

Data Storage and Data Access

This section provides information and guidelines regarding data storage and data accesses.

Where relevant, alternative approaches are presented. All techniques can be combined with the use of the frame gallery production frames. Specific components or suggested codes which support a given alternative are also described.

The following topics are covered below:

- Terminology
- Concepts for Data Storage
- Time Stamped Data
- Histories
- Multiple Control
- Logical Deleting
- Multilingual Applications
- Access Paths
- Structuring Physical Files
- Synchronizing Competing Accesses

Terminology

Adabas C terminology is used throughout this chapter. The chart below displays the equivalent SQL terminology.

Adabas Terminology	SQL/Adabas D Terminology
File	Table
Record	Row
Data record	
Field	Column
Descriptor Index	Primary key
Super descriptor	Named Index (key with several parts)
READ	SELECT
READ(1)	SELECT SINGLE SELECT DIRECT

Concepts for Data Storage

Some business requirements influence not only the logic of the business functions but also the data storage.

To allow for these requirements, in certain circumstances, technical fields in addition to the fields for business data must be inserted into the entities.

The following sections describe some of the possible requirements, including the necessary modifications or extensions to the entities or files.

- Time Stamped Data
- Histories
- Multiple Control
- Logical Deleting
- Multilingual Applications

The implementation of the functions for processing these data is not described in this section.

Time Stamped Data

General

If there is a requirement to be able to process data for a key value dependent on particular periods, without modifying the data contents of other periods, then the data must be stored and processed in dated form.

Such data are only valid for a particular period. Since the defined periods for a key value cannot overlap, the whole data over all defined periods yields a life cycle of the "data record" for a key value.

Definition:

An entity is time stamped if, for each key value, several occurrences of a descriptive attribute can exist, where each occurrence is identifiable by a distinct period which does not overlap with another period. This guarantees that, at any point in time, only one value of a descriptive attribute exists.

Accordingly, a data record which contains an account status is not a time stamped data record, since the value of the account is only valid for exactly one point in time. The date, in this case, is a business content.

There are many models for utilizing the concept of time stamping for data storage and processing. The requirements that call for the use of a time stamping model can, in spite of their complexity, be summarized in a few words:

- The object must have a life cycle.
- The life cycle may or may not have gaps.
- Data from past or future periods is viewed from a given point in time, and can be modified or not.
- The limits of the periods are milliseconds, seconds, minutes, hours, days, and so on.

Time Stamping Concept Recommendation

The following time stamping concept is a recommendation for data storage of time stamped objects as well as access to the data from business functions.

The concept supports the following requirements:

- The data record has a life cycle which has either a defined end or can extend to "infinity".
- Gaps in the life cycle are allowed.
- Pre-time stamping, that is, processing the data for a period that at the time of processing is not yet valid, is allowed.
- Post-time stamping, that is, processing the data for a period that at the time of processing is no longer valid, can optionally be implemented.
- The limits of the periods are days, that is, the smallest possible definable period is one day.
- Frame gallery time stamping relates to the whole object, it is not field related.

If an entity contains time stamped and unstamped descriptions, then the division of the entity into one time stamped and one unstamped is to be considered.

Data Storage

- For each time stamped object, one physical data record is stored for each period in the database.
- The data structure is extended by two fields that contain the start and end dates of the period.
- Each period is identified through a combination of specialized key and start date.
- To simplify the read access, the time stamp values are stored in complementary form with its complementary value "999999...9 - date". The exit time stamp must be in the format N12 of the Natural variable *DATX.

To prepare an entity for dating

1. Insert the following fields:

xxx_EFD_INV (N12)	effective-from-date	(complementary)
xxx_ETD_INV (N12)	effective-to-date	(complementary)
2. Define the superdescriptor for the access:

xxx_KEY (...),	consisting of
xxx_specialized_key	and
xxx_EFD_INV	
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:

xxx_KEY_SEC (...),	consisting of
xxx_secondary key	and
xxx_EFD_INV	

Access to Time Stamped Data

Data accesses are basically through a superdescriptor xxx_KEY or xxx_KEY_SEC.

The frame gallery philosophy to always encase data accesses in so-called access modules is also valid for time stamped data.

The suggested codes for implementing accesses to time stamped data are contained in the skeleton modules for creating access modules. The appropriate skeletons are identified by the suffix 'PE', for "Period effective".

The following standard operations are typical:

Reading a Period

A physical data record is read using a key value and a date. The date is provided in complementary form. Using the value provided as from-date, the data record is read using a READ(1) (see following example).

Example:

For article number 4711, the following periods exist (represented in the form *YYYYMMDD*):

Art.Nr.	Valid from	Valid to
4711	19930101	19930330
4711	19930401	19930515
4711	19930516	19930831
4711	19930901	unlimited

Physically, the values are represented in the form "*9999..9 - date*":

Art.Nr.	Valid from	Valid to
4711	80069898	80069696
4711	80069598	80069484
4711	80069483	80069168
4711	80060098	00000000

The date is *02/05/93* with the complementary value *80069497*. Using this date, the start date is accessed. By using the access READ(1), the next highest value is read.

Result of read access: *80069598* - which is the equivalent of *01/04/93*.

With this, the validity period is identified. The date exists in the period *01/04/93 to 15/05/93*.

Existence Checking on Key Value

A read access READ(1) with key value and date "infinity", complementary value "0", is used. This results in a record found, if one exists with the specified value.

Adding a New Key Including a Period

The data record is stored with key value including start and end date in complementary form.

Modifying the Data of a Period

The descriptive attributes of the data record are modified, without modifying start or end date.

Modifying the Limits of a Period

Depending on the kind of modification and the specialized requirements, quite varied activities can be necessary:

- **Overlapped Periods.** All periods except the modified period itself, which are completely overlapped by the modification of a period are deleted.
- **Partly Overlapped Periods.** Already existing periods which are partially overlapped as a result of a time period modification are automatically shortened.

The existing period which is impacted by new Start Date of the modified period is:

- End date of old period = start date of the modified period minus 1 day.
- New end date of the impacted period is prior to its current end date.

The existing period impacted by the new End Date of the modified period is:

- Start date of the old period = end date of the modified period plus 1 day.
- New start date of the impacted period is **after** its current start date.

Fixing the New Limits

The limits of the period being processed are adjusted correspondingly.

<p>Example:</p> <p>Existing period:</p> <ol style="list-style-type: none"> 1 01st May - 31st May 2 01st June - 15th June 3 16th June - 17th August 4 18th August - 20th September <p>Period to be modified: 01st June - 15th June</p> <p>New limits: 15th May - 01st September</p> <p>To delete: 16th June - 17th August</p> <p>To modify: 01st May - 31st May -> 1st May - 14th May (new end date) 18th Aug. - 20th Sept. -> 2nd Sept. - 20th Sept. (new start date)</p> <p>Result:</p> <ol style="list-style-type: none"> 1 01st May - 14th May . 2 15th May - 01st Sept. 3 02nd Sept. - 20th Sept.

Adding a New Period for a Key

- **Overlapped Periods.** All periods completely overlapped by the new period are deleted.
- **Partly Overlapped Periods.** All partly overlapped periods are shortened, as described in the section: Modifying the limits of a period.

- **Adding a new period.** A new data record, with the limits given, is added.

Deleting a Period

- **Overlapped periods.** All periods that are completely overlapped by the deleted period are deleted.
- **Partly Overlapped Periods.** All partly overlapped periods are shortened, as described in the section: Modifying the limits of a period.

Example:	
Existing periods:	
1	01st May - 31st May
2	01st June - 15th June
3	16th June- 17th August
4	18th August - 20th September
Period to be deleted: 15th May - 01st September	
To be deleted:	01st June - 15th June 16th June - 17th August
To be modified:	01st May - 31st May -> 01st May - 14th May 18th Aug. - 20th Sept. -> 02nd Sept. - 20th Sept.
Result:	01st May - 14th May 02nd Sept. - 20th Sept.

Deleting a period

The data record is deleted.

Deleting a key

All data records belonging to a specialized key are deleted. For this, a READ access is used with start value *key-value* and *start-date equals "infinity",complementary-value '0'*. This results in all affected data records to be found.

Histories

Histories are used when every data modification must be logged.

With every data modification, the original data record is retained. It is merely identified as historical. The modified data record is recorded as a copy of the original record including the modification as a "valid" record in the data content. Every historical record must contain the time of the modification, to be able to find out the chronological order of the modifications.

The concept of keeping histories can be implemented in various ways. The use of this concept affects the data modelling and also the implementing of the specialized functions affected.

In the remainder of this section, three ways of keeping histories are described:

- Histories in the original file with validity identifier.
- Histories in the original file with additional key.
- History keeping in a separate file

Histories in the Original File with Validity Identifier

In this way, all data records that have become historical by modification are kept in the original file. To separate the historical data from the current, a validity identifier is introduced.

Access to the current data is only through the combination of validity identifier and specialized key. Access to historical data is through a separate key.

Data Storage

For logging the modification time, the entity is extended by a field which is used to enter the time stamp of the modification time. For this, the format N20 is provided. To guarantee the uniqueness of the time stamp, the value of the Natural system variable *TIMESTAMP (B8) is used as modification time. It can be completely portrayed in N20.

If the need exists to read all historical data in a chronological order independent of the specialized key, the time stamp must be defined as key value.

Additionally, a validity identifier is introduced and a combined key is constructed from validity identifier and specialized key.

Finally, a history key for access to the historical data is built as a combination of specialized key and time stamp.

► To prepare an entity for history keeping

1. Insert the following fields:
xxx_PTS_HIST (N20) time stamp of modification (Processing Time Stamp)
xxx_ACTIVE (N01) validity identifier (Active Flag)
Important:
 The validity identifier must, in any case, be added with null-value suppression.
2. Define the superdescriptor for the access to current data:
xxx_KEY (...) consisting of:
xxx_ACTIVE and
xxx_specialized_key
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:
xxx_KEY_SEC (...) consists of:
xxx_ACTIVE and
xxx_secondary_key
4. Define the superdescriptor for the access to historical data:
xxx_KEY_HIST (...) consisting of:
xxx_specialized_key and
xxx_PTS_HIST
5. If the entity secondary key must be read through the historical data, define for each secondary key a further superdescriptor in the form:
xxx_KEY_SEC_HIST (...) consisting of:
xxx_secondary_key and
xxx_PTS_HIST

Accesses

- **Reading a valid data record**
 Access is through the superdescriptor **xxx_KEY** with start value specialized key and validity identifier 1 (historical records have validity identifier 0).
- **Existence checking of a valid data record**
 According to specialized requirements, an unsuccessful read access to valid data must follow an access to historical data, to find out whether a key was already used at some time in the past.
- **Adding a valid data record**
 The record is stored with validity identifier 1.
- **Modifying a valid data record**
 The modifications are on a **copy** of the original data record. The copy can be stored in a temporary store or as historical record in the original file (then access to the copy must be through the time stamp).
 After finishing the modifications, the original record is **copied** and provided with validity identifier 0 and the current time stamp. Then the modifications are brought into the old original record. All finishing activities are completed within one database transaction.
- **Deleting a valid data record**
 Deleting data records is, as a rule, not usual when using a histories concept. Should a deletion still be required, the validity identifier of the original record is merely set to 0 and the current time stamp entered.
- **Reading a historical data record**
 Access is through the superdescriptor **xxx_KEY_HIST** with start value specialized key and required time stamp.
- **Adding a historical data record**
 See section Modifying a valid data record.
- **Modifying a historical data record**
 Modification of historical records is, as a rule, not usual.
- **Deleting a historical data record**
 The physical deletion of individual historical data records is not usual. Historical data are, as a rule, only removed from the data content in the course of archiving.
 Access can then, according to requirement, be either through the time stamp **xxx_PTS_HIST** or through the

history key `xxx_KEY_HIST`.

Usually, at a particular point in time, all historical data records that are older than a particular date are archived. For this, the time stamp `xxx_PTS_HIST` is used.

Histories in the Original File with Additional Key

In this way, all data records that have become historical by modification are kept in the original file. To separate the historical data from the current, an additional key for the historical specialized key is introduced.

Accesses to current data are through the valid specialized key. Access to historical data is through the separate historical specialist key.

Data Storage

For logging the modification time, the entity is extended by a field which is used to enter the time stamp of the modification time. For this, the format N20 is provided. To guarantee the uniqueness of the time stamp, the value of the Natural system variable *TIMESTAMP (B8) is used as modification time. It can be completely portrayed in N20.

If the need exists to read all historical data in a chronological order independent of the specialized key, the time stamp must be defined as key value.

Moreover, an additional field is inserted which matches the specialized key in format and length.

Finally, a history key for access to the historical data is built as a combination of specialized key and time stamp.

To prepare an entity for history keeping

1. Insert the following fields:
 - `xxx_PTS_HIST (N20)` **time stamp of modification (Processing Time Stamp)**
 - `xxx_ID_HIST (...)` **historical specialized key (Format and length = specialized key)**
2. If the entity contains secondary keys through which there must be access, define for each secondary key a further field in the form:
 - `xxx_ID_SEC_HIST (...)` **historical secondary key (Format and length = secondary key)**
3. Define the superdescriptor for the access to historical data:
 - `xxx_KEY_HIST (...)` **consisting of:**
 - `xxx_ID_HIST` **and**
 - `xxx_PTS_HIST`
4. Define the superdescriptor for the secondary keys:
 - `xxx_KEY_HIST_SEC (...)` **consisting of:**
 - `xxx_ID_HIST_SEC` **and**
 - `xxx_PTS_HIST`

Accesses

Accesses through the specialized keys are not affected.

All accesses to historical data are through the key `xxx_KEY_HIST` or `xxx_KEY_SEC_HIST`.

Remarks on these accesses are to be taken from the previous section.

History Keeping in a Separate File

This approach results in a complete separation of the valid from the historical data. All historical data are kept in a separate entity.

This construct presupposes no technical fields in the file for the original data. The history file is an image of the original file, where additionally, for the purpose of chronologically sorting the historical data records, a time stamp is maintained which indicates the time of the history.

Data Storage

To store the historical data, a separate entity is added, which has exactly the same structure as the original.

Additionally, a time stamp is added as field, to document the modification time of a data record. This time stamp is defined as a descriptor or, as required, as part of a superdescriptor from specialized key and time stamp.

To prepare the data structures for history keeping

1. Add a file that has the same field structure as the original file.
According to requirement, descriptor definitions can be dropped.
2. Insert the following field:
xxx_PTS_HIST (N20) time stamp of modification (Processing Time Stamp)
3. Define this field as descriptor if there is a need to read all data of the file in chronological order.
4. Define the superdescriptor for the access to historical data:
xxx_KEY_HIST (...) consisting of
xxx_specialized_key and
xxx_PTS_HIST
5. If the entity contains secondary keys through which the historical data must be read, define for each secondary key a further superdescriptor in the form:
xxx_KEY_SEC_HIST (...) consisting of:
xxx_secondary_key and
xxx_PTS_HIST

Accesses

- **Reading a valid data record**

Not affected.

- **Existence checking of a valid data record**

According to specialized requirements, an unsuccessful read access to valid data must follow an access to historical data, to find out whether a key was already used at some time in the past.

- **Adding a valid data record**

Not affected.

- **Modifying a valid data record**

The modifications are on a temporary copy of the original data record.

After the modifications are finished, the contents of the original record including the current time stamp are transferred into the history file. Then the modifications are brought into the original record.

All further activities are completed within one database transaction.

- **Deleting a valid data record**

The deletion of data records is, as a rule, not usual when using a history concept.

If a deletion is still required, then a copy of the original with the current time stamp is transferred into the history file and the original is deleted.

Access to historical data is analogous to the method used in the previous section, however, with the difference that another file or user view is accessed.

Multiple Control

From a business area perspective, certain data modifications must be confirmed before being used.

The number of necessary confirmations depends upon specialized requirements. Usually, one additional confirmation is enough (**double-review principle**).

In rare cases, the **triple-review principle** is used. For this, 2 additional confirmations are necessary.

Data modification and each confirmation are, as a rule, carried out by different people.

The use of this concept must be considered both with the design of the entity and also with the function structure.

There are various ways of implementing this concept. The spectrum ranges from very simple to very complex.

Complex Variant

The following complex variant, which is user friendly, is not however described below in general terms:

- There is an entity that contains all information about the processing state of all affected data records. Here, not only are the current states recorded, but also the entire processing history is noted, with specialized key, status, processor and time of processing.
- There is a further entity, that contains all status info that are necessary for a specialized entity.
- Finally, there is an entity that contains information about which user can process which data record in which status and, if the data record must be further processed, to whom appropriate information must go.

For all entities, appropriate maintenance functions must be generated.

With this variant, the specialized entities remain untouched. Before each access to a specialized data record, there are accesses to the status files.

Simple Variant

A very simple variant would be the following, which is described in detail:

- To each specialized data record, a status field is allocated.
- Additional fields for logging the last processor and the date of processing can be added.

Data Storage

The state of a data record is documented with the help of a status field. The status field contains information about the progress of processing, such as, "*added*", "*checked once*", "*approved*" and "*declined*".

As required, additional data fields are to be provided for the user who carries out an activity and the time that the activity is carried out.

To prepare an entity for the multiple review principle

1. Insert the following fields:

xxx_STATUS (A01)	status field
xxx_USER (A08)	last processor
xxx_PTS (N20)	time stamp of last processing
2. Define the superdescriptor for access to the data in dependence on its status:

xxx_KEY_STAT (...)	consisting of:
xxx_STATUS	and
xxx_specialized_key	
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:

xxx_KEY_SEC_STAT (...) consisting of:
xxx_STATUS and
xxx_secondary_key

Accesses

All accesses to the entity are through a superdescriptor

xxx_KEY_STAT or
xxx_KEY_SEC_STAT

The remaining processing, such as, modifying or deleting, is dependent on the specialized requirements.

Logical Deleting

In order that data previously deleted can be recreated, it must not be physically deleted, but only logically deleted.

Various solutions are possible. With the most comprehensive, the data are provided with a history, where the data only have a history in the case of a logical deletion. The data structure here matches that for complete history.

In the following, a simpler method is introduced. Here only the last content of the data to be logically deleted is stored. In the case of a recreation of the data, the data record is simply "activated" again.

Data Storage

The data structure is extended by a validity identifier. For access to valid data records, again a superdescriptor is defined. Access to deleted data is through the specialized key.

To prepare an entity for the logical deleting

1. Insert the following field:
xxx_ACTIVE (N01) validity identifier (Active Flag)
2. Define xxx_ACTIVE null-value suppressed.
The field will contain 0 when the record is logically deleted.
3. Define the superdescriptor for access to the valid data:
xxx_KEY (...) **consisting of:**
xxx_ACTIVE **and**
xxx_specialized_key
4. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:
xxx_KEY_SEC (...) **consisting of:**
xxx_ACTIVE **and**
xxx_secondary_key

Accesses

All accesses to valid key values are through a superdescriptor xxx_KEY or xxx_SEC_KEY. Accesses to logically deleted data are through the specialized key.

- **Reading a valid data record**
Access is through the superdescriptor xxx_KEY with start value specialized key and validity identifier 1 (logically deleted records have validity identifier 0).
- **Existence checking of a valid data record**
According to specialized requirements, an unsuccessful read access to valid data must follow an access to logically deleted data, to find out whether a key was already used at some time in the past.
- **Adding a valid data record**
The record is stored with validity identifier 1.
- **Modifying a valid data record**
Not affected.
- **Logically deleting a valid data record**
The active identifier is set to 0.

- **Physically deleting a valid data record**
Deleting data records is, as a rule, not usual when using logical deletion. Should a deletion still be required, reading and deleting is then through the specialized key.
- **Reading a logically deleted data record**
Access is through the specialized key. Through this, a logically deleted or a valid data record can be found. The identification of a logically deleted record is through further selection: xxx_ACTIVE must be 1.
- **Adding a logically deleted data record**
See section Deleting a valid data record.
- **Modifying a logically deleted data record**
Not usual.
- **Deleting a logically deleted data record**
Deleting logically deleted data records is, as a rule, not usual. This deleting is, in general, only done with physical deletion of a valid data record.
- **Recreating a logically deleted data record**
The xxx_ACTIVE identifier is set to 1.

Multilingual Applications

Many applications must be available in multiple languages. For this, first the interface (dialogs and command language) must be prepared in multiple languages. Also on the data level, entities will be identified that contain multilingual elements.

Language-dependent data elements of an entity are frequently identified in the design phase. As a first step, it must be decided whether the multilingual fields are to be left in the data structure or whether a separate entity that contains the multilingual fields will be generated.

Using a Separate Entity

The new entity contains all language-dependent fields of the data structure. In addition, a language field is inserted.

Use this variant only when a doubling of the number of read accesses when accessing language-dependent fields is justifiable.

Data Storage

To define the structure of the language-dependent entity

1. Define per language-dependent field a field of the same format and same length as the original:
xxx_Field (A..)
2. Define all key values that are used to read language-dependent data in the language-dependent entity:
xxx_ID (...)
3. Define additionally a field with the form:
xxx_LANG (N01) language code of the language considered
Take the language codes from the frame gallery table *Language*. Use the content of the field *xxx_ID*.
4. Define per key value of the original entity a superdescriptor of the form:
xxx_KEY consisting of xxx_LANG and xxx_ID
In rare cases, it can be sensible to turn round the fields of the superdescriptor and so have the key value in front.

Accesses

All accesses to the specialized object are performed by two real accesses:

- Access to the language-**in**dependent data,
- Access to the language-dependent data.

Accesses to the language-dependent data are always through a key of the form *xxx_KEY*.

Language-Dependent Fields in the Entity

With this variant, language-dependent and language-independent fields are held in one data structure. Naturally such structures require a special design.

Data Storage

There are numerous possibilities for portraying language-dependent fields. The "right" design can, however, only be selected depending on the context.

In the following, only the variants "multiple field" and "periodic group" are represented.

- **Multiple Field**

The storage of language-dependent data as multiple field is recommended when there is a small number of multiple-language fields per record and when the field must be a descriptor or part of a superdescriptor through which there must be sequential reading.

One occurrence of a multiple field is used per language.

To enable direct access to a particular occurrence of the multiple field, a byte (N1) is placed before the actual field content.

The byte contains the frame gallery language position of the language to which the field content belongs.

The multiple field is therefore one place longer than the original field:

Define then per language, instead of the language-dependent field, a multiple field:

xxx_LFIELD(A..) = language_position (N1) + field(A..)

When defining the structure, take note of the maximum possible physical record length. It must not be exceeded.

- **Periodic group**

The portrayal of language-dependent fields in a periodic group can only be used when none of the fields is used as a key value or part key.

A language-dependent field content is put in the occurrences of the periodic group that matches the frame gallery language position (LZ_LANG_POS in LDA ZXXGLOBL).

Define then a periodic group that contains all language-dependent fields that are not used as key or part key:

xxx_LANG_FIELDS (9)

xxx_FIELD 1 (A..)

:

xxx_FIELD n (A..)

With frame gallery, 9 languages are available.

When defining the structure, take note of the maximum possible physical record length. It must not be exceeded.

Accesses

When you are reading through a key value, descriptive language-dependent data can be taken without problem from the occurrence that matches the frame gallery language position.

Reading accesses are only possible when using multiple fields.

In the following, only access to multiple fields is considered:

- **Taking out language-dependent attributes**

The language-dependent field content must always be taken through an intermediate field that is redefined analogous to the data field into language and original field.

- **Reading through a language-dependent key**

Access is through the multiple field xxx_ID with required language position and specialized key.

- **Sequential read access to language-dependent key**

When sequentially reading through multiple fields, there is generally the problem of being able to immediately identify the last data record of a language. Therefore special break-off conditions must be formulated within the read loop:

```
xxx_FIELD (language) LT Start_value
xxx_FIELD (language) LE #OLD_xxx_FIELD
```

If no start value was given, the first value in the language should be established by a HISTOGRAM statement with an end condition.

```
#OLD_xxx_Field Value last time through read loop
```

- **Adding a language-dependent field**

When first adding the record that contains the multi-language field, the occurrences for all other languages defined in the system are also to be added, to avoid the multiple fields pushing together and the consequent destruction of the direct access path. For this, only the language position is filled and the actual field content stays blank.

Access Paths

In addition to the access paths using a descriptor or superdescriptor, there are accesses that require certain preconditions to lead to a result simply and with good performance.

These access paths are described below.

- Sequential Reading through Nonunique Key
- Upper/Lower Case

Sequential Reading through Nonunique Key

Nonunique key fields, usually names, can cause problems in the paging logic.

To avoid this problem when reading, all keys that are not unique by nature are made unique with the help of additional fields.

For a nonunique key, a superdescriptor is defined which consists of the nonunique key and one of this data record's unique identifying fields (for example, the identifying key).

The nonunique key must not always be taken completely into the superdescriptor. As a rule, the first 15 to 20 characters are enough.

To extend the entity

- Define for each nonunique secondary key a superdescriptor:

xxx_KEY_SEC	unique secondary key consisting of:
xxx_NAME (1-10)	bytes 1-10 of the name
xxx_ID	specialized key

When no nonsequential accesses are required, the nonunique key need not be defined as a descriptor. All sequential accesses are then through the superdescriptor.

Upper/Lower Case

Frequently, alphanumeric values, for example, names, should be included on the dialog and also displayed in upper and lower case.

If this is a key value, through which there must be access, storing in upper and lower case can have the danger that the value sought may not be found because, for example, during entry an upper case letter was accidentally entered in the middle of the key.

This problem can be avoided as follows:

- The field value is stored twice, once in upper and lower case and once only in upper case.
- As key for reading, the value stored in upper case is used, the value in upper and lower case is only used for display.

This procedure requires the insertion of an additional field into the entity.

 **To extend the entity**

- Define for each key value that must be displayed in upper and lower case an additional field.
xxx_CODE_UC specialized key in upper case
This field is added as descriptor. The field that contains the value in upper and lower case is defined as descriptive attribute.

Read accesses are always through xxx_ID_UC. Modifications of the field content must be made in both fields.

Structuring Physical Files

Note:

This section can be skipped if you use SQL since the SQL database design uses a "flat" file structure.

When creating Adabas C files, there are the following possibilities:

- translating one entity from the requirements analysis 1:1 into one physical file,
- defining several entities in one physical file.
The entities are arranged one behind another in the physical file.

The preferred method for an entity can be determined from the context.

For example:

- number of data elements,
- expected data volume,
- expected access and modification frequency,
- logical associations of the entities.

An entity with a large number of data elements, high expected data volume, and extremely high likelihood of modification is certainly to be represented as its own physical file.

On the contrary, several smaller entities can be put together in one physical file to reduce the number of physical files.

The following rules should be observed:

- The entities are placed one behind another in the file. At the beginning of a new entity, a corresponding comment is inserted in the PREDICT file description.
- Mixing of data elements from different entities is not allowed.
- One field is basically not used by two different DDMs, to avoid influence of the DB designs on the program logic.
This is also not done if the field content is the same. A field per DDM is always added, see the following example:

physical file

**** View ABCD ***

ABCD_ID
ABCD_NAME ! DDM for ABCD
ABCD_FIELD1 !

**** View EFDH ***

EFGH_ID
EFGH_NAME ! DDM for EFGH
EFGH_FIELD1

- READ physical or READ BY ISN on the DDMs of such a file is to be avoided, since the ISNs of the other DDM of this file would also be read.

Synchronizing Competing Accesses

The following topics are covered below:

- General
- Use of Locking Concepts
- Pessimistic Locking Concept
- Optimistic Locking Concept
- Organizational Locking Concept
- Processing Without Locking Concept

General

Concepts for synchronizing competing accesses are, in general, needed when the possibility exists that data records could be modified at the same time by several users or programs.

Various concepts can be used to avoid this problem.

These concepts extend from purely organizational measures, through checking for data modifications during the processing time to active locking of data records, as soon as the data records to be processed are identified.

A selection of these concepts is described below. All concepts can be used when implementing specialized functions on the basis of the frame gallery production frames.

Use of Locking Concepts

Locking concepts are used in the case of competing updating. Display functions usually do not require the use of a locking concept.

Use a locking concept, if:

- several users per dialog function could modify the same data records at the same time;
- parallel with the dialog service, batch programs are running that modify data;
- evaluations are being made that require a defined state of the basic data;
- data that serve as a basis for the current processing of data but must not be modified during the processing.

Important: Decide on one locking concept per entity.

The use of various locking concepts on one entity will lead to difficulties with the implementing of the specialized function or even lead to data inconsistencies.

More reasonable for implementing and maintenance of the application is the use of one locking concept for all data that are processed in the application.

Pessimistic Locking Concept

Concept

As soon as a data record is identified for processing, a lock is written for this data record. This lock remains until the close of the logical transaction. In this time, no other user can modify the data record.

This Naturally presupposes that, after the identification of a key value, there will first be a check whether the data record is already reserved for another user. If this is the case, then access to the data record is rejected.

The pessimistic locking concept is supported by the frame gallery production frames and suggested codes. If, when implementing, you do not depart from the suggested codes, pessimistic locking is automatically implemented.

The pessimistic locking concept is the "safest" and, for the end user the most comfortable of the concepts represented here.

Operation

From the viewpoint of pessimistic locking, a transaction looks as follows:

1. identification of the data record.
2. checking whether a lock already exists.
 - if yes, the access is rejected.
 - if no, a lock is written.
3. data modification.
4. transaction end: the lock is removed.

For each further data record that is identified for modification during the running transaction, a lock can, if necessary, be written. All locks are removed at the end of the logical transaction.

Data Storage

No particular measures are necessary.

Determining the Data to be Locked

Besides individual data records, key areas or entire objects can be locked. This is part of the frame gallery basic functionality.

Note:

When using frame gallery, the primary key is locked. Any additional keys must be manually locked with subroutine `Z_LOCK_RECORD`.

Which data key in which specialized function must be locked, must be decided from the specialized context.

Frequently, it is not necessary to lock every individual data key that is processed in a specialized function:

- Entire objects should be locked when the majority of the data records in a specialized function is modified or else must not be modified during the running transaction, as, for example, happens from time to time in batch programs.
- Key ranges should be locked when, at the beginning of the transaction, it is already clear that precisely these ranges are affected by the processing.
- Individual keys should be locked when the affected key values can only be identified one after another.
- Keys should not be locked when they are stored in a tree-like structure, whose root object is already locked and there is no possibility of otherwise accessing these hierarchically subordinate data keys.

Application

This approach guarantees that even very time-intensive logical transactions can be successfully closed. Once a transaction has begun, the data are reserved until the end of the transaction.

The approach is therefore especially suitable for dialog systems, since even specialized functions, which run through a large number of dialogs can not be destroyed during processing by other users or transactions.

Even in parallel use of dialog and batch functions, the approach is very suitable. The two types of functions cannot conflict with one another.

Optimistic Locking Concept

Concept

Note:

Frame gallery does not generate optimistic locking concept. It must be manually implemented.

Every data record contains a time stamp that documents the time of the last modification.

As soon as a data record is identified for processing, the time stamp of the original record is saved in a temporary store, in general in the GDA.

At the close of the logical transaction, before the transfer of data, there is a check whether the time stamp in the original record has been modified.

If the time stamp in the original record was **not** modified, the transaction is closed in an orderly way.

If the data record, however, was modified, data transfer is terminated and the modifications of this transaction are lost.

The optimistic locking concept can be applied when using the frame gallery production frames. However, there must be a departure from the suggested code.

In contrast to the pessimistic locking concept, the optimistic locking concept is less comfortable, since not until the close of the transaction is there a check whether the modifications can be transferred into the database.

Operation

From the viewpoint of optimistic locking, a transaction looks as follows:

1. identification of the data record.
2. saving of the time stamp.
3. data modification.
4. transaction end: the time stamp is the same.
 - if yes, the transaction is closed in an orderly way.
 - if no, the transaction is stopped. The modifications are lost.

For every further data record that is identified for modification during the running transaction, a time stamp must be saved, which at the end of the transaction must be checked.

Data Storage

Every data record must contain a time stamp. Since this time stamp must be unique, it is provided with the converted content of the Natural system variable *TIMESTMP.

To extend your entity

xxx_PTS (N20) Modification time stamp

Determining the Data to be Locked

Which data key in which specialized function must be locked must be decided from the specialized context.

Since the locking runs through a time-stamp comparison, the time stamp of every affected data record must be checked.

The single exception is hierarchically structured data records. If a data record is stored in a tree-like structure, whose root object is already locked and there is no possibility of otherwise accessing this hierarchically subordinate data key, no time-stamp comparison must take place.

Application

This approach can lead to data that are modified in a logical transaction not being taken over into the data content, because the original data meanwhile were modified by another logical transaction.

The approach is therefore not suitable for dialog systems that also contain specialized functions that run through a large number of dialogs, since the likelihood of data being modified during the transaction is great.

It is suitable when the probability of competing accesses is small. This is the case when the logical transactions can be closed very quickly and few transactions access the same data.

Organizational Locking Concept

Concept

This procedure presupposes that the possibility of competing accesses is hindered by organizational measures.

This can, for example, be guaranteed when only one user is responsible for the processing of particular key ranges. This should then, however, be guaranteed by measures for protecting field content.

The organizational locking concept can be applied when using the frame gallery production frames. Here you must, however, deviate from the suggested code.

This concept pursues a completely different approach from the concepts previously described. At data access there is no effort in locking. The effort, however, appears at another place.

Data Storage

No measures are necessary.

Application

This concept can only be used in exceptional cases, since normal dialog systems are constructed to make the data available to multiple users.

The application of this approach is only useful for very small systems, which are only used by very few users.

Processing Without Locking Concept

This procedure is not advisable for multi-user systems, since the dialog processing can be disturbed by competing accesses, deadlocks, mutual overwriting, and so on.