TITLE

OSB™ Data Access

AUTHOR

David Crichton, Impulse Solutions Ltd.

CLASSIFICATION

Technical Reference

RELEASE

1.6 – 8th January 2011

This document is intended to provide a guide to the data access functions available to users of OSB™ (Tibco Object Service Broker) formerly referred to as ObjectStar™.

1 Contents

1.1 Changes from Previous Issue

1.1 Use of MAP tables
1.2 FIELD and PARAMETER Usage
1.3 EES table type added for Rel 5.0
1.4 Explanation of EES usage
1.5 GET WITH MINLOCK
1.6 Rules and Libraries

1.2 Document Cross References

The distributed OSB™ manuals should be used to provide additional data for many of the topics referred to in this guide.

1.3 Revision History

1.0 6th Sept 2006

1.1	3rd March 2007
1.2	29th June 2007
1.3	6th October 2008
1.4	13th July 2010
1.5	24th October 2010
1.6	8th January 2011

1.4 Impulse Solutions Ltd

Impulse Solutions Ltd is an ObjectStar Consultancy which has been developing utilities and add-ons for OSB(ObjectStar)™ for over fifteen years. It has also provided resource for application development and maintenance at a number of European and Australian customer sites and contributed a number of functions to the ObjectStar™ Rel 4.1 IDE graphical tool.

2 Introduction

This document is intended to provide a description of the concepts of data access in OSB™ and the various functions and tools available to users of the product. It assumes little or no knowledge of the processes described but is intended both for new users and also for those with greater experience of the product.

2.1 Acknowledgements

My thanks to Andy Hampshire of Tibco Software Inc for reviewing the draft of this guide and offering some very useful advice and the occasional correction. Thanks also to Dave Clark and Ron MacRae in the Tibco OSB™ Support team for equally helpful input on performance and DOB behaviour.

2.2 Disclaimer

The views expressed in this document are those of the author and do not necessarily represent the official views of Tibco Software Inc.

2.3 Intellectual Property Rights

3 OSB™ Data Access Concepts

3.1 Overview

Data access in most programming languages and database systems requires definitions to be declared in each program or code section where data access takes place.

OSB™ adopts a different approach by defining the characteristics of a data source once and assigning a unique identifier to the definition. All access to that data source in any piece of code then simply requires a reference to that identifier.

There are many benefits to be gained from this approach. First, the existence of a single definition ensures that all references to a data source refer to exactly the same thing. Secondly, the physical source of the data may be changed by amending this common definition. This allows, for example, testing and development of code to use limited internal test data with the production version switched to an external legacy data base without any change to the code. In addition, the ability to use indirect references as described later in this guide allows individual users and user groups in an application to reference alternative data sources through common code.

In addition, in OSB™ everything is treated as data including code (referred to as Business Rules). This gives a unique scalabilty and portability where a database can be reorganized physically or copied to another hardware node without change.

3.1.1 OSB™ Architecture

OSB™ uses Client/Server concepts regardless of which platform it is running on – Z/OS, Windows XP/2000 or Sun Solaris (UNIX). Release 5.2 also runs on Windows Vista or Windows 7.

There are three components required to use OSB™

The Data Object Broker (DOB)
The Execution Engine (EE)
Client Interfaces (Workbench, Batch, API, ODBC)

In any implementation, all may run on the same platform or some may run on different ones – for example, a mainframe-based implementation may be accessed by the ObjectStar™ IDE or the OSB™ UI running on a Windows XP platform.

This guide relates to the Data Object Broker component only. Information about the other components and their relationship can be found in the distributed OSB™ manuals.

An installation may – and usually does – have more than one DOB. There will be a Production version and at least one for Development. There may be other versions for User Test, Education etc as required.

Each version is identified by a unique Node name which is installation-defined and is used primarily to determine the Client/EE/DOB linkages in use. It is also used when Distributed Processing is implemented which allows a Client to access data in more than one DOB at the same time. Reference is made later in this guide to the benefits and other implications of this feature which uses TCP/IP links to provide the required connections.

Each DOB can have up to 256 Segments each of which can be held on from 1 to 128 physical Page datasets. A DOB will have at least 3 segments – Segment 0 otherwise know as the MetaStor and Segment 1 which holds user-defined data. There is also a Secure Access Log segment (default Segment 99) which holds unmodifiable security logging data.

3.1.2 The MetaStor

Segment 0 holds the system-related data tables, including data definitions, screen and report definitions and the code library tables. The unique features of OSB™ including the method of holding code (or Business Rules) give the product is portability and scalability benefits.

3.1.3 The **TABLES** Table

The **TABLES** table is the anchor point of the complete OSB™ implementation. It is the only data table which is not a child object of another table. All other defined tables are included in the **TABLES** table. Note that the **TABLES** table does not follow the same behaviour as other tables and should not be modified except by the Table Definers.

A table is identified by a set of values in the **TABLES** entry, the most important of which are the Name which is a 1-16 character identifier and a 3-character **TYPE** value which defines the class of table and by doing so determines the other tables which contain components of the table definition. For example, all tables (with special exceptions for **SCR** (Screen) and **RPT** (Report) tables) have at least one **FIELD** entry to define the data fields in each row (record) of the table.

3.1.4 Parameters

All types of table can be unparameterised – that is, there is only one occurrence of the data. Most types can also have from 1 to 4 data parameters which implement a multi-dimensional array of occurrences of a table, each identified by a set of parameter values. This allows logical separation of data within a table.

A further **LOCATION** parameter can be defined to support Distributed Processing. Data from a specific Node is accessed by setting this additional parameter (which can be the only parameter for a table) in referencing code. If this value is omitted the default is the current Node.

Parameters have both a physical and logical effect. Each occurrence is stored in a separate set of database pages so that the minimum space requirement is one 4kb page. This should be considered when designing parameterised tables since if they consist of a large number of occurrences with few rows in each the storage required will be significantly more than that required for an unparameterised table with the same total number of rows. The logical implication is that locking occurs at occurrence level so that different occurrences of the same table can be processed at the same time without lock interference.

There is a further type of parameter which occurs when a **SUB**view is created which has more parameters than the source as described in the later section on Table Types. This type of virtual partitioned data does not have a matching **PRM** table so the occurrences cannot be displayed as for a normal parameterised table. Impulse Solutions Ltd have developed a Parameterised Data Browser which can generate and display the derived occurrences for a parameterised **SUB**view; this is available as part of the ISL Tool Kit.

3.1.5 Segmentation

Permanent user data held internally (**TDS** or Table Data Store) is assigned to a data segment from 1 to 255 depending on the segments defined for a specific Node. By default the segment assigned is the default segment in the user parameters for the person who creates the definition. Definitions for tables which contain no data may be assigned to a new segment if required; those containing data require a more complex process.

3.1.6 Distributed Processing

Where Distributed Processing is implemented code may access data in the same table on different Nodes by means of the **LOCATION** parameter. Note that when a table exists in more than one node it can be of different types in the different nodes – e.g. **TDS** on one and **DB2** on another. Although the data type and source may differ between Nodes it is essential to ensure that the Field and Parameter definitions are the same to prevent code failures or incorrect results occurring. The use of Minimal Definitions as described later can assist in assuring consistency when a single physical table is to be made available on a number of different nodes.

3.2 Table Definitions

Each table accessible by a DOB requires a Table Definition. The Table Definer tool allows authorised users to specify the characteristics of the table, including the type, the fields, any parameters, and optional data such as ordering, default values and required field indicators.

3.2.1 Ordering

All data is held in the order of its key fields in ascending sequence and by default processed in the same order. Individual fields in the table definition can defined as having ascending or descending sequence so that the data is retrieved in that order automatically. For example, a **SUB**view can be set with the data presented in a different order from the base table. Note that specifying any Order value will mean that the data is ALWAYS sorted during retrieval unless one of the following is true:

- The code specifies ordering of the data so that it is returned to ascending key sequence
- The first primary key is specified as Descending order when the data will be retrieved in descending primary key sequence without any sorting required.

3.2.2 **TDS** Tables

Permanent data internal to the MetaStor is defined as Table Data Store or **TDS** data. An alternative permanent data type known as Hash Data Store data (or **HDS**) is no longer valid.

3.2.3 **PRM** Tables

If a table other than Screen and Report types has data parameters the corresponding **PRM** table allows the current occurrences to be browsed. Although it is not required to have a **PRM** table it is strongly advised to create one as there are a number of functions that cannot be carried out on parameterised data where no **PRM** tables exists. More than one PRM can be defined for a parameterised table but this is not recommended except in special circumstances related to security. One reason for this is if the parameters of a table are changed and in particular if data parameters are added or deleted then the corresponding **PRM**(s) MUST be rebuilt or data access failures may occur. It is sufficient to open the PRM definition in the Table Definer and then save it with PF3 to achieve this result.

3.2.4 **EES** Tables

An **EES** (Execution Environment Session) table has been added to the product at Rel 5.0 and is used to hold data that persists for the duration of an Execution Environment session. This allows data to be shared between sessions running in the same Execution Environment and may be considered as an intermediate state between ephemeral data such as **SES** or **TEM** tables and permanent data such as **TDS**. **EES** tables will

usually be populated in a control task or by shared/sharing tasks which test for the existence of data and populate on first access.

An **EES** table always includes 2 fields which are added automatically by the table definer and must be present. These are both QB4 and named **@@UPDATE_COUNT** and **@@REF_COUNT**. Both are set to 1 when a row is inserted.

@@UPDATE_COUNT is incremented by 1 each time a row is replaced. Because an **EES** will usually be shared by a number of users the buffer value is compared to the database value when attempting to replace and if the values differ a **LOCKFAIL** condition is raised. The row must then be read again to gain control. This mechanism allows effective synchronization in multi-user situations and in particular for online transactions where more than one user may access and update an individual transaction and an **EES** table is used to manage this. However, it must be remembered that **EES** data is shared ONLY by users of the same EE and therefore its use depends on the operational architecture.

The **@@REF_COUNT** field is updated each time the row is read and provides a mechanism to monitor data usage.

3.2.5 **SES** Tables

A **SES**sion table is used to hold data that persists for the duration of a session – either a batch job or a workbench activation. **SES** tables need to be cleared if they are to be reused in a code loop. This type of table provides a simple mechanism for passing data into an update transaction in multiple row format. Note that **SES** tables cannot have a **LOCATION** parameter and are only available in the node in which they are populated.

3.2.6 **TEM** Tables

A **TEM**porary table holds data that persists only for the duration of a transaction. This type of table is intended to be used as a working area where data of one or more rows is being manipulated and used. Unlike **SES** tables, **TEM** tables can have a **LOCATION** parameter and be shared between different nodes in a distributed system.

3.2.7 Screen and Report Tables

SCReen and **RPT** (Report) tables are used to populate the defined areas in 3270-format Screens and Reports respectively. They have a single parameter which is the name of the Screen or Report to which they are currently associated. A Screen or Report table can be associated with many Screens or Reports as determined by the structure of the parent object as specified in the relevant definer tool.

3.2.8 Legacy Data Gateways

OSB™ has a virtually unique ability to access Legacy Data sources directly through a series of software Gateways. This allows data in various external databases to be mapped by the Table Definer in exactly the same way as an internal table, with the addition of a cross-mapping feature which establishes the relationship between the fields of the external database and the internal OSB™ mapping. Data can then be both read from and updated in the external databases directly. The automatic checkpoint/recovery mechanism applies as for **TDS** tables with the proviso that the data update protocols of the external database also apply. This means that Legacy Data can be maintained without knowledge of the coding techniques specific to those databases.

The supported external databases (with their Table **TYPE** values) are

- **ADA** Adabas
- **DAT** CA Datacom
- **DB2** DB2
- **IDM** IDMS
- **IMS** IMS/DB
- **204** Model 204

There is also a Gateway for ODBC and Oracle data (**TYPE** is **SLK**). Details of this can be found in the relevant OSB™ manual.

3.2.8.1 Fail Safe Processing

An option exists in defining Legacy Gateways to ensure that when both **TDS** and Legacy data are updated in the same OSB™ transaction both internal and external data is successfully updated for the transaction to succeed. This process is described in the various OSB™ Legacy Gateway guides and the actual requirements depend on the Legacy Data type. However, the effect in each case is to cause the transaction to roll back and discard the current intent list if the external data update does not complete. An option exists to retry the external update when no completion signal is returned from the external Gateway and the transaction is placed in 'In Doubt' status.

3.2.9 External Files

OSB™ supports external flat files in all environments and also mainframe VSAM data. The **IMP** and **EXP** files allows data from other sources to be read into OSB™ or data to be passed to external applications in tabular or delimited format.

- **IMP** Import files – inward flat file data
- **EXP** Export files – outbound flat file data
- **VSM** VSAM data

3.2.10 **SUB**views

A **SUB**view is an alternate representation of a **TDS**, VSAM or Gateway table used to provide a number of additional features and usages.

- Where another record is required from a table during a FORALL loop or when a second occurrence of a parameterised table needs to be accessed within the loop, the original table definition cannot be used again. An additional view of the data is needed which is obtained by creating a 'mirror' **SUB**view. When Table **TYPE** of **SUB** is specified in the Table Definer and the original table is used as the source table name, the parameters and fields of the source are copied with 'empty' definitions. The actual values are retrieved at runtime from the source table. Any number of such 'mirror' **SUB**views can be defined if further data retrieval from the source is required in this way.

- Where it is required to provide access only to a subset of the fields in a table then a partial **SUB**view can be defined as above but with fields removed that are not to be available. These CANNOT include the parameter and key fields which must be present.

- Where fields of the source table are to be used but in a different format and with a different name. For example, a date field in Julian format (YYDDD) can be defined as a DB Date field.

- Where additional fields are to be included in the **SUB**view populated at runtime by process rules using either data from the table or external values. These fields can NOT be used for data selection or ordering as they are ephemeral. See the later section on Event Rules for tools which can be used by generic Derived Field Rules to determine the table or occurrence which has invoked the rule.

- Parameterised **SUB**views of unparameterised data or SUBviews with a different number of parameters can be defined if required. For example, the OSB™ **USERID** table which defines valid users (**@USERSOPTIONS**) which is unparameterised has a parameterised **SUB**view (**@USERS_OPTIONS**) where each individual user has its own occurrence. This allows locking of data to be partitioned. There is no standard tool to show the occurrences of a **SUB**view but the Extended Data Browser supplied by Impulse Solutions Ltd has this capability.

- A **SUB**view may have a selection setting which determines, for instance, the subset of occurrences of a parameterised table accessible to the **SUB**view or a set of filters which pre-determines the rows which will be selected regardless of any WHERE clauses in the accessing code.

3.2.10.1 Browse Mode SUBviews

A particular feature of **SUB**views is the ability to specify that it runs in Browse mode. This means that no locks are taken on the data but it also means that no updates can be made. This is a very useful feature in application design which allows any **TDS** table for which a Browse **SUB**view has been defined to be used within an update transaction with out the locking overhead when the data within it is effectively static at that point in time. Such **SUB**views normally contain all the fields of the parent table with no additional defined fields. It is strongly recommended that ALL application **TDS** tables have a Browse **SUB**view defined and maintained for this purpose. Care must be taken to reflect any changes to the parent table in the corresponding **SUB**view.

3.2.11 **CLC** Tables

Calculation Tables provide a simple and effective way of obtaining counts of data within a table or occurrence. When a **CLC** table is defined the fields are those of the source table plus a final **COUNT** field. Note that the source table cannot already have a **COUNT** field if this is to be part of the summary. Fields not required in the summary process are deleted – this can include key fields – and the remainder saved. Browsing the **CLC** table will now give a rollup count for the selected fields.

3.2.11.1 COUNTOCCURRENCES

CLC tables are of use where a single occurrence count is required from a table. However, if different summaries are required from the same source table a number of different **CLC** tables would be required, all of which would have to be managed as part of any update process.

An alternative is to use the **COUNTOCCURRENCES** builtin function for the different summaries required. This has the format

result **= COUNTOCCURRENCES(***table, selection***)**

where

result is the number of rows meeting the selection criteria
table is the name of the table to be searched

selection is the set of field names and values to be used, and the parameters required if the table is parameterised. This argument is a string which can be in **C**, **V** or **UN** format.

The string must consist of a set of

field/parameter **=** *value*

specifications separated by **&**.

Note that use of fields that are not either the first primary key or secondary indexes will result in a table sweep which may take some time to complete.

3.2.12 Memory Tables

There are three types of memory tables used in OSB™. Two of these, **MEM** and **MSG** are used for internal processing only and are outside the scope of this document but the third is used for communication between OSB™ code and external processes in either direction.

This is the **MAP** (Memory Storage Mapping) type. As the name suggests, it maps an area of memory which will be accessed by the external process either to insert data or retrieve it. A **MAP** table has a single (other than **LOCATION**) parameter named **ADDRESS** which is a 4-byte binary value. **MAP** data is accessed by setting this parameter to the value of the physical location at which the mapped area starts in memory.

3.2.13 Minimal Definitions

A Minimal Definition is used when data exists ONLY on a node or nodes other than the current one. It consists of the **NAME** and **TYPE** fields and a **LOCATION** parameter only. The rest of the definition must exist in full on the target node. Minimal definitions are used only in configured distributed systems.

3.2.14 Rules

OSB™ code, known as Business Rules, is held in tables in the metastor as data. Rules are stored in executable object form in the **RULEBODY** field of the **@RULESLIBRARY** table which is the only table of type **OBJ**. This field is 2198 bytes in length thus restricting the maximum size for an individual rule. Full details of rule maintenance are given in the Processing manual but it is important to explain that there is no source code as such so that this table is the sole repository for code. The Rule Editor converts input statements to object code when a rule is saved and detranslates it when an existing rule is re-edited.

3.2.15 Libraries

OSB™ supports multiple code libraries. There are two defined with the distributed system, both of which are bound. One, the **COMMON** library, contains rules distributed with the package which are used by system functions or available for inclusion in user code and the **SITE** library which is initially empty but used to hold production copies of user rules. All libraries are defined in the **@LIBRARIES** table.

3.3 Fields and Parameters

A Table definition must contain at least one Field definition to describe the row structure of the corresponding Table (two exceptions are for Screen and Report tables as described later). If the table is parameterised up to 4 data Parameters can be defined as well as a single Location parameter used for Distributed Processing and described later. An unparameterised table may also have a Location parameter. However, not all Table Types allow Location Parameters or in some cases any Parameters. The system table **TABLE_FORMATS** defines the allowed characteristics for each table type.

Field definitions are held in the **FIELDS** table and Parameter definitions in the **PARMS** table both of which have a single parameter which is the name of the table to which they refer.

The Field definition defines the characteristics and relative position of an element of the row in terms of Type, Syntax and Length. Most Field Types are of fixed format so that a specific length of storage is required regardless of whether the data requires it; the main exception is for Variable Character fields which are stored with a length prefix and the actual number of characters entered with trailing blanks removed.

Parameters are defined in the same way but have a CLASS attribute which is set to D except for the LOCATION parameter which is CLASS L. LOCATION parameters are always Type I, Syntax C and Length 16.

3.3.1 External Tables

External tables (e.g. IMPort, EXPort and Legacy Servers) have a corresponding set of external specifications for each Field in order to allow data to be read or written in the correct formats. The Table Definer for each requires the user to specify the external characteristics appropriate to the specific table type.

3.3.2 Screen Tables

A Screen is defined by a set of Screen Tables which have specific start and end columns and a start row. They are defined in the occurrence of the **SCREENTABLES** table for the Screen and either have a specific number of rows or are scrollable. An occurrence of the **SCREENFIELDS** table parameterised by the name of the corresponding Screen Table defines the position by row and column and the length of each data field and literal within the Screen Table relative to the start co-ordinates of the Screen Table. Where Screen Tables contain data fields, these are specified in the occurrence of **FIELDS** for that Screen Table. However, a Screen Table may only contain literals and will therefore have no **FIELDS** occurrence. This is one of the exceptions referred to previously.

3.3.3 Report Tables

A Report is defined in a similar way by a set of Report Tables held in the occurrence of the **@REPORTTABLES** table for the Report. The elements of each Report Table are defined in the occurrence of **@REPORTFIELDS** for the Report Table in a similar way to Screen definitions. Again, if a Report Table contains only literals then no **FIELDS** occurrence exists. This the other exception referred to earlier.

3.3.4 Data Assignment

When data is to be written to a table occurrence a data buffer must first be populated to set values for all key field and any non-key fields where the REQUIRED attribute is set to Y. There are 3 basic options available to do this:

- Absolute value e.g. *tablea.fielda = value*
- Value returned by a function e.g. *tablea.fielda = function*
- Field assignment e.g. *tablea.fielda = tableb.fieldb*

OSB™ provides a convenient option for an extended Field Assignment (Assignment By Name) which has the form

tablea. = tableb.**

This results in the values of all fields in the current data buffer for *tableb* which are also defined in *tablea* being copied to the data buffer for *tablea*. This allows changes to be made to the definitions of either table without any change in code.

3.3.5 Field Consistency

There is no automatic mechanism to prevent the same field name being used in different tables with different characteristics but there is a severe risk associated with this. If a specific or extended assignment is used between two tables where the field names are common but differ in definition the assignment may fail or generate unexpected results for one of the following reasons:

- Type and/or Syntax differs and is inconsistent – this will cause the rule to fail
- Type and Syntax are identical or compatible but the length of the target field is less than the source – if the source value exceeds the maximum value of the target the rule will fail
- The source is a Variable Character field and the target Fixed Character – any lower-case characters will be converted to upper case

It is strongly recommended that a consistent approach is made to the use of specific field names to avoid these problems occurring. If it is required to have a target field with different characteristics from the source then a different field name should be used.

3.3.6 Global Fields

OSB™ provides a facility to assist with maintaining field consistency by use of the Global Field table (**@GLOBALFIELDS**) which holds a unique definition of a field including a Business Description and also a Display Length for use in Screens or Reports. When a table is defined, the Global Field table can be accessed and all required fields selected from it. This guarantees that consistent usage will be preserved. In addition, a system control table is provided which allows an installation to specify the degree of enforcement of these stadards which is to apply.

However, this facility is not used widely, mainly because it depends on an effective DBA role with authority to manage the process. In addition, changes to the Global Fields table may require a review of all existing table definitions and this may be impracticable in the absence of resource.

3.3.7 Data Dictionary

In order to assist in managing an effective level of field consistency, ISL provide a Data Dictionary tool which build and maintains a cross-reference of Field to Table usage, flagging those fields where inconsistent usage had been detected and allowing the details of usage for a specific field to be shown.

3.4 Primary and Secondary Indexes

TDS and most other tables must have at least one Primary Index field (which must be the first field in the definition) and can have up to 16. The total length of the key fields must not exceed 127 bytes. Each data row must have a different combination of values for these fields. Primary Keys are identified in the Table Definer by setting a **KEYTYPE** of **P**. Primary fields must be contiguous – that is, if there are, for example, 5 these must be fields 1 to 5.

When data is stored it is indexed by the first key field which allows rapid retrieval of rows when this field is used in the retrieval selection. The importance of this in optimising performance is described in the later section on Performance Implications.

OSB™ also supports Secondary Indexes which can be defined on any field other than the first, including other Primary Keys. Secondary keys are identified in the Table Definer by a **KEYTYPE** of **S** or **Q** if they are also Primary Keys. Up to 16 Secondary Indexes can be defined for a table. Note that if a Secondary Index is specified for a numeric field it cannot contain null values.

Secondary Indexes are used by the DOB in data retrieval as an alternative to scanning a whole table. The conditions when this occurs depend on the precise situation involved in the retrieval selection.

3.4.1 Defining a Secondary Index

Secondary Indexes cannot be defined directly from the Table Definer. They can be defined or removed by a feature available to Level 7 Users only.

If a table is empty and Secondary Indexes are defined, they will be populated automatically as each row is inserted, replaced or deleted. If the table already contains data, or if additional Secondary Indexes are to be added to a populated table then they can be populated for existing rows either by running the **SIXBUILD** tool from a Level 7 session if the data is less than a specified size or by running the **HRNBRSIX** batch utility with the segment containing the table offline. The advantage of the batch utility is that it can build multiple Secondary Indexes in one pass; if this is done, the **KEYTYPE** values are added to the table manually afterwards by editing the **FIELDS** table for the table to be updated.

If a Secondary Index build fails (for example, if a null value is detected in a numeric field), the **KEYTYPE** value in the table definition changes from **S** or **Q** to **s** or **q**. If this is detected, the error cause should be corrected and the index data then be rebuilt with the batch utility described above.

3.4.2 Secondary Index Structure

A Secondary Index entry consists of the Secondary Key value, the Primary Key value and the page number of the data page containing the row to which it refers. This allows direct access to the corresponding row or rows by reference to a Secondary Index value.

3.4.3 Secondary Index Management

When Secondary Index data is updated by adding rows, deleting rows or modifying an existing row the stored Index data is modified accordingly. Under certain circumstances, the page number value cannot be updated. When this happens, subsequent accesses use the Primary Key value as the route to accessing the required row. Over a period of time, this can lead to the efficiency of retrieval diminishing.

If this happens, the Secondary Index so affected should be rebuilt as described above.

4 Data Retrieval Options

4.1 Data Retrieval

Since each data occurrence is identified uniquely by a specific table name and a parameter specification if it is parameterised, data retrieval in OSB™ is extremely simple.

There are two basic verbs – one to retrieve a single row and one to process a complete table or selection of rows. Both refer to the table name or a specific occurrence without any further data declarations being required in the actual code.

This allows considerable flexibility in data management as the actual source for a code process can be changed simply by modifying the data type and details of the table definition used. In addition, the ability to use an indirect reference as the table name allows generic data-driven processing to be developed very easily.

4.1.1 GET

The basic format is

GET *tablename*

which will retrieve the row with the lowest key set (unless the definition specifies ordering). The actual row to be retrieved may be determined by qualifying the **GET** with **WHERE** clauses as described in the next section of this guide. Note that for a parameterised table the format is

GET *tablename*(*p1,..*)

If the table is empty or no row matches the selection criteria a **GETFAIL** exception is raised which may be trapped by either the generic

ON GETFAIL:

or the specific

ON GETFAIL *tablename***:**

as well as any parent exception in the defined standard hierarchy.

4.1.1.1 GET after INSERT DELETE or REPLACE

Because of the way that OSB™ processes updates to permanent data – in particular **TDS** – special consideration needs to be given when a row that has been inserted or replaced within the SAME transaction is to be retrieved (see also the section on the **COMMIT** verb later).

The new or modified row is at that point held in the Intent List and has not been written to the physical table. If a **GET** is issued with ALL primary key values specified in a **WHERE** clause the row in the Intent List is retrieved. If ANY key field value is omitted from the statement an attempt will be made to retrieve the row from the actual physical source instead. This can result in unexpected **GETFAIL**s or field values in the data buffer which do not reflect recent changes. Equally, if a row has been deleted but not committed a non-unique **GET** may retrieve the deleted row.

4.1.1.2 GET WITH MINLOCK

A new option has been added at Release 5.2 to limit the scope of locking when the selection is not unique by primary key. The **Processing** manual contains the following description:

If a GET statement specifies a row that is unique by primary key, then a share lock is taken upon that row. If a GET is not unique by primary key, then a share lock is taken on the table (or table instance in the case of a parameterized table). If the keywords WITH MINLOCK appear at the end of a GET statement, and either the GET is ordered by anything but the primary key, or the GET includes selection that is not unique by primary key, then a share lock will be taken on the table (or table instance in the case of a parameterized table) only during GET processing. Once the row to be returned has been determined, the lock will be reduced to a share lock on only that table row.

4.1.2 FORALL

The basic format is

```
FORALL tablename.....
        code
        code
        ...
END;
```

which will retrieve all rows in the table (or occurrence as described for **GET**) and execute the code placed before the terminating **END** statement.

If no data exists (or meets any selection criteria) the included code is not executed at all and processing continues with the next selected statement following the **END**.

FORALL can be qualified with **WHERE** clauses as for **GET**, the order or retrieval modified by **ORDERED** statements and early termination specified by **UNTIL** statements, all of which are described in the next section.

4.1.3 FORALLA/B/E

Data-driven processing may require that the specification of a data retrieval has to be constructed dependent on the actual run-time conditions. It is impracticable and inefficient to try to code explicitly all possible optional **GET** or **FORALL** statements in such a situation.

OSB™ provides a generic retrieval option which allows the user to build the required information dynamically and then retrieve either a single row or a set of rows to match the required specification.

There are 3 functions provided to do this.

FORALLA(*tablename,parmspec,wherespec,orderspec***)**

retrieves the first row matching the specified arguments or if none exists raises the **ENDFILE** exception.

Further rows can then be retrieved if multiple rows are required by coding

```
UNTIL ENDFILE:
      CALL FORALLB(tablename);
      code
      code
      ...
END;
```

Once all required rows have been retrieved the table is closed by calling

FORALLE(*tablename***);**

This statement should follow a **FORALLB** loop and also be present in any **ON ENDFILE:** block.

4.1.3.1 TAM Parameters

The parameters for **FORALLA** can be constructed manually if required. They use a form of Reverse Polish Notation as described later in this guide. A null parameter indicates that no qualification for that part of the selection is required.

Note that if invalid parameters are passed to **FORALLA** an untrappable fatal error will occur.

4.1.3.2 PARSE_TAM

A simpler way to build the **TAM** parameters is use the

PARSE_TAM(*tablespec***)**

tool. This populates the fields of the TAM table buffer with the required values which can then be used directly in the call

FORALLA(TAM.TNAME,TAM.PSTR,TAM.WSTR,TAM.OSTR);

If an invalid *tablespec* value is used the **PARSER_ERROR** exception is raised.

The *tablespec* string can be built by concatenating literals and field/parameter values together to build the required result. Remember that field and parameter values or the table name which are themselves character strings must be enquoted either by including pairs of single quotes in the string or by using the **QUOTE** tool. An example could be

S = *name* **|| 'WHERE KEY = ' || QUOTE(***keyval***) || ' & SEL > ' ||** *selval*
CALL PARSE_TAM(S);

4.1.4 @FORALLA

This is a version of **FORALLA** introduced with Unicode support for situations where any table parameter or selection criterion is 100 characters or greater. Note that an error in the parameters will cause a fatal error as for **FORALLA**.

4.1.5 @READDSN

The **@READDSN** tool is used to retrieve a row from an external file as a character string. This must then be deconstructed by the user as required. Note that the default behaviour of this tool differs for Windows and Solaris access from that on a mainframe node. This is because it is used within the Promotion system and therefore always accesses data in EBCDIC format and cannot therefore read Windows text files directly.

The file from which the record or records are read is specified by a preceding **@OPENDSN** call.

4.1.6 Use of **MAP** tables

MAP tables provide a simple method of processing data read from **IMP**ort files or by **@READDSN** which contains more than one record format.

Define a series of **MAP** tables where one has a **KEY** field and either a single data field large enough to hold the input records or two or three fields to define the record identifier and the remaining data; the non-identifier fields should have **SYNTAX** of **V**. Then define one **MAP** table for each possible record format specifying the required fields as offsets into the record buffer.

At the start of the processing code, get the memory address to use by retrieving or initialising the **SESSION** occurrence of **@MAP** with **SIZE**

set to the required buffer value. This will provide the **ADDRESS** value to use as the key for storing and retrieving each input row.

Process the input row by row, storing the data in the base **MAP** table by use of **INSERT**. Extract the data from this buffer which identifies the record type and then use this value to **GET** the data in the appropriate format using the memory address obtained above.

4.2 Indirect Addressing

As noted earlier, the argument for a **GET** or **FORALL** can be an indirect reference. This allows the actual table to be processed to be determined by code checks or by control data values. This is of particular importance in that it allows applications to be designed where each user or group of users can have their own set of data tables but access them with common code.

4.2.1 Rule Arguments

The name of the required table can be passed as a rule argument. If this form is used the **GET** or **FORALL** uses a simple value as it does for a specific table name except that it is a parameter and not a literal. This form cannot be used to access data with a simple value defined as a variable value within the rule which is calling the **GET** or **FORALL**.

4.2.2 Coded Access

An alternative is to assign the required table name to a 16-character Identifier field in a table buffer. Any table which is not currently in use can be used for this purpose as no modification takes place to the data in that table. The **TABLES** table itself is often used for this purpose – for example:

```
TABLES.NAME = 'XYZ';
GET TABLES.NAME;
```

This usage, like the rule argument usage above, means that an explicit **GETFAIL** condition (**ON GETFAIL** *tablename*) cannot be used to trap unsatisfied calls; the generic **ON GETFAIL** must be used and if any possibility of multiple use exists the actual table causing the failure must be identified by code – for example, by setting a variable before the call and testing it in the exception.

4.2.3 Table-Driven Access

This is variation of the above usage where the required table value is taken from a field of a control or reference table which itself has been retrieved. The technique mentioned earlier of providing alternate data sources for different users depends on this type of process.

5 Data Retrieval Processes and Options

An unqualified **GET** will retrieve the first row in the selected table which has the lowest key values or the first row determined by any **ORDER** values. An unqualified **FORALL** will retrieve all rows in the selected table in the same order.

Both verbs can be qualified by a **WHERE** clause which determines one or more sets of field evaluations which determine which rows in the selection process meet a particular set of criteria. The order in which the retrieved rows are returned to the user can be modified by use of the **ORDERED** clause. For a **FORALL** loop early termination can be made by using the **UNTIL** clause to determine one or more termination conditions.

5.1 WHERE Clauses

5.1.1 Structure Options

A **WHERE** clause consists of one or more sets of evaluations consisting of a field or parameter name, a logical comparison and a value. If more than one set exists they can be separated by **&** (**AND**) operators which requires all of the conditions to be met or **|** (**OR**) operators which define alternate sets of conditions any one of which will satisfy the selection. A combination of these operators can be used. The **LIKE** comparison can be negated by preceding the field or parameter name with a **^** (**NOT**) operator. Evaluation sets can also be grouped into subsets by enclosing in parentheses (**(** and **)**) . Such groups can themselves contain further groups and can also be negated by a preceding **^** operator.

5.1.1.1 Operators

The allowed operators are

=	equal to
^ =	not equal to
>	greater than
<	less than
> =	greater than or equal to
< =	less than or equal to
LIKE	character ‘wild card’ match

5.1.1.2 Values

The format of the value in an expression depends on the characteristics of the field to which it is compared. Values can be unquoted numerics, quoted strings (including those containing ‘wild cards’ (***** or **?**) for **LIKE** expressions), a variable or rule argument value, a *table.field* value or a reference to another field in the table being processed in the format **.field*.

5.1.1.3 Basic Selection

The simplest form of **WHERE** consists of a single evaluation expression. This will either for a **GET** retrieve the first row that meets the selection criterion or for a **FORALL** will return all rows meeting it. This format creates a single test set which is either satisfied or not.

A simple test can also consist of a number of evaluation expressions separated by **&**. This will either for a **GET** retrieve the first row that meets all the selection criteria or for a **FORALL** will return all rows meeting them. This format also creates a single test set which is either satisfied or not.

A simple multiple-choice selection consists of a number of evaluation expressions separated by an **|** operator but without any **&** operators or parentheses. Each expression is treated as an alternate test so that a **GET** statement will return the first row that meets ANY of them while a **FORALL** will return all rows that meet any (or more than one). The clause is translated into a number of test sets, one for each expression.

More complex selection clauses require an understanding of how they are interpreted and executed as described in the following sections.

5.1.2 Reverse Polish Notation

Reverse Polish Notation is a method of representing algebraic statements in a form which can be easily interpreted by a mechanical device such as a computer. It was devised by the Australian mathematician Charles Hamblin in the 1950's. An excellent description of this process and other topics can be found in the online Wikipedia at http://en.wikipedia.org/wiki/Reverse_Polish_Notation.

An example of this notation is the expression 3 + (4 * 7) which is entered in RPN as 3 4 7 * +. OSB™ uses a form of RPN both to interpret **WHERE** clauses and also in the generated **TAM** strings used by the **FORALLA** tool.

5.1.3 Interpretation of Complex Clauses

A conditional clause which contains **&** and **|** operators but with no parentheses is interpreted as follows:

1. Process the expressions from left to right in order.
2. When an **|** operator is encountered start a new test set.
3. When an **&** operator is encountered add the following expression to ALL existing test sets.

An example of this is

WHERE *field1* **=** *value1* **&** *field2* **=** *value2* **|** *field1* **=** *value3* **&** *field3* **>** *value4*

would result in the following 2 test sets:

field1 **=** *value1* **&** *field2* **=** *value2* **&** *field3* **>** *value4*

field1 **=** *value3* **&** *field3* **>** *value4*

Note that each test set contains a set of expressions connected by **&** - i.e. all must be satisfied for the set to be triggered.

5.1.3.1 Parentheses

Pairs of parentheses (**(** and **)**) are used to group expressions which are considered as a single compound expression so that their result becomes an effective expression in each set to which they are assigned. The rule editor will automatically remove any pairs which are redundant or add them if required (see later). If the set of expressions within the parentheses contains ONLY **&** operators and no nested parentheses in turn, the parentheses are redundant as all the expressions will be added to existing sets as in the previous simple example. If, however, the set contains one or more **|** operators then a multiple set of sub-expressions is generated which are added in turn to existing sets to create multiple occurrences. For example:

WHERE *field1* **=** *value1* **&** **(***field2* **=** *value2* **|** *field3* **=** *value3***)**

results in 2 test sets

field1 **=** *value1* **&** *field2* **=** *value2*

field1 **=** *value1* **&** *field3* **=** *value3*

Note that if the parentheses had been omitted

WHERE *field1* **=** *value1* **&** *field2* **=** *value2* **|** *field3* **=** *value3*

the result would have been the following 2 test sets

field1 **=** *value1* **&** *field2* **=** *value2*

field3 **=** *value3*

Parentheses can be nested to any required depth (subject to an overall limit imposed by the rule executor). Each lower-level group will in turn operate on all higher-level test sets to qualify them and replicate sets where appropriate. Great care must be taken when constructing complex multi-level structures in this way as the result can very easily be other than that intended. It is strongly recommended that parentheses should be used at all levels when defining such structures – the Rule Editor will remove any which are unnecessary.

5.1.3.2 ^ (NOT) Operators

The **^** (**NOT**) operator has two main usages. It can be used to negate a **LIKE** operator in a single expression or to negate a complete simple or complex expression within a set of parentheses. The **^** operator is ALWAYS followed by a leading parenthesis and will be added by the rule editor in the first usage where necessary. If the code is

^ *field1* **LIKE** *value1*

it will be shown as the following when the rule is saved and redisplayed

^(*field1* **LIKE** *value1***)**

The effect of an **^** operator on a group of expressions depends on the exact nature of the group structure. In certain cases where the group is a simple set of expressions separated by **|** the result is the same as the expressions without the parentheses but with inverse operators. An example is:

^(*field1* **=** *value1* **|** *field2* **>** *value2* **|** *field3* **^=** *value3***)**

which is the same as

field1 **^=** *value1* **&** *field2* **<=** *value2* **&** *field3* **=** *value3*

5.1.4 Example of Complex Selection

The following condition

WHERE *field1* **LIKE** *value1* **&** *field2* **=** *value2* **&** *field3* **LIKE** *value3* **|** *field2* **^=** *value4* **|(***field2* **=** *value4* **&(***field3* **LIKE** *value5* **&(***field3* **LIKE** *value3* **|** *field3* **LIKE** *value6***) |** *field3* **LIKE** *value6***)) |(***field2* **=** *value7* **&(***field3* **LIKE** *value8* **|** *field3* **LIKE** *value9***)) |(***field3* **LIKE** *value10* **|** *field3* **LIKE** *value11***) &** *field4* **LIKE** *value12* **&** *field5* **=** *value13*

generates 9 test sets as follows:

1: *field1* **LIKE** *value1* **&** *field2* **=** *value2* **&** *field3* **LIKE** *value3* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
2: *field2* **^=** *value4* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
3: *field2* **=** *value4* **&** *field3* **LIKE** *value5* **&** *field3* **LIKE** *value3* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
4: *field2* **=** *value7* **&** *field3* **LIKE** *value8* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
5: *field3* **LIKE** *value10* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
6: *field2* **=** *value4* **&** *field3* **LIKE** *value6* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
7: *field2* **=** *value4* **&** *field3* **LIKE** *value5* **&** *field3* **LIKE** *value6* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
8: *field2* **=** *value7* **&** *field3* **LIKE** *value9***) &** *field4* **LIKE** *value12* **&** *field5* **=** *value13*
9: *field3* **LIKE** *value11* **&** *field4* **LIKE** *value12* **&** *field5* **=** *value13*

If a row satisfies any of the above sets it will be returned to the caller.

5.2 ORDERED

ORDERED [ASCENDING/DESCENDING] *fieldname* **{AND ORDERED…}**

The **ORDERED** clause determines the search pattern for the result of a **GET** statement or the order in which rows are returned for a **FORALL**. It allows the default sequence determined by the key order and any Table Definition Ordered flags to be overridden and the records presented in a different sequence. The **ORDERED** clause can contain as many fields from the table definition as are required specified in the field precedence order and whether they are to be in ascending or descending order. If the **ORDERED** clause refers only to the first primary key in **DESCENDING** order then the records will be presented in reverse order with no sorting and if a table has its primary key defined as Descending then an **ORDERED ASCENDING** *primary key* clause with no other fields specified will result in the reverse.

5.2.1 Sorting

For all other **ORDERED** conditions the selected rows will be sorted by the most appropriate sort option; this will cause a processing overhead which will depend on the number of records to be sorted and the installation sort options.

5.2.2 Implications for **GET**

An **ORDERED GET** will almost always cause a sort to occur in order to determine the actual record to be returned from all those meeting any other selection criteria.

5.3 UNTIL

UNTIL *condition [table]* **{OR** *condition [table]* **{OR…}}**

The **UNTIL** clause of a **FORALL** statement – it is not valid for a **GET** - can specify one or more conditions which if met will cause the loop to terminate before the table end condition which is the default cause. If any of the specified conditions occur then the loop will exit. The values may be any exception – standard or user-defined – raised within the **FORALL** – **END** loop which is not trapped explicitly by the included code (excluding any embedded **EXECUTE**d sub-transactions). Exceptions which can be qualified with a specific table name can be coded in either form.

5.4 Performance Implications

The simplicity and flexibility of OSB™ can itself become a problem. It is possible to build good applications faster in OSB™ than in almost any other language. It is also easy to build bad systems faster…

When designing data access routines there are a number of factors to consider. These are offered as guidelines as there are often many ways of approaching the design of a solution which may depend on installation-specific factors such as naming standards and transaction design requirements. However, the following are general points to consider in the light of local needs.

5.4.1 Data Persistence

OSB™ supports both explicit and implicit data persistence. By this, we mean retention in storage of data which is to be reused so that the overhead of repeated I/O activity is reduced to a minimum.

5.4.1.1 Binding

OSB™ supports binding of both definitions (which are simply control data) and data. This means that when a table is accessed either the definition components or both the definition and the actual data are read into storage in a system area and retained. Note that data cannot be bound unless the definition is also bound. This option should be considered for frequently-used application table definitions and also for static control data which is frequently used but has relatively few rows. There is an overall limit to the amount of data that can be bound and care must be taken in determining priorities so as to obtain the maximum benefit from its use. It must also be remembered that bound definitions will remain unchanged to the user even if updated elsewhere. Bound data should be static – that is, updated only as an explicit maintenance activity. With these restrictions, significant performance improvement can be gained by using this option.

5.4.1.2 Memory Retention

OSB™ also has a transient retention facility in that definitions and data which have been read into storage can be re-used without further recall until either modified or overlaid by further data. This means that in particular circumstances repeated operations appear to speed up as the fetch overhead is excluded from the later invocations. This can be seen best from the Workbench by running a routine a number of times. Usually, the first execution has a delay while data including definitions is retrieved then if the same operation is performed without any other process intervening further executions perform much faster.

5.4.2 Reducing DOB Calls

Each time a **GET** or **FORALL** is issued a call is made to the relevant DOB to extract and return the appropriate row or rows. The more this

can be reduced in number the better will be the performance of the routine.

This can be used to advantage in the design of data access routines which contain nested **FORALL**s. If, for example, there are two tables involved, one with a few rows and the other with many and linked by common fields, then using the smaller table as the outer loop will reduce the number of calls to the DOB. For example, if one table contains 10 rows that meet the selection criteria and the other 1000, then the number of calls will be 11 (1 for the first table and 10 for the second) as opposed to 1001 (1 for the first table and 1000 for the second) if the tables are nested the other way. This schema can be applied to multi-level nesting in the same way. However, the benefits may be outweighed if it is not possible to use the first Primary key for the inner loop or loops because of the retrieval overheads mentioned in the following sections.

5.4.3 Mode

A simple rule to observe is 'Run in Browse Mode wherever possible'.

This is because it not only reduces the possibility of locking but as part of this reduces the 'handshaking' message traffic to and from the DOB which can be considerable. There is a trace facility which can be used to report this traffic where a performance issue is being investigated.

Where it is required to update persistent data an Update sub-transaction is used. This does, however, raise a possible synchronization issue which is discussed later in the Coding Techniques section.

Note that OSB™ now has a **$TRXMODE** function which returns the current mode and can be used in generic routines to determine if a further transaction level is required to perform update actions.

5.4.4 Table Sweeps

The earlier section on **WHERE** clauses describes how these clauses – especially complex ones – are interpreted. The way in which the DOB processes such a call depends on the fields used in the selection.

If ALL test sets generated by the deconstruction of the **WHERE** clause include at least one test against the first Primary key then the index pages will be used to access those rows with matching key values. This results in a relatively fast access. If the index key is not referenced in ANY set a check is made to determine if a Secondary Index is referenced and if so, this will be used. If not, a sweep of the entire table or occurrence will result. For a large table this can take a significant time.

However…

The DOB analysis routines are complex and best described as idiosyncratic (or 'The Piece of Code that Passeth Understanding'…) and the way that any particular request is processed depends on both the selection criteria as explained previously and also – where a Secondary Index is involved – the actual distribution of data in this index. The DOB routines may decide to switch to a full table sweep part-way through a Secondary Index search if the data distribution and index state is such that a full sweep is actually faster (or considered to be).

There is one exception to the direct retrieval of an equated primary key that should be noted. If the primary key is a character field, either fixed or variable, containing numeric values then the following applies:

- If the value is enquoted (e.g. **WHERE** *primarykey* **= '1'**) a direct retrieval will occur if this explicit value exists
- If the value is NOT enquoted (e.g. **WHERE** *primarykey* **= 1**) a complete table sweep will occur as this value can match any value which consists of the stated value and any number of leading zeros (e.g. 1, 01, 001 , 0001 etc)

One other point to remember:

While a **GET** will cause a search for the first row matching the search criteria, an **ORDERED GET** is converted in the DOB into a full **FORALL** search so that the matching row meeting the ordering criteria can be returned. This can cause an unexpected table sweep in many cases and the implications should be considered when designing access routines.

6 Data Modification

There are three verbs which allow data to be updated. All can refer to an explicit table name, a rule argument or and indirect address of the form table.field.

6.1 INSERT

This verb takes data in the buffer for the required table and inserts it into that table. For volatile data types such as **SES** and **TEM** the data is inserted immediately. For an **EXP**ort table the data is written out to the external file. For a **TDS** table the data is placed into the Intent List (described later in this guide) and written to the physical table on either transaction end or an explicit **COMMIT** instruction. For Legacy data the row is passed to the external database handler for updating as determined by the Gateway settings.

The format is one of

INSERT *tablename* (or *rule argument* or *table.field*)

INSERT *tablename***(***p1*,..**)** for parameterised data OR

INSERT tablename **WHERE** *p1* = *parmva*l **&**... as an alternative

If the row already exists an **INSERTFAIL** exception is raised.

If the target table is a **TDS** or Legacy Data type the transaction must be executing in Update mode or an **INSERTFAIL** exception will again be raised.

If the target table or occurrence is locked then a **LOCKFAIL** exception will occur.

6.2 DELETE

This verb deletes the row in the required table identified by the keys in the buffer for that table. Alternatively, the full key specification required to identify the row can be specified in the **WHERE** clause of the **DELETE** instruction without a preceding **GET**. For volatile data types such as **SES** and **TEM** the data is deleted immediately. For a **TDS** table the delete request is placed into the Intent List (described later in this guide) and the row removed from the physical table on either transaction end or an explicit **COMMIT** instruction. For Legacy data the keys are passed to the external database handler for deleting as determined by the Gateway settings.

The format is one of

DELETE *tablename* (or *rule argument* or *table.field)*

DELETE *tablename***(***p1*,..**)** for parameterised data OR

DELETE tablename **WHERE** *p1* **=** *parmval* **&**... as an alternative

or the above with **WHERE** *key1* **=** *val1* **&** *key2* **=** *val2* etc

If the row does not exist a **DELETETFAIL** exception is raised.

If the target table is a **TDS** or Legacy Data type the transaction must be executing in Update mode or a **DELETEFAIL** exception will again be raised.

If the target table or occurrence is locked then a **LOCKFAIL** exception will occur.

6.3 REPLACE

This verb takes data in the buffer for the required table and replaces the non-key fields in the corresponding row in the table. For volatile data types such as **SES** and **TEM** the data is updated immediately. Rows in an **EXP**ort table cannot be replaced. For a **TDS** table the modified row is placed into the Intent List and written to the physical table on either transaction end or an explicit **COMMIT** instruction. For Legacy data the row is passed to the external database handler for updating as determined by the Gateway settings.

The format is one of

REPLACE *tablename* (or *rule argument* or *table.field*)

REPLACE *tablename***(***p1*,..**)** for parameterised data OR

REPLACE *tablename* **WHERE** *p1* **=** *parmval* **&**... as an alternative

If the row has not been retrieved into the data buffer previously or the key fields in the buffer do not match an existing row in the table a **REPLACEFAIL** exception is raised.

If the target table is a **TDS** or Legacy Data type the transaction must be executing in Update mode or a **REPLACEFAIL** exception will again be raised.

If the target table or occurrence is locked then a **LOCKFAIL** exception will occur.

6.4 @WRITEDSN

This is the output equivalent of **@READDSN** and writes the data in the string or field referred to in the call to the external file currently opened by the **@OPENDSN** tool.

6.5 Data Update

While data in volatile tables such as **TEM** and **SES** is updated directly, **TDS** data is held in a work area until either the current transaction ends or an explicit **COMMIT** call is made. Until one of these events happens no changes have been made to the physical data. Coding considerations relating to usage of data rows modified in a transaction but not yet completed are described earlier in this guide.

It is essential that the elements of the data update process described in this section are taken into consideration when designing data modification processes, whether online or batch. In particular, the Unit of Work concept described in the Transaction Design section is critical in assuring data integrity in the case of code or system failure.

Note also that if Trigger Event Rules are used to perform linked updates the data rows from these associated processes are also included in the Intent List requirements and must be taken into consideration when designing update transactions. This is especially important when modifying an existing transaction to add a Trigger process as this may cause a previously-successful process to overflow the Intent List size.

6.5.1 The Intent List

Earlier releases of OSB™ had a fixed Intent List size which constrained the ability to build effective Units of Work for complex application transactions. Current releases allow the Intent List size to be defined as being from 16k to 32k and application designers should be aware of the current value specified for their installation. In addition, it is important to know whether the same value applies to all nodes as this may affect the behaviour of code at different stages of development.

6.5.2 Transaction Completion

When all code which has been referenced after the execution of an **EXECUTE** statement has completed and the transaction therefore completes and returns control to the next higher level any rows held in the Intent List are written to the physical database.

This ensures that all associated data where a transaction modifies more than one permanent table is updated at the same time so that in the event of failure the data structure referential integrity is preserved.

6.5.3 Explicit Updates - **COMMIT**

OSB™ allows explicit completion of updates by using the COMMIT verb which will cause all pending changes to be written to the physical database within a transaction. This may be necessary where the logical data update being performed exceeds the capacity of the Intent List.

This should be used with great care to avoid possible loss of data integrity where failure occurs after part of a related set of updates have been written but before the rest have been done. This includes the possibility that one or more of the target tables has been locked by another user. Wherever possible, transaction and data design should ensure that each logical data update can be performed in a single operation.

An example of where **COMMIT** can be used safely is where the data to be updated in a single table consists of a number of rows which are passed into the update transaction in a SES table. The update code then performs a **FORALL** on this table calling the update code for each row, deleting the input row after this has been done and issuing **COMMIT** after a specific number of rows have been processed so that the Intent List size is not exceeded. This process can then be safely restarted in the event of failure, especially locking, as the **SES** table will only contain unprocessed rows.

6.5.3.1 COMMITLIMIT

If the Intent List size is exceeded the **COMMITLIMIT** exception is raised. This can be trapped and processing continued by calling **COMMIT** followed by the same operation (**INSERT**, **REPLACE** or **DELETE**) to the SAME table for which the exception was raised. This technique can be used safely with simple updates where a retry can be made without any possibility of incorrect data modification but must be done with great care.

IMPORTANT: If any of the tables to which updates are being written have Trigger rules which in turn generate updates there is a high risk of compromising data integrity as the exception may occur on either the base table or an associated Trigger table in which case the re-execution of the failing call may cause the primary data to be modified incorrectly. This can occur because any updates in the Trigger rules are processed before the base table change. If the **COMMITLIMIT** exception is raised by this update, the retry in the **ON COMMITLIMIT** will result in the Trigger changes being repeated. Unless the Trigger rule code specifically allows for this situation then incorrect results can result.

Note also that the **COPYTABLE** utility supplied as a standard tool uses the **ON COMMITLIMIT: COMMIT** technique. It should NOT be used to copy data which has triggered updates unless the above code precaution has been made.

The **$GETOPT** functions has two values which can assist in determining in code whether the Intent List size is about to be exceeded.

$GETOPT('COMMITSIZE') returns the maximum Intent List size
$GETOPT('COMMITUSED') returns the current size.

6.5.4 ROLLBACK

The **ROLLBACK** verb clears all current pending changes from the Intent List. It should always be used where an update transaction has failure recovery processes and further modification can occur in the same transaction in order to ensure that no change elements of the withdrawn update package remain and are then processed with the next set of changes in the same transaction.

6.5.5 Locking

A full description of the way OSB™ manages locking is included in the distributed Processing manual. The following is a brief overview of this.

In order to guarantee data integrity, any data access, whether retrieval or update, raises a lock condition to prevent any other process from making changes which could compromise this integrity.

There are two types of locks, Shared and Exclusive. The type of lock raised depends on both the current access mode (Browse or Update) and whether the data access is retrieval or update.

Locks are ALWAYS held until the current transaction has ended and cannot be released manually.

6.5.5.1 Browse Mode

A Shared lock is raised on the table definition (and for a **SUB**view on the source table definition) when a **GET** or **FORALL** is issued to any table or an update is made to a non-persistent table type (such as **TEM**) to prevent the definition(s) being changed during the process. The data is not locked.

6.5.5.2 Update Mode

A Shared lock is again raised on the table definition(s) as described above.

A **GET** or **FORALL** with a **WHERE** clause that specifies equality for ALL Primary keys will raise an Exclusive lock ONLY on that specific row. Any other **GET** or **FORALL** will raise an Exclusive lock on the complete occurrence. The two exceptions are where a Browse Mode **SUB**view is used when NO locks are taken on the data and on access to a **CLC** table when a Shared lock is taken on the occurrence.

A **DELETE**, **INSERT** or **REPLACE** will upgrade the Shared lock taken when the data was retrieved to an Exclusive lock for the WHOLE occurrence.

6.5.5.3 Legacy Data

When accessing Legacy data in Browse mode, no data locks are taken and no updates can be made. In Update mode the locking actions taken depend on the specific options used within the Legacy Gateway. The

relevant published Gateway manual should be used as a reference for this.

6.5.6 Recovery

OSB™ has as an integral part of its design a very effective recovery process which if used correctly guarantees data integrity in the case of a system or hardware failure.

Any data updates which have not been written to the physical database in such a case are held in the Redo Log which is processed each time the DOB is started. This means that before any further activity starts all pending updates are cleared. The only potential exception to this relates to Legacy data where additional steps may have to be taken to guarantee updates to the external databases.

6.6 Distributed Data

If Distributed Data has been implemented so that one or more remote nodes can be accessed from the current one, data in the other nodes can be accessed by means of the **LOCATION** parameter described earlier. There are three main benefits provided by this feature.

1. Data can be maintained on a single node but accessed from others directly. For example, where an application is implemented at a number of different sites, data at one site can be viewed or merged with data from other sites to provide high-level reporting.
2. Data accessible directly only from a particular platform (e.g. Legacy data) can be accessed by applications running on a different platform which does not support this data type directly.
3. Objects or data in different nodes can be compared directly to ensure consistency or to identify differences. Impulse Solutions Ltd provides a number of tools specifically designed to support this requirement.

6.6.1 Definition

In order to access remote data, the local definition of the table to be accessed must have a **LOCATION** parameter. **SUB**views of remote tables can be defined and accessed in the same way.

6.6.2 Access

When a data table for which a **LOCATION** parameter has been defined is accessed without a value for this parameter, data on the local node is used as in the case where no **LOCATION** parameter is defined.

If data on a specific node is needed this should be added to the call either as the last (or only) parameter or in a **WHERE** clause.

For example:

GET *table*(*location*)

GET *table*(*parm,…,location*)

GET *table* **WHERE LOCATION =** *location*

GET *table*(*parm,..*) **WHERE LOCATION =** *location*

If the requested node is not active or does not exist the **SERVERFAIL** exception is raised. If the data on the selected node cannot be accessed the **SERVERBUSY** exception is raised. If data exists in both the local and remote node, both copies must be available at the time of the call.

Note that if a **LOCATION** parameter has NOT been defined access will fail with a condition dependent on which of the above call options has been used.

6.6.3 Consistency

Although Minimal Definitions as described earlier can be used where data is available ONLY on a remote node, there may be good reasons related to Promotion consistency for using a full definition on the calling node. Although consistent definition across all nodes is very important to guarantee integrity, for distributed data it is ESSENTIAL that the local and remote definitions are consistent. It is recommended that a full definition is created in the development environment and promoted to all other nodes in the development chain; where a minimum definition is required the definition should then be modified to change it into this format.

6.7 Event Rules

OSB™ has a very powerful feature which can be used to automate a number of key data-related processes. This is the ability to define Event Rules as part of table definitions. These have control flags which determine for which aspects of data access or modification they are to be run and which type they are. There are two types – Validation and Trigger – which are described in the following sections.

Note that until Event Rules have been promoted to the SITE Library the Local Library in which they are held must be used for any testing of related processes and, in addition, certain standard workbench functions cannot be used directly as they use a search path which bypasses the Local Library. The two most affected are the **BR** and **ED** menu options. When testing Event Rules it is necessary instead to execute directly the actual rules called by these menu commands; these are **STEBROWSE** and **STE** respectively which take a single argument of the table name or

an enquoted table specification string for an occurrence of a parameterised table.

6.7.1 Validation

A Validation Rule is called when the data action for which it is specified is performed and is used to validate the data conditions that currently exist. It must be a Function which returns the value Y if the validation process is successful or a message to be displayed if it is not when the **VALIDATEFAIL** exception will also be raised. It may in turn call other rules. Screen definitions can also contain Validation Rules but these are not within the scope of this guide.

A common usage is to check entered values which are in the current data buffer for the source table against reference data in other tables. Note that this is a situation where the use of Browse **SUB**views as described earlier can be of very great assistance in minimising both the possibility of locking and also process overhead.

6.7.2 Triggers

A Trigger Rule is called when the data action for which it is specified is performed. This is equivalent to – for example – an SQL Stored Procedure. Trigger Rules cannot be Functions although they may use embedded Function Rules. They allow additional code to be run automatically when a particular data action occurs.

A common use for Trigger Rules is when a data structure consists of a set of related data tables. Although the updates to these can be done in 'open code' Trigger Rules allow related updates to be done as a standard 'prepackaged' function.

6.7.3 Change Identification

In both types of Event Rule it may be necessary to compare a field value with the existing one when an update is taking place. An effective technique for doing this is to have both a **TEM** and Browse **SUB**view defined for the table to be checked. The current buffer is transferred to the **TEM** table buffer, a fully-keyed **GET** issued to retrieve the current version of the record (which will also show if it exists) and the fields compared as required. The updated data is then restored to the base table buffer to allow the action to complete.

6.7.4 Current Values

Although Event Rules can be written for specific base tables, there are many cases where a generic process is needed. In this case, two tools are provided to provide the necessary information.

EVENTTABLE returns the name of the table which has caused the event

PARMVALUE(*parmname***)** gives the current parameter values

6.7.5 COMMIT Implications

Where a Trigger Rule performs updates to additional tables, the new or modified rows are included in the current Intent List. This must be taken into consideration when designing Trigger processes to ensure that the installation limit is not exceeded by this.

6.8 UNTIL... END Loops

Although it was the original intention to have no traditional **UNTIL... END** loops in OSB™, there are two situations where this can be used effectively.

The first, which is outside the scope of this document, is in handling screen processing where the **UNTIL** *condition* format is used to allow repeated access to the screen until a termination condition is set.

The second is used in data access to support situations where a code process is repeated a number of times until a condition is raised and trapped either by the **UNTIL** or by an external **ON** routine.

6.8.1 Repetitive Processing

Where a code process is dependant on a data source which may change during the timespan of the process, this usage allows it to be repeated until a condition is detected that indicates that no further action is required.

6.8.2 Lock Management

This is a variation of the above for situations where an update process may fail because of a transient lock (for example, where the data table to be updated is in use by another process for a limited time).

An effective method of handling this situation is to use an **UNTIL... END** loop which calls an inner rule which in turn **EXECUTE**s the update code. When the update is successful the inner rule raises the condition to end the loop. The call to the inner rule is followed by an increasing delay using the **$SLEEP** function to allow the external lock to be released before retrying the process.

The inner rule should test the value related to the delay and, when it exceeds a specified value, cause a user-define exception to be raised which is trapped elsewhere as a genuine lock condition. The inner rule should have an **ON EXECUTEFAIL** exception with a **ROLLBACK** to allow the code to drop out to the outer rule for retry.

This method provides a very effective way of handling contention situations, particularly in an online application.

7 Coding Techniques

This section is not intended to be a comprehensive guide to the development of data access code but simply to highlight a number of areas where the performance of a routine can be improved or where techniques are available to assist with access issues.

7.1 Efficient Data Design and Access

When designing a database structure the key structure and relationship between different tables in a related group should be considered to allow retrieval wherever possible to be by the use of the first Primary key as part of any selection. While this is not always possible, this approach will optimise performance of the routine and can provide significant performance benefits.

7.2 Complexity v Inner Rules

If a complex selection is required in a **WHERE** clause it may be more effective (and easier to understand for maintenance) to use a simpler selection and filter the returned rows further in an inner rule which can test for exclusion conditions. The number of excess rows likely to be returned by the initial selection should be taken into consideration in designing the routine, however.

7.3 Multiple Use of a Table

The processing logic in a **FORALL** loop may require access to either another row in the current occurrence or a row in another occurrence if the table is parameterised.

This can be met by use of a 'mirror' **SUB**view table (or in some cases tables) which allow effective multiple access to the same data without losing position in the loop.

Note that this applies only to persistent data. Where a similar requirement exists when processing **SES** or **TEM** data it is necessary to define one or more matching or similar tables (depending on the field usage required) and copying the data from the main table before processing. This copy must, of course, be refreshed if the loop on the transient table is repeated.

7.4 Transaction Design

It has already been established that routines should wherever possible run in Browse mode until an update is performed by an Update sub-transaction. This can raise integrity issues especially in online applications where a retrieved row has been updated by another user before the current user performs the update.

A common technique to resolve this is to add a counter field to the definition of the table or root table for a structure which is updated each time the row is modified. The value of this field is stored by the calling application and passed into the update routine where the row is retrieved (even when inserting a row). The next counter value is also obtained from a control table which is updated to ensure no other user can get the same value; this is also passed into the update routine. This will first check that a row to be inserted does not already exist, that a row to be deleted still exists and an existing row has the same counter value as when retrieved. If the last condition is true the counter is updated with the new value passed in. If an error condition is detected the update routine completes without update and returns a value which causes the calling routine to retry the process; for an online routine this will usually return control to the screen handler with a message to allow the user to retry the action and which will usually repopulate the screen fields with the new values found.

Note that in Release 5 an **EES** table can be used to assist this process as described earlier in this publication.

7.5 Shadow Tables

The use of Secondary Indexes has been described earlier and these can provide an effective method of viewing data from a different perspective. However, where the data has a high volatility with frequent inserts, deletions and updates, this may not provide an efficient retrieval process.

An alternative in this situation is to define a ‘shadow’ table containing the same fields as the base table but keyed on the field(s) intended to be used for Secondary Indexes. This table can be managed by a Trigger process to ensure that each base row has an equivalent ‘shadow’ row. This technique then allows fast access in either usage by means of the primary keys of either the base table or the shadow copy.

8 Tools

In order to assist developers of data access routines, Impulse Solutions Ltd provide a number of tools to check for potential issues at coding stage. In addition to checking the consistency of data access statements in relation to the tables and fields referenced, they also examine conditional access clauses to determine if inefficient searches are likely to occur.

These tools are described more fully in the relevant Impulse Solutions Ltd User Guides.

8.1 TAM Analyser

This routine scans a rule for table-related TAM statements and validates them for consistency. If the table reference is to an explicit table it is checked to ensure that it exists and all field and parameter references are checked against the table definition. Any errors are reported including an incorrect number of parameters.

Any **WHERE** clauses are deconstructed into selection sets and if any are detected which are liable to cause a table sweep an error report is generated

8.1.1 Extended Rule Editor Line Command

A new line command **I** (IOCheck) has been added to the Extended Rule Editor to invoke this function.

8.1.2 Rule Set Analyser

This tool scans a set of rules selected by Library, Rule Name Mask and/or UNIT and reports on any where errors are detected.

8.2 Cross-Reference Auditor Extensions

A new table has been added to the Data Auditor output set to record all table-related TAM call elements and a checker routine provided which reports on errors in these.

8.3 Data Dictionary

Tools have been added to maintain the Data Dictionary by detecting new and modified table definitions and adding or updating the corresponding entries.

A further tool allows the user to view the usage of a specific field.