_c t_r$. This is likely for severe errors in the operating system and the DBMS kernel but not so much for application code. Based on this assumption the total unavailability for a given time interval then becomes:

$$\begin{aligned} U &= f_s n_c t_r + n_c f_c t_r \\ &= r_s l_c n_c s I_s n_c t_r + n_c (1 - r_s) l_c s I_c t_r \\ &= l_c n_c s t_r (r_s I_s n_c + (1 - r_s) I_c) \\ &= l_c n_c s t_r (I_c + (I_s n_c - I_c) r_s) \qquad (3.1) \end{aligned}$$

The interesting part here is $(I_s n_c - I_c) r_s$. It shows that *the unavailability increases with increasing fraction of code running on the server* as long as $I_s n_c > I_c$. For a system with 100 clients that means that the client must run code with 100 as many errors pr. line as the server. For systems with a large number clients and reasonable client code quality the condition is likely to hold. The conclusion must therefore be that the server should run as little of the total code volume as possible favoring systems based on alternative **A** (best) and **B** (fair).

Usually system software like operating systems, DBMSs, and communication software have a higher technical complexity than application code and is complicated to debug. It is therefore reason to believe that the error intensity should be higher for system software. To compensate for this the developers put much efforts into the production of this kind of code through careful design and thorough testing. The operating system is also used by a large number of applications and users with the likelyhood for eventual errors to be spotted and fixed. It is therefore reasonable to believe that system software has better quality than application code.

Even though a failure in an application running on the server is unlikely to directly crash the server, indirect effects can result in a crash. A failure will cause extra *stress* on the surrounding system. The failure represents a (possible) new execution path which again might introduce new failures. The failure must also be detected, masked, and repaired meaning more work for other modules in the server. This again results in higher load of the system moving it closer to its performance limit. In [BEHS86] it is shown that a systems with a low load is less exposed to failures than a system with a high load.

Since bad application code might be a serious threat for server availability, *the server should not be allowed to run any application code*. The system administrator of the server can not make any guarantees for system availability if application programmers freely can run parts of their software on the server. Thus, the quality of all software running on the server including application code must be known to make any guarantee on the availability.

All new server software represents a threat to server availability. Replacing software often means that the service must be stopped, data converted, the new software loaded, and the new server started, tested, and set into operation. During the time this work is done the service is unavailable. Modern microkernel-based operating system partly solves this by *port migration* [RR81, RAA$^+$88].

New software releases often have new latent bugs and the behavior from old revisions has changed, intended or not. New software should therefore not be introduced without rigorous testing and only at well planned occasions with procedures ready to go back to the old version in case of serious trouble. It is also advisable to do few software replacements, but with larger increments.

New client software is not that critical. The new software can be introduced with small increments, one client by another. During installation only a single client is blocked and can be scheduled to non-office hours. By keeping the functionality of the server to a minimum most software upgrades are only necessary on the less critical clients improving server availability. This conclusion is also supported by the fact that application software are more dynamic than system software. Applications have reduced quality and change with changing user requirements. System software is more often locked to a fixed specification (e.g. an international standard).

The conclusion so far in this section has been that no application code must be allowed to execute on the server. So what about SQL procedures? For purists the answer will be *no*, but on the other hand they represents a compromise between reliability and performance. The consequences of a failure are also not too serious as for general application code. The expression richness of the language is very restricted and the procedures are executed under DBMS resource and transaction control.

3.3.5 DBMS server availability

The definition of availability given in chapter 2 is based on the measure “fraction of time” which do not match the request-response nature of a server where a response is either correct or not. This presents a problem since requests and responses represents events in time, not periods. A simple way out is to define a service unavailable for the interval from a given request gets no correct response, until a request gets a correct response again.

A DBMS server (and servers in general) has different availability measures based on the observers’ viewpoint and their service definition. The end user of the system sends DB requests and receives replies. The service expected depends both on the DBMS server itself, communication channels, and client hardware and software. For the user it is equally important that all these components operate correctly. More important, he or she is not able to observe any degrees of quality except response latency. Either the server responds with an answer within the latency limit or not.

The DBMS server administrator on the other hand sees one server and a large collection of clients. For him or her the most important goal is to have the server and most clients operate correctly. If a client has a malfunction only one (or a few) user does not get its service, but if the server is out of service, nobody can get their work done. The server administrator does not consider the system unavailable even though a client is out of service.

The service availability definition seen by the client is therefore:

> The service is available if more than a fraction S of the requests sent from the client gets a correct response with a latency less than ℓ_{client} after the request was sent measured over a time interval Δt.

An error response due to a system condition (e.g. a transaction rollback caused by resolving a deadlock) is not considered to be a correct response even though it occurs inside the allowed latency time.

The service availability definition seen from the server becomes:

> The service is available if more than a fraction S of the requests received by the server returns a correct response with a latency less than ℓ_{server} after the request was received measured over a time interval Δt.

Since we will focus on the server availability aspects the latter definition will be used.

3.3.6 DBMS server service definition

The previous sections have discussed the standard and failure semantics of a server in addition to the definition of server availability. They are all part of the service definition of a server. Table 3.2 lists the set of parameters in the service definition which will be considered in this work. *Response* and *state transition* is the standard semantics of the server. The *maximum load* specifies under which conditions the server is operating. *Maximum latency*, *success rate*, and *measurement interval* specify the conditions for determining whether the system is available or not.

Metric	Symbol	Unit	Description
Response			The correct response to the request
State transition			The state change of the server caused by the request
Failure semantics			Allowed failure behavior
Maximum load	C	requests per second	Maximum number requests the server is able to receive per second
Maximum latency	ℓ	seconds	The maximum allowed delay from a request is issued until the response is received
Success rate	S	%	Minimum fraction of the requests to complete successfully according to the service definition
Measurement interval	Δt	seconds	Time interval used to compute success rate

Table 3.2: Service definition for a server

Metric	Symbol	Value
Response		New account balance
State transition		Update account, teller and branch records. Insert history record.
Failure semantics		Performance failure and omission failure
Maximum load	C	tpcB transactions per second
Maximum latency	ℓ	2 seconds
Success rate	S	90 %
Measurement interval	Δt	15 minutes

Table 3.3: Service definition for a DBMS server based on the TPC-B benchmark.

Table 3.3 shows an example of a service definition based on the TPC-B benchmark ([Gra91]). The "tpcB" number is the highest value for which the system is considered available.

3.4 Database systems and fault-tolerance

Database systems providing a transaction mechanism are also providing a certain fault-tolerance functionality. The transaction ACID properties — *atomicity*, *consistency*, *isolation*, and *durability* — are concepts making the database system able to handle a large class failures as for example a computer crash.

Logging and recovery are essential mechanisms employed by DBMS' to implement transactions. In addition to transactions, logging and recovery can also be used to improve availability by other means, e.g. on-line production of consistent replicas and keeping them synchronized. Hvasshovd [Hva92] does an extensive analysis of logging and recovery for very high availability applications.

Replication is another important concept for fault-tolerance and improved availability and are together with transactions a powerful pair. The rest of this work will focus on how replication can be used and implemented to achieve an as good as possible availability.

3.5 Causes for DBMS unavailability

There are several sources for service interruption of a DBMS server. A classification of these is given in figure 3.6. As noted before, unavailability is not necessary coupled to failures. A failure does not need to cause unavailability, and unavailability can be caused

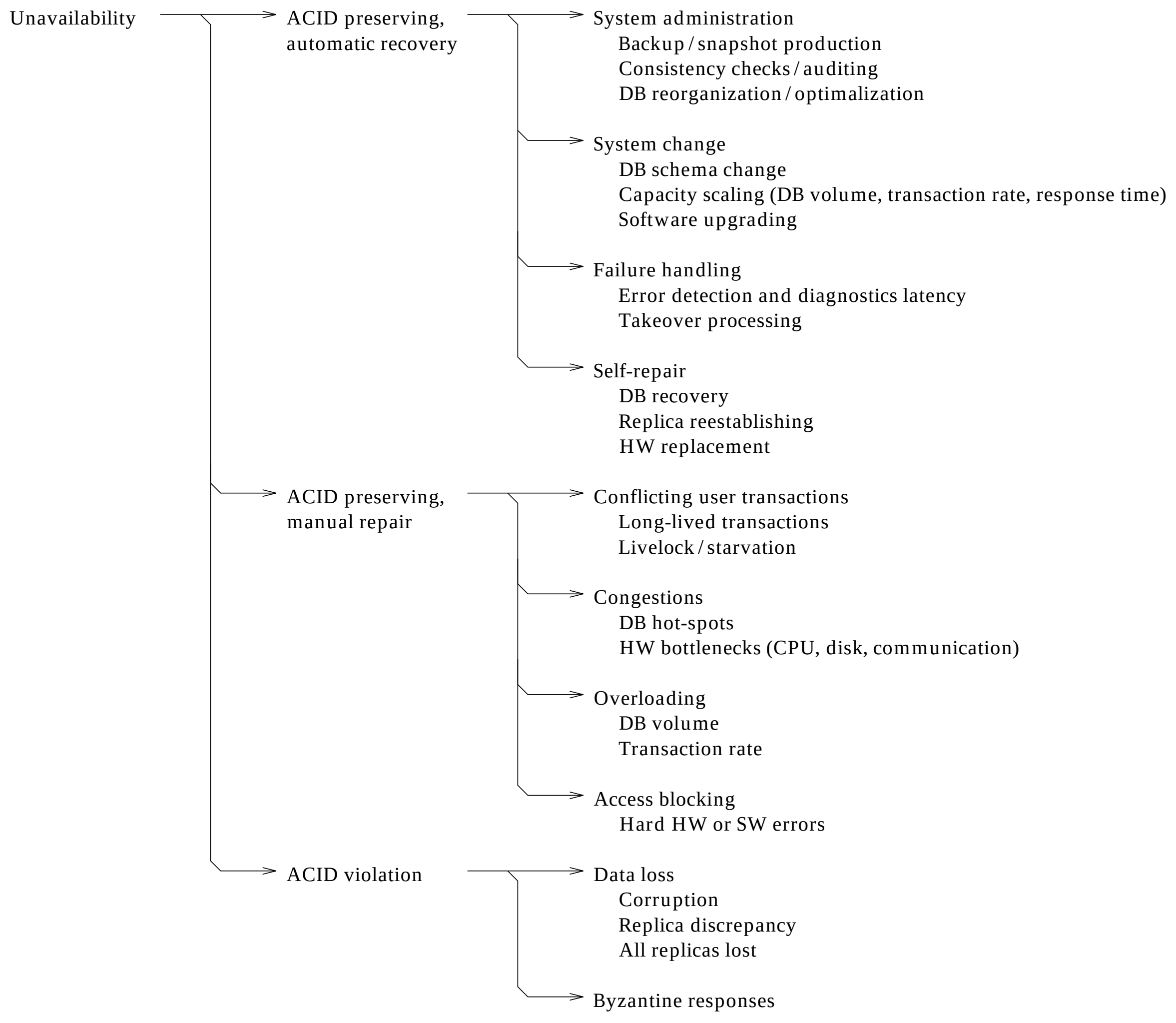


Figure 3.6: A classification of sources for unavailability for a DBMS.

by other sources than failures. This classification differs from Grays classification [Gra90] in that Gray uses the source of the event causing service *outage* as basis for classification: software, hardware, maintenance, operations, environment, or process. This classification on the other hand is based on the phenomena causing the unavailability. As the figure shows, there are three main classes of unavailability:

ACID preserving, automatic recovery: This class of unavailability is only intermediate and availability will be automatically be restored after the phenomena that caused the unavailability has disappeared.

For many commercial database systems *system administration* (e.g. taking backups, doing checks, and database reorganization) can not be done when the system is on-line. Also, most systems can not handle on-line *system change* like changes to the database schema, hardware, and system software.

Even if a system is resilient to failures, the *failure handling* can cause some intermediate unavailability. The service is normally unavailable until the failure is detected and diagnosed. The processing required for an alternative module to take over the task from a failed one can also cause unavailability.

The last entry in this class is *self-repair* which is unavailability caused by the system during reestablishment of its own service ability. Database recovery after a crash is the most typical but also replica reproduction and hardware reintegration can cause service interruption.

ACID preserving, manual repair: This class is characterized by the need for assistance from an external entity like a human or another computer. Note that the unavailability does not need to be continuous in that the server might alternate between an available and an unavailable state as long as one phenomena is the cause. An insufficiently configured computer might be saturated only during busy hours. In these busy periods the specified service is not provided, but outside them it will be classified as available. Insufficient computer resources are a non-continuous phenomena. Physically upgrading the computer is manual repair of the problem.

Conflicting user transactions are a source for unavailability that needs manual interaction to resolve. Section 3.2 described how long-lived OLCP transactions could block service for OLTP transactions. Live-locks and starvation are related problems. To solve this either the data model must be modified, the applications changed, or the DBMS improved (as described in section 3.2).

This class of unavailability is less desirable that the first one. In this case the system depends on an external entity for providing a specified availability.

ACID violation: This is the worst of the three unavailability classes in that it is outside the failure semantics of a DBMS and is considered a breach of the safety. In many cases the failures causing this kind of unavailability are not possible to recover and if they are, it is very time consuming and requires manual interaction.

Chapter 4

System platform

This chapter will discuss a machine platform suitable for a highly available DBMS server. With platform we mean both hardware and lower level software. The software includes operating system and communication software.

In the previous chapter we looked upon the DBMS server as a single abstract entity. As we will see in this chapter all the proposed architectures for highly available DBMS servers consist of multiple components for redundancy and performance. Despite this the server software can hide the details of distribution from the clients which will observe the server as a black box just doing its job.

4.1 Hardware

4.1.1 Machine architectures

Stonebraker [Sto86] describes three architectures for parallel database systems: *shared-nothing*, *shared-disk*, and *shared-everything*. Copeland and Keller [CK89] extend these three architectures to the six which are illustrated in Figure 4.1. Common for all architectures are the three types of components — processors, memories, and disks — which are interconnected through local buses and a global communication network. The collection of components connected with a local bus are called a *node*. The six architectures are:

Shared-nothing: One processor, one disk, and a local memory are connected together with a local bus. Multiple of these nodes are interconnected through a communication network.

Shared-disk: Multiple nodes, each composed of one processor with local memory, are interconnected through a communication network. Multiple disk units are also connected to the network so that any processor directly can access any disk.

Shared-memory: Multiple nodes, each composed of one disk and one processor, are interconnected through a communication network. One or more memory modules are also connected to the network so that any processor directly can access any memory module.

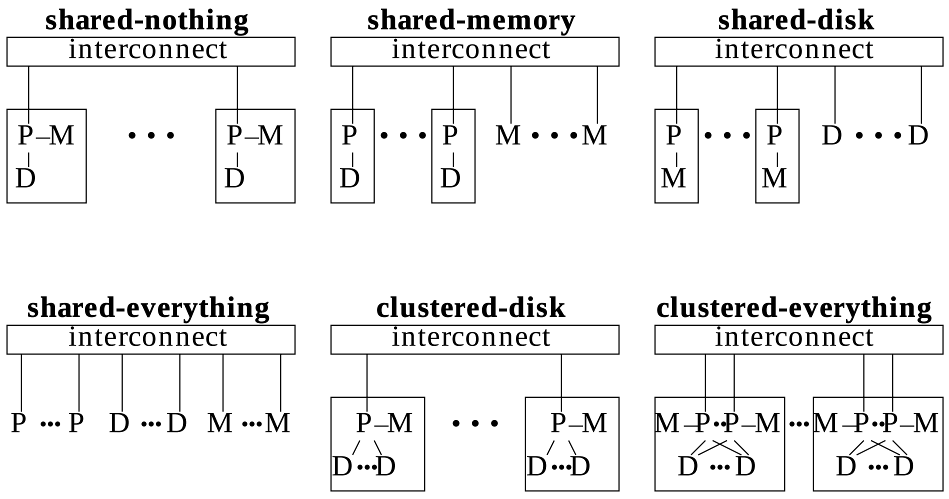


Figure 4.1: The six machine architectures for parallel database systems described in [CK89]. P=processor, D=disk, M=memory

Shared-everything: Multiple disk, memory, and processors are individually connected to a communication network so that any processor can access any disk or memory module.

Clustered-disk: Same as shared-nothing except that each node can consist of more than one disk.

Clustered-everything: Same as shared-nothing except that each node can consist of several processors, disks, and memory modules connected with a local bus. In essence each node is a shared-everything computer.

Sharing resources means that there is a shorter path to the resource and therefore less delay, less overhead, and better performance. Thus, from a performance viewpoint shared-everything is the preferred architecture. For database systems this is confirmed by an analysis performed by Bhide in [Bhi88]. The sharing property also provides fault-tolerance by allowing any component to access a resource. If a component fails another component can immediately take over its task since it has access to the same resources.

Unfortunately, sharing has some obvious flaws that discourage its use. If a failed component is allowed to continue to access other components after the failure, it can corrupt the system state and cause unrecoverable damage to data. A failed process running on one processor can for example overwrite the address space belonging to another process running on another processor.

The behavior of failed components can also block good components from accessing a resource. A failed processor can for example jam a bus shared by several processors. One approach is to build the components with a fast-fail property and make it possible to isolate them from the interconnection network when they have failed. For hardware this has proven successful (e.g. Stratus [Fre82]) but for software there are mixed results.

Rigorous n-version programming approaches [Avi85] handle some faults but are still prone to faults caused by design-failures.

Sharing memory between multiple processors also suffers from bad scalability, a high interconnect cost, and complicated data-structure management. Memory sharing requires a low communication latency and a high bandwidth that up to now only have been available using bus solutions. These buses limit the maximum distance between the components and have an upper limit on the number of components that effectively can access it. New approaches based on cache coherence protocols like *Scalable Coherent Interface* (SCI) [JLGS90] are meant to overcome the scalability problem but do not solve the fault-tolerance issues. Processes sharing memory still can corrupt the data contents for each other. A problem that is difficult to solve is the management of concurrent accesses to shared data.

The shared-nothing, clustered-disks and clustered-everything architectures are much less prone to these problems. The performance requirements for the interconnect are less strict and allows much larger configurations and distribution further apart. It is also much easier to isolate failed components and implement hot-replacement modules. The communication between nodes is based on message passing or remote procedure calls (RPCs). Inside a node a failed component can still corrupt the other components but it is most likely to be limited to that node only. When a component fails all components on that node can be suspected to be contaminated and the whole node be considered to have failed.

A *sharing domain* is the collection of processors, disks and memories where all components directly can access each other. For the shared-nothing, clustered-disk, and clustered-everything architectures each node is a sharing domain. For shared-everything the full system is a sharing domain.

Since a component fault is a threat against the other component in the same sharing domain it is a conservative approach to consider all the components in the sharing domain to have failed. In the rest of this thesis we will assume this conservative approach and consider the sharing domain to be the *unit of failure*. Obviously, we do not want the full system to always fail when a single component fails. This rules out the shared-everything / memory / disk architectures and we are left with shared-nothing, clustered-disks and clustered-everything. These three will be treated identically in the reminding work and shared-nothing will be used as the group term. The number of processors, disks, and memories in each node is a question of balance between different types of resources and the granularity of the system. Coarser granularity means that a failure will knock out a larger fraction of the system with the consequential reduced performance and service degradation.

In addition to node failures we can also expect failures in the communication interconnect. A full interconnect failure will result in that no nodes can communicate and the system must be considered down. Partial failures can isolate some nodes but still offer the connectivity necessary for the system to provide service. In the best case no nodes loose their ability to communicate and only a reduction in bandwidth is seen. Bus systems and other shared media interconnects (e.g. Ethernet) are prone to full failures. Fault-tolerant architectures based on buses get around this by using replicated buses (e.g. Tandem's NonStop architecture [Kat78] and Stratus [Fre82]). Full failures can in most cases be avoided using switched interconnects.

A large number of comparative studies have been written on existing fault-tolerant systems and database machines. See [Kim84, Ser84, Cua89, Sie90] for surveys of fault-tolerant systems and [Su88] for a extensive overview over database machines.

The current dominating architecture for commercial large-scale database machines seems to be shared-nothing. Both commercial systems like Teradata DBC / 1012 [Ter85] and ICL Goldrush [ICL93], and research systems like GAMMA [DGS$^+$90], and HC16 / 386 [BG89] are examples on such systems. Even though both Teradata and ICL computers have built-in fault-tolerance mechanisms, they are not sold as general fault-tolerance computers. The most successful vendor in this respect is Tandem with their NonStop architecture [Kat78], which mainly is a shared-nothing architecture, but allows two nodes to share the same disks. Among the fault-tolerant systems based on a shared-everything architecture are Stratus [Fre82] and IBM mainframes [Sie92].

4.1.2 Site failures

In this chapter we have classified failures as either *node failures* or *interconnect failures*. By assuming fail-fast nodes we know that no node-internal fault will spread to other nodes, thus only one node can fail caused by a single node-internal fault.

An external fault can cause more than one node to fail at the same time. Let us define *site* as the set of nodes and the part of the interconnect located on the same physical location so that they can collectively fail as the result of a single event. Such an event can be a disaster like a fire or earthquake, a power failure or a human mistake. A *site failure* is a failure causing all or a major part of a site to fail.

The disadvantage with most (if not all) existing computer systems is that they only offer fault-tolerance inside a single computer room or very closely located buildings. Robustness against a site failure is achieved by replicating the full system on a remote site. This adds to the redundancy already used to implement local fault-tolerance inside the site.

This restriction has been caused by expensive and slow Wide-Area Network (WAN) technology. But the arrival of fast and cheap[1] technology based on optical fibers and cell switching opens up for new solutions. For high-availability database systems this has been suggested both by [SS90, Hva92]. The next section will describe how a system can be achieved with fairly straight-forward workstation class hardware and commercially available interconnect hardware. This avoids the need for specially designed (and therefore expensive) hardware components, something which has been common for fault-tolerant systems up to now.

4.1.3 Workstation clusters for database systems

Workstation clusters is an architecture that recently has been introduced by the major computer vendors like IBM and Sun. This is a shared-nothing architecture where conventional workstations are the nodes. A high capacity network, e.g. FDDI or ATM, interconnects the nodes [Par93].

Since workstations are general computers there should be no reasons against using a work-

[1]Read: less expensive.

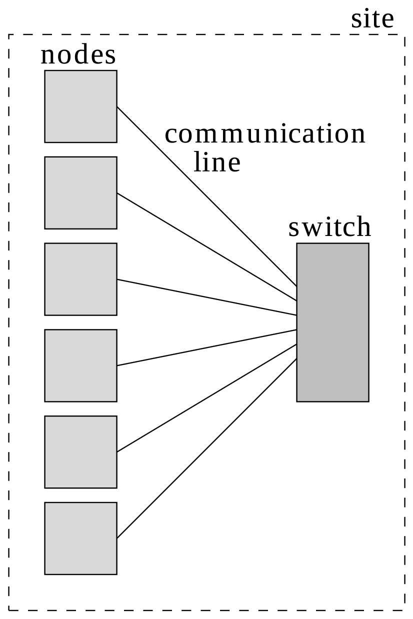


Figure 4.2: A single-site, single-connected (SS) database cluster.

station cluster for database applications as long as the communication network provides sufficient capacity, but for use in fault-tolerant computing there are some points that may cause concern.

Let us call a cluster solution for database processing a *database cluster*. For the system to be scalable with respect to the number of nodes the communication network must be able to handle the communication from a wide node count range. Shared medium solutions like Ethernet, Token Ring and FDDI have an upper limit on the aggregate communication bandwidth that prohibits their use in large systems. Asynchronous Transfer Mode (ATM) (see Section 4.1.5) and frame-relay architectures on the other hand are based on switching and do not have the bottleneck inherent in the shared medias. Our model of a database cluster therefore becomes a set of nodes connected to a communication switch with communication lines. The WAN capability of ATM also allows for multiple sites and the possibility to mask site failures.

Since a communication solution using ATM consists of two distinct components — switches and communication lines — there are four types of failures in such a database cluster that should be masked: node, communication line, switch and site failures. Below follows the descriptions of four different configurations and how they handle the four failure types.

Single-site, single-connected (SS)

Figure 4.2 shows a database cluster located on a single site which has a single network connection for each node. The configuration is robust against node failures but will not be able to handle neither a site nor a switch failure. Communication channel failures can not

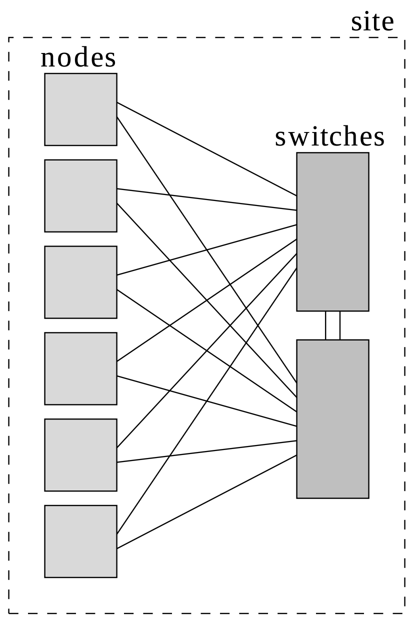


Figure 4.3: A single-site, double-connected (SD) database cluster.

be distinguished from node failures and must therefore be handled as a node failure. Node failures are handled by replicating data over multiple nodes and migrating processes after a failure.

Single-site, double-connected (SD)

A double-connected cluster (Figure 4.3) differs from a single-connected cluster by having two network connections on each node where each is attached to independent switches. The switches are also connected together with one or more communication lines to ensure full connectivity when multiple communication lines have failed. The redundant switch allows SD to mask a switch failure. An additional advantage is that communication line failures can be masked without treating the connected node as down. As for SS, SD can not mask site failures.

Compared with SS the additional cost is a fully replicated communication network.

Dual-site, single-connected (DS)

A dual-site cluster (Figure 4.4) is built by taking two single site systems and connecting the switches at the two sites with at least one communication line. This configuration is able to handle both node, communication line, switch and site failures as long as there are at least one replica of all data stored on each site. A switch failure will be observed as a site failure by the other site. The nodes connected to the failed switch will observe this identical to a intra-site communication line failure. If all inter-site communication lines fail

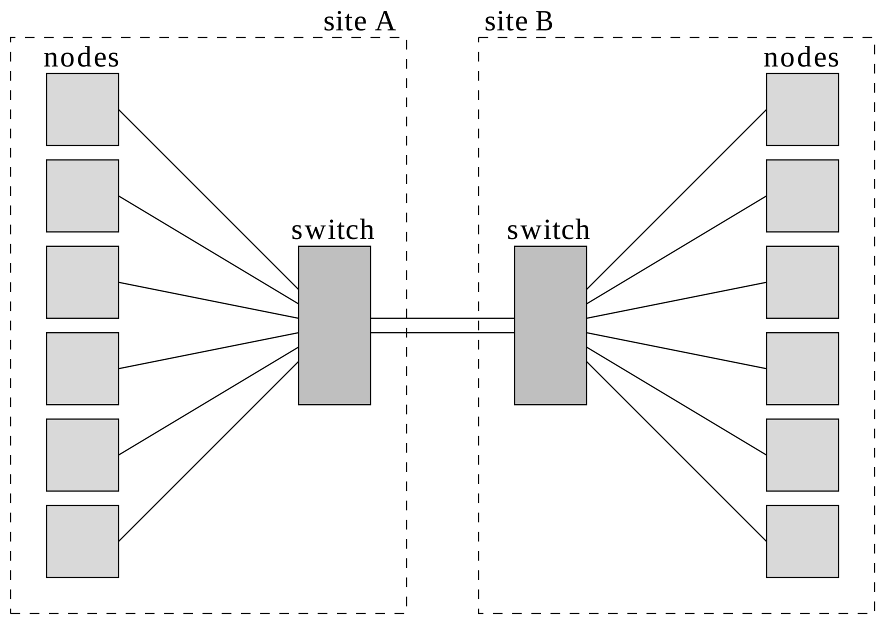


Figure 4.4: A dual-site, single-connected (DS) database cluster.

both sites will assume that the other site has failed. To avoid inconsistencies when the sites are re-connected only one of the sites should be allowed to provide full database service during the interruption.

Note that DS configuration also can be used inside a single site reducing the number of necessary connections to nearly the half compared with the SD. The disadvantage is a restriction on how the data is replicated over the nodes.

Dual-site, double-connected (DD)

A dual-site double-connected database cluster (see Figure 4.5) combines SD and DS. It handles both node, line, switch, and site failures without making more than the failed unit unavailable. For both SS and DS a switch failure meant that all the nodes on that site became unavailable. The disadvantage is the higher cost caused by replicating communication lines and switches.

Table 4.1 shows a summary of how the various types of failure are observed for the four alternative configurations.

Note that both DS and DD can be extended to more than two sites for increased redundancy.

4.1.4 Client communication

Up to now in this chapter the communication with the client computers has been ignored. There are two options:

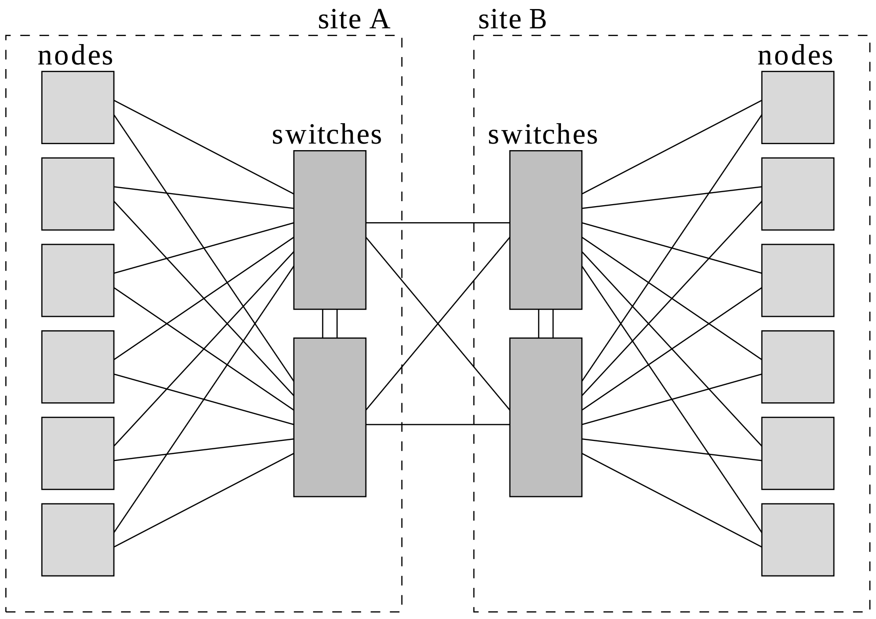


Figure 4.5: A dual-site, double-connected (DD) database cluster.

	Failure			
	Node	Switch	Line	Site
SS	N	A	N	A
SD	N	X	L	A
DS	N	S	N	S
DD	N	X	L	S

N=node unavailable
X=switch unavailable
L=communication line unavailable
S=site unavailable
A=system unavailable

Table 4.1: Comparison of failure consequences for four database cluster configurations.

- Some or all nodes have communication interfaces which connect the node to one or more client communication networks. When a client issues a request, the request is received by a node which processes the request itself or forwards it to another node.
- The client network is directly connected to the server interconnect, i.e. a switch. Requests are received by the switch and routed to a destination node.

For fault-tolerant client communication the client network should be redundant and connected to a node or switch at both sites. In the case of a site failure the client can then connect to the other site for service.

4.1.5 Hardware components

A prerequisite for masking an error is to be able to detect it before it becomes visible. It is important not to let a fault changing the state of the system pass undetected. The fail-fast property of the components is the essential mechanism to achieve this but is it reasonable to assume this for the suggested hardware?

Most workstations consist of a CPU-board integrating processor, I/O controllers, and memory. In addition there are one or more disks, a floppy drive, display, keyboard, mouse, power supply, and some cabling. The floppy drive, display, keyboard, and mouse is of no use and will not be considered. The cabling are much more reliable than the other components and will be ignored as a source for faults.

CPU-board

The CPU board of a modern workstation consists of a processor (CPU), memory, and I/O controllers. Memory is the component most prone to errors. Most workstations are therefore equipped with parity or error-correcting codes (ECC). Parity can only *detect* single-bit errors in a machine word but this is sufficient for being fail-fast in most cases. The ECC used by most computer systems corrects all one-bit errors and detects all two-bit errors. The detection capability means that a larger class of errors are caught.

To detect CPU failures two CPUs can be run in lock-step with one supervising the other. If one of the processors fail the discrepancy between the output of the two CPUs is detected and the system is reset. Several high-performance microprocessors offer this functionality and when used will provide fail-fast operation. Unfortunately this is not used in normal workstations and the fail-fast assumption is questionable.

Disks

Originally hard-disk drives with their mechanics were among the less robust parts of a computer, but in recent years this has changed. Modern 3.5” hard-disks with 500,000 hours projected MTTF are common and IBM has announced the 0662 drive family in the 800,000 hour range, i.e. more than 90 years. Note that this MTTF figures must not be confused with the drives lifetime. The MTTF figure gives the failure rate during the lifetime of the system.

In real life disks fail more often than the lifetime and MTTF figures indicate since they expresses failures of the full drive. More often the drive is not able to retrieve one or more written sectors. This can for example be caused by wear of the magnetic coating on the platters, bad spots on the platter or temperature changes causing change in the dimensions.

A disk drive has embedded some fault-tolerance mechanisms to handle most of those failures. The data is stored on the disk with an error correcting checksum and most errors are masked with it. If the error can not be corrected the disk automatically retries to read the disk a few times before giving up and flags the error to the operating system. The Fujitsu M2622 data-sheet [Fuj91] specifies that every 10^{10} bit read from the disk is a read error that can be detected and corrected by the disk. Only once in 10^{15} bits an error is passed undetected to the user. Detected but unrecoverable errors are forwarded to software layers which must handle them. The data must be recovered if possible and the damaged disk sectors isolated from further use. With this low error rate it is reasonable to believe that disks are fail-fast.

Communication

A promising standard for communication between the nodes is Asynchronous Transfer Mode (ATM) [Par93]. It is a communication standard initially designed for telecommunication applications that have evolved into computer communication use. It allows a wide range of bandwidths and can be used to build large networks. Today it exists workstation adapter cards supporting 155 Mbits/s bi-directionally between the workstation and the network through either optical fiber or twisted pair copper wire. Optical fiber has very low error rates and will electrically isolate the workstation from the network and the other components in the network.

The most important components in an ATM network are the switches. Through communication lines a switch can be connected to either nodes with adapter cards or other switches. Communication between two nodes is achieved by establishing a virtual circuit from the source node through a set of intermediate switches to the destination node.

The simple and general nature of ATM allows fast and efficient switching of messages and is suitable for use in both LAN and WAN applications. The LAN capability is good for intra-site communication and the WAN capability is necessary for communication between sites. The standardized interface makes it possible to also connect the client computers to the database cluster through the ATM interconnect.

In addition to permanent switch and communication link failures, ATM is also allowed to drop messages. This message loss can be caused by a transient congestion in the network or noise. Software protocols must recover this message loss.

Traditional communication techniques using checksums, sequence numbers, acknowledge messages, and timeouts guarantee that communication failures do not pass unnoticed and violate the fail-fast property.

Power supply and cooling

Up to now we have not discussed power supply and cooling of the equipment. In the case of workstation nodes we have a *Power Supply Unit* (PSU) for each node. If a PSU fails, the node fails and the failure is handled like a node failure. A power outage will seldom happen on two sites at the same time if the sites are sufficiently far apart and will therefore stop the operation of one site only. If the site is equipped with an *Uninterruptible Power Supply* (UPS) with batteries supplying the power if the primary feed is lost, the number of site failures can be reduced. Unfortunately introducing an UPS also introduces another component that can fail.

Electrical components are specified to operate inside a limited temperature range, and if this range is exceeded, the components will probably fail. To ensure correct operation of a computer, the generated heat must be transported away. For a desk-top system in a normal office environment this is handled by the system itself. If the environment temperature is too high, artificial cooling must be provided. Since reducing the temperature to below room temperature also will increase component life and reduce the failure rate, cooling might be considered just for increasing availability.

Traditional air cooling equipment for computer rooms should be sufficient for cooling a database cluster site. The outcome of a cooling equipment breakdown is uncertain. None, some, or all nodes on the site might fail before repair has taken place. It depends on how much the temperature rises and the length of the exposure. Redundant cooling equipment will reduce the probability of such a site failure.

4.1.6 Node restart

A frequent failure mode is that a node is unable to communicate with the other nodes in the system. This can happen if the node is deadlocked (either in software or hardware), or the operating system or communication has crashed or entered an odd state. Even though parts of the node still are operating the rest of the system observes that the node is unreachable. Since it is not possible to communicate with the node, it is also impossible to tell the node to restart itself so that the failure can be corrected. Two possible solutions to this problem are:

Reset through the communication network: The communication network must provide a mechanism which allows some nodes to reset other nodes independent of the state of the node to reset. This can be done by special reset lines or subchannels in the communication network. To avoid failed nodes accidentally resetting correctly operating nodes, at least two nodes should be required to initialize a reset.

I-am-alive messages and watchdog timers. With regular intervals (e.g. each second) the nodes should send I-am-alive messages to each other (which in any case is necessary for detecting failed nodes). Each time a node receives such a message, it resets a watchdog timer. If the timer is allowed to reach a given value (e.g. two seconds) without being reset, the timer physically restarts the whole node, including hardware and operating system. The watchdog timer itself is a simple timer circuit independent of the rest of the node hardware.

Of the two strategies the second seem to be the simplest for a workstation cluster. Since no commercial communication solutions provides the necessary reset capability. Watchdog timers on the other hand are frequently offered on CPU boards.

4.2 Operating system

The operating system provides the programmer access to the hardware resources, but also restricts accesses which will threaten the system integrity. It is also responsible for allocation and sharing of resources between different users and program executions. The access to the resources is through an abstract interface hiding the actual hardware implementation for the programmer. This interface also eases the portability of application code since the underlying hardware can change without changing the interface. Randell [Ran75] calls this an *abstract machine* (or a *virtual machine*).

An abstract machine defines the facilities available for the user like memory segments, disk storage and text I/O. From a fault-tolerance viewpoint it is important to know their failure semantics.

An abstract machine should not be defined by random. It is important that it is possible to implement the abstract machine on the anticipated target platforms, and even if a facility is implementable, it must be efficient, i.e. there must be low impedance mismatch between the abstract machine facility and the underlying hardware. The abstract machine must also provide the facilities required by the application programs. For the suggested database cluster architecture this includes communication and disk access primitives.

Some work has been done on operating system kernels and abstract machines for database machines and fault-tolerant systems. Wilkinson and Boral [WB87] describes the KEV kernel designed for the Bubba parallel database computer. Borg et al [BBG$^+$89] describes a fault-tolerant operating system with a UNIX personality. The system is implemented through process pairs, checkpointing and use of memory management hardware. The idea was to offer fault-tolerance for existing software without any modifications needed. The problem is that even if OS and HW faults can be masked, it is not able to detect and correct software errors in the application.

The approach taken by the Tandem NonStop kernel [Bar78] is not to hide the fault-tolerance facilities from the application programmer, but instead provide it to his or hers direct control. The mechanisms provided are process pairs, messages and checkpointing.

For a further discussion on abstract machines for a high-availability database server we refer to the article reproduced in Appendix C.

Chapter 5

Data availability strategies

There are four types of resources for a DBMS server that require redundancy to ensure high availability to the DBMS service: information, processing capacity, storage capacity, and access ports.

Information includes the database contents, the database schema, and log and recovery information. The ACID properties of the database system demands durability, i.e. no loss of information stored or updated by committed transactions. Redundancy through replication provides an implement to overcome inevitable hardware errors.

Sufficient **processing capacity** must be available to handle the database workload that the DBMS server is specified to handle. In case of a hard error affecting processing capacity there must be sufficient redundant hardware resources to replace the lost capacity.

Sufficient **storage capacity** must be available to handle the database volume the server is specified to handle, also in the case of a hard error.

The DBMS server must provide enough **access ports** (communication channels) into the server so the clients can issue the required transaction load even in the case of a hard error.

This chapter will discuss strategies for maintaining the information available, even in case of an error, provided that there are enough redundancy for the other resources: processing, storage and access ports.

5.1 General aspects

The shared-disk and shared-everything architectures described in Section 4.1.1 allow any processor to directly access any tuple or disk page in the system. This is not possible for the other architectures listed (e.g. shared-nothing) and an intermediate processor has to be requested for the relevant data. The penalty for a remote access compared with a local one is the additional delay and an extra load on the communication network. The volume of remote accesses affects the capacity of the network and therefore also the cost of it. If the data accessed resides in primary memory the remote access delay is several magnitudes larger than local access. Table 5.1 shows typical latency and bandwidth numbers for disk and memory compared with those for node requests. The remote node accesses do not include disk access latency which eventually must be added. Reading the table from a

Access type	Latency	Bandwidth
	μs	MB/s
Local memory	0.1	100
Local disk	15,000	5
Intra-site node (<1 km)	1,000	10
Inter-site node (1,000 km)	10,000	10

Table 5.1: Typical latency and bandwidth for memory, disk, and remote node accesses. Latency figures are for 100 byte accesses.

performance viewpoint the following observations can be made:

- Local disk access latency is lower than for remote disk accesses, but with less than a factor two.
- Local memory accesses are *much* faster than remote memory accesses.
- Remote memory accesses are faster than local disk accesses!
- The bandwidth penalty is less than the latency penalty. A few larger remote accesses are preferred over many small.

From these observations we see that the placement of the data should be carefully considered so that the hardware resources are utilized as best as possible. We also know that data replicas must have independent failure modes and this sets a restriction on the placement of the data.

Declustering strategy[1] is the generic term we will use for a data placement strategy. Declustering has two aspects:

Distribution: How to spread the database content over the nodes.

Replication: How to maintain multiple copies of the database content placed on different nodes.

Declustering was in [Liv87] originally used as a term for distribution of subsets of tuples over many disks but has in the past years also included replication.

A declustering strategy should ensure:

- Balanced load distribution over the nodes.
- Balanced storage requirements for the nodes.

[1]*Clustering* = locate together, *Declustering* = locate apart.

- Efficient use of the hardware resources.
- Maximum system availability.
- Scalability over a wide range for the following parameters: transaction load, database size, number of clients, and availability.

5.1.1 Information entities

An *Information Entity* (IE) is an atomic unit of information either grouped logically or physically.

Physical IEs	**Logical IEs**
• disk page	• tuple
• disk	• fragment
• node	• relation
• site	• file
• system	• database

The physical IEs hold the information stored on a hardware entity like a disk or a node while the logical IEs are the abstract units of information the software presents to the user.

IEs are the basis for both replication and distribution. *Unit of replication* and *unit of distribution* denotes the IE granularity used for respectively replication and distribution.

A *unit of failure* is an IE where the different instances of the IE have independent failure modes. In our DB cluster architecture sites are the unit of failure with respect to site and some communication failures. Nodes are the unit of failure with respect to disk, memory, processor and the rest of the communication failures.

5.2 Replication

Replication is the task of producing and maintaining multiple logically identical copies of information units. Such a copy is called a *replica*. If one of the replicas is changed, all the replicas should atomically be changed to ensure ACID properties. If X is the unit of replication then an X *replica* is an instance of the replicated IE.

The reason for replication is threefold. First, it improves data availability by providing alternative sources for data if one replica is lost through replica corruption or failure of the access mechanisms. Second, it provides multiple access paths to data increasing the available access capacity. Third, by distributing replicas communication can be reduced through accessing the closest replica, which in the best case can be a local replica.

Unfortunately, there are also drawbacks with replication. Updating multiple replicas gives serious performance penalties and the additional hardware resources required to store and process the multiple replicas increase system cost and aggregated component failure rate. Creating two replicas of data requires twice the storage capacity, i.e. twice the number of disks. Even though replication can reduce the system failure rate by masking disk errors, each individual disk will fail with the same intensity giving twice the number of disk failures over a given time interval.

From the replication point of view the unit of replication is atomic and can not be partially available. If one part of the IE fails, the complete IE is considered to have failed. Take for example *disk replication*. The content of a disk is replicated over multiple disks, each called a *disk replica*. If a block on one of the disk replicas is corrupted and the corruption is detected, the whole disk replica is considered corrupt and therefore unavailable. Replication can also be applied to more that one IE granularity at the same time, e.g. by using both site replication and disk replication (see Section 5.6).

Each replica of an IE must be externally equivalent and contain the same amount of information. This includes explicit information like the data content itself and implicit information like data sequence, relative data address, and other relationships between data items.

A single failure can cause multiple replicas to become unavailable, e.g. a disk failure might cause multiple disk pages to be lost. For fault-tolerance reasons replicas of the same unit of information should have independent failure modes, i.e. not be stored in such a way that one failure will cause the loss of more than one replica. Assuming node failure granularity, storing replicas of the same information unit on different nodes is sufficient for fault-tolerance. For site failure granularity replicas must be located on different sites.

The replicas are either maintained *symmetric* or *asymmetric*:

Symmetric replication treat all replicas identically without giving any replicas special roles. A read operation can access any replica and write operations have to directly update all replicas.

Asymmetric replication give one of the replicas a special role as a *primary replica*. All operations (both read and write) are directed towards the primary replica and serialized there. The other replicas are called *hot stand-by replicas*. When write operations succeed on the primary they can be forwarded to the hot stand-bys and re-done there. There are no need for concurrency control mechanisms on the hot stand-bys, since the operations already have been serialized on the primary.

Symmetric replication is simple and gives a high level of concurrency for read operations. Since we can choose which replica to read locality can be utilized to reduce communication latency and bandwidth. Write operations are more expensive and suffer from potential high conflict rates on hot-spot data. Asymmetric replication is less resource demanding for write operations but provides less concurrency for reads.

In the case of a hard failure all replicas residing on the unit of failure areis considered unavailable (permanently or temporarily). For symmetric replication, read operations must only be directed towards available replicas. Write operations will just ignore the unavailable replica and update the remaining replicas as normal. If it is a temporary unavailability the update can be queued for deferred update when the unavailable replica becomes available again.

In the case of asymmetric replication this is a bit more complicated. If a hot stand-by replica fails, the replica is just ignored. If the primary replica fails, a new primary must be elected and all read and write operations directed to this instead. The process of changing primary is called *replica takeover* or only *takeover*.

To directly access an IE instance X the logical address k_X of the instance must be known. This address is mapped to a physical address a_X by the DBMS. If replication is used, the address should not map to only one of the replicas, but to the set of all available replicas. For asymmetric replication the mapping should also denote which replica is the current primary. For relational databases the logical address k_X can be the primary key of a tuple and the physical address a_X the tuple (node number, disk number, page number).

5.2.1 Location of declustering

One of the decisions that must be taken during the design of a parallel DBMS is where the distribution and replication should be handled. Modern design guidelines favor implementing functionality in smaller, more manageable modules over monolithic designs. Figure 5.1 shows a simplified and generalized layered design of a DBMS. The *application* issues requests to the *logical DB processor* (LDP). The LDP is a collection of components which includes transaction coordinators and execution plan compilers. The LDP converts a high-level request (e.g. an SQL command) into a form the *DBMS kernel* can handle. The kernel uses the *access method* to retrieve, update, insert, or delete records from database files. The *storage server* provides the access method with the disk pages it needs. The storage server requests raw disk I/O transfers from the *device driver*. The hatched horizontal line illustrates the border between logical (above) and physical IEs (below). Above the line the IEs are databases, relations, tuples, and attributes. Below it can sites, nodes, disks, and disk pages be found. The left side of the figure shows examples on where various systems implements replication and distribution. Not all of them are DBMSs: Tuxedo [MAC92] is a TP monitor which can handle replication and distribution on top of a DBMS. Mirroring and RAID are typical operating system or disk controller features which can be used by DBMSs.

5.3 Physical declustering

A physical disk address consists of the tuple (*cylinder number, head number, sector number*). Knowledge of the physical layout of the disk and its access characteristics is valuable for the DBMS for optimizing disk usage. Layers between the disk and the DBMS, like disk controllers and the operating system device drivers, try to hide this information from programmers by only providing an abstracted linear address space of disk blocks. There also exists a mismatch between the size of the physical sectors on the disk surface, the sector size presented by the disk controller, the block size used by the operating system and device drivers, and the logical page size managed by the DBMS. For the purpose of the discussion on distribution and replication this is not important and the mismatch will be ignored. For our discussion it is sufficient to view the disk as a collection of pages with a size matching the page size used by the DBMS. Since the page is the minimum granularity of transfers between disk and primary memory, *byte* and *word* granularity are of little interest and will not be considered here.

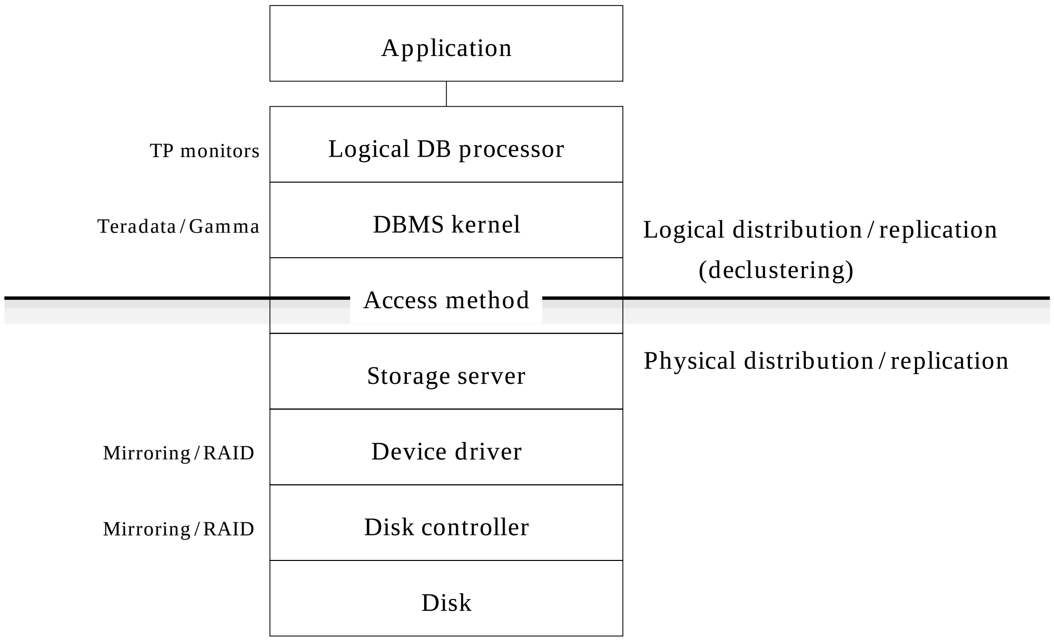


Figure 5.1: Data distribution and replication location.

The reasons for using distribution and replication on a physical level are both performance and reliability. Replicating pages on two independent disks double the number of disk arms being able to read data and thereby increasing the number of disk reads per time unit. The penalty is that new and updated pages have to be written on both locations (mirroring / shadowing [BG88]).

The read and write throughput of a disk is limited. Distributing a relation over multiple disks provides the opportunity to transfer data in parallel between the processor and the disk and thereby increasing the overall node-internal disk throughput (striping [SGM86]). This argument can be used to advocate for many small disks over a single large, but from a reliability viewpoint this is bad. The reliability of a disk drive does not depend on the storage capacity of the drive but can instead be considered constant. If reliability is important few and large disks should be used.

RAID (*Redundant Array of Inexpensive Disks*, [PGK88]) is a family of physical, page level, declustering mechanisms. RAID level 0 offers distribution only, RAID level 1 redundancy only and RAID levels 2 through 5 both distribution and redundancy. The redundancy is either through replication (RAID level 1) or parity (RAID level 2 through 5). Even though redundancy through parity can not be called replication, the reader can consider it as a logical replication scheme. Data reconstructed from other data and parity disks form a logical replica. Section 6.2.1 describes RAID in more detail.

Based on Figure 5.1 we see that physical declustering must be located in either the disk controller (DC), the device driver (DD), or the storage server (SS) (illustrated in Figure 5.2). Seen from the layers above, the DD and DC can hide physical data distribution

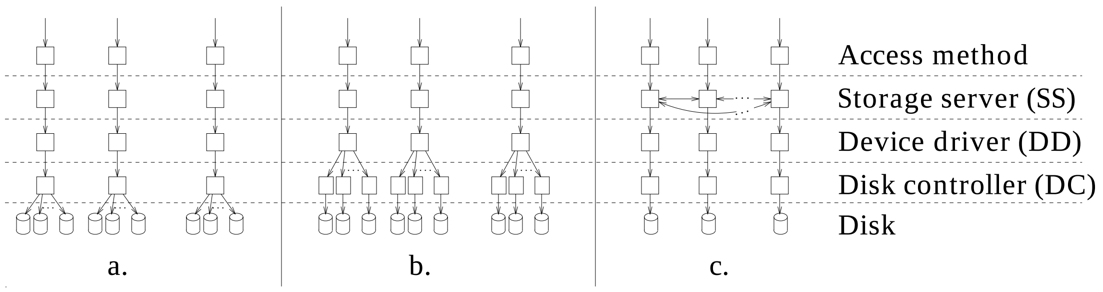


Figure 5.2: Alternatives for implementing physical distribution and replication. (a) in the disk controller, (b) in the device driver, and (c) in the storage server

and replication. The layer above only sees a single, large disk and details on concurrent accesses, error masking, and sparing are hidden. Based on plain single-ported disks and disk controllers, the DC and DD are restricted to distribution and replication over multiple disks on a single node only. Multi-ported disks and disk controllers (e.g. [Kat78]) allow distribution and replication over a few (two) nodes, but they can not be considered commodity hardware and the functionality can be emulated in the SS layer (but with some performance penalty). The DC and DD provides no mechanisms to synchronize concurrent accesses to disk pages and should therefore only be accessed by a single entity in the layers above.

The SS, on the other hand, provides inter-node access to disk pages. The access method manager (AM) requests the local SS for disk page reads and writes. If the page is non-local the SS request the SS having access to the page to complete the operation. Some architectures allow multiple AMs to request the same page and let the SS handle the concurrency control.[2] The SS can also handle caching of remote pages. In short, the SS can provide a virtual shared-disk architecture.

Since the DC and DD solution only can mask disk drive and / or disk controller failures, they are not sufficient for handling node failures and therefore of no use in a system with a node failure granularity. On the other hand, it should be pointed out that they can be combined with any other mechanism to yet increase reliability and in some cases also the performance. Placing the distribution in the DC or DD can potentially degrade the DBMS performance, since the higher layers of the DBMS know more about the access pattern to the data than the DC and DD and can potentially do more optimalizations.

The DC is normally considered to be hardware and the DD a component in the operating system. This means that neither of them in fact are parts of the DBMS and should therefore not be required to be specially made for the DBMS. Using standard DC and DD components improves portability.

Physical declustering does not have any knowledge about the data being stored and can therefore not distinguish between data that must be replicated and data that not necessarily

[2]This concurrency control does not have to be the sole concurrency control mechanism for the DBMS. To the contrary, the page granularity is coarse and can serious affect the performance for transaction systems exposed to hot-spots.

need to be replicated, e.g. temporary files.

Based on this analysis physical declustering will be evaluated based on the assumption that it is located to the SS.

5.4 Logical declustering

In the relational model the IEs are attributes, tuples, relations, and databases. Distributing the data over the nodes based only on locating whole relations or databases to nodes is not sufficient. The relations vary in size and some relations are accessed more often than others, and thus violating the requirement for even load and storage capacity distribution. The solution must therefore be to divide the relations into smaller sub-units, called *fragments*, either based on tuples (*horizontal fragmentation*), attributes (*vertical fragmentation*), or both (*mixed fragmentation*). Since vertical fragmentation is of little interest for load distribution (relations with few attributes can only be divided in few fragments), we will consider horizontal fragmentation only.

A horizontally fragmented relation R is divided into n fragments F_i, one for each node the relation should be distributed over. A tuple t in R is a member of one and only one fragment F_i.

$$R = F_1 \cup F_2 \cup \cdots \cup F_n$$

$$F_i \cap F_j = \emptyset, \text{ for } 1 \leq i < j \leq n$$

The cardinality should approximately be the same for all the fragments for storage capacity balancing. With ideal distribution of access to the data this will also give load balancing. Copeland et al [CABK88] suggest using the usage intensity ("heat") of the data to determine the size of the relations so that each fragment gets the same usage frequency, i.e. the same load.

In the relational model each relation has a *primary key* (PK) which is a non-empty sub-set of the relation's attributes. The primary key uniquely identifies a tuple in the relation. The *partitioning key* (FK) is a similarly defined key used to partition the relation into non-intersecting fragments. A *partitioning function* is applied to a tuple's FK returning the fragment (or fragments) the tuple belongs to. FK will in most cases be identical to PK or a subset of it. The Gamma database machine [DGS$^+$90] provides three methods to partition the tuples among the fragments:

Range partitioning: The partitioning key domain is divided into n ranges and a fragment is assigned to each range (assuming that the domain has an ordering). A tuple belongs to the fragment with the assigned range which the tuple's partitioning key value falls into. The ranges can be defined by splitting the domain into equal sized value ranges or by manually providing the range limits.

Hash partitioning: A hash function $h(FK)$ returning values in the range $[1, n]$ is applied to the partitioning key value. The returned value determines which fragment the tuple belongs to.

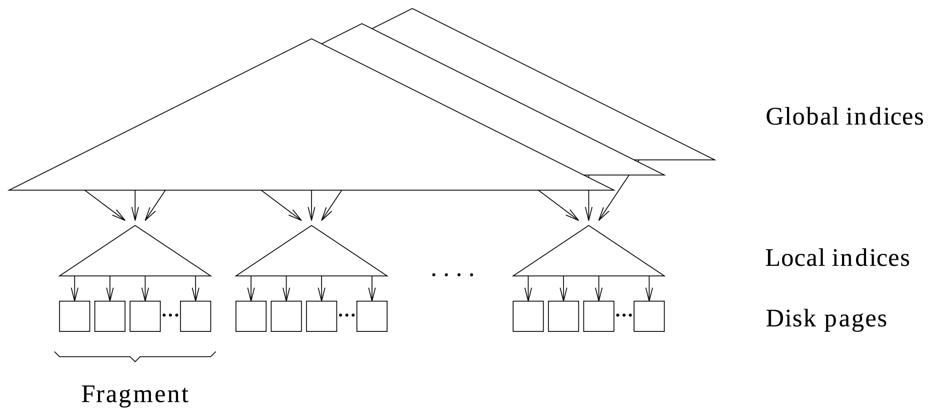


Figure 5.3: A two level index hierarchy.

Round-robin partitioning: Tuples are inserted into the fragments in a round-robin scheme without using a partitioning key.

With the range and hash partitioning schemes it is possible to directly locate the fragment a tuple is stored in if the partitioning key value is known. If the value is unknown or if round-robin partitioning has been used all fragments must be probed to locate the tuple.

The selection of which partitioning scheme that should be chosen depends on the nature of the data (key value distribution, tuple size, relation size) and access patterns (single tuple or range queries, access frequency, access distribution, hot-spots). It should therefore be possible to chose partitioning strategy when a relation is created and later change it if the nature of the data changes.

A more general view of partitioning is taken by [CK89] which defines two levels of indexing. A *global index*[3] is used to locate the home fragment of a tuple and a *local index* directs to the correct disk page inside the fragment (see Figure 5.3). The global index can be maintained centralized by assigning it to a single node. All transactions accessing the relation have to go through this centralized resource which is a potential bottleneck limiting the concurrency and transaction rate scalability. Replicating the global index over some or all the nodes removes this potential bottleneck but at the same time introduces a consistency problem. Solutions to the consistency problem is either to keep the global index static over the lifetime of the relation or to allow discrepancies which is automatically detected and corrected.

Linear Hashing* (LH*) is such a global index [LNS93] based on Linear Hashing [Lit80]. Each node has an approximate knowledge of the current bucket allocation and sends requests to other nodes based on this knowledge. If the request reaches the wrong node, the request is forwarded based on the knowledge of the intermediate node. LH* guarantees that the correct target node will be reached after maximum two intermediate nodes. When the request has been processed by the target node the reply is returned to the requester together with information that can be used to update the bucket allocation information

[3]This should have read *global access method* since it also is possible to use access methods not using indices, e.g. hashing. We will remain faithful to the source and use the original term.

on that node. This means that the replicas of the global index do not need to be strictly synchronized but are instead updated lazy at the cost of sometimes having to forward requests. A similar scheme can also be developed for a range partitioning scheme.

The local index only exists in one replica and can be clustered with the corresponding fragment data. The global index will be an index on FK while the local index is on PK. Both the local and the global index can be traditional index methods like B-trees or hashing but do not need to be identical. If the local and global index is identical they can be combined into a single index structure [CK89].

To generalize the two level index model for round-robin partitioning of tuples, the global index can qualify multiple fragments for tuple locations.

Usually an index structure will have to be updated for dynamic data. The global index can be classified as either:

Static The global index is built when the relation is created and kept unchanged over the lifetime of the relation.

Semistatic The global index can be changed through a database schema change operation, either manually (by the DBA) or automatic tuning operations. The will probably cause many tuples to move to new fragments.

Dynamic A global index change can be caused by ordinary database operations. The change should only cause few tuples to move to avoid delaying the transaction trigging the operation.

There are schemes that do not map onto the two level index model. Aske in [Ask89] describes a distributed, clustered B-tree index with multiple roots and fully distribution of both internal and leaf tree nodes.

5.4.1 Application level declustering

A logical declustering mechanism can be located to either the application, logical DB processor or the DBMS kernel (see Figure 5.1). Handling declustering in the application layer means that the application is alone responsible for sending database requests to the correct nodes handling the data in question and ensuring distributed transaction consistency. To be able to do this the application must know the partitioning information. Application level declustering is outside the scope of this work and will not be considered further here.

5.4.2 Logical DB processor based declustering

The logical DB processor receives database requests from the application. The requests are analyzed and converted into a form the layer below understands. The conversion done by the logical DB processor includes query rewriting, optimalization, and compilation of a high level query language as SQL into intermediate code or executable machine code. It is also possible to envision multiple logical DB processor layers on top of each other, e.g. the application uses a transaction monitor which again accesses an SQL compiler.

The table TABLE1 is range partitioned as:

Key range	Table-Node	Key range	Table-Node
$[0, 100 >$	TABLE1-0	$[500, 600 >$	TABLE1-5
$[100, 200 >$	TABLE1-1	$[600, 700 >$	TABLE1-6
$[200, 300 >$	TABLE1-2	$[700, 800 >$	TABLE1-7
$[300, 400 >$	TABLE1-3	$[800, 900 >$	TABLE1-8
$[400, 500 >$	TABLE1-4	$[900, 1000 >$	TABLE1-9

The query:

```
SELECT *
FROM TABLE1
WHERE K>=120 AND K<340;
```

is rewritten into the following three queries:

```
SELECT *                SELECT *                SELECT *
FROM TABLE1-1            FROM TABLE1-2            FROM TABLE1-3
THERE K>=120 AND K<200;  WHERE K>=200 AND K<300;  WHERE K>=300 AND K<340;
```

which then can be simplified to:

```
SELECT *                SELECT *                SELECT *
FROM TABLE1-1            FROM TABLE1-2;           FROM TABLE1-3
WHERE K>=120;                                    WHERE K<340;
```

Figure 5.4: Rewriting a query into subqueries on a range partitioned table.

If the logical DB processor does the declustering it also decides which node(s) will handle the query and sends the converted request to the DB kernel at that(those) node(s). With the partitioning information it is able to rewrite a single SQL query into multiple queries. If range partitioning has been used and a sequential scan is issued for a limited range, one query is generated for each node with tuples inside the query range (see Figure 5.4). Replicas are handled by generating the same update queries to all replicas of the data item to be updated. Faults are masked by changing the rewriting rules which reflects the available replicas. Dynamic load balancing can be achieved by directing read queries to the fragment replica located on the node with lowest load.

While the application definitively is no good location for handling declustering, the logical DB processor is much better suited. To allow standard DB transaction semantics, the logical DB processor must handle global concurrency control and a distributed commit protocol. The lower levels do not need to know anything about distribution and replication and this allows the DBMS kernel and the other lower levels to be implemented with traditional DBMS techniques. It is even possible to implement a distributed and replicated system using an existing single processor DBMS products in the lower levels.

The database schema is necessary for compiling a query into executable code and this is therefore an obvious location for the partitioning information. This is for example done in Tandem NonStop SQL [Gro88] where the partitioning information is given when an SQL table is created. The disadvantage with using the database schema to store this information is that the fragment and replica information is more dynamic than the other schema information. The schema is traditionally a part of the application and the declustering aspects should be hidden from the application programmer. The programmer should not need to know the actual number of nodes in the system and what the optimal fragment size is. There might therefore be a need for automatic calculation of the number of fragments and the location of them. System upgrades (e.g. adding more nodes) and node faults will also change the number of available nodes which again must result in a change of the declustering information.

This mismatch is even more clear when precompiled access plans are considered (precompiled transactions). Since it is not possible to know at compile time which database tuples to actually access, the access plan must be parameterized. At run-time parameters will give the partitioning key values of the tuples to access. There is no need for retrieval of information about record layouts, access method, data types, and optimalization since all this is compiled into the query based on the database schema. Since the key values are not known at compile time, either the complete declustering information must be compiled into the access plan or the declustering lookup must be done at run time. The first alternative can result in large code sizes for access plans and the need for recompiling them each time the declustering changes.

5.4.3 DBMS kernel based declustering

The last alternative is to place the declustering into the DBMS kernel. The kernel is often the most complex layer of an DBMS and this extra responsibility does not make it any better. It also excludes using DBMSs with no special kernel support (which means most of current products). Despite these disadvantages there are several good reasons to actually implement declustering in the DBMS kernel.

Implementing declustering in the DBMS kernel limits the logical DB processor to send requests to the local DBMS kernel only. The kernel maintains the fragmentation and replication information and forwards the request to the correct nodes. While the logical DB processor has little knowledge about the recovery and concurrency mechanisms that exists in the DBMS kernel, the integration of declustering into the kernel can make better use of these mechanisms. Hvasshovd [Hva92] shows how the integration of logging and recovery with hot stand-by updating will give significant performance advantages. All updates are performed on primary replicas with normal concurrency control. Hot stand-bys are updated by directly shipping log records from the primary to the hot stand-bys and executing *redo*. This allows higher intra-transaction concurrency, lower concurrency control overhead and reduces the communication overhead. The tighter synchronization of the replicas can also speed up both the detection of errors, the takeover process, and the recovery after the failure.

5.5 Availability through replication

There are three ways to achieve improved availability through replication:

Detect errors: If one replica is different from another an error has occurred.

Mask errors: If one replica is lost as the consequence of an error it is possible to mask it by accessing another replica instead.

Self repair: The remaining replicas after a replica loss can be used to reproduce the lost replica on an alternate location reestablishing the fault-tolerance level.

In this work we will investigate masking and self repair leaving error detection to later research.

Availability to DBMS services depend on the availability to at least one replica of each fragment. For the shared-nothing architecture there are several types of errors that will cause the loss of one or more replicas, either temporarily or permanent. An error causing a replica to become unavailable without damaging the data itself (e.g. a communication network failure making the node and residing replicas inaccessible) can cause a temporary replica loss. If the disk surface is damaged, the replicas stored on this disk are probably lost forever.

Even though multiple replicas are employed to achieve high availability to data there are some potential negative consequences of failures that must be taken into account:

Reduced fault-tolerance level: A failure will cause a reduction of the fault-tolerance level until the error has been repaired. In this state the system is more vulnerable to faults that can cause loss of all replicas.

Performance degradation: An unavailable hardware unit means that there are less resources available to process the requests issued by clients. Recovery and repair procedures also use these resources. If this degradation causes the system not to give the specified performance the system can be considered unavailable. Another effect is that a higher load may cause a higher failure intensity and this happens just when the system is most vulnerable.

Temporary unavailability: There is a latency from a fault occurs until it is detected, diagnosed, recovered, and the system is ready to mask the error. In this time period the service will appear unavailable.

It is possible to trade these effects against each other. A fast fault detection mechanism can reduce temporary unavailability but require extra resources for frequently sending *I-am-alive* messages and excessive checking of data and results. These extra resources could otherwise be used for reducing performance degradation and faster repair and recovery. Giving repair maximum priority will reduce the time when the system is vulnerable for double faults, but seriously degrade the performance. By giving client requests maximum priority we ensure service availability during repair, but on the other hand increases the probability of a double fault.

5.5.1 Information loss granularity

A failure can be classified based on the IE that is lost as a result of the failure. These failures have varying granularity and are partially contained in each other. A disk failure can for example also be considered to be both a node and a site failure. We consider the following failures:

A *page failure* is the unability to read or write a disk page. Typically caused by a bad sector on a disk. Detected by the disk controller.

A *disk failure* is the loss of access to data stored on a disk. Caused by failed disk drives or disk controllers. Detected by the device driver.

A *node failure* is the loss of access to all data on a node. Caused by failed hardware components like processor, memory, local communication, and disk controller. Detected by RPC timeouts, watchdogs, or communication sub-system.

A *site failure* is a failure of all or most of the nodes on a site. It is typically caused by power outages or disasters like fire or earthquake. First symptom is that a burst of node failures are detected. When the number of node failures on a site exceeds a certain limit, it is considered to be a site failure.

A *fragment failure* is the unability to access the content of a fragment. A fragment can be stored spread over more than one disk. This can be caused by bad software or transient hardware failures corrupting the physical data structures. This is detected by the database access method.

Figure 5.5 shows the hierarchy of failure granularity. The size of the information loss unit is important for the consequences of the failures. If only a small amount of data is lost, little work is required to repair the damage, only a small fraction of the operations will be affected, and the amount of extra work that must be moved to other nodes is minimal. The effects of a site failure for a two site system is more dramatic. The remaining site will then have to handle all the requests that previously were handled by the two in cooperation. When the failed site becomes available again it must be recovered either by an incremental recovery or a full reload. Both these alternatives mean a significant overhead for the

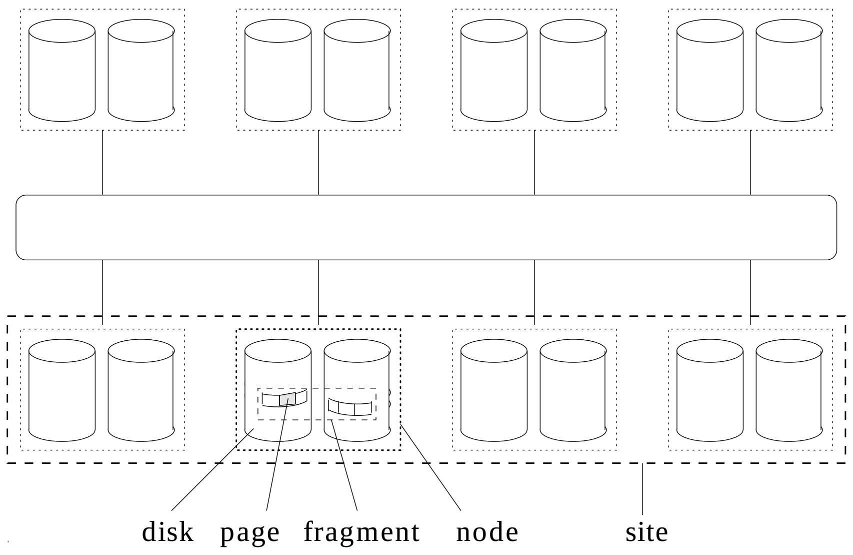


Figure 5.5: Information loss unit granularity for a fragment replicated system.

operating site. Is is also clear that the loss of a large IE results in a state with a higher probability for loss of all replicas than the loss of a small one.

One important prerequisite for fault-tolerance through replication is that the replicas have independent failure modes. That means that the failure causing the loss of a replica should not affect the other replicas of the same IE. Applied to the shared nothing architecture this means that distributing replicas over different sites theoretically gives protection against all types of failures except those crashing all the sites.

It is also possible to have a varying degree of fault-tolerance for different type of failures. Assume that a system has two sites but are maintaining four replicas of all fragments, two at each site. The replicas are organized so that no nodes store two replicas of the same data (require at least two nodes at each site). This scheme can tolerate only one site failure, but three page, fragment, disk, and node failures.

5.5.2 Trading between types of unavailability

As discussed in Section 3.5 there are several reasons for unavailability in a DBMS server which all have different characteristics. The probability and duration of each of these unavailabilities depends on both server design and implementation and the way it is used by the applications, users, maintenance, environment, etc. The designer of the DBMS server primarily gives three contributions to achieve a specified availability. First, the server has an availability figure of its own, second, mechanisms protect against external availability threats, and third, mechanisms are provided for external components to help them contribute to a better system availability.

Some of the types of unavailability are preferable over others and some should by all means be avoided. The server designer can trade between them to achieve as good availability as possible at the lowest possible cost. Below follows a list of guidelines for the DBMS designer.

1. **Avoid ACID violations.** Permanent data loss is violating the durability requirement on the transactions and byzantine responses can violate any of the ACID properties. Most users will consider an ACID violation a catastrophic event. An example of this is recovery of data loss by rolling in a backup that is not completely up to date. Updates done since the backup was produced are lost. For a bank this means that transactions are lost and that account balances might be wrong.

2. **Prefer temporary unavailability over permanent.** If the system becomes permanently unavailable, the system will never fulfill its misson. A car engine being overheated is better immediately halted and fixed rather than driving on resulting in a completely destroyed engine.

3. **Prefer planned unavailability over event triggered.** If the unavailability can be planned, the users can take precautions and prepare for the outage. Planned unavailability can also be scheduled to periods of low usage intensity or less critical hours. For a car several parts are being replaced during regular service calls. These calls can be scheduled at the owners convenience in contrast to sudden breakdowns.

4. **Prefer delayable unavailability over immediate.** A system event (like a component failure) can cause triggered unavailability. This unavailability can either be effective immediately or delayed with an increased vulnerability of additional failures. Unavailability should be avoided at critical moments and it might be better to delay it (if possible) to a less critical point in time. Delaying it too long will on the other hand compromise reliability.

5. **Avoid unavailability requiring manual repair.** Some failures require human interaction before availability is reestablished again, i.e. the failure is outside the domain of failures that automatically can be masked by the system. This kind of unavailability is unpredictable since the system itself can neither guarantee when the system becomes available again nor that repair will be done correctly.

6. **Reduce the length of service interruption.** Reducing the delay from the system becomes unavailable until it is up and running again will increase the total availability. This can be done by keeping standby repair personnel.

7. **Avoid service interruption.** A goal to strive for is to avoid unavailability all together. Modules should be tested well before set into service. The designer should employ mechanisms making it possible to automatically mask errors.

Unfortunately these guidelines are often conflicting and the DBMS designer has to trade between the different sorts of unavailability. Say for example, a computer system employing replicated disks. When a disk fails during busy hours we have two alternatives. Either we can stop the system, replace the disk, and reproduce the lost replica, or we can let the computer run until the evening when usage is low before it is being repaired. In the first case the unavailability during the day causes a lot of annoyed users. The second alternative hide this but increases the risk of loosing both replicas before the repair has been done causing a lot of angry users. Chapter 8 contains an analysis of this and other tradeoffs.

5.5.3 Partial availability

From the definitions in Section 3.3.6, service availability is based on the ability to correctly answer a given percentage of the client requests over a time interval. Even though all replicas of an IE (e.g. a fragment) can be lost resulting in all requests involving these data to be rejected, the system can satisfy the service definition. The single user trying to get access to these data will certainly call the system unavailable but the system as a whole is available. To satisfy the service definition the lost data items must either be small compared to the complete database or rarely accessed. It is therefore desirable to have a unit of failure of fine granularity. The allowed rejection percentage is certainly also important for the availability. A higher rejection percentage makes the system availability less sensitive to loss of data.

5.5.4 Replica inconsistencies

Replica inconsistency is a serious problem with replication. Data corruption also applies to single replica databases, but more copies mean increased probability for such errors, especially if the replicas are updated by software made with design diversity in mind.[4] The source for such inconsistencies is undetected or unmasked hardware or software faults. Detecting and correcting inconsistent replicas are outside the scope of this work.

5.5.5 Skew

One of the goals for a partitioning scheme is to distribute tuples equally among the fragments. Unfortunately this is not always possible. A skewed partitioning key distribution can result in disk overflow and / or processing bottlenecks on the nodes storing the largest fragments. The nodes with the smallest fragments will be running idle part of the time and have free storage capacity. If this skew results in a sufficiently high number of client requests to be rejected the system will be unavailable. Walton et al [WDJ91] describes several types of skew and their effect on database join processing.

An obvious solution to the skew problem is to over-dimension the system making it handle a skew of a given size. The disadvantage is increased system cost.

An alternative solution is to try to avoid skew completely by carefully selection of declustering strategy. If the partitioning key value distribution is known in advance, the partitioning scheme and its parameters can be selected before the data is inserted into the database.

Without this advance knowledge the designer should try to use a general partitioning scheme which perform well for most key value distributions. The round-robin partitioning can distribute the tuples perfectly when it is initially loaded or if the relations only grows. Unfortunately it does not support direct access to tuples when the partitioning key value is known. A dynamic database where tuples are often inserted and deleted can also lead to skew despite a perfect initial distribution.

The unmodified range partitioning scheme require that the DBA defines the ranges for each fragment since there do not exist any range subdivision that will fit all kinds of data.

[4]Some will argue that it is better with an undetected, incorrect database record than two differing records causing confusion. If nobody detects the error, it is no failure!

Even if the data is known when the relation is created and ranges giving an unskewed distribution can be computed, insertions and deletions can change the distribution and gradually cause the distribution to become skewed.

The *Hybrid-Range Partitioning Strategy* [GD90] is an extension to range partitioning which divides a relation into a large number of small logical fragments that do not depend on the number of processors in the system. Each fragment is a small range of the partitioning key domain. For large relations the number of fragments will be much larger than the number of nodes and multiple fragments will be allocated to each node. By dynamically moving fragments around, skew can be adjusted.

A hash partitioning scheme behaves well for most applications but there are pathological cases where nearly all tuples can end up in a few fragments. Hashing schemes handle dynamic data well. A solution similar to hybrid-range partitioning can be used for hashing to reduce sensitivity to the quality of the hash function and statistical variations. As we will see later, these hybrid schemes can also be exploited for achieving better availability.

5.5.6 Importance of repair

Without repair a fault-tolerant system gradually degrades and the ability to withstand faults deteriorates. After the first non-transient system failure a system without repair will not be able to provide services any more. In the context of replicated systems, *replica repair* is the reproduction of a replica after it has been lost. MTTR for replica repair is the average time from a replica becomes unavailable until it is ready to be accessed again. Low MTTR values means a low probability of double faults leading to system unavailability.

Manual interaction from human operators will increase the MTTR unnecessarily if it already is redundant resources available to reproduce the lost replica. *Self repair* is the process where the system automatically and without delay initiates a reestablishment of the fault-tolerance level. Self repair for declustered database systems has previous been proposed by Hvasshovd et al [HST91a, HST$^+$91b, Hva92] and Chamberlin and Schmuck [CS92].

After a transient error causing loss of data, self repair can compensate the loss immediately by recovering or reproducing the lost data on the same location, i.e. on the same node assuming a node failure granularity. For a permanent HW error this is not possible since access to the failed unit is lost. For these types of failure there are three self repair strategies:

Await HW replacement: Wait until the failed node has been replaced by maintenance personnel. When the new node has been detected and reintegrated, the replicas are reproduced on the new node. MTTR is the sum of manual replacement and replica reproduction time. For most maintenance organizations the manual repair will dominate giving repair times between a few hours and one day.

Distributed spare: After a failure the lost replicas are reproduced on spare capacity on the remaining nodes in the system. As long as there is enough spare capacity the MTTR is just the time it takes to automaticly reproduce the lost replicas, typically less than one hour. Since the load of the failed node now must be served by one or more of the other nodes the system capacity is degraded. If there is insufficient spare capacity to provide full availability, the system can set priority on the transaction classes and the data needed by the transactions. The data with the highest priority can be repaired first delaying the replication of the lower priority data. This technique applied to mainframe systems has been discussed by Brooks [Bro85].

When the node has been replaced, the replicas are moved back to the original location. The the spare capacity must be reclaimed on the nodes so they are ready to handle new failures. The load balance is also brought back to normal and symmetry restored. Compared with the first alternative, the data must be copied twice instead of only once.

Dedicated spares: This alternative is similar to the distributed spare strategy but instead of spreading spare capacity over active nodes, the spare capacity is left on a few passive nodes which act as spare nodes. They do not take part in normal operations but after a node failure one of the spares is automatically set to replace the failed one. All lost replicas are reproduced on the spare node. When the failed node is replaced it becomes a spare node and the data distribution is kept as it is. Thus data need only be copied once for each failure.

The two strategies for using spares provide immediate replica repair which increases the overall availability and reduces the need for corrective repair.

Self repair can either be done *off-line* or *on-line*. Off-line repair executes exclusive without any transaction conflicts. All processor, disk and communication resources can be used for the repair reducing the copy time to a minimum. The disadvantage is that the system is unavailable during the repair.

On-line self repair complicates the production of consistent replicas. We will assume that concurrency control mechanisms allowing on-line production of copies without blocking transaction access are used [HST$^+$91b]. Since client requests are processed during the repair, only unused capacity can be used for self repair. This prolongs the self repair time and increases the probability for a double-fault.

5.6 Multi-site replication

5.6.1 System and database replication

The main motivation behind using multiple sites is tolerance against site failures like natural disasters or major operator mistakes. The traditional mechanisms provided have either been *system pairs* or *database replication* where each site is an independent running computer with a complete copy of the database [Gue91]. System pairs is a system replication scheme where a complete computer system is replicated on another site. This is an asymmetric replication strategy where a primary system serves transactions as long as it is operating. In the failure-free situation the primary keeps the hot stand-by up to date by sending a stream of journaling records to it. The hot stand-by receives the records and applies them to its local replica of the data. If the primary fails, the hot stand-by takes over as primary and all transaction requests have to be rerouted to it. A hot stand-by does not have to update its copy of the database immediately, but can instead defer them and apply the changes in larger batches. This means that there will be an inherent delay from a failure is detected until the hot stand-by has been notified, has applied the outstanding updates, and then is ready to provide service. Some configurations even demand manual operator actions for a takeover to take place.

When the original primary becomes available again the database is recovered and resynchronized with the current database content. Eventually a takeover can be done so the old primary gets the primary role again.

Database replication is similar to system pairs but instead of replicating a physical IE, the logical IE is replicated. This allows more sophisticated configurations where one system can store hot stand-by replicas for many other sites or two systems can mutually store hot stand-by replicas for each other.

One question to ask is whether system and database replication is sufficient as the only replication mechanism for user data. There are three main arguments against this approach:

Long takeover delays: For large configurations using site replication having numerous disks, processors, and communication lines, a takeover is a major operation involving actions of all units on the sites, communication reconfiguration, notification of client computers, and eventually manual operator actions. Together with the delay caused by making the hot stand-by up to date this means that a takeover has a major service interruption (in the order of minutes). This penalty is significantly larger than the unavailability following a takeover in single-site fault-tolerant computers (in the order of seconds).

Reduced fault-tolerance level: If a whole site becomes unavailable when only a small component has failed (e.g. a node), a complete replica of the whole database is unavailable, i.e. the fault-tolerance level for the whole database is reduced with one. For a large system this is much more severe than only getting the fault-tolerance level reduced for the data items lost.

Assume a system with two sites and N_s nodes at each site. Let τ_{fn} be the MTTF for a node and τ_r is the MTTR for any failure. The MTTF for a site τ_{fs} is then $\tau_{fs} = \tau_{fn}/N_s$. The $\text{MTTF}_{\text{pair}}$ of a pair of failfast module is:

$$\text{MTTF}_{\text{pair}} = \frac{{\text{MTTF}_{\text{module}}}^2}{2\text{MTTR}_{\text{module}}}$$

Using this we can compute the system MTTF for respectively a site replicated system (T_s) and a node replicated system (T_n):

$$T_s = \frac{{\tau_{fs}}^2}{2\tau_r} = \frac{(\frac{\tau_{fn}}{N_s})^2}{2\tau_r} = \frac{1}{{N_s}^2}\frac{{\tau_{fn}}^2}{2\tau_r} \quad (5.1)$$

$$T_n = \frac{1}{N_s}\frac{{\tau_{fn}}^2}{2\tau_r} \quad (5.2)$$

Comparing the two equations we get a difference in system MTTF with a factor of N_s in the favor for node replication, i.e. the difference increases with the number of nodes on each site.

Expensive self repair: The purpose of self repair is to reestablish the fault-tolerance level as soon as possible to reduce the time gap when the system is exposed to double-failures. To mask a site failure the complete database content on that site must be copied to sites where it si *not* already stored. This is clearly possible through having a spare system on an independent site but it is not likely to be justified from a cost perspective.

Another aspect is the huge volume of data that has to be transferred between sites to reestablish a new database copy. Either the network interconnecting the sites must be of extremely high capacity or a long copy duration is unavoidable. One is tempted to ask whether the self repair will complete before manual repair has finished or not. In most cases self repair of system and database replicated systems therefore seems not to be a good idea.

Site-takeovers should therefore be avoided whenever possible and other means used to handle node failures. The traditional solution is to have an additional replication inside each site and thus increasing initial system cost.

The main advantage of system and database replication is that each site is fairly independent and can be maintained by separate maintenance organizations reducing the chance of failures caused by operator errors. The site independence also opens up for on-line software maintenance and major hardware changes by updating one site at a time while other sites provide service.

The loose interaction between the sites also allows for heterogeneous sites, i.e. sites of different design and manufacture. This improves the assumption of independent failure modes between the copies of data and thus improves the overall reliability of the system.

On the other hand, the need for double replication increases the system cost significantly (e.g. four copies of data instead of two). Two different replication methods also mean higher system complexity leading to higher cost, more software errors, and worse reliability.

Another argument against system replication is the low utilization of the processing capacity during normal operation. Even though the primary might be saturated with transaction requests, the capacity on the hot stand-by site is only used to keep the hot stand-by up to date and can not be used to offload some work of the primary. Database replication can avoid this if multiple databases are stored in the system.

Two examples on such systems are Tandem *Remote Duplicate Data Facility* (RDF) [Gue91] and IBM *Extended Recovery Facility* (XRF) [IBM87] which are database replicated systems which work with their respective database products.

5.6.2 Multi-site fragment replication

The HypRa declustering strategy proposed by Hvasshovd et al [HST91a, HST^{+}91b, Hva92] combines multiple sites with *fragment replication* and is an alternative to system and database replication. Fragment replication allows single fragment replica takeovers instead of full site takeovers.

While transactions being run on a system replicated configuration are executed on the primary site only, fragment replicated configurations must run transactions globally over multiple sites. Since all the primary fragment replicas no longer are located on one site only, one transaction might have to access more than one site.

Thus, multi-site fragment replication requires a high degree of interaction between the sites. Compared with system replication this sets higher demands on the communication between the sites both with respect to capacity, latency and redundancy. The development of high capacity switched communication technologies (e.g. ATM) as described in Chapter 4 may provide the necessary foundation for implementing multi-site fragment replication.

The consequence of communication delays

Physical limitations (speed of light) set a lower limit on the delay of messages sent between nodes. The speed of light through optical fiber is approximately $2 \cdot 10^8$ m/s. An absolute minimum round-trip time of a dialog between two sites located 1000 km apart (e.g. Tromsø–Oslo) is 10 ms which is a significant delay compared with internal communication latencies for state-of-the-art parallel computers (1–100 μs). The distance between two points on the earth which is located as far apart as possible (pole to pole) is approximately 20,000 km following the surface of the earth giving a 200 ms round-trip time. Satellite communication will have round-trip times of at least 480 ms for a single hop. Multiple hops are required to reach an arbitrary point on the earth.

For system and database replicated systems the transfer of journal records from the primary to the hot stand-by can be done outside the time critical execution path of a transaction and communication delay will not affect transaction execution latency. On the other hand, a fraction of transactions running on a multi-site fragment replication configuration will have to access multiple sites inside the time critical execution path. For those transactions the communication delays of inter-site messages will be a part of the total transaction execution time. For typical transaction processing systems demanding sub-second response times, surface communication (electrical or optical fiber) between reasonable distant sites will be acceptable. For those systems communication delay will be dominated by disk and queuing delays.

In cases when long delays are acceptable from the applications point of view, they have consequences for database internals. Aggregate processing requirements (number of disk accesses or processor cycles) for each transaction does not increase with increasing communication delays, but so does the total time each transaction locks the resources it uses (communication buffers, process slots, memory, database locks). This means that the resource demands increase and the probability for resource contention grows.

Distance between sites

Since a long distance between sites clearly affects the overall performance of transaction processing it is tempting to locate the sites close to each other. Unfortunately this is in conflict with the availability goal. The reason we used for multiple sites was to reduce the chance of a single event causing all sites to fail. Both power and communication failures, flood, and earthquakes are likely to affect whole regions, not only the location of a site.

The distance between sites also affects the way the client computers are connected to the database server. For a single site configuration, the clients connect to this site for service.

If the connection is lost, either the server has become unavailable or the communication channel is broken. A client should try to reestablish the connection through an alternative communication path.

For a system or database replicated configuration a client connects to the primary site (which might be the most remote site for the client). If the primary becomes unavailable, the client must then establish a connection to the hot stand-by taking over as primary. Since all clients will be doing this just after the failure, the workload will peak. This only adds to the workload required by catching up the transactions being queued during takeover processing. If the sites are located far apart, a client can not be close to all sites and will — depending on the current primary — have to communicate over long distances causing long delays.

Multi-site fragment replicated configurations allow a client to connect to the closest site (or any other site) and might outweigh the penalty of long communication delays internally in the database server. If a site fails only those clients connected to this site need to establish new connections. Clients can reduce the delays caused by a site failure by always keeping connections established with two sites and using whichever is available.

The discussion above assumes that full DBMS service should be available as long as at least one site is available. This assumption is based on the fact that systems with few (two) sites are the most sensible configuration from a cost efficiency viewpoint. Two sites are sufficient for handling natural disasters and maintenance failures. One exception here is military systems where a high redundancy is crucial for service during combat.

Communication cost is limited to connecting only two sites. Housing, maintenance, and security cost is mainly a function of the number of sites, less of the amount of hardware at the sites. When employing more than two sites it is possible to design multi-site declustering schemes that require that more than one site are operable for service to be available. For the rest of this work we will assume that one site should be sufficient to provide DBMS services.

If this assumption holds a few observations can be made. First, each site must store the complete database, and second, the number of replicas of a data item will be greater or equal to the number of sites.

Even though the HypRa declustering (HDc) strategy was the first declustering strategy proposed for multi-site fragment replication, other proposed declustering strategies actually provide the necessary characteristics to be multi-site declustering strategies. *Mirrored declustering* (MDc) [CK89] and *Chained declustering* (CDc) [HD90] are two such strategies that will be discussed together with HypRa declustering and *Minimum Intersecting Sets declustering* (MISDc). MISDc is a novel declustering strategy proposed in this work.

To conclude this section we can summerize the advantages of multi-site fragment replication: minimal number of replicas to maintain, site unavailability can be masked, node failures can be masked without a significant reduction in processing and fault-tolerance level, and most important in our context, self repair through sparing is reasonable.

Chapter 6

Existing declustering strategies

This chapter describes existing multi-site declustering strategies. Several declustering strategies have been described in previous work but only RADD and HypRa declustering are proposed as multi-site. As this chapter will show, single-site declustering strategies can also be adapted to multi-site use.

The chapter will start with general considerations about the declustering strategies and then proceed to describe and analyze the individual strategies.

6.1 General considerations

Before describing the specific declustering strategies we will take a look on the parameters used to characterize them. Using these parameters it is easier to precisely describe them and analyze their behavior. We will also list criterias that can be used to compare the strategies against each other. The symbols used in this chapter are listed in Appendix A.

6.1.1 Assumptions

- We are studying one large relation distributed over all nodes. For more than one relation the methods described can be applied for each relation, one by one. Combining the knowledge of multiple relations can potentially give better results but this is not considered here.

- There is no data skew and all initial fragments are of the same size. For some datasets this hold, but in the case it don't we assume that fragment sizes can be dynamically adjusted so they give a balanced load.

- All sites have the same number of nodes. We have full control on allocation of nodes to sites. Each time a node is added to one site we have to add a node to all other sites.

- One fragment replica belong to one database replica. Initially a database replica is complete but failures may cause it to become incomplete. One node stores fragment replicas from one database replica only. This causes a node failure to only affecting one database replica.

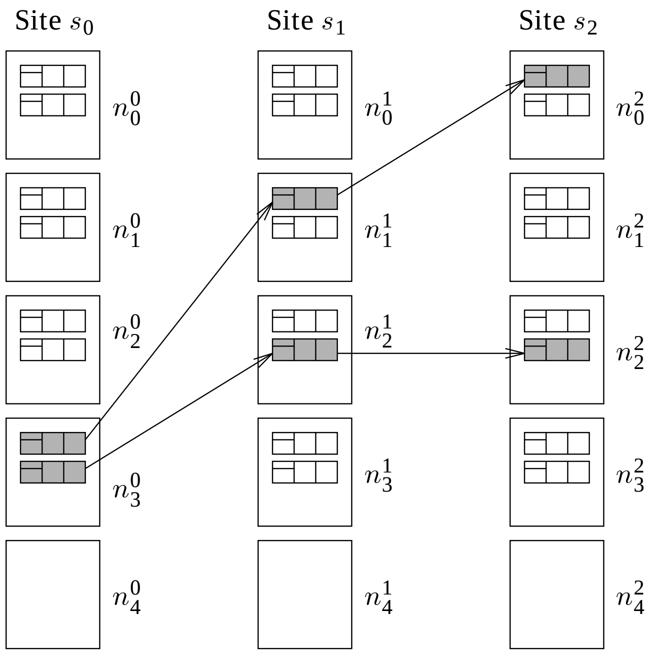


Figure 6.1: A system with 3 sites and 15 nodes. One node on each site is a spare (nodes n_4^0, n_4^1, and n_4^2). Each site stores a complete replica of the database which is partitioned into eight fragments. Each node stores two fragment replicas. The first fragment stored one node n_3^0 is replicated on node n_1^1 and n_0^2 (indicated by arrows).

Anyway, neither of the assumptions significantly favor one declustering strategy over another but where this occurs remarks are made in the text.

6.1.2 Characteristic parameters

The declustering strategies have the following characteristic parameters:

Number of sites S**:** Number of sites.

Number of nodes N**:** Total number of nodes including spares.

Active nodes per site N_s**:** Number of nodes per site excluding spares.

Spare nodes per site N'_s**:** Number of spare nodes per site. The relationship between S, N, N_s, and N'_s are given by: $N = S(N_s + N'_s)$.

Number of replicas R**:** Number of replicas of a relation.

Number of fragments F**:** Number of fragments the relation is partitioned into.

Replica fanout Z_r**:** The (average) number of nodes belonging to one database replica storing common fragment replicas with *one* node in another database replica. Figure

6.1 shows a system with $S = 3$ sites, $N = 15$ nodes, $N_s = 4$ active nodes per site, $N'_s = 1$ spare node per site, $R = 3$ database replicas, and $F = 8$ fragments. Each site constitutes a database replica. Since the fragments stored on node n_3^0 is distributed over 2 nodes both on site s_1 and s_2 the replica fanout must be $Z_r = 2$.

Takeover fanout Z_t: The (average) number of nodes taking over the transaction processing load immediately after a node failure. Assuming the configuration in Figure 6.1 and that the load can be evenly distributed over the remaining replicas of lost replicas, the takeover replica must be $Z_t = 4$.

Refragmentation fanout Z_f: The (average) number of nodes the fragments stored on a failed node is redistributed over after self repair. If node n_3^0 fails and the two lost replicas are reproduced on the spare node n_4^0 the refragmentation fanout is $Z_f = 1$. If the lost fragments are subfragmented into three small fragments and distributed over node n_0^0, n_1^0, and n_2^0 the refragmentation fanout becomes $Z_f = 3$.

Site fanout Z_s: The number of sites refragmentation is activated on when a node fails. If the refragmentation described above only is activated on site s_0 the site fanout becomes $Z_s = 1$. On the other hand, if the refragmentation is done on all the sites, the fanout becomes $Z_s = 3$.

6.1.3 Evaluation criterias

The declustering strategies will be evaluated based on the following criteria:

Unavailability U: Fraction of service time the system is not providing service.

Storage efficiency: The fraction of the storage required by the declustering strategy that a non-redundant storage would have used for the same data volume. The efficiency range from 0 (infinite storage needed) to 1 (no redundancy). E.g. if the efficiency is 0.40, redundant information constitutes 60% of the total storage volume used.

Range stride Δ_R: Average continuous fraction of the partitioning key value domain range assigned to fragments located on different nodes. The range stride varies from 0 to 1.

Fragment stride Δ_F: Average number of continuous fragments in the partitioning key value domain located on different nodes. The relationship between range and fragment stride is given by $F \cdot \Delta_R = \Delta_F$. The value should be as close to the total number of nodes to achieve maximum parallel execution of queries.

Self repair support: What kinds of self repair does the scheme support (dedicated or distributed sparing)?

Planned repair support: Does the scheme provide planned repair (yes or no)?

Homogeneity: If no nodes have a role different from the other nodes the scheme is homogeneous (yes or no).

Single site service: Can the system provide service if only one site is available when $S \geq 3$ (yes or no)?

Simplicity: How simple is the scheme? A complicated scheme adds to the software complexity and the number of software errors (++=very simple, +=simple, 0=normal, −=complicated, −−=very complicated).

6.2 Physical declustering

RAID and RADD are both page-based declustering strategies which provide increased disk I/O throughput and improved availability. RAID is a disk array architecture accessible by processors directly attached to the disks. RADD is an application of RAID techniques for distributed system and is therefore the most interesting of the two in this study. RADD is described by Stonebraker and Schloss in [SS90] which this analysis is based on. The analysis will first look into the original RAID architectures and will then more specificly describe RADD.

6.2.1 RAID

The RAID term originates from University of California, Berkely. RAID was originally an acronym for *Redundant Array of Inexpensive Disks* but has lately been changed to *Redundant Array of Independent Disks*[1]. RAID is an organization of multiple small disk units (PC / workstation technology) into a single large logical disk.

The first work on parallel disks was for supercomputer applications. In [BD83] Boral and DeWitt suggest this architecture for database machine applications. Other work includes [Kat78, Kim86, SGM86, BG88, GHW90]. This overview is based on [PGK88, LK91, CLG$^+$94] which structure previous work into a framework with common terminology.

The motivation for RAID is higher performance, better reliability, and a lower price / performance ratio compared to single large disk unit with the same storage capacity. Multiple disks will have many independent heads which can do independent seeks and parallel disk I/O. Small disks are produced in high volumes for the PC market with low prices but have unfortunately showed low reliability figures. According to the manufacturers' specification this has improved in the last years and the rated MTTF has improved with an order of magnitude. Another problem with disk arrays is that multiple units have a worse system MTTF than a single disk ($\text{MTTF}_{\text{system}} = \text{MTTF}_{\text{disk}}/n$) where n is the number of disks). The solution to this problem is redundant disks. How this redundancy is organized is what mainly differentiates the RAID levels from each other.

The RAID papers describe seven different RAID schemes denoted RAID level 0 through 6. Figure 6.2 illustrates these six RAID levels.

RAID level 0

RAID level 0 is identical to *disk striping* [SGM86]. This level was not listed in the original RAID paper [PGK88] and has later been added. It should be questioned if RAID level 0

[1]Due to current low sales volumes of RAID systems they tend to be more expensive than conventional controllers. Compared with the old mainframe disks they are inexpensive, but not with respect to current micro-computer single disk solutions.

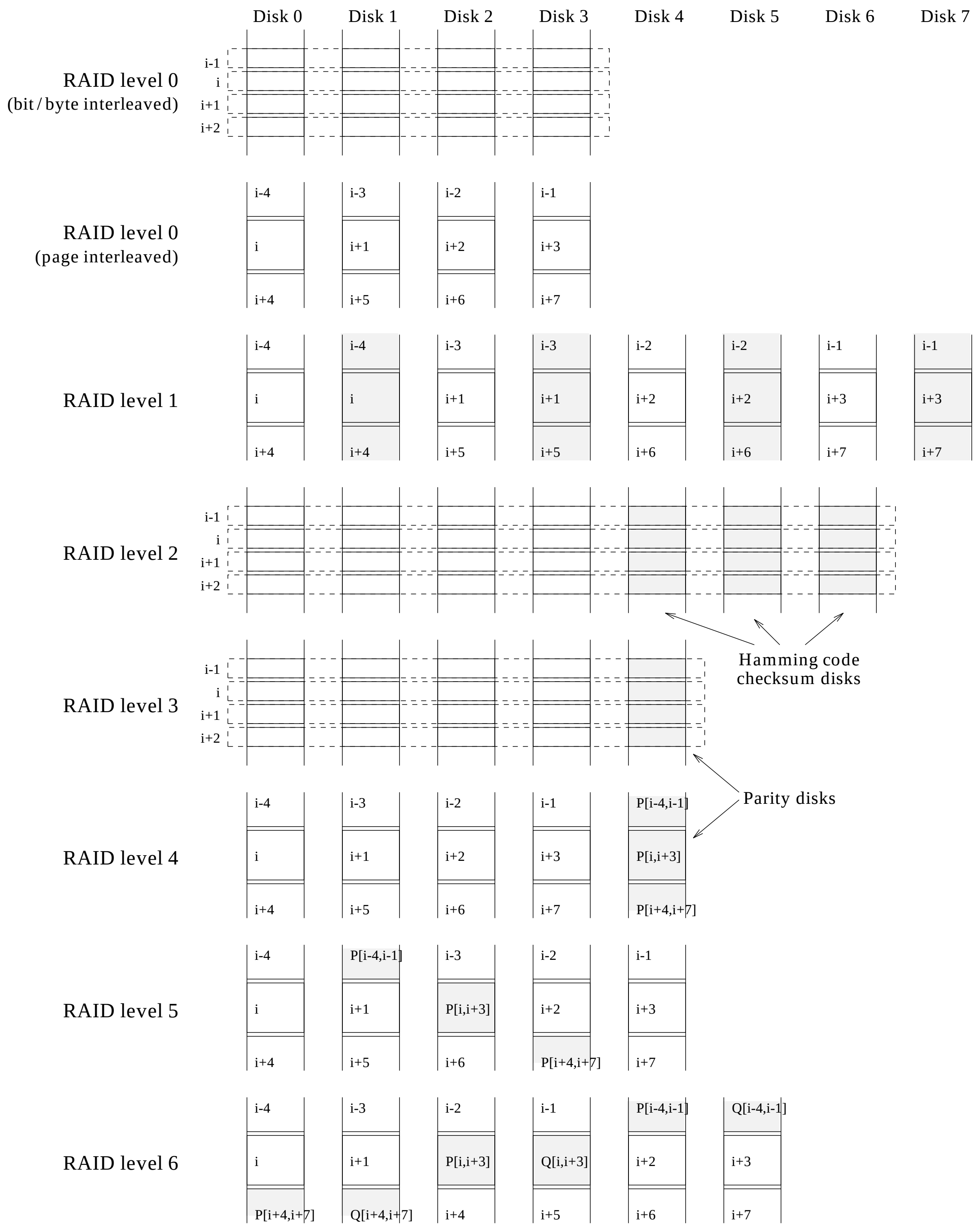


Figure 6.2: RAID level 0 through 6. Logical disk pages are enumerated $i-4$ through $i+7$. Dashed frames group together smaller units from multiple disks into a full logical page. Redundant disk pages are shaded. $P[i,i+3]$ and $Q[i,i+3]$ denotes parity page for logical page i through $i+3$.

can be included among the RAID levels since it contains no redundancy.

Data is interleaved over the G disk in a group in a round-robin fashion, either *bit*, *byte*, or *page interleaved*. The interleave unit says what unit of information is to be retrieved from one disk before retrieving from the next disk in a round-robin sequence.

Bit and byte interleaved RAIDs must read and write *all* disks to transfer a logical disk page. Since the transfer can be done in parallel the I/O rate is potentially high for a single task accessing big chunks of data sequentially. This fits well in a supercomputing environment but is poor for typical transaction processing systems where disks accesses are small, random, and concurrent.

A page interleaved RAID increases the granularity for the interleaving and ensures that the minimum unit of transfer between disk and memory (i.e. a page) is not spread over multiple disks. Independent transfers of pages mapping to different disks can therefore be done in parallel providing performance suitable for transaction processing.

RAID level 1

The RAID level 1 scheme is a well known and widely used method for achieving higher availability. Depending on the implementation it is also called *Mirroring*, *Shadowing*, and *Disk duplexing*. See for example [BG88]. RAID level 1 stores, bit for bit, identical data on two (or more) disks, one for each replica. When a page is to be read any of the disks can be accessed (i.e. independent parallel reads are possible). When a page is written all disks must do the write.

RAID level 2

RAID level 2 is an extension to bit or byte interleaved RAID level 0 where extra disks are provided to store a *Hamming code checksum* [Ham50] of the data stored. With G data disks and C checksum disks one bit from each of the $G + C$ disks compose a $G + C$ bit word where all single bit errors can be corrected and double bit errors detected. With 10 data disks ($G = 10$) the number of checksum disks must be 4 ($C = 4$), i.e. a total of 14 disks. This is identical to the method used by error correcting RAM.

Since RAID level 2 is bit interleaved, all disks must be accessed at each read and write operation (also the checksum disks) making it unsuitable for transaction processing environments.

RAID level 3

Current disk technology includes both error detecting and correcting codes internally, and each individual disk is able to detect whether it can recover the data written to the disk surface or not. If the disk logic has failed no data is returned at all, and this can be detected with a timeout. The probability of returning wrong data from the disk is negligible. This means that bits read from a disk are either “0”, “1”, or “failed” which differs from RAM where each bit is either “0” or “1”. Hamming codes are suitable for RAM error correction and must use multiple check bits both to detect that something has failed, which bit(s) that have failed and the correct value of the corrupted bit(s). Since disks are able to flag errors themselves, a disk array is able to detect and locate a bit error without any extra checksum disks. When the bit position is known a single checksum disk is all that is necessary to correct a single bit error.

RAID level 3 is such a bit or byte interleaved disk array with a single checksum disk. The checksum disk stores the parity of the content of the other disks and is therefore called the *parity disk*. The parity is constructed by taking a bitwise *exclusive or* operation (XOR) over the corresponding bits of the data disks. When all disks operate correctly only the data disks need to be accessed when data is read. To write data both the data disks and the parity disk must be written to.

When one of the data disks has failed, the lost data can be reconstructed by taking an XOR between the remaining data disks and the parity disk.

RAID level 4

RAID level 4 differs from level 3 in that it is page interleaved instead of bit or byte interleaved. This accommodates for independent parallel page reads with one disk operation for each read. Writing is a bit more complicated since the parity disk contents is a function of the contents of all data disks. The data page to be changed can simply be overwritten. The parity page contents on the other hand is computed by taking a bitwise XOR of the old parity page, the old data page, and the new data page contents. It seems necessary to read two disk pages and write the same two pages back to disk again for each single write operation to the disk, a total of four disk accesses. Since most transaction processing operations are page modifications, the data page is most likely to already be in memory when the writing is executed. Each write operation will therefore cost three disk accesses. The read of the parity disk can also be hidden by smart prefetch algorithms.

The disadvantage with RAID level 4 is that all write operations have to access the parity disk reducing the potential for parallel operations. All small write operations must update the parity disk in addition to one of the data disks. Having G data disks and only doing write operations the parity disk will see G times more accesses than the data disks.

RAID level 5

To avoid the parity disk bottleneck, RAID level 5 avoids localizing the parity task to a single disk. The parity pages are distributed over the disks in a round robin scheme such that all disks get the same load. Several different strategies for placement of parity are analyzed by Lee and Katz in [LK91].

RAID level 6

RAID level 2 through 5 can only mask a single disk error. RAID level 6, which was proposed by [BM93], adds a second checksum disk per disk stripe. The two checksum pages per stripe contain a Reed-Solomon error correcting code able to mask two concurrently missing disk pages in a stripe. Otherwise the scheme is similar to RAID level 5 with checksum pages distributed over the disks in the array.

RAID	interleaving	#	Storage	read load		write load	
level		disks	efficiency	ok	failure	ok	failure
0	bit / byte	G	1	1	–	1	–
0	page	G	1	$\frac{1}{G}$	–	$\frac{1}{G}$	–
1	–	$2 \cdot G$	$\frac{1}{2}$	$\frac{1}{2G}$	$\frac{1}{G}$	$\frac{1}{G}$	$\frac{1}{G}$
2	bit / byte	$G + C$	$\frac{G}{G+C}$	1 / 1	1 / 1	1 / 1	1 / 1
3	bit / byte	$G + 1$	$\frac{G}{G+1}$	1 / 0	1 / 1	1 / 1	1 / 1
4	page	$G + 1$	$\frac{G}{G+1}$	$\frac{1}{G}$ / 0	$\frac{2}{G}$ / $\frac{1}{G}$	$\frac{1}{G}$ / 2	$\frac{1}{G}$ / $\frac{2G-1}{G}$
5	page	$G + 1$	$\frac{G}{G+1}$	$\frac{1}{G+1}$	$\frac{2}{G+1}$	$\frac{3}{G+1}$	$\frac{3G+1}{G(G+1)}$
6	page	$G + 2$	$\frac{G}{G+2}$	$\frac{1}{G+2}$	$\frac{2G+1}{(G+1)(G+2)}$	$\frac{5}{G+2}$	$\frac{5G+3}{(G+1)(G+2)}$

Table 6.1: Properties of RAID level 0 through 6.

Writing a disk page results in a total of six disk I / O's, three reading the old data page and the two checksum pages plus writing them back again. If the writing is caused by an update the data disk page probably still is in the disk buffer after the read and we end up with five disk I / O's.

Discussion

A summary of the seven RAID levels are shown in Table 6.1. The "storage efficiency" column shows the fraction of the total storage available for user data. The two "read load" columns show the average number of disk accesses per disk that one logical page read causes when no disks have failed and when one data disk has failed respectively. RAID levels 2, 3, and 4 show both the numbers for the data disks and the parity disks separated by a " / ". The same applies to the "write load" columns except showing disk write numbers. It is also possible to construct other combinations of interleaving and RAID level (e.g. page interleaved RAID 2) but they are not discussed in the original RAID papers and will not be discussed here either.

To decide the best RAID level to use in a transaction processing environment we want to compare them quantitatively. Transaction processing typically has a majority of randomly distributed single disk page accesses. We will assume there are twice as many reads as writes and that all writes on a page are preceded by the page already being read.

Three metrics to compare RAID levels over are *storage efficiency*, *speedup*, and *speed efficiency*. Storage efficiency is the fraction of the disk storage that is available for user data. Speedup is the number of times faster the RAID level is compared with a single disk. In this context *speed* denotes the average number of disk accesses per time unit. Speed efficiency is the speedup divided by the number of disks used. As performance degrades when a disk has failed it is also interesting to find the speedup and efficiency in this situation.

Table 6.2 shows speedup and speed efficiency of the RAID levels compared with a single

RAID level	interleaving	# disks	speedup		speed efficiency	
			ok	failure	ok	failure
single disk		1	1	–	1	–
0	bit / byte	G	1	–	$\frac{1}{G}$	–
0	page	G	G	–	1	–
1	–	$2 \cdot N$	$\frac{3}{2}N$	N	$\frac{3}{4}$	$\frac{1}{2}$
2	bit / byte	$G + C$	1	1	$\frac{1}{G+C}$	$\frac{1}{G+C}$
3	bit / byte	$G + 1$	1	1	$\frac{1}{G+1}$	$\frac{1}{G+1}$
4	page	$G + 1$	$\frac{3}{2}$	$\frac{3G}{2G+1}$	$\frac{3}{2(G+1)}$	$\frac{3G}{(2G+1)(G+1)}$
5	page	$G + 1$	$\frac{3}{5}(G + 1)$	$\frac{3G(G+1)}{7G+1}$	$\frac{3}{5}$	$\frac{3G}{7G+1}$
6	page	$G + 2$	$\frac{3}{7}(G + 2)$	$\frac{3(G+1)(G+2)}{9G+5}$	$\frac{3}{7}$	$\frac{3(G+1)}{9G+5}$

Table 6.2: Speedup and speed efficiency of RAID level 0 through 6 for a transaction processing type application relative to a single large disk when there are twice as many reads as writes. Larger numbers are better.

disk system. Since some disks might be accessed more than others they will become bottlenecks and limit the overall disk array performance. The equations in the table accounts for this effect. Table 6.3 gives the numbers for three different array sizes.

As the tables show, RAID level 1 has the best speedup and speed efficiency of all error correcting RAID levels, but with a high disk storage overhead. RAID level 5 is the second best and not so far behind RAID level 1 with respect to speedup and speed efficiency, but without the poor storage efficiency.

Analysis have shown that RAID level 6 clearly is the best alternative with respect to availability. RAID level 1 takes the second place followed by RAID 5 and the other parity strategies. RAID level 0 has no redundancy and clearly has the worse availability. See the papers listed above for a more detailed discussion on RAID availability.

The RAID concept has no central position in this work since it is best suited to operate inside a single node but as we will see the results are also valid for multi-site configurations. Both RAID level 1 and 5 are therefore of interest as declustering strategies. In this section we will discuss RADD (RAID level 5) and continue with Mirrored Declustering (RAID level 1) in Section 6.3.1.

Spare disks

In addition to data and parity disks it is also possible to include spare disks in a RAID disk array. After a disk failure the lost disk content can be immediately reconstructed and written to a spare disk as a background task. Later, when the failed disk has manually been replaced, it becomes the new spare. Figure 6.3 shows an example of a RAID level 5

RAID level	interleaving	$G = 5$			$G = 10$			$G = 25$		
		storage eff.	speed eff ok	speed eff fail	storage eff.	speed eff ok	speed eff fail	storage eff.	speed eff ok	speed eff fail
0	bit/byte	1.00	0.20	–	1.00	0.10	–	1.00	0.04	–
0	page	1.00	1.00	–	1.00	1.00	–	1.00	1.00	–
1	–	0.50	0.75	0.50	0.50	0.75	0.50	0.50	0.75	0.50
2	bit/byte	0.56	0.11	0.11	0.71	0.07	0.07	0.83	0.03	0.03
3	bit/byte	0.83	0.17	0.17	0.91	0.09	0.09	0.96	0.04	0.04
4	page	0.83	0.25	0.23	0.91	0.14	0.13	0.96	0.06	0.06
5	page	0.83	0.60	0.42	0.91	0.60	0.42	0.96	0.60	0.43
6	page	0.71	0.43	0.36	0.83	0.43	0.35	0.93	0.43	0.33

Table 6.3: Storage and speed efficiency of RAID level 0 through 6 for three values of G when there are twice as many reads as writes.

with a spare disk.

The advantage of immediate reconstruction is twofold. First, it reduces the time window when the system is vulnerable to a second disk failure causing irrecoverable data loss. Second, since a disk array has a significant reduction in performance while a disk has failed and pages have to be reconstructed for each disk read, immediate reconstruction will reduce the time with reduced performance.

The use of dedicated spare disks is by Menon and Mattson in [MM92] called *dedicated sparing*. The same paper also shows two other strategies; *distributed sparing* and *parity sparing*. With distributed sparing spare disk pages are distributed over all the disks in the disk array similar to the distribution of parity pages for RAID level 5. Figure 6.4 shows how parity, spare and disk pages can be distributed over six disks. Note how each row consists of four data, one parity, and one spare page. If one of the data pages or the parity page fail, the lost page can be reconstructed and written to the spare page in the same row. Also observe how six consecutive pages always are assigned to different disks. This speeds up accesses to large continuous data by allowing parallel I/O.

Parity sparing requires the presence of more than one disk array. If one disk fails, two disk arrays are merged. Of the two original parity pages per stripe, one is used as the new parity page and the second replaces the lost disk page.

Distributed and parity sparing are both graceful degradation strategies. While distributed and parity sparing exhibits higher performance and are symmetric (no disks has specialized tasks) when all disks are functional, this changes after a disk failure. After the reconstruction of lost pages the regular allocation of pages becomes disrupted and some consecutive pages are allocated to the same disk. By using the dedicated sparing strategy the original page allocation is maintained after reconstruction. Also, it requires only one

	Disk					
Block	1	2	3	4	5	6
0	P	0	1	2	3	S
1	4	P	6	7	8	S
2	9	5	P	12	13	S
3	14	10	11	P	18	S
4	19	15	16	17	P	S

Figure 6.3: A RAID level 5 disk array with one spare disk. Parity (P) and data pages $(0, 1, 2, \ldots 19)$ are distributed over disks 1 through 5 while disk 6 only contains spare pages (S). Only the five first pages on each disk is shown. This pattern is repeated for each group of five pages on the disks.

	Disk					
Block	1	2	3	4	5	6
0	P	S	0	1	2	3
1	4	P	S	7	8	9
2	10	5	P	S	14	15
3	16	11	6	P	S	21
4	22	17	12	13	P	S
5	S	23	18	19	20	P

Figure 6.4: Distribution of spare (S), parity (P) and data pages $(0, 1, 2 \ldots 23)$ over the disks of a RAID level 5 with six disks and distributed spare pages.

copying of the lost disk content after a failure since the spare becomes a data disk and the replaced disk the new spare. After replacing the failed disk for an array using distributed or parity sparing, the content must be copied back for the spare pages to be reclaimed, resulting in copying each page twice for each disk failure.

6.2.2 RADD

The *Redundant Array of Distributed Disks* (RADD) is an application of RAID level 5 for distributed systems being robust against site failures. RADD as described in [SS90] uses $G + 2$ sites including distributed spare capacity for reproduction of data lost due to a site failure. It is also possible to construct a RADD with a single spare site or without any spare at all.

For RADD each site corresponds to a disk in a RAID array. Each site consists of either a single- or multiprocessor computer with one or more disks. The site is the unit of failure and has independent failure modes from the other sites. In our terminology, each site is considered logically to have a single node. If G sites are necessary to store the data without any redundancy, $G + 1$ are necessary for data and parity and $G + 2$ for data, parity, and an additional spare.

Since remote communication is relatively slow and has long latencies, [SS90] suggest that data local to a site is allocated to disks on that site. Due to this the striping provided by RAID is not efficient for RADD. Anyhow, in a transaction processing environment striping is of less importance, but locality is a more important aspect. Many transaction processing applications have good data locality and can utilize this, but in this evaluation we choose not to consider this.

For the storage efficiency to be good, the number of sites should be high. With sparing the number of sites must be three or higher. Four sites give an efficiency of 50% and eight sites have a respectable 75%. For good efficiency numbers the high required number of sites is a serious cost penalty. Even if the computer hardware at each site might be priced low, the cost of establishing and maintaining this number of sites will be prohibitively expensive for most organizations. Also remember that it is not possible to fake this by placing multiple sites on the same location since they will be exposed to disasters. If this is done, a disaster will possibly knock out several logical sites making it impossible to mask the error and do self repair.

During self repair after a failure a RADD system is exposed to a massive amount of stress. This stress origins from the following extra work that has to be done during repair:

Transaction service: Some of the transactions will try to access pages on the lost site which are not reconstructed yet. These transactions will have to wait for those pages to be reconstructed. Even though the reconstruction can be initiated immediately, this results in G disk accesses where at least $G - 1$ are on remote sites. This involves virtually all sites and will probably slow down all transactions executed on the system.

Repair disk I/O: To do repair *all* data and parity pages, except those on the failed site, have to be read. Each site will have to read all data and parity pages on that site and ship them to the site having the spare pages. For the dedicated spare site strategy this is the full disk capacity while the distributed spare strategy have to read a $\frac{G}{G+2}$ fraction of the pages. This is respectively G and $\frac{G\cdot(G+1)}{G+2}$ times the gross disk capacity of the failed site. If $G = 6$ then the disk volume to read is respectively 6 and 5.25 times the lost disk volume! In addition comes writing the reconstructed pages accounting to respectively the full disk capacity of the spare site and a $\frac{1}{G+2}$ fraction of the disk capacity of each site.

Also, since the redundancy is based on physical units (pages), the system will also have to reconstruct pages that are unused or storing file and database system overhead (index pages, directories etc.). A logical replication strategy will only read pages storing valid tuples and the target can recreate and add management data like indices.

Repair communication: All the pages read have to be shipped to other sites where the spare pages are located. The total data volume that has to be sent with inter-site communication is equal to the data read, i.e. respectively G and $\frac{G\cdot(G+1)}{G+2}$ times the gross disk capacity of the failed site. This requires the communication network among the sites to have an extra high capacity compared to the other declustering strategies.

In addition to sending superfluous data like unused disk pages and system overhead, also unused capacity in each disk page must be sent between sites. Popular disk based data structures for transaction processing like B-trees [BM72] and hashing [Knu73] have a storage efficiency down to 65%. This overhead has to be read from disk independent of doing logical or physical replication, but can be removed before sending data over the communication network. Physical replication has no knowledge of the data structure inside a page and musr send complete pages only.[2]

Even during normal operation the cost of physical replication can be costly. Remote accesses are shipping full pages instead of only the requested tuple. Also, an access to a tuple will (depending on access method) often require more than one disk access. More than one disk access mean more than one remote access increasing the number of messages sent to access one tuple.

Advantages of RADD are simplicity and the robustness against site failures. There is no need for a global index which must be maintained consistently over the nodes of the system. Instead, location of data can be computed from the logical page number alone. In case of a site failure, the lost data can be reproduced without waiting for manual repair of the site.

Table 6.4 gives a summary of the three different RADD strategy characteristics.

6.2.3 Multi dimensional redundancy (nD-RADD)

To increase the reliability further [SS90] suggests a 2-dimensional RADD (2D-RADD). This is possible by forming a 2-dimensional array of sites where each row and each column in the array are checked by independent parity checksums and with spare capacity. This can be further extended into n dimensions (nD-RADD) by having parity checksum and spare

[2]By filling unused parts of a block with the binary value 0 and doing compression before sending pages this overhead can be reduced.

		RADD		
		no spare	dedicated spare	distributed spare
Number of sites	S	$G + 1$	$G + 2$	$G + 2$
Number of replicas	R	NA	NA	NA
Number of fragments	F	NA	NA	NA
Replica fanout	Z_r	G	G	$G + 1$
Takeover fanout	Z_t	G	G	$G + 1$
Refragmentation fanout	Z_f	NA	1	$G + 1$
Site fanout	Z_s	NA	1	$G + 1$
Storage efficiency		$\frac{G}{G+1}$	$\frac{G}{G+2}$	$\frac{G}{G+2}$
Range stride	Δ_R	depends on page size		
Fragment stride	Δ_F	$G + 1$	$G + 1$	$G + 2$
Self repair		no	dedicated spare	distributed spare
Planned repair		no	(yes)	(yes)
Homogeneous		yes	no	yes
Single site service		no	no	no
Simplicity		++	++	+

Table 6.4: RADD parameters. NA = not applicable.

capacity for each dimension. This increases the reliability by adding more redundancy, but unfortunately it also increases the required number of independent sites. For any reasonable storage efficiency figures the number of sites are unreasonable high and will therefore not be considered further.

6.3 Logical declustering

6.3.1 Mirrored declustering

Mirrored declustering (MD) is a well known declustering strategy used in several commercial transaction processing environments. Physical MD is the distributed version of RAID 1 and is in [SS90] denoted the *Read-One-Write-Both* scheme (ROWB). Two fault-tolerant commercial systems which are using physical MD are Stratus [Fre82] and Tandem Non-Stop Systems [Kat78].

While RADD only could be used for physical declustering, MD can also be used for logical declustering. Due to the major flaws with physical declustering which were found during the analysis of RADD, we will only discuss logical MD. The variant of MD described here is based on dedicated spare self repair. HypRa declustering is a MD based declustering strategy with distributed spare self repair and is described separately in Section 6.3.4.

Multi-site MD uses one site for each replica of the of the DBMS ($R = S$) where each site stores a complete replica of the database. Each site has the same number of nodes N_s excluding N_s' spares. One node from each site form a *replica group*. The R nodes in the replica group store the same $F_n = R$ fragment replicas. One of the fragment replicas on a node is a primary replica and the rest are hot stand-bys. There is one primary replica and $R - 1$ hot stand-bys of each of the $F = R \cdot N_s$ fragments. This is illustrated in Figure 6.5.

If a node failure makes the primary fragment replicas stored on that node unavailable, one of the hot stand-bys is transformed into a primary. Figure 6.6 shows that node n_1^1 has become unavailable and that fragment replica f_5^0 on node n_1^0 has become a primary. If the node failure is permanent and the node must be replaced, a spare node (node n_4^1) replaces the failed node in the replica group and the operating nodes in the group start to copy the replicas to the new group member.

When the lost fragment replicas have been completely reconstructed the spare node takes over as a normal member in the replica group and the primary task for one fragment is moved to it from the node which has two primaries. This is shown in Figure 6.7. When the failed node is replaced it becomes the new spare.

Table 6.5 gives a summary of MD with and without a spare node.

6.3.2 Interleaved declustering

Interleaved declustering (ID) is a logical declustering strategy that is used in the Teradata DBC / 1012 database computer [Ter85]. Originally it was used as a single site declustering strategy but, as we will see, it can be extended into multiple sites. ID can only support two replicas of the database. This description is based on [Su88] and [CK89].

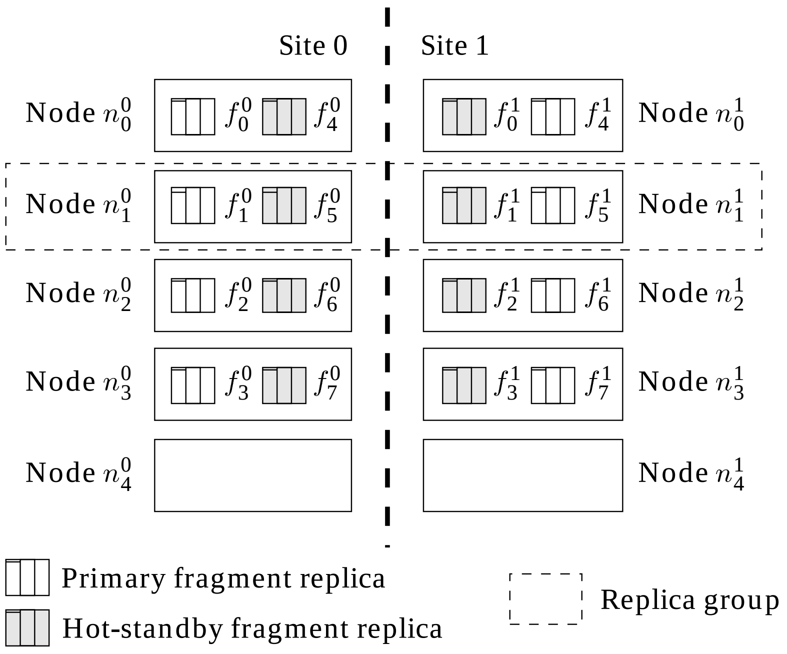


Figure 6.5: Mirrored declustering for two sites, each with five nodes. Each site has four data nodes and one spare. $R = S = 2$, $N_s = 4$, $N'_s = 1$, $F = 8$, $F_n = 2$

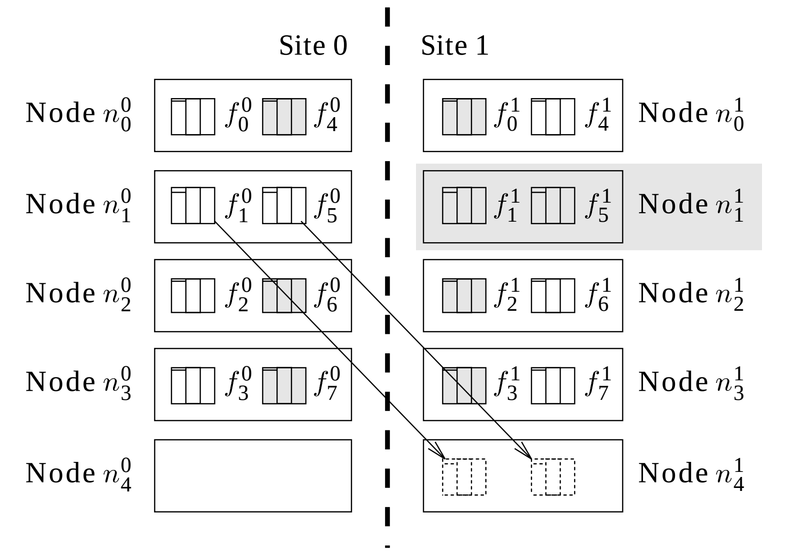


Figure 6.6: Mirrored declustering where node n_1^1 has failed.

		Mirrored declustering	
		no spare	dedicated spare
Number of sites	S	R	R
Number of replicas	R	≥ 2	≥ 2
Number of fragments	F	$R \cdot N_s$	$R \cdot N_s$
Replica fanout	Z_r	1	1
Takeover fanout	Z_t	$R - 1$	$R - 1$
Refragmentation fanout	Z_f	NA	1
Site fanout	Z_s	NA	1
Storage efficiency		$\frac{1}{R}$	$\frac{N_s}{R(N_s+N_s')}$
Range stride	Δ_R	1	1
Fragment stride	Δ_F	F	F
Self repair		no	dedicated spare
Planned repair		no	yes
Homogeneous		yes	no
Singe site service		yes	yes
Simplicity		++	++

Table 6.5: MD parameters.

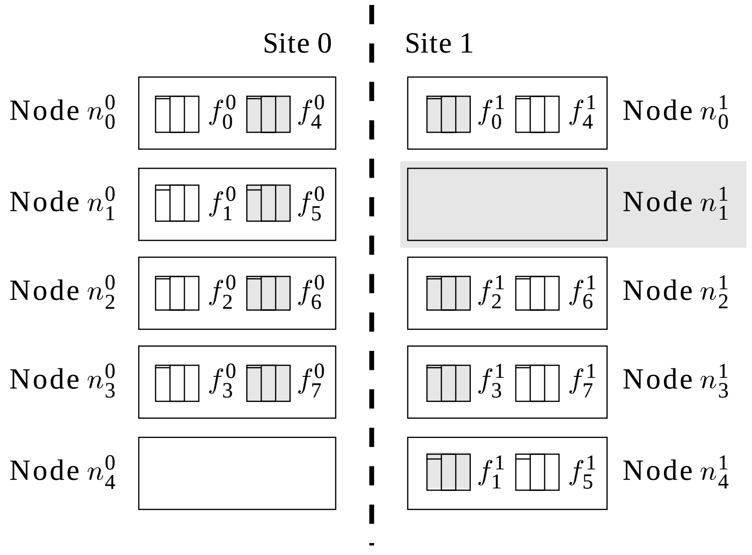


Figure 6.7: Mirrored declustering where node n_1^1 has failed and a spare node n_4^1 has taken over its task.

For a system with N nodes and no spares, partition the relation into $F = N \cdot (N - 1)$ fragments. Let there be two replicas of each fragment, one primary and one hot stand-by. $N-1$ primary and $N-1$ hot stand-by fragment replicas are assigned to each of the N nodes. The $N - 1$ primary fragment replicas that are assigned to a node have their corresponding hot stand-by replicas spread over the other $N - 1$ nodes.

Table 6.6 illustrates the interleaving scheme for four nodes. A row in the table lists the primary fragment replicas assigned to a node while a column lists the hot stand-by fragment replicas. For example, the primary replica of fragment f_7 is stored on node n_1 and the hot stand-by on node n_2. Since the diagonal elements are empty, no nodes store both the primary and hot stand-by replicas of the same fragment. This interleaving scheme can be generalized for N nodes using the following procedure; draw an $N \times N$ matrix and enumerate the rows and columns 0 to $N - 1$ respectively top to bottom and left to right. Insert the $N(N - 1)$ fragments into the matrix, one for each element, leaving the diagonal empty.

A failed node in a system using mirrored declustering causes only one node (the mirror node) to serve the load the failed node is unable to handle. ID ensures that the load from a failed node will be distributed over all the remaining nodes, i.e. a takeover fanout equal to $Z_t = N - 1$.

Unfortunately, the number of fragments F soon becomes very high when the number of nodes N increases since it is a function of order $O(N^2)$, e.g. with 20 nodes the relation must be partitioned into 380 fragments. The solution chosen by Teradata is to partition the nodes into C clusters of $N_c = \lceil \frac{N}{C} \rceil$ nodes. The interleaving scheme outlined above is used inside each cluster giving a total number of fragments $F = N(N_c - 1)$ and a replication fanout of $Z_r = N_c - 1$. The number of fragment replicas on each node $F_n = 2(N_c - 1)$ is

		Hot-standby replica node			
		n_0	n_1	n_2	n_3
Primary replica node	n_0		f_3	f_6	f_9
	n_1	f_0		f_7	f_{10}
	n_2	f_1	f_4		f_{11}
	n_3	f_2	f_5	f_8	

Table 6.6: Interleaving used by interleaved declustering to distribute 12 primary and hot stand-by fragment replicas (enumerated f_0 to f_{11}) over the nodes in a four node system (enumerated n_0 to n_3).

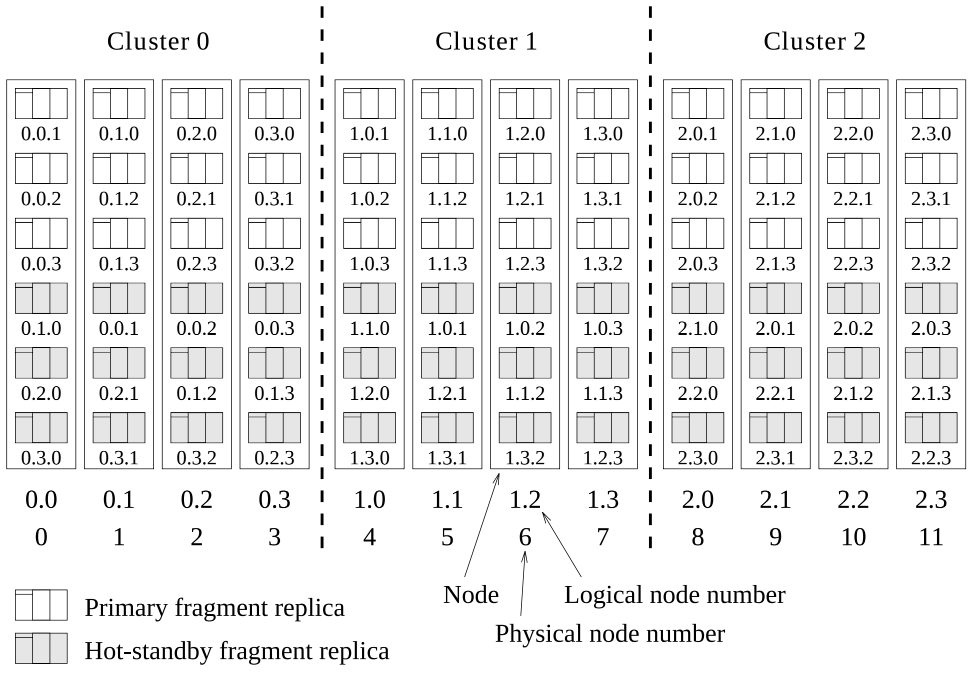


Figure 6.8: Interleaved declustering for three clusters with four nodes in each cluster. The fragment labeled $c.p.h$ is located on cluster c with primary on logical node $c.p$ and hot stand-by on logical node $c.h$.

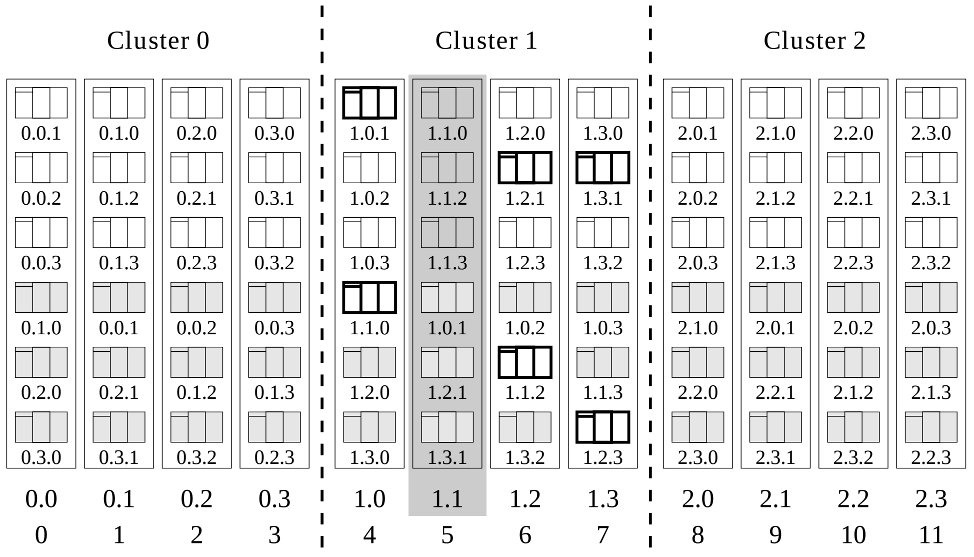


Figure 6.9: Interleaved declustering where logical node 1.1 has failed.

a function of the cluster size only and not the total number of nodes. Figure 6.8 shows an example of a 12 node system with three clusters.

In case of a node failure the remaining nodes inside the same cluster will share two fragments with the failed node, one primary and one hot stand-by fragment replica. The hot stand-by will take over as the primary giving an added load to each node inside the cluster. Figure 6.9 illustrates this situation. Logical node 1.1 has failed and the primary role for fragments 1.1.0, 1.1.2, and 1.1.3 has been moved to nodes 1.0, 1.2, and 1.3 respectively. Figure 6.10 shows how a spare node replaces the failed node as node 1.1 and how data are copied to this node.

Even though ID originally was designed as a single site declustering strategy it can easily be extended to multiple sites. Let the number of sites be equal to the number of nodes in each cluster, i.e. $S = N_c$, and assign one node from each cluster to each site. Based on the configuration in Figure 6.8, nodes 0.0, 1.0, and 2.0 can be assigned to site 0, nodes 0.1, 1.1, and 2.1 to site 1 etc. A sparing strategy can be achieved by allocating a suitable number of spare nodes to each site. There must be at least $S - 1$ sites available for the system to provide access to all data. This violates our assumption that one site should be sufficient to provide database service.

An ID configuration with $N_c = 2$ is incidentally equivalent with mirrored declustering with $R = 2$. Table 6.7 gives a summary of ID with and without spare nodes.

6.3.3 Chained declustering

Chained declustering (CD) is a declustering scheme developed by Hsiao and DeWitt [HD90]. With its low replication fanout ($Z_r = 2$) the method is robust against double failures but

		Interleaved declustering	
		dedicated spare	
		single site	multi site
Number of sites	S	1	N_c
Number of replicas	R	2	2
Number of fragments	F	$CN_c(N_c-1)$	$CN_c(N_c-1)$
Replica fanout	Z_r	N_c-1	N_c-1
Takeover fanout	Z_t	N_c-1	N_c-1
Refragmentation fanout	Z_f	1	1
Site fanout	Z_s	1	1
Storage efficiency		$\frac{N_s}{R(N_s+N_s')}$	$\frac{N_s}{R(N_s+N_s')}$
Range stride	Δ_R	$\frac{1}{N_c-1}$	$\frac{1}{N_c-1}$
Fragment stride	Δ_F	$C\cdot N_c$	$C\cdot N_c$
Self repair		dedicated spare	dedicated spare
Planned repair		yes	yes
Homogeneous		no	no
Single site service		no	no
Simplicity		0	-

Table 6.7: ID parameters.

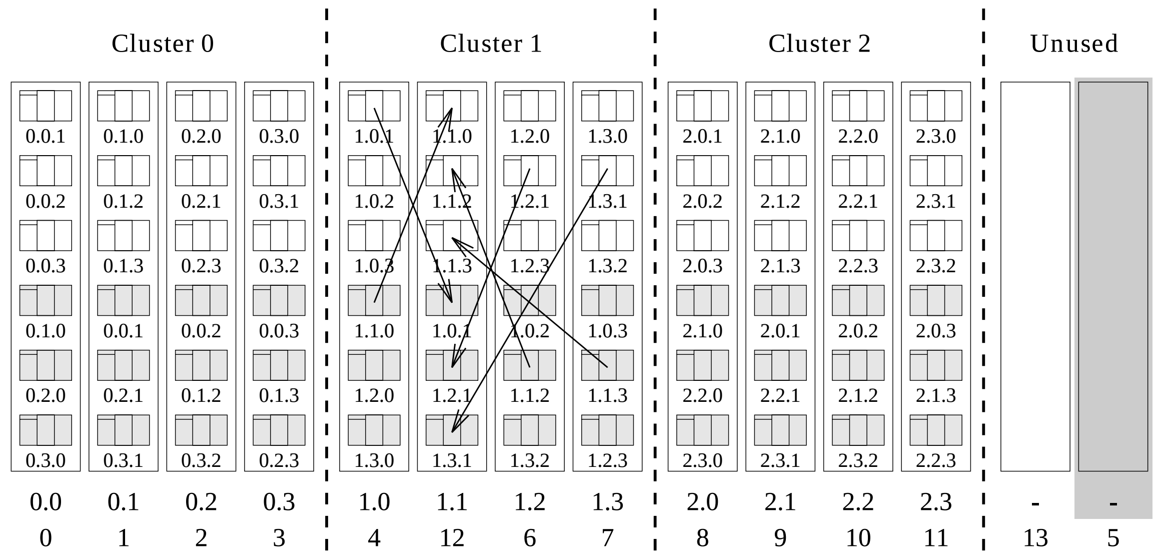


Figure 6.10: Interleaved declustering where logical node 1.1, physical node 5 has failed and a spare node (physical node number 12) has taken over as logical node 1.1.

at the same time able to balance the load after a node failure evenly over the remaining nodes. Originally the method was developed as a single site method but it is very simple to extend to two sites. CD as described in [HD90] supports only two replicas of the data, i.e. $R = 2$.

Single site CD is illustrated in Figure 6.11. With N nodes the relation is partitioned into N fragments. Assume that range partitioning is used. Each node is assigned two consecutive fragments so that it has one fragment in common with the previous node and one in common with the next node. The first node overlap with the last one as shown in the figure. Since range partitioning is used the two consecutive fragments will form a single range of partitioning key values (not valid for the last node). The first of the two fragment replicas assigned to a node becomes the primary and the second the hot stand-by.

In case of a node failure the partition ranges remain unchanged, but primary and hot stand-by ranges inside each node are modified so that all nodes are responsible for primary fractions of the same size. In normal mode a node is responsible for a $\frac{1}{N}$ fraction of the value range as primary and a $\frac{1}{N}$ as hot stand-by. After a node failure the primary responsibility becomes a $\frac{1}{N-1}$ fraction and hot stand-by a $\frac{N-2}{N(N-1)}$ fraction. This is shown in Figure 6.12. The figure also shows what started out as six evenly sized fragments results in twelve fragments, partially overlapping and with varying size. Locally at each node this can be handled by storing the fragments into the same physical file. The refragmentation is then reduced to redefining fragment borders and roles in the global index. For a high number of nodes this can result in a major overhead, especially since in average 50% of the distribution key value range must change role between primary and hot stand-by. While takeover for the other declustering strategies only involved the fragments directly affected by a node failure, CD has to update the information for *all* fragments. Also the switching between

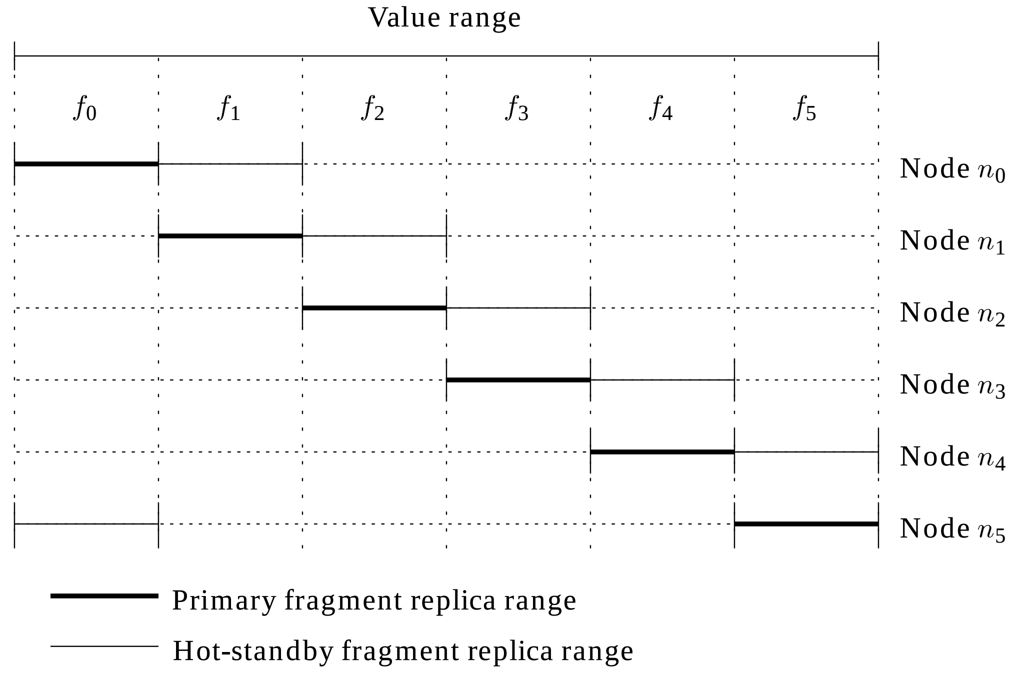


Figure 6.11: Fragment value range allocation for single site CD.

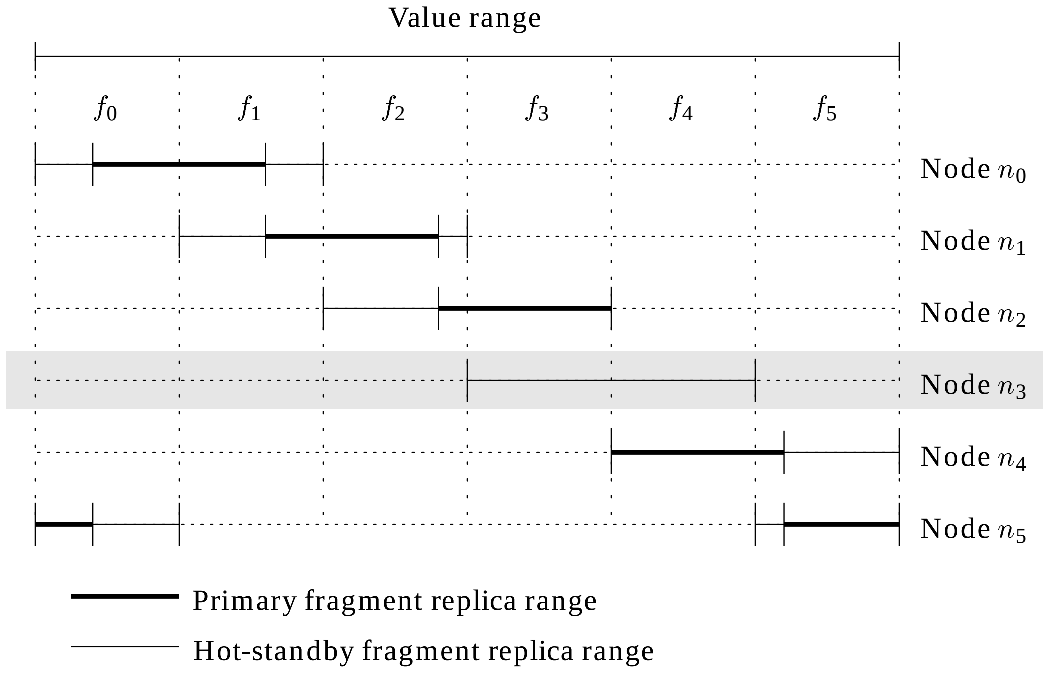
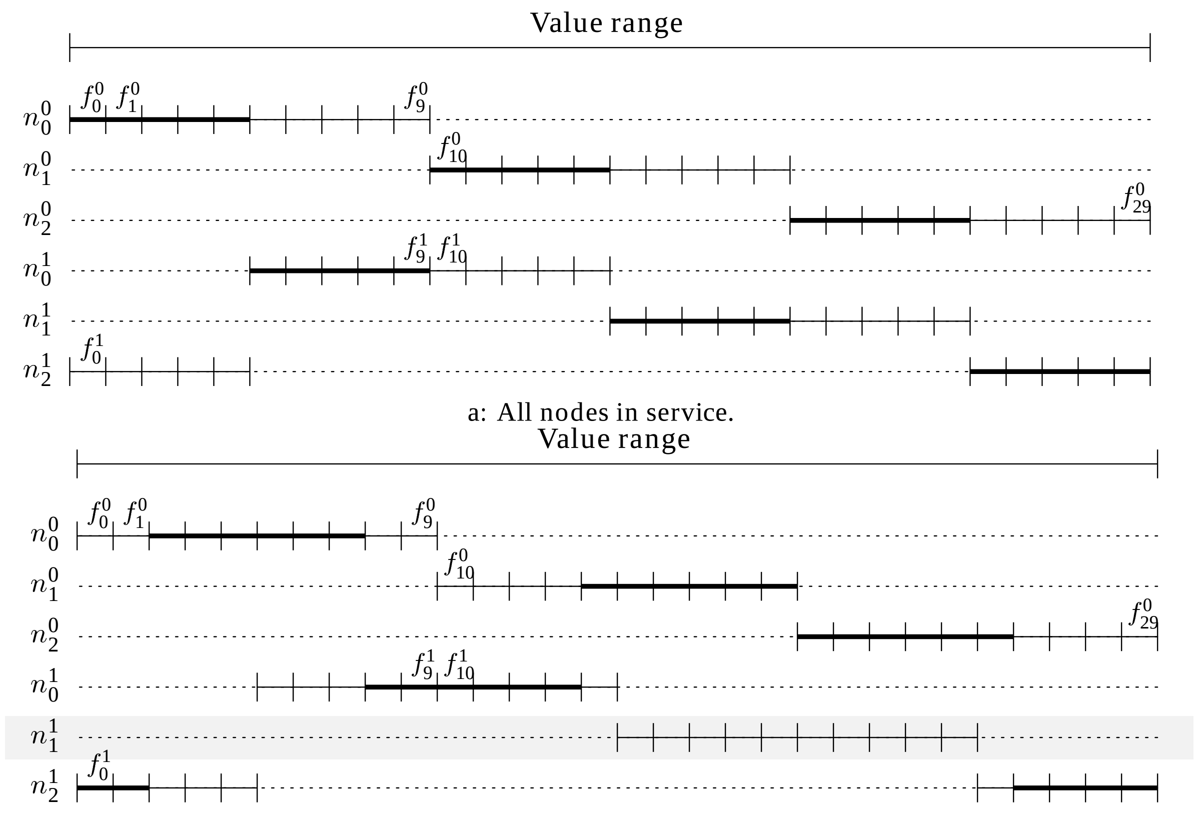


Figure 6.12: Load balance for single site CD during node failure.



a: All nodes in service.

b: Node n_1^1 has failed.

Figure 6.13: Assignment of partitioning key value ranges for two site CD declustering over six nodes in the normal mode (a) and in a node failure situation (b).

the primary and hot stand-by role is a complicated task demanding a synchronization to achieve a takeover satisfying the ACID rules. This potentially long-lasting massive takeover is critical since it can prolong the service unavailability caused by a node failure.

Even though it was assumed in the description above, CD is not limited to range partitioning. By using a hash formula $h(k)$ mapping the partition key values k to a high number of buckets B where $B \gg N$, the range of hash values $h(k)$ can be used to partition the relation.

A prerequisite for a declustering strategy resilient to site failures is that all data stored on a site is replicated on at least one other site. In particular, for a declustering strategy with only two replicas of the data, no site can store more than one replica. By studying Figure 6.11 it is easy to see that even numbered nodes store a single, complete replica of the relation. The same also applies to the odd numbered nodes. Thus by locating the even numbered nodes on one site and the odd numbered ones on a second, chained declustering becomes robust against site failures. Figure 6.13 illustrates CD for two sites both in the normal mode and after a node failure.

The figure also shows how the original N fragments each can be partitioned into $N - 1$

		Chained declustering	
		no spare	dedicated spare
Number of sites	S	2	2
Number of replicas	R	2	2
Number of fragments	F		
— normal operation		$2N_s$	$2N_s$
— one node failure		$2N_s - 1$	$2N_s - 1$
Replica fanout	Z_r	2	2
Takeover fanout	Z_t	$2N_s - 1$	$2N_s - 1$
Refragmentation fanout	Z_f	1	1
Site fanout	Z_s	1	1
Storage efficiency		$\frac{1}{2}$	$\frac{N_s}{2(N_s+N_s^!)}$
Range stride	Δ_R	1	1
Fragment stride	Δ_F	F	F
Self repair		no	dedicated spare
Planned repair		no	yes
Homogeneous		yes	no
Single site service		yes	yes
Simplicity		0	0

Table 6.8: CD parameters.

smaller fragments, in total $F = N(N-1)$ fragments. This technique avoids adjustment of fragment limits during takeover after a failure and reduces the takeover process work. A potential disadvantage with the prepartitioning strategy is that it can only handle a single node failure. While the dynamic adjustment scheme can be repeated at the second and third failure, the prepartition scheme can not assign equal sized ranges over the operating nodes for more than one node failure. For a high number of nodes this is no serious problem since the number of fragments also is fairly large ($F = N(N-1)$). Giving some nodes one more primary fragment than others makes little difference when each fragment is small. The relative difference for the second failure is limited to $\frac{N}{N-1}$ and results in a maximum skew of 10% and 2% respectively for 10 and 50 nodes.

CD only supports dedicated spare self repair. The two nodes adjacent to a failed node store the replicas of the lost fragments and are the only one able to send them to the new node during repair. Even though the transaction load is distributed even over the operating nodes, the system performance will be limited by the two adjacent nodes both serving transactions and reconstructing lost replicas.

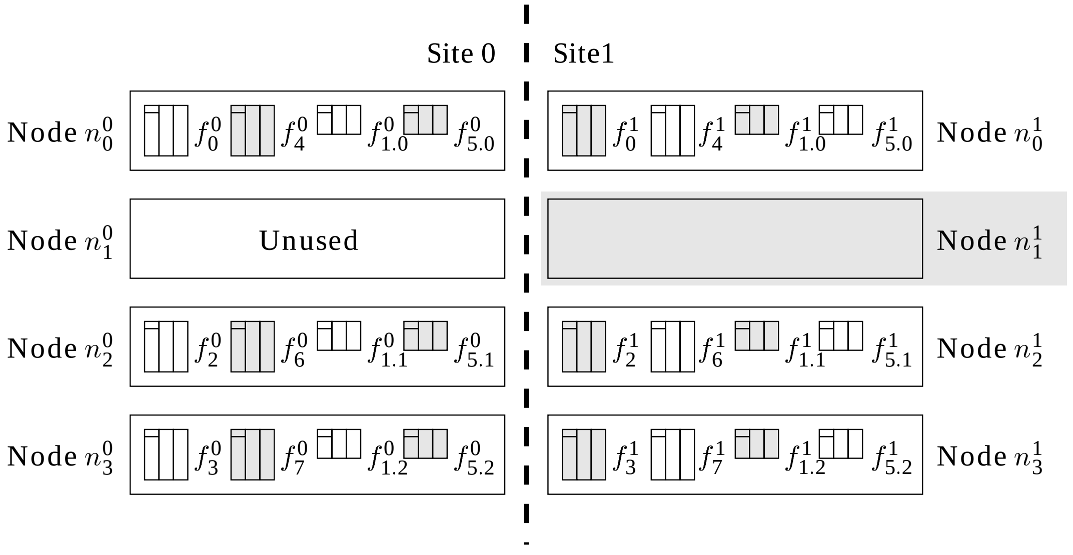


Figure 6.14: HypRa declustering after a node failure has been repaired.

6.3.4 HypRa declustering

HypRa declustering (HD) is a declustering strategy providing distributed sparing based on mirrored declustering. While the logical declustering strategies described in the previous sections have not been designed with multi-site declustering in mind, HD has. HD was originally proposed in [Hva92, HST$^+$91b].

In case of no node failures MD and HD are indistinguishable. Corresponding nodes at each site store replicas of the same fragments (see Figure 6.5 on page 88). But, instead of making an identical clone of the failed node onto a spare node after a node failure, the fragments which have lost a replica are partitioned into $N_s - 1$ small subfragments and distributed over the remaining $N_s - 1$ fragment groups. This is shown in Figure 6.14 where there is no spare nodes. Node n_1^1 has failed and the fragment replicas f_1^1 and f_5^1 are lost. Fragment f_1 is partitioned into the subfragments $f_{1.0}$, $f_{1.1}$, and $f_{1.2}$ and fragment f_5 into $f_{5.0}$, $f_{5.1}$, and $f_{5.2}$. Two replicas (one for each site) of each of the subfragments are created and the content read from the available fragments in the same replica group. After the copy is complete the original replicas are removed and the nodes in the same replica group become unused and can be used as spares for later failures.

We will denote this strategy as *global subfragmentation*. Alternatively we can limit the subfragmentation to the site of the failed node only, denoting it *site subfragmentation*.

One aesthetic argument for global subfragmentation is that it maintains the symmetry offered by mirrored declustering. One node on one site stores the same data as one node on the other sites. The fragmentation is identical over the sites. The one-to-one relationship between fragments makes it easy to designate primary and hot stand-by tasks to fragment replicas.

A more important argument against site subfragmentation is that the fault-tolerance level is reduced after self repair has been completed. Global subfragmentation has a replica fanout Z_r equal to one both in a normal system and when a failure has been repaired, while site subfragmentation changes the fanout from one to $N_s - 1$. The increased fanout means an increased probability for a double failure during self repair (see Chapter 8).

The penalty of the increased reliability offered by global subfragmentation is the work required to create the subfragments on all sites and to recreate the original fragments when the node hardware has been replaced. Site subfragmentation would have limited this work to only the site where the node failure occurred.

Additional failures before the failed node has been replaced (without any spares available) are handled identical to the first failure. In addition to partitioning normal fragments into subfragments, subfragments are partitioned into subsubfragments etc. The only restricting factor is the number of nodes at each site (they can not all being have failed) and the available spare capacity on each node. As long as these resources are sufficient there is no reason for being afraid of running out of dedicated spare nodes. While dedicated spare nodes offer extra resources for self repair only, distributed sparing provides extra resources for both self repair and handling of unpredicted performance peeks.

An option is to combine mirrored declustering (MD) with HD. Provide a few spare nodes on each site and let the system use MD until there is no spares left on a site. When this happens it can use HD on that site until the failed nodes have been replaced.

6.4 Summary

This chapter has described five existing multi-site declustering strategies: Redundant Array of Distributed Disks (RADD), Mirrored Declustering (MD), Interleaved Declustering (ID), Chained Declustering (CD), and HypRa Declustering (HD). RADD is a physical declustering strategy based on the RAID disk architectures. The other four can be both physical or logical but are only described as logical ones.

RADD, MD, and HD are the only schemes designed for multi-site declustering. The other two are shown how they can be adopted to multi-site declustering. RADD and HD both support a distributed spare self repair strategy, while the other can use a dedicated spare strategy.

A high number of sites is required for the good properties of RADD and ID to be evident. In practical cases the user can only afford to run at a few independent sites (two or three) and the two strategies show few (or no) advantages in this case. CD on the other hand is limited to two sites only.

As we will se in Chapter 8 the unavailability is to a great degree decided by the fanout values. It is therefore desirable to have the opportunity to select optimal fanout values. Unfortunately the described strategies have little degree of freedom with regard to this since the fanouts mostly are decided by the number of nodes, sites, etc., or are constant. The next chapter describes a class of declustering strategies and in more depth proposes one strategy with a greater freedom with respect to the number of sites, replicas, and fanout values. This strategy also allows self repair using both dedicated and distributed sparing.

		HypRa declustering	
		site subfragmentation	global subfragmentation
Number of sites	S	R	R
Number of replicas	R	≥ 2	≥ 2
Number of fragments	F		
— normal operation		RN_s	RN_s
— one node failure		$RN_s/2R(N_s-1)$	$2R(N_s-1)$
Replica fanout	Z_r		
— normal operation		1	1
Takeover fanout	Z_t	$R-1$	$R-1$
Refragmentation fanout	Z_f	$1 \leq Z_f \leq N_s-1$	$1 \leq Z_f \leq N_s-1$
Site fanout	Z_s	1	N_s
Storage efficiency		$\frac{1}{R}$	$\frac{1}{R}$
Range stride	Δ_R	1	1
Fragment stride	Δ_F	F	F
Self repair		distributed spare	distributed spare
Planned repair		yes	yes
Homogeneous		yes	yes
Single site service		yes	yes
Simplicity		0	-

Table 6.9: HD parameters.

Chapter 7

Minimum Intersecting Sets Declustering

This chapter consists of two main parts. It starts with describing and analyzing a family of declustering strategies called *Minimum Intersecting Sets Declustering*. The discussion ends up with the development of a novel declustering strategy called *Q-rot Declustering*.

7.1 General concepts

Minimum Intersecting Sets Declustering (MISD) is a family of declustering schemes meant to alleviate the problems of the declustering strategies described in the previous chapter.

Similar to interleaved declustering, MISD fragments the base relation into a number of fragments F that is much higher than the number of available nodes N, but contrary to ID, MISD supports multiple sites better. Figure 7.1 illustrates a MISD scheme on a two site configuration with $N = 10$ and $F = 20$.

The general concept of MISD is as follows:

> Relations are partitioned into a (high) number of fragments. Each fragment is initially created with one replica for each site. Inside a site, fragment replicas are assigned evenly over the nodes. The fragments assigned to a node form a *fragment set*. The sets should be assigned such that the maximum cardinality of the intersection between any pair of fragment sets is minimized. In case of a node failure the lost fragment replicas are moved to other fragment sets on the same site.

The intersection of two fragment sets is the set of fragments the two sets have in common. The intersection between two different fragment sets from the same site is always empty. If a node fails, the nodes having intersecting fragment sets have to take over all work on the common fragments. A smaller intersection means less common fragments and therefore also less added load in a failure situation. By insisting on a "minimum largest intersection cardinality" the worst case added load to a node is minimized, thus reducing the over-capacity required to mask errors. The configuration in Figure 7.1 satisfy this condition.

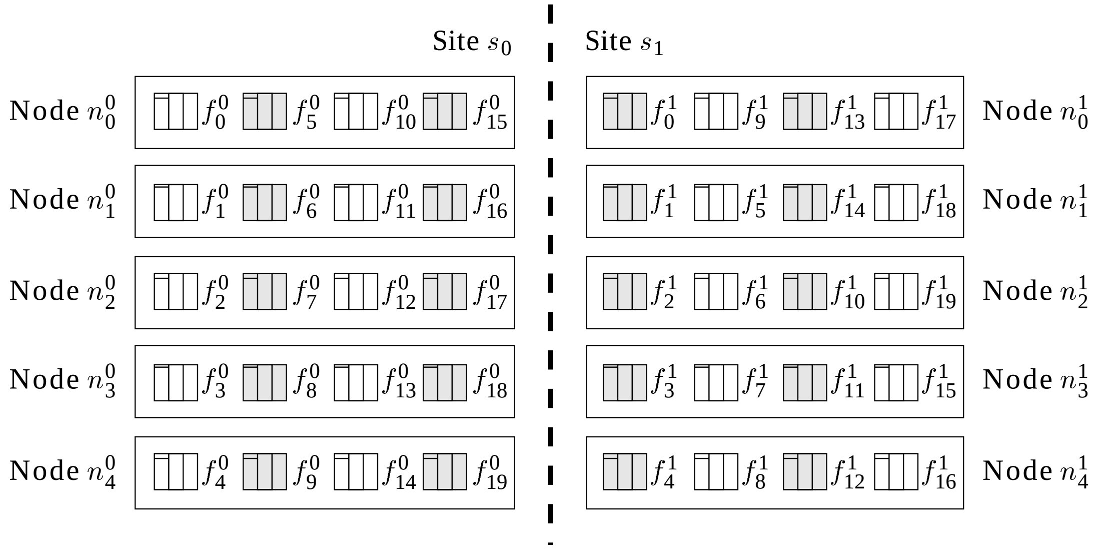

Site s_0
Site s_1
Node n_0^0
f_0^0 f_5^0 f_{10}^0 f_{15}^0
f_0^1 f_9^1 f_{13}^1 f_{17}^1
Node n_0^1
Node n_1^0
f_1^0 f_6^0 f_{11}^0 f_{16}^0
f_1^1 f_5^1 f_{14}^1 f_{18}^1
Node n_1^1
Node n_2^0
f_2^0 f_7^0 f_{12}^0 f_{17}^0
f_2^1 f_6^1 f_{10}^1 f_{19}^1
Node n_2^1
Node n_3^0
f_3^0 f_8^0 f_{13}^0 f_{18}^0
f_3^1 f_7^1 f_{11}^1 f_{15}^1
Node n_3^1
Node n_4^0
f_4^0 f_9^0 f_{14}^0 f_{19}^0
f_4^1 f_8^1 f_{12}^1 f_{16}^1
Node n_4^1

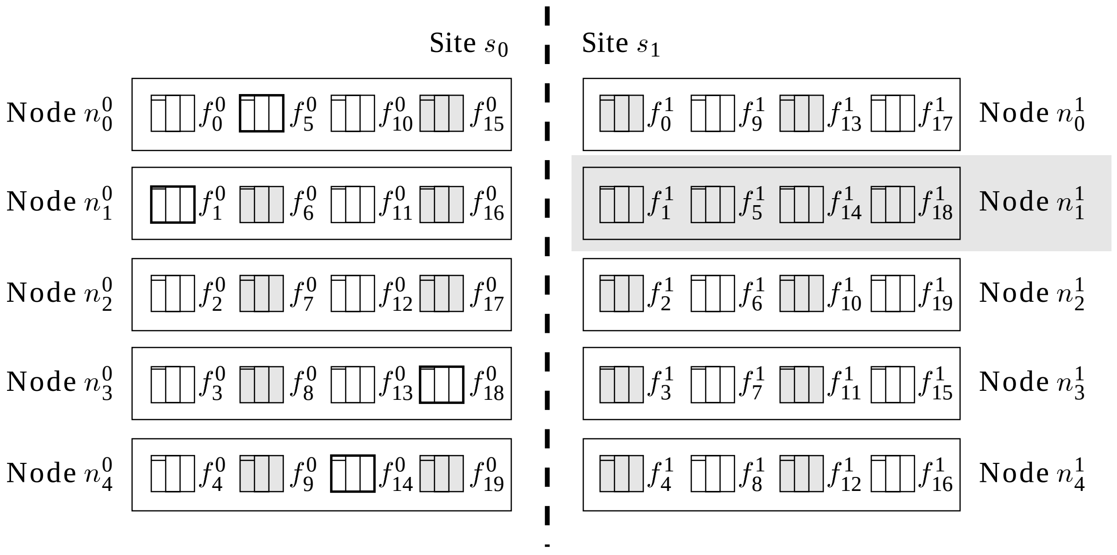
Figure 7.1: Minimal Intersecting Sets Declustering for a two site configuration with five nodes on each site and 20 fragments.

The figure shows 20 fragments distributed over ten nodes and two sites. No nodes on site s_0 share more than one fragment with nodes on site s_1.

Figure 7.2 shows that node n_1^1 has failed and that fragment replicas f_5^0 and f_{18}^0 that originally were hot stand-by replicas have to take over as primaries. Figures 7.3 and 7.4 show how respectively distributed sparing and dedicated sparing self repair can be done. When the failed node is replaced with a functioning one, the distributed sparing strategy has to copy the replicas back to their original location. For dedicated spare this is not necessary and the replaced node becomes the new spare (Figure 7.5).

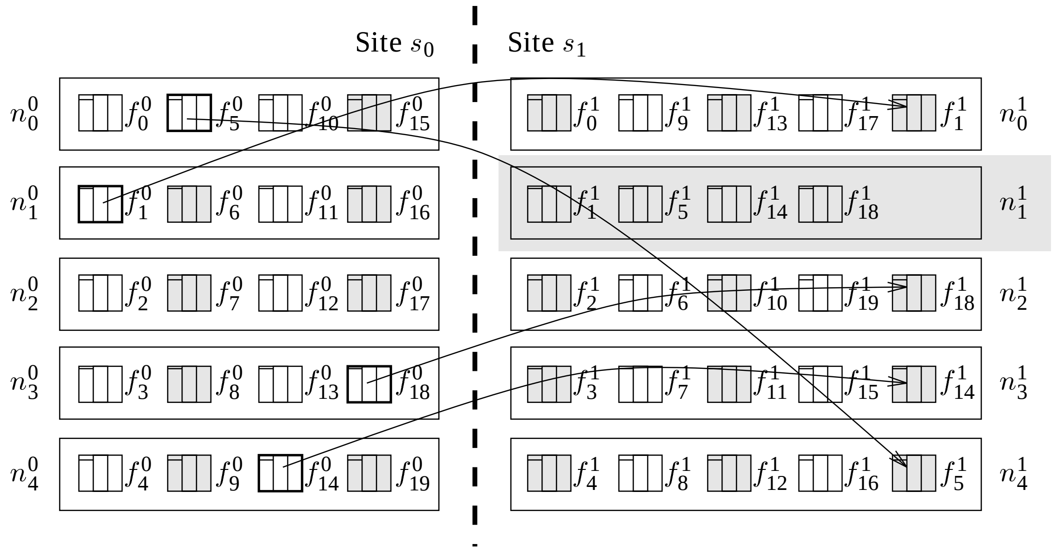
MISD keeps one and only one replica of each fragment on each site ensuring that each site can take over service alone. Reproduction of lost replicas inside the same site ensures that this condition holds also after repair.

The definition of MISD above is general but not sufficient for practical use. For a given number of nodes and sites it is not trivial to find an assignment that satisfy the requirements, and if one is found there is no guarantee that this is the optimal one. To be usable there must therefore exist a systematic approach for assignment of primary and hot stand-by fragments to nodes, and for deciding the degree of fragmentation. Such a scheme will be called an *assignment scheme* and should include reassignment rules for self repair and node reintegration. Further on, the assignment scheme should give a fragment allocation resembling the *Hybrid-Range Partitioning Strategy* [GD90] (described in Section 7.1.5) to ensure scalable performance for a wide range of query types.



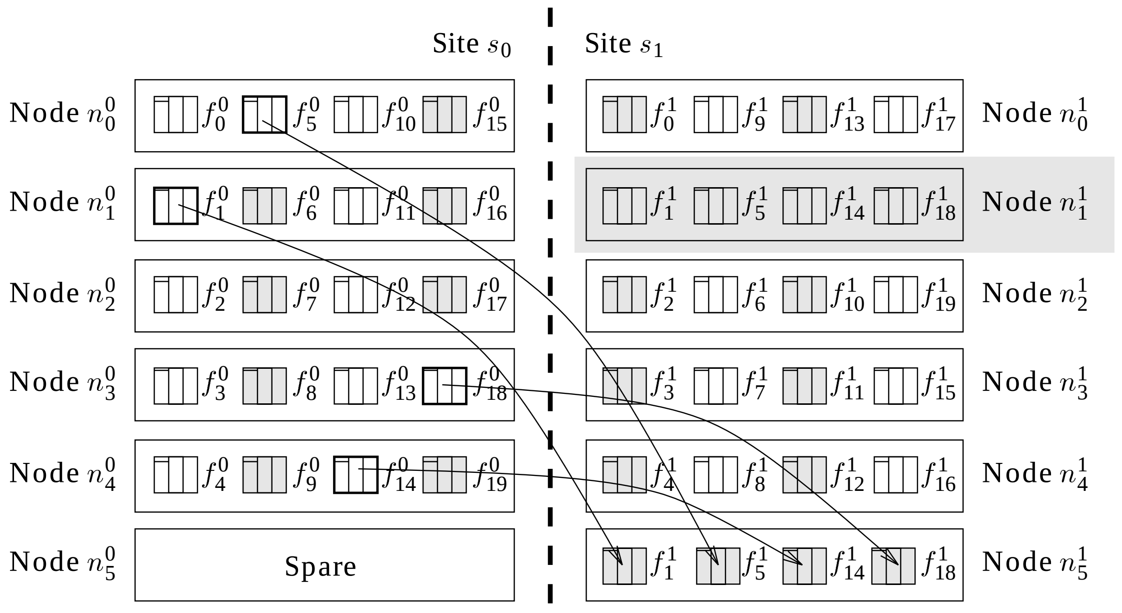
Site s_0
Site s_1
Node n_0^0
f_0^0 f_5^0 f_{10}^0 f_{15}^0
Node n_1^0
f_1^0 f_6^0 f_{11}^0 f_{16}^0
Node n_2^0
f_2^0 f_7^0 f_{12}^0 f_{17}^0
Node n_3^0
f_3^0 f_8^0 f_{13}^0 f_{18}^0
Node n_4^0
f_4^0 f_9^0 f_{14}^0 f_{19}^0
f_0^1 f_9^1 f_{13}^1 f_{17}^1
Node n_0^1
f_1^1 f_5^1 f_{14}^1 f_{18}^1
Node n_1^1
f_2^1 f_6^1 f_{10}^1 f_{19}^1
Node n_2^1
f_3^1 f_7^1 f_{11}^1 f_{15}^1
Node n_3^1
f_4^1 f_8^1 f_{12}^1 f_{16}^1
Node n_4^1

Figure 7.2: Minimal Intersecting Sets Declustering after node n_1^1 has failed and the hot stand-by fragment replicas have become primaries.



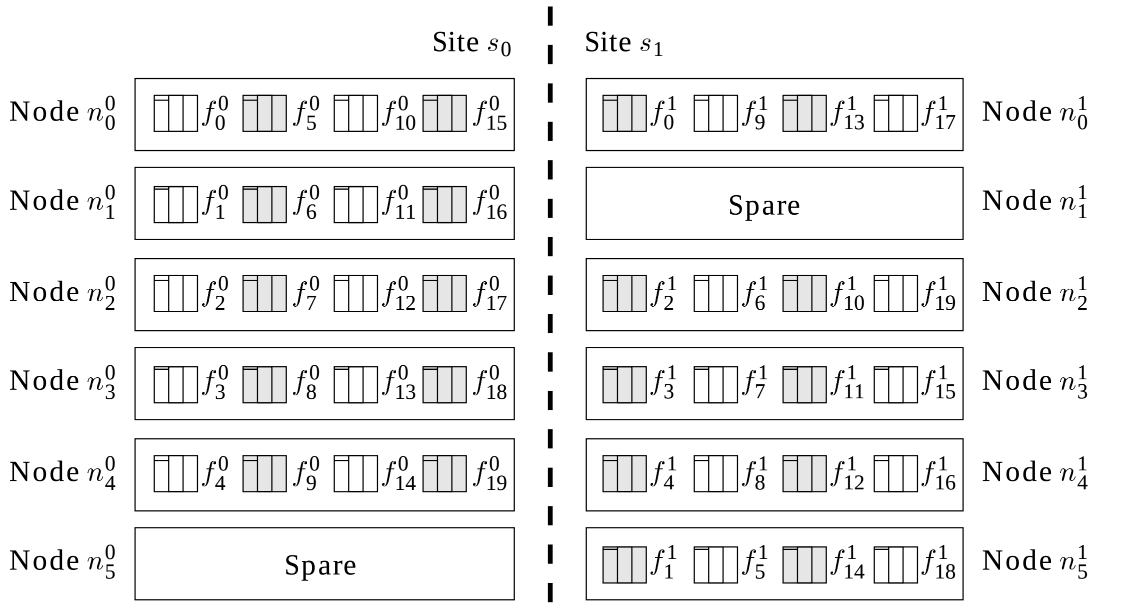
Site s_0
Site s_1
n_0^0
f_0^0 f_5^0 f_{10}^0 f_{15}^0
n_1^0
f_1^0 f_6^0 f_{11}^0 f_{16}^0
n_2^0
f_2^0 f_7^0 f_{12}^0 f_{17}^0
n_3^0
f_3^0 f_8^0 f_{13}^0 f_{18}^0
n_4^0
f_4^0 f_9^0 f_{14}^0 f_{19}^0
f_0^1 f_9^1 f_{13}^1 f_{17}^1 f_1^1
n_0^1
f_1^1 f_5^1 f_{14}^1 f_{18}^1
n_1^1
f_2^1 f_6^1 f_{10}^1 f_{19}^1 f_{18}^1
n_2^1
f_3^1 f_7^1 f_{11}^1 f_{15}^1 f_{14}^1
n_3^1
f_4^1 f_8^1 f_{12}^1 f_{16}^1 f_5^1
n_4^1

Figure 7.3: Minimal Intersecting Sets Declustering after node n_1^1 has failed and distributed spare self repair has been performed.



Site s_0
Site s_1
Node n_0^0
Node n_1^0
Node n_2^0
Node n_3^0
Node n_4^0
Node n_5^0
f_0^0 f_5^0 f_{10}^0 f_{15}^0
f_1^0 f_6^0 f_{11}^0 f_{16}^0
f_2^0 f_7^0 f_{12}^0 f_{17}^0
f_3^0 f_8^0 f_{13}^0 f_{18}^0
f_4^0 f_9^0 f_{14}^0 f_{19}^0
Spare
f_0^1 f_9^1 f_{13}^1 f_{17}^1
f_1^1 f_5^1 f_{14}^1 f_{18}^1
f_2^1 f_6^1 f_{10}^1 f_{19}^1
f_3^1 f_7^1 f_{11}^1 f_{15}^1
f_4^1 f_8^1 f_{12}^1 f_{16}^1
f_1^1 f_5^1 f_{14}^1 f_{18}^1
Node n_0^1
Node n_1^1
Node n_2^1
Node n_3^1
Node n_4^1
Node n_5^1

Figure 7.4: Minimal Intersecting Sets Declustering with spare after node n_1^1 has failed and self repair has been performed.



Site s_0
Site s_1
Node n_0^0
Node n_1^0
Node n_2^0
Node n_3^0
Node n_4^0
Node n_5^0
f_0^0 f_5^0 f_{10}^0 f_{15}^0
f_1^0 f_6^0 f_{11}^0 f_{16}^0
f_2^0 f_7^0 f_{12}^0 f_{17}^0
f_3^0 f_8^0 f_{13}^0 f_{18}^0
f_4^0 f_9^0 f_{14}^0 f_{19}^0
Spare
f_0^1 f_9^1 f_{13}^1 f_{17}^1
Spare
f_2^1 f_6^1 f_{10}^1 f_{19}^1
f_3^1 f_7^1 f_{11}^1 f_{15}^1
f_4^1 f_8^1 f_{12}^1 f_{16}^1
f_1^1 f_5^1 f_{14}^1 f_{18}^1
Node n_0^1
Node n_1^1
Node n_2^1
Node n_3^1
Node n_4^1
Node n_5^1

Figure 7.5: Minimal Intersecting Sets Declustering with spare after node n_1^1 has failed, self repair has been performed and node n_1^1 replaced.

7.1.1 Optimal fragment assignment

Assume a two site configuration with N_0 nodes on site s_0 and N_1 on site s_1. Let $I_{n_a^0,n_b^1}$ be the intersection of the fragment sets assigned to respectively node n_a^0 on site s_0 and node n_b^1 on site s_1. Since a given fragment will be assigned to only one node on site s_0 and only one node on site s_1, the fragment will be a member of one and only one $I_{n_a^0,n_b^1}$. The union of all $I_{n_a^0,n_b^1}$ will therefore be the set of all fragments and have a cardinality of F:

$$F = \#\bigcup_{a,b} I_{n_a^0,n_b^1} = \sum_{a,b} \#I_{n_a^0,n_b^1}$$

where $\#A$ denotes the cardinality of set A. The largest intersection L for a fragment assignment is written:

$$L = \max_{a,b}(\#I_{n_a^0,n_b^1})$$

A fragment assignment with the smallest possible L is *optimal*. Since L is greater or equal to all $\#I_{n_a^0,n_b^1}$, we see that:

$$F = \sum_{a,b} \#I_{n_a^0,n_b^1} \leq \sum_{a,b} L = N_0 N_1 L,$$

thus,

$$F \leq N_0 N_1 L$$

Since L must be an integer we get:

$$L \geq \left\lceil \frac{F}{N_0 N_1} \right\rceil \tag{7.1}$$

This result gives the lower limit for L and tells us when we can stop looking for a better fragment assignment. It will also be used later to show that assignment schemes are optimal. This theoretical lower limit will be denoted L_{min}.

Even though this result assumed two sites, it can be extended to any number of sites greater or equal to two. Assume s_x and s_y are the sites with the lowest number of nodes. Applying Equation 7.1 using N_x and N_y gives the theoretical "minimum largest intersection cardinality" $L_{\text{min}(x,y)}$ for that site pair. Since N_x and N_y are the smallest site sizes, no other pair of nodes has a larger minimum L, thus $L_{\text{min}(x,y)} = L_{\text{min}}$ applies to the whole multisite configuration.

It should be noted that L_{min} is a theoretical lowest value for L and it is not guaranteed that it actually exists any fragment assignment that has $L = L_{\text{min}}$.

The "minimum largest intersection cardinality" requirement ensures optimal load balance in case of failure. Near optimal load balance can in most cases be "good enough" and makes

Site	Node	Transposed matrix					Rotated columns					Empty diagonal				
s_0	n_0^0	⓪	5	⑩	15	⑳	⓪	5	⑩	15	⑳	⓪	5	⑩	15	
	n_1^0	1	⑥	11	⑯	21	①	6	⑪	16	㉑	①	6	⑪	16	
	n_2^0	②	7	⑫	17	㉒	②	7	⑫	17	㉒	②	7	⑫	17	
	n_3^0	3	⑧	13	⑱	23	③	8	⑬	18	㉓	③	8	⑬	18	
	n_4^0	④	9	⑭	19	㉔	④	9	⑭	19	㉔	④	9	⑭	19	
s_1	n_0^1	0	①	2	③	4	0	⑨	13	⑰	21		4	⑧	12	⑯
	n_1^1	⑤	6	⑦	8	⑨	1	⑤	14	⑱	22	0		⑨	13	⑰
	n_2^1	10	⑪	12	⑬	14	2	⑥	10	⑲	23	1	⑤		14	⑱
	n_3^1	⑮	16	⑰	18	⑲	3	⑦	11	⑮	24	2	⑥	10		⑲
	n_4^1	20	㉑	22	㉓	24	4	⑧	12	⑯	20	3	⑦	11	⑮	

ⓘ Primary replica of fragment f_i

i Hot stand-by replica of fragment f_i

Figure 7.6: Three alternative assignment schemes for Minimum Intersecting Sets Declustering for dual site configurations. Shaded fragment numbers denote primary replicas, unshaded are hot stand-bys.

heuristic assignment schemes sufficient. We will now describe three optimal assignment schemes for two site configurations which assume that the degree of fragmentation can be chosen without restrictions. It is assumed that both sites have the same number of nodes when no nodes has failed. The three assignment schemes are illustrated in Figure 7.6.

The three assignment schemes distribute the fragments identically on site s_0. In Figure 7.6 the fragments are enumerated from 0 and upwards. The assignment for each site can be viewed as a matrix where the rows denote the fragments assigned to the same node. If a sequence preserving partitioning scheme is used, the fragments should be enumerated based on increasing partitioning key values. The fragments are then assigned to nodes by a round-robin scheme. What separates the assignment schemes is how fragments are assigned to the nodes on site s_1, how primaries and hot stand-bys are assigned, and the degree of fragmentation.

7.1.2 Transposed Matrix (TM)

Visually TM can best be illustrated by transposing a matrix (thereby the name). This analogy can easily be seen by studying Figure 7.6. Since the sites have the same number of nodes N_s, each node s_0 must store N_s fragments, one for each node on site s_1. This requires that relations are fragmented into $F = {N_s}^2$ fragments.

TM can also be generalized to $F = k({N_s}^2)$ fragments where k is a positive integer. The fragments are divided into groups of ${N_s}^2$ fragments where each group can be assigned as described above.

It is easy to verify that TM is an optimal assignment scheme. Using Equation 7.1 we get:

$$L \geq \left\lceil \frac{F}{N_0 N_1} \right\rceil = \left\lceil \frac{k({N_s}^2)}{{N_s}^2} \right\rceil = k$$

The optimal value for L is therefore k. Since the assignment on one site is a transposition of the assignment on the other site for each of the k groups of fragments, each node on the first site has k fragments in common with any node on the other site. The largest intersection therefore becomes k which is equal to the optimal value, i.e. **TM is an optimal assignment scheme**.

Primaries and hot stand-bys are assigned in a checkerboard pattern. This is possible due to the transposition symmetry.

7.1.3 Rotated Columns (RC)

RC assigns fragments to nodes at site s_1 in a round-robin fashion similar to the assignment for site s_0, except that the start node is shifted one node each time a column fills up. On Figure 7.6 it visually looks like the columns are rotated one step down relative to the previous column.

Primaries and hot stand-bys are assigned by letting fragments in alternating columns have the primary on site s_0.

Section 7.2 will show that a generalization of RC is a MISD and therefore an optimal assignment scheme.

7.1.4 Empty Diagonal (ED)

The ED assignment scheme is a specialization of RC suitable for distributed sparing. ED assumes that the relation is split into $F = kN_s(N_s - 1)$ fragments. The fragments are assigned round-robin at site s_1, but for each N_s'th fragment one node is skipped starting with the first node. By studying Figure 7.6 it is easy to see that RC and ED assigns fragments identically just with another node enumeration at site s_1 (node n_0^1 becomes n_1^1, n_1^1 becomes n_2^1 and so on). The name of the assignment scheme can be explained by taking a look on the fragment assignment shown on the figure. It appears to be a N_s by N_s matrix with an empty diagonal. Primarys and hot stand-bys are assigned identically to RC.

When a node fails kN_s fragment replicas are lost. Using distributed sparing each remaining node is assigned k new fragments increasing the load with a fraction $\frac{1}{N_s-1}$. This ensures an even load redistribution. Unfortunately this is only valid for the first failure at a site. The example used in the beginning of this chapter and shown in Figure 7.1 through 7.5 is actually an ED scheme and Figure 7.3 shows a dedicated spare reassignment.

7.1.5 Discussion

The three assignment schemes are summarized in Table 7.1.

	Transposed matrix	Rotated columns	Empty diagonal
F	$k(N_s{}^2)$	kN_s	$kN_s(N_s - 1)$
$\mathcal{N}_0(i)$	$i \bmod N_s$	$i \bmod N_s$	$i \bmod N_s$
$\mathcal{N}_1(i)$	$i \operatorname{div} N_s$	$(i + (i \operatorname{div} N_s)) \bmod N_s$	$(1 + i + (i \operatorname{div} N_s)) \bmod N_s$
Primary replica site	$((i \bmod N_s) + (i \operatorname{div} N_s)) \bmod 2$	$(i \operatorname{div} N_s) \bmod 2$	$(i \operatorname{div} N_s) \bmod 2$
Minimum Δ_F	2	$2(N_s - 1)$	$2(N_s - 1)$
Average Δ_F	$\frac{4}{kN_s} + \frac{5}{2}$	$2N_s - 1 + \frac{3}{2N_s}$	$2N_s - 1 + \frac{3}{2N_s}$

Table 7.1: Comparison of three assignment schemes for two site Minimal Intersecting Sets Declustering. $\mathcal{N}_0(i)$ and $\mathcal{N}_1(i)$ is the mapping function from fragment number i to node number on respectively site s_0 and site s_1. Δ_F is the fragment stride. N_s is the number of nodes on each site.

Hybrid-Range Partitioning

The *Hybrid-Range Partitioning Strategy* (HRPS) [GD90] ensures efficient execution of both exact match, small range, and long range queries on the partitioning key through partitioning the relation into a number of fragments which are independent of the available number of nodes in the system. Small relations will be partitioned into few fragments and large into many. For large relations the strategy can end up with a fragmentation degree which is much higher than the number of nodes. Queries accessing many tuples (long range queries) should be distributed over as many nodes as possible to get a high degree of parallelism. The degree of parallelism must be traded against the high startup and synchronization cost for each node involved in the query [CABK88, GD90]. Queries accessing just a few tuples (exact match and small range queries) should therefore only involve one or a few nodes.

Since MISD also can handle a varying number of fragments (e.g. RC), it is possible to achieve the same efficiency as for HRPS for large relations. Long range queries must access all fragments with tuples mapping into the partitioning key value range specified by the query. To get the desired degree of parallelism these fragments should be distributed evenly over as many different nodes as possible. This is not the case if neighbor ranges, or even worse, several consecutive ranges, are assigned to one and the same node. A measure for the quality of an assignment is the *fragment stride* (Δ_F). The optimal fragment stride is the number of active nodes in the system $S \cdot N_s$. Since the assignment can neither be symmetric (equal on all sites) nor regular (repeating the same assignment pattern) the fragment stride will vary depending on the fragment range we choose to start counting at. The minimum fragment stride given in Table 7.1 is the worst case value while the average value gives the overall quality of the assignment scheme.

Figure 7.7 shows how the three different assignment schemes distribute partitioning key

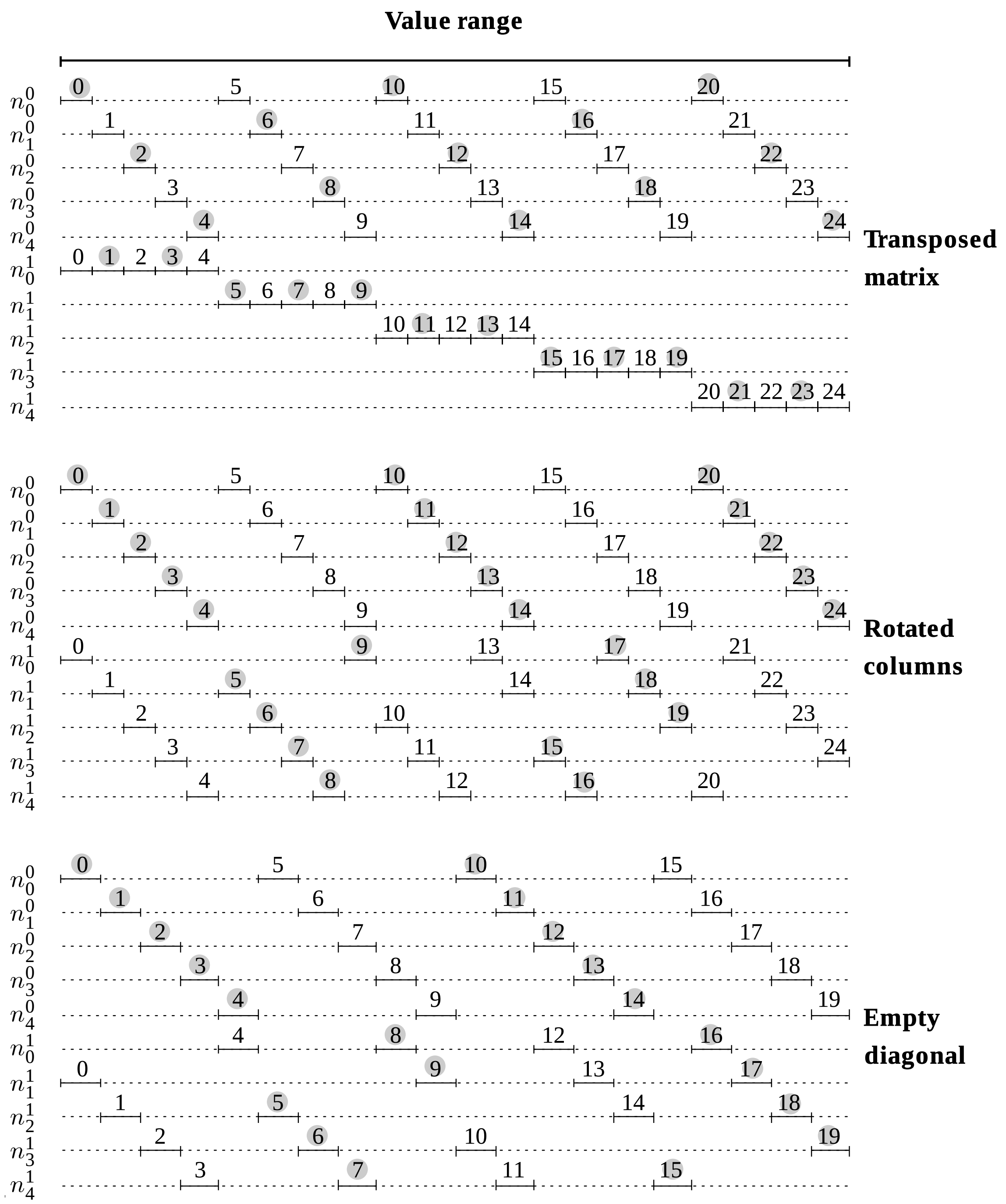


Figure 7.7: Range allocation for the three assignments schemes when range partitioning is used.

value ranges over the nodes when range partitioning is used. It is easy to see that TM is bad for long range queries. Although site s_0 has an ideal assignment, i.e. an internal fragment stride equal to the number of nodes N_s, site s_1 store five consecutive fragments on each of the five nodes. RC and ED does not have this flaw and the average stride is close to the optimal value.

An alternative metric is *range stride* denoted Δ_R which expresses the fraction of the partitioning key value range which can be selected without picking more than one fragment from the same node. The relationship between range stride and fragment stride is:

$$\Delta_F = \Delta_R F$$

Access method considerations

A large number of fragments also means many entries in the global index. Since the global index is replicated over all transaction executing nodes it can be rather expensive to store and maintain.

Locally on each node we can choose between storing the fragments from the same relation into the same access structure (e.g. B-tree) or if we will use one access structure per fragment. One advantage with one common access structure is a minimum of node internal files. Sequential accesses and range queries only need to handle a single file. In combination with a hash based partitioning scheme and using an access method keeping records sorted locally on the node we can also scan the node local fragments without having to keep track of the position concurrently in several files. It is also easier to change lower and upper fragment borders dynamically by just adjusting the border values.

However, separate access structures for each fragment makes it easier to reorganize the system, e.g. by moving a full fragment from one node to another. If the fragment is merged with several others it might be required to scan much more tuples than necessary.

7.2 Q-rot declustering

This section will formally present the rotated columns assignment scheme (RC) generalized for an arbitrary number of sites (i.e. arbitrary number of replicas). The description will show the remapping strategy for self repair and how to efficiently assign fragment replicas to nodes, both in the case of absence of failures, during self repair, and after self repair has been completed.

7.2.1 Description

The name of Q-rot declustering is a shorthand for *quotient rotation*. Each site is assigned a rotation quotient. When all nodes at a site have been assigned a fragment, the assignment of fragments starts at a node shifted the number of steps decided by the quotient. RC uses quotients equal to 0 for site s_0 and 1 for site s_1.

Q-rot (QD) is characterized by the following parameters:

S	Number of sites
N_s	Number of active nodes per site
N'_s	Number of dedicated spare nodes per site
Z_r	Replica fanout, $1 \le Z_r \le N_s$
$q_0 \cdots q_{S-1}$	Rotation quotients for each site, $0 \le q_x < N_s$

In the normal state there is one replica of the database stored on each site, i.e. $R = S$. At each site the database replica is declustered over N_s nodes (same for all sites) with a replica fanout of Z_r. Note that the fanout can not be larger than the number of active nodes at each site, i.e. $Z_r \le N_s$. The table is partitioned into $F = N_s \cdot Z_r$ fragments.

Each site s_x is assigned a unique *rotation quotient* denoted q_x where $0 \le q_x < N_s$. Note that in the general case any rotation quotient that can be written $q_x + kN_s$ where k is an integer, are equivalent with a rotation quotient of q_x.

Fragments f_i are assigned to nodes at site s_x with an *assignment function* $\mathcal{N}_x(i)$. The *primary replica function* $\mathcal{P}(i)$ returns the site where the primary replica of fragment f_i is located.

For QD $\mathcal{N}_x(i)$ and $\mathcal{P}(i)$ are given by Equation 7.2 and 7.3 respectively.

$$\mathcal{N}_x(i) = (i + (i \text{ div } N_s)q_x) \bmod N_s \tag{7.2}$$

$$\mathcal{P}(i) = (i \text{ div } N_s) \bmod S \tag{7.3}$$

Example 7.1:

Assume a system with three replicas ($R = S = 3$), nine nodes at each site ($N_s = 9$), and a replica fanout of six ($Z_r = 6$). For site s_x ($x \in \{0, 1, 2\}$) use rotation quotient $q_x = x$ (e.g. $q_2 = 2$). Totally there will be 54 fragments ($F = N_s \cdot Z_r = 9 \cdot 6 = 54$) and 162 fragment replicas ($F \cdot R = 54 \cdot 3 = 162$) distributed over 27 nodes ($N_s \cdot S = 9 \cdot 3 = 27$). Figure 7.8 shows how the fragment replicas are assigned to each node. n_i^x is node number i on site s_x. f_i^r denotes replica number r of fragment number i. The fragment replicas suffixed by an asterisk ($*$) are the primary fragment replica.

$\square$

7.2.2 Proof of minimal intersection

Theorem:

QD is a MISD **iff** Equations 7.4 and 7.5 hold between any two sites s_x and s_y in the system.

$$q_x \neq q_y \tag{7.4}$$

$$\text{gcd}(|q_x - q_y|, N_s) \le \frac{N_s}{Z_r} \tag{7.5}$$

$\square$

Proof:

Site 0 with $q_0 = 0$

n_0^0	f_0^0*	f_9^0	f_{18}^0	f_{27}^0*	f_{36}^0	f_{45}^0
n_1^0	f_1^0*	f_{10}^0	f_{19}^0	f_{28}^0*	f_{37}^0	f_{46}^0
n_2^0	f_2^0*	f_{11}^0	f_{20}^0	f_{29}^0*	f_{38}^0	f_{47}^0
n_3^0	f_3^0*	f_{12}^0	f_{21}^0	f_{30}^0*	f_{39}^0	f_{48}^0
n_4^0	f_4^0*	f_{13}^0	f_{22}^0	f_{31}^0*	f_{40}^0	f_{49}^0
n_5^0	f_5^0*	f_{14}^0	f_{23}^0	f_{32}^0*	f_{41}^0	f_{50}^0
n_6^0	f_6^0*	f_{15}^0	f_{24}^0	f_{33}^0*	f_{42}^0	f_{51}^0
n_7^0	f_7^0*	f_{16}^0	f_{25}^0	f_{34}^0*	f_{43}^0	f_{52}^0
n_8^0	f_8^0*	f_{17}^0	f_{26}^0	f_{35}^0*	f_{44}^0	f_{53}^0

Site 1 with $q_1 = 1$

n_0^1	f_0^1	f_{17}^1*	f_{25}^1	f_{33}^1	f_{41}^1*	f_{49}^1
n_1^1	f_1^1	f_9^1*	f_{26}^1	f_{34}^1	f_{42}^1*	f_{50}^1
n_2^1	f_2^1	f_{10}^1*	f_{18}^1	f_{35}^1	f_{43}^1*	f_{51}^1
n_3^1	f_3^1	f_{11}^1*	f_{19}^1	f_{27}^1	f_{44}^1*	f_{52}^1
n_4^1	f_4^1	f_{12}^1*	f_{20}^1	f_{28}^1	f_{36}^1*	f_{53}^1
n_5^1	f_5^1	f_{13}^1*	f_{21}^1	f_{29}^1	f_{37}^1*	f_{45}^1
n_6^1	f_6^1	f_{14}^1*	f_{22}^1	f_{30}^1	f_{38}^1*	f_{46}^1
n_7^1	f_7^1	f_{15}^1*	f_{23}^1	f_{31}^1	f_{39}^1*	f_{47}^1
n_8^1	f_8^1	f_{16}^1*	f_{24}^1	f_{32}^1	f_{40}^1*	f_{48}^1

Site 2 with $q_2 = 2$

n_0^2	f_0^2	f_{16}^2	f_{23}^2*	f_{30}^2	f_{37}^2	f_{53}^2*
n_1^2	f_1^2	f_{17}^2	f_{24}^2*	f_{31}^2	f_{38}^2	f_{45}^2*
n_2^2	f_2^2	f_9^2	f_{25}^2*	f_{32}^2	f_{39}^2	f_{46}^2*
n_3^2	f_3^2	f_{10}^2	f_{26}^2*	f_{33}^2	f_{40}^2	f_{47}^2*
n_4^2	f_4^2	f_{11}^2	f_{18}^2*	f_{34}^2	f_{41}^2	f_{48}^2*
n_5^2	f_5^2	f_{12}^2	f_{19}^2*	f_{35}^2	f_{42}^2	f_{49}^2*
n_6^2	f_6^2	f_{13}^2	f_{20}^2*	f_{27}^2	f_{43}^2	f_{50}^2*
n_7^2	f_7^2	f_{14}^2	f_{21}^2*	f_{28}^2	f_{44}^2	f_{51}^2*
n_8^2	f_8^2	f_{15}^2	f_{22}^2*	f_{29}^2	f_{36}^2	f_{52}^2*

Figure 7.8: Example on assignment of fragments to nodes using Q-rot declustering.

Based on Equation 7.1 the cardinality of the minimal intersection of two different sites s_x and s_y (i.e. $x \neq y$) can be written:

$$L \geq \left\lceil \frac{F}{N_s N_s} \right\rceil = \left\lceil \frac{N_s \cdot Z_r}{N_s N_s} \right\rceil = \left\lceil \frac{Z_r}{N_s} \right\rceil$$

Since the replica fanout can not be larger than the number of nodes at a site (i.e. $1 \leq Z_r \leq N_s$) we get:

$$L \geq 1$$

Hence, we will here show that $L = 1$ if 7.4 and 7.5 hold and therefore minimal intersection exists.

Thus, if $L = 1$ we have:

$$\forall i, j \;\mid\; i \neq j \wedge \mathcal{N}_x(i) = \mathcal{N}_x(j) \Rightarrow \mathcal{N}_y(i) \neq \mathcal{N}_y(j) \tag{7.6}$$

Equation 7.6 says that for all pairs of different fragments f_i and f_j that are assigned to the same node on site x, they are assigned to two different nodes on site y. If this equation does not hold L must be greater than 1. If Equation 7.6 is violated we can write:

$$\mathcal{N}_x(i) = \mathcal{N}_x(j) \quad \wedge \quad \mathcal{N}_y(i) = \mathcal{N}_y(j)$$

which expands to:

$$\begin{aligned} (i + (i \text{ div } N_s)q_x) \text{ mod } N_s &= (j + (j \text{ div } N_s)q_x) \text{ mod } N_s && (7.7)\\ &\wedge \\ (i + (i \text{ div } N_s)q_y) \text{ mod } N_s &= (j + (j \text{ div } N_s)q_y) \text{ mod } N_s && (7.8) \end{aligned}$$

The fragment numbers i and j can be written as respectively:

$$i = i' + i'' N_s \quad \text{and} \quad j = j' + j'' N_s$$

where $0 \leq i', j' < N_s$ and $0 \leq i'', j'' < Z_r$. Due to *the division theorem* (see for example theorem 3-10 in [Gil76], page 83) there exist unique solutions for i', i'', j', and j''. Also note that:

$$i \text{ div } N_s = (i' + i'' N_s) \text{ div } N_s = (i' \text{ div } N_s) + i'' = i''$$

Using this alternative form for i we can rewrite Equation 7.7 and 7.8 into:

$$\begin{aligned} (i' + i'' N_s + i'' q_x) \text{ mod } N_s &= (j' + j'' N_s + j'' q_x) \text{ mod } N_s \\ &\wedge \\ (i' + i'' N_s + i'' q_y) \text{ mod } N_s &= (j' + j'' N_s + j'' q_y) \text{ mod } N_s \end{aligned}$$

The $i'' N_s$ and $j'' N_s$ parts are both cancelled by the mod's giving:

$$\begin{aligned}(i' + i''q_x) \bmod N_s &= (j' + j''q_x) \bmod N_s \\ &\wedge \\ (i' + i''q_y) \bmod N_s &= (j' + j''q_y) \bmod N_s\end{aligned}$$

The mod's can be replaced by adding or subtraction N_s an unknown number of times to one of the sides of the equations. Doing this we get:

$$(i' + i''q_x) - (j' + j''q_x) = (i' - j') + (i'' - j'')q_x = k_x \cdot N_s \tag{7.9}$$

$$\wedge$$

$$(i' + i''q_y) - (j' + j''q_y) = (i' - j') + (i'' - j'')q_y = k_y \cdot N_s \tag{7.10}$$

where k_x and k_y are integers. By subtracting Equation 7.10 from Equation 7.9 we get:

$$(i'' - j'')(q_x - q_y) = (k_x - k_y)N_s \tag{7.11}$$

Let $d_{x,y} = |q_x - q_y|$ and $k = |k_x - k_y|$. Equation 7.11 can then be written:

$$|i'' - j''| \cdot d_{x,y} = k \cdot N_s \tag{7.12}$$

Let $\gamma = \text{gcd}(d_{x,y}, N_s)$ and set $d_{x,y} = \gamma \cdot \hat{d}_{x,y}$ and $N_s = \gamma \cdot \hat{N}_c$. Equation 7.12 then becomes:

$$|i'' - j''| \cdot \hat{d}_{x,y} = k \cdot \hat{N}_c \tag{7.13}$$

Note that all the variables in Equation 7.13 must be integers. With this in mind and the fact that $\hat{d}_{x,y}$ and $\hat{N}_c$ have no common factors, $\hat{d}_{x,y}$ must be a factor of k. If we write $k = \hat{d}_{x,y} \cdot k'$ the equation simplifies to:

$$|i'' - j''| = k' \cdot \hat{N}_c \tag{7.14}$$

Remember that $q_x \neq q_y$ from the prerequisite in Equation 7.4 implying that $|q_x - q_y| = d_{x,y} = \gamma\hat{d}_{x,y} \neq 0$ so we are free to divide with $\hat{d}_{x,y}$.

By studying Equation 7.14 we see that there are two cases when the equation might hold and which must be investigated further:

Case 1: $|i'' - j''| = k' = 0$

Case 2: $\text{gcd}(|i'' - j''|, \hat{N}_c) = \hat{N}_c$

Assume that $|i'' - j''| = 0$, i.e. $i'' = j''$. Insert this into Equation 7.9:

$$(i' - j') + (i'' - j'')q_x = i' - j' = k_x \cdot N_s \tag{7.15}$$

Since $0 \leq i' < N_s$ and $0 \leq j'' < N_s$ then $-N_s < i' - j' < N_s$. Thus, the only integer solution to Equation 7.15 is $i' = j'$. But, since we assumed that $i'' = j''$ then $i = j$ violating the $i \neq j$ requirement from Equation 7.6, i.e. **Case 1 is impossible**.

Case 2 demands that $\hat{N}_c$ must be a factor of $|i'' - j''|$ for Equation 7.14 to have an integer solution. This means that $\hat{N}_c \leq |i'' - j''|$. Knowing that $0 \leq i'' < Z_r$ and $0 \leq j'' < Z_r$ we find that $0 \leq |i'' - j''| < Z_r$, thus

$$\hat{N}_c \leq |i'' - j''| < Z_r \tag{7.16}$$

Looking back to the prerequisite given by Equation 7.5 we have:

$$\begin{aligned} \text{gcd}(|q_x - q_y|, N_s) &\leq \frac{N_s}{Z_r} \\ \gamma &\leq \frac{N_s}{Z_r} \\ Z_r &\leq \frac{N_s}{\gamma} = \hat{N}_c \end{aligned}$$

which contradicts with what we just found in Equation 7.16, i.e. **Case 2 is impossible**.

Since both case 1 and 2 are impossible when the prerequisites hold then L can't be greater than 1 and we can conclude that QD is a member of the MISD family.

$\square$

7.2.3 Configuration rules

The practical consequences of this proof is that the number of nodes in each cluster, the replica fanout and the rotation quotient must be carefully selected. The relationship given by Equation 7.5 is not trivial and can not be applied directly. By investigating this equation some practical rules have been developed which will assist the application of this new declustering strategy. These rules are listed and explained below. It should be noted that it is sufficient that only one of these rules is satisfied for Equation 7.5 to hold.

Rule 1:
If N_s is a prime, then Z_r and the q_x's can be chosen without other restrictions than that all the q_x's must be different.

If N_s is a prime number, then $\text{gcd}(|q_x - q_y|, N_s) = 1$ since both q_x and q_y are less than N_s. This means that Equation 7.5 holds as long as $Z_r \leq N_s$. Thus as long as N_s is a prime, then there is no restrictions on the rotation quotients and the fanout. This rule obviously sets a strict restriction on the number of nodes that the system can have compared with the other declustering strategies.

Rule 2:
If N_s is a product of two primes P_l and P_h where $P_l \leq P_h$ and $Z_r \leq P_h$, then $|q_x - q_y| < P_h$ must hold for all pairs of sites (s_x, s_y).

Since N_s is a product of the two primes P_l and P_h then $\gcd(|q_x - q_y|, N_s) \in \{1, P_l, P_h, N_s\}$. If $|q_x - q_y| < P_h$ then $\gcd(|q_x - q_y|, N_s) \in \{1, P_l\}$. Thus as long as $P_l \leq \frac{N_s}{Z_r}$ is true, then Equation 7.5 holds. By manipulating this relation we get $Z_r \leq \frac{N_s}{P_l} = \frac{P_h P_l}{P_l} = P_h$.

Let q_{min} and q_{max} be the smallest and the largest rotation quotient respectively then $|q_x - q_y| \leq q_{\text{max}} - q_{\text{min}}$ holds for all pairs of sites (s_x, s_y). The maximum number of sites supported is limited by the number of different rotation quotients. If all quotients between q_{min} and q_{max} are assigned the maximum number of sites $S_{\text{max}} = q_{\text{max}} - q_{\text{min}} + 1$. Thus rule 2 can be rewritten to:

Rule 2':
If N_s is a product of two primes P_l and P_h where $P_l \leq P_h$ and $Z_r \leq P_h$ then $S \leq P_h$ must hold.

Example 7.2 shows an application of rule 2.

Example 7.2:

Let $N_s = 14 = 2 \cdot 7$. Then we can have $Z_r = 7$ and $q_s \in \{0, 1, 2, 3, 4, 5, 6\}$. The maximum number of sites is $S = 7$. Figure 7.9 shows the declustering of fragments over four sites with rotation quotients of respectively 0, 3, 5, and 6. The fragments assigned to node n_0^0 are labeled with an asterisk ($*$) and it is easy to see that all these fragments are assigned to different nodes on the three other sites.

□

Rule 3:
If N_s is a product of k primes, $P_1, P_2, \cdots P_k$, where $P_i \leq P_{i+1}$ and $Z_r \leq P_2 \cdot P_3 \cdots P_k$, then $|q_x - q_y| < P_2$ must hold for all pairs of sites (s_x, s_y).

This is a generalization of rule 2. Since N_s is a product of primes then $\gcd(|q_x - q_y|, N_s)$ can be written as a product of a subset of those primes. If $|q_x - q_y| < P_2$, then $\gcd(|q_x - q_y|, N_s) \in \{1, P_1\}$. Thus as long as $P_1 \leq \frac{N_s}{Z_r}$ is true then Equation 7.5 holds. By manipulating this relation we get $Z_r \leq \frac{N_s}{P_1}$.

Substituting the rotation quotients with the number of sites in rule 3 we get:

Rule 3':
If N_s is a product of k primes, $P_1, P_2, \cdots P_k$, where $P_i \leq P_{i+1}$ and $Z_r \leq P_2 \cdot P_3 \cdots P_k$, then $S \leq P_2$ must hold.

Rule 4:
If N_s is a product of primes P_i, then $|q_x - q_y| < P_i$ must hold for all pairs of sites (s_x, s_y) and all P_i's.

Since $|q_x - q_y| < P_i$ for all P_i then $\gcd(|q_x - q_y|, N_s) = 1$. Thus $1 \leq \frac{N_s}{Z_r}$ and $Z_r \leq N_s$. Since this restriction already is inherent in the definition of replica fanout, rule 4 actually allows Z_r to be selected without restrictions.

Site 0 with $q_0 = 0$

n_{0}^{0}	$f_{0}^{0}*$	$f_{14}^{0}*$	$f_{28}^{0}*$	$f_{42}^{0}*$	$f_{56}^{0}*$	$f_{70}^{0}*$	$f_{84}^{0}*$
n_{1}^{0}	f_{1}^{0}	f_{15}^{0}	f_{29}^{0}	f_{43}^{0}	f_{57}^{0}	f_{71}^{0}	f_{85}^{0}
n_{2}^{0}	f_{2}^{0}	f_{16}^{0}	f_{30}^{0}	f_{44}^{0}	f_{58}^{0}	f_{72}^{0}	f_{86}^{0}
n_{3}^{0}	f_{3}^{0}	f_{17}^{0}	f_{31}^{0}	f_{45}^{0}	f_{59}^{0}	f_{73}^{0}	f_{87}^{0}
n_{4}^{0}	f_{4}^{0}	f_{18}^{0}	f_{32}^{0}	f_{46}^{0}	f_{60}^{0}	f_{74}^{0}	f_{88}^{0}
n_{5}^{0}	f_{5}^{0}	f_{19}^{0}	f_{33}^{0}	f_{47}^{0}	f_{61}^{0}	f_{75}^{0}	f_{89}^{0}
n_{6}^{0}	f_{6}^{0}	f_{20}^{0}	f_{34}^{0}	f_{48}^{0}	f_{62}^{0}	f_{76}^{0}	f_{90}^{0}
n_{7}^{0}	f_{7}^{0}	f_{21}^{0}	f_{35}^{0}	f_{49}^{0}	f_{63}^{0}	f_{77}^{0}	f_{91}^{0}
n_{8}^{0}	f_{8}^{0}	f_{22}^{0}	f_{36}^{0}	f_{50}^{0}	f_{64}^{0}	f_{78}^{0}	f_{92}^{0}
n_{9}^{0}	f_{9}^{0}	f_{23}^{0}	f_{37}^{0}	f_{51}^{0}	f_{65}^{0}	f_{79}^{0}	f_{93}^{0}
n_{10}^{0}	f_{10}^{0}	f_{24}^{0}	f_{38}^{0}	f_{52}^{0}	f_{66}^{0}	f_{80}^{0}	f_{94}^{0}
n_{11}^{0}	f_{11}^{0}	f_{25}^{0}	f_{39}^{0}	f_{53}^{0}	f_{67}^{0}	f_{81}^{0}	f_{95}^{0}
n_{12}^{0}	f_{12}^{0}	f_{26}^{0}	f_{40}^{0}	f_{54}^{0}	f_{68}^{0}	f_{82}^{0}	f_{96}^{0}
n_{13}^{0}	f_{13}^{0}	f_{27}^{0}	f_{41}^{0}	f_{55}^{0}	f_{69}^{0}	f_{83}^{0}	f_{97}^{0}

Site 1 with $q_1 = 3$

n_{0}^{1}	$f_{0}^{1}*$	f_{25}^{1}	f_{36}^{1}	f_{47}^{1}	f_{58}^{1}	f_{83}^{1}	f_{94}^{1}
n_{1}^{1}	f_{1}^{1}	f_{26}^{1}	f_{37}^{1}	f_{48}^{1}	f_{59}^{1}	$f_{70}^{1}*$	f_{95}^{1}
n_{2}^{1}	f_{2}^{1}	f_{27}^{1}	f_{38}^{1}	f_{49}^{1}	f_{60}^{1}	f_{71}^{1}	f_{96}^{1}
n_{3}^{1}	f_{3}^{1}	$f_{14}^{1}*$	f_{39}^{1}	f_{50}^{1}	f_{61}^{1}	f_{72}^{1}	f_{97}^{1}
n_{4}^{1}	f_{4}^{1}	f_{15}^{1}	f_{40}^{1}	f_{51}^{1}	f_{62}^{1}	f_{73}^{1}	$f_{84}^{1}*$
n_{5}^{1}	f_{5}^{1}	f_{16}^{1}	f_{41}^{1}	f_{52}^{1}	f_{63}^{1}	f_{74}^{1}	f_{85}^{1}
n_{6}^{1}	f_{6}^{1}	f_{17}^{1}	$f_{28}^{1}*$	f_{53}^{1}	f_{64}^{1}	f_{75}^{1}	f_{86}^{1}
n_{7}^{1}	f_{7}^{1}	f_{18}^{1}	f_{29}^{1}	f_{54}^{1}	f_{65}^{1}	f_{76}^{1}	f_{87}^{1}
n_{8}^{1}	f_{8}^{1}	f_{19}^{1}	f_{30}^{1}	f_{55}^{1}	f_{66}^{1}	f_{77}^{1}	f_{88}^{1}
n_{9}^{1}	f_{9}^{1}	f_{20}^{1}	f_{31}^{1}	$f_{42}^{1}*$	f_{67}^{1}	f_{78}^{1}	f_{89}^{1}
n_{10}^{1}	f_{10}^{1}	f_{21}^{1}	f_{32}^{1}	f_{43}^{1}	f_{68}^{1}	f_{79}^{1}	f_{90}^{1}
n_{11}^{1}	f_{11}^{1}	f_{22}^{1}	f_{33}^{1}	f_{44}^{1}	f_{69}^{1}	f_{80}^{1}	f_{91}^{1}
n_{12}^{1}	f_{12}^{1}	f_{23}^{1}	f_{34}^{1}	f_{45}^{1}	$f_{56}^{1}*$	f_{81}^{1}	f_{92}^{1}
n_{13}^{1}	f_{13}^{1}	f_{24}^{1}	f_{35}^{1}	f_{46}^{1}	f_{57}^{1}	f_{82}^{1}	f_{93}^{1}

Site 2 with $q_2 = 5$

n_{0}^{2}	$f_{0}^{2}*$	f_{23}^{2}	f_{32}^{2}	f_{55}^{2}	f_{64}^{2}	f_{73}^{2}	f_{96}^{2}
n_{1}^{2}	f_{1}^{2}	f_{24}^{2}	f_{33}^{2}	$f_{42}^{2}*$	f_{65}^{2}	f_{74}^{2}	f_{97}^{2}
n_{2}^{2}	f_{2}^{2}	f_{25}^{2}	f_{34}^{2}	f_{43}^{2}	f_{66}^{2}	f_{75}^{2}	$f_{84}^{2}*$
n_{3}^{2}	f_{3}^{2}	f_{26}^{2}	f_{35}^{2}	f_{44}^{2}	f_{67}^{2}	f_{76}^{2}	f_{85}^{2}
n_{4}^{2}	f_{4}^{2}	f_{27}^{2}	f_{36}^{2}	f_{45}^{2}	f_{68}^{2}	f_{77}^{2}	f_{86}^{2}
n_{5}^{2}	f_{5}^{2}	$f_{14}^{2}*$	f_{37}^{2}	f_{46}^{2}	f_{69}^{2}	f_{78}^{2}	f_{87}^{2}
n_{6}^{2}	f_{6}^{2}	f_{15}^{2}	f_{38}^{2}	f_{47}^{2}	$f_{56}^{2}*$	f_{79}^{2}	f_{88}^{2}
n_{7}^{2}	f_{7}^{2}	f_{16}^{2}	f_{39}^{2}	f_{48}^{2}	f_{57}^{2}	f_{80}^{2}	f_{89}^{2}
n_{8}^{2}	f_{8}^{2}	f_{17}^{2}	f_{40}^{2}	f_{49}^{2}	f_{58}^{2}	f_{81}^{2}	f_{90}^{2}
n_{9}^{2}	f_{9}^{2}	f_{18}^{2}	f_{41}^{2}	f_{50}^{2}	f_{59}^{2}	f_{82}^{2}	f_{91}^{2}
n_{10}^{2}	f_{10}^{2}	f_{19}^{2}	$f_{28}^{2}*$	f_{51}^{2}	f_{60}^{2}	f_{83}^{2}	f_{92}^{2}
n_{11}^{2}	f_{11}^{2}	f_{20}^{2}	f_{29}^{2}	f_{52}^{2}	f_{61}^{2}	$f_{70}^{2}*$	f_{93}^{2}
n_{12}^{2}	f_{12}^{2}	f_{21}^{2}	f_{30}^{2}	f_{53}^{2}	f_{62}^{2}	f_{71}^{2}	f_{94}^{2}
n_{13}^{2}	f_{13}^{2}	f_{22}^{2}	f_{31}^{2}	f_{54}^{2}	f_{63}^{2}	f_{72}^{2}	f_{95}^{2}

Site 3 with $q_3 = 6$

n_{0}^{3}	$f_{0}^{3}*$	f_{22}^{3}	f_{30}^{3}	f_{52}^{3}	f_{60}^{3}	f_{82}^{3}	f_{90}^{3}
n_{1}^{3}	f_{1}^{3}	f_{23}^{3}	f_{31}^{3}	f_{53}^{3}	f_{61}^{3}	f_{83}^{3}	f_{91}^{3}
n_{2}^{3}	f_{2}^{3}	f_{24}^{3}	f_{32}^{3}	f_{54}^{3}	f_{62}^{3}	$f_{70}^{3}*$	f_{92}^{3}
n_{3}^{3}	f_{3}^{3}	f_{25}^{3}	f_{33}^{3}	f_{55}^{3}	f_{63}^{3}	f_{71}^{3}	f_{93}^{3}
n_{4}^{3}	f_{4}^{3}	f_{26}^{3}	f_{34}^{3}	$f_{42}^{3}*$	f_{64}^{3}	f_{72}^{3}	f_{94}^{3}
n_{5}^{3}	f_{5}^{3}	f_{27}^{3}	f_{35}^{3}	f_{43}^{3}	f_{65}^{3}	f_{73}^{3}	f_{95}^{3}
n_{6}^{3}	f_{6}^{3}	$f_{14}^{3}*$	f_{36}^{3}	f_{44}^{3}	f_{66}^{3}	f_{74}^{3}	f_{96}^{3}
n_{7}^{3}	f_{7}^{3}	f_{15}^{3}	f_{37}^{3}	f_{45}^{3}	f_{67}^{3}	f_{75}^{3}	f_{97}^{3}
n_{8}^{3}	f_{8}^{3}	f_{16}^{3}	f_{38}^{3}	f_{46}^{3}	f_{68}^{3}	f_{76}^{3}	$f_{84}^{3}*$
n_{9}^{3}	f_{9}^{3}	f_{17}^{3}	f_{39}^{3}	f_{47}^{3}	f_{69}^{3}	f_{77}^{3}	f_{85}^{3}
n_{10}^{3}	f_{10}^{3}	f_{18}^{3}	f_{40}^{3}	f_{48}^{3}	$f_{56}^{3}*$	f_{78}^{3}	f_{86}^{3}
n_{11}^{3}	f_{11}^{3}	f_{19}^{3}	f_{41}^{3}	f_{49}^{3}	f_{57}^{3}	f_{79}^{3}	f_{87}^{3}
n_{12}^{3}	f_{12}^{3}	f_{20}^{3}	$f_{28}^{3}*$	f_{50}^{3}	f_{58}^{3}	f_{80}^{3}	f_{88}^{3}
n_{13}^{3}	f_{13}^{3}	f_{21}^{3}	f_{29}^{3}	f_{51}^{3}	f_{59}^{3}	f_{81}^{3}	f_{89}^{3}

Figure 7.9: An example of assignment of fragments to four sites using Q-rot declustering.

Substituting the rotation quotients with the number of sites in rule 4 we get:

Rule 4':
If N_s is a product of primes P_i, then $S \leq P_i$ must hold for all P_i's.

Rule 4' leads to an important result: For two site configurations any number of nodes and any replica fanout can be used. This is due to the fact that the smallest existing prime is 2.

Rule 5:
Choose N_s such that $N_s \geq |q_x - q_y| Z_r$ holds for all pairs of sites (s_x, s_y).

Since the greatest common divisor between any two positive integers must be less or equal to both integers we can write gcd$(|q_x - q_y|, N_s) \leq |q_x - q_y|$. Thus if $|q_x - q_y| \leq \frac{N_s}{Z_r}$ holds, then also gcd$(|q_x - q_y|, N_s) \leq \frac{N_s}{Z_r}$ must be true.

The same rule as a function of number of sites S becomes:

Rule 5':
Choose N_s such that $N_s \geq (S-1)Z_r$ holds.

Thus for a given configuration of S sites and a fanout Z_r the minimum intersecting set property holds as long as the number of nodes in each cluster is greater than or equal to $(S-1)Z_r$. Example 7.3 demonstrates the use of rule 5.

Example 7.3:

Let $Z_r = 8$ and $S = 3$ then there must be at least $(3-1) \cdot 8 = 16$ nodes at each site. Note that rule 4 gives no absolute minimum value of N_s since there might exist some values that still satisfy Equation 7.5. When $N_s \in \{11, 13\}$ then N_s is a prime and rule 1 holds (also 2,3,5, and 7 are primes but remember that $Z_r \leq N_s$). When $N_s \in \{9, 15\}$ then rule 4 can be used.

$\square$

The need for the rules certainly makes it more complicated to use QD compared to the other declustering schemes, but for most practical purposes a system designer should be able to find a rule that suits his or her use.

7.2.4 Masking failures

To be able to mask node and site failures we need to know which of the hot stand-by fragment replicas that should take over for the unavailable primaries. For two site systems this is trivial, just select the second replica. For higher number of replicas we need an alternative primary replica function $\mathcal{P}'(i)$. The goal for this function is to spread the masking work over the nodes sharing fragments with the failed node. One possible solution is to divide each of the lost fragments in $R-1$ non-overlapping subfragments. The nodes of one of the sites sharing fragments take the first subfragment, those on the second site takes the second subfragment etc.

This strategy results in that the takeover fanout gets the value $Z_t = (R-1)Z_r$.

```
function mapnode(fragno, site)
begin
    avail ← $N_s$;
    for vnode ← 0 to avail-1 do
        nodemap[vnode] ← vnode;
    end do;
    forever do
        vnode ← (fragno + (fragno div $N_s$) · rq[site]) mod avail;
        if state(nodemap[vnode])=up then
            return nodemap[vnode];
        end if;
        avail ← avail − 1;
        if avail = 0 then
            return -1;
        end if;
        nodemap[vnode:avail-1] ← nodemap[vnode+1:avail];
    end do;
end;
```

rq[s]	= rotation quotient for site ’s’
state(n)	= state of node, ’up’ or ’down’
avail	= number of nodes believed to be ’up’
nodemap	= array of nodes believed to be ’up’

Figure 7.10: Pseudocode for mapping a fragment replica to a node when a distributed spare is used.

7.2.5 Self repair

Self repair for dedicated sparing is trivial. The refragmentation fanout Z_f is equal to 1 since all data is reproduced on a single node. The site fanout Z_s is also 1 since data is only reorganized on the site where the node failed. However, note that the nodes must be given logical node numbers for the assignment function $\mathcal{N}_x(i)$ to work after a node failure followed by repair.

The problem with dedicated sparing is that it is possible to run out of spare nodes. If this happens distributed sparing can be used as a fallback strategy until nodes have been replaced.

For dedicated sparing it is necessary to remap fragment replicas assigned to failed nodes to operating ones. Figure 7.10 shows pseudocode for a function implementing such a remapping. It works by trying to assign a fragment as if no nodes are down. If the resulting node is down, the node is removed from the list of known nodes and the remapping is retried with one node less. This is repeated until an operating node is found. This function is idempotent with respect to node failures, i.e. nodes can fail and be repaired independent of each other without needing to move other fragments than those directly affected. Such a remapping can potentially reduce both fanout and stride values.

7.2.6 Fragment stride

The fragment stride Δ_F depends on the difference between the rotation quotients used and varies between $(S-1)N_s + 1$ and SN_s.

Masking node failures means that the primary function for some fragments moves to other nodes and sites. This will to some degree reduce the stride. Masking site failures is more serious since so many nodes disappear at the same time reducing the average fragment stride with at least the number of nodes on a site.

A dedicated spare self repair strategy will not reduce the stride, but if a distributed spare strategy is used, the regular assignment scheme will be distorted and the stride reduced. This penalty is of the same magnitude as masking node failures.

7.3 Summary

Table 7.2 show a summary of the parameters for QD both using distributed and dedicated spare.

As the table shows, Q-rot Declustering gives the system designer a wide range of options on how to dimension the system. It supports both types of self repair and can support planned repair and single site service. The most negative aspect is the restrictions set by the configuration rules which complicates the use of the scheme.

Up to now we have not looked into the availability offered by the different declustering strategies. This has been left to the next chapter which will look on different types of errors and provide analytic models for computing the unavailability when the system parameters are given.

		Minimum Intersecting Sets Declustering	
		Q-rot Declustering	
		dedicated spare	distributed spare
Number of sites	S	R	R
Number of replicas	R	≥ 2	≥ 2
Number of fragments	F	$N_s \cdot Z_r$	$N_s \cdot Z_r$
Replica fanout	Z_r	$1 \leq Z_r \leq N_s$	$1 \leq Z_r \leq N_s$
Takeover fanout	Z_t	$(R-1)Z_r$	$(R-1)Z_r$
Refragmentation fanout	Z_f	1	$1 \leq Z_f \leq Z_r$
Site fanout	Z_s	1	1
Storage efficiency		$\frac{N_s}{R(N_s+N_s')}$	$\frac{1}{R}$
Range stride	Δ_R	$\frac{N_s(R-1)+1}{F} \leq \Delta_R \leq \frac{N_sR}{F}$	$\frac{N_s(R-1)+1}{F} \leq \Delta_R \leq \frac{N_sR}{F}$
Fragment stride	Δ_F	$N_s(R-1)+1 \leq \Delta_F \leq N_sR$	$N_s(R-1)+1 \leq \Delta_F \leq N_sR$
Self repair		dedicated spare	distributed spare
Planned repair		yes	yes
Homogeneous		no	yes
Single site service		yes	yes
Simplicity		0	-

Table 7.2: MISD parameters.

Chapter 8

An unavailability analysis

There are various reasons for service unavailability. For the user of the service the reason is of less importance whether it is a software bug or an earthquake. The service is unavailable and the user can not get his or her work done. For the system designer this is different. If he or her knows the types of events that potentially can cause outage and the frequency of those events, precautions can be taken for a suitable set of these type of events so the desired availability can be reached. But even though precautions are taken, it is not possible to achieve 100% continuous service availability. All methods for fault-tolerance only reduce the probability of a failure and do not fully eliminate it.

This chapter looks into some of the reasons for unavailability for a fault-tolerant database server with a self repair capability and analyze their contribution to the total system unavailability. The chapter also looks into design options and the consequence these have to the duration and frequency of the unavailability.

The unavailability reasons considered are:

Node-node double-faults (U_{NN}) Simultaneous single node failures causing the loss of all replicas of one or more fragments.

Site-node double-faults (U_{SN}) Node failures following a site failure causing the loss of all replicas of one or more fragments.

Node-site double-faults (U_{NS}) A site failure occurring while a set of nodes are down causing the loss of all replicas of one or more fragments.

Site-site double-faults (U_{SS}) Simultaneous site failures causing the loss of all fragment replicas.

The total system unavailability is a sum of the unavailability caused by these reasons. Thus:

$$U = U_{NN} + U_{SN} + U_{NS} + U_{SS} \tag{8.1}$$

This model does not model communication failures and the risk of running out of spare capacity. Most communication failures can be described as either site or node failures and are in this way incorporated into the model. We assume that spare capacity is sufficiently large for the replacement frequency used so that the probability of running out of spares are negligible. These types of failures will not be discussed further in this work.

Note that the unavailability is based on the *loss* of data and the reconstruction of lost data. We do not consider transient errors causing a short temporary loss of database service. An OS reboot or DBMS restart will not cause any database data to be permanently lost if normal log and recovery mechanisms are used. If a node storing the only reminding replica of a fragment is rebooted, the fragment is unavailable only until the reboot has finished and DBMS recovery is complete. On the other hand, if the reboot causes the database content on that node to be corrupted we have a permanent data loss which the model copes with.

Previous unavailability analysis of declustering strategies include [CK89] and [HD91]. Both restrict their study to node-node double-faults and study some declustering strategies specifically. [CLG$^+$94] analyses the mean time to data loss for RAID architectures incorporating the probability of page, disk, and system errors.

This work will take a more general approach and provides a tool for evaluating declustering strategies in general based on their characteristic parameters: Number of sites, nodes, spares, replicas, fragments, and fanout values.

This chapter is divided into two sections. The first analyzes the unavailability caused by node-node double-faults. The second looks briefly into unavailability caused by combinations of site failures. The analyses are done in a transaction processing context.

8.1 Node-node double-faults

It is mainly with respect to node-node double-fault unavailability that the declustering strategies differ from each other. This section will therefore make much effort into analyzing the effects the characteristic parameters of the declustering strategies have on the unavailability. This does not only make it possible to choose the best strategy but also to find the most desirable values which again might lead to the development of new and improved methods.

But, before we can do this analysis we need to develop a failure model reflecting the system behavior. Below this is done in two steps. First a detailed model is developed for a two site configuration. The second step is to develop a simplified more general multi site model which is verified by comparing it with the detailed two site model.

8.1.1 Two site failure model

Figure 8.1 shows the state diagram for a DBMS server with N nodes and no dedicated spare nodes storing two replicas of all data and implementing self repair of lost replicas. The system is in state S_i where $0 \leq i < N$, when at least one replica of all data still is available and fragment replicas lost by i node failures not have been reproduced by self repair. If both replicas of one or more fragments are lost due to node failures the system is in state S_a and is considered unavailable. Symbols used in this chapter are listed in Appendix A.

The failure model does the following assumptions:

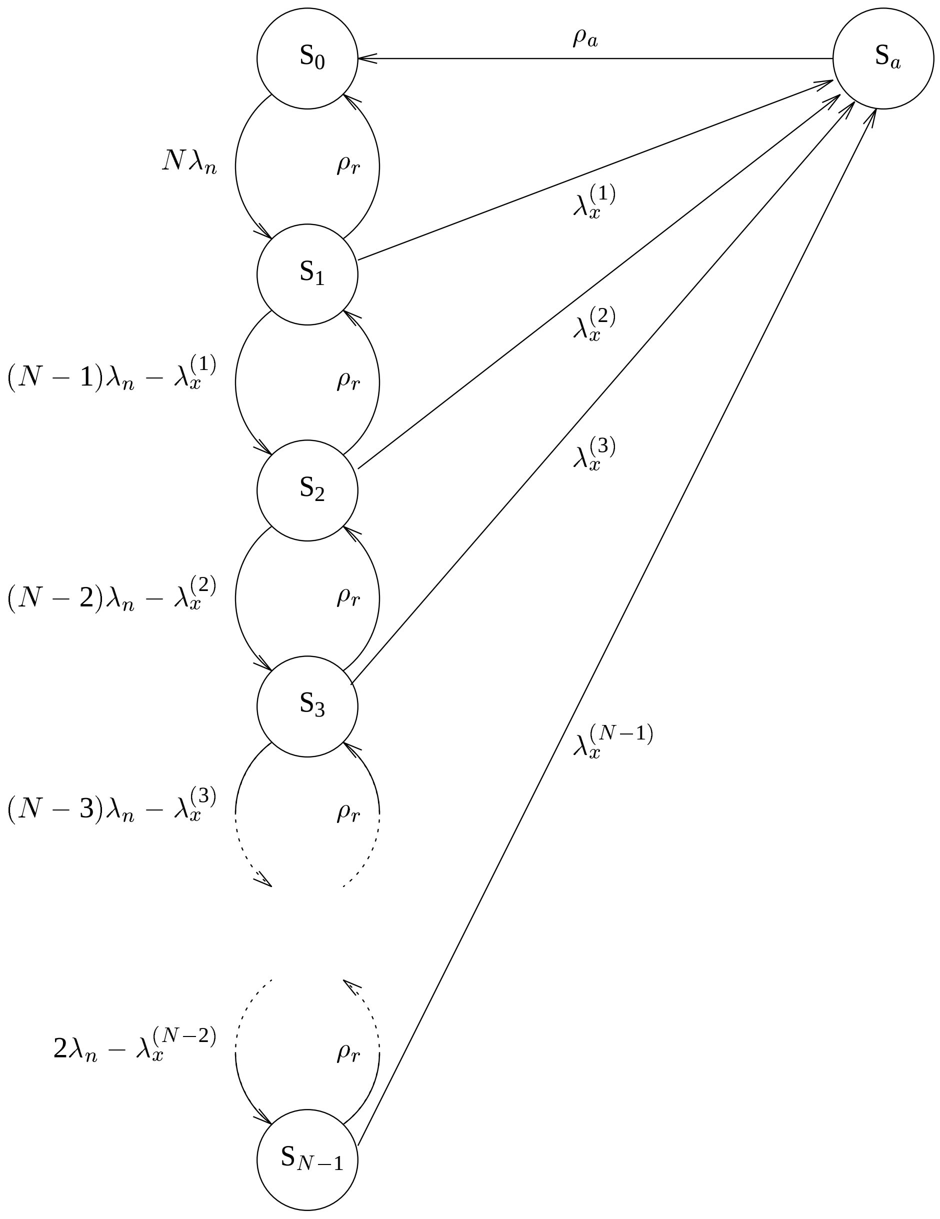


Figure 8.1: Failure state transition diagram for node failures.

- All node failures are causing permanent loss of data, i.e. there is no unavailability caused by transient node failures. A transient failure can be handled by doing a fast restart and recovery and no data will be lost.
- Independent nodes have independent failure modes. Nodes sharing common fragments are not independent. The failure of one node will only change the probability of a failure for the nodes sharing common fragments with the failed node. This accounts for correlated errors between nodes storing replicas of the same fragment.
- Self repair is performed one node after another. After a node failure, self repair is not started before any ongoing repair has completed. This avoids overloading nodes and communication network with concurrent reproduction of lost fragment replicas.
- Node failure, replica repair, and system recovery is all exponentially distributed with parameters respectively λ_n, ρ_r, and ρ_a. (Replica repair times will in practice be relatively constant. Even though this violates the assumption we will use it for simplicity reasons.) When a node has failed and yet not repaired, the failure rate for nodes sharing common fragments with it increases by a factor G to $G\lambda_n$.
- A node failure causing double failure is exponentially distributed with parameter $\lambda_x^{(i)}$ when the system is in state S$_i$.
- The tuples are uniformly distributed over the nodes.
- The load is uniformly distributed over the database tuples.
- The load is uniformly distributed over time.

In a stable state, the sum of the intensities of transitions into a state is equal to the intensities of transitions out of the same state. Hence we can write:

$$
\begin{aligned}
N\lambda_n p_0 &= \rho_r p_1 + \rho_a p_a \\
(((N-1)\lambda_n - \lambda_x^{(1)}) + \lambda_x^{(1)} + \rho_r)p_1 &= N\lambda_n p_0 + \rho_r p_2 \\
(((N-2)\lambda_n - \lambda_x^{(2)}) + \lambda_x^{(2)} + \rho_r)p_2 &= ((N-1)\lambda_n - \lambda_x^{(1)})p_1 + \rho_r p_3 \\
(((N-3)\lambda_n - \lambda_x^{(3)}) + \lambda_x^{(3)} + \rho_r)p_3 &= ((N-2)\lambda_n - \lambda_x^{(2)})p_2 + \rho_r p_4 \\
&\vdots \\
(\lambda_x^{(N-1)} + \rho_r)p_{N-1} &= (2\lambda_n - \lambda_x^{(N-2)})p_{N-2} \\
\rho_a p_a &= \lambda_x^{(1)}p_1 + \lambda_x^{(2)}p_2 + \lambda_x^{(3)}p_3 + \cdots + \lambda_x^{(N-1)}p_{N-1}
\end{aligned}
$$

p_i denotes the probability of being in state S$_i$. These equations can be simplified into:

$$
\begin{aligned}
N\lambda_n p_0 &= \rho_r p_1 + \rho_a p_a \\
((N-1)\lambda_n + \rho_r)p_1 &= N\lambda_n p_0 + \rho_r p_2 \\
((N-2)\lambda_n + \rho_r)p_2 &= ((N-1)\lambda_n - \lambda_x^{(1)})p_1 + \rho_r p_3 \\
((N-3)\lambda_n + \rho_r)p_3 &= ((N-2)\lambda_n - \lambda_x^{(2)})p_2 + \rho_r p_4
\end{aligned}
$$

$$\begin{aligned} &\vdots \\ (\lambda_x^{(N-1)} + \rho_r)p_{N-1} &= (2\lambda_n - \lambda_x^{(N-2)})p_{N-2} \\ \rho_a p_a &= \lambda_x^{(1)}p_1 + \lambda_x^{(2)}p_2 + \lambda_x^{(3)}p_3 + \cdots + \lambda_x^{(N-1)}p_{N-1} \end{aligned}$$

Even though this is a set of $N+1$ equations with $N+1$ unknowns ($p_i, 0 \leq i < N$ and p_a), there exists no distinct solution. The additional knowledge that makes the system solvable is the fact that the sum of the probabilities of being in any state must be 1, i.e.:

$$(\sum_{i=0}^{N-1} p_i) + p_a = 1 \tag{8.2}$$

By replacing one of the equations in the system with Equation 8.2 and writing it on matrix form we get:

$$\begin{bmatrix} -N\lambda_n & \rho_r & 0 & 0 & 0 & 0 & \rho_a \\ N\lambda_n & -(N-1)\lambda_n - \rho_r & \rho_r & 0 & 0 & 0 & 0 \\ 0 & (N-1)\lambda_n - \lambda_x^{(1)} & -(N-2)\lambda_n - \rho_r & \rho_r \ \cdots & 0 & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & 0 & 0 & 2\lambda_n - \lambda_x^{(N-2)} & -\lambda_x^{(N-2)} - \rho_r & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{N-1} \\ p_a \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{8.3}$$

The failure intensities $\lambda_x^{(1)}, \lambda_x^{(2)}, \lambda_x^{(3)}, \ldots$ depend on the node failure rate λ_n, the error correlation factor G, and the declustering strategy used. If only the replica fanout Z_r is known about the declustering strategy, we can compute only $\lambda_x^{(1)}$ exactly. If more than one node has failed there is a probability that the failed nodes store replicas of fragments which also have replicas on a common node, thus:

$$\begin{aligned} & & \lambda_x^{(1)} &= Z_r G \lambda_n \\ Z_r G \lambda_n &\leq & \lambda_x^{(2)} &\leq 2Z_r G \lambda_n \\ Z_r G \lambda_n &\leq & \lambda_x^{(3)} &\leq 3Z_r G \lambda_n \\ & & \vdots & \end{aligned} \tag{8.4}$$

The size and complexity of Equation 8.3 makes it difficult to understand the overall properties of the system. If we assume that $N\lambda_n \ll \rho_a$ and $NG\lambda_n \ll \rho_r$ it is reasonable to believe that the system in Figure 8.1 will spend most of its time in state S$_0$. The probability of being in any of the other states S$_i, i > 0$, will rapidly decrease with increasing i. We can therefore simplify the system by assuming that i never exceeds a number e. If a failure occurs while in state S$_e$ we can assume a system failure and go into state S$_a$. This is a conservative approach since the system goes to a state where the system is unavailable instead of one where it is available. Figure 8.2 shows the transition diagram where a maximum of three failed nodes is allowed, i.e. $e = 3$. We have also substituted the maximum values given by

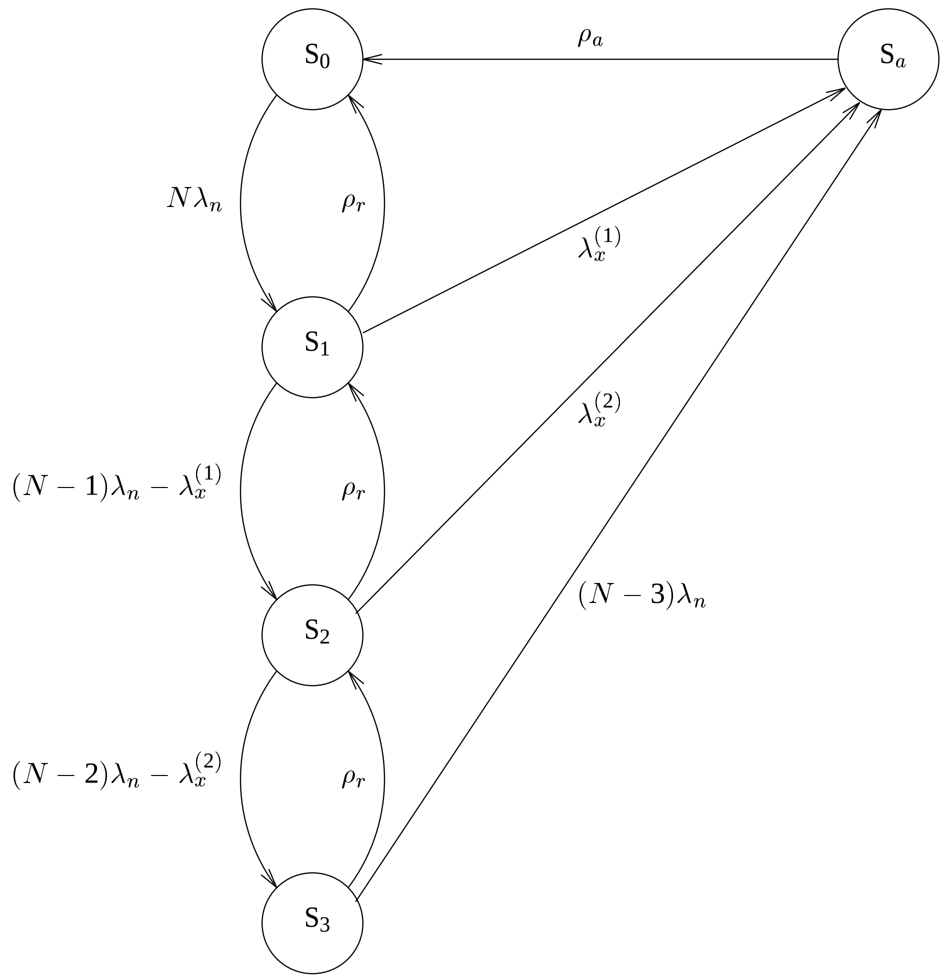


Figure 8.2: Failure state transition diagram assuming maximum three concurrent node failures. The intensity of transitions from state S_3 to state S_a is given by the aggregate failure intensity of the remaining $(N-1)$ nodes.

Equations 8.4 for $\lambda_x^{(i)}$. Equation 8.5 can be used to compute the probability vector for the states given by Figure 8.1.

$$\begin{bmatrix} -N\lambda_n & \rho_r & 0 & 0 & \rho_a \\ N\lambda_n & -(N-1)\lambda_n - \rho_r & \rho_r & 0 & 0 \\ 0 & (N-1-GZ_r)\lambda_n & -(N-2)\lambda_n-\rho_r & \rho_r & 0 \\ 0 & 0 & (N-2-2GZ_r)\lambda_n & -(N-3)\lambda_n-\rho_r & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_a \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{8.5}$$

The general solution for the state probability vector can be found through substitution and the symbolic equations for each probability p are listed in Appendix B on page 159. The form of these equations is a bit too complex to work with and understand and we need to find approximations that give answers good enough.

Assuming that $NG\lambda_n \ll \rho_r$ and $N\lambda_n < \rho_a$ most of the factors in the general solution of Equation 8.5 can be eliminated. Appendix B lists the approximate equations for the probabilities p_i.

When is the system unavailable?

The system can under no circumstances provide full availability when the system is in state S_a, since all replicas of some data have been lost. If the system does not provide on-line service during self repair the system is available only when in state S_0. This repair strategy will be called *off-line self repair*. The opposite strategy which is called *on-line self repair* provides service during repair and results in only state S_a being an unavailable state.

Unavailability is therefore the probability of any time being in a state when the system is unavailable. For off-line self repair the unavailability becomes after the approximate value for $1 - p_0$ found in Appendix B has been inserted:

$$U_{NN,\text{ off-line}} = p_a + \sum_{i=1}^{N-1} p_i = 1 - p_0 = \frac{\lambda_n N}{\rho_r} \tag{8.6}$$

and using on-line self repair we get

$$U_{NN,\text{ on-line}} = p_a = \frac{\lambda_n^2 G Z_r N}{\rho_a \rho_r} \tag{8.7}$$

If all parameters are kept unchanged, then on-line self repair would always offer better availability than off-line self repair, but since on-line self repair must use some of its resources processing user transactions, less is available to self repair. Thus, the repair is slowed down, reducing the repair intensity ρ_r, and therefore increasing unavailability. This effect and other factors influencing the unavailability are described in the next section.

8.1.2 General multi site failure model

Equations 8.6 and 8.7 are valid for only two sites. The same methodology can also be used for three, four or even more sites. A state transition diagram can be developed and transformed into a system of linear equations which can be solved. Unfortunately this requires different state diagram for different number of sites. What we want is a single, general solution valid for any number of sites.

The number of states in the transition diagram soon becomes unmanageable, even with a moderate number of sites. The resulting system of equations and the general solution will also be complex. We will therefore look into an alternative way to attack the problem.

As said before, the failure intensity of a given node is λ_n. With a N node system the node failure rate for the system becomes N times higher, i.e. $N\lambda_n$. This means that the MTTF for node failures is $\frac{1}{N\lambda_n}$. Using off-line self repair the system becomes unavailable for each failure and remains so for a time $\tau_{rr} = \frac{1}{\rho_r}$ until self repair has completed.

$$U_{NN,\text{ off-line}} = \frac{\frac{1}{\rho_r}}{\frac{1}{\lambda_n N}} = \frac{\lambda_n N}{\rho_r} \tag{8.8}$$

Equation 8.8 does neither account for more than one failure at the same time, nor the risk of loosing all replicas of data. But, comparing this equation with Equation 8.6 we see that they are identical. This confirms that the simplifications done are valid for two sites. It is interesting to see that number of sites does not directly influence the unavailability. But, since ρ_r is a function of S it therefore exists an indirect influence.

For on-line self repair this is a bit more complicated. Figure 8.3 shows a system with three sites and a total of 15 nodes with three fragment replicas. The three fragments with replicas on stored node n_3^0 have replicas on nodes n_1^1, n_2^1, n_4^1, n_0^2, n_2^2, and n_3^2. Since the replicas stored on one node are distributed over three nodes on each of the other sites the replica fanout Z_r is three. Assume that the system is available when node n_3^0 fails. The system becomes unavailable if the two other replicas of any of the three fragments are lost due to the other nodes being unavailable. The probability that both node n_1^1 and n_0^2 are unavailable is:

$$p_2 \approx 2G(\frac{\lambda_n}{\rho_r})^2$$

This equation can be found by solving the steady state probabilities of a system with two components that can fail. The system is given by the equations:

$$\begin{aligned}
2\lambda_n p_0 &= \rho_n p_1 \\
(G\lambda_n + \rho_r)p_1 &= 2\lambda_n p_0 + \rho_n p_2 \\
\rho_r p_2 &= G\lambda_n p_1 \\
p_0 + p_1 + p_2 &= 1
\end{aligned}$$

where p_i is the probability of i nodes having failed. Assuming $p_0 \approx 1$ and $G\lambda_n \ll \rho_r$ we get:

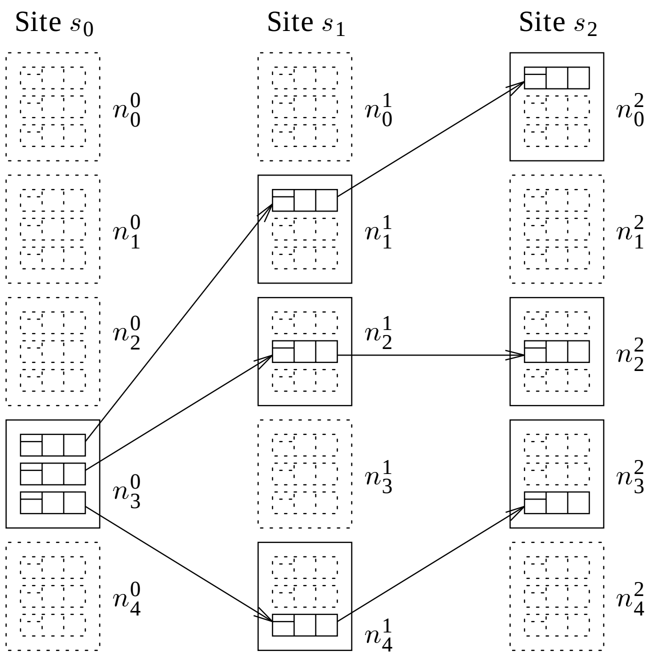


Figure 8.3: Alternative replicas to fragment replicas residing on one node. The three fragments with replicas stored on node n_3^0 on site s_0 also have replicas distributed over several nodes on site s_1 and site s_2.

$$\begin{aligned} p_1 &= \frac{2\lambda_n}{\rho_n} p_0 \approx \frac{2\lambda_n}{\rho_n} \\ p_2 &= \frac{G\lambda_n}{\rho_r} p_1 \approx 2G(\frac{\lambda_n}{\rho_n})^2 \end{aligned}$$

Thus, the probability that n_1^1 and n_0^2, or n_2^1 and n_2^2, or n_4^1 and n_3^2 are unavailable is (assuming p_2 is very small):

$$p = Z_r p_2 = 3 \cdot 2G(\frac{\lambda_n}{\rho_r})^2$$

Applying this to an arbitrary number of sites S we get:

$$p_{(S-1)} = \frac{(S-1)!}{G}(G\frac{\lambda_n}{\rho_r})^{S-1}$$

Thus, the probability of all the other replicas of any of Z_r fragments are unavailable becomes:

$$p = \frac{Z_r(S-1)!}{G}(G\frac{\lambda_n}{\rho_r})^{S-1}$$

This probability expresses the fraction of node failures that will cause a loss of all replicas of a fragment and cause system unavailability during full system recovery. The failure intensity for system failure is the product of node failure intensity, number of nodes and this probability. Thus,

$$U_{NN,\text{ on-line}} = \frac{\frac{1}{\rho_a}}{\frac{1}{G\lambda_n N p}} = \frac{Z_r N (S-1)! \lambda_n}{\rho_a}(\frac{G\lambda_n}{\rho_r})^{S-1} \qquad (8.9)$$

Substituting $S = 2$ into Equation 8.9 we see that it is equivalent to Equation 8.7. This confirms that the simplified model is correct for two sites.

8.1.3 Factors influencing unavailability

The unavailability is a function of the parameters λ_n, $\lambda_x^{(1)}$, G, N, ρ_r, and ρ_a. The most important factors influencing the parameters and therefore indirectly also the unavailability are:

HW technology: The failure rate of the nodes (λ_n) and the associated reliability of the nodes are mainly given by the technology used to implement the nodes and the communication network. Hardware defects common between different components are reflected through the G factor.

Environment: The environment the hardware operates in also affects the node failure rate λ_n. High temperature, temperature fluctuations, vibrations, mechanical shocks, unstable power supply, and high humidity all reduce the lifetime of electrical equipment and thereby increase the failure intensity. Environment problems will also cause highly correlated errors, i.e. influencing G.

Maintenance organization: The maintenance organization is responsible for replacing failed (corrective maintenance) or soon to fail (preventive maintenance) components. Preventive maintenance avoids the increase of the failure rate when the HW reaches the end of the lifetime and therefore affects λ_n. The maintenance organization is also responsible to act upon serious failures and get it running again. A loss of all replicas of one or more fragments is such a failure. The time it takes from the system becomes unavailable until it has recovered gives ρ_a and is a function of the organizations ability to handle such problems.

Problem size: The transaction load and database volume will decide the size of a scalable system. For the type of systems described here the number of nodes in the system N is the dimensioning factor.

Declustering strategy: The declustering strategy has several characteristics which influence the unavailability. The various fanouts Z_r, Z_t, Z_s, and Z_f affects probability of double failure and time it takes to do self repair. The declustering strategy also decides the redundancy and sparing strategy. More nodes are necessary to handle an increased redundancy, and the sparing strategy is important for the self repair times. The result is that ρ_r and N depend on the declustering strategy.

Self repair strategy: Off-line self repair minimizes repair times compared to on-line self repair and therefore influences ρ_r. A prerequisite for on-line self repair is extra capacity available for doing the repair. This means that more nodes than strictly necessary must be provided for this strategy to be deployed. Off-line self repair does not need this redundant capacity resulting in a lower number of nodes N.

Read/write operation mix Read operations only need to access a single tuple replica. A write operation (insert, delete, or update) must access all replicas of the tuple. A write operation on a tuple located on a disk page not cached in primary memory, must first read the page from disk, update the page, and write the updated page back to disk. This is twice as many disk accesses as a simple read tuple operation. If the fraction of write operations increases, the load increases and therefore the need for more nodes N.

If on-line self repair is provided, hot stand-by replicas must take over as primaries until the lost replicas are reconstructed. Since hot stand-bys only are accessed by write operations, read operations will contribute to an additional load on the nodes having hot stand-by replicas. This additional load will both contribute to an increased need for processing capacity on each node and/or reduce the capacity available for self repair. A higher fraction of read operation will therefore either lead to an increase of the number of nodes N or an increase of self repair time ($\frac{1}{\rho_r}$).

Given a required transaction load and database volume, the choice of parameters giving an optimal unavailability is not trivial. Especially the relationship between ρ_r, N, $\lambda_x^{(1)}$ and the declustering strategy is complex. An analysis of the system is therefore required.

8.1.4 Comparative unavailability study

Given a multi site configuration with known node failure rate λ_n, catastrophe recovery time $\frac{1}{\rho_a}$, transaction load, and read / write mix, we want to determine what are the optimal choices for self repair strategy, sparing strategy, declustering strategy, and number of nodes N. The repair strategy is either off-line self repair or on-line self repair. Sparing strategy is either dedicated spare or distributed spare. The common characteristics for the declustering strategies are the fanouts Z_r, Z_t, Z_f, and Z_s, which range from 1 and upwards in integer steps. Number of nodes N is downward limited by the minimal setup able to mask a site failure. Let the number of replicas be identical to the number of sites ($S = R$).

Node load and system size

Let L denote the transaction processing load on a node. This load is an abstraction composed of CPU and disk load, where the capacity of a node is limited by whatever is the weakest resource. The actual capacity measured in transactions per second or similar is neither of interest here, nor necessary for finding the optimal strategy and parameters. Instead we will set the capacity of a node to 1.0 and let the load vary between 0.0 (unused) and 1.0 (saturated).

A node will have various tasks that must be served concurrently, and the overall load L is the aggregate of the load contributed by each task. There are four major types of tasks:

L_{normal}: Load caused by normal transaction processing. Performed on all nodes storing fragment replicas.

$L_{masking}$: Transaction processing load caused by masking a node failure. Performed on nodes storing hot stand-bys forced to take over as primaries.

$L_{copyout}$: Load caused by copying data to another node during self repair. Performed on nodes storing replicas of lost fragment replicas.

L_{copyin}: Load caused by receiving and reconstructing fragment replicas. Performed on the new home locations of the fragment replicas.

Which tasks that are active at any time and the load of each of these tasks vary with the state of the system and the repair and sparing strategies. Table 8.1 shows how the overall load vary with the role of the system. A *sender node* is a node storing a fragment replica that is reconstructed and is sending it to one or more *receiver nodes* which have spare capacity to store the new data and handle the extra load. *Other nodes* are the nodes not involved in the self repair process, and which are neither sender nor receiver nodes. By forcing an ongoing self repair to be completed before a new one is started, it is ensured that it is impossible for a node to be both a sender and a receiver at the same time.

The capacity of a node must be large enough to handle normal transaction processing load L_{normal} and masking load caused by a specified number of concurrent node failures $L_{masking}$. In case of on-line self repair the unused capacity can be used for self repair (L_{copyin} or $L_{copyout}$). The time self repair takes, τ_{rr}, is a function of:

- the number of sender nodes,

	Off-line self repair		On-line self repair	
	Dedicated spare	Distributed spare	Dedicated spare	Distributed spare
Sender nodes	$L_{copyout}$	$L_{copyout}$	L_{normal} $+L_{masking}$ $+L_{copyout}$	L_{normal} $+L_{masking}$ $+L_{copyout}$
Receiver nodes	L_{copyin}	L_{copyin}	L_{copyin}	L_{normal} $+L_{copyin}$
Other nodes	0	0	L_{normal}	L_{normal}

Table 8.1: Node load profile depending on sparing and repair strategy.

- the number of receiver nodes,
- the unused capacity on the sender nodes,
- the unused capacity on the receiver nodes,
- the work necessary to read from disk and send each tuple,
- the work necessary to receive and write onto disk each tuple, and
- the number of tuples to replicate (same on sender and receiver side).

It is obvious that the sender side can not send tuples faster than the receivers can receive them (and opposite), and the time will therefore depend on the slowest of them.

The operations performed on a node are classified as *sequential read* (sr), *random read* (rr), *sequential write* (sw), and *random write* (rw) operations. Tuple insert, update, and delete are all write operations, while tuple retrievals are read operations. Let W_{sr}, W_{rr}, W_{sw}, and W_{rw} denote the work the respective operations require as a fraction of the capacity of a node over a time interval (i.e. $1/W_{rr}$ is the service rate for random read operations). Also let ℓ_{sr}, ℓ_{sw}, and ℓ_{rw}, denote the work of the three operations relative to random read. Thus:

$$\begin{aligned} W_{rw} &= \ell_{rw} W_{rr} \\ W_{sr} &= \ell_{sr} W_{rr} \\ W_{sw} &= \ell_{sw} W_{rr} \end{aligned}$$

Read and write operations on a given tuple arrive at given rates α_r and α_w respectively. $\alpha = \alpha_r + \alpha_w$ is the total tuple operation arrival rate. β_r and β_w is the fraction of all operations arriving that are read and write operations respectively, i.e.:

$$\beta_r = \frac{\alpha_r}{\alpha_r + \alpha_w} \qquad \text{and} \qquad \beta_w = \frac{\alpha_w}{\alpha_r + \alpha_w}$$

thus:

$$\alpha_r = \beta_r \alpha \qquad \text{and} \qquad \alpha_w = \beta_w \alpha = (1 - \beta_r)\alpha$$

Self repair times

We will use off-line self repair as a reference when comparing the various strategies. Let τ_{rr0} be the time it takes to off-line copy all tuples from one node to another, i.e. the time off-line self repair takes when $Z_f = Z_r = 1$ and $S = 2$. During copying the tuples are read sequentially on the sender side and written sequentially on the receiver side. It is reasonable to assume that sequential read and write can be done at the same speed ($\ell_{sr} = \ell_{sw}$). This assumption is safe to make if this process is disk bound since disks read and write at the same speed. Therefore, the sender can send the data to the receiver just as fast as the receiver can receive them and that both will be fully saturated.

Increasing Z_r and Z_f can reduce the time it takes to copy the tuples through parallelism. The tuples stored on a failed node is also replicated over Z_r nodes on each of the other $(R-1)$ sites. These nodes can read the lost data in parallel and send them to the Z_f receiver nodes which in parallel will receive them and write the tuples to disk. The parallelism at the sender and receiver side is therefore respectively $Z_r(R-1)$ and Z_f. For moderate parallelism we will have near linear speedup giving an self repair time equal to:

$$\tau_{rr} = \frac{1}{\min(Z_r(R-1), Z_f)} \tau_{rr0} \tag{8.10}$$

L_{normal} is the sum of the load caused by reading and writing primary tuples and only writing hot stand-by tuples. t_p is the number of primary tuples on each site and t_{hs} is the number of hot stand-by tuples. The total read operation arrival rate on a node is the product of arrival rate per tuple α_r and the number of tuples t_p on the node. The read load is computed by multiplying the aggregated arrival rate with the work required for each tuple read W_{rr}. The same method is used to compute the write load and gives the following equation for L_{normal}.

$$L_{normal} = t_p \alpha_r W_{rr} + (t_p + t_{hs}) \alpha_w W_{rw} \tag{8.11}$$

$L_{masking}$ is the increased load caused by read operations on tuples located in fragment replicas modified from hot stand-by to primary fragment replicas while masking a node failure. The read work originally performed by the failed node will be evenly distributed over Z_t nodes. E.g. $Z_t = (R-1)Z_r$ if the nodes sharing fragments with the failed node distribute the load even among them. The primary fragment replicas located on the failed node will each have $R-1$ hot stand-by replicas distributed over Z_r nodes on each or the $R-1$ other sites. Equation 8.12 shows the masking load, $L_{masking}$, computed as the read part of L_{normal} divided by Z_t.

$$L_{masking} = \frac{t_p}{Z_t} \alpha_r W_{rr} \tag{8.12}$$

During reproduction of fragment replicas lost due to a node failure, the $t_p + t_{hs}$ tuples originally stored on the failed node will be read in parallel from the other $(R-1)Z_r$ nodes storing replicas of the same fragments. If copying of these $\frac{t_p+t_{hs}}{(R-1)Z_r}$ tuples takes τ_{rr} time units, $\frac{t_p+t_{hs}}{(R-1)Z_r\tau_{rr}}$ will be copied per time unit. Multiplying this number with the work required for sequentially reading a tuple W_{sr} we get the load $L_{copyout}$:

$$L_{copyout} = \frac{t_p + t_{hs}}{(R-1)Z_r} \frac{1}{\tau_{rr}} W_{sr} \tag{8.13}$$

The subfragmentation fanout Z_f gives the number of nodes receiving the tuples during self repair. Using the same argumentation as above the equation for L_{copyin} becomes:

$$L_{copyin} = \frac{t_p + t_{hs}}{Z_f} \frac{1}{\tau_{rr}} W_{sw} \tag{8.14}$$

Let M denote the number of nodes excluding spares, i.e. $M = SN'_s$. In the initial state all nodes will have $t_p = \frac{T}{M}$ primary tuples (total number of tuples divided by the number of nodes). Using a dedicated spare or a demand replacement scheme this holds also when node failures have been repaired. For distributed spare the tuples originally distributed over $M = N$ nodes will in average be distributed over $M - \hat{e}$ nodes, where $\hat{e}$ is the average number of failed nodes not replaced. So in *average* each node will have the responsibility for $t_p = \frac{T}{M-\hat{e}}$ primary tuples. The number of unreplaced nodes will for all practical purposes be much lower than the total number of nodes in the system, i.e. $M \gg \hat{e}$, and we can use $t_p = \frac{T}{M}$ as an approximation.

Each primary tuple also has $R-1$ hot stand-by replicas. These $(R-1)t_p$ tuples will be uniformly distributed over the M nodes giving $t_{hs} = (R-1)t_p$.

Substituting these results into Equations 8.11, 8.12, 8.13, and 8.14 together with the alternative forms for α_r, α_w, W_{rw}, W_{sr}, and W_{sw} we get:

$$\begin{aligned}
L_{normal} &= \frac{T}{M} W_{rr} \alpha(\beta_r + R(1-\beta_r)\ell_{rw}) && (8.15)\\
L_{masking} &= \frac{T}{M} W_{rr} \alpha \frac{1}{Z_t} \beta_r && (8.16)\\
L_{copyout} &= \frac{T}{M} W_{rr} \frac{R}{(R-1)Z_r} \frac{1}{\tau_{rr}} \ell_{sr} && (8.17)\\
L_{copyin} &= \frac{T}{M} W_{rr} \frac{R}{Z_f} \frac{1}{\tau_{rr}} \ell_{sw} && (8.18)
\end{aligned}$$

It can be shown that these equations also hold for symmetric tuple access (read any, update all) instead of a primary / hot stand-by strategy. The rest of this evaluation is therefore valid for both strategies.

Let $N = N_0$ be the minimum number of nodes necessary to handle the peek work load without any node failures and only a single replica ($R = 1$) of data. Since this is the minimum number of nodes necessary to provide service, the nodes will be saturated and the load equal to 1, i.e. $L_{normal} = 1$.

From the definition for τ_{rr0} earlier in this chapter we know that $L_{copyin} = L_{copyout} = 1$ when $\tau_{rr} = \tau_{rr0}$, $Z_r = Z_f = 1$, $M = N = N_0$ and $R = 2$.

To minimize the time used by self repair the capacity should be maximum utilized, i.e. load should be as close to 1 as possible. Either the sender nodes or the receiver nodes will be the bottleneck during this repair. By computing the minimum repair times for both the sender and receiver nodes the overall repair time will be the maximum of the two.

As Table 8.1 on page 135 showed, the load is composed of normal, masking, copyin and copyout work for various combinations of repair and sparing strategies. For each of those cases the equations found above can be inserted and solved with respect to τ_{rr}. There are five distinct cases to solve and Equations 8.19 through 8.23 list the solutions. $\tau_{rr-x,y,z}$ stands for the minimum time where x is the role (s=sender, r=receiver), y is the repair strategy (b=on-line, o=off-line) and z is the sparing strategy (d=distributed spare, s=dedicated spare). A diamond ($\diamond$) means that equation is valid for any options in that position.

$$\tau_{rr-s,b,\diamond} = \frac{R}{Z_r 2(R-1)(\frac{M}{N_0} - R + (R - 1 - \frac{1}{Z_t})\frac{\beta_r}{\beta_r + (1-\beta_r)\ell_{rw}})}\tau_{rr0} \tag{8.19}$$

$$\tau_{rr-s,o,\diamond} = \frac{1}{Z_r(R-1)\frac{M}{N_0}}\tau_{rr0} \tag{8.20}$$

$$\tau_{rr-r,b,d} = \frac{R}{Z_f 2(\frac{M}{N_0} - R + (R-1)\frac{\beta_r}{\beta_r + (1-\beta_r)\ell_{rw}})}\tau_{rr0} \tag{8.21}$$

$$\tau_{rr-r,o,d} = \frac{1}{Z_f\frac{M}{N_0}}\tau_{rr0} \tag{8.22}$$

$$\tau_{rr-r,\diamond,s} = \frac{1}{\frac{M}{N_0}}\tau_{rr0} \tag{8.23}$$

Note that all these equations contain the ratio $\frac{M}{N_0}$ which expresses the degree of redundancy of the active nodes in system (excluding spares). With $N = M = N_0$ the ratio is 1, meaning that there is no redundancy in the system. If the ratio is equal to 2 then the system has twice as much processing capacity as necessary to serve one replica of the database. Adding spares the ratio $\frac{N}{N_0}$ expresses the total over-capacity of the system. This ratio is directly proportional to the hardware cost of the system.

The self repair times can be considered to be the MTTR for fragment replica loss. This means that the repair intensity can be written $\rho_r = \frac{1}{\tau_{rr}}$. Inserting this into Equations 8.8 and 8.9 for unavailability when using off-line and on-line self repair we get respectively after setting $S = R$:

$$U_{NN,\text{ off-line}} = \frac{\lambda_n M}{\rho_r} = \lambda_n M \max(\tau_{rr-s,o,\diamond}, \tau_{rr-r,o,\diamond}) \tag{8.24}$$

and

$$U_{NN,\text{ on-line}} = \frac{Z_r N(R-1)!\lambda_n}{\rho_a}(\frac{G\lambda_n}{\rho_r})^{R-1} = \frac{Z_r N(R-1)!\lambda_n}{\rho_a}(G\lambda_n \max(\tau_{rr-s,b,\diamond}, \tau_{r,b,\diamond}))^{R-1} \tag{8.25}$$

Among the parameters in the equations for unavailability τ_{rr0}, N_0, β_r, ℓ_{rw}, G, λ_n, and ρ_a are functions of application work load, node hardware design, and environment characteristics. Let γ be a short form for the expression:

$$\gamma = \frac{\beta_r}{\beta_r + (1-\beta_r)\ell_{rw}}$$

This value depends only on application behavior and details in the DBMS' buffer policy. This value will range from 0 to 1.

The other parameters R, N, M, Z_r, Z_f, and Z_t, the repair strategy and the sparing strategy may we as database system designers choose with the goal to achieve as good availability figures as possible.

Fanouts and replica strategy

A general observation with respect to optimal value of the fanouts Z_r, Z_f, and Z_t is that increasing their value either reduces the unavailability or has no effect at all. The only exception is the case when the receiver side is the saturated side and on-line self repair is used.

It should be noted that for the distributed spare strategy the refragmentation fanout Z_f affects the receiver side only while Z_r and Z_t affects the sender side. For the case of dedicated spare, Z_f is by definition equal to 1, and therefore the receiving speed is not a function of any fanout. The balance between sending and reception speed can be found by solving $\tau_{r,y,z} = \tau_{s,y,z}$ (if any solution exists). Doing this we find that sender and receiver balance when:

	Off-line self repair	On-line self repair
Distributed spare	$Z_f = Z_r(R-1)$	$Z_f = Z_r(R-1)(1 - \frac{\gamma}{Z_t(\frac{M}{N_0}-1-(1-\gamma)(R-1))})$
Dedicated spare	$Z_r = \frac{1}{R-1} \leq 1$	$Z_r = \frac{\frac{M}{N_0}R}{2(R-1)(\frac{M}{N_0}-R+(R-1-\frac{1}{Z_t})\gamma)}$

For a given declustering strategy some of the fanouts can take a single value only, while others are allowed to vary. Table 8.2 shows possible fanout values for the earlier described declustering strategies. As the table shows few of the declustering strategies give the designer any option for adjusting the fanouts for a best possible unavailability.

Although it is sufficient to only study the possible combinations of the fanout values and find the optimal method, it is also of interest to evaluate the interaction between the fanouts in the general case to find desirable combination of fanout values. Such an analysis can

Strategy	Z_r	Z_t	Z_f	dedicated spare
Mirrored	1	$R-1$	1	yes
HypRa	1	$R-1$	N_s-1	no
Interleaved	$S-1$	$S-1$	1	yes
Chained	2	RN_s-1	1	yes
Q-rot	$\leq N_s$	$Z_r(R-1)$	$< N_s$	yes

Table 8.2: Fanout values and sparing strategies for various declustering strategies.

then lead to the development of a new declustering strategy improving upon the existing ones.

For the existing declustering strategies the takeover fanout Z_t is either equal to $M-1$ (all active nodes except the failed one) or the same as the product of replica fanout and the number of remaining replicas, i.e. $Z_r \cdot (R-1)$ (the nodes sharing fragments with the failed one). It is possible to envision declustering strategies using another Z_t but it will not be discussed here. Section 8.1.5 will briefly discuss the takeover fanout and an alternative takeover strategy.

Three alternatives for sparing have been studied, two based on distributed spare, and one using dedicated spare. The two using distributed spare have a subfragmentation fanout of respectively a fixed value 4 and a value equal to the replica fanout.

Figures 8.4 and 8.5 show the unavailability for various fanout values and capacity levels $\frac{N}{N_0}$ in systems with two replicas. Note that even though the curves in the figures are continuous, they are only valid for integer fanout values.

Common for both figures is that they start out at low replica fanout with coinciding curves for distributed spare independent of Z_f value. For the same $\frac{N}{N_0}$ ratio the curve for dedicated spare is a bit above since some of the nodes are reserved to be spares. This condition holds as long as the sender node is the bottleneck. The breakpoints show the transition when the receiver takes over as bottleneck. When Z_r increases, the time it takes to copy out the data decreases. For the dedicated spare strategy and the case of a fixed Z_f the receiver capacity is independent of the Z_r value and therefore when the receiver nodes become the limitation the self repair time is constant. If we let Z_f scale with Z_r, the receiver nodes never become the bottleneck and therefore lack the break on the curve.

In general we can say that the unavailability is reduced or constant for increasing Z_r's as long as the sender is the limiting factor. If the receiver is the limiting factor the unavailability is either increasing or constant for increasing Z_r's.

Figures 8.4 and 8.5 also show that distributed sparing with $Z_f = Z_r$ is always either equal to or better than the two other sparing strategies shown in ith figure. The relative rank between dedicated spare and the distributed spare strategy with fixed Z_f depends on the $\frac{N}{N_0}$ ratio. For a low $\frac{N}{N_0}$ ratio distributed spare is the preferred strategy. Since the unavailability

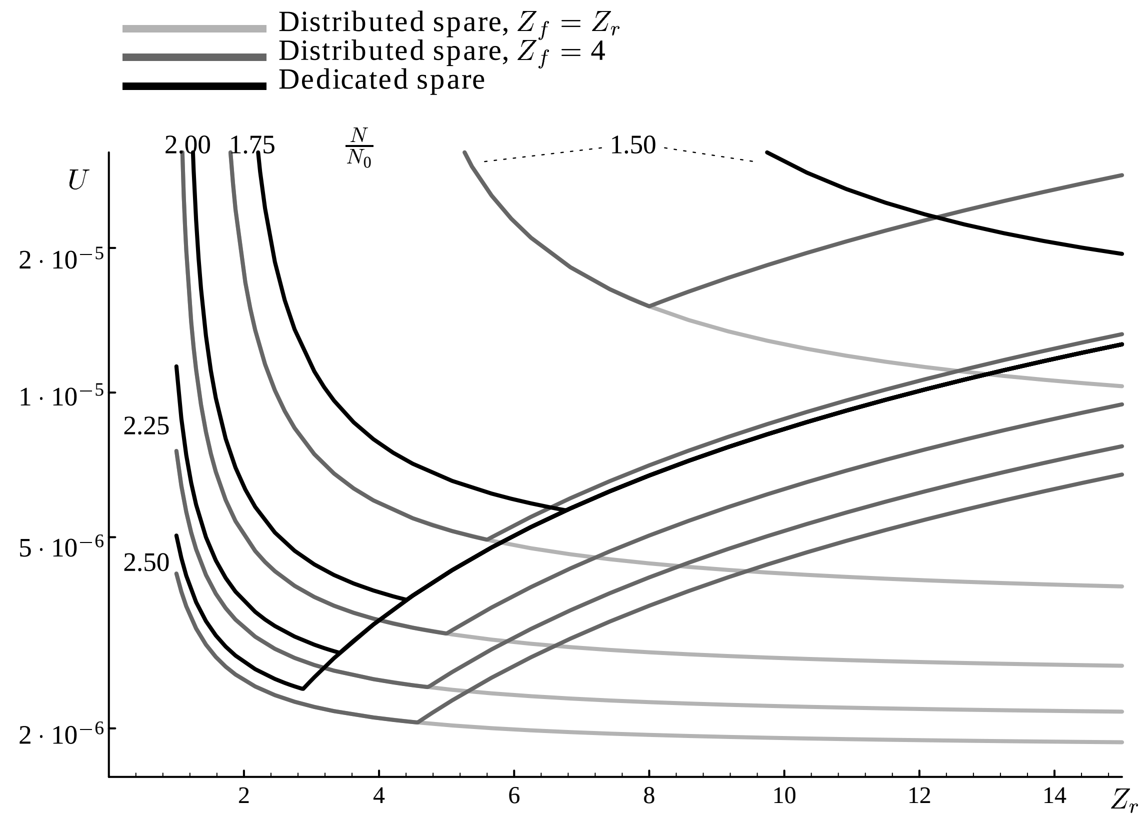


Figure 8.4: Unavailability as a function of replica fanout when $Z_t = Z_r(R-1)$ for various capacity levels and target strategies. $\ell_{rw} = 1$, $\beta_r = \frac{2}{3}$, $N_0 = 100$, $R = 2$, $\tau_{rr0} = \frac{1}{2}$hours, $G = 1$, $1/\lambda_n = 100,000$h, $1/\rho_a = 7 \cdot 24$hours = one week. For dedicated spare 4% of the nodes are reserved as spare nodes (i.e. $M = 0.96N$).

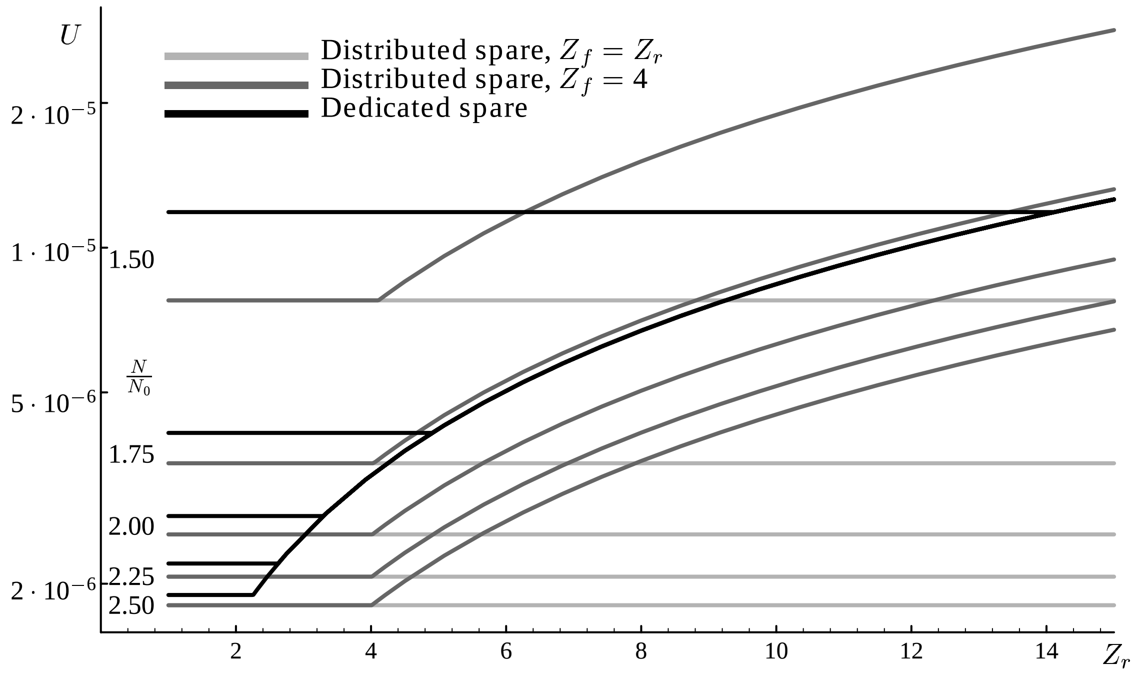


Figure 8.5: Unavailability as a function of replica fanout when $Z_t = N - 1$ for various capacity levels and target strategies. $\ell_{rw} = 1$, $\beta_r = \frac{2}{3}$, $N_0 = 100$, $R = 2$, $\tau_{rr0} = \frac{1}{2}$hours, $G = 1$, $1/\lambda_n = 100,000$h, $1/\rho_a = 7 \cdot 24$hours = one week. For dedicated spare 4% of the nodes are reserved as spare nodes (i.e. $M = 0.96N$).

decreases when the ratio increases using a fixed Z_f and is constant for dedicated sparing, there is a given ratio value where the rank positions are swapped. The limit value is given by Equation 8.26.

$$\frac{N}{N_0} = \frac{\beta_r + (1-\beta_r)\ell_{rw}R}{(\beta_r + (1-\beta_r)\ell_{rw})(1 - \frac{R}{2Z_f}\frac{M}{N})} \tag{8.26}$$

Using the equation and inserting the numbers used to plot Figure 8.4 and Figure 8.5 we get that the dedicated spare is better that distributed spare with $Z_f = 4$ when $\frac{N}{N_0}$ is less than $\frac{16}{9} \approx 1.76$.

Another common observation that can be made based on both the figures is that increasing the over-capacity ratio $\frac{N}{N_0}$ (without increasing the degree of replication) also improves the availability. This is a non-obvious result since increasing the number of nodes also increases the absolute number of node failures per time unit. This is discussed more in depth later in this section.

By comparing Figure 8.4 and 8.5 we see that as long as the receiver is the bottleneck the takeover fanout has no effect and the unavailability is identical for the two figures. On the other hand, when the sender is the limiting factor, the largest takeover fanout value possible ($Z_t = N - 1$) gives the best unavailability. We even see that varying the replication fanout value has no visible effect in our case and increasing it gives no reward. This means that it can be held on a low fixed value (e.g. 2).

Since a combination of high Z_r and Z_f results in a highly parallel copying of data from one site to another the intersite communication capacity might become saturated. The communication capacity should therefore be considered when fanout values are chosen so this is avoided. Either the communication system has to be dimensioned to handle the load or the fanout values must be adjusted. In general we can say that a more parallel replica reproduction requires a higher communication bandwidth. Keeping the fanout values low also keeps the parallelism low and therefore reduces the bandwidth requirement for the communication.

The conclusion from the discussion on the fanout values is the maximum takeover fanout gives the best availability. In this case a low replica fanout gives the best result independent of sparing strategy. Unfortunately using this takeover strategy requires a massive takeover after a node failure and can in itself result in a significant unavailability which is not accounted for in the analysis.

Using a minimal takeover fanout (i.e. $Z_t = Z_r$) involves a takeover for only the fragments with a primary replica on the failed node. For this option a high replica fanout combined with a distributed spare strategy with a redistribution fanout equal to the replica fanout gives the best availability. For a dedicated spare strategy there exists an optimal value for the replica fanout.

$\frac{N}{N_0}$ ratio

The $\frac{N}{N_0}$ ratio expresses the degree of hardware redundancy over a “conventional” single replica system. In other words, this is the hardware cost of the system. By studying the equations for unavailability it is easy too see that for off-line self repair the ratio has no

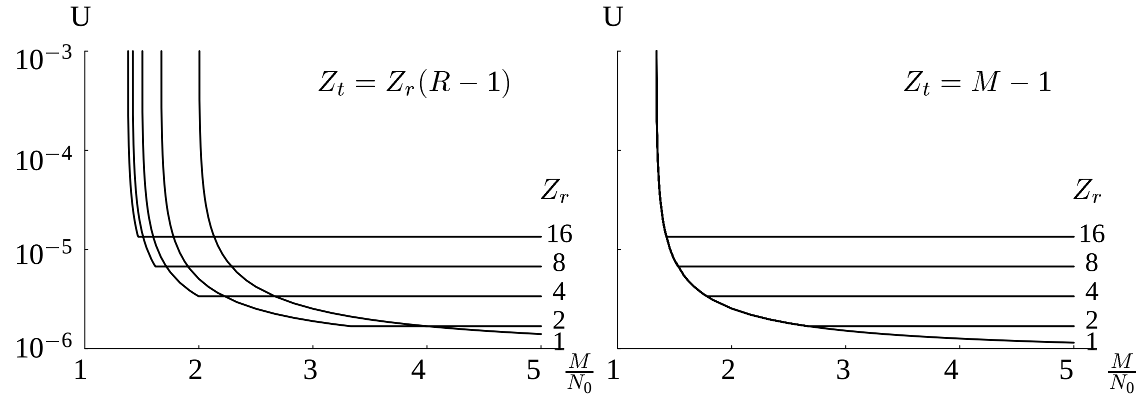


Figure 8.6: Unavailability of a system using on-line self repair and dedicated spares as a function of the $\frac{M}{N_0}$ ratio for various fanout values. $\ell_{rw} = 1, \beta_r = \frac{2}{3}, N_0 = 100, \tau_{rr0} = \frac{1}{2}h, G = 1, 1/\lambda_n = 100,000$h, $1/\rho_a = 7 \cdot 24$h $=$ one week.

effect at all! This means that as long as the system is able to handle the work load the degree of hardware redundance can be kept on a minimum without affecting the unavailability.

For on-line self repair this is a bit more complicated. In case of three or more replicas ($R \geq 3$) a higher ratio always means less unavailability. For two replicas this holds for the distributed spare strategy and for the dedicated spare strategy with the sender as the bottleneck. If the receiver is the bottleneck, the ratio has no effect at all. Increasing the ratio will in this case reduce the Z_r value giving the optimal unavailability (see previous discussion on optimal fanout values).

In general we can conclude that increasing the $\frac{M}{N_0}$ ratio will either reduce the unavailability or have no effect at all. This can be seen in the Figures 8.4, 8.5, and 8.6. Unfortunately, increasing the ratio will also proportionally increase the system cost.

In this work we have not tried to optimize the total cost with respect to hardware and unavailability cost (direct and indirect). The cost function related to unavailability with respect to frequency and duration is complicated and depends on the application.

Note that the $\frac{M}{N_0}$ ratio has a lower limit which ensures that the work load is served. Systems using off-line self repair can (in theory) be configured exactly on this minimum limit. On-line self repair systems will have a higher system cost since they also must include capacity to handle masking of a node failure and the copy process. The ratio for on-line self repair systems must be larger than a lower limit value where the unavailability with increase asymptotic approaching the limit. The limit will be for a system handling the normal work load and masking of a node failure but not having any capacity for self repair. By solving the load equations (e.g. $L_{normal} + L_{masking} \leq 1$) with respect to the ratio $\frac{M}{N_0}$ the minimum values becomes:

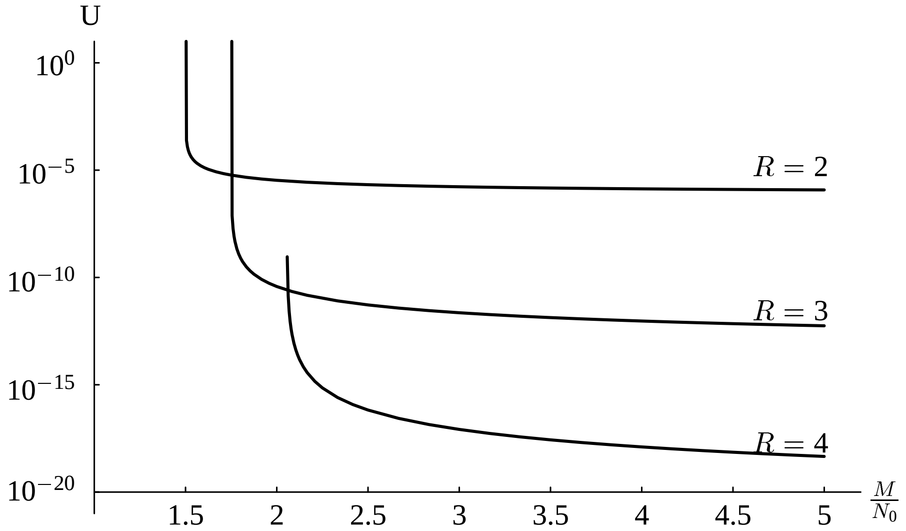


Figure 8.7: Unavailability of a system using on-line self repair as a function of the $\frac{M}{N_0}$ ratio using two, three and four replicas. $\ell_{rw} = 1$, $\beta_r = \frac{2}{3}$, $N_0 = 100$, $\tau_{rr0} = \frac{1}{2}$h, $Z_r = 4$, $Z_t = Z_r(R-1)$, $G = 1$, $1/\lambda_n = 100,000$h, $1/\rho_a = 7 \cdot 24$h $=$ one week. The sending and reception speed is assumed to balance.

Off-line self repair	$\frac{M}{N_0} \geq R - (R-1)\gamma$
On-line self repair	$\frac{M}{N_0} > R - (R - 1 - \frac{1}{Z_t})\gamma$

The asymptotic behavior of the unavailability close to the limit for on-line self repair can be seen on Figure 8.6.

Replicas

As long as the node MTTF is much larger than the self repair times and sufficient over-capacity for self repair is provided, will an increasing number of replicas always give an improved availability. This is certainly depending on the system to have the capacity to handle the increased number of replicas both with respect to processing and storage capacity. Figure 8.7 shows the unavailability for two, three, and four database replicas with respect to the $\frac{N}{N_0}$ ratio. The unavailability is equal to one (always unavailable) for $\frac{N}{N_0}$ ratio values close to the limit but drops rapidly for increasing ratio values. Note that the curves soon flatten and little improvement in unavailability is achieved after the initial fall. For the same $\frac{N}{N_0}$ values it is best to go for the highest number of replicas supporting it. E.g. the ratio $\frac{N}{N_0} = 2$ is supported by both two and three replicas, but three replica give a much better unavailability figure.

Correlated failures

The G factor expresses the higher probability of a node failure when another failure is not repaired yet. None of the repair times are influenced by this factor. Therefore the factor only is a scaling of the unavailability independent of declustering strategy. In our examples we have used a factor of 1 (i.e. no correlation), but increasing it will only increase the expected unavailability by a constant factor independent of declustering strategy.

Repair strategy

The ratio between unavailability using off-line self repair and on-line self repair can be used to decide which strategy is the best, i.e. causes least unavailability. If the ratio is greater than 1 then on-line self repair should be preferred.

$$\frac{U_{NN,\text{ off-line}}}{U_{NN,\text{ on-line}}} = \frac{\lambda_n N \tau_{rr-\diamond,o,\diamond}}{\frac{Z_r N(R-1)!\lambda_n}{\rho_a}(G\lambda_n \tau_{rr-\diamond,b,\diamond})^{R-1}} = \frac{\rho_a}{Z_r(R-1)!}\frac{\tau_{rr-\diamond,o,\diamond}}{(G\lambda_n \tau_{rr-\diamond,b,\diamond})^{R-1}} > 1$$

Reordering this a bit we get that on-line self repair should be used when:

$$\frac{\tau^{R-1}_{rr-\diamond,b,\diamond}}{\tau_{rr-\diamond,o,\diamond}} < \frac{\rho_a}{Z_r(R-1)!(G\lambda_n)^{R-1}}$$

We see that shorter system recovery times, lower node failure intensity and smaller replica fanouts favor on-line self repair. We can also assume that $1/G\lambda_n \gg \tau_{rr-\diamond,b,\diamond}$ i.e. MTTF for a node is much longer than self repair. This means that a higher number of replicas R greatly contributes to the favor of on-line self repair.

By inserting $1/\rho_a = 1$ week, $G = 1$, $1/\lambda_n = 100,000$ hours, $Z_r = 10$, and $R = 2$ we get that off-line self repair must at least be 60 times faster that on-line self repair before it will be the best choice. This will only occur for very few situations when the $\frac{N}{N_0}$ ratio is very close to its minimum. In general on-line self repair will be the optimal choice.

8.1.5 Further reduction of unavailability

There is a couple of methods which can be used to reduce the time it takes to do automatic reproduction of a lost replica that is not incorporated in the model above. Two methods that can be used when on-line self repair is used are informally discussed in this section. A formal analysis of them is left for further research.

At a sender node we know that the need to serve transaction requests steals capacity from the repair process and prolongs it. In the model above we assumed that the sender nodes had to serve the normal transaction load plus masking a fraction of the transactions that should have been performed by the failed nodes in addition to reproduction of a lost replica. Some declustering strategies actually allow both the normal transaction load and the masking load to be significantly reduced.

Moving away normal transaction load

Assume that a node n shares data with Z_r other nodes on another site. If one of the Z_r nodes fails, then the $Z_r - 1$ other nodes on the site of the failed node should remain unaffected. Instead of continuing to serve primary fragment replicas shared with these $Z_r - 1$ nodes, node n can push its own primary roles to these other nodes. While the other nodes get a higher load the node that has to send data and mask the failure is relieved from some of its normal transaction load. This idea is illustrated in Figure 8.8. The only primary fragment roles that remain are those for the fragments that has no other replicas that can take over, i.e. the fragments that have to be reproduced.

On the figure we can see how the sender nodes after a failure have the responsibility for two primary fragment replicas. The remaining nodes on the right hand side which can not be sender nodes (they do not share data with the failed node) increase their workload from four to eight primary replicas. When the self repair has completed and the fault-tolerance level has been restored, the primary and hot stand-by roles are returned to the original state.

Note that only read operations are moved away from the sender nodes. Write operations must still be performed on all fragment replicas to keep them up-to-date. This mechanism therefore has best effect on systems with a low fraction of database writes.

Gradually takeover

Another interesting technique is to open up for transaction processing on the receiver node(s) before all fragment replicas have been received. Immediately when a fragment has been completely received the receiver can take over as a primary and gradually become a normal transaction processing node. It will take over the primary role from the sender node and relieve it from a part of its load. For each fragment replica that is moved, the transaction processing load on the sender is reduced, and therefore results in increased reproduction speed. On the receiver side the increasing transaction load eats of the capacity available to receive data. This mechanism will therefore only have a positive effect when the sender is the limiting factor and there is sufficient margin at the receiver nodes to handle the gradually increasing transaction loads.

This mechanism assumes that each sender node will send several fragments to each receiver node. This is likely since a database will probably contain several relations, each partitioned into multiple fragments. If this do not hold, the sender can partition one large fragment into several small logical fragments.

Both the gradually takeover strategy and the primary role offloading for sender nodes will have best effect when the $\frac{N}{N_0}$ ratio is low and there are little capacity left to the self repair process. In this case the increase in reproduction capacity can be several times larger and thereby reducing the reproduction time to only a fraction.

Both methods described here do not fit into the analysis model developed in this chapter. The problem lies in the takeover fanout Z_t. In the model Z_t is invariant over time, which means that the load is constant as long as repair is done. This does not hold when using gradually takeover which actually results in a reduction of the load at the sender side as the repair proceeds. A similar problem applies to the primary role offloading technique.

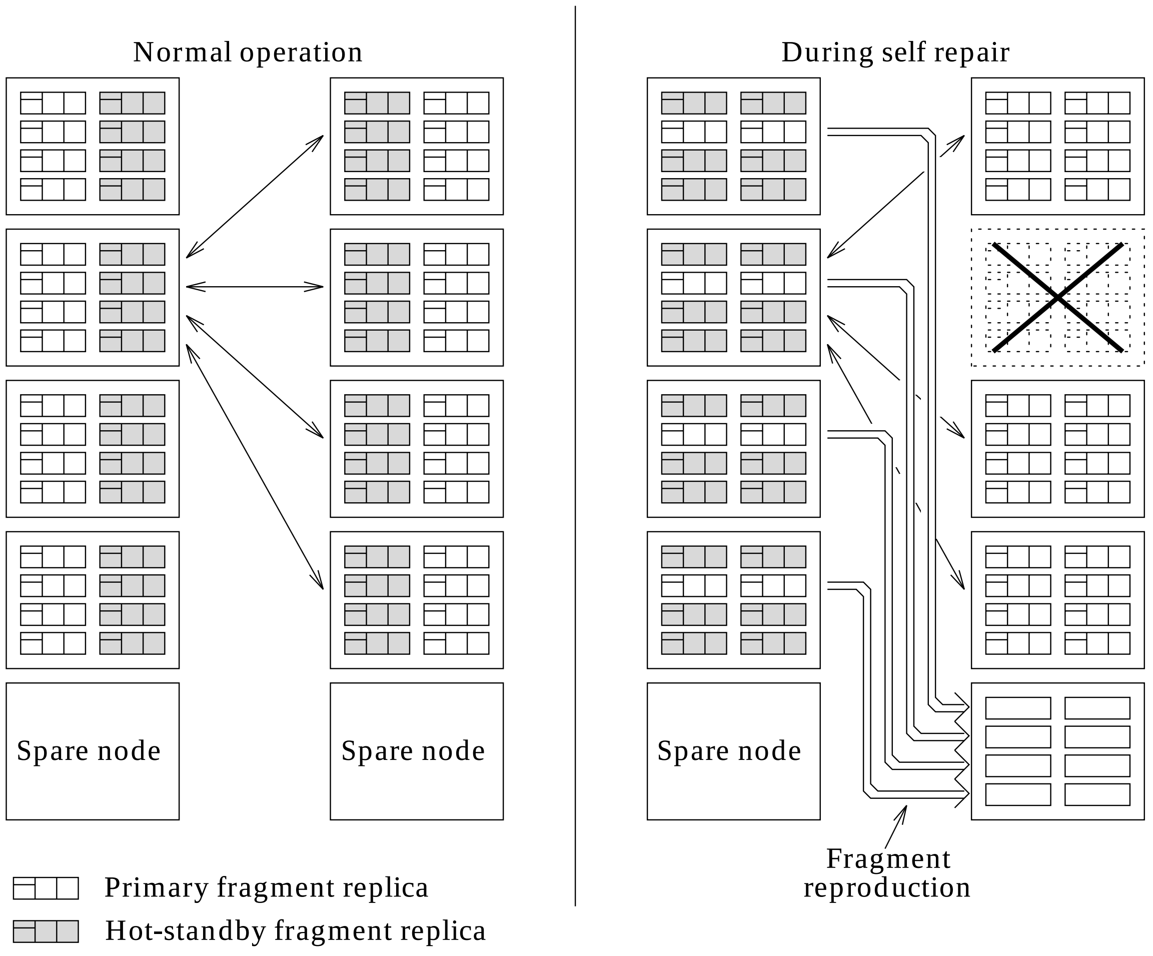


Figure 8.8: Moving away normal transaction load from sender nodes during dedicated spare replica reproduction. There are two sites, each with five nodes including one dedicated spare. Each node stores eight fragment replicas and the replica fanout Z_r is four. In the normal situation half of the fragment replicas are primaries and the other half are hot stand-bys. The wide arrows show how the replicas are reproduced on the spare node. The other arrows show the corresponding replicas of fragments located on a node on the site on the left hand side.

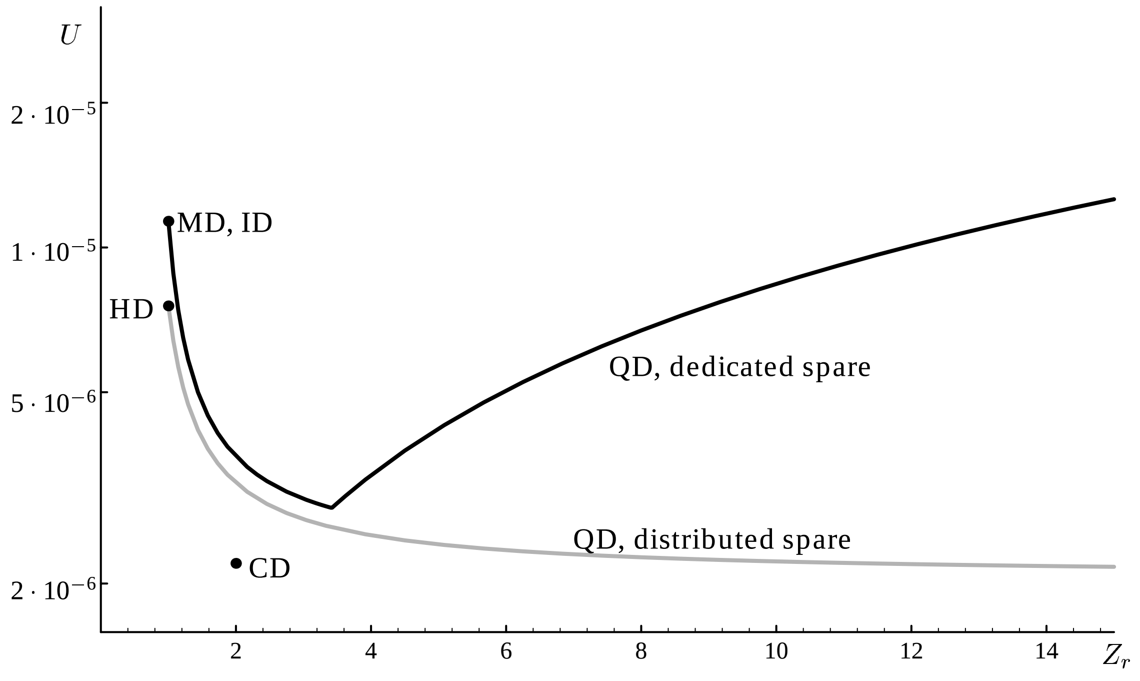


Figure 8.9: Unavailability as a function of replica fanout for the described declustering strategies. $\ell_{rw} = 1$, $\beta_r = \frac{2}{3}$, $N_0 = 100$, $\frac{N}{N_0} = 2.25$, $R = 2$, $\tau_{rr0} = \frac{1}{2}$hours, $1/\lambda_n = 100,000$h, $1/\rho_a = 7 \cdot 24$hours $=$ one week. For dedicated spare 4% of the nodes are reserved as spare nodes.

The model assumes that the normal transaction processing load is the same in the normal state as during self repair. This can be handled by allowing Z_t to be negative reflecting the reduction in transaction processing load.

8.1.6 Declustering strategies comparison

Figure 8.9 shows the position of the different declustering strategies on the unavailability plots shown in Figures 8.4 and 8.5. The plot is based on an $\frac{N}{N_0}$ ratio equal to 2.25 and two replicas of the data.

The figure shows that mirrored declustering (MD), interleaved declustering (ID), HypRa declustering (HD), and chained declustering (CD) is only represented with points in the diagram since they have no freedom with respect to replica fanout. ID is only a point since we have chosen to only consider two site systems. Q-rot (QD) is drawn both for a dedicated and a distributed spare strategy.

With respect to unavailability MD and ID is the two worst strategies headed by HD which is a bit better. CD is the best choice with its low unavailability figure. QD using distributed sparing can be configured to have an unavailability close to CD by using a large replication fanout. Dedicated spare QD is a bit worse than distributed sparing for small replication fanouts but are much worse for large fanouts.

The disadvantage with CD is the expensive takeover processing globally adjusting the primary and hot stand-by fragment partition limits. This can result in a short service interruption after each node failure which can be classified as unavailability. This can quickly consume the advantage it has over the other methods which have a much lighter takeover switch.

In addition to giving low unavailability QD can efficiently support both OLTP and OLCP operations similar to the Hybrid-Range Partitioning Strategy. It also supports more than two replicas and as Figure 8.7 showed, a three replica system is several magnitudes better than those with two replicas. CD is limited to two replicas and can therefore not compete if three sites can be afforded.

Note that the plots and some of the comparisons are based on a specific set of system parameters like system size, transaction profiles, and failure and repair rates. The results are therefore not necessarily correct for all parameter sets and should be used with care. There are many free variables in the system which makes it difficult to make firm conclusions. After saying this it should be pointed out that the numbers used are representative for a real world case and the results we have found give a good indication of system behavior.

8.2 Site failure caused double-faults

The previous section analyzed the unavailability caused by successive failures resulting in loss of all replicas of one or more fragments. This section will discuss site failures causing system unavailability either in combination with another site failure, or one or more node failures. Using two sites we have the following scenarios:

- One site is out of service. The second site fails and the system becomes unavailable.
- One site is out of service. A node on the remaining site fails. All replicas of the fragments stored on that node are now lost and a part of the database is unavailable.
- One or more node failures have occurred on a site and the system has not managed to complete self repair yet. The other site fails making the source fragment replicas unavailable. The site with the failed nodes has an incomplete replica of the database and the database is partially unavailable.

Note that a site outage not necessarily is caused by a failure resulting in a permanent loss of the data stored on the site. A network partitioning can be considered to be a temporary loss of the sites in one of the partitions. When the network is reestablished the sites being down can be re-synchronized to a consistent database by applying the log collected on the active sites. This avoids a long lasting full rebuild of the database.

Another cause for downtime not resulting in data loss is maintenance work. A site can be gracefully shut down to do hardware or software upgrade without the database content is lost on the site.

For the sites we are investigating there are two types of site downtime: *planned* and *unplanned* downtime. The planned downtime can be postponed if the other site is down or any nodes are currently being repaired.

Equations 8.27 through 8.29 give a simplified model for the unavailability of the three scenarios listed above. U_{SS} is the unavailability caused by a double site failure, U_{SN} is the unavailability for node failures during downtime for a site, and U_{NS} for site failures during node failures. λ_s is the failure intensity for unexpected site failures, λ_p is the intensity of planned site outages, and λ_n is the node failure rate. ρ_s is the site recovery intensity (both for planned and unplanned outage), ρ_a is the rate of full system restoration after loosing all replicas of data, ρ_r is the node self repair rate, and finally N_s is the number of nodes per site.

$$U_{SS} = \frac{(\lambda_s + \lambda_p)\lambda_s}{\rho_s \rho_a} \tag{8.27}$$

$$U_{SN} = \frac{(\lambda_s + \lambda_p)N_s\lambda_n}{\rho_s \rho_a} \tag{8.28}$$

$$U_{NS} = \frac{N_s\lambda_n\lambda_s}{\rho_r \rho_a} \tag{8.29}$$

Continuing with our example system assuming a two site system with 200 nodes and the following parameters:

$1/\lambda_n =$	100,000	hours
$1/\lambda_s =$	1	year
$1/\lambda_p =$	2	months
$1/\rho_s =$	1	hour
$1/\rho_a =$	7	days
$1/\rho_r =$	1/2	hour
$N_s =$	100	nodes

giving the following unavailability figures:

$$\begin{aligned} U_{SS} &= 1.6 \cdot 10^{-5} \\ U_{SN} &= 2.5 \cdot 10^{-4} \\ U_{NS} &= 9.7 \cdot 10^{-6} \end{aligned}$$

For the parameters used by the example U_{SN} is dominating the two others with more than a factor of 10. Looking on the examples presented for U_{NN} in the previous section (e.g. Figure 8.9) ranging from $1.1 \cdot 10^{-5}$ to $2.2 \cdot 10^{-6}$ shows us that the same also applies to double node failures. To improve the overall system availability Equation 8.28 should be studied and the factors influencing the input variables to the equation changed until the availability goal is reached.

Note that U_{SN} is the dominating factor only for this example. Other configurations have other parameters and perhaps have another factor that dominate the unavailability.

The equations above are only valid for system with two sites. For systems with three or more sites all combinations of node and site failures causing the loss of all replicas of one or more fragments must be considered and the unavailability computed.

In the previous section we learned that reducing the time it takes to do self repair $\tau_{rr} = 1/\rho_r$ not necessarily reduced the unavailability caused by double node failures U_{NN}. Since ρ_r is variable in the equation for U_{NS} it also influences this unavailability figure. U_{NN} and U_{NS} should therefore be seen together when the fanouts and other parameters for the declustering strategies are chosen. Also the use of N_s in the equations for U_{SS} and U_{NS} is an interaction with U_{NN}. N_s is a function of the $\frac{M}{N_0}$ ratio (i.e. $N_s = \frac{M}{S} = \frac{M}{N_0} \cdot \frac{N_0}{S}$). While a higher capacity ratio reduces U_{NN} it also increases U_{SS} and U_{NS}. In the context of the example above it is better to reduce U_{NS} than U_{NN} and therefore $\frac{M}{N_0}$ should be kept as low as possible.

Chapter 9

Conclusions and further work

9.1 Contributions

This thesis has made the following contributions:

Multi-site database cluster

A multi-site architecture for achieving very high system availability has been proposed. This architecture is based on standard "workstation class" hardware components which potentially can keep the total system cost down. (Chapter 4).

Multi-site declustering

Several multi-site declustering strategies have previously been proposed. In this work the declustering strategies have been set into a system. It has been shown how some single-site declustering strategies can be adapted to multi-site systems, and one of these (Chained Declustering) seems to provide excellent availability. (Chapter 5 and 6).

New declustering strategies

A new family of multi-site declustering strategies called *Minimal Intersecting Set Declustering* has been suggested. This has lead to the invention of a new strategy called *Q-rot Declustering* with many degrees of freedom making it suitable to a wide range of applications. It supports any number of database replicas, executes both OLTP and OLCP transactions efficiently, and provides good availability. (Chapter 7).

Declustering strategy characterization

A set of characteristics describing the properties of a declustering strategy has been proposed. Functions that can be used to compute the resulting unavailability for declustering strategies have been developed using a failure model based on these characteristics. This model has been used to compare existing declustering strategies. The model has also lead to a better understanding of the behavior resulting from changing these characteristics, which again can lead to the development of new and better declustering strategies. (Chapter 6 and 8).

Self repair

The work has been centered around the concept of a self healing database system. Through spare capacity the fault-tolerance level is restored after a node failure by recreating lost database fragment replicas. This approach can give unavailability figures several magnitudes better than a scheme awaiting manual replacement of failed hardware components. This scheme can also lead to new and more cost efficient maintenance procedures using periodic instead of immediate replacement of components. (Chapter 5).

A comparison of repair and sparing strategies has been done in connection with self repair. It has been found that on-line self repair for all practical purposes is better than off-line repair. Distributed and dedicated sparing used with self repair have also been evaluated resulting in a small advantage for distributed over dedicated sparing. (Chapter 8).

9.2 Criticism

The failure model used in the evaluation of the declustering strategies is rather coarse and many simplifications have been made. One might ask if it is too coarse and whether it is sufficient to precisely represent the system it models. The goal of the model has not been to find the precise unavailability. This is not even possible as many of the system parameters necessary as input to such a model are rather approximate. Hardware reliability figures are often only predictions and the reliability of software is notoriously difficult to even predict. Independent of the accuracy of the unavailability figures produced, the model can still be used to compare the declustering strategies. The model can also be used to study the effect of changing the various system parameters. It should also be noted that a complex model might be complicated to use and might lead to not necessarily finding important results.

The study has, like previous work in the field of declustering strategies, only looked on permanent failures. However, transient errors are much more frequent in the real world and can be a much more serious problem. It is difficult to draw strong conclusions on system availability before the effects of such errors are analyzed.

Even data with a nice, evenly distribution will have statistical skews invalidating several assumptions made in this thesis. The occurrence of real world heavily skewed data makes this an even worse problem. The absence of treatment of skewed data can make the results found seriously distorted.

9.3 Further work

This work started out with the goal to solve all aspects of highly available database system and as one could expect this soon turned out to be an unachievable goal. One after another, the planned fields of work were trimmed away ending up with a narrow subject covering multi-site declustering strategies. It is even more frustrating that there still remains lot of work in this field.

Below follows a list of topics that should be of interest for further investigations:

- What effect has the communication bandwidth and latency on the system availability? Especially the use of workstations interconnected with ATM technology could be studied.
- According to the literature most failures are transient and will disappear when the operation is retried. These failures are more than a magnitude more frequent than the hard errors. Even though they are easy to repair they can still contribute to the unavailability the short time it takes to repair or mask them. These type of failures should therefore be studied to be able to predict the unavailability caused by them. It is of little use to have excellent availability figures with respect to permanent failures if transient failure caused unavailability is dominating.
- The latency from a node fails until the hot stand-bys are ready to mask the error can cause service interruption and contribute to the unavailability. This latency and the influence the declustering strategy has on it is a candidate for further research.
- Of the four fanout parameters, takeover fanout was shown not to be able to fully handle all declustering schemes. Both moving normal transaction load away from nodes and time variable masking load during self repair showed the insufficiency of the takeover fanout. Negative fanout values can partially solve this problem. Future work should try to incorporate these aspects.
- Moving normal transaction load away from nodes participating in self repair would reduce the duration of self repair and therefore also the unavailability. Eventually the system cost could be reduced since less over-capacity is required for masking and self repair for the same unavailability level. A declustering strategy should therefore be developed utilizing this mechanism.
- There is also a statistical probability for running out of spare capacity before failed components are replaced. Replacement strategies and spare capacity planning to optimize system cost and unavailability are topics for future work.
- One assumption made in this thesis was that the system was composed of fail-fast modules. Even though most errors will behave in this manner there will always exist some that violate these assumption. Further research should therefore investigate detection mechanisms deployable in database systems reducing the frequency of these events.

Several of these paths will be pursued in an on-going project at Telenor Research. A primary task is to build a research prototype based on some of the ideas proposed in this thesis. The prototype will be used to both verify these ideas and be a testbed for experiments in future projects. After having spent so much time on this thesis it is rewarding to see that there is so much interest in the work that someone is willing it invest in further research in the field.

Appendix A

Symbols used in the availability analysis

Legend to mathematical symbols used for describing declustering strategies and for the availability analysis.

S	—	number of sites
C_s	—	clusters pr. site
N	—	number of nodes including on-line spare nodes (over all sites)
N_0	—	minimal number of nodes handling required request intensity
M	—	number of nodes excluding on-line spare nodes (over all sites)
N_s	—	nodes per site excluding on-line spare nodes
N'_s	—	on-line spare nodes per site
N_c	—	nodes per cluster
s_i	—	site number i
n_i^s	—	node number i on site s
F	—	number of fragments
F_n	—	fragments per node
R	—	number of replicas of each fragment
f_i	—	fragment number i ($0 \leq i < F$)
f_i^r	—	replica r of f_i ($0 \leq r < R$)
Z_r	—	replica fanout
Z_t	—	takeover fanout
Z_f	—	subfragmentation fanout
Z_s	—	site fanout
Δ_F	—	fragment stride ($1 \leq \Delta_F \leq N$)
Δ_R	—	range stride (fraction of key value range $0 \leq \Delta_R < 1$)
t_p	—	maximum number of primary tuples pr. node
t_{hs}	—	maximum number of hot stand-by tuples pr. node

S_i	—	system state where i nodes are out of service
S_a	—	system state where the full system is out of service
p_i	—	probability of being in state S_i
$\hat{e}$	—	average number of repaired node failures
$\tau_{fn} = 1/\lambda_n$	—	MTTF node
$\lambda_x^{(i)}$	—	failure intensity for double failures in state S_i
G	—	failure intensity scaling factor for double failures
$\tau_{fs} = 1/\lambda_s$	—	MTTF site
$\tau_{fp} = 1/\lambda_p$	—	mean time between planned site downtime
$\tau_{fa} = 1/\lambda_a$	—	MTTF system (non maskable double failure)
$\tau_{rn} = 1/\rho_n$	—	MTTR node (node replacement)
$\tau_{rs} = 1/\rho_s$	—	MTTR site (site recovery)
$\tau_{ra} = 1/\rho_a$	—	MTTR system (system recovery after double failure)
$\tau_{rr} = 1/\rho_r$	—	mean self repair time after node failure (replica repair)
τ_{rr0}	—	mean time to copy all data from one node to another
α	—	operation arrival rate pr. tuple
α_w	—	arrival rate pr. tuple for write operations
α_r	—	arrival rate pr. tuple for read operations
β_r	—	read fraction of total operation count
β_w	—	write fraction of total operation count
W_{rw}	—	work to randomly write a tuple (fraction of node capacity)
W_{rr}	—	work to randomly read a tuple
W_{sw}	—	work to sequentially write a tuple
W_{sr}	—	work to sequentially read a tuple
ℓ_{rw}	—	cost of a random write operation relative to a random read
ℓ_{sw}	—	cost of a sequential write operation relative to a random read
ℓ_{sr}	—	cost of a sequential read operation relative to a random read
γ	—	load profile (see equation 8.1.4 on page 139)
A	—	availability (fraction of time)
U	—	unavailability (fraction of time)

Appendix B

General solution to equation 8.5

Let

$$\begin{aligned}
D &= \lambda_n^4 N(-6+11N-6N^2+N^3)+\\
&\quad \lambda_n^3\rho_a(-6+(22+7GZ_r+2G^2Z_r^2)N+(-18-4GZ_r)N^2+4N^3)+\\
&\quad \lambda_n^3\rho_r GZ_r N(-5+3N)+\\
&\quad \lambda_n^2\rho_r\rho_a(6-5GZ_r+(-9+4GZ_r)N+3N^2)+\\
&\quad \lambda_n^2\rho_r^2 GZ_r N+\\
&\quad \lambda_n\rho_r^2\rho_a(-3+3GZ_r+2N)+\\
&\quad \rho_r^3\rho_a
\end{aligned}$$

The general solution to

$$\begin{bmatrix} -N\lambda_n & \rho_r & 0 & 0 & \rho_a \\ N\lambda_n & -(N-1)\lambda_n-\rho_r & \rho_r & 0 & 0 \\ 0 & (N-1-GZ_r)\lambda_n & -(N-2)\lambda_n-\rho_r & \rho_r & 0 \\ 0 & 0 & (N-2-2GZ_r)\lambda_n & -(N-3)\lambda_n-\rho_r & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_a \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

then becomes:

$$\begin{aligned}
p_0 &= \rho_a(\frac{\lambda_n^3(-6+11N-6N^2+N^3)+\lambda_n^2\rho_r(6-5GZ_r+(-5+3GZ_r)N+N^2)}{D}+\\
&\quad \frac{\lambda_n\rho_r^2(-3+3GZ_r+N)+\rho_r^3}{D})\\
p_1 &= \lambda_n\rho_a N\frac{(6-5N+N^2)\lambda_n^2+(-3+2GZ_r+N)\lambda_n\rho_r+\rho_r^2}{D}\\
p_2 &= \lambda_n^2\rho_a N\frac{(-3\lambda_n+\lambda_n N+\rho_r)(-1-GZ_r+N)}{D}
\end{aligned}$$

$$\begin{aligned}
p_3 &= \lambda_n^3 \rho_a N \frac{(-1 - GZ_r + N)(-2 - 2GZ_r + N)}{D} \\
p_a &= \lambda_n^2 N \frac{\lambda_n^2(-6 + 11N - 6N^2 + N^3) + \lambda_n \rho_r(-5GZ_r + 3GZ_r N) + GZ_r \rho_r^2}{D} \\
1 - p_0 &= \lambda_n N (\frac{\lambda_n^3(-6 + 11N - 6N^2 + N^3) + \lambda_n^2 \rho_r 5GZ_r(-5 + 3N)}{D} + \\
&\quad \frac{\lambda_n^2 \rho_a(11 + 7GZ_r + 2G^2Z_r^2 + (-12 - 4GZ_r)N + 3N^2)}{D} + \\
&\quad \frac{\lambda_n \rho_r^2 GZ_r + \lambda_n \rho_r \rho_a(-4 + 2N + GZ_r) + \rho_r^2 \rho_a}{D})
\end{aligned}$$

Assuming $NG\lambda_n \ll \rho_r$ and $N\lambda_n < \rho_a$ we get:

$$\begin{aligned}
D &\approx \rho_r^3 \rho_a \\
p_0 &\approx 1 \\
p_1 &\approx \frac{\lambda_n N}{\rho_r} \\
p_a &\approx \frac{\lambda_n^2 GZ_r N}{\rho_r \rho_a} \\
1 - p_0 &\approx \frac{\lambda_n N}{\rho_r}
\end{aligned}$$

Appendix C

Article: Application of an abstract machine supporting fault-tolerance in a parallel database server

This appendix contains an article published in *International Journal of Computer Systems Science & Engineering*, volume 9, number 2, April 1994 [TH94].

Article, page 134

Article, page 135

Article, page 136

Article, page 137

Article, page 138

Article, page 139

Article, page 140

Article, page 141

Bibliography

[Ada84] Edward N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, January 1984.

[Ask89] Ole John Aske. Aksessmetoder for parallell utførelse. Diploma Thesis, In Norwegian, March 1989.

[Avi85] Algirdas Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[Bar78] Joel F. Bartlett. A nonstop operating system. In *Proceedings 11'th Hawaii International Conference on System Sciences*, volume 3, pages 103–117, 1978.

[BBG$^+$89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[BD83] Haran Boral and David J. DeWitt. Database machines: An idea whose time has passed? A critique of the future of database machines. In Leilich and Missikoff, editors, *Database Machines*, pages 165–187. Springer Verlag, 1983.

[BEHS86] E. Brederup, K. Evensen, B. E. Helvik, and A. Rygh Swensen. The activity dependent failure intensity of SPC systems. Some empirical results. *IEEE Journal on Selected Areas in Communication*, SAC-4(7):1052–1059, 1986.

[BG88] Dina Bitton and Jim Gray. Disk shadowing. In Francuis Bancilhon and David J. DeWitt, editors, *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 331–338, Los Angeles, California, USA, August 1988.

[BG89] Kjell Bratbergsengen and Torgrim Gjelsvik. The development of the CROSS8 and HC16-186 parallel (database) computers. In H. Boral and P. Faudemay, editors, *Proceedings from Sixth International Workshop on Database Machines, Lecture Notes in Computer Science 368*, pages 359–372. Springer-Verlag, June 19-21 1989.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Inc., 1987.

[Bhi88] Anupam Bhide. An analysis of three transaction processing architectures. In Francuis Bancilhon and David J. DeWitt, editors, *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 339–350, Los Angeles, California, USA, August 1988.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BM93] W. Burkhardt and J. Menon. Disk array storage system reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, June 1993.

[Bro85] Russel C. Brooks. An approach to high availability in high-transaction-rate systems. *IBM System Journal*, 24(3/4):279–293, 1985.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in Bubba. In *Proceedings of the ACM SIGMOD Conference*, pages 99–108, June 1988.

[CK89] George Copeland and Tom Keller. A comparison of high-availability media recovery techniques. In *Proceedings of the ACM SIGMOD Conference*, pages 98–109, June 1989.

[CLG$^+$94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[Cox90] Butler Cox. *Computer Disasters and Contingency Planning*. Amdahl International Management Services Ltd, Hartley Wintney, Hampshire, England, 1990. Research Report EM001024 001 [1:15] 7/90.

[Cri85] Flaviu Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, January 1985.

[Cri90] Flaviu Cristian. Understanding fault-tolerant distributed systems. Research report, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099, USA, 1990.

[CS92] Donald D. Chamberlin and Frank B. Schmuck. Dynamic data distribution (D^3) in a shared-nothing multiprocessor data store. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 163–174, August 1992.

[Cua89] William G. Cuan. Fault tolerance: Tutorial and implementations. *IEEE Computing Futures*, (Inaugural Issue):30–39, November 1989. Supplement to IEEE Computer, November 1989.

[Dat86] C. J. Date. *An Introduction to Database Systems*, volume 1 of *System Programming Series*. Addison-Wesley, fourth edition, 1986.

[Dep85] Department of Defence. *Department of Defence Trusted Computer System Evaluation Criteria*, December 1985. DOD 5200.28-STD.

[DGS$^+$90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[ES92] Steven A. Elkind and Daniel P. Siewiorek. Reliability techniques. In Daniel P. Siewiorek and Robert S. Swarz, editors, *Reliable Computer Systems: Design and Evaluation*, chapter 3, pages 79–227. Digital Press, Burlington, MA, USA, 2 edition, 1992.

[Fre82] Robert Freiburghouse. Making processing fail-safe. *Mini-Micro Systems*, pages 255–264, May 1982.

[Fuj91] Fujitsu M2622SA, M2623SA and M2624SA datasheet. Posted on Usenet comp.periphs.scsi conference May 11. 1992, February 1991.

[GD90] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 481–492, August 1990.

[GHW90] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 148–161, August 1990.

[Gil76] Arthur Gill. *Applied Algebra for the Computer Sciences*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1976.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA, 1992.

[Gra85] Jim Gray. Why do computers stop and what can be done about it. Technical Report 85.7, PN87614, Tandem Computers, June 1985.

[Gra90] Jim Gray. A census of Tandem Systems availability between 1985 and 1990. Technical Report 90.1, Part Number: 33579, Tandem Computers, January 1990.

[Gra91] Jim N. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA, 1991.

[Gro88] The Tandem Performance Group. A benchmark of NonStop SQL on the Debit Credit transaction. In *Proceedings of the ACM SIGMOD Conference*, pages 337–341. ACM Press, June 1988.

[GS91] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[Gue91] Jorge Guerrero. RDF: An overview. *Tandem Systems Review*, 7(2):34–43, October 1991.

[Ham50] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, April 1950.

[HD90] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machine. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 456–465, Los Alamitos, CA, February 1990. IEEE Computer Society, IEEE Computer Society Press.

[HD91] Hui-I Hsiao and David J. DeWitt. A performance study of three high availability data replication strategies. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 18–28, Los Alamitos, CA, 1991. IEEE Computer Society, IEEE Computer Society Press.

[HST91a] Svein O. Hvasshovd, Tore Sæter, and Øystein Torbjørnsen. Critical issues in the design of a fault-tolerant multiprocessor database server. In *Proceedings from the 1991 Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 226–231. IEEE Computer Society Press, September 26-27 1991.

[HST$^+$91b] Svein O. Hvasshovd, Tore Sæter, Øystein Torbjørnsen, Petter Moe, and Oddvar Risnes. A continuously available and highly scalable transaction server: Design experience from the HypRa project. In *Proceedings from The Fourth International Workshop on High Performance Transaction Systems*. Springer-Verlag, September 22-25 1991.

[Hva92] Svein-Olaf Hvasshovd. *HypRa/TR - A Tuple Oriented Recovery Method for a DDBMS on a Shared Nothing Platform*. Dr.Ing. thesis, Division of Computer Science and Telematics, Department of Electrical Engineering and Computer Science, The Norwegian Institute of Technology, University of Trondheim, Norway, 1992.

[IBM87] IBM Corporation. *IMS/VS Extended Recovery Facility (XRF): Technical Reference*, April 1987. GG24-3153.

[ICL93] ICL. *GOLDRUSH MegaSERVER: Technical Overview*. Sales brochure, ICL Corporate Systems, Manchester, UK, 1993. B93 / 307.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. New directions in scaleable shared-memory multiprocessor architectures: Scaleable Coherent Interface. *IEEE Computer*, 23(6), June 1990.

[Kat78] James A. Katzman. A fault-tolerant computing system. In *Proceedings 11'th Hawaii International Conference on System Sciences*, volume 3, pages 85–102, 1978.

[Kim84] Won Kim. Highly available systems for database applications. *ACM Computing Surveys*, 16(1):71–98, March 1984.

[Kim86] Michelle Y. Kim. Synchonized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Inc., 1973.

[Lap85] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings 15th Int. Symp. Fault-Tolerant Computing*, pages 2–11, Los Alamitos, California, USA, 1985. Computer Society Press.

[LI93] Inhwan Lee and Ravishankar K. Iyer. Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 20–29, June 1993.

[Lit80] Witold Litwin. Linear hashing: A new tool of file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 212–223, October 1980.

[Liv87] Miron Livny. Multi-disk management algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77. ACM Press, May 1987.

[LK91] Edward K. Lee and Randy H. Katz. Performance consequences of parity placement in disk arrays. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 190–199, Santa Clara, California, USA, April 1991. ACM.

[LNS93] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH* — Linear hashing for distributed files. In *Proceedings of the ACM SIGMOD Conference*, pages 327–336. ACM Press, June 1993.

[MAC92] M. Randall MacBlane, Juan M. Andrade, and Mark T. Carges. The TUXEDO enterprise transaction processing system. In Lorenzo Bonanni, editor, *Proceedings of USING '92*, pages 341–348, Philadelphia, Pennsylvania, USA, May 1992.

[Mey80] J. Meyer. On evaluating the performability of degradable computer systems. *IEEE Transactions on Computers*, C-29(8):720–731, August 1980.

[MM92] Jai Menon and Dick Mattson. Comparison of sparing alternatives for disk arrays. In *The 19th Annual International Symposium on Computer Architecture*, pages 318–329. ACM Press, May 1992.

[Mot91] Motorola Computer Group, OEM News, no. 3. Newsletter published by Motorola AB. Sweden, April 1991.

[Org89] International Standards Organization. ISO / IS 7498 / 2, Information Processing Systems - Open Systems Interconnection - Basic Reference Model, part 2: Security Architecture. International standard, 1989.

[Org92a] International Standards Organization. ISO / IEC 9075:1992, Database Language SQL92. International standard, 1992.

[Org92b] International Standards Organization. (Working Draft) Database Language SQL3. Working draft, June 1992.

[Par93] Craig Partridge. *Gigabit Networking*. Addison-Wesley Publishing Company, Inc., 1993.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference*, pages 109–116, June 1988.

[RAA$^+$88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. CHORUS distributed operating systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 64–75. ACM Press, December 1981.

[Ser84] Omri Serlin. Fault-tolerant systems in commercial applications. *IEEE Computer*, 17(8):19–30, August 1984.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *International Conference on Data Engineering*, pages 336–342. IEEE Computer Society, Computer Society Press, February 1986.

[Sie90] Daniel P. Siewiorek. Fault tolerance in commercial computers. *IEEE Computer*, 23(7):26–37, July 1990.

[Sie92] Daniel P. Siewiorek. The IBM case. In Daniel P. Siewiorek and Robert S. Swarz, editors, *Reliable Computer Systems: Design and Evaluation*, chapter 7, pages 485–523. Digital Press, Burlington, MA, USA, 2 edition, 1992.

[SS90] Michael Stonebraker and Gerhard A. Schloss. Distributed RAID — A new multiple copy algorithm. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 430–437, Los Alamitos, CA, February 1990. IEEE Computer Society, IEEE Computer Society Press.

[Sto86] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9(1):4–9, March 1986.

[STR88] R. M. Smith, Kishor S. Trivedi, and A. V. Ramesh. Performability analysis: Measures, an algorithm, and a case study. *IEEE Transaction on Computers*, 37(4):406–417, April 1988.

[Su88] Stanley Y. W. Su. *Database Computers; Principles, Architectures & Techniques*. Computer Science. McGraw-Hill, 1 edition, 1988.

[Ter85] Teradata. *DBC/1012 Database Computer System Manual Release 2.0*. Teradata Corp., November 1985. Document No. C10-0001-02.

[TH94] Øystein Torbjørnsen and Svein-Olaf Hvasshovd. Application of an abstract machine supporting fault-tolerance in a parallel database server. *International Journal of Computer Systems Science & Engineering*, 9(2):134–141, April 1994.

[WB87] W. Kevin Wilkinson and Haran Boral. KEV — A kernel for Bubba. In Masaru Kitsuregawa and Hidehiko Tanaka, editors, *Database Machines and Knowledge Base Machines*, pages 31–44. Kluwer Academic Publishers, 1987.

[WDJ91] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 537–548, September 1991.