# Real-time databases : an overview

# Real-time databases
## An overview

M.P. Bodlaender
*Department of Computer Science, TUE*
*The Netherlands*

April 7, 1995

# Contents

# Chapter 1

# An introduction to (real-time) distributed databases

Real-time distributed databases extend the power of centralised and distributed databases. Mechanisms are provided to incorporate the notion of time within the database semantics.Though databases already have a notion of time, since they try to compute as fast as possible, this is not sufficient in time-critical environments. In this chapter an overview of the differences between centralised, distributed and real-time databases is provided. Note that it is possible to construct a real-time centralised database as well as a real-time distributed database.

## 1.1 Centralised databases

The theory of **centralised databases** is well-developed, see for example [Pap79], [KR81], [YA88], [Vid85], [Vid91]. In general, arbitrary actions on a database have to satisfy the following two requirements: they must not disturb the logical consistency of the database and they must be efficient. The primitive actions that can be applied to the database are read and write actions. These actions access a single data item to either read or change its value. To be able to reason about database actions a **transaction** is defined as a collection of primitive actions (i.e. Read, Write) that is applied to the database.

Even when each transaction on the database leaves the database in a consistent state, a collection of transactions that is executed in an interleaved fashion can destroy that consistency. The (partial) order in which transactions are executed is called a **schedule**. If two transactions are unordered, their basic actions can be executed in any interleaved fashion. In articles [Pap79], [Vid85], [Vid91] correct schedulers are defined that order the transactions in such a way that database consistency is preserved.

The most important notion that has been developed is **serializability**: if a partially ordered schedule is serializable (proven equivalent to a totally ordered

schedule), database consistency is ensured. Note that articles [GM83] and [GS85] illustrate that the class of serializable schedules is a strict subset of all consistency-preserving schedules.

Naturally, the transactions on a database must be efficient. Often, large amounts of data must be manipulated when complex transactions are performed on the database. The order in which certain basic steps are applied to the database has a great influence on the execution time of the transaction and a transaction manager that executes transactions in an efficient way is needed. In articles [IK94], [SY82] and [JK84] transaction management and query optimization are treated in depth.

## 1.2 Distributed databases

Recently, the wide-spread availability of computer networks calls for **distributed databases**. These databases try to exploit the properties of a computer network to increase the reliability, concurrency, capacity and speed of databases. A book that combines most aspects of distributed databases is [ÖzsuV91].

Why these enhancements can be expected from a distributed database is shown easily. Reliability can be increased because information can be replicated over multiple sites, thus lowering the probability that the crashing of a site leads to loss of information.

Because the database is actually divided into several smaller databases, it is often possible that small tasks are only performed at one or a few sites, leaving the other sites available for other tasks. This feature increases the amount of concurrency in the system, as multiple users can access the database at the same time.

In the current information age, large databases are needed to store all the information needed in complex organisations. However, the current state of hardware technology limits the size of a database a single computer can handle. The trend in computer architecture is towards a local area network of computers of intermediate size. These architectures are more powerful and are able to store unlimited amounts of data, as the size of the database can be expanded by adding an extra computer to the local area network. Therefore, mechanisms must be provided to deal with this fundamentally different architecture.

When a database is distributed over more than one computer, the computational power of the individual computers ceases to be the bottleneck of the architecture. While the maximal speed of a centralised database is dependent on technology of its CPU, this is not the case for distributed databases. This is because computations can be divided over the computers in the network. If a structural overload of the system occurs, it is possible to add more computation power to the system by adding extra computers to the network.

The bottleneck of the distributed database design is the communication cost.

If an information intensive transaction is processed that needs to access large parts of the network, the costs of communication rise rapidly. Even worse: the more computers participate in the distributed database, the more communication will be needed. Part of the research in distributed databases is directed at minimizing the communication cost of transactions that are performed on the distributed database.

## 1.3 Real-time databases

Databases have been used in various ways, but most applications of databases have been administrative. Databases typically try to fulfill two basic requirements:

- Operations on the database have to preserve the consistency of that database.
- The transaction throughput of the database should be as high as possible.

In real-time systems the computer interacts with an outside world that is constantly changing. Real-time systems often deal with temporal data, e.g. data that is only valid for a certain interval in time. This means that old data is as good as no data (i.e. Take data about the position of a moving object at some moment t. After several seconds the data will no longer reflect the position of the object in the real world. The data is no longer valid). Likewise, if a computer controlling a bridge decides that at time-interval $[t, t+d]$ it must be open because a ship will then pass, we don't want that bridge to be open long before time $t$ or after time $t + d$, for this would hold the traffic longer than necessary. These two examples illustrate two extra conditions that we impose on **real-time databases** to preserve logical consistency:

- Internal data that represents the status of objects in the real world should accurately reflect the real status of the objects within an acceptable margin.
- Transactions of the database may only be executed in a certain time-interval. Most important, all transactions have a deadline after which the transaction fails.

In real-time databases schedulers should dispatch transactions such that they meet their deadlines. Therefore transactions that are nearing their deadline should be scheduled before other transactions. And in overtaxed systems that cannot meet all deadlines, we want to ensure that certain important transactions never fail, thus sacrificing other, less important transactions.

While in classical databases the primary goal is to preserve the database consistency, this is not always the case in real-time databases. For some applications it is more important that a transaction completes before its deadline than it is

to preserve the database integrity. Therefore, current research is investigating the tradeoff between consistency and speed, see [KR92] or [KM93]. In a lot of applications, inconsistency can be tolerated as long as it is bounded.

## 1.4 Comparing the various database types

Each database-structure has been designed for a specific environment and with specific goals in mind. Low-cost centralised databases are very well suited for administrative purposes. The theory has been well-developed and 2PL (two-phase locking, a scheduling mechanism) is used all over the world.

Distributed databases offer all the services of a centralised database. More than a centralised database they offer concurrent access by multiple users. Data replication can make a distributed database more reliable than a centralised database. Distributed databases can easily be upgraded, as a good database design will allow for adding computers and storage to the distributed network.

Real-time databases explicitly deal with the notion of time. In applications where computers are used to control some environment they offer essential services. The most important service they provide is the meeting of transaction deadlines. A real-time database guarantees that, if the system is not overloaded, all transactions will finish execution before their deadline.

A priority mechanism can also be offered by Real-time databases. When the database cannot complete all transactions in time, it tries to ensure that transactions with higher priorities still meet their deadline. Thus real-time databases are also useful in areas where critical processes must be monitored along with less critical activities.

## 1.5 Organization of this paper

In the next nine chapters the main issues in real-time distributed database design will be briefly introduced. In no way an attempt is made to give a complete overview of the field, but hopefully the reader develops some global insight in the strengths and weaknesses of real-time distributed databases.

Chapter two is the justification of the research area, it provides a high level description of what services a real-time database offers and the resources that it needs to do so.

Chapters three to eight give introductions to different issues that relate to real-time distributed databases. In chapter nine it is observed that testing and comparison techniques used to date are fairly ad hoc and could use a more systematic approach. Chapter ten concludes with a summary of the issues that still need development in order to produce efficient real-time databases.

# Chapter 2

# What can a real-time database do for you?

The database design that has been used in many applications is a centralised, non-real time database. It provides access to the database to a limited number of users at the same time. Data-consistency is ensured and the database tries to execute as efficiently as possible. Distributed databases allow the databases to be implemented on a more general system architecture. They increase the reliability and availability of the database.

Real-time databases, centralised or distributed, deal explicitly with the notion of time. Data items in the database can reflect objects in the real world. These data items have to be updated by the real-time database, to maintain a correct view of the real-world. Also, the changing of a data item in the database may have effects in the real-world, for instance the movement of a robot-arm.

## 2.1 Real-time scheduling

To interact correctly with the environment, the real-time database allows transactions on a database to be scheduled according to some time based criterion. For each transaction $t$ an interval $[s_t, d_t]$ can be specified such that the transaction $t$ will not be executed before starting-time $s_t$, and $t$ will be finished before deadline $d_t$.

If the information stored in the real-time database is used to derive an action that should be taken by the database somewhere in the future, it is possible to schedule this action. At the appropriate time it will be executed. This is best illustrated by an example. Suppose that inputs from an automated factory have been used to conclude that between 2am and 3am the workload is low enough to shut off the machines. With a real-time database it is possible to schedule two transactions, one at 2am and one at 3am that shut down and restart the machines, respectively. It can be seen that the real-time database can be used to

interact with the environment, controlling parts of it.

It is important to realize that a number of different implementations of real-time systems are possible. These implementations could offer different services to the users, depending on the application of the real-time database. In the next sections some properties that a real-time system could provide are investigated. However, although these properties are often useful, they have their drawbacks. Therefore, not all real-time databases will offer all these properties. It should be clear that real-time databases must be tailored to suit each individual application.

## 2.2 Transaction priorities

In the ideal situation all transactions that are executed by the real-time database compute correctly and meet their deadlines. Unfortunately this is often not a very realistic assumption, the database can be confronted with an overload of transactions that all have to be completed within reasonable time. Even if the database is very efficient and fast, it could occur that it is unable to meet all deadlines.

In these situations, a number of transactions have to be cancelled. To provide the user with some control over the cancelling of transactions, each transaction is given a **priority** by the user. Now transactions with high priorities take precedence over transactions with low priorities if the database cannot meet all deadlines.

In general, this leads to an abort of an executing transaction, to allow high priority transactions to complete in time. The work that was already done by the aborted transaction is wasted. So a priority based scheduler degrades the throughput of the system. Although several schemes to reduce this degradation of throughput have been proposed, none of them do fully solve this problem. If throughput of the system is more important than the timely execution of individual transactions, user priorities should not be used.

## 2.3 Performance of real-time databases

In not-realtime databases the performance of the database is judged by its transaction throughput. This criterion is not satisfactory for real-time databases, as it does not take the deadlines of transactions into account. The performance of a real-time database is expressed in the number of transactions that meet their deadlines.

There is a sharp distinction between these two notions of performance. According to the real-time performance criteria, a database that processes thousand transactions in one hour, but misses each deadline by a few seconds is less efficient than a database that processes only hundred transactions which meet their

deadlines. If the classic notion of database performance is used this would not be the case.

In the next two subsections two interesting techniques that can be used to increase the performance of the real-time database are mentioned.

### 2.3.1 Sacrificing correctness for performance

Whereas correctness is the main issue in classical databases, it is often more desirable to have some (partially) incorrect result on time than a correct result that arrives too late. Correctness can be traded for an increase in speed, raising the probability that transactions meet their deadlines. Of course, this is very application specific, but it is an interesting tradeoff that should not be forgotten.

A number of techniques have been proposed to bound the amount of inconsistency that can be allowed without invalidating the database to a point where it does no longer produce sensible output. For instance, it is not a big problem if a door-controlling computer opens the door once in a while without anyone present to enter the door. However, if it remains closed when people are waiting to enter, it is unacceptable. Another clear example is a climate controlling system. If it heats the room to 25 degrees, we find it irritating. But when the climate control decides that the room should be heated to 40 degrees, we shut it down as soon as possible!

### 2.3.2 Sacrificing generality for performance

A quite different approach that is used to increase the speed and throughput of real-time databases, is restricting the generality of the actions that can be applied to the database. As real-time databases are often applied for very specific purposes, this does not have to restrict the power of the database too much. If information about the types of transactions that will be processed by the database is available in advance, it is often possible to produce more efficient schedulers. This increases the performance of the database.

As a small example, suppose that it is known in advance that there is only one (periodic) transaction that writes to a data item. Other transactions only read the data item. With this information about the access behaviour of transactions, efficient scheduling of the transactions in question is possible. In fact, if a multi-version database is implemented, no concurrency control is needed at all! The writing and reading transactions can execute completely concurrent [1].

This technique can not be used in environments where no knowledge is available in advance or in environments where the transactions have no 'nice' properties that can be exploited for this purpose. Notwithstanding these negative observations, this can be a useful method to improve the database performance.

[1] For more information about multi-version databases see for instance [Wei87]

# Chapter 3

# Atomic transactions

One of the most important properties of database management systems is guaranteed data consistency. There can be various syntactic and semantic constraints on the information stored in the database. The technique that is generally used to enforce these constraints is the notion of **atomic transactions**.

## 3.1 Defining transactions

A **transaction** is a set of operations that is applied to the database in a certain order. Programmers of transactions have to ensure that the execution of a transaction on a consistent database leaves that database in a consistent state. Transactions are 'atomic', because either all the effects of a transaction are carried out, or the transaction doesn't take place at all. Other transactions will either see all effects of an atomic transaction, or no effects at all. In this way, the database consistency is preserved if all transactions are executed in a sequential way.

At some point, a transaction has to decide whether to complete the execution or to abort. This is called **committing** the transaction. Once a transaction has been committed, it is certain that all its effects are visible to other transactions.

## 3.2 Constructing transactions

It has been observed that information about transactions that is known in advance sometimes enables more efficient scheduling of transactions. Transactions are required to leave the database in a consistent state. In this section it is specified how transactions can be constructed.

### 3.2.1 Linear transaction model

The classic way to represent a transaction is as a list of read and write actions on data items. These actions are executed in a specified order. It is assumed that a transaction does some computation depending on the data items it reads and that some of the results of this computation are written back to data items in the database. Computations are not explicitly represented in this model. Typically, a **read action** or a **write action** accesses only a single data item in an atomic way (i.e. if the transaction fails, it does so between two basic actions, not during a basic action).

#### Linear transactions as a computation-model

The representation of transactions defined above is very well suited for reasoning about the scheduling of transactions and about interleaving of executions. This is because most scheduling is based on the "reads-from" relation. In general, transactions interact with each other by reading and writing data items. By representing a transaction as just a sequence of reads and writes the constraints of this interaction are explicitly captured.

#### The reads-from and writes-writes relations

The **reads-from relation** between transactions is defined as follows: a transaction $t_1$ reads from $t_2$ if $t_1$ reads a data item $X$ whose actual value has been written by $t_2$. Analogously a **writes-writes** relation exists between $t_1$ and $t_2$ if $t_1$ overwrites the value of a data item $X$ that has been written by $t_2$.

### 3.2.2 Nested transaction model

The transaction model presented in the previous section imposes only a simple structure on transactions. Worse, it supposes that each transaction is a sequential execution of basic actions. To express more general (concurrent) transactions while maintaining a strong grasp on the structure of transactions, the **nested transaction model** is introduced. Each transaction is represented as a hierarchy of transactions nested in transactions. Before, database consistency was required before and after the execution of a transaction. During the transaction, the database could be in an inconsistent state.

It is possible to require that each sub-transaction is a complete transaction itself: if it finds the database in a consistent state, it will leave the database in a consistent state. If this choice is made, the number of ways in which a transaction can be fragmented into sub-transactions is reduced. Thus it is harder to define transactions. On the other hand, it becomes possible to allow parts of a transaction to be used by the rest of the database while other parts are still in progress. Each sub-transaction can be regarded as a complete transaction. It can

therefore commit without waiting for other transactions, as it leaves the database in a consistent state. A more fine-grained concurrency control is possible when sub-transactions are complete transactions themselves.

### Nested transactions as a computation-model

In the nested transaction model a transaction is represented by a tree structure, where the leaves of the tree are the basic read and write events. This is a quite natural way to represent transactions. A lot of our programming languages are constructed as trees, where procedures are nodes and function-calls are links between nodes. Leaves are made up from the language-primitives.

If nodes that sequentially execute their children and nodes that execute their children in parallel are allowed, a very generic computation model is obtained. Some additional synchronisation between concurrent computations in different nodes can be obtained by communication between these computations. Allowing parallel execution within a transaction increases the amount of concurrency that the system allows, thus improving the performance of the system.

### Use of nested transactions in a distributed network

Another benefit of nested transactions is that it is easy and natural to implement the distribution of transactions with them. The sub-transactions that have to execute on other sites than the initial transaction can be represented by sub-trees of the transaction-tree.

By representing the computation of each site by a sub-transaction, the execution of a transaction is defined by the execution of the sub-transactions and the communication between them. Communication between sites is often a bottleneck in distributed systems. By making the distinction between sites explicit in the model, it is possible to analyse the message complexities of transactions in terms of communication between sub-transactions.

# Chapter 4

# Concurrency control

Modern system designs have made it possible to execute processes concurrently, thus increasing the throughput of the systems. By concurrent execution of transactions the number of transactions that can be processed in a period of time is increased.

It is not possible to execute all transactions at the same time. A transaction that uses the result of another transaction has to wait until that result becomes available. Also, two transactions that both try to access a critical section (for example a printer) cannot run concurrently.

If two transactions are executed in parallel we imagine that their basic steps are executed in an interleaved fashion, not exactly at the same moments. This eases reasoning about concurrent transactions.

## 4.1 Concurrency and consistency

There is a strong relation between the amount of concurrency allowed by the databases and the maintenance of data-consistency. Transactions are designed in such a way that the execution of a single transaction leaves the database in a consistent state.

It is much harder to satisfy the consistency requirement if transactions are processed in an interleaved fashion. Other transactions can interfere with the execution of a single transaction, thus invalidating its execution. An example of this is given in figure 4.1. The consistency requirement is that accounts $A$ and $B$ sum to zero. However, due to the incorrect interleaving of basic actions of two transactions, this consistency is destroyed.

An explanation of the figure is probably helpful. Normally, a transaction is modelled as a sequence of read and write actions on data items. To show that arbitrary interleaving of transactions destroys the consistency of the database the internal computation of the transactions 1 and 2 is represented by the small statements. For each data item it accesses, a transaction has an internal vari-

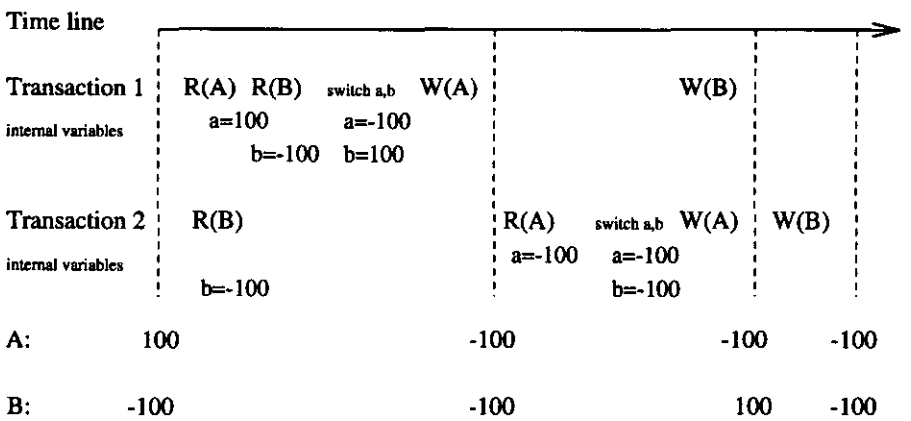


**Figure 4.1:** CONCURRENT TRANSACTIONS DESTROY CONSISTENCY

able representing that data item. If a transaction reads from the database, the result is stored in the corresponding internal variable. Four different "snapshots" show what the state of the database is. As would be expected, the database consistency is disturbed during the execution of the database. The database is still inconsistent after both transactions have finished execution. Therefore the schedule is incorrect.

The schedule show in 4.1 is incorrect because transaction 2 reads part of its data while transaction 1 is executing. During this execution the database consistency is not guaranteed, so transaction 2 can read from a (temporarily) inconsistent database. The correct execution of a transaction is only specified if a transaction reads a consistent database, no consistency requirements are placed on a transaction that reads from an inconsistent database. As can be seen from the example transaction 2 is capable of destroying the database consistency. Therefore a method to determine whether transactions can execute in parallel is needed. This is called **concurrency control**.

## 4.2 The serializability concept

A sequential execution of transactions always preserves the consistency of the database. This leads to the notion of **serializability**. A schedule $s$ is called serializable if there exists some sequential schedule that has an equivalent effect on the database and executes the same transactions. In general the reads-from and writes-writes relationships of $s$ should be preserved, and the final state of the database should be the same. This is called **conflict serializability**.

**Theorem 4.1** *A serializable schedule is a consistency preserving schedule.*

Only an intuitive proof of the theorem is given. Two unrelated transactions can be executed concurrently or in a sequential way, without disturbing the database consistency. Consistency can only be broken by transactions that do have a reads-from or writes-writes relation. If these transactions are executed in an interleaved fashion the database consistency can be destroyed. Exactly this behaviour is prevented by the serializability requirement. Conflicting transactions are scheduled either before or after each other, but not interleaved.

To be able to maximize the amount of concurrency in the database, as many as possible schedules should be allowed.

Note however that there exist consistency-preserving schedules that are not serializable. Therefore, the set of serializable schedules is only a proper subset of the set of consistency preserving schedules. Checking that a schedule is serializable has been proven to be NP-complete.

A schedule is **legal** for a certain scheduler if it can be generated with that scheduler. Existing efficient schedulers all restrict the set of legal schedules to a subset of the serializable schedules, in order to reduce the complexity of generating legal schedules.

### 4.2.1 View serializability

It can be argued that the writing of a data item $X$ that is never read before it is written again is useless. As no one has observed the writing of $X$, there would be no difference if the first write of $X$ had never taken place. To represent this, the notion of view serializability is defined. A certain schedule is **view serializable** if it is equivalent to a sequential schedule that executes the same actions and preserves the reads-from relation between the transactions. Also the final states of the database should be the same. Note that the writes-writes relation between transactions is no longer important. Again, proving that a concurrent schedule is view serializable is NP-complete in the worst case.

It is interesting to note that if some writes are actually useless, the entire execution of these writes can be skipped. This assumes of course, that the overwriting transactions do no fail to complete their execution and abort.

### 4.2.2 Final-state serializability

Where view serializability abstracted from useless writes, final state serializability observes only the final state of a database. Intermediate states during the execution of a schedule are regarded as temporary states. Only the final result of the database is important. This assumption will probably not hold in real-time databases, where transactions can have a visible effect, not only on the database state but also on the real world.

A schedule is **final-state serializable** if it is equivalent to a sequential schedule that executes the same transactions. Equivalence of the schedules is now

defined as equivalence of the final database states that result from executing the schedules.

### 4.2.3 Constructing workable schedules

Solving an NP-complete problem every time a set of transactions has to be scheduled is not a feasible option. Therefore, efficient schedulers that allow only a subset of the serializable schedules to be generated have been constructed. These schedulers can be regarded as heuristic methods to solve the NP-complete scheduling problem. Although they do not provide optimal concurrency, they introduce an acceptable overhead on the system.

A short description of the widely used **two phase locking** protocol is given and the difference between pessimistic and optimistic protocols is examined. Note that the two phase locking protocol serves as an implementation of the two phase locking scheduler.

#### Two Phase Locking

It is assumed that the scheduler is given a set of transactions $T$ and a partial order $\prec$ on $T$. Transactions $t_1$ is ordered before $t_2$ if (wlog.[1]) $t_2$ reads or writes a data item $X$ that has been previously written by $t_1$.

Suppose $t_1 \prec t_2$. The two phase locking protocol forces $t_2$ to wait until $t_1$ has finished, by locking data item $X$. A locked data item cannot be accessed by any other transaction, and $t_1$ does not release the lock until it is about to finish.

For simplicity it is assumed that only one transaction can have a lock on a data item, although optimizations can be made. So transaction $t_2$ has to wait or must abort, unless $t_1$ releases the lock on $X$.

Serializability is not yet enforced by this simple locking mechanism, but with a slight adaptation it will. Two phase locking (2PL) received its name from this adaptation: the protocol consists of a locking phase and an unlocking phase.

A transaction acquires all the locks it needs to execute in the **locking phase.** In the **unlocking phase**, a transaction releases its locks. Once the transaction is in the unlocking phase, it cannot obtain locks anymore.

The two phase locking protocol prevents the following, not-serializable behaviour: transaction $t_1$ locks $X$, writes $X$, releases $X$. Transaction $t_2$ locks, reads, writes and releases $X$. Transaction $t_1$ locks $X$ again and reads it. This is not serializable: $t_2$ reads $X$ from $t_1$. Therefore, $t_2$ must be executed after $t_1$. But $t_1$ reads $X$ from $t_2$, so it should occur after $t_2$! This is a contradiction, so the scheduled transactions are not serializable.

The scheduler does not prevent deadlocks. Deadlocks may occur if two transactions need data items $X$ and $Y$. Transaction $t_1$ has acquired a lock on $X$ and

[1] without loss of generality

needs access to $Y$, transaction $t_2$ has acquired a lock on $Y$ and needs access to $X$. Both must wait for the other transaction.

#### Optimistic versus pessimistic schedules

Two phase locking is a perfect example of a **pessimistic protocol**. It assumes that a lot of conflicts between transactions occur. Therefore, it does not execute a transaction until it is absolutely sure that it does not conflict with any other transaction in progress. This is ensured by the locking mechanism.

However, in a large database the chance that a conflict over a piece of information occurs between two transactions may be very low. If almost no conflicts occur transactions must unnecessarily wait for the locking of their own data items before they are allowed to execute. This observation has led to the construction of **optimistic schedulers**.

An optimistic scheduler first executes the transactions and then validates whether the transaction was executed according to a serializable schedule. If a conflict between two transactions occurs, one of them is aborted just before commit. It is still a point of study to determine under what conditions optimistic schedulers out-perform pessimistic schedulers.

#### Time stamps

Another well known method of scheduling uses **time-stamps**. Each transaction receives a unique time-stamp at some point. Now two transactions that both access the same data item have to be executed in an order that depends on the value of their time stamps. Time stamp schedulers can either be optimistic or pessimistic, depending on the moment that transactions receive their time-stamp and the moment that these time stamps are checked.

## 4.3 Weakening serializability

Determining if a schedule is a serializable schedule is an NP-complete problem. Efficient schedulers that produce serializable schedules never provide optimal concurrency because they use only heuristic solutions of the serializability problem. Also serializability does not completely capture the notion of consistency. To increase the amount of concurrency that schedulers allow, different approaches to database-consistency have been explored.

### 4.3.1 Epsilon Serializability

The first approach to mention is the notion of **epsilon serializability**. Epsilon serializability is a generalization of classic serializability. It explicitly allows some limited amount of inconsistency in transaction processing. This increases the

concurrency allowed by the database as some not-serializable schedules are permitted. In particular, read-only transactions are allowed to run concurrently with update transactions. This might result in a inconsistent view of the database, but the database consistency is not affected. In general, Epsilon serializability bounds the amount of inconsistency that transactions are allowed to see.

#### Implementation outline

With each state of the database an amount of inconsistency is associated. This is defined as the distance of the state to a consistent state. Assume that a function distance$(u, v)$ exists that defines the distance between every pair of states $u$ and $v$. The database state space is **metric** if the distance function is symmetric and satisfies triangle inequality (for all states $u, v, w$ holds: distance$(u, v)$ + distance$(v, w) \geq$ distance$(u, w)$).

Now an epsilon-serializable schedule allows read-only transactions to run concurrently with update transactions if the amount of inconsistency they introduce is bounded by some import-limit. Likewise, an update transaction has some export-limit that specifies the maximum amount of inconsistency that it can export to concurrent, conflicting reading transactions. What limits can be allowed is dependent on the application that uses the real-time database.

Note that reducing the limits to zero gives us the classic serializable schedule. Pessimistic approximations of the amount of inconsistency can be computed if the database state space is metric.

### 4.3.2 Similarity Serializability

**Similarity serializability** is based on the observation that in real-time systems data items will never exactly match the status of objects they are describing. Similarity is a binary relation on the domain of a data object. Intuitively, two objects are similar if they are almost the same.

A schedule is view similar to another schedule if it schedules the same transactions and if these transactions read similar data. So intuitively, a transaction would in both schedules receive almost the same input. View similar schedules are only one version of schedules that are based on similarity. Whether two values are similar depends on the nature of the application of the real-time database.

#### Similarity of time

This concept is introduced to real-time database systems for the notion of time. Two measurements of an object that were taken at approximately the same time can be regarded as similar. This allows the use of slightly older values for read actions, even while the new values are being measured. This eases the problem

of scheduling transactions in real-time. A discussion is presented in the chapter about time management.

## 4.4 Restricting transactions

Advance knowledge about the behaviour of transactions can enable us to do more efficient scheduling. All previous schedulers used the reads-from relation to govern the scheduling of conflicting transactions. However, if for example it is known that all data items are written by only one transaction, all transactions can execute concurrently if a version management scheme is implemented. In the next subsection the version management mechanism is explained.

By restricting the types of transactions allowed in the database the NP-complete serializability problem can be circumvented and efficiently produce highly concurrent schedules. This does of course limit the power of transactions. In the next section an example of a scheduler that exploits this property is provided.

### 4.4.1 Version management

A common transaction is the **read-only transaction**. Typically the user requires information and is not going to change the state of the database. The common occurrence of the read-only transaction justifies the separate treatment that is given here.

In a distributed database several transactions can be issued at roughly the same time. It is often not very clear in what order transactions should be processed. Therefore, if both a read-transaction and a write-transaction are issued and both transactions access the same data item, it does not matter in what order they are serialized. So for a read-only operation it makes no difference if it reads the most recent value or a slightly older one! Bearing this in mind, read-only transactions can be optimized by running them concurrently with update transactions.

#### Multi-version databases

To be able to serialize read-only transactions multiple versions of each data item are kept. Now if a read-only transaction is scheduled, it reads the latest, completed version of the data items it needs that are available at the moment the read-only transaction is scheduled. Update transactions can write newer versions of the data items that are being read, but this does not influence the outcome of the read-only transaction. With this construction it is always possible to serialize a read-only transaction. These transactions can always proceed with their execution.

#### Discarding versions

The existence of more than one version of each data item in the database places a huge demand on the resources that it can use. If old versions of data items are not discarded, the amount of data that needs to be stored by the database will grow out of bounds. A mechanism that discards versions that will not be necessary anymore is needed to make multi-version databases a viable option.

Depending on the exact scheduling mechanism a number of implementations is possible. A general solution is to keep the latest version always in the database and to keep track of the number of transactions that are still using an older version. If transactions are not allowed to be scheduled late (i.e. if a transaction arrives late it is aborted) old versions can be discarded as soon as no read-only transaction uses them anymore.

## 4.5 Handling deadlock and lifelock

Two important problems that should not be forgotten when designing schedulers are the problems of **deadlock** and **lifelock**. In a distributed database system these events cannot be locally detected. It is possible that a transaction $t_1$ waits on $t_2$ in site $a$, while $t_2$ waits on $t_1$ in site $b$. To be able to detect and do something about deadlock in a distributed system communication between the different sites of the database is needed.

### 4.5.1 Deadlock

If the database system makes use of some locking scheme to enforce serializable behaviour, deadlock may occur if two transactions lock a subset of the data items that they both need. No transaction acquires all its data items, so no transaction can proceed. They both wait on each other to release the locks they need. Deadlock can be prevented by checking that the "waits-for" dependencies introduced by the locking scheme are partially ordered. This means that no cyclic waiting may occur. If such a cycle exists, one of the transactions that is part of the deadlock has to be aborted. This checking is done by maintaining a so-called dependency graph. Vertices in the dependency graph denote transactions, and edges between vertices denote "waits-for" relations. An excellent overview of the theory of deadlock detection can be found in [Kna87].

#### Distributed deadlock

A deadlock in a distributed database can extend over more than one site. The information that is known about transactions at a single site is not sufficient to detect deadlocks. Several methods have been designed to detect deadlocks. A few methods are named without going into details.

- Transaction timeouts. If an upper bound of the transaction execution time is known, deadlock can be detected with the use of timers. If a transaction fails to terminate in time, a deadlock has occurred.

- Constructing a global dependency graph. If all sites send their local dependency graph to one site, all dependency graphs can be combined to produce a global global dependency graph.

- Chasing dependencies. If a site notices that a transaction it is processing is dependent on a transaction that is executing at another site, it sends the relevant dependency information to that site. If that site concludes that a cycle occurs (with aid of the received information), the deadlock is detected.

Methods to resolve the deadlock need to follow the detection. The methods are all based on the aborting of one or more transactions that are part of the deadlock. A problem in this area are "shadow deadlocks", i.e. the detection mechanism decides to abort transactions before deadlock has actually occurred.

### 4.5.2 Lifelock

Deadlock cannot occur in optimistic schedules, as transactions never wait. However, lifelock might occur. Lifelock is the situation that, although the database keeps processing transactions, a single transaction is never processed. Suppose that an executing transaction always finds out in the validation phase that it conflicted with a committed transaction. It has to abort the execution and reschedule.

Lifelock can be prevented if the scheduler can choose the transaction it aborts. In general, a transaction has to abort because it conflicts with a set of other transactions. If it is possible to abort this conflicting set, lifelock can be prevented by aborting the transactions that have aborted the least. This ensures that the oldest transaction in the system is not aborted. Eventually each transaction will be the oldest in the system or it will have committed. So eventually each commits.

# Chapter 5

# Reliability

Databases are meant to store information over long periods of time. With our current state of technology it is unrealistic to assume that the database system will never fail. Hardware errors, communication failures, software errors, almost anything can happen. It is possible to design hardware that uses redundancy to decrease the probability of a hardware failure. Likewise, software techniques are shown that prevent failures of the system to leave the system in an inconsistent state.

## 5.1 Failure models

Many different types of failures can occur, as was written in the introduction of this chapter. Two types of failures are recognised, based on the severity of the failure:

- **Fail-stop failures.** If a fail-stop failure occurs in a system, the system simply halts with its computation. After an unknown period, it restarts or continues its computation. When a system is able to continue its computation without losing its program state, the failure is called an **omission.** Omissions preserve the program state, but some results (messages) may have been lost. After a fail-stop failures, the program state has been lost and the system has to reboot. The time that a site needs to recover can be arbitrarily long.

- **Fail-insane failures.** When a fail-insane failure occurs in a system, the system doesn't stop, but it executes in an unpredictable way.

Fail-insane failures are more severe than fail-stop failures. The proof of this is simple: a fail-insane system can decide to behave like a fail-stop system. But it can also decide to continue the computation, acting quite normal but twisting its output. Conclusions based on this output will be incorrect. There is no fool-proof

way of telling if a system behaves correctly, as the checking algorithm itself may produce incorrect output.

## 5.2 Maintaining consistency

In the previous chapter atomic transactions have been defined. The effects of an atomic transaction are either implemented entirely, or not at all. This property is used to maintain the consistency of the database. In this section methods to implement the behaviour of atomic transactions are examined. Transaction atomicity is preserved even when the system fails in the middle of a transaction.

### 5.2.1 Recovery from fail-stop failures

If the system fails in the middle of a transaction, this could lead to an inconsistent database. This happens for example when the system fails after half of the writes of a transaction have been carried out.

So the database has to be repaired when the system recovers. In the worst case, all main memory has been erased by the failure. Stable storage is needed to reconstruct the previous system state. **Stable storage** is a storage device (hard-disk, tape, etc.) that is failure-free. This is often implemented in hardware.

#### The undo/redo mechanism

In order to recover from failures all relevant transaction information is stored in a sequential file on stable storage. This file is called the **log**. Now before the results of a transaction are written, the previous values of the database are saved in stable storage. Then a "begin transaction" message is written to the (sequential) stable storage. Next, the updates of the transaction are actually carried out. When all updates have been applied to the database, "transaction finished" is written to stable storage.

The claim is that with this extra information, the atomicity of transactions can be ensured. Suppose the system has failed. Now when the database recovers, it reads its stable storage until it reaches the last "begin transaction" message. If an "end transaction" message follows, the system failed after completion of the transaction. Nothing needs to be corrected, the system is in a consistent state. If no "end transaction" message has been written to stable storage, the transaction was still in progress. All its writes are undone, by rewriting the previous state of the database that was saved on the stable storage.

The writing to permanent storage[1] may take place after a transaction has committed. If the system fails after the commit but before the actual write to

[1] With permanent storage the normal database storage (hard-disk) is meant. Note that the writing to stable storage is never delayed.

stable storage, it is impossible to undo the transaction. When this happens the log is used to redo the transaction.

This is the simple, centralised implementation of atomic transactions. Adaptations have to be made in a distributed environment but they will still be based on the existence of logs.

### 5.2.2 Handling fail-insane failures

Fail-insane failures are much harder to handle. The assumption of stable storage can not be made, as a fail-insane computer can overwrite its own storage. A solution could be a stable write-once, read-many storage. In this way, all correct actions of the system are preserved. It would be very hard to analyze this storage on recovery, because there will be no sharp boundary between the correct behaviour and the fail-insane behaviour of the system. I personally know of no results in this direction.

In a centralized system nothing can be done once a fail-insane failure occurs. One has to pray that it does not wipe out the entire database. Fail-insane failures can be handled to some extent in distributed databases. Replication of data prevents information to be destroyed by one fail-insane site.

### 5.2.3 Voting on actions

The adverse effects of fail-insane sites can be negated by voting on actions taken by the distributed database. An action on the database will only be executed by all sites if at least a majority of the sites concludes that it is a legal action. For these schemes to be successful, it is necessary that there is a bound on the number of sites that may fail-insane at the same time. Typically, at least a majority of the nodes participating in a vote must be correct.

### 5.2.4 Input certification

A noteworthy technique is that of **input certification**. An insane site that participates in a protocol does not need to send the same information to all sites. This can sometimes result in different conclusions in different correct sites. If the system tries to come to a global decision, this cannot be tolerated.

To prevent insane sites from sending different messages to different sites when they should be broadcasting a single message, a broadcast $b$ from a site $s$ that arrives at site $t$ is not passed on to the controlling system. Rather, site $t$ sends a message $(s, b)$ to all other sites. This message effectively states "I received broadcast $b$ from site $s$". Now if a node receives the same $(s, b)$ message from at least half of the sites in the network, it accepts this message as a correct message. In this way, a fail-insane node can only send the same message to all nodes in the network.

## 5.3 Availability of the database

Another issue of dependability is the availability of the database. If a centralized database fails, the information stored is no longer available. But in a distributed network access to the remaining database sites in the network can still be provided.

### 5.3.1 Fault tolerance

By introducing redundancy in the database it is possible to make the system more fault tolerant. A very simple scheme that is used to build reliable computers is replicating the entire database $X$ times. This $X$-redundant system can now handle $X - 1$ system crashes. If recovery mechanisms are provided, the system can handle $X - 1$ system crashes at roughly the same time.

### 5.3.2 Distributing data

There are several ways to store data in a distributed database. If the database is not redundant, each item is stored at a single site, the crashing of a site will prevent access to items stored at that site. In a lock-based system transactions that accessed data stored in the crashed site have to be aborted. Only transactions that use data items stored in surviving sites can continue execution.

#### Replicating data

Instead of the crude mechanism of replicating the entire database, single data items can be replicated and stored at more than one site. If one site fails, other sites are still able to provide access to all the information in the database.

There are several problems with this approach. Of course, replication of data reduces the overall capacity of the database. Algorithms that were simple and elegant in the not-replicated version become much more involved, if the system incorporates replicated data.

For instance assume that a transaction running at site $t$ has locked a data item. Subsequently the site crashes. A mechanism has to be provided that releases all locks held by transactions on a crashed site. If no such mechanism exists the failure of a single site will prevent access to large parts of the database and no performance is gained from the replication of data.

So the design of the concurrency control mechanism should explicitly deal with the distribution of data. In the chapter on distributed systems, protocols that make use of replicated data to increase availability are discussed.

# Chapter 6

# Distributed systems

Distributed databases are useful because they enhance the reliability and availability of databases. They allow more concurrency than centralised systems and appeal to object-oriented programming approaches.

However, there is a price to be paid for these extra features. The database controlling protocols are more complex than in centralized databases and communication between sites is often a bottleneck. For instance, implementing a lock in a centralised database can be realised with simple semaphores. Implementing a lock in a distributed database requires the exchange of lock information between sites. If information about a lock is distributed over more than one site (to increase availability), the message cost grows in proportion.

In this chapter a few protocols are presented that are specially designed for distributed systems. This is meant to provide some insight in the complexities that arise in distributed systems.

## 6.1 Atomic commit protocols

One of the first problems that is unique for the distributed environment is the global commit. In a centralised database a transaction commits by writing a single message to stable storage. How this could be implemented in distributed databases is not instantly clear. A transaction consists of several sub-transactions. For each site that participates in the transaction a separate sub-transaction is defined. A protocol is needed to ensure that either all sub-transactions commit or that all sub-transactions abort. This is known as an **atomic commit.** All sites should agree on the same decision.

Decisions made by a site are un-reversible, and should be available within finite time. Finally, a transaction should commit if all of its sub-transactions commit, and no failure occurs. This property prevents the obvious solution of always aborting transactions.

### 6.1.1 Blocking

One additional feature that is important for the functionality of an atomic commit protocol is the so called **non-blocking property**. A protocol is blocking if the failure of a site that participates in the protocol blocks further execution. In particular: the protocol cannot abort and has to wait for recovery of the crashed site. The non-blocking property is not easily implemented. Theoretical results show that it cannot be guaranteed if no time-out mechanism or hardware detection of site failures exists. Therefore it is assumed in the rest of this chapter that such a failure-detection mechanism exists.

### 6.1.2 Two phase commit protocol

This is a simple, blocking protocol that offers just the basic services that we demand from an atomic commit protocol. It works as follows: The initiating site sends messages containing the necessary information for the sub-transactions to all sites. Once a site has finished its local computation it either aborts and sends "aborting" to the initiating site or it sends "ready". The initiating site receives all messages. If at least one message is an "aborting" message, the initiating sites sends "abort" to all participating sites and aborts. Otherwise it sends "commit". All participating sites receive the message and abort or commit accordingly.

### 6.1.3 Uncertainty of sub-transactions

Uncertainty is a fundamental property of (sub-)transactions. At the beginning of an execution the sub-transactions are not certain whether the transaction will commit or abort. The computation can still go both ways. At some point in the computation, the decision is made to either abort or commit by each site. Once it is possible that some site has decided on either of the two, no site may decide on an action without information about the decision in the other sites, for otherwise two different decisions could be taken.

With this property in mind, let us analyse the behaviour of the simple two phase commit protocol. At the beginning, no site is allowed to decide to commit. All sub-transactions can safely decide to abort. Therefore as soon as some site fails, the remaining sites abort the transaction.

The analysis becomes interesting once it becomes possible that some site has decided to commit. In the two phase commit protocol, the first site that decides to commit is the site where the transaction was initiated. Suppose some participating site $p$ has sent its "ready" message to the initiating site and the initiating site fails before $p$ has received the decision. Site $p$ is now uncertain whether it has to abort or commit. Using some broadcast protocol it can try to gain certainty from other participating sites.

Suppose the initiating site is the sole failing site. The total set of messages that were sent to the initiating site can be gathered, so the decision that was taken by the initiating site can be deduced. If at least one other site failed, this does not apply. The remaining sites miss relevant information so they cannot infer what the initiating site was about to decide. When all remaining sites are uncertain, no site can decide whether to abort or to commit. The protocol is blocked.

It can be seen that the initiating site is never uncertain, so it can always decide on a course of action. This is because the initiating site is the first site that is allowed to decide to commit. Therefore, as long as the initiating site has not failed, the protocol is not blocked. Likewise, the protocol is not blocked if some remaining site has not yet sent its "ready" message or if some remaining site has already received the decision. The only scenario in which the two phase commit protocol becomes blocked is the scenario just described.

### 6.1.4 Non-blocking commit protocols

It is possible to construct commit protocols that have the non-blocking property. Instead of showing and analysing an entirely new protocol, it is briefly shown how improving a basic step of the two phase commit protocol does provide the non-blocking property.

Recall that the only scenario in which the standard two phase commit protocol is blocking, is when the initiator fails and at least one other site does the same. These two sites could have committed before they failed, so the remaining sites cannot abort. This is because they are uncertain about the decision of the initiator. Implementing an atomic broadcast suffices to realise the non-blocking property. An **atomic broadcast** is a broadcast where either all sites receive the message, or no site receives the message.

#### Achieving non-blocking with atomic broadcast

Now the initiator does not decide on commit until it has finished its atomic broadcast. If it crashes before it has broadcast the decision, it has not yet taken that decision, so the other sites can abort. If it crashes after the broadcast, all sites will have received the decision. Observe that in the previous protocol, delaying the decision till after the broadcast was not sufficient to provide the non-blocking property. This is because a participating site that received the "commit" message and subsequently failed could be the only site that committed if the initiator failed in the middle of the broadcast.

The implementation of an atomic broadcast is beyond the scope of this overview. It suffices to say that it can be achieved at an increased delay in time and with a higher message cost.

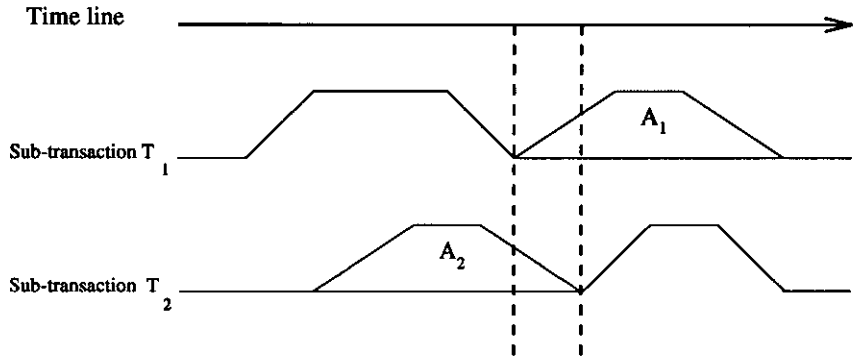


**Figure 6.1**: NOT-TWO PHASE LOCKING BEHAVIOUR

### 6.1.5 Global synchronisation

In many distributed algorithms a global synchronisation point is needed. An example of that is the commit protocol. The initiating site knows that all participating sites have progressed to a certain point (they have all sent their status messages), before it broadcasts its decision. So before sites decide to commit, all sites have at least responded once. Note that reasoning about time in distributed environment is a little more complex than presented here, as sites have no real notion of "global time".

Several different algorithms have been constructed that achieve global synchronisation. The algorithm described above is dependent on its initiating site. Other variants have been designed that increase robustness, decrease time complexity or decrease message complexity.

#### Distributed two phase locking

To be able to design a distributed version of the two phase locking protocol a global synchronisation protocol is needed. Recall that essential for the two phase locking protocol was the existence of a locking phase and an unlocking phase.

Suppose the two phase locking protocol is used to schedule a distributed transaction. It is not sufficient to ensure a local two phase behaviour, as the sites are not synchronised in time. In picture 6.1 an example is given of not-two phase locking behaviour that arises because the sites are not synchronised. Sub-transactions $T_1$ and $T_2$ execute on different sites that are not synchronised. Because of communication delays or because of the difference in speed of the two sites the sub-transactions do no start and stop their locking and unlocking phase at the same moment. The phases are so far apart that $T_2$ begins its locking phase after $T_1$ has finished its unlocking phase. Another transaction $A$ could now read

the results of $T_1$ and write the data items that $T_2$ is going to use. This not-serializable behaviour exists because transactions have no global synchronisation point between the locking and unlocking phase.

## 6.2 Availability of data

Data that is stored in the database should be accessible at all times. Even if some of the sites fail, one would like to manipulate data. This is clearly impossible if data is stored at a single site. However, if data is replicated over multiple sites, it will be available as long as at least one of the sites remains functional.

Although data replication increases the availability of data, it introduces problems for maintaining correctness. Recovery management is needed to update sites that recover from crashes, as changes will have been made to data items that are also stored at the recovering sites. But most important, the concurrency control algorithms have to deal with the replication of data.

Access to a data item is no longer centralised at a single site, but is distributed over the network. Producing correctness preserving schedules requires communication between the sites in the network. A number of distributed concurrency control algorithms are mentioned.

### A simple strategy

Each copy of a data item is treated as a separate data item. If a transaction wants to read or write an item it has to obtain locks on all copies. Obviously this leads to a high communication and storage cost without an increase in concurrency or availability. To design an efficient concurrency control algorithm, mechanisms are needed that increase concurrency and provide access to data even if a few sites fail.

### Single read-lock strategy

A minor adaptation to the previous scheme is that a transaction that just reads a data item $X$ only locks the local copy of $X$. In this way, read actions can be executed concurrently. Write actions still conflict with other writes and reads. Read actions are not blocked if a site fails, write actions have to lock all copies and cannot proceed.

### Primary copy strategy

This section is concluded with presenting a simple version of the primary copy protocol. This protocol maintains a high level of availability, even if some sites fail.

For each data item a **primary site** is defined. The copy of the data item stored there is the primary copy. All other copies are backup copies. Transactions request read and write locks only at the primary site. Therefore, actions are not blocked as long as the primary site remains functional. Other adaptations of the primary site protocol deal with the crashing of the primary site. Still, the simple primary copy strategy is an improvement over the single read-lock strategy where an arbitrary site failure would block the protocol.

### 6.2.1 Network Partitioning

If the network that is the foundation for a distributed database becomes partitioned it would be nice if the two separated parts of the database would remain functional. Information that has only been stored in one partition is unavailable for the other partitions of the database. Transactions that act on this information can only execute if they are issued in the same partition.

So network partitioning cripples the performance of the database in the unreplicated case. But problems are not over if the available data in the database is replicated. If update transactions are applied to data while the network is partitioned, it is possible that two different values are assigned to copies of the same data item. If the network is connected again the database is no longer consistent. Only read-only transactions are allowed in all partitions while the network is partitioned. Update transactions can be allowed in one partition. If updates would be allowed in more than one partition, two or more different copies of one data item can exist. If updates are only allowed in one partition, all other partitions can use the old copy of the data item. (As the network is partitioned, it is certain that transactions running in different partitions can be serialised by putting all the transactions from the read-only partitions before the update-partition).

# Chapter 7

# Time management

In conventional databases information is static: as long as no transaction changes the information, it does not change. This is not the case in real-time databases. Often, information loses value as it grows older. This is especially the case when the information in question is a representation of the real world (hence the name real-time databases), as the real world changes in time. Likewise, if two pieces of information are gathered at completely different times, they do not relate to each other.

## 7.1 Temporal consistency

From the observations just made **temporal consistency** can be formulated in two components:

- **Absolute consistency**. A direct relation must exist between the state of the environment and its representation in the database. If the system has an incorrect view of the environment its actions will be nonsensical.
- **Relative consistency**. Data derived from the environment must be temporally consistent with the other data that has been derived.

Examples of both absolute and relative temporal consistency are given. Suppose a computer is used to monitor the amount of people in a room. If it counts the number of people every five seconds, it will always have a good approximation of the number of people in the room. If it counts once every hour and everybody leaves after thirty minutes, the representation in the computer would no longer reflect the real world, it would be inconsistent.

Now suppose the computer monitors two rooms. First it counts all people in room one in five seconds, then all people in room two. However, while the computer was counting, people were switching rooms. If both counts were ten, can it be concluded there were twenty different people in the rooms? If the rooms

are adjacent about five people could have switched, so only fifteen different people are needed to arrive at our result. The two counts would not be temporally consistent.

But when the rooms are fifty meters apart, almost no one could have crossed that distance in five seconds, so there are indeed twenty different people. The results of the two counts are then indeed temporally consistent, i.e. the fact that in the real world people cannot be in two rooms at the same time combined with the data allows the computer to conclude that there are at least twenty different people present.

### 7.1.1 Absolute consistency

Information that reflects the real world is only valid for a certain interval in time. The length of this interval is dependent on the nature of the object that is represented and the amount of inconsistency that is allowed by the system.

If an object is changing rapidly, the interval during which information is valid will be short. If the database requires that information about an object may only deviate five percent from the real status of the object, the information may have to be refreshed more often than when it is allowed to deviate ten percent.

So far, it has implicitly been assumed that the behaviour of objects is predictable. If an object can suddenly change its entire state, it is impossible to prevent inconsistencies to exist in the database. Likewise, linear behaviour of the information about objects has been assumed. When a small change in data reflects a major change in the state of the object, even small inconsistencies in data can lead to totally wrong conclusions. For example, if a five degree error is allowed in the temperature of water, the difference between ice and water if it is just below freezing point cannot be specified. While for normal temperatures a deviation of five degrees might be acceptable, it is not acceptable around the freezing point of water.

If it is impossible to refresh the information stored in the database often enough to maintain an acceptable representation of the real world, one can use the predictable behaviour of objects to extrapolate the history of an object. If bounds on the speed with which an object can change its status are known, numerical methods can be used to bound the error that is made in the prediction.

### 7.1.2 Relative consistency

Related data about objects in the real world is only consistent with each other if the data was gathered at approximately the same time. This is called the relative consistency of data. As with absolute consistency, relative consistency is dependent on the speed with which the represented objects are changing and the amount of error that is permitted.

But where absolute consistency is only preserved for a small interval in time, relative consistency is a permanent property of a pair of data items. Data items are consistent if they have been gathered within a specific period of time from each other.

Relative temporal consistency is not a transitive relation, if $A$ and $B$ are temporally consistent, and $B$ and $C$ are temporally consistent, it is not necessary that $A$ and $C$ are temporally consistent.

This is easily illustrated. Suppose data items are temporally consistent if their age does not differ by more than five seconds. Now $A$ is gathered at time 10. $B$ has been gathered at time 14. Clearly, $A$ and $B$ differ only 4 seconds and they are temporally consistent with each other. Now $C$ has been gathered at time 17. $B$ and $C$ differ only 3 seconds, so they are consistent. But $A$ and $C$ differ 7 seconds, and they are not relatively consistent.

## 7.2 Time critical scheduling

In a real time environment transactions will have a time-interval associated with them. The real-time database must ensure that each transaction is executed within its own time-interval.

### 7.2.1 Time based scheduling

The problem of scheduling a set of transactions that all have time-intervals in which they have to be processed on a database with limited computation power is NP-complete. This does not even take into account that the transactions also have to be executed in such a way that the database consistency is preserved. Finding a serializable schedule, the most common notion of database consistency, is in itself an NP-complete problem. It is therefore unrealistic to assume that the optimal solution to the time based scheduling problem can be found.

#### Behaviour under overload

If too many transactions have to be processed in a time interval, there will be no solution of the scheduling problem. The database is unable to process all transactions in time. We would like a scheduler that even under these circumstances processes as many transactions within their intervals as possible.

#### Transaction execution length

To do any intelligent scheduling, information about the execution length of transactions is needed. If no such knowledge is present the optimal strategy is to schedule transactions as early as possible. The notion of **slack time** is important. The amount of slack time that a transaction has is the length of the interval in which

it is allowed to execute minus the amount of time that it needs to execute. A correct schedule is easier to find if transactions have a lot of slack time.

### 7.2.2 Missing deadlines

In an overloaded system, the scheduler will not be able to execute all transactions within their associated time-intervals. Where in normal databases it is possible (though not desirable) to queue incoming transaction until workload decreases and the system catches up, in real-time databases the value of a transaction will dwindle away once its deadline has been missed. Dependent on the type of transaction three types of deadlines are recognised: soft, firm and hard deadlines.

#### Soft deadlines

Transactions with **soft deadlines** do not lose their complete value once they have passed the deadline. Instead, the value of a transaction that has passed its soft deadline slowly dwindles away. The most famous example of this is the large project (be it bridge-building, software construction, whatever). The number of times that projects do not make their deadlines is staggering. However, most of them are still completed. It is often better to have finished a project too late, than not to have finished it at all. Clearly, a project cannot go on forever. If no goal is within sight, eventually funding will be stopped, the project will be cancelled, its value is decreased to zero.

#### Firm deadlines

Transactions with **firm deadlines** do lose all their value once the deadline has passed. There is no use in continuing the transaction. Examples of this are all around us. Think of going to the supermarket. If you are half-way to the supermarket and it is closing time, there is absolutely no sense in continuing your trip.

#### Hard deadlines

The last type of deadline that is recognised is the **hard deadline**. If a database fails to execute a transaction with a hard deadline in time, not only does the transaction lose all value, but this failure imposes a heavy negative value on the system. Examples of this lie for instance in computer-controlled security systems. A crude example is the computer monitoring a nuclear power plant. Once the reactor temperature rises above a certain limit, the computer must activate the emergency cooling system. If the computer fails to react in time, a major disaster might occur.

## 7.3 Priority scheduling

The existence of different types of transactions that each have a different impact on the system, should their deadline be missed, leads to the introduction of **transaction priorities**. Clearly the priority of the emergency cooling system is higher than the priority of the daily memo delivery program. The mechanism of priorities can be provided as a service to the users of the real-time database or it can be an implementation of the different types of deadlines that transactions have. On the other hand, priorities can also be used as part of the implementation of a scheduling protocol. For instance in the "Earliest Deadline First" protocol, transaction priorities are defined as the inverse of their deadlines.

### 7.3.1 Defining priorities

Before priorities for all transactions are defined, it must be defined what these priorities exactly mean. In most literature, if transaction $t_1$ has a higher priority than $t_2$, $t_1$ will always have precedence over $t_1$. It is also possible that the scheduler tries to maximize the total sum of priorities of transactions that are executed. That would mean that a transaction of priority five could be aborted by five transactions of priority one.

Imagine a vending machine. Its goal is to earn as much money as possible. Now if it spends too much time with a customer that is about to buy a very expensive article, it can better spend that time selling cheap articles to multiple customers.

The normal priority scheme, where transactions of higher priority always take precedence is more suited to implement critical processes, so in the remainder of this chapter such a priority scheme is used.

### 7.3.2 Handling priorities

Suppose that in a very general scheduler a high priority transaction has a conflict over a data item with a low priority transaction. One transaction has to wait or abort. Suppose too that the low priority transaction is already being executed. Two options are clear: aborting the low priority transaction, or letting the high priority transaction wait.

#### Aborting low priority transactions

A transaction with a high priority takes precedence over transactions with low priorities. A simple implementation to enforce this rule is to abort all low priority transactions that conflict with a high priority transaction. This mechanism ensures that the transaction with the highest priority will always be allowed to

execute. There are several drawbacks to this scheme. First of all, even transactions that were very near commit point are aborted. This results in a large waste of database resources and reduces the overall throughput of the system. Several schemes have been designed to remedy this problem to some extent. In general, they abort transactions that are in early stages of their computation and allow transactions to finish if they are nearly done.

The second problem is that low priority transactions can be life-locked by this mechanism, if there are a lot of high priority transactions. Often if a transaction with a low priority is aborted several times, its priority will rise. Take for example maintenance: missing one maintenance checkup is not very important. However, regular maintenance is essential for a system to keep functioning in a reliable way. Maintenance cannot be postponed indefinitely. This results in a "race for priority" that can disrupt the entire priority scheme.

### Priority inversion

If a high priority transaction that is not yet nearing its deadline has a conflict with a low priority transaction, is does not need to abort that transaction.

Instead of aborting the low priority transaction, it runs to conclusion. But as the high priority transaction is now waiting on a low-priority transaction, it is effectively blocked as the low priority transaction will have to wait on higher priority transactions.

To remedy this problem it has been suggested that the low priority transaction inherits the high priority of transactions that are waiting for it to finish. However, this means that a (once) low priority transaction is now allowed to abort medium priority transactions. This is called the problem of **priority inversion.**

# Chapter 8

# Integrating operating system & database design

Normally when researchers start investigating a subject, an abstract representation of a real problem is formulated. The research focuses on one aspect, instead of looking at the big picture. Once a solution to such an abstract problem is found, this solution is translated back to the real environment and implemented.

Most of the time it is assumed that real-time databases are built on some operating system that offers storage services. The properties of these operating systems are only roughly defined. If the analysis of the database is combined with the analysis of the operating system, more realistic assumptions about the reaction time of the operating system can be made. This enhances the time-control and the precision in which the length of transactions can be predicted.

Operating systems offer file storage services, much in the same way that databases offer more fine-grained storage services. By combining the database with the operating system this replication of services can be avoided, thus reducing the overhead imposed by the system.

These two observations justify the combining of database design and operating system design. In this chapter it will be investigated what can be gained from this combination.

## 8.1 Data caching

To increase the efficiency of the hard-disk it is useful to keep some of the information on the hard-disk stored in main memory. Even if the same information is accessed multiple times, only two disk accesses are needed. One to get the information from the hard disk and one to update the hard disk before the main memory is erased. This technique is called **data caching**.

In large database systems, the cache cannot hold all information that is retrieved from the hard-disk. At some moment, information stored needs to be

erased to make room for new data items that are not yet in the cache. The efficiency of the caching-mechanism depends on the selection of data items that are removed from main memory and stored back to disk.

It is important to take the disk cache explicitly into account during real-time database design. There are two major drawbacks that have to be considered:

- System crashes. If at some moment the system crashes and the main memory is wiped, all changes to data items that were cached are lost. If the disk is cached, it is not certain that a write to the disk is instantly carried out. The recovery mechanism has to be adapted to cope with this extra complication. In general all transactions will be redone whose results might have been lost in the crash.

- Transaction time bounds. In a real-time system tight bounds on the execution time of transactions are necessary to do intelligent scheduling of transactions. If the behaviour of the caching mechanism is not analysed only the worst case scenario can be assumed: the cache is full, data has to be written back to the disk before the new data is retrieved and stored in the cache. So although caching does increase the performance of the hard-disk, it degrades the worst-case analysis as for a single read operation at least two disk-accesses will be needed instead of one.

## 8.2 Virtual memory

Another technique that is frequently used in operating system design is virtual memory. The actual main memory of a computer is often not large enough to completely hold very large programs. The CPU can only access a very small portion of that memory at the same time (typically one or two locations). Therefore large parts of the main memory will not be accessed for some time.

The technique called virtual memory makes use of this property by allowing programs to use more main memory than what is actually available. If a program accesses memory that does not exist, the (virtual) memory manager stores a currently unused part of main memory on hard-disk and offers the now free memory to the program. If main memory that is stored on hard disk is accessed by a program, the memory is retrieved and some other part of main memory is swapped to the hard-disk.

The virtual memory mechanism actually uses the hard disk as slow main memory. To make optimal use of the available (fast) main memory several swapping algorithms are possible. Nevertheless, virtual memory degrades the speed of main memory access.

A tradeoff between available memory and memory speed might be envisaged, if the degradation of speed would be a gradual process. But this is not the case. Memory access is a fast as normal until a program (or transaction) accesses

memory that is stored on hard disk. That memory access initiates the swapping of memory to and from the hard disk. This is disastrous for a worst case analysis of the execution time of transactions.

## 8.3 Conclusion

As illustrated by the examples above, a lot of practical problems surface if the solutions to database problems are exported from an experimental environment to a real environment. Especially, the worst-case execution time of a transaction is affected by disk-IO. Without previous knowledge about transactions, cache hits and page swapping cannot be predicted. As in real-time databases timeliness is often more desirable than fault tolerance, a main-memory system with delayed writes to disk may be more effective.

# Chapter 9

# Analysis of database designs

In real-time database management systems, it is not so important that the database has a high transaction throughput, but rather that each individual transaction has a high chance of completing before its deadline. Although these two notions overlap, they are not the same, as has been illustrated in an earlier chapter.

## 9.1 Existing results

The analysis of the efficiency of real-time database designs has been rather rudimentary. Almost all articles that deal with efficiency give simulation results. Although simulations can be very useful for comparisons between schedulers they lack the thoroughness of the analytical approach. Relative few articles have been written that analyse not-realtime database efficiency instead of using simulations.

In article [YDL93] both two phase locking and pure optimistic concurrency control are analysed using Poisson processes. This paper presented an analytical approximation of the average transaction length, given that the transactions arrive at the scheduler as a Poisson process. Unfortunately the analysis of the two schedulers is mixed, which muddles the article. This distracts from some important assumptions that where made to arrive at the result.

To be able to say anything about the probability that a transaction will finish before its deadline, the average execution time is insufficient. The analysis of the complete probability distribution instead of the average execution time is in general much more involved.

## 9.2 Comparison problems

An analysis of a scheduler should result in a set of success probabilities for an arbitrary transaction under a given workload. This probability will depend also on the transaction length and the size of the database. Even if a distribution

can be specified, it is not easy to compare the efficiency of different real-time schedulers. This is because the efficiency of a scheduler depends on a number of parameters:

1. **Centralised versus distributed environment.** The communication delay introduces problems that are very specific for distributed systems. At the same time, distributed systems offer more computation power. It is clear that distributed systems are unsurpassed if availability is the criterion.

2. **Read-only queries versus updates.** More efficient scheduling is possible if read-only queries are treated as a special type of transaction. Dependent on the application of the database (mostly the percentage of transactions that are read-only transactions) this optimization will be more or less useful.

3. **Real-time scheduling versus normal scheduling.** The performance criteria for real-time databases and normal databases differ. A comparison between a real-time database and a normal database is therefore complex, but it might be useful to analyse the behaviour of a real-time schedule in a non-realtime environment and vice versa.

4. **Priority based or not.** A real-time scheduler can offer a priority mechanism to the users, to give them some influence over the behaviour of the scheduler under a high system load. Of course, this affects the efficiency of the scheduler.

5. **Conflict-rate.** Schedulers behave differently under different system loads. A scheduler can be very efficient under a low system load, but lose performance as soon as the system load increases. Another scheduler can have a rather constant performance, not perfect under relatively low system loads, but handling well under high loads.

6. **Amount of possible deadlocks.** Some schedulers do not prevent the possibility of deadlock. While this permits them to execute more efficiently, deadlocks have to be detected and resolved. Dependent on the nature of the transaction system, deadlocks can be allowed to exist for some time before they are resolved. Especially in a distributed system this reduces the message cost that is associated with deadlock detection.

7. **Variance of transaction lengths.** Several schedulers perform well as long as all transactions are of the same execution length, while degrading when lengths of transactions vary. For instance, pure optimistic concurrency control can lifelock long transactions if a steady stream of short transactions enters the system. Two phase locking would not suffer from this problem.

8. **The level of reliability and availability that is required.** Reliability and availability of the database are desirable properties, but they come at a cost. A distributed scheduler that implements no global commit protocol is unreliable in case of site failures, but is very efficient. A distributed scheduler that uses the two phase commit is less efficient but reliable, but suffers from blocking, thus reducing availability. A scheduler that uses the three phase commit protocol is both reliable and does not suffer from blocking, but the three phase commit protocol introduces more overhead than the other two approaches.

9. **Required advance knowledge about transactions.** Schedulers that rely on access invariance[1] can overcome the problems of lifelock and deadlock easily. Consequently a more constant response time can be guaranteed. The rate-monotonic scheduler is a perfect example. This scheduler knows that all transactions are periodic, with deadlines equal to the beginning of the next period. All transactions can be preempted and continued later. The last assumption distinguishes the allowed transactions of the rate monotonic scheduler from transactions that are normally allowed by databases.

## 9.3 Conclusion

Very few articles deal with the efficiency of transaction schedulers in an analytical way. In the field of real-time systems, where transactions lose their value once their deadline has passed, guarantees about transaction execution times are even more important than in normal databases.

In normal databases, the throughput of the system is of primary concern. Such throughput can be easily measured in a testing environment. In real-time systems, the execution time of each individual transaction is important and more elaborate testing techniques are necessary. At the same time, the analysis of the transaction execution time becomes more involved, as the average execution time no longer suffices.

It will probably be hard to give sharp analytical results, as the problem has remained almost without results for so many years. Research should start with analysing simple schedulers with certain restrictions on the transaction types and frequencies. Nevertheless, the field of real-time schedulers does need a fundamental basis, that cannot be completely provided by test-results.

[1] The data items needed by a transaction are known in advance.

# Chapter 10

# Research issues

After presenting this overview of the field of real-time distributed databases it is time for some reflection. Although a lot of good results are already available in the separate subfields it is not instantly clear that we are now able to build the optimal real-time database with these mechanisms. In this chapter the overview is completed by pointing out the areas where further research is still needed. This will be contrasted by a short summary of results that are already known.

## 10.1 From user-interface to implementation

The real-time distributed database stores information that corresponds to the real world and offers various services to its users. Instead of restricting the notion of users to humans, users can range from other computers to air passing a pressure valve. This wide range of users has no knowledge of the database structure and high level services have to be provided.

These services are typically provided by transactions that have been programmed in advance. Transaction programmers deal with input/output devices and prefer to represent actions in an abstract language, independent from the underlying implementation of real-time databases. Algebraic or relational languages exist that allow arbitrary complex database transactions (for example SQL). Classic languages do not deal with real-time aspects and languages that do take real-time into account are just beginning to emerge.

The translation from algebraic (or relational) operators on an abstract representation of the database to actual distributed transactions is the domain of transaction managers. Different translations of an algebraic expression can differ exponentially in execution time and message costs of the resulting transactions (in a distributed environment). Finding the optimal transaction corresponding to an algebraic expression is NP-hard. Several heuristic Transaction Managers have been developed that offer good approximations.

## 10.2 Transaction scheduling & correctness

To increase the efficiency of transactions it is beneficiary to execute transactions in parallel. However, the database consistency is defined only between transactions. During a transaction, a temporary database inconsistency is allowed to enable efficient execution. If transactions are allowed to execute concurrently, transactions might read data that are temporally inconsistent and act as if the data are consistent. Permanent inconsistencies can occur. To determine whether two transactions can execute concurrently the database system provides a transaction scheduler.

The transaction scheduler tries to maximize the amount of concurrency (executing transactions in parallel) while it preserves the database consistency. Again, finding the optimal schedule is NP-complete. Existing transaction schedulers preserve consistency at the cost of reduced concurrency. As these schedulers are simple approximations of the optimal schedule, they can be improved.

### 10.2.1 Transaction classes

It has been observed that for several classes of transactions the scheduling problem is not NP-complete at all. A taxonomy of transaction classes that have nice properties that allow the scheduler to generate optimal schedules efficiently can be very useful but does not (completely) exist. A well-known class consists of read-only transactions. All read-only transactions can execute concurrently.

### 10.2.2 Periodic transactions

A lot of scheduling research has gone into the the scheduling of transactions with hard-realtime constraints. These transactions are not allowed to fail, they have to run to completion within their execution interval or otherwise the entire schedule is incorrect. A lot of these schedules were constructed at pre-runtime. Therefore the complete set of transactions that was to be scheduled was known in advance. Often the scheduled transactions are periodic, i.e. a transaction is at regular intervals or sporadic, i.e. a transaction is run at regular intervals, but may skip some of these runs.

The periodic nature of transactions can probably be exploited as well in soft real-time scheduling systems. In soft real-time, the number of successful transactions is optimized. It is permitted that some transactions fail to meet their deadlines, as long as it is a small percentage of the total number of transactions. An optimal soft-realtime schedule can always be found, opposed to a hard real-time schedule that may not exist. An optimal real-time schedule of periodic transactions should probably guard against life-lock.

### 10.2.3 Allowing inconsistencies

Other research tries to increase the amount of concurrency allowed at the cost of introducing inconsistency in the database. Such a scheduler can be useful if the amount of inconsistency introduced is somehow bounded. Especially in real-time databases it cannot be avoided that inconsistency in data gathered from the real world occurs. Therefore this seems a natural way to increase concurrency.

## 10.3 Real-time transaction scheduling

In a real-time database, transactions are only allowed to execute within certain time intervals. A scheduler that not only preserves data consistency but also ensures that all transactions are executed in their interval has to be provided. The problem of generating a schedule that executes all transactions within their intervals in an environment with limited resources is NP-complete.

Ordinary schedulers can be slightly modified to incorporate deadlines. Unfortunately most schedulers behave badly under high system loads. An ordinary scheduler will try to execute all transactions. If the system load is high, this will mean that all transactions run to completion, but also that almost all transactions will have missed their deadlines. A real-time scheduler must decide to abort transactions that miss their deadlines in order to complete a (constant) number of transactions in time.

A lot of research has been directed at hard real-time scheduling. In hard real-time, not even a single transaction is allowed to fail. This is a very restrictive constraint, that often cannot be realised. Also a common assumption is that transactions can be pre-empted, that is put on hold and resumed later. This is quite contrary to the correctness constraints of databases, where the database consistency is temporarily disturbed during a transaction. It therefore seems more logical to use the existing correctness preserving algorithms as a starting point instead of using the real-time scheduling algorithms as a starting point for real-time database schedulers.

### 10.3.1 How many resources are required?

Surprisingly little analytical studies have been published about transaction execution times, and I know no analytical results in real-time scheduling. The performance of a soft-realtime transaction scheduling mechanism can be defined by the probability that a transaction executes within its time interval. Several simulation studies have been made to determine these probabilities, but no analytical analysis of transaction schedulers are available. An analytical analysis would present us with a set of hardware requirements, as well as clear assumptions on the behaviour of the users of the real-time database.

Associated with this probability it is interesting to specify the relation between the performance of the real-time database and the hardware that supports the database. If for a given real-time scheduler this relation can be specified an estimate of required resources can be given analytically for a given problem.

### 10.3.2 Disk-based systems

As mentioned in the previous chapter, a lot of timing problems arise when the underlying operating system optimizes disk access by buffering, or when virtual memory is implemented. Periodic cache-flushes could be scheduled when no transactions are in progress. This would prevent unreliable reaction times of the operating system, at the cost of an extra transaction (the flushing of the cache) that has to be scheduled.

### 10.3.3 Combining correctness and timeliness

Scheduling transactions in such a way that consistency of the database is preserved while offering optimal concurrency and scheduling transactions within their execution intervals are related. As these problems are tough to solve on their own, they are often treated separately. In real-time databases, a scheduler has to be provided that takes both requirements, correctness and timeliness into account. The current approach is to use existing, correctness preserving schedulers and prove that under a restricted workload sufficient transactions meet their deadlines.

## 10.4 Distributed transactions

The distribution of a database can increase the availability, reliability and capacity of the database. This does come at a cost. First of all, communication between the different sites of the database becomes an important factor of time-delay. Secondly, scheduling of distributed transactions becomes more involved because of distributed deadlocks, global correctness and routing problems.

### 10.4.1 Communication delay

As mentioned, communication delay is an important factor in distributed databases. Therefore algorithms that were fairly trivial in a centralised database have to be optimized in the distributed environment. A way to reduce message costs is to replicate data over the different sites, but this introduces new consistency problems. Several solutions have been proposed and exist, but the field is still under development.

### 10.4.2 Query optimization

The transaction manager that optimizes transactions to reduce the number of execution steps of a transaction has to be adapted. The size and the number of messages between sites is more important. A simple optimization is to compute selections on tables at the local sites.

### 10.4.3 Fragmentation of the database

Important design choices are made when the database is distributed over the available sites. To what extent should the information in the database be replicated? What is a good fragmentation of the information in the database? The answer to these choices depends on the topology and capacity of the sites that are cooperating to form the distributed database.

The problem to fragment a database in such a way that with a uniform access distribution the workload is optimally divided over the sites is NP-complete. It is therefore interesting to investigate what extra knowledge about the access behaviour of the database is needed to come up with good distributions of the database. An interesting starting points is for example knowledge about the access points of data items. If a data item is only accessed by users from one or two sites, it is natural to store the requested item on at least one of these sites.

The relation between fragmentation and replication can be studied. Of course, access times are optimal in the fully replicated case. However, if a lot of updates take place in the database, the replication of data introduces extra overhead to maintain correctness instead of speeding up transactions. To what extent a database should be replicated to provide the optimal access behaviour is an open question.

### 10.4.4 Synchronising sites

The scheduler can receive new transactions at more than one site. Existing (centralised) schedulers that make use of unique time-stamps have to ensure that time-stamps issued at different sites are unique and somehow related (thus time-stamps issued at roughly the same time should have roughly the same value).

In general, important execution steps of transactions should be synchronised over all sites. This means that algorithms have to make sure that all sites have finished such an important step before they proceed to the next step of the transaction. Examples are synchronisation between the locking and unlocking phase in two phase locking, and the commitment of transactions. Unpredictable results will follow if a transaction commits on one site and aborts at another site.

Adapted algorithms for distributed databases have been provided for most existing (centralised) database mechanisms. It is hard to provide algorithms that are efficient, reliable and offer graceful degradation of the database in case of

failures, but there is already a large library of generic algorithms that provide good communication protocols between sites in a distributed network.

# Bibliography


[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS vol. 8 pp 186,213*, 1983.

[GS85] N. Goodman and D. Shasha. Semantically-based concurrency control for search structures. *Proceedings of the ACM pag 8-19*, 1985.

[IK94] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relation joins. *ACM transactions on database systems pp 482-502*, 1994.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM computing survey pp 111-152*, 1984.

[KM93] Tei-Wei Kuo and Aloysius Mok. Ssp: a semantics-based protocol for real-time data access. *Proceedings 14th real-time systems symposium*, 1993.

[Kna87] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys 19(4) p. 303*, 1987.

[KR81] H.T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM 0362-5915/81/0600-0213*, 1981.

[KR92] Mohan Kamath and Krithi Ramamritham. Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. Technical report, University of Massachusetts, 1992.

[ÖzsuV91] M. Özsu and P. Valduriez. *Principles of distributed database systems.* Prentice-Hall International Editions, 1991.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the association for computing machinery*, Oktober1979.

[SY82] M. Saccoo and S. Yao. Query optimization in distributed database systems. *Advances in computers, volume 21 pp 225-273*, 1982.

[Vid85] K. Vidiasankar. A simple characterization of database serializability. *5th conf. on foundations of software technology and theoretical computer sciency, LNCS 206*, 1985.

[Vid91] K. Vidiasankar. Unified theory of database serializability. *Fundamenta Informaticae XIV*, 1991.

[Wei87] William E. Weihl. Distributed version management for read-only actions. *IEEE transactions on software engineering No. 1*, 1987.

[YA88] Shyan-Ming Yuan and Ashok. K. Agrawala. A class of optimal decentralized commit protocols. *IEEE?*, 1988.

[YDL93] P. Yu, D. Dias, and S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM p. 831-872*, 1993.

# Index