THE NECESSITY FOR CHECK-ON-COMMIT IN THE PROTECTION OF THE INTEGRITY OF A DATABASE

Saskia C. van der Made- Potuijt,
Department of Computer-Science
Erasmus University Rotterdam.

Preface

The value of the data in a database can be increased by specifying and enforcing integrity constraints on the data.
There are several methods for the implementation of integrity constraints. The used method in a particular situation depends on the features offered by the used Database Management System (DBMS). However, not all these methods offer the same certainty that the constraints are really enforced. There are also important differences in the methods regarding programming effort and maintainability.
In this article integrity constraints and their implementation will have our attention. It will be explained why it is necessary for the protection of the integrity of the data in the database to be able to check integrity on the commit time of a transaction.

The report is partially based on an article of the author on transactions [9] and the contribution of the author to an article on constraints [8].
The different methods used by the vendors of DBMS's for implementing constraints have been investigated by a group of independent dutch database experts (IDT-Holland, which stands for Independent Database Team Holland). The author is the chairman of IDT-Holland. This article offers the theoretical framework for the understanding of the results of the investigation.
Certain parts of this report are of tutorial nature.

1984 CR catagories: D.3.2, H.1.1, H.2.0, H.2.4, H.2.7, K.6.2.
1980 Mathematics subject classification (1985 revision):
68N15, 68P15,
Keywords & Phrases: Value of information, Integrity, Benchmarks, Datadictionary/Directory, Transaction Processing.

Note: A excerpt of this article will be submitted for publication elsewhere.

Introduction.

In the first part of this article it will be explained what integrity is, what the position is of integrity constraints in the three-schema architecture and how integrity constraints can be implemented.
In the second part a subdivision of integrity-constraints will be given for the relational environment.
The logical unit of integrity, the transaction, will have our attention in the third part of this report. In this part the concept check-on-commit and the need for it in the protection of the integrity of a database will be explained.
IDT-Holland has developed a functional benchmark that has been used to evaluate the functionality of DBMS. In this benchmark the importance of the role of a DBMS in the protection of the integrity of the database is emphasized. In the last part the IDT-benchmark will be explained, how it was applicated and the results of it.

I. Integrity.

A. The reliability of data.

Decisions in organizations are often based on information retrieved from an information system. It is important to support a high quality of this information. Beside completeness, relevance and whether the information can be retrieved in time, it is important that the information system is reliable. There are three aspects to the reliability of an information-system: its behaviour in situations where more users or programs access the same data, the capabilities to recover from a crash and the reliability of the data itself. In the reliability of data three levels can be distinguished: Accuracy, integrity and consistency.

B. Integrity and security.

Security is not an aspect of the quality of the information in the information system, but of the information system itself. An information system must provide possibilities for protecting the system against persons who are not authorized to access the system or a part of it. Although the purpose of security and integrity is quite different, protection against unauthorized users or protection against certain data, there are some similarities:

- Security and integrity can both be specified in terms of constraints.

- The DBMS is responsible for the user interaction and must ensure the enforcement of the constraints.

- There are several methods for implementing the constraints, including specification in the DD/D and not all the methods offer the same certainty that the constraints cannot be circumvented

In the rest of the report no attention will be given to security constraints. If the word constraint is used without any indication to a type, an integrity constraint is meant.

C. Accuracy, integrity and consistency.

The state of a database system is determined by records and devices that have changeable values.
There are certain constraints referring to these values, the static integrity constraints. If none of these integrity constraints are violated the database system is in a correct system state. The dynamic constraints specify which transformations of the database states are permissable.

The correctness of a system state is strongly related to consistency. In fact correctness is a stronger demand on a database system than consistency. A database system is in a consistent state if there are no internal contradictions. If a system is in an correct state however, it is consistent and all the integrity constraints are fulfilled.

Accuracy is the highest level of reliability of data that can be achieved. A system state is accurate if it is represents the reality.
In an accurate system state no integrity constraints can be violated, therefore it will always be correct. The accuracy of the information is ultimately the responsibility of the end-user: no system can check on the actual age of a person. Only the person that inputs the data may have means to check it. Organizational measures are necessary to support the accuracy of the data.

D. The position of integrity constraints in the three-schema architecture.

On each of the three levels in the schema there is a datamodel, a description of the operations and a specification of the integrity constraints. Integrity constraints have to be formulated that indicate which data are acceptable. If constraints can be enforced to data they can serve as a ’filter’ to keep out data that do not fulfill these integrity constraints.
Another word for constraints is businessrules: all types of rules that are in force in an organization can be specified as a constraint on the data in the information system of the organization.

Building an information system involves the integration of the schema’s of the external level to the conceptual schema. This implies also the integration of the constraints: often this will result in the sharpest constraints of the external level. In the conceptual schema no implementation considerations play a role: it is a pure logic level.
The conceptual scheme has to be mapped to the internal schema. In this step decisions concerning the implementations have to be taken: will a DBMS be used, if so what type and what product. Attention has to be given to the possibilities of the candidate DBMS’s for the support of the enforcement of constraints. In the classical datamodels for databases, the hierarchical, network and relational model, the structure of the data can be specified in terms of entities, attributes and relations. However not all the constraints can be controlled by

the structure of the data. If a DBMS does not support adequate support for the specification of the other constraints it is not possible to guarantee these constraints will be enforced: a very dangerous situation for the quality of the information. The constraints that cannot be implemented in the DBMS have to be implemented by methods that are less reliable like applications.

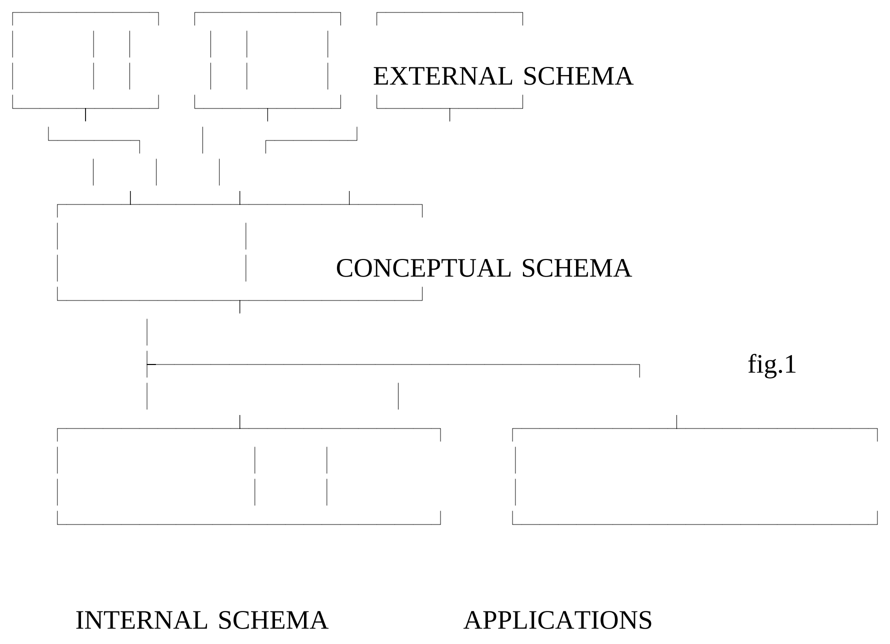


fig.1

E. The methods used to implement the constraints.

The following 4 methods for the specification and and enforcement of constraints can be used. The methods are arranged in order of desirability :

1. By specification in the DD/D.

This is the best method because of the following reasons :

- The DBMS is forced to validate the constraints in any case of database alteration. Therefore there is certainty that the constraints will be verified.
- Other database definitions are alse registered in the DD/D. Specification in the DD/D means central accessibility of the constraints in a place, where the user would expect to find such definitions. They can be accessed with standard tools generally used for queries on operational data in the database and therefore familiar to the user.
- Definition in the DD/D implies a standardised and formalised description of the constraints. This improves the speed of such specification. Suppliers who have this DD/D-facility can specify a complex case in considerably less time than other suppliers can.

2. By specification using a trigger.

A trigger can be regarded as a procedure which is activated through an exit of the DD/D, which the DBMS is forced to use. This method gives the same certainty of verification, but has a few disadvantages

- The specification is not directly in the DD/D and therefore not as easily approachable by the user.
- The specification will usually be less formalised and therefore not as easy to understand.
- Moreover in this case the parameters that determine the data-environment will have to be specified, which makes the process more complex and more time-consuming.

3. By specification via a view with check-option.

Because a view is a user-oriented and not a database-oriented mechanism it is not easy to assure that an application or user cannot bypass it for alterations of the database. This implies that the integrity of the data cannot be guaranteed. To overcome this disadvantage organizational measures are required. The view with the constraints must be used by every application or user, the base-table should not be used anymore for updates. Using this method for the specification of constraints often requires a layer of views for the various constraints. The disadvantages of the previous solution also apply here.

4. By specification of the constraint in a program or module.

A guarantee of dataconsistency can in this case, just like the solution with view's, only be obtained via procedural controls. Here too, all disadvantages of the former solutions apply. Obviously the least desirable of the four.

<u>II. Constraints</u>

Constraints can be divided into static and dynamic constraints.
Static constraints specify to which rules the contents of a database should obey at any time, while dynamic constraints specify which transitions in the state of the database are allowed.

<u>A. Static constraints.</u>

Static constraints can be subdivided into the next five groups:

1. <u>Attribute constraints</u>

Examples:

- The age of an employee is between 15 and 66.
  (a range- constraint, an interval)
- The value of this attribute is not allowed to be null.
  (an exclusion)
- Only foreign currencies of this list can be exchanged by this bank
  (a random set of values, a list)

To understand attribute constraints well, it is necessary to know the concept of a domain. A domain can be defined as the set of possible values of an attribute. A domain description consists of:

1. A type-specification. A type can be:

   a. A base type (e.g. text, boolean, integer, date, time, etc.) or a
   b. Structured type (e.g. record, array of.., set of .., etc.)

2. A domain constraint. Three classes of domain constraints can be distinguished:

   a. Interval (e.g. 1000-9500 , n.b. Numeric(4) describes type as well as domain-constraint)
   b. Random set of values
      - direct, e.g. {1,2,7,9,10,11}, a list
      - indirect through calculations,
        e.g. all even numbers or all numbers divisible by 11.
   c. Exclusion (e.g. <attribute> not null.)

An attribute constraint will always refer to a domain. In fact it is a further restriction of a domain constraint. Suppose a domain age is defined as (0-150), then the age-constraint of the first example is a subset of this.

b. Tuple or inter-attribute constraints

This describes the mutual dependency of attribute-values within the same tuple-occurrence.

Examples

- If age < 18 then the value of the attribute ’driver-licence’ is null.
- If age + number of years in the company > 70 then ...

c. Table or intrarelation constraints

Four types of table constraints can be distinguished:

1. Unicity,
   e.g. unique on client-no.

2. Referential integrity within one table,
   e.g. attribute spouse in table person.

3. General table constraints,
   like functional and multi-valued dependencies,
   e.g. all clients living in the same city have the same area-code.

4. Specific table constraints.
   Examples:
   - The number of patients of one doctor should not exceed 2500.
   - The average salary of the sales-department should not be more than 10% higher than the average salary of the production-department.

d. Database or interrelational constraints

These can be divided into two groups:

1. Referential integrity rules between tuples in different tables
   - normal, e.g. an orderline should always refer to an existing order.
   - cyclic, e.g. an account should always refer to a client and for every client there is an account.

2. Other database constraints, e.g. functional dependencies.
   Example: A patient can only have a GP that lives in the same municipality as the patient.

e. Update constraints

In order to maintain referential integrity certain rules have to be specified that determine what should happen to tuples in related tables. Update constraints have to be specified in situations where referential integrity is involved and for some other constraints.
There are delete, update and insert rules.

1. Delete rule

This rule must be specified for every table that contains a foreign key to another relation and for situations where other database-constraints are involved. It determines what should happen to the foreign-key tuples if the corresponding primary-key tuple is deleted.

The delete and the update rule will be illustrated with the example of a doctor with its patients. A patient has a few atttributes and is assigned to one doctor. This association between a patient and his or her doctor is represented by a foreign key in the relation that describes the patients to the table with the description of the doctors.

Three standard options are desirable for the delete rule:

- Restrict

  A primary-key tuple can only be deleted if there are no referring foreign-key tuples.

  Example 1:
  A doctor can only stop his practice if there are no patients assigned to him anymore.

- Cascade

  If a primary-key tuple is deleted, the corresponding foreign-key tuples are deleted as well.

  Example 2:
  If a doctor stops his practice the data of his patients are deleted also.

- Nullifies

  If a primary-key tuple is deleted, the corresponding foreign-keys are made null.

  Example 3:
  If a doctor stops his practice the data of his patients are kept, but for some time they are not assigned to any doctor.

Not all the desirable possibilities are captured in the three options described. An alternative delete rule in this situation is.

Example 4:
The deletion of a doctor will be refused if there are some patients referring to him as being his patients. A list of these patients with their telephone numbers is generate. These patients will be called to determine to which other doctor they want to be assigned. The deletion of the doctor can only take place if all the patients are assigned to their new doctor.

In situations with a general database constraint also a delete rule has to be specified. We will illustrate this with the second example of the previous paragraph.

Example 5:
A patient can only have a doctor that lives in the same municipality.
Suppose the alternative delete-rule of example 4 is specified. Above this rule it has to be checked whether the new doctor lives in the same municipality. This can be seen as a more complicated delete rule or as the delete rule of example 4 plus an extra database constraint check.

2. Update rule

This rule must, just like the delete rule, be specified for every table that contains a foreign key to another relation. It determines what should happen to the foreign-key tuples if the corresponding primary-key tuple is updated.
The same three standard options as for the delete rule should be possible for the update rule plus the opportunity to specify an alternative rule.

- Restrict

  A primary-key tuple can only be changed if there are no referring foreign-key tuples.

Example 1:
A doctor can only sell his practice if there are no to him assigned patients at the moment.

- <u>Cascade</u>
  If a primary-key tuple is updated, the corresponding foreign-key tuples are updated accordingly.

  Example 2:
  If a doctor sells his practice, his patients are automatically transferred to the new doctor.

- <u>Nullifies</u>
  If a primary-key tuple is updated, the corresponding foreign-keys are made null.

  Example 3:
  If a doctor sells his practice, his patients are temporarily not assigned to any doctor.

Because updating a primary key has to be avoided, care must be taken in choosing the cascade and nullifies-rule. Specifying the update-rule restricted is one way to avoid problems while updating a primary key.
Just like with the delete-rule not all the desirable possibilities are captured in the three options described. An alternative rule in this situation is:

- Example 4:
  An update will be refused until it is clear what the related individual patients want: to the new doctor or to an existing one.

We end with an example of an update-rule in case not only a referential-integrity is involved but also a general database constraint. Example 2 of the database constraints will be used to illustrate this type of update-rule: A patient can only have a doctor that lives in the same municipality as the patient.

- Example 5:
  Suppose a patient moves to another municipality. A new doctor has to be assigned to this patient that lives in the same municipality.
  In this case the update of a field that is not a foreign key, but that is involved in a database constraint, makes a check on referential-integrity necessary.

3. Insert rule

Referential integrity implies certain insert rules: it makes it possible to only insert tuples in a relation A with a foreign key to relation B if there is a matching primary key in relation B. It often determines the order for inserting related tuples.

Example 1:

There is a referential integrity relation between the Room and Department tables.

<u>Room table</u>	<u>Department table</u>
Room.Nr (key)	Dept.Nr (key)
Floorspace	Dept.Name
Status	Manager.Nr
Type	
Dept.Nr (for. key)	

fig.2

Every room belongs to one department. This is represented by the foreign key Dept.Nr as an attribute of room that refers to a department.

Suppose a new wing is added to the hospital with some rooms belonging to a new department. It is not allowed to add the rooms first including the number of the department, because then the rooms would have foreign keys that do not refer to any existing department.
The department-tuple should be inserted first, after that the room-tuples, no integrity constraints have to be violated.
If null is allowed as the value for Dept.Nr in the table Room, it is also possible to add the room-tuples first, with null as the Dept.Nr, after that the tuple in the table Department and finally to update the null-values for Dept.Nr in the table Room to the correct value.

Referential integrity implies a certain insert-constraint. There are however other insert constraints.

Example 2 , a insert-attribute constraint:

- the amount of money deposit into an account should not be less than *f* 1000,= .

B. Dynamic constraints.

The dynamic or transition constraints describe which transitions are possible for the values in the database. This can refer to attributes, tuples, table and the database as a whole.

Example 1, a dynamic attribute constraint:

In accordance with this diagram, can a second hand car can not become a new car etc..

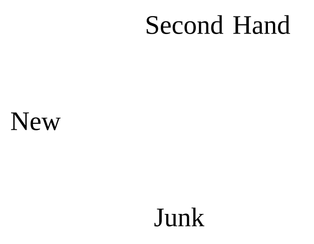


fig.3

Example 2, a dynamic database constraint:

A health insurance company has contracts with several companies for the insurances of their employees. There are some differences between the contracts with these companies. For instance, employees that work for company A are not allowed to decrease their own-risk factor, but employees that work for company B are allowed to do so.

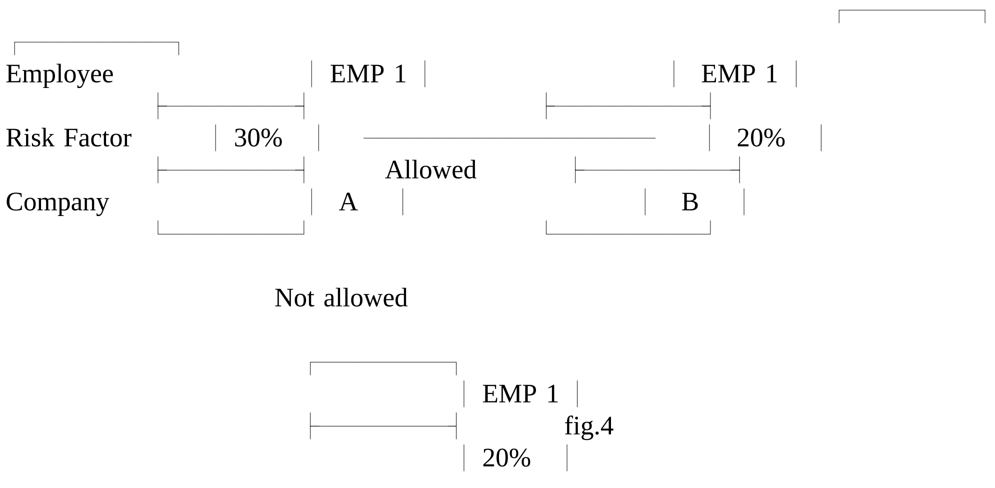
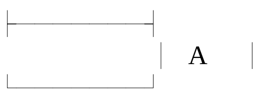


fig.4



Again, the dynamic constraints determine which transitions are allowed. In this case not only the update in the table of the employees has to be checked, but also the table of the companies where is registered what type of insurance the company has.
Notice that the third transition, employee 1 working for company B having risk factor 20% to employee 1 working for company A risk factor 20 is allowed.

The difference between the dynamic and the update-constraints is, that the update constraint specifies something of the new database-state. If the new database-state is not correct, some update rule was violated.

e.g. If a doctor stops his practice, there should not be any referring foreign key to this doctor any more. If so, the delete-rule has been violated.

If a dynamic constraint is specified, the new system state does not determine whether this constraint has been violated.

Example 1 of the dynamic constraint:
A car can be registered as being second hand. Only if it was previous a junk car, this registration being a second hand car cannot be correct.

Example 2 of the dynamic constraints:
It is not impossible for employee 1 to be registered as having a risk factor of 20% and working for company A. If he worked previously for A and his factor was 30% however this is not permissable.

III. The transaction concept.

In the database environment transactions are usually related to the problems of concurrency-control and recovery. Not much attention is given to the fact that this is a consequence of the fundamental nature of a transaction as a logical unit of integrity. In this third part of the report we will describe what a transaction is and pay attention to the three roles of transactions: as logical unit of recovery, concurrency and integrity

A. What is a transaction?

As explained in 1.3 the state of a database system is determined by records and devices that have changeable values.
There are certain constraints referring to these values, the static integrity constraints. If none of these integrity constraints are violated the database system is in a correct system state. The dynamic constraints specify which transformations of the database states are permissable.
During a transaction certain integrity-constraints can be violated, but before and after a transaction the database should be in a correct system state.

Definition of a transaction:

> A transaction is a permissable transformation of one correct system state of the database system to another correct system state.

2 Examples of system-states:

The system states describe two cars: we only look at the attributes
number, colour and status. The primary key is number and there are two integrity-constraints:

- The status of a car can only be [New, Second Hand , Junk],
- The permissable colors are [Green, Blue, Red]

System state 1

GY-67	FT-83	Number
Green	Red	Colour
New	Second Hand	Status

System state 2

GY-67	FT-83	
Yellow	Red	
New	Junk	

fig.5

System state 1 is correct: all the integrity-constraints are fulfilled.
The second systemstate however is not correct since yellow is not a permissable colour.

The next diagram shows how a dynamic constraint determines what transformations in system state are permissable and which are not.
The dynamic constraint of figure 4. determines that transformation 1 is allowed, but transformation 2 is not: this violates the constraint that a second hand car can never again be registered again as being new.

GY-67	

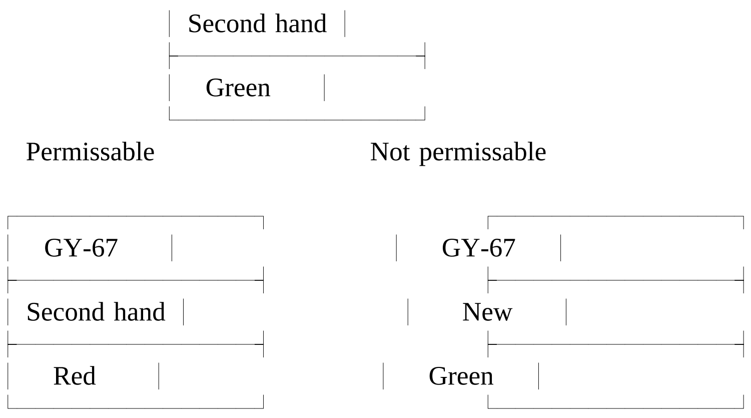


fig.6

Transactions have three primitive operations: BEGIN, the start of a transaction, ROLL_BACK, which indicates that something has gone wrong so that the transaction will not be able to terminate in a correct way and COMMIT: the transaction has terminated successfully.

Transactions have three important features. They ensure:

<u>Consistency</u>:	The transaction ensures that all the consistency constraints will be fulfilled in case the transaction is committed is rolled-back.
<u>Atomicity</u>:	Either all actions that comprise the transaction are done and the transaction is said to commit or none of the effects of the transaction survive and the transaction is rolled-back.
<u>Durability</u>:	Once a transaction is committed, it cannot be rolled-back. Its effects can only be undone by running compensating transactions.

As a result of this fundamental property of a transaction as the unit of integrity, transactions are also the unit of recovery and concurrency.

The recovery system should be able to restore the database to a correct state after some failure. Because it is certain that at the start and at the commit of a transaction the database is correct, the recovery manager must take care that it does not bring the state of the database back to a situation in which some transaction was not completed. So a transaction is the logical unit for recovery.

It is undesirable that a user or a program can access data that might not be correct. Because during a transaction the data that are used by that transaction may not be correct, other users should not be allowed to approach these data unrestrictedly. This implies that a transaction is also the logical unit of concurrency.
In the next three paragraphs the three aspects of datamanagement in which transaction play an important role, recovery, concurrency and integrity, will be discussed.

B. <u>Transaction as the logical unit of recovery.</u>

<u>1. The parts of the recovery-system.</u>

There are four features a DBMS should supply in order to support an efficient recovery process:

1. Procedures to enable an easy execution of backups
2. The checkpoint mechanism
3. A logging-facility
4. A recovery-protocol.

There are several ways in which one can distinguish the different types of failure that can occur in a database-environment. Here we present a distinction into three types:

- <u>Media failures</u>, for example a disk-crash.

In general media-failures can be solved using the backup and the log. With the backup a previous correct state will be reestablished. After this the transactions that were registered in the log as being committed are executed again. This will result in the last correct state before the failure.

- Logical errors, such as bad input format or a bug in an application-program.
  Debugging and correcting is necessary to enable the continuation of the program.

- System errors, like a power failure where the content of the main memory is lost.
  The recovery-manager is responsible for a return to a previous correct state. The log will be used to decide which unsuccessful transactions that may have lost their values have to be redone or undone.

2. The recovery-procedure in case of a system-error.

We will have a closer look at this last type of failure in which the recovery manager plays an important role.

There are various possible recovery protocols that can be implemented in a recovery-manager, depending on the properties of the parts involved in the protocol.

An important part of the recovery-mechanism is the checkpoint. There are several definitions and implementations of checkpoints used by vendors.

- One definition often used implies that a checkpoint-record is placed in the log-buffer and the contents of the log-buffers in which also before and after images of the database records are collected are written to disk, but that the databasebuffers, containing the actual database-records in which some alterations have taken place, are not forced to disk. Arrow 1
- In another definition taking a checkpoint also implies the flushing of the database-buffers to disk. Arrow 1+2

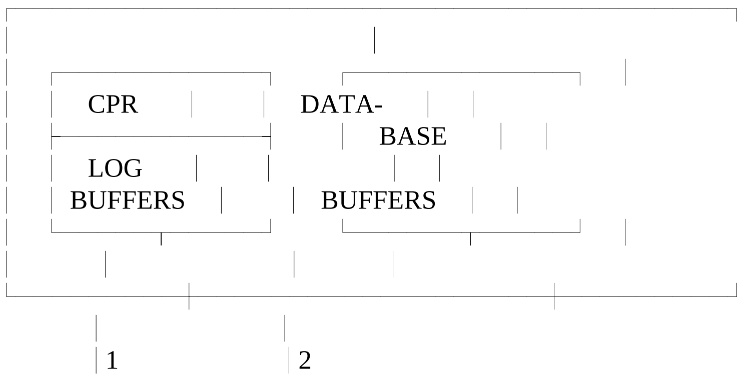


fig 7.

Another influence on the recovery protocol is the adoption of the deferred-update protocol or the immediate-update protocol.

- Using the deferred-update protocol means that no alterations are made to the database until the transaction reaches its commit. If the transaction is aborted for some reason, it is not necessary to undo alterations because they have never been made to the original file. The log does not have to contain a before-image of the relevant data, because it will never have to restore these.
- The immediate-update protocol makes it necessary that the before-images of the data are kept, because in the process of the transaction certain updates to the database may have been written already to disk.

The third influence on the recovery-mechanism is the frequency used to write the logbuffers to disk.

The logbuffers contain the information to enable the system to recover from a crash. It is therefore necessary that these logbuffers are frequently transferred to disk in order to make sure the system can return to a very recent correct state.

Figure 8 illustrates the process of recovery, using

- the definition of checkpoint in which the db-buffers are made empty and
- the immediate update protocol.

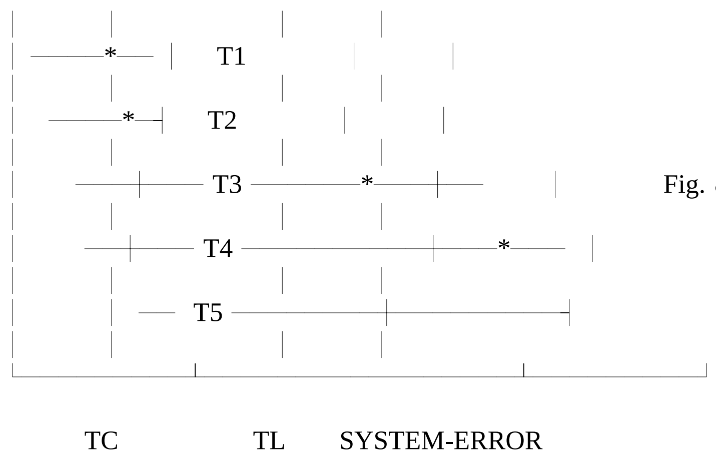


Fig. 8

The asterisk indicates the commit of a transaction, after this the database buffers are not emptied yet; that moment is indicated by the end of the line.

- TC is the moment the checkpoint is taken.
- TL is the moment the log-buffers are transferred to harddisk.
- T1 is committed and all its alterations have been transferred to the harddisk, so T1 is not involved in the recovery-process.
- The same applies to T2, although at the moment of the checkpoint the alterations were not completely written to harddisk. This is done at TC so T2 is not involved in the recovery-process.

Transactions that are registered in the log as being active on the moment the checkpoint is taken have to be rolled back in order to establish a previous correct state. If it is certain however that one of these transactions have reached its commit it is assured that this transaction did not cause the system error due to some logical error, so that transaction can be redone.

- Transaction T3 can be redone: it has reached its commit, this is registered in the logfile and this information has survived the system-error.
- T4 has reached its commit, but this commit has not been registered in the log-files on disk before the system-error. This transaction therefore must not be redone, just like the other transactions that have not reached their commit at TL, for example T5.

C. A transaction as the logical unit of concurrency.

1. Introduction

Consider the next two transactions: T1 is the transfer of an amount of one account to another account, T2 displays the sum of the two amounts in the accounts.

	T1	T2
80	READ (A)	

```
    A:=A-50        |
30      WRITE ( A )   |
15      READ (B)      |
    B:=B+50        |
65      WRITE (B)     |
                     |  READ ( A )        30
              |  READ ( B )       65
                     |  DISPLAY ( A + B )        95
```

Schedule 1

If the transactions are executed serially, so first T1 then T2 like shown in schedule 1, or first T2 then T1, transaction T2 will display the situation of a correct system state.

If however the two transactions are interleaved, as shown in the next schedule, transaction T2 will show the situation from a non-correct state.

```
           T1          |        T2
        ------------------------+-------------------------------
                               |
80      READ ( A )    |
        A:=A-50       |
30      WRITE ( A )   |
                     |  READ ( A )                30
              |  READ ( B )                 15
                     |  DISPLAY ( A + B )         45
15      READ (B)      |
        B:=B+50       |
65      WRITE (B)     |
```

Schedule 2

This situation can be avoided by using locks and a locking protocol, or another type of concurrency control method.

2. The locking protocol.

The next schedule shows the situation where locks are used.
S indicates a shared lock: other transactions can read this dataitem, but are not able to write it. X means an exclusive lock is placed, which implies no one else can read or write this record.
WAIT indicates in this case that T2 cannot have a S-lock on A because A is already X-locked, therefore T2 will have to wait until T1 releases its X-lock on A.

```
   T1         |        T2
  -------------------------+---------------------------
            |
 S   READ ( A )     |
A:=A-50        |
 X   WRITE ( A )    |
                    |  READ ( A )           WAIT
                |  READ ( B )
            |  DISPLAY ( A + B )
 S   READ (B)       |
B:=B+50        |
 X   WRITE (B)      |
```

Schedule 3

Transaction T2 must wait with its execution until T1 has released its lock on A. This happens at the moment of commit of T1. It will result in a serial execution of the two transactions: First T1 will reach its commit, the locks are released, then T2 will continue its execution.

As the next schedule shows things do not always go as smoothly as in the previous case. The difference with the prior situation is, that here transaction T2 reads item B before it intends to read A.

```
                T1        |      T2
            ________________|______________________

S       READ ( A )    |
        A:=A-50       |
   X             WRITE ( A )  |
                          | READ ( B )              S
                       | READ ( A )         WAIT
                              | DISPLAY (A + B)
 S      READ (B)      |
          B:=B+50       |
WAIT     WRITE (B)     |
```

Schedule 4

This reading of B by T2 locks B in a shared-mode, making it impossible for transaction T1 to lock B in an exclusive-mode.
Transaction T2 will wait for T1 to release its lock on A which will take place at the commit of T1, but T1 cannot commit because T2 has a lock on an item T1 needs. The result: a deadlock.

Deadlock can occur in situations where locking is used in order to support concurrency. However there are methods of concurrency control that avoid deadlock.

3. Non-Locking Concurrency Control Techniques

We will sketch briefly three methods of concurrency control in which deadlock cannot occur:

If the Timestamp Method is used by a DBMS a unique identifier is created for each transaction.
Furthermore with each database-item X two timestamps are associated: a read-timestamp and a write-timestamp.
The transactions are scheduled on basis of their timestamps.
Whenever a transaction T reads or writes a data-item, the timestamp of T is compared with the read and write- timestamps of that item. If the time-stamp order will be violated by the transaction T, T will be aborted and rolled back. This will happen e.g. if T wants to alter an item that is being read by a transaction that has started later but has not yet completed. Then T will be restarted with a new timestamp.
Deadlocks cannot occur because transactions never are in a waiting state, on the other hand it may occur that transactions may be rolled back although this was strictly speaking not necessary. Another disadvantage of this method is, that a complex transaction can be rolled back in favour of a very simple one.

In the Optimistic Concurrency Control Method no checking is done while the transaction is executed. This reduces the overhead during the transaction.
At the end of the transaction a validation procedure checks whether any part of the transaction has violated the concurrency-rule. This is done using timestamps. If the complete transaction satisfies the validation procedure, the transaction can be committed, otherwise the transaction is rolled-back and restarted at a later time.
Updates in transactions are not applied to the real data during the transaction in order to make an abort quite easy.
Although conflicting situations are solved by roll-backs, with the chance that a complex transaction is rolled-

back in favour of a simple one, in situations where a majority of read-only transactions is to be expected, this concurrency-control method may be advantageous compared to locking methods due to the reduction of overhead.
If the system load is high, conflicting situations may lead to a very low system performance: there is no guarantee that a transaction that has been rolled back will successfully commit the next time. This leads to a trial and error situation.

The Multiversion Concurrency Control Technique uses timestamps to identify the several versions of an item. This method has the advantage that a read-command can always be executed. The disadvantage however is the overhead in managing the different versions of an item. If the serial order of two or more transactions will be violated, one of the transactions will be rolled back. The victim for the roll-back is chosen on basis of a time-stamp, so again there is the chance that a complex transaction is aborted in favour of a simple one.

D. The transaction as the unit of integrity.

The fact that at the BEGIN and the COMMIT of a transaction the database is in a correct state and that a transaction determines which transformations in the database are allowed has implications for the moment at which the constraints have to be checked. First we will discuss the implications for the static constraints.

1. The moment of control for the static constraints

Static integrity constraints that involve just one table like attribute, tuple and table constraints, can be checked at the moment the update takes place. There are also static integrity constraints however that do not involve just one table, but in which more than one table may be involved, like certain other update-rules and referential integrity (this can also involve only one table). In order to be able to guarantee that a database will comply with constraints that involve more than one table, the constraints-checks have sometimes to be done on the two tables at the same time. This can only be achieved using the transaction-concept. This will be illustrated with three examples: the first integrity-problem can be solved without a transaction, the second and the third one really demand a transaction.

Example 1:

The Room and Department tables, the first example in 2.1 (p.8) used to illustrate the insert rule. We saw that the department-tuple should be inserted first, after that the room-tuples.
In this case it is not necessary to use a transaction to add the relevant tuples: if the department is added first, then the rooms no integrity constraints have to be violated.
It assumes a certain knowledge of the user: one order of insert will be rejected, the other order accepted. If however the two update are done as one transaction and the integrity constraints can be checked at the moment of commit the insert-order is irrelevant.

Example 2:

If the number of a department in the example of 2.1 has to be changed, a transaction is necessary: if you update the room-table first, the new department-numbers would not refer to an existing department. On the other hand, if you update the department-relation first, the old department-numbers in the room-table would not be valid. So the update of both tables should be done as one logical unit of work, a transaction, and the check at the moment of commit.

In the next example a transaction is necessary to preserve integrity while inserting related tuples.

Example 3: The Department and Staff Member table.

Department table	Staff member table
Dept.Nr (key)	Id.Nr (key)
Dept.Name	...

Man.Nr	Dept.Nr (for. key)
Assman.Nr (for. key)	...

There are two associations between tuples of the department table and the staff member table: the number of the assistant manager in the department-tuple must refer to an identificationnumber in the staff member table and the department to which a staff member is assigned must reflect an existing department.

These two referential integrity constraints together imply that you cannot insert a department unless the corresponding assistant-manager is already present as a staff member. On the other hand you cannot insert a staff member unless its department-tuple is present.

This "chicken and egg"-problem is illustrated in figure 9.

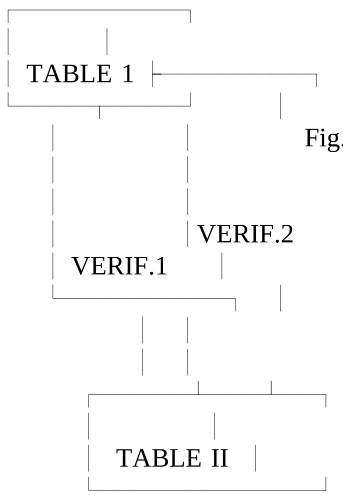


Fig.9

Verification of a tuple in table I requires the existence of a tuple in table II, and this requires the original tuple in table 1.
Insertion of one tuple at the time should be rejected, only insertion of the two related tuples as a whole is acceptable.

This cyclic control problem can be solved by checking the types of constraints that require that a transaction is completed, at the moment of commit. Constraints that can be checked within the transaction should be checked within the transaction, at the moment of update, with correction oppertunities.

2. The moment of control for the dynamic constraints.

We will use the examples of 2.2.

Example 1, a dynamic attribute constraint:

In accordance with fig. 3 a Second Hand car can not become a New car etc..
The check on the violation of the dynamic constraint can in this case be done at the moment the values are updated. Almost by definition however, dynamic constraints have to be checked at the moment of commit: it is a constraint on the values at the beginning and just before the commit of a transaction. These values have to be compared and then it can be decided whether the transaction can really commit because the dynamic constraint is fulfilled or whether this is not the case and the transaction has to be rolled back.

Example 2, a dynamic database constraint:

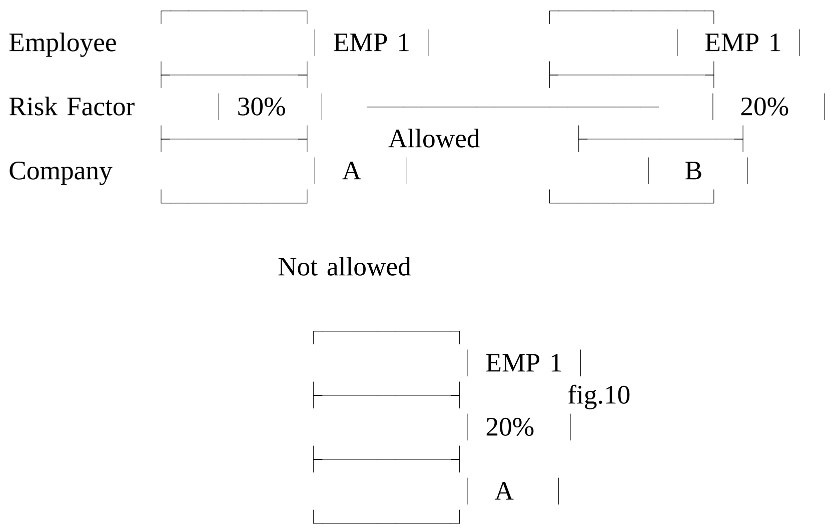


fig.10

At the update of the Risk-Factor it is not clear whether this is allowed or not, a check on the type of contract of the company where the employee works is necessary. If the Risk-Factor update is done first, a check on update-time will result in a roll-back of the transaction. If the company is updated first, the transaction can be executed succesfully .
Because the update of the employee record involves the updates of two fields, the check on the two updates have to be done at the moment that it is certain the two updates are done: at the moment of commit.
This seems another example of the user having to know the structure of the database. In this case not only the structure, but also the constraints and the contents of the database has to be known to the user to be able to make the updates succesfully.

The examples show that:

- For static constraints involving one table check on commit is not necessary

- For other constraints, static constraints involving more than one table and dynamic constraints, sometimes a check on commit is necessary and sometimes not. In case it is not necessary it is still very desirable, otherwise the user needs to have a reasonable knowledge of the structure of the database and referential integrity.

IV. The functional benchmark of IDT-Holland

A. The structure of the benchmark.

The benchmark consists of two parts:

1. The implementation of the conceptual schema of the Hospital case, developed at the TU Eindhoven, by the suppliers using their products.
   Of this Hospiatel case both an informal and a formal description were distributed to the participating vendors. The formal description represented in fact a complete analysis and description of the database contents with the relevant constraints. Some of the constraints are of a referential nature, others are related to actual permissable values in the database. The emphasis of the case is on the validation of constraints. There are no update and dynamic constraints described in the case.

2. The completion of a questionnaire covering 222 different questions on functional features.

Apart from the functionality the DBMS offers in controlling the quality of the data, however, there is a multitude of other factors to consider when selecting a particular DBMS.

Aspects such as the quality of the vendor, the soft- and hard-ware environment in which the selected DBMS should function, performance features and price, are a few, but not all the aspects that play an important role in the selection of 'the best' DBMS for an organization.
The goal of the IDT-comparison was not to provide an overall evaluation of the different products available, but to report as objectively as possible on the features found in the different DBMS's that control the information quality and the usefulness.

B. The application of the benchmark.

The case implementation and the replies to the questionnaire have been extensively tested by representatives of the organising committee.

The procedure for the vendors was:

1. To implement the database universe (The hospital case).

2. To assist members of IDT-Holland in the execution of a number of transactions on their DBMS to check the correct implementation of the various types of constraint included in the case. During this session, it was established in which way the various constraints had been implemented: by central control via the DD/D, via the VIEW-concept or via an application module. A print-out of the solution had to be supplied to the team.

3. The answers to the questionnaire had to be given, sometimes these were discussed and evaluated.

C. The publication of the results of the IDT-benchmark.

The benchmark has been applied in 1987 on 7 RDBMS's, in 1988 on 14 DBMS's. In the summer of 1989 the benchmark has been applied to 16 DBMS's.
The results of the benchmark were communicated to the public in a few conferences. The conference days were prepared in close cooperation with the suppliers. One day congresses were held in Holland june 1987 and june 1988. The second dutch conference was repeated in December 1988 in London as part of the Codd & Date conference on Relational DBMS's. The results of the summer 1989 were presented at a two day's conference in Berlin, also in cooperation with Codd & Date.
For 1990 conferences are planned in Belgium, Canada and the USA.

The suppliers are also involved in the conferences: a presentation had to be given where an explanation had to be given of the implementation of 8 different constraints that were part of the case and that were selected in advance by IDT-Holland.

V.Conclusion.

By vendors and in textbooks on DBMS's a lot of attention is given to two aspects of datamanagement in which transactions play an important role: recovery and concurrency. Because a transaction is the logical unit of integrity it should be expected that vendors who want to support the concept of transaction offer sufficient possibilities to check integrity.
The answers in questionnaire of IDT show that all systems support a check of the constraints at update time, but that only one system supports a check-on-commit time.
Constraints that involve only one table often can be checked at the moment of update. However, for certain static and dynamic constraints you need to be able to check integrity on the moment of commit of an transaction. This is a vital point in the protection of the consistency and integrity of the database and more support for this feature from the side of the suppliers is necessary.

VI. References

1. An introduction to Database Systems, Volume I, Fourth edition.
   C.J. Date, Addison Wesley 1986.

2. Fundamentals of Database Systems.
   Elmasri & Navathe, Benjamin/Cummings 1989.

3. Database System Concepts.
   Kort & Silberschatz, Mc Graw Hill 1986.

4. De Database Club van de Sectie Informatiesystemen van het Nederlands Genootschap voor Informatica. Het praktische gebruik van relationele systemen gepresenteerd. Amsterdam 1988.

5. Independent Database Team-Holland, Relational database in practice.
   Codd & Date Ltd. London 1988.

6. Independent Database Team-Holland, The evaluation of 16 Top RDBMS's using the functional benchmark. Codd & Date Ltd. Berlin 1989.

7. The transaction Concept: Virtues and Limitations, Jim Gray. Readings in Database Systems, Morgan Kaufmann Publishers Inc. 1988.

8. Constraints and their role in the functional benchmark of IDT-Holland. E.O. de Brock, H. Gersteling, M.J.Krijger, S.C. van der Made-Potuijt. In: The proceedings of the conference of the Unisys Users Group. Oxbridge, England, juni 1989.

9. The transaction concept and its implementation, S.C. van der Made-Potuijt . In: The proceedings of the conference 'Benchmarking Database Functionality'. Codd & Date Ltd. Berlin, oktober 1989.

CONTENTS.