Converting a PRO*IV* Application from Pro-ISAM to an RDBMS

Internet: http://www.proiv.com

THE AMERICAS
PRO*IV* Technology, Inc
101 Academy, Suite 200
Irvine, CA 92612
USA
Tel: +1 (949) 823-1000
Fax: +1 (949) 823-1010

EMEA
PRO*IV* Ltd.
Kings Hall
Parsons Green
St Ives
Cambridgeshire
PE27 4WY
UK
Tel: +44 1480 494330
Fax: +44 1480 494039

Document Amendment

Version	Date	Author	Description
1.0	07/03/03	R Gadsby MIMIS	Initial document detailing the procedures needed to convert a PRO*IV* application to an RDBMS.

Table of Contents

Overview

The intention of this document is to describe several methods for converting an existing application written in PRO*IV* using Pro-ISAM files to use Oracle, SQL Server, or other RDBMS tables. This document deals solely with the PRO*IV* coding side of the conversion. It is assumed that the user will acquire the knowledge to install, set up, tune and administer their chosen database system.

Some of the stages in the process of conversion can be automated. If you need help with this, or you would like PRO*IV* Ltd to convert your system for you, you should contact your account manager.

Three methods for conversion will be described.

1. We will start by describing a low impact conversion route. This method seeks to provide the quickest route to get an application up and running using an RDBMS whilst maintaining compatibility with a system running with a Pro-ISAM filing system. Note, however, that this route takes no advantage of features available with an RDBMS. It is likely that, without an increase in the specification of the system running the PRO*IV* kernel, a noticeable decrease in performance will be experienced.

2. The second method will describe the correct method of conversion in order to achieve the greatest benefit from the decision to convert to an RDBMS.

3. The third and final method describes a staged method of converting to an RDBMS. The initial stage will lead to a slight degradation in performance. Each subsequent stage will gradually improve the performance.

Potential Issues

When converting from Pro-ISAM to an RDBMS you are likely to experience various issues if the conversion is not fully analysed and completed. Amongst the issues that you may experience are:

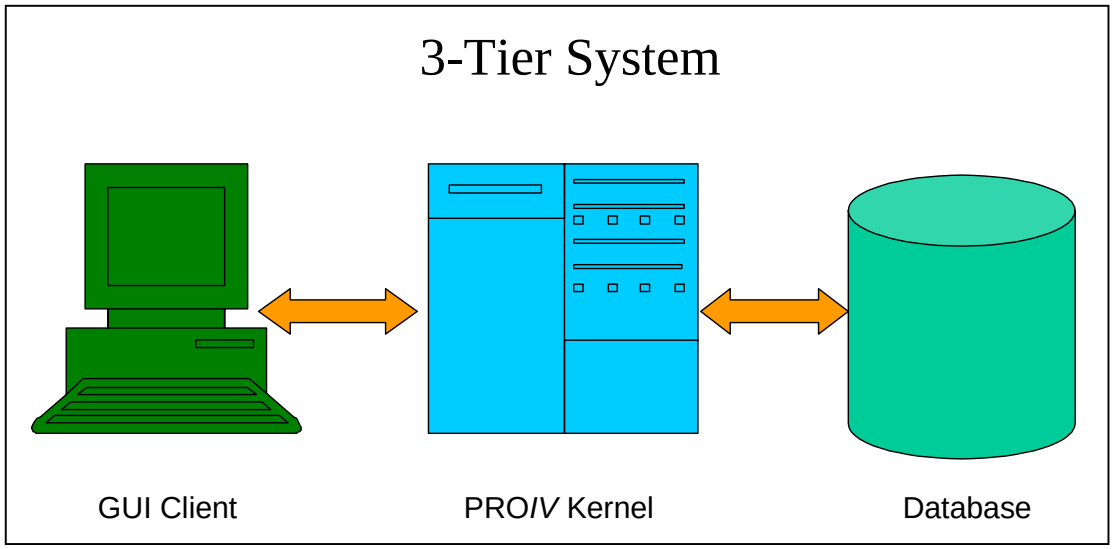
1. Increased system hardware requirements. In order to achieve the same performance level as Pro-ISAM more resources are required.
2. Locking problems. Some files will be locked for long periods of time, reducing the throughput of the application.

The following sections of this document will show ways to maintain the performance of your system and still reap the benefits of converting to an RDBMS.

Benefits of an RDBMS

Using a relational database a unit of work is defined as a transaction. If this transaction does not complete then all data updates belonging to that transaction will be 'rolled back'. With ISAM storage systems, which do not have rollback capability, the code tends to be written such that it processes work files to ensure that updates only occur at the end of a transaction. The use of an RDBMS therefore reduces the number of files that are required in a system and also allows better control over data integrity. With ISAM systems where data is updated at the end of the transaction data integrity can still be compromised if the update function fails for any reason.

Because an RDBMS supports a transport mechanism, for remote access of the data, the PRO*IV* kernel can be on a separate computer from the database. This allows for additional security of the data. This separation of the presentation (GUI client), business rules (PRO*IV* kernel) and database is known as a 3-tier system. Each layer of the 3-tier system can be modified easily without seriously impacting the other 2 tiers. For example, it is possible to change the database from Oracle to SQL Server with the only requirement being a change to the pro4.ini and the creation of an ODBC configuration entry on the PRO*IV* kernel tier.



Using an RDBMS the complexity of an application is reduced. This is brought about by, for example, the removal of the need to maintain cross-reference files. It is also possible in some cases to replace a complete PRO*IV* function with a single SQL statement.

Using an RDBMS increases the accessibility of the data to third party tools. Writing well-presented reports in PRO*IV* is not easy. By using an RDBMS it is possible to use industry standard reporting tools, such as Cognos Impromptu or Seagate Crystal Reports, which can be run by users with little or no knowledge of computer programming.

Low impact conversion route

This method of converting from Pro-ISAM or C-ISAM to an RDBMS should only be considered if it is essential to maintain compatibility with non-RDBMS installations of the same system. Using this conversion methodology none of the advantages of using a RDBMS will be used and a significant performance overhead will be experienced. The performance problems can be worked around by using a faster processor and installing more RAM than would be needed to achieve the same throughput had the system been implemented using a non-RDBMS file system.

This conversion method requires minimal changes to the application. The changes required can be automated easily, if access to the bootstrap file definitions is available. All that needs to be done for this conversion route is to change the file definitions to specify the external file and field types and then recompile the affected PRO*IV* functions.

Ideal conversion route

In an ideal world, where there are no deadlines and there is no limit on the resources that can be employed, this is the method that should be employed to convert a flat filing system to an RDBMS. This route can be broken down into three phases: analysis of the data items in the existing file system; re-engineering of the application functionality to take account of the new data structure; and conversion of data from Pro-ISAM to the RDBMS.

In effect this is a rewriting of the application, using the existing application as the basis for the design specification. However the business rules from the existing application will be re-used in the new application.

Data analysis

In order to achieve the best results from the use of an RDBMS the data needs to be normalized. The data items can either be analysed from the top downward or the bottom upward. A typical PRO*IV* application written to use either Pro-ISAM or C-ISAM will not use fully normalized data. Typically there will be file definitions that contain field arrays or there will be data that relates to the same key information spread across multiple file definitions.

The separation of arrays on PRO-ISAM files to individual tables in the RDBMS can significantly improve performance. I have seen an application that was moved from PRO-ISAM to Oracle without data analysis where one file had an array containing around 200 elements. The resultant table in Oracle had over 300 columns. Each time the table was accessed an SQL statement was created that contained over 10,000 characters. The table was used in a major update, where it was accessed thousands of time, resulting in a significant volume of traffic over the network between the PRO*IV* kernel and the Oracle database. By creating a separate table for the information contained in the array it would have been possible to access just the individual elements that were required for the update, instead of retrieving and then writing back every single element.

There are three stages to data normalisation.

1. Remove repeating groups → 1st Normal Form
2. Remove partial dependencies → 2nd Normal Form
3. Remove indirect dependencies → 3rd Normal Form

A system that uses Pro-ISAM files will also probably use temporary work files to store the data being entered or updated prior to actually updating the master files. With an RDBMS it is not necessary to use work files because the data can be rolled back if an error occurs or the user wishes to cancel the current operation.

Top down analysis

In the top down analysis methodology the items that uniquely identify groups of data are established first. Having found these unique identifiers the data items required to define the group need to be allocated.

Bottom up analysis

In the bottom up analysis methodology the data items required are first identified. Having done this the identifier that uniquely defines the occurrence of the item is established. All items that use the same identifier are then grouped together to form a table.

Re-engineer application

Having established the normalized form of the data and created the tables with their primary index constraints in the RDBMS it is now necessary to change the existing PRO*IV* functions to use the new structure of the data. The amount of effort required to effect this modification depends to a large degree on whether the original Pro-ISAM file system was normalized or not and whether master file updates were delayed to the end of the transaction.

Remove cross reference files

The first stage is to remove the maintenance of cross-reference files from the functions, as this is not necessary with RDBMS. Cross-reference files are used to access flat file systems using alternative key sequences. Typically the file would contain the alternate access fields and the key fields to the main file. When retrieving the data the PRO*IV* function would first read the cross-reference file and then access the main data file. In an RDBMS this is achieved, where necessary, by adding additional indexes to the data table and creating a PRO*IV* file definition specifying the alternate key fields and all the data fields of the main data file. This definition would have the table name in the alternate file name field of the file definition header.

For example, in a Pro-ISAM system, to be able to access a customer file in postcode sequence efficiently you would need to create two files, as follows:

CUSTOMER		CUSTPCOD	
K	CUSTCODE	K	POSTCODE
A	CUSTNAME	K	CUSTCODE
A	ADDRESS		
A	POSTCODE		

Using these file definitions to list the customer names in postcode sequence you would have to first read CUSTPCOD and then read the CUSTOMER file. In the customer maintenance function you would have to delete the record from CUSTPCOD and then add a new record if the postcode for a customer was changed. When we convert this to an RDBMS we would create the following PRO*IV* file definitions:

CUSTOMER		CUSTPCOD	
K	CUSTCODE	K	POSTCODE
A	CUSTNAME	K	CUSTCODE
A	ADDRESS	A	CUSTNAME
A	POSTCODE	A	ADDRESS

We only need to create the table in the RDBMS for the CUSTOMER file. The CUSTPCOD file would have the alternate name set to CUSTOMER. When we list the customers in postcode sequence we now only need to access the CUSTPCOD file. When maintaining the CUSTOMER file we now no longer need to delete records from and add records to CUSTPCOD, as this done automatically for us by the RDBMS.

Remove long locks

Next, commonly accessed records that will suffer from long lock problems under an RDBMS have to be identified. A typical example of this is an odometer file (a control file used to generate the next available sequence number). In a flat file system odometer files would normally be accessed by an LSCALL or global LSCALL that would read the file to retrieve the current value, increment it and then write the value back. The record would then be available for another user to access, whilst the rest of the transaction was completed. With an RDBMS the information on the updated odometer record does not become available for use by others until the transaction is committed and will lock the other users out until the commit is executed. For this reason odometer and other commonly accessed files should either be moved outside of the transaction, or should be moved to near the end of the transaction and after all user input has been completed.

Another way that records can be locked for an extensive period of time is in paging screen cycles when the primary file is in change or both mode. In this situation each record that is read when paging down will be locked until the transaction is committed. To prevent records that have not been changed being locked a file read in look up mode should be inserted as the primary file for the cycle.

Replace simple updates

Some logical updates/cycles can be replaced with SQL statements. A typical example of this is an update that selects a group of records and then modifies one or more fields, e.g. select all employees for a company that have been employed for 3 years and increase their holiday entitlement by one day. For a non-RDBMS this would be achieved by using a SEL-ONLY on the company and then DSELing any employee that has been employed for less than 3 years or more than 4 years. The holiday entitlement for the remaining records can then be incremented. To achieve the same result in an RDBMS a single SQL statement could be used. The following example statement assumes that start-date is held using the PRO*IV* internal date format and ignores leap years:

```
#DATE = @DATE
SQL
   UPDATE employee SET holiday = holiday + 1 WHERE company = '005' and
   start-date - :#DATE BETWEEN (3 * 365) AND (4 * 365)
ENDSQL
```

Another example of a logical update that can be replaced with a single SQL statement is the kind that is used to count, or total up the value of one or more fields on, a group of records. To achieve this using SQL a dummy PRO*IV* file definition is required to contain the results of the SQL statement. This file should then be specified as a secondary file in a cycle and the before read logic for the file should contain the required SQL statement, e.g.

```
SELECT COUNT(*) FROM INVLINE WHERE INVNO = :$INV_NO
```

or

```
SELECT SUM(TRANSACT-AMOUNT), SUM(DISCOUNT-AMOUNT) FROM INVLINE WHERE INVNO =
:$INV_NO
```

The results of the SQL is accessible in the function using the fields from the PRO*IV* file definition, which must match the result column definition of the SQL statement in terms of type and must also be large enough to contain the results.

The dummy file definition is defined as an external file type for the chosen database. All fields on the file definition must have the appropriate external data type set. However a table does not need to be created in the database for the file definition.

Replace sort select logics

Wherever possible the use of sort/select logic to reduce the number of returned rows from a table should be avoided. It is far more efficient to avoid selecting the rows in the first place, by using additional clauses on the SELECT statement in the default logic of the cycle. For example, the following two logics can be replaced with one SELECT statement that would be far more efficient.

Default logic

```
COMPANY = $COMP-SEL
```

```
SEL-ONLY(COMPANY)
```

Sort/select logic

```
IF EMP-STATUS = 'D','X'
   DSEL
ENDIF
IF EMP-PAYPER = 'W'
   DSEL
ENDIF
```

Can be replaced with the following Default logic

```
SQL
   SELECT FROM EMPLOYEE WHERE COMPANY = :$COMP-SEL
   AND EMP-STATUS NOT IN ('D','X') AND EMP-PAYPER <> 'W'
ENDSQL
```

Analyse transactions

In a typical application there are two general types of transactions. The first type is interactive, takes a relatively long time to complete and may span multiple PRO*IV* functions. The second type is non-interactive, takes a relatively short time to complete and occurs multiple times in a single function.

The first of these transaction types may require the default PRO*IV* commit action to be switched off, particularly if the update of the files within the transaction is spread over several functions. This is effected by issuing ENABLE(&#@SUPP-COMM) in a logic prior to the start function of the transaction. The transaction begins when the first file access occurs, so the logic can be in the same PRO*IV* function. If this is not the only transaction for which the suppress commit flag is being enabled it is a wise precaution to issue an explicit commit first. When the transaction is complete the DISABLE(&#@SUPP-COMM) command should be issued. This will cause the transaction to be committed when PRO*IV* gets to the next natural commit point.

N.b. care must be taken when suppressing the PRO*IV* default commit behaviour because it is possible to lose all, or some of, the work performed in a session if the data is not committed prior to logging out of PRO*IV*.

The second, non-interactive, type of transaction can cause a large number of table rows to be locked from access by other users. It can also lead to heavy disk usage by the database in order to store the roll back information prior to committing. Additionally, if an error occurs, all the updates within the PRO*IV* function will be rolled back, which may take some considerable time. In order to reduce the amount of uncommitted data that has to be held by the database in its rollback segment transactions should be committed when complete, or after a certain number of transactions have been completed. There are three methods for committing data being processed by a single PRO*IV* function.

1. Use the iteration counter (Oracle only).
2. Exit the function on a regular basis.

3. Issue explicit commits, e.g. #X = COMMIT().

Using the iteration counter is a quick and easy way of committing transactions periodically. The counter can be set using a numeric literal, or by using either a numeric scratch or file variable. Use of this method on a RDBMS that uses cursors to maintain position on a table will lead to the database returning errors relating to non-existent or deleted cursors. Consequently this should only be used for Oracle.

The use of explicit commits suffers from the same problem as use of the iteration counter, with the exception that it is possible to code your way round it. When an explicit commit is issued all data currently in the rollback segment will be committed to the database and any associated cursors will be released. This action of releasing the cursors leads to problems with databases other than Oracle because when the driving file for the update is next read the cursor will no longer be available, so an error message will be issued by the database indicating that the cursor is no longer available. This problem does not occur with Oracle because we use OCI (Oracle Call Interface) to access tables and the individual rows are read using the ROWID. To overcome the problem of losing the cursors you should exit the PRO*IV* function, linking back to the same function and positioning such that it carries on where it left off. The easiest way of continuing an update after exiting is to have the function driven by a trigger file, where the records are deleted as they are processed.

It is possible to enable an environment setting to generate an error if a commit is executed in the wrong place. The setting, which should be in the environment stanza of the pro4.ini file, is as follows:

```
SQL_TRANSACTION_ERROR=Y
```

This will generate PRO*IV* error message 356 if a commit is issued in an incorrect location. This error message will be output if the iteration counter or explicit commit is used with Oracle within a transaction, even though the transaction could have completed successfully due to the use of ROWID and OCI.

N.b. the use of explicit commits can lead to unexpected results if they are issued at points within the file access part of the PRO*IV* timing cycle. For example, if you were to issue an explicit commit in the after read no error logic of an RDBMS table the read lock will be released, allowing another session to commit its pending changes. When the first session now performs the update part of the PRO*IV* cycle the updates will be based on the data read prior to the update by the other session. Consequently the data written to that table by the second session will be lost.

The use of the iteration counter with Oracle or explicit commits between file accesses can also cause problems if another process is updating rows within the group of rows selected. For example, consider the following: a PRO*IV* update function is used to process records in an RDBMS auto sequenced file with either an iteration counter set at 5 or a logic in after write no error that issues an explicit commit after every 5 records written. At the same time that this PRO*IV* function was started another process was started that deletes sequence numbers 16 to 19. Initially this second process is locked out by the first one, however when the commit is executed after writing 5 records the second process will be able to delete the records. The first process will then continue until it gets to sequence number 16, at which point no more records will be processed, due to the way that auto sequenced files are handled.

Staged conversion route

This is the recommended route for conversion of an application in a live environment where time constraints preclude the use of the ideal conversion route. The intention of this route is to quickly achieve a workable system using an RDBMS and to then gradually apply more of the RDBMS specific modifications.

This section will describe 3 stages that can be used to gradually produce a system that is well tuned to the database. In order to reap all the benefits of moving to an RDBMS all stages described here should be completed. Completion of the first stage of the conversion will allow an application to be deployed using an RDBMS in a very short time.

Stage 1

Stage 1 will get an existing application up and running using an RDBMS in the shortest possible time.

The first step to the staged conversion process is converting your file definitions to have the external record format. To do this set the EXTERNAL RECORD FORMAT flag to Y and the FILE TYPE to the appropriate value for your chosen database. The external type and storage format also needs to be set on each field within the file. It may also be necessary to specify alternate field names if the field name used in the PRO*IV* file definition is a reserved name in the chosen RDBMS, or does not conform to the naming standards of the RDBMS. For example, in SQL Server the word 'KEY' is a reserved name. If you had a field name in the PRO*IV* file definition called this you should add an alternative name field, accessible in the native development environment using the expand key. For further information on this subject see the appropriate section of the *ENVIRONMENT GUIDE* for your chosen database.

Next you should locate any places where commonly used records are locked for an extended period of time. Where these are found the accessing of the record needs to be moved as close as possible to the end of the transaction. As an alternative it may be worth considering leaving the file as a Pro-ISAM file, though this will not then be rolled back if the transaction fails. Paging screen cycles where the primary file is in change or both mode can cause records to be locked until the transaction is committed. A second access of the file should be inserted in look up mode as the primary file to prevent records that have not been changed being locked unnecessarily.

An alternative method that can be employed for odometer type files with Oracle is to use a *sequence*. Sequences are Oracle objects that can be used to generate unique consecutive numbers and are not affected by transaction processing. To create a new sequence use SQLPLUS and type a command similar to the following:

```
create sequence  sequence_name increment by  x  start with y
```

Typically x would be 1 (one) and y would be 0 (zero).

If you then put the following code in the before read of a dummy file definition created to receive the results the sequence will be incremented and returned.

```
SQL
   SELECT  sequence_name.nextval  FROM DUAL
ENDSQL
```

Similar techniques to update odometer files outside of the main transaction may exist with other RDBMS. It is up to the user to determine if a suitable method is available for their chosen RDBMS. Use of database specific features will make it more difficult to switch from one RDBMS to another at a later stage.

Stage 2

Stage 2 will take the system that is now running using an RDBMS and implement various changes to improve performance.

The first step of the second stage is to remove the unnecessary maintenance of cross-reference files. The cross-reference file definitions themselves should be retained and the alternate name set the file that they are cross-referencing. If the cross-reference file contains just the key information and is used as the primary file in a cycle, with the main file then accessed to obtain the information to be processed, the cross-reference file definition will need to be changed to include the appropriate fields and the second file access removed. This could be a very time consuming process to analyse and implement.

Next you should look for cycles that have sort/select logics that deselect records or conditional deselects in the after read logic of the primary file of a cycle. These should be removed and the default logic of the cycle changed to use full function SQL with additional WHERE clauses to remove the unwanted records from the returned set.

The final step to this stage is to look for any cycles that can be replaced with single SQL statements. A typical example of this is an update that selects a group of records and then modifies one or more fields, e.g. select all employees for a company that have been employed for 3 years and increase their holiday entitlement by one day. For a non-RDBMS this would be achieved by using a SEL-ONLY on the company and then DSELing any employee that has been employed for less than 3 years or more than 4 years. The holiday entitlement for the remaining records can then be incremented. To achieve the same result in an RDBMS a single SQL statement could be used. The

following example statement assumes that start-date is held using the PRO*IV* internal date format and ignores leap years:

```
#DATE = @DATE
SQL
   UPDATE employee SET holiday = holiday + 1 WHERE company = '005' and
   start-date - :#DATE BETWEEN 1095 AND 1460
ENDSQL
```

Another example of a logical update that can be replaced with a single SQL statement is the kind that is used to count, or total up the value of one or more fields on, a group of records. To achieve this using SQL a dummy PRO*IV* file definition is required to contain the results of the SQL statement. This file should then be specified as a secondary file in a cycle and the before read logic for the file should contain the SQL statement, e.g.

```
SELECT COUNT(*) FROM INVLINE WHERE INVNO = :$INV_NO
```

or

```
SELECT SUM(TRANSACT-AMOUNT), SUM(DISCOUNT-AMOUNT) FROM INVLINE WHERE INVNO =
:$INV_NO
```

The results of the SQL are accessible in the function using the fields from the PRO*IV* file definition, which must match the result column definition of the SQL statement in terms of type and must also be large enough to contain the results. The file definition must be defined as an external with the file type set to the appropriate RDBMS type, however the file should not be created in the database.

Stage 3

This final stage is make sure that the transaction boundaries are correctly set to maintain data integrity in the event of a problem with the update of the data. Depending on the design and complexity of the application the analysis of transactions could be a very time consuming process. In order to disable the default commit behaviour of PRO*IV* the value variable &#@SUPP-COMM needs to be enabled at the start of the transaction and disabled at the end.

If the maintenance of data integrity is of prime concern or a motivating factor in moving to an RDBMS then this stage could be combined with the implementation of stages 1 or 2.

Advanced techniques

This section will give additional information that may help to tune your application more to your requirements. The settings described below can either be set in the [ENVIRONMENT] stanza of the pro4.ini file or set as environment settings using the appropriate technique for the platform, unless otherwise specified.

TP_ROLLBACK=Y

Use of this setting causes certain PRO*IV* error messages to perform a rollback on the database. The following error messages will generate a rollback on the database by default when this setting is enabled:

> 3 – 6, 11, 13 – 15, 17 – 19, 24, 29, 35 – 40, 48 – 50, 54, 124, 140, 157, 161 – 162, 172, 176, 191, 210, 300, 360 – 375, 377 – 399, 660 – 661, 663 – 669, 712, 722 – 725, 899

It is possible to override the messages that will cause the rollback behaviour to be performed by adding an additional stanza to the pro4.ini as follows:

```
[ROLLBACKMESSAGE]
NO=3-4,24
YES=28,356
```

These lines cause message 3 to 4 and 24 (security violation, security level violation and division by zero) to no longer cause a rollback and for messages 28 and 356 (exponent out of range and transaction integrity error) to now cause a rollback. Note that individual message numbers are separated by commas and ranges of message numbers are included or excluded by separating the start and end message numbers by a hyphen (-) without any spaces. At present (PRO*IV* version 5.0r109) this facility only works on Windows platforms.

SQL_TRANSACTION_ERROR=Y

This will generate PRO*IV* error message 356 (SQL ERROR: APPLICATION TRANSACTION INTEGRITY ERROR) if a commit is issued in an incorrect location. This error message will be output if the iteration counter or explicit commit is used with Oracle within a transaction, even though the transaction could have completed successfully due to the use of ROWID and OCI.

SQL_CURSORS=<num>

This setting defines the number of cursors that PRO*IV* will use. By default, if not specified, this value will be set to 128. If the number specified on this setting is greater than the value specified in the database configuration the database will return an out of cursor error, e.g. ORA-01000 from Oracle (default installation value for Oracle is 50 and for PRO*IV* is 128).

PRO*IV* uses one cursor for a file in look up mode. If the file is in add, change, both or delete mode then two cursors will be used for the file access. If the value for SQL_CURSORS is lower than the number of cursors required for a transaction to complete PRO*IV* error 361 will be output. If you increase the value of SQL_CURSORS then each PROIV user will use more memory on the PRO*IV* kernel system. Having a value of SQL_CURSORS that is very high can also adversely affect the performance of processing database commands.

Ideally, the value of SQL_CURSORS should be set to the lowest value required to allow the most complex transaction to complete. However, this can be an extremely difficult thing to determine. Different users can be set up with different values for SQL_CURSORS by either using different script files or, on Windows platforms, using different user ini files.

The new SQL layer, as provided in kernels from 5.5r210 onwards, includes a new setting. The value, AUTO, allows the kernel to dynamically increase the number of cursors available for use by the SQL layer. There is a slight performance degradation when using this option. However, this option can be useful for determining, in conjunction with tracing, the maximum number of cursors used by the application.

SQL_NOSIG=Y

This setting should be used on Windows platforms if problems are experienced with the connection. The problem is caused by an error with the Windows TCP/IP protocol stack, which only allows one blocking operation to be in effect at a time. When the PRO*IV* kernel issues its signal check to see if the database server is still processing the request it causes the connection between the database and the operating system to be interrupted because of the bug in the operating system.

Enabling this setting will prevent the waiting for database response message from appearing on the bottom line of the client. It will also stop the PRO*IV* kernel from processing any lock logics.

CONNECTION=<user/password/dsn>

This setting, which appears in the [DATABASE] stanza of the pro4.ini file, specifies the connection details used for the database. There are three values that can be specified for

the setting. The first two parameters are the user name and password for the database. The third parameter is used for connection through ODBC and is used to determine which system DSN (Data Source Name) is used. The old method of specifying the DSN using the SQL_DBNAME setting in the environment stanza is still supported.

For a connection to Oracle the password can have a SQLNet name appended, separated by a @ sign. Some systems require that the string containing the @ sign be enclosed in quotes. For example, CONNECTION=”system/manager@wheels”.

OS Authentication

In the normal method of setting up the database connection the database user name and password are entered in plain text into the pro4.ini file, the user specific ini file or the script used to start the kernel. It is possible to configure most database engines to accept connections using the Operating System authentication, which removes the need to hold the log on information in plain text.

To set this up for Oracle you will need to complete the following steps:

1. Set up, or determine the current value of, the OS_AUTHENT_PREFIX value within Oracle. By default this value is set to OPS$.
2. The next step is to create the appropriate user profiles within the database using the OS_AUTHENT_PREFIX value, e.g. OPS$user1. The password for these users must be set to IDENTIFIED EXTERNALLY.
3. The PRO*IV* file definitions may need to be changed such that the alternate file name includes the table owner, e.g. DBA.TABLENAME.
4. Finally, the user name and password settings used by PRO*IV* need to be changed to be null, e.g. CONNECTION=/ in the database section of the pro4.ini file, or export SQL_USERNAME= and export SQL_PASSWORD=.

PRODB_CHARSET=<value>

This setting, which appears in the [DATABASE] stanzas of the pro4.ini file, determines which character will be used to specify the upper limit when PRO*IV* creates the WHERE clause of the SQL statements. The valid values are A, a, Z, z, 0 and 9. For support of legacy settings the values of 7 and 8 (default value) are also supported. The value of 7 is mapped to z. Typically, if the wrong value for PRODB_CHARSET is used, a SEL-RANGE or SEL-PARTIAL will stop when the key value read from the database reaches W or Y if an ASCII (American Standard Code for Information Interchange) collating sequence is used.

Providing the correct collating sequence is used on the database then either the default value or z will result in the correct rows being returned from the table for ASCII encoding. For the EBCDIC (Extended Binary Coded Decimal Interchange Code) encoding the value of 9 should be used if numbers are to be included, otherwise Z should be used.

LOCKED_ROWS_RETURNED=Y

This setting should only be used with Oracle and is used in the [DATABASE] stanza of the pro4.ini file. When this is set the message that is displayed when a record lock is encountered will be changed to indicate which file is being locked. Additionally this will allow the record lock logic to be processed for Oracle.

ORACLE_DLLNAME=<dll name>

This setting should be used on Windows platforms when connecting to an Oracle database. The DLL name should be the fully qualified name, i.e. including the drive and path name, of the DLL supplied with the PRO*IV* installation. The DLL will have a name similar to PROORA815.DLL and will be installed into the same directory as the PRO*IV* kernel. The DLL supplied with PRO*IV* 5.0r104 and subsequent kernels is compatible with the Oracle 9.0.1 Windows client.

REPARSE=Y

This setting should be used if full function dynamic SQL is used in the PRO*IV* logic. If you do not use dynamic SQL you should not use this setting, unless requested to by PRO*IV* support personnel. This option is deprecated in 5.5r209.

Savepoint

If you need the ability to rollback a partial transaction to a known point then you can use the SAVEPOINT command of SQL. Savepoint is a SQL feature that may not be supported in your chosen RDBMS. To activate a savepoint use the following logic lines:

```
#X = SYS-SQL('SAVEPOINT <savepoint name>')
```

To roll back to the savepoint use the following logic lines:

```
#X = SYS-SQL('ROLLBACK TO SAVEPOINT <savepoint name>')
```

A typical use of this command would be for use with order entry. The customer details entry would be written as one function, where the PRO*IV* commit processing is turned off in the entry logic. After entering the customer details the function would exit and then link to another function to allow entry of the order details. In the entry logic of the order details function the savepoint would be set. The user could then enter the order lines. Two buttons could be provided one to accept the order, the other button would allow the order to be cancelled, retaining the customer details, by using the rollback to savepoint. A third button could also be provided that performs a normal rollback, thus removing the customer details as well.

&#@SQL-SORT

This system value variable when enabled causes the default PRO-ISAM type sort to be replaced with an ORDER BY clause, which is appended to the end of the Type 1 Full Function or Transparency Mode SQL statement. Without this statement, which must be the first logic command in a sort-select logic, PRO*IV* will select all the records for the cycle using the appropriate table index for the key sequence defined in the primary file of the cycle and then process the sort in the normal way. By enabling this option the database engine can determine the best index to use to retrieve the rows from the table in the required sequence.

The ENABLE(&#@SQL-SORT) command only takes effect if all files up to the sort-select end file are RDBMS tables and all secondary files included in the sort can be accessed using column names from tables higher up the list of files. This means that if any Before Sort Read logics are used in the cycle this option will have no effect. Additionally, this option will be ignored if any of the sort fields specified are scratch variables.

The ORDER BY clause of a SQL statement is used by most RDBMS to decide which index is used to access the data. If the RDBMS is unable to use an index for the table due to the ORDER BY clause generated by the sort field(s) then the RDBMS access could take a considerable amount of time on a table that contains large volumes of data. Creating the correct indexes on a database can significantly improve performance. Conversely, creating too many, or the wrong ones, can degrade performance.

The new SQL layer will, by default, enable this option. The option can be disabled if any of the conditions described above are not met.

Full function SQL

There are two types of full function SQL that are supported by the PRO*IV* version 4.0 and 5.0 SQL engines, plus Transparency Mode. Full function SQL is enabled by using the SQL…ENDSQL command pair in the default logic of a cycle with a SELECT statement between them. If full function is not enabled PRO*IV* will generate its own SELECT statements – this is known as Transparency Mode. In addition to these three modes the user can also use non-SELECT statements in logic.

Type 1

Type 1 SQL is a format similar to that generated by PRO*IV* in transparency mode. The PROIV kernel generates the list of columns to be selected from the table specified in the SELECT statement. The list is generated by referring to the fields specified in the PRO*IV* file definition and will only include those used within the function. This PRO*IV* file definition need not include all the columns defined on the table and need not have the fields defined in the same sequence as the columns. If a field on the file definition has an

alternate field name specified this will be used in the SELECT statement instead of the field name used within the PRO*IV* function.

The file specified in the select statement is the PRO*IV* file name. If an alternate file name is specified on the PRO*IV* file definition this will be passed to the database, otherwise the PRO*IV* file name will be used.

The general form of type 1 SQL is as follows:

SELECT
FROM <PRO*IV* file name>
[WHERE]
[ORDER BY]

N.b. the use of SELECT * FROM <PRO*IV* file name> in full function SQL is now handled as type 1 SQL. In versions prior to 4.6 this would be treated as type 2 SQL and would return all columns from the table instead of just those used in the function. However, the use of an * qualified by a table name or table name alias does still result in type 2 SQL statement processing, i.e. SELECT CUSTOMER.* FROM CUSTOMER is type 2 SQL, whereas SELECT * FROM CUSTOMER is type 1. In type 2 processing the * will return all columns from the table in the order that they are defined on the database.

Type 2

Type 2 SQL is passed to the database as coded with only minor changes being made by the PRO*IV* database engine. The general form of a type 2 statement is as follows:

SELECT [column list/SQL function]
FROM <table name>
[WHERE]
[ORDER BY]
[GROUP BY]
[HAVING]

The name specified on the FROM clause must be the table name and is only associated with a PRO*IV* file definition by its location in the PRO*IV* processing cycle. The columns specified in the SELECT statement must be compatible with the fields specified in the PRO*IV* file definition that will receive the results. SQL functions include MAX, MIN, AVG and COUNT.

N.b. the use of an * qualified by a table name or table name alias results in type 2 SQL statement processing, i.e. SELECT CUSTOMER.* FROM CUSTOMER is type 2 SQL, whereas SELECT * FROM CUSTOMER is type 1. In type 2 processing the * will return all columns from the table in the order that they are defined on the database.

There is no mechanism for a type 2 SQL statement to access a table via a logical database entry defined in the pro4.ini file. If you need to access a table that exists in a logical database using type 2 SQL you must make sure that a route to the table exists via the database defined in the default database connection. The performance and transaction handling of a table accessed by this method may not match that experienced had the table been accessed directly through the logical database route.

Non-SELECT statements

The three modes just described are all processed by PRO*IV* within the file access part of the PRO*IV* timing cycle. Non-SELECT statements are processed in logic at the point that they are encountered. Non-SELECT statements do not return data, merely a return code. The statement is defined by either enclosing it in the SQL…ENDSQL pair, as with full function SQL, or by specifying the statement in the SYS-SQL logic command.

Transaction processing, cursor manipulation and database connect statements are not allowed to be specified in SQL statements within PRO*IV* logic. The PROIV database engine handles these functions. It is also not possible to create new tables or alter existing table definitions from within a PRO*IV* session.

Dynamic SQL

The dynamic SQL available in PRO*IV* from version 5.0r104 allows construction of the WHERE clause of an SQL statement at execution time. This is particularly useful for complex search screens where optional search criteria can be selected from within the application. Without the use of dynamic SQL this functionality is extremely difficult to implement. Use of dynamic SQL in PRO*IV* requires that REPARSE=Y is implemented in the pro4.ini file or set as an environment variable in kernels prior to 5.5r209.

The standard, non-dynamic, full function SQL allows binding of variables to the WHERE clause of the SQL. This binding is done at runtime, meaning that the SQL itself is static. The database engine creates an execution plan when the SQL is sent to the database the first time. Each time the SQL is executed the same execution plan is used but different variable values are used. When a PRO*IV* function that contains non-dynamic SQL is compiled the SQL is converted into a form acceptable to the database that can have variables bound in. For example

```
SELECT COUNT(*) FROM ACCT_TABLE WHERE ACCT_CODE BETWEEN :$LOW AND :$HIGH
```

becomes

```
SELECT COUNT(*) FROM ACCT_TABLE WHERE ACCT_CODE BETWEEN ? AND ?
```

and the PRO*IV* system provides pointers to the :$LOW and :$HIGH data at runtime.

With non-dynamic SQL there is no way to alter the SQL itself, only the bound variables can change. When building a search screen each possible form would need to be listed in a CASE statement. For simple cases it is possible to write out all the alternative SQL statements, however if you had ten independent selection criteria it would be infeasible to

list all 1024 possible alternatives. With the new dynamic SQL you can build up the selection criteria in a scratch variable and then specify this on the WHERE clause of the SQL statement.

Dynamic SQL does not bind the variables at run time and, consequently, each time the SQL is sent to the database a new execution plan will have to be created. The WHERE clause is built up in logic and then used in the SQL statement, for example:

```
$$SEL = "DO_CUSTOMER_CODE LIKE 'B%'"
IF $STATUS # ''
   $$SEL = $$SEL + ' AND DO_CUS_STATUS_COD = ' + $STATUS
ENDIF
IF  #CRED_LOW > 0
   IF #CRED_HIGH > 0
      $$SEL = $$SEL + ' AND DO_CUS_CREDIT_LIMIT BETWEEN ' + CONV(#CRED_LOW) +
              ' AND ' + CONV(#CRED_HIGH)
   ELSE
      $$SEL = $$SEL + ' AND DO_CUS_CREDIT_LIMIT >= ' + CONV(#CRED_LOW)
ELSE
   IF #CRED_HIGH > 0
      $$SEL = $$SEL + ' AND DO_CUS_CREDIT_LIMIT <= ' + CONV(#CREDIT_HIGH)
ENDIF
SQL
   DYNAMIC SELECT FROM DO_CUST WHERE :$$SEL
ENDSQL
```

Note that the quoted string ('B%') passed to the SQL on the first line has to be enclosed in single quotes therefore, for ease of reading, the whole string on this line has been enclosed in double quotes. It is also possible using dynamic SQL to change the column used in the selection statement, e.g.

```
CASE $TEST
 WHEN 'ACC': $FIELD = "ACCT_CODE"
 WHEN 'NAM': $FIELD = "ACCT_NAME"
 WHEN 'ADD': $FIELD = "ACCT_ADDRESS_LINE_1"
 OTHERWISE: LSEXIT
            EXIT
ENDCASE
$CONDITION = " LIKE "
$VALUE = " 'AB%' "
SQL
   DYNAMIC SELECT COUNT(*) FROM ACCT_TABLE WHERE :$FIELD :$CONDITION :$VALUE
ENDSQL
```

By changing the value of $FIELD in the CASE statement a different column will be used for the selection criterion. Note that the strings in the assignment statements are surrounded by spaces, this is to ensure that the select statement passed to the database engine has spaces between the various components. Conversely, you should not rely on the lack of spaces between the components to concatenate strings on the SELECT statement. PRO*IV* may insert a space before and after every string it expands prior to sending it to the database engine.

If a numeric scratch variable is included in the WHERE clause of a dynamic SQL statement this will be treated the same way that a non-dynamic variable would be treated. That is, the variable will be bound to the SELECT statement at execution time, rather than being passed as a fixed value.

The generated SQL will be sent to the database exactly as supplied. It is important that any user-supplied data that is inserted into the SQL is properly sanitized, otherwise it may be possible for a user to alter the SQL and potentially read or alter data to which they should not have access. For example, consider the following code that should only allow the user to access "low security" accounts, furthermore assume that account "AB1" is a "high security" account:

```
$SELECTION="ACCT_CODE = ‘" + $ACCT_NO + "’"
SQL
   DYNAMIC SELECT FROM ACCT_TABLE WHERE SECURITY_LEVEL<2 AND :$SELECTION
ENDSQL
```

If $ACCT_NO can be supplied by the user then they could supply the following value:

```
AB1' OR ACCT_CODE='AB1
```

Note the unmatched single quotes (').

When this is expanded we end up with the following SQL:

```
SELECT FROM ACCT_TABLE WHERE SECURITY_LEVEL<2 AND ACCT_CODE = 'AB1' OR
ACCT_CODE='AB1'
```

Because AND binds tighter than OR in SQL this will have the effect of avoiding the hard-coded SECURITY_LEVEL<2 condition.

Logical databases

The pro4.ini file allows multiple database connections to be defined. Logical database definition allows the different sections of an application to connect to different databases, e.g. payroll to one database and the purchase ledger to another. It is even possible to specify that the separate logical databases use different RDBMS. Extreme caution should be used when multiple databases are used within a single transaction as this can lead to severe problems with data integrity should a problem occur during the commit phase.

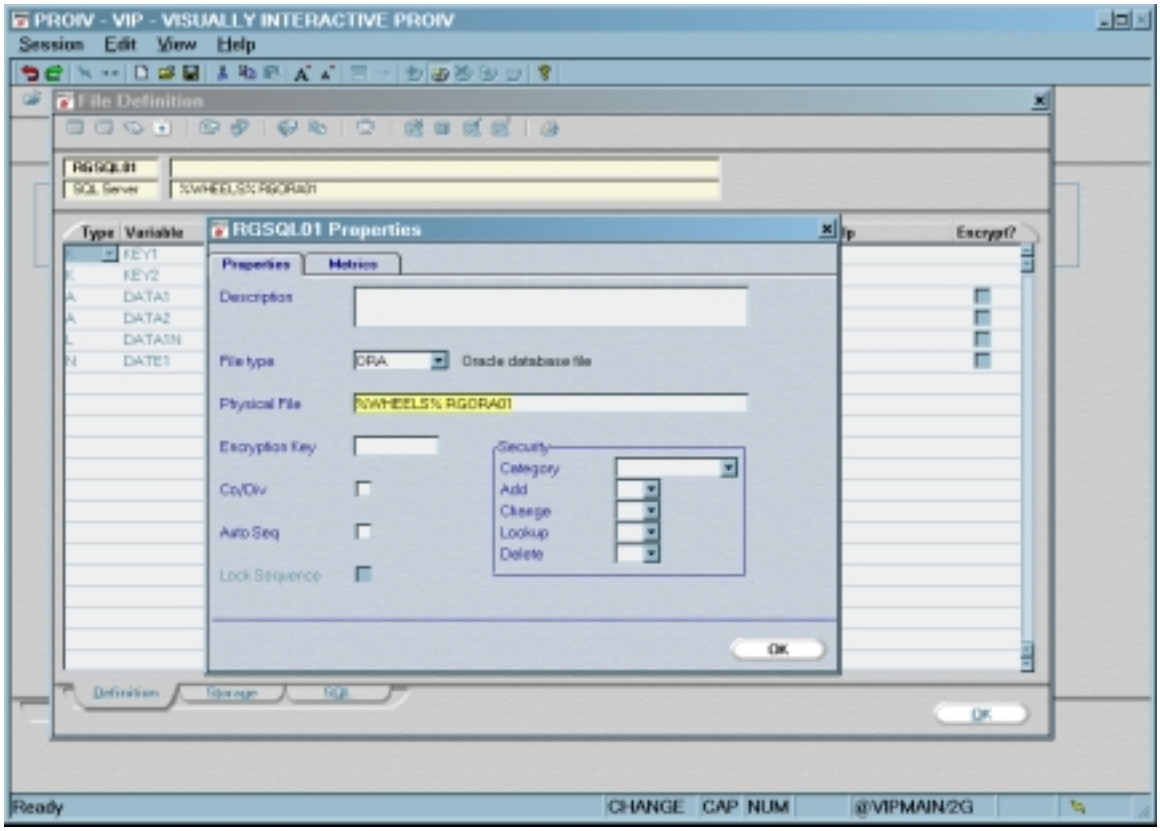
To set up logical databases you will need to have multiple [DATABASE] stanzas defined in the pro4.ini file. The name specified in the stanza heading can then be used in the alternate file name within PRO*IV*. When coding the alternate name in your application it must be contained within % signs. The logical database name can also be used within the ALIAS logic command. The following example shows a section of a pro4.ini file with two logical databases defined, plus screen prints of two file definitions. The first of these file definitions shows the use of a logical database name along with mapping to a different name in the database. The second file definition shows the use of a logical database name without reassigning the name. Notice that the names in the % signs match the names in the [DATABASE] stanza heading. If the named stanza is not found in the pro4.ini then the error message ‘390 – SQL_DBTYPE EITHER NOT SET OR INVALID’ is displayed.

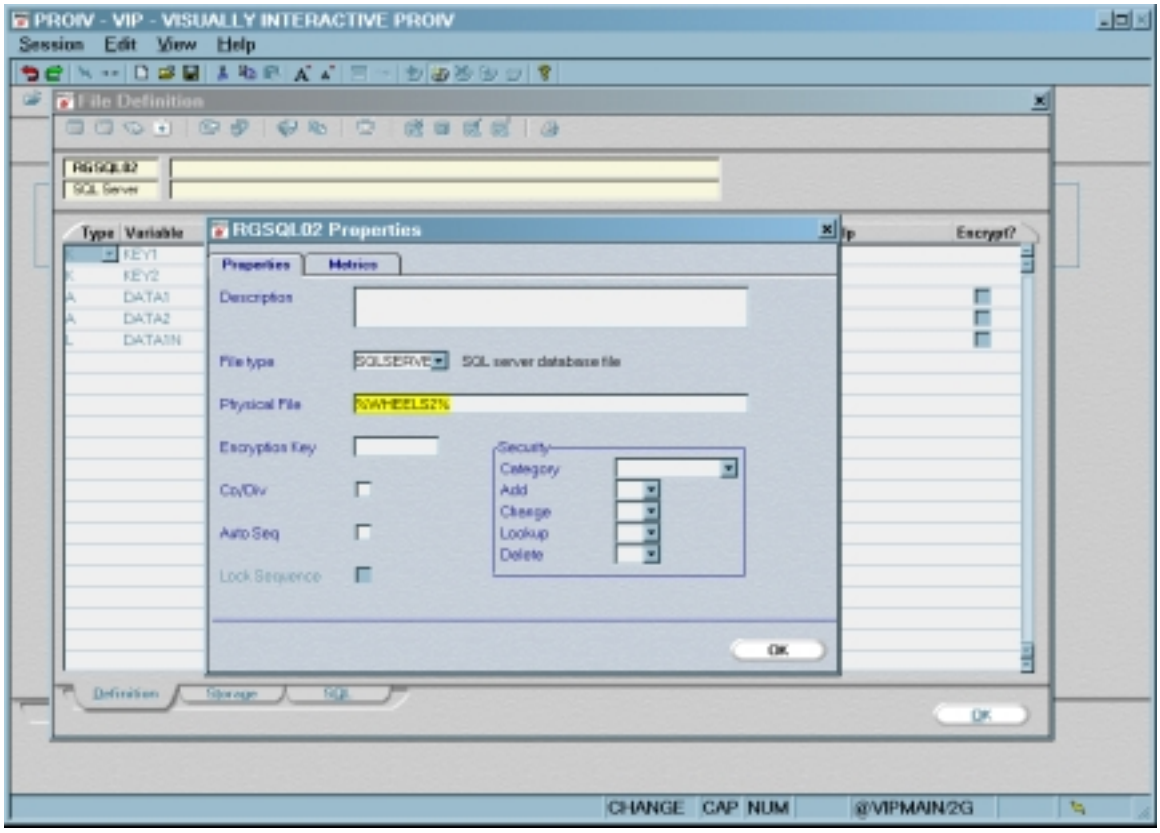
```
[DATABASE - SQLDEFAULT]
FILETYPE=SQLSERVE
```

```
CONNECTION=sa//default
PRODB_CHARSET=z

[DATABASE - WHEELS]
FILETYPE=ORACLE
CONNECTION=system/manager@database1
PRODB_CHARSET=z

[DATABASE - WHEELS2]
FILETYPE=SQLSERVE
CONNECTION=sa//database2
PRODB_CHARSET=7
```





A logical database name can be used on the ALIAS logic command but the name to which the PRO*IV* file is mapped *must* also include the table name to be accessed in the database, e.g. ALIAS('RGSQL01', '%WHEELS2% RGSQL01'). Failure to include the table name in the ALIAS logic command will lead to a malformed SQL statement.

ALIAS logic command

The only valid point at which the ALIAS logic command can always be used is in the 'logic in' of the first function of a transaction. Use of the ALIAS command when a file or RDBMS table is open can lead to unpredictable results.

The ALIAS command can be used without problems on PRO-ISAM files at points that are not valid for RDBMS tables, if the programmer is aware of the way that the kernel opens and closes files. This is feasible because the PRO*IV* kernel will close a PRO-ISAM file that is in look up mode if it runs out of file handles. With an RDBMS all tables are held open until the transaction completes.

It is possible to alias an RDBMS file to an alternative table or to a logical database using the ALIAS command. A couple of examples of using ALIAS on RDBMS files follow.

To map the file WRCUST to a table of the same name but in a different logical database:

```
ALIAS('WRCUST','%BETTY%WRCUST')
```

N.b. the following is <u>not</u> valid:

```
ALIAS('WRCUST','%BETTY%')
```

To map a file from a logical database to the default database changing the table name accessed:

```
ALIAS('WRCUST','%SQLDEFAULT%CUSTOMER')
```

&#@SUPP-COMM

This value variable should ideally be enabled in the logic in (On Function Entry event) of a function in order to disable the standard PRO*IV* commit behaviour. Suppressing the normal commit behaviour allows an RDBMS transaction to be split over multiple PRO*IV* functions.

When the value variable is disabled, such that the normal PRO*IV* commits are again performed, a commit will not happen immediately. The commit will instead be performed at the next normal automatic commit point. See the section entitled 'Committable cycle' for details of where this will occur. To maintain control of the commit in this situation an explicit commit should be issued at the point that the PRO*IV* automatic commit processing is switched on. The ideal place to disable &#@SUPP-COMM is in the exit logic (On Function Exit event) of a function and to follow the command immediately with an explicit commit (e.g. #RC = COMMIT()).

> N.b. You should not use the database transaction executive COMMIT in either the SQL…ENDSQL pair or the SYS-SQL logic command, as the PRO*IV* kernel has no knowledge of the state of the database after executing this command. By using the PRO*IV* logic command the kernel is aware that cursor positioning has been lost and can re-open the database cursors.

Note that if you log out of PRO*IV*, or otherwise terminate your connection to the PRO*IV* kernel, with &#@SUPP-COMM enabled that any uncommitted work will not be saved. With SQL Server this action will cause an 'Invalid transaction state' error, which will be reported as a SYSTEM E366 error by the kernel.

Committable cycle

With the standard PRO*IV* commit behaviour, i.e. when it has not been switched off by issuing an ENABLE(&#@SUPP-COMM), every non-global function that has a file has at least one committable cycle. The committable cycle is defined as being the outermost currently active cycle that has a file. The file does not have to be an RDBMS file. A global function will only have a committable cycle if the calling function has not accessed a file and the automatic commit behaviour has not been disabled. Only one committable cycle can exist at any time.

Reports and updates

When the committable cycle in a report or update type function is exited a commit is issued to the connected database(s). An LSUPDATE cycle within a screen type function will also issue a commit on exit if it is the current committable cycle.

Screens

'Flat' cycles will issue a commit to the database after each primary file, and its associated secondary files, has completed its write phase. Paging cycles will be committed when a page of records has been processed. This will occur when the page up, page down or EOD key is pressed. A delete or insert of a group of records in a paging cycle is regarded as a single transaction and a commit will be issued when the delete or insert is completed.

SQL optimization

In order to improve performance when executing SQL statements in PRO*IV* there are several rules that can be followed.

From clause

Where more that one table is reference in a SELECT statement the table names in the FROM clause should be ordered such that the table that will return the fewest rows is specified first. Subsequent tables referenced should be organized in ascending sequence of expected number of returned rows.

Where clause

The tests in the where clause should be performed in the following sequence for maximum efficiency

1. Comparisons with scratch variables
2. Comparisons with variables from other tables
3. Table joins to be included in the output

Indexes

Where a specific sequence of accessing rows in a table is used regularly an index should be created on the table to improve access efficiency. However, the number of indexes on a table should be kept to a minimum because there is an overhead when maintaining the data on a table for each index. Most databases come with a tool for analyzing which indexes are required for the most efficient access of the data.

Multiple PROIV file definitions

In some circumstances it is worth defining multiple file definitions for the same RDBMS table. The reason for doing this is to reduce the size of the SQL statement sent to the database and the amount of data that needs to be deblocked by the PRO*IV* kernel.

An example of where this would be appropriate is if you had a table with 100 columns that needed to be accessed in a PRO*IV* application but you also needed to periodically update a column that was near the end of the table. In this circumstance you would have one file definition that had fields for each column and another one that contained only the key fields plus the field that needed to be updated. Because PRO*IV* uses the external field name to construct the SQL statements individual columns can be addressed directly without retrieving what would be intervening fields on a flat filing system, such PRO-ISAM or C-ISAM.

Things to avoid

Wherever possible you should avoid using BETWEEN, IN, EXISTS and nested SELECT statements that reference variables from an outer SELECT statement.

Do not use variables that are retrieved from any file used in the cycle in the WHERE clause. If you do, the results returned might not be what is expected. Instead you should assign the file variable to a scratch variable and use that in its place in the SELECT statement.

Do not use LIKE, SOUNDEX or comparison operators (e.g. >, < or =) on the key fields in a full function SQL statement in the default logic of an LS (cycle). If you do use any of these the database engine may not be able to use any indexes built on the table.

Kernel response waiting message

With the introduction of MFC GUI client version 326 a new method of indicating that the client is waiting on a response from the kernel was introduced. Reasons that the client is waiting on a response from the kernel include: database record lock, awaiting completion of long SQL statement and a communication problem between client and kernel.

To implement the new method of reporting the wait state in the client, the following changes have to be made:

On the PRO*IV* server either export or include in the [ENVIRONMENT] stanza of the pro4.ini file the following line:
SQL_NOSIG=Y

On each client PC insert the following line in the proiv.ini file in the [Settings] stanza:
KernelWaitTimeout=<number of seconds>
KernelWaitHourGlass=<0 or 1>

The message out put is display on the bottom line of the client, overwriting any user or system message that may have been displayed. When the kernel responds the message line will be cleared. If the value of KernelWaitTimeout is set too low then the message will appear during normal processing. Setting KernelWaitHourGlass to 1 causes the 'Normal Select' mouse pointer to change to the 'Busy' pointer when the timeout is reached.

ODBCCMPT

If you connect to SQL Server from PRO*IV* and you have installed the SQL Server 2000 client onto the PRO*IV* kernel system running a Windows server version you should run a utility provided with the SQL Server client installation. The command to run is:

```
ODBCCMPT pro32srv /v:7
```

Before running this command you should stop the PRO*IV* service from the service manager and then restart it afterwards.

This command puts the SQL Server ODBC driver into SQL Server 7 compatibility mode. The PRO*IV* Windows kernel is compiled using SQL Server 7 libraries in order to provide backwards support for customers still using SQL Server 7.

This command must be run for kernel up to, but not including 5.5r210. If you have run this command previously then you must disable it when upgrading to 5.5r210 or higher. To disable ODBCCMPT version 7 compatibility you must type the following command:

```
ODBCCMPT pro32srv /v:7 /d
```

The reason that this is no longer required is that the new SQL layer now uses ODBC version3.0 commands.