Performance Considerations

This chapter covers the following topics:

- Introduction
- Data Processing and Pointers
- Initial Load and Reload the Database
- Using the Adabas Multifetch Feature
- Insert and Delete
- Splitting a DBD
- Field Definitions
- Enqueue Logic
- CICS/Batch Communication
- Application Program and DB Design

Introduction

The performance of DL/I applications depends on various factors:

- The environment: The hardware, how the data is physically stored, the operating system, the TP monitor, the number of users on the system, the number of users accessing the database management system, whether there are batch jobs running at the same time as online applications and more.
- The database structure: The `ACCESS` method used, the number of hierarchical levels, the complexity of the structure including logical relationships and secondary indices, the distribution of the database over different Adabas files and more.
- The application: The way it retrieves the data: direct (`GU` calls) or sequential (`GN, GNP` calls), the amount of data for insert, update or delete requests, the complexity of the SSAs, the fields used for qualifying the call and more.

Further on, there are some side-effects related to the performance which should also be considered:

- The disk space required to store the data.
- How often the database has to be reorganized.
- How flexible the database system is, that is, how much effort it takes to change the layout of the database.

- How the database system affects the loading of the TP monitor.
- What tools are available to debug performance-critical applications.

When the DL/I application runs against migrated data, two Software AG products are involved: ADL and Adabas. For the tuning of Adabas, you should refer to the appropriate Adabas documentations, such as, the *Adabas DBA Reference* documentation . It helps you to optimize your physical database layout (disk usage), the buffer pool management, the usage of Multifetch and more. You may use Adabas facilities, such as Review, Adabas Statistics Facility and AOS/Adabas Manager to debug performance-critical applications. You should always keep in mind that, from the Adabas point of view, your DL/I application is like any other application making Adabas direct calls.

The current section explains the tuning of ADL. It describes in detail how the performance is affected by

- direct and sequential data processing,
- the layout of the ADL internal pointer fields,
- unload and reload the data with ADL utilities,
- initial load of HDAM databases,
- Adabas Multifetch (Prefetch) feature,
- the Multifetch table (MFT parameter) and the Record Buffer Extension (RBE parameter),
- the last-call save area (LCS parameter),
- mass insert by user application,
- deletion of data,
- splitting the segments of a DBD over several Adabas files,
- Adabas Fastpath,
- the field definitions,
- the enqueue logic (ET parameter),
- the communication between Adabas and the CICS or batch region,
- the way your application accesses the data.

Data Processing and Pointers

Three DL/I commands are available to retrieve data:

- Get Unique ("GU") for direct access of specific data,
- Get Next ("GN") for sequential read of the whole database,

- Get Next in Parent ("GNP") for sequential read of a part of the hierarchy.

For the sequential read commands, DL/I supplies pointers which reflect the sequence of the processing:

- The hierarchic forward pointer which points to the hierarchical "next" segment, that is, the segment which will be retrieved at the next unqualified "GN" command.
- The physical child first and the physical twin forward pointers. These pointers are used when processing a "GNP" command.

For the direct access of data, DL/I treats the root segment differently from the others.

- The root segment is handled by the appropriate ACCESS method, for example, VSAM if this was specified at the DBD generation.
- Dependent segment data is searched by running through the pointers, that is, for each level, the physical child first pointer is used to get the first occurrence of the segment type wanted, and then the physical twin forward pointer is repeatedly used until the specified data occurrence is reached.

In order to reflect the hierarchical structure, ADL mainly uses one field, the ADL physical pointer field Z1 which is defined as a unique Adabas descriptor. In fact it is a "descriptor" rather than a "pointer" because it describes its own position instead of pointing to other data. In addition to the Z1 field, the root sequence field is defined as a descriptor too (the "root descriptor").

The layout of the ADL physical looks like:

Field	Parent Logical ID	Segment Number	Parent ISN	Segment Sequence Field	Appendage
Length	1	1	4	variable	variable

The Logical ID and segment number are each 1 byte long, the Parent ISN is 4 bytes long, the length of the sequence field is defined by the DBD definition and the appendage has a variable length. If the segment sequence field is not unique or does not exist at all, the appendage is generated to make the Z1 field unique. The Z1 field can be up to 126 bytes long. To minimize the space requirements, an appendage as short as possible is used.

Note:
The layout of the ADL physical pointer field has been changed with ADL 2.3. The former DBID (1 byte) and FNR (1 byte) have been replaced by the Logical ID (1 byte); and the parent ISN has been increased from 3 bytes to 4 bytes. ADL 2.3 uses only one layout of the ADL physical pointer where the segment number is in front of the parent ISN corresponding to the ADL 2.2 "SEQ=SEG" layout.

For the direct access of data, ADL treats the root segment differently from the others (like DL/I):

- On the root level, ADL makes use of the root descriptor.
- For dependent segments, the Z1 field is used.

For each level, an L3 or L9 call is issued, depending whether the data is wanted (for path call or lowest level) or not. The start value for these calls is built up for dependent segments from the already known Logical ID and ISN of the parent and from the requested segment number. If the SSA searches for a specific sequence field value (with "greater" or "equal"), this value is included in the start value too. In this case, the value on that level can be accessed directly; no run through a pointer queue is required.

Sequential reading of the root segment data is handled by sequential reading of the root descriptor. Sequential reading through the hierarchy cannot be translated into one Adabas sequence. For each (sensitive) segment, there is one Adabas sequence (one command ID). To read the first occurrence in a twin chain, ADL makes a value start call, where the sequence field part of the Z1 field in the value buffer is supplied with a minimal value.

To retrieve the next occurrence in the twin chain, ADL continues reading in the Adabas sequence. Since there is no information which says how many records are in the twin chain, ADL has to check whether the retrieved record belongs to the current twin chain. This is done by checking whether the Logical ID, segment number and ISN match those requested. This has the effect that ADL reads one record too many at the end of a twin chain. This additionally-read record is used by the last-call save area as described later.

Initial Load and Reload the Database

The data is initially loaded to the ADL files with the Adabas utilities `ADACMP` and `ADALOD`. The physical sequence in which it is loaded is defined by the unload utility, usually `DAZUNDLI` (see the section *ADL Data Conversion Utilities* in the *ADL Conversion* documentation for more information). This utility reads the data in the hierarchical sequence, that is, for each root, it reads all dependent segments before it accesses the next root. Therefore, the loaded data is in the hierarchical sequence too. This means that, if a root segment is on one data block, then the dependents will be on the same block, the next root also, and so on, depending on how many records can be stored on one data block.

An application which reads the data in the hierarchical sequence can therefore issue many DL/I calls before the next physical I/O is required. This is, of course, only true as long as the data is in the hierarchical sequence. When new data is inserted and old data deleted, it comes more and more out of that sequence. If you notice a considerable decrease in performance, you should think about restoring the database. Restoring with Adabas utilities makes no sense, because Adabas has no idea about the hierarchy which is inherent in your data. The easiest way is similar to the initial load: unload the database with `DAZUNDLI` and reload it with `ADACMP` and `ADALOD`. In this case, `DAZUNDLI` runs as a normal (that is, not mixed mode) application, as described in the section *Managing ADL Files* in this documentation. This brings the data back to the hierarchical sequence.

Additional considerations should be made when initially loading a randomized database, such as, an HDAM database. Root segments are retrieved by DL/I in the randomized sequence. When reading with ADL, they are accessed in the sequence of the root sequence field. If these two sequences do not match, you may notice poor performance for sequential reads. In this case, you should load the data in the root key sequence, as described in the section *Unloading a HDAM Database* in the *ADL Conversion* documentation.

The unload utility, `DAZUNDLI` (or `DAZREFOR`), also assigns the Adabas ISNs which are used when the data is loaded into the Adabas file. Therefore, after an initial load or restoring of the data, the ISNs reflect the hierarchical sequence too. The sequence of the ISNs is only important when the last-call save area (LCS) is used (described below). This feature works best when the ISNs are in the hierarchical sequence. Thus, if the LCS is used, restoring of the data with ADL utilities should be considered if the ISNs are out of the hierarchical sequence for the most part.

Using the Adabas Multifetch Feature

The Adabas Multifetch feature reduces the communication overhead between the application program and the Adabas nucleus for sequential reads. See the *Adabas Operations* documentation for more information. The Multifetch feature is similar to the Prefetch feature. If not otherwise stated, we use Multifetch as a synonym for Multifetch and Prefetch.

The Multifetch feature is activated for batch applications by specifying the `ADARUN PREFETCH=YES` parameter. The number of sequences (command IDs) which are multifetched is defined by the keywords `PREFTBL` (total buffer length) and `PREFSBL` (single buffer length), exactly `PREFTBL` divided by `PREFSBL`. This number has to be evaluated very carefully. Multifetching "bad" sequences can considerably decrease the performance. For example, an L3 sequence with many value-restarts ("V" option) is bad, because Multifetch throws away the data of the corresponding buffer at every restart. The Multifetch buffers are occupied by the sequences which are used first, that is, the first CID uses the first buffer, the second the next one, and so on. You can exclude specific command/file number combinations from Multifetch by specifying the `PREFXCMD` and `PREFXFIL` keywords. Note that an Adabas OP (open) command refreshes all the buffers.

The ADL parameters `RBE` and `MFT` can be used to restrict the number of multifetched records for a specific `PCB/SENSEG` combination. (See the section *ADL Parameter Module* in the *ADL Installation* documentation for a detailed description.) Here we have to distinguish between Multifetch and Prefetch. The `MFT` can only be used with the Multifetch feature and, in this case, you are recommended to use it instead of the `RBE`.

With the Multifetch Table (MFT), you can explicitly specify the number of multifetched records. For an unspecified `PCB/SENSEG` combination, the maximum number of records will be multifetched. If you want to multifetch not the maximum number but a smaller amount, you should specify an MFT entry for this PCB/SENSEG combination. Because of the double buffering of Multifetch, Adabas will return twice as many records as you specify. To minimize the amount of returned data for a specific sequence you should specify a value "1", which will result in 2 returned records.

With the Record Buffer Extension (`RBE`) parameter, you can increase the record- buffer length for specific `PCB/SENSEG` combinations. This has the effect that fewer records fit into the the ISN buffer for the corresponding Prefetch call. Thus, the fewer records you want to prefetch, the bigger the corresponding `RBE` entry should be. To retrieve the maximum number of records for a specific `PCB/SENSEG` combination, you should omit the corresponding `RBE` entry.

The use of Multifetch for migrated data requires knowledge of the application and of how ADL translates the DL/I calls into Adabas calls. With the ADL trace facility, you can obtain this information (see the section *Debugging Aids - ADL Trace Facility*. It lists you the DL/I calls of the application and the resulting Adabas calls. From this list, you can see, for example, that a GU call is translated into an Adabas "L3" with value-start. Therefore the corresponding CIDs are bad candidates for Multifetch. On the other hand, if a root segment is read sequentially (GN calls), there is one value-start at the first call and then no other restart. Therefore, you can multifetch this sequence.

With the last-call save area (LCS), the record last retrieved is saved. This feature can be activated by specifying the LCS parameter, as described in the section *ADL Parameter Module* in the *ADL Installation* documentation. If an application program reads the same record twice or more, only one Adabas call will be issued by ADL because it finds this value in the LCS. But the main importance of the LCS is when reading sequentially through dependent segments.

As described earlier, ADL makes a value-start call at the beginning of each twin chain. Then it reads sequentially through the chain until it finds one record that does not belong to the chain. Now, under certain circumstances, it can happen that this additionally read record is the first record of the next twin chain of the same segment type. In this case, the LCS preserves ADL for reading the same record twice. Since this saved call was the call with the value-start, a sequential read (GN or GNP) through a dependent segment type is translated into sequential Adabas reads without intermediate restarts. Therefore, you can use the Multifetch feature for dependent segments too. The remaining questions are: which record does ADL find after the end of a twin chain and under which condition is this record the first of the next twin chain for the same segment type? The sequence in which ADL retrieves the data is determined by the layout of the ADL physical pointer fields, which is described earlier in this section.

The Z1 physical pointer field sorts the data in the sequence "segment number / parent ISN". In general, the record found after the end of a twin chain is from the same segment type with the next higher parent ISN. If this ISN belongs to the next parent, the record found is the first of the next twin chain. Therefore, the ISNs should match the hierarchical sequence. This is true after an initial load of the data or after a restore. If the ISNs are mainly out of the hierarchical sequence, you should reestablish the hierarchical sequence as described in the section Managing ADL Files in this manual. If you do not want to reestablish the hierarchical sequence by any reason, adjust the number of multifetched records with the MFT or RBE parameter or avoid multifetching those sequences at all.

Summary

- Do not use Multifetch for direct reading ("GU").
- Use Multifetch for sequential reading of root-segment data.
- Use the last-call save area (LCS) and Multifetch for sequential reading of dependent data.
- Do not use Multifetch for sequential reading of dependent data if you do not use the last-call save area (LCS).
- Adjust the number of multifetched/prefetched records with the `MFT` or `RBE` parameter, respectively.
- Check with the ADL trace facility or with Adabas utilities whether the data and ISNs are still in the hierarchical sequence. If required, restore the database with ADL utilities, so that the ISNs match the hierarchical sequence again.

Insert and Delete

A DL/I insert request (`"ISRT"`) is translated by ADL into an Adabas N1 call. A special case is given when loading the database. For the initial load during the conversion process, the data which have been unloaded by ADL utilities are loaded into Adabas with the Adabas utilities `ADACMP` and `ADALOD`. `ADACMP` requires the ADL user exit `DAZUEX06`.

Another way of initially loading data is using insert calls by a user application against a PCB with `PROCOPT=L`. This insert call must be qualified with one SSA which specifies the segment to be inserted but no other qualification. The data must be inserted in the hierarchical sequence. How such an insert call is treated by ADL depends on the setting of the `LOAD` parameter.

If LOAD=DIRECT is specified, an Adabas N1 call is issued, as with any "normal" insert against a PCB with PROCOPT=I.

If LOAD=UTILITY is specified, the data is written to a sequential file. It has the same layout as the data generated by the ADL data conversion utilities. It is loaded into Adabas with ADACMP (with the ADL user exit) and ADALOD.

Which of these two methods should be used depends on the amount of data to be loaded. If it is only a small amount of data, the easiest way is to load it directly into Adabas. If, however, there is a lot of data to be loaded, you are recommended to use the Adabas utilities, because the utilities are considerably faster than a direct insert.

A DL/I delete request ("DLET") is translated by ADL into Adabas E1 calls for the specified segment occurrence and all its dependents. Normally, DL/I does not really delete the data; it sets a flag which indicates that the record is "deleted". When retrieving the data, DL/I has to run through such "deleted" data, because the pointers are still there. Therefore, the DL/I database has to be reorganized to get rid of that garbage and to release the corresponding storage.

When Adabas deletes a record, the storage is released immediately. This takes more time than just setting a flag and you may notice an increased time, especially for mass deletes by user applications. On the other hand, there is no running through "deleted" data and no requirement for a reorganization to release the storage.

Splitting a DBD

With the GENSEG statement of the ADL CBC utility, you can distribute the segments of a DBD to multiple Adabas files.

The following reasons for splitting a DBD are related to performance.

- If the application accesses some segment types and not others, splitting will reduce the number of physical I/Os. This is because more of the accessed data fits on one data block, since the information not requested is no longer stored with it.
- If some of the segments contain constant data like tables, which are often read but not updated, you can use Adabas Fastpath especially for these files.
- For some requests, you can use Adabas utilities if the segment is separated. For example, if you want to delete all the data of a dependent segment type, you can refresh the corresponding Adabas file.

But splitting a DBD can also have disadvantages:

- The initial load will consume more time, because the ADL user exit always has to process the whole DBD data.
- If your application accesses segments that reside on different Adabas files, it probably requires more physical I/Os. If it reads, for example, one root segment occurrence and all its dependents, at least one I/O is performed for each Adabas file. As long as your application reads sequentially through the data, this will not lead to any problem. This is because Adabas will find the subsequent data in the buffer pool and therefore, the total number of I/Os will not increase. If, however, it reads the data out of sequence, the number of I/Os is multiplied by the number of Adabas files accessed.

Field Definitions

The fields are primarily defined in the DL/I DBD source. As described in the section *Conversion of the Data Structure - General Considerations* in the *ADL Conversion* documentation , you can change the default Adabas field specifications before or during the conversion process. One reason for such a modification can be better compression. The more the data is compressed, the fewer I/Os are required. However, the compression itself also needs some time. Therefore, if you add new field definitions for "better" compression, but, in fact, there is nothing to compress, you keep Adabas occupied with unnecessary work

If a Natural program accesses all the data of one segment, the corresponding format buffer (FB) contains all Adabas fields that build up the Adabas group related to the segment. More field definitions require more time for the format buffer translation. This is especially the case if the call has to be handled by the ADL Consistency Interface, since the format buffer translation of ADL is not as sophisticated as of Adabas.

Enqueue Logic

Batch applications which run in multi-user or shared-database (so called "online batch") mode should supply their own enqueue logic. Nevertheless, you can run a program in online batch, although it was originally designed to run in normal mode. ADL guarantees the integrity of the data by putting each accessed root record into hold. If the application does not make checkpoints, the Adabas hold queue will soon be full. For this situation, ADL offers the automatic-ET feature, which is controlled by the ET parameter described in the *ADL Installation* documentation , section *ADL Parameter Module*. Because each ET call consumes some time, you should set the ET parameter to a value right below the critical value where an Adabas hold queue overflow would occur.

Under CICS, ADL puts each accessed root record into hold, as in online batch. It is released as soon as the next root record is accessed if there was no update on that hierarchy. Here, the ET parameter helps you reduce the number of RI calls. The records not released remain in hold status until the next syncpoint or terminate call is issued. This may lead to situations where other users have to wait for such records or, fatally, to deadlock situations. If, however, your application avoids such situations, you can adjust the ET parameter to save the time for the RI calls.

CICS/Batch Communication

Usually, DL/I runs in the same region/partition as the user application. This saves interregion communication. On the other hand, it loads the online system, especially when big buffers are required on smaller machines. In addition, if batch applications access online databases, the CICS system has to forward the calls to the "online" DL/I. Adabas runs in its own region/partition (if not in single user mode), Therefore, you have the overhead of the interregion communication, but the online system is relieved of the database management system and, of course, no batch request must be handled by CICS.

Application Program and DB Design

A good programming style is a general task and not directly related to a conversion. However, some problems become obvious when you are using features of ADL or Adabas. If, for example, you see one DL/I call in the ADL trace which is translated into myriads of Adabas calls, you should not hesitate to take a look into your application program or into the DB design. Changes in this area would help DL/I as

well as ADL. In addition, if your application makes one million calls too many, you can tune your database system as much as you like, but you will never get rid of these superfluous calls in this way.

The examples that follow are far from complete. Most of them have been seen on real sites. They should give you an idea of what can be done wrong and how to make things work better.

Examples

Non-descriptor Search

Description:	In a qualified SSA, a normal field is used for searching, that is, no sequence field and no secondary index field.
Effect:	This is a "non-descriptor search". For a root segment, the range of the search extends from the first record to the end of the database. For dependent segments, it is restricted to the children of one specific parent occurrence. The search stops if a record fulfils the qualification; otherwise, it runs until the end of the range is reached.
Action:	Use, as far as possible, the sequence field or an already existing secondary index. Otherwise, define a new secondary index which specifies the field from the SSA as search field.

Wrong Key

Description:	In a qualified SSA for a root segment, the primary key is used, but the PCB specifies an alternate processing sequence.
Effect:	This is also a non-descriptor search, as in the previous example. If a `"PROCSEQ"` statement is specified in the PCB definition, the only usable key field is the corresponding secondary index field. All the other fields, including the primary key, are treated like non-descriptors.
Action:	Use a PCB without an alternate processing sequence.

Second-level Qualification

Description:	The SSA specifies two levels: the first level is not qualified but the second level is qualified.
Effect:	This is a non-descriptor search on the first level. It stops when the correct second-level segment is found or at the end of the database.
Action:	Define a secondary index on the second-level segment and use this in the call.

Retrieve same Data

Description:	The application program issues a path call over two levels. The first level always specifies the same segment occurrence.
Effect:	The first-level segment occurrence has to be read for each call.
Action:	Once the first level is read, avoid reading it again. You can specify only the lowest-level segment or make no path call.
Note:	If you use the last-call save area (LCS), the record is saved in the LCS buffer. For the repeated read, the value of the LCS is taken and the overhead is minimized.

Concatenated Segment

Description:	In a concatenated segment, the sequence or any other field of the destination parent is used for the qualification in the SSA.
Effect:	This is a non-descriptor search.
Action:	The only key field you can use for concatenated segments is the sequence field of the logical child segment.

No Sequence Field

Description:	A qualified SSA is used for a dependent segment, but this does not have a sequence field.
Effect:	This is a non-descriptor search.
Action:	Define a (non-unique) sequence field for the field used in the SSA.

Read in Sequence

Description:	The application reads the data in the sequence of the sequence field, but it uses qualified direct reads, such as: GU ROOT (SQF = 1) GU ROOT (SQF = 2) GU ROOT (SQF = 3) This can be, for example, in an HDAM database, where the randomized sequence does not match the key-field sequence.
Effect:	Each call is translated into an Adabas call with value-start. Multifetch can not be used. In addition, if there are gaps in the data, a lot of unnecessary calls are issued.
Action:	Read the data with GN calls and an unqualified SSA, such as GN ROOT Under this condition, you can use Multifetch. This also works for HDAM databases, since ADL always returns the data in the sequence of the sequence field.

Segments without Data

Description:	An application makes unqualified `GN` or `GNP` calls and some of the sensitive segments in the PCB are never supplied with data.
Effect:	, one Adabas call is issued, in order to realize that there is no data of that segment type under this parent.
Action:	Use qualified calls or do not specify the sensitive segments in the PCB.
Note:	If you use the last-call save area (LCS), the next record is saved in the LCS buffer. When reading the empty segment, ADL realizes from the value saved in the LCS that there is no data for this segment and the overhead is minimized.