Polyhedra®
Real-Time Relational Database

Enea Polyhedra Ltd
Copyright notice

Copyright © Enea Software AB 2018. Except to the extent expressly stipulated in any software license agreement covering this User Documentation and/or corresponding software, no part of this User Documentation may be reproduced, transmitted, stored in a retrieval system, or translated, in any form or by any means, without the prior written permission of Enea Software AB. However, permission to print copies for personal use is hereby granted.

Disclaimer

The information in this User Documentation is subject to change without notice, and unless stipulated in any software license agreement covering this User Documentation and/or corresponding software, should not be construed as a commitment of Enea Software AB.

Trademarks

Enea®, Enea OSE® and Polyhedra® are the registered trademarks of Enea AB and its subsidiaries. Enea OSE®ck, Enea OSE® Epsilon, Enea® Element, Enea® Optima, Enea® LINX, Enea® Accelerator, Polyhedra® Flash DBMS, Enea® dSPEED, Accelerating Network Convergence™, Device Software Optimized™ and Embedded for Leaders™ are unregistered trademarks of Enea AB or its subsidiaries. Any other company, product or service names mentioned in this document are the registered or unregistered trademarks of their respective owner.

Manual Details

<table>
<thead>
<tr>
<th>Manual title</th>
<th>Real-Time Relational Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document number</td>
<td>manuals/rtrdb</td>
</tr>
<tr>
<td>Revision number</td>
<td>9.4.3</td>
</tr>
<tr>
<td>Revision Date</td>
<td>10th September 2018</td>
</tr>
</tbody>
</table>

Software Version

This documentation corresponds to the following version of the software:

| Version | 9.4 |
PREFACE

This document describes the Real-time Relational Database component of Polyhedra. The manual is aimed at Polyhedra system configurers and application writers. The section on Running the RTRDB is also useful to application users.

It is recommended that the reader also see the Polyhedra Overview and the Polyhedra User’s Guide prior to reading this document. It is assumed that the reader has knowledge of relational databases, SQL and object-oriented concepts.

Throughout the document the following notation is used:

- `font` Code
- `font` reserved words
CONTENTS

1. OVERVIEW OF RTRDB .......................................................... 9

2. THE POLYHEDRA FLASH DBMS RTRDB ...................................... 11
   2.1 Introduction .............................................................................. 11
   2.2 The Database File ................................................................. 11
   2.3 Flash DBMS Standby to Standard Master ................................. 12
      2.3.1 Restrictions ....................................................................... 12
      2.3.2 Compatibility ................................................................. 12
      2.3.3 Configuration .................................................................... 12
   2.4 Swap File Mode ...................................................................... 12
   2.5 Restrictions ........................................................................... 13

3. THE POLYVIEW READ-ONLY RTRDB (STANDARD ONLY) ............. 14
   3.1 Introduction ............................................................................. 14
   3.2 Restrictions ........................................................................... 14

4. SERVER FUNCTIONALITY ......................................................... 15
   4.1 SQL Procedures ...................................................................... 15
      4.1.1 The DDL restriction ......................................................... 15
      4.1.2 Run-time arguments ....................................................... 16
   4.2 Stored SQL procedures (Standard only) .................................... 16
      4.2.1 Storing an SQL procedure ............................................... 18
      4.2.2 Executing a stored SQL procedure .................................. 18
      4.2.3 Updating a stored SQL procedure ................................... 18
   4.3 Queries .................................................................................. 18
      4.3.1 Data set specification ....................................................... 18
      4.3.2 Procedures for fixed queries ............................................ 19
      4.3.3 Procedures for active queries .......................................... 19
      4.3.4 Schema changes and active queries ................................. 20
   4.4 Transactions .......................................................................... 20
      4.4.1 Data modification through SQL procedures (Standard only) 20
      4.4.2 Data modification through active queries ........................ 20
      4.4.3 Data modification failure ............................................... 20
      4.4.4 Conflict detection ......................................................... 21
   4.5 Locks .................................................................................... 22
   4.6 Multi-Threaded Execution ..................................................... 22

5. RUNNING THE RTRDB ............................................................. 23
   5.1 Configuration ......................................................................... 23
   5.2 Starting the RTRDB ............................................................... 23
   5.3 RTRDB Initialisation .............................................................. 23
   5.4 System Clock ......................................................................... 24

6. LIMITATIONS ........................................................................... 25
   6.1 Records ................................................................................ 25
   6.2 Tables ................................................................................... 25
   6.3 Columns ............................................................................... 25
   6.4 Connections .......................................................................... 25
   6.5 Polyhedra Flash DBMS File Size (Flash DBMS only) .................. 25

7. RESOURCES ............................................................................ 26
8. SCHEMA TABLES ................................................................. 34
  8.1 The Tables table ......................................................... 34
  8.2 The Views table .......................................................... 35
  8.3 The Attributes table .................................................... 35
  8.4 The Indexes table .......................................................... 36
  8.5 The IndexAttrs table ..................................................... 36
  8.6 The schema definition .................................................... 37
9. SECURITY ........................................................................... 39
  9.1 Enabling security .............................................................. 39
  9.2 Connections and sessions ................................................... 39
  9.3 Users .............................................................................. 39
  9.4 Ownership ...................................................................... 40
  9.5 Privileges ...................................................................... 40
  9.6 Configuration ................................................................... 42
  9.7 Configuration for ODBC ..................................................... 42
  9.8 Object queries .................................................................. 42
  9.9 Password security ............................................................ 43
    9.9.1 Configuration ............................................................ 43
    9.9.2 Level 1 protection ....................................................... 43
    9.9.3 Level 2 protection ....................................................... 43
    9.9.4 Setup ...................................................................... 43
  9.10 Summary of security tables ................................................ 44
10. RTRDB SUPPORT FOR DEVELOPING CL SCRIPTS .......................... 47
  10.1 Loading CL .................................................................... 47
  10.2 CL execution control ....................................................... 47
11. PERSISTENCE (STANDARD ONLY) .......................................... 49
  11.1 Journalling Disabled ......................................................... 49
  11.2 Journalling Enabled .......................................................... 49
  11.3 Memory mapped file persistence ......................................... 51
12. PERSISTENCE (FLASH DBMS ONLY) ....................................... 53
13. FAULT TOLERANCE ............................................................... 54
  13.1 Service Provided .............................................................. 54
    13.1.1 Queries ................................................................... 54
    13.1.2 Transactions ............................................................. 54
    13.1.3 Control Messages ....................................................... 54
  13.2 Faults Tolerated ............................................................... 55
    13.2.1 Software Faults ......................................................... 55
    13.2.2 Hardware Faults ......................................................... 55
    13.2.3 System Faults ............................................................ 55
  13.3 Clients ........................................................................... 55
    13.3.1 Call-back and ODBC API ........................................... 56
  13.4 System Architecture ......................................................... 56
  13.5 System Operation ............................................................. 57
    13.5.1 Start-up sequence ....................................................... 57
    13.5.2 Normal operation ....................................................... 57
    13.5.3 Activation ................................................................. 58
    13.5.4 Deactivation ............................................................. 58
  13.6 The Arbitrator ................................................................. 58
    13.6.1 Database arbitrator ..................................................... 58
    13.6.2 TCP Arbitrator Protocol ............................................. 60
  13.7 Configuration ................................................................. 62
14. DATA REPLICAATION (STANDARD ONLY) .............................. 63
  14.1 Configuration ................................................................. 63
  14.2 Standard Replica Configuration .......................................... 63
14.3 Replica Listen Mode .............................................................. 64
14.4 Connection ........................................................................ 64
14.5 Journal Service ................................................................. 64
14.6 Replication Depth .............................................................. 64
14.7 Loss of Connection ........................................................... 65
14.8 Shutdown ........................................................................... 65
14.9 Snapshot ............................................................................ 65
14.10 Historian ............................................................................ 65
14.11 Security ............................................................................. 65

15. DATA SUBSCRIPTION (STANDARD ONLY) ................................................. 66
15.1 Configuration ...................................................................... 66
15.2 Data Aggregation ............................................................... 67
15.3 Example .............................................................................. 67

16. LOCAL DATA ........................................................................... 68

17. HISTORIAN (STANDARD ONLY) ............................................................... 69

18. UNICODE .................................................................................. 70
18.1 Load and Save ..................................................................... 70
18.2 Journaling .......................................................................... 70
18.3 Security .............................................................................. 71
18.4 DVI ....................................................................................... 71
18.5 Historian ............................................................................. 71

19. MEMORY EXHAUSTION RECOVERY ...................................................... 72
19.1 Conditions That Will Trigger Recovery .............................. 72
19.2 Recovery .............................................................................. 72
19.3 Limitations ......................................................................... 72

20. PERFORMANCE METRICS ................................................................. 74
20.1 Configuration ...................................................................... 74
20.2 Operation ............................................................................ 75
  20.2.1 Finding Available Metrics ........................................... 75
  20.2.2 Reading a Metric ........................................................ 76
  20.2.3 Auto-generation of Metrics ......................................... 76
  20.2.4 Startup and Fault Tolerant Behaviour ....................... 77
20.3 The Providers ...................................................................... 77
  20.3.1 The SQL Provider ....................................................... 77
  20.3.2 The Client Provider .................................................... 78
  20.3.3 The Memory Provider ................................................. 78
  20.3.4 The Thread Provider .......................................................... 78
  20.3.5 The Journal Provider .................................................... 78
  20.3.6 The CL Provider .......................................................... 79
  20.3.7 The DVI Provider (Standard only) .............................. 79
  20.3.8 The Historian Provider (Standard only) ..................... 80

21. ERRORS .................................................................................. 81
21.1 Error Code Format .............................................................. 81
21.2 Error Text .......................................................................... 81
21.3 Error Codes ....................................................................... 81
1. Overview of RTRDB

This document describes the Polyhedra component called the Real-Time Relational Database (RTRDB). The RTRDB is designed for the storage of real-time data, characterised as being small in volume, but high in turnover, such as found in the area of process control applications. This document describes both the standard RTRDB but also the Polyhedra Flash DBMS RTRDB. See the next chapter for details.

This document is for use with all editions of the Polyhedra product. However, some of the sections in this document are not applicable to all editions of Polyhedra as they describe features that are not supported by all editions. The supplied feature.txt file lists the features supported by each edition of Polyhedra.

The RTRDB offers the following features:

- a relational object store
- memory resident data allowing very high speed storage, access and update
- an object-oriented data server
- a notification mechanism providing subscribers with information on data changes within the server
- Control Language (CL), an event-driven object-oriented programming language. CL is used to define methods on objects in the store. Methods are executed within the RTRDB itself
- ISO-based SQL, with extensions to support the object oriented features of the RTRDB
Overview of RTRDB

Real-Time Relational Database

Figure 1.1 The structure of the RTRDB

Figure 1.1 outlines the modules of the RTRDB. There is a Schema, which contains the definitions of the current database, defined by the Data Definition Language (DDL) of SQL. The Object store contains all the data that exists at run-time. This data complies with the data definitions in the schema. The remaining modules SQL, CL, and Server are event driven either from the outside or from data changes internally. New or modified data or definitions entered into the system through the Server will cause events to be sent to the Schema and the Object store to be updated. Data changes in the Object store and definition changes in the Schema may cause events to be sent to the other modules.

- The CL module consists of methods defined in Control Language (CL) that are associated with data in the object store.
- The SQL module provides an SQL language interface to the Object store.
2. The Polyhedra Flash DBMS RTRDB

2.1 Introduction

The Polyhedra Flash DBMS RTRDB is a member of the Polyhedra family of products targeted at stand-alone, low-power devices such as feature phones. Polyhedra Flash DBMS RTRDB differs from the standard Polyhedra RTRDB by storing data in a flash file, rather than in-memory. Its usage of RAM is therefore independent of database size and lower than the standard product.

Polyhedra Flash DBMS is designed for use with NAND flash, but will also work with disk-based file stores. The Flash DBMS RTRDB accesses the database file via POSIX file I/O functions. A Flash Disk API is also supported, allowing users to provide their own flash file routines.

The database file contains a number of fixed sized pages. A page cache is used to speed up access to the database. The page size, the number of pages in the file and the size of the cache can be configured as appropriate for the size of database, available RAM and flash device. The appropriate configuration resources are detailed in section 7.

The Flash DBMS RTRDB may also be initialised from a standard Polyhedra load file.

Polyhedra Flash DBMS shares many of the features of the standard product, including:

- SQL, ODBC and JDBC.
- ACID-compliant transactions.
- Client-server architecture with cross-platform support.
- Active queries to avoid the need for polling.
- CL (trigger language) for embedding business logic into the database.
- Portability of database files.

Not all features available in the standard product are available in Polyhedra Flash DBMS. Those unavailable are marked with “(Standard only)”. Those features unique to the Flash DBMS product are marked “(Flash DBMS only)”

2.2 The Database File

Polyhedra Flash DBMS stores table data in a disk file. Each transaction is saved to disk before the commit operation returns to the client. I/O to and from the file is performed in units defined by the flash_page_size resource. For optimum performance, this value should be a multiple of the underlying file system page size.

It is possible to specify the database storage file size (initial, limit and extent) in pages using the flash_file_size, flash_file_max and flash_file_extent resources. The file is created with a size specified by the flash_file_size resource.

During database operation, if more pages are needed, the number of pages specified by the flash_file_extent resource is added until the limit specified by the flash_file_max resource is reached.

On start-up and load of the database storage file, if the flash_file_size resource is present and the current size of the file is smaller than that specified by the resource, the file is grown until the size matches that specified by the resource. It is therefore possible to enlarge a file that has reached the limit by restarting the database with modified resource settings.

On start-up the Flash DBMS RTRDB displays both the total number of pages and the number of used pages in the database storage file. It is therefore possible to determine if and when modifying the
resource settings is required.

On start-up and load of the database storage file, if the Boolean `flash_shrink_file` resource is set to “true”, the file is reduced in size as much as possible. If the `flash_file_size` resource is also set, the file is not reduced to smaller than the size specified by the resource.

The Flash DBMS RTRDB implements column and table transience by deleting transient records and resetting transient column data on startup. Hence database startup time may be influenced by the amount of transient data existing in the database file.

### 2.3 Flash DBMS Standby to Standard Master

The Flash DBMS FT module supports the ability for a Flash DBMS RTRDB to operate as a standby to a standard master. This allows a standard FT system to be migrated to Flash DBMS without the need for a discontinuity of service.

#### 2.3.1 Restrictions

When a Flash DBMS RTRDB is operating as a standby to a standard master, the following restrictions apply:

**2.3.1.1 No Schema Changes**

The Flash DBMS RTRDB, when operating as a standby to a standard RTRDB, will not support any schema changes on the master. The Flash DBMS standby will output an error message and terminate in this case.

**2.3.1.2 Limited Functionality**

If the master RTRDB uses any functionality not supported by Flash DBMS (e.g. domains, arrays), the Flash DBMS standby RTRDB will output an error and terminate.

#### 2.3.2 Compatibility

The Flash DBMS RTRDB will operate as a standby to a standard RTRDB master of version 6.3 or later.

#### 2.3.3 Configuration

The Flash DBMS standby should be configured in the same way as a standard standby using the resources: arbitrator_service, journal_service and other journal_service as described in section 13.7.

### 2.4 Swap File Mode

The Flash DBMS RTRDB supports a mode of operation that improves transaction throughput at the expense of transaction durability. Normally, the Flash DBMS RTRDB synchronises changes made by a transaction to the database file such that after any system crash, the changes will be seen on restart. The synchronisation overhead can have an effect on transaction throughput. The swap file mode performs no transaction synchronisation and attempts to minimise any writes. Any changes are written back to a separate swap file and the initial database file is left untouched. A snapshot of the current state of the database may be performed using the “SAVE INTO” statement.

This mode is useful when it is desirable to start from a known position on startup and intermediate changes can be discarded.

The mode has the following characteristics:

- The mode is enabled by the `flash_swap_file` resource being set to the file to be swapped to.
If the swap file exists, it is used (the contents being discarded), otherwise the file is created.

- The swap file will be affected by the load_file_directory resource.

- The initial state of the database is taken from the load_file resource, however the file will not be written to. The file must be on a file system that allows random access.

- The swap file will be grown as necessary and in summation with the initial load file will be limited by the flash_file_max resource.

- Executing the SAVE statement will not update the original file specified by the load_file resource.

### 2.5 Restrictions

The following features present in the standard product are not present in Polyhedra Flash DBMS:

- DVI sub-system.
- Historian sub-system.
- Data replication using replica functionality.
- Data subscription.
- SQL CREATE DOMAIN, DROP DOMAIN and ALTER DOMAIN statements.
- SQL array data types.
- SQL stored procedures and the SQLPROCEDURES table.
- Modification of primary key columns, unless no references exists to them.
- User-created indexes on catalogue tables.
- CL OLD operator.
- Access from CL to instances of the Thread class.
- Local indexes.
- Unordered indexes. All user-defined indexes are ordered even if ORDERED is not specified.
3. The PolyView Read-only RTRDB (Standard Only)

3.1 Introduction

The PolyView component provides read-only access to databases constructed by the RTRDB component. The PolyView component may be started and accessed in the same way as the RTRDB with a number of restrictions. The main restriction is that no changes may be made to the data in the database.

Clients may access a PolyView component using the same methods (ODBC, Call-back, JDBC, SQLC, etc.) as with the RTRDB and may use the same SQL.

3.2 Restrictions

The restrictions in force when using PolyView are as follows:

- No transactions may be applied to the database. Transactions will be failed with an error code (0x0141) indicating that the database is read-only.
- The CL component is not available.
- The Data Port and Data Connection facilities are not available.
- The Debugger is not available.
- Fault tolerance is not available, as either a master or standby.
- Replication is not available.
- The GDI component is not available.
- The Historian component is not available.
- The \texttt{save} and \texttt{save into} SQL statements are not available.

Note that tables and data previously created by these components and saved into a load file will be viewable using PolyView.

If security is enabled in a load file, then security will be enabled when the load file is accessed by PolyView.
4. Server functionality

This section outlines the functions of a Polyhedra server. What is contained here is generally applicable to all Polyhedra servers. Although some servers may only offer a subset of the full functionality, the RTRDB offers the full functionality of a Polyhedra server.

The services described are only available to Polyhedra clients. A client may be a Polyhedra component such as the SQL Client (SQLC), or a user-supplied client based on the OBDC API, Callback API or JDBC API. It is common for clients to access only a part of the server functionality, depending on the needs of the client. The means of accessing any service from a client depends on the client and is described in the appropriate documentation.

In brief, a Polyhedra server offers the following:

- A server manages data. A client may request for some of that data. This is known as a query.
- A query that returns a single snapshot of data in the server is known as a fixed query.
- A query may additionally request notification of changes to data in the server. This is known as an active query.
- A client may request data to be changed in the data server. This is known as a transaction. All change requests in the same transaction are applied in an atomic action. They either all succeed or all fail.
- A client may use a fixed query to obtain a lock. A lock is used to either detect or prevent changes made to data in the data server by other clients.

4.1 SQL Procedures

Before we proceed, the following abbreviations need to be defined:

DDL  Data Definition Language. SQL statements that define or alter the structure of the database, such as create, alter and drop.

DML  Data Manipulation Language. SQL statements that modify the data stored in the database, such as update, insert and delete.

A procedure is a sequence of SQL statements separated by semi-colons. This sequence can be sent to the RTRDB to be executed in an atomic manner. Note that these procedures are not stored procedures. See the next section for more information on stored procedures. SQL procedures may be used within fixed or active queries during a client-server dialogue, but there are restrictions on the combination of SQL statements such procedures may contain. The restrictions are detailed in section 4.3. An example of a procedure is as follows:

```sql
update mytable set myname = 'Fred' where myid = 1247;
insert into mytable (myid, myname) values (9341, 'Bob');
delete from mytable where myid = 4563;
```

This procedure consists of three SQL statements: an update, an insert and a delete. The statements are executed in order.

4.1.1 The DDL restriction

A DDL statement is always executed atomically; it either completely succeeds or completely fails and cannot be rolled back once it has succeeded. For this reason, SQL procedures with DDL are restricted to containing exactly one DDL statement and no other SQL.
4.1.2 Run-time arguments

SQL procedures may be defined with arguments whose values are supplied on each execution of the procedure. This allows a class of general SQL statements to be defined by a single procedure.

Arguments may be substituted for any constant expression in an SQL procedure. An argument is either named or unnamed. A named argument takes the form:

```sql
::<type>name
```

where `type` is any of the following:

- `binary`
- `bool`
- `char`
- `datetime`
- `float`
- `integer`
- `integer64`

for example

```sql
::<integer>maxint
::<float>maxvalue
```

If the `type` is omitted, the argument is assumed to be `char`.

Example:

```sql
select * from tables where name = ::myname
and depth = ::integer>mydepth;
```

An unnamed argument takes the form:

```sql
?
```

The type of an unnamed argument is determined, if possible, from the context in which it is used.

Example:

```sql
select * from tables where name = ? and depth = ?;
```

4.2 Stored SQL procedures (Standard only)

The RTRDB supports the use of stored SQL procedures. These are procedures which are compiled when first executed by each client and may be executed any number of times thereafter. Stored SQL procedures thus provide a means for an application to reuse commonly executed SQL statements and queries.

The procedures are defined as a string of SQL text and can be any valid SQL procedure except DDL.

SQL procedures are stored in the built-in RTRDB table, `sqlprocedure`. The table is defined as follows:

```sql
create table sqlprocedure
(
    persistent,
    name   large varchar primary key,
    text   large varchar not null
)
```
4.2.1 Storing an SQL procedure

SQL procedures are stored in the RTRDB by inserting a new row into the table sqlprocedure. A procedure name must be supplied along with the SQL text. During the insert, the text is compiled against the current schema and the compiled form is stored with the name supplied. The transaction executing the insert statement is aborted in the event of a compilation error.

Any suitable client may be used to insert an SQL procedure. Once inserted, the procedure is available to all other clients.

Example:

```
insert into sqlprocedure values
    ('proc1', 'select * from tables where name = :myname');
```

4.2.2 Executing a stored SQL procedure

Once stored, a procedure may be executed by any client using the SQL call statement. The client simply supplies the procedure name and the values, and optionally the names, of any arguments. Stored procedures that meet the requirements may be used to construct fixed and active queries.

4.2.3 Updating a stored SQL procedure

Once stored, SQL procedures cannot be updated. Attempting to do so is an error, causing the transaction to fail.

4.3 Queries

A query is the only way for a client to retrieve data from a server. A query may be fixed or active. It can be specified in one of following ways:

- as a single SQL select statement
- as an SQL procedure
- as an object reference - An object reference consists of the table queried, the columns to be returned and the value of the columns that are uniquely indexed. This is an efficient way of retrieving data where a single record is required and the value of the columns uniquely identifying the record are known.

It should be noted that two SQL queries evaluated at two different times but where there is the same data in the database may yield records in two different orders. If an application requires a particular order, then it should have the appropriate configuration, such as making sure that an “order by” clause is present in relevant queries.

4.3.1 Data set specification

The set of data returned by a query depends on the specification of the query. SQL queries may return a data set with zero or more records. Object queries return either zero or one record in the data set.

An SQL query may limit the data set size, stipulating a maximum number of records to be returned. An active query limited in this way guarantees to keep the client informed of the maximum number of records, providing sufficient records are found in the data store. If there are more records eligible for inclusion than the maximum specified, then the server will favour those records closer to the start (as specified by the ordering criteria in use) over those further from the start.
4.3.2 Procedures for fixed queries

A procedure for a fixed query is evaluated exactly once and the resulting data is sent back to the client. The procedure may contain any number of DML statements followed by exactly one select statement, for example:

```
insert into timer (id, interval) values (1, 1000000);
select * from timer where id = 1;
```

Note that the select statement must be the last in the procedure. No DDL may be supplied with a fixed query.

In summary, a procedure for a fixed query must

- contain exactly one select statement
- contain no SQL after the select
- contain no DDL

4.3.3 Procedures for active queries

The procedure supplied for an active query may be executed at any time in the lifetime of the query. The procedure is therefore restricted to exactly one select statement. Note that an active query should not return values based on the current time, as the values returned will be seen to change at arbitrary times.

When an active query is first established, the server returns the complete data set to the client. Subsequent data changes in the server cause updates to the data set to be sent. These updates are known as deltas.

Deltas are evaluated at the end of each transaction that changes the query results, though a delta may not be sent for each transaction that affects the query. A transaction that affects a query will schedule that query to be refetched and in a busy system it is quite likely that more than one transaction can occur before the refetch occurs. Hence a delta may contain the changes made by more than one transaction. An option also exists to impose a minimum time interval between deltas irrespective of transaction rate; this is recommended to reduce the load where data is changing rapidly.

Evaluating deltas requires the server to compare data sets before and after the transaction. In order to do this the server saves the 'primary key' of each record in the data set. This primary key is defined as the primary key of all of the tables specified in the from clause of the select, except those of tables joined by their primary key to another's foreign key. In order to allow this comparison to succeed, active queries are restricted to queries where the data set primary key is selected.

In summary, a procedure for an active query must

- contain exactly one select statement
- contain no DDL or DML
- select the complete data set primary key

The processing of certain forms of active query is optimised. There are currently two sub-classes of active query where more efficient processing of changes is performed.

1. A non-aggregate, non-group-by, non-ordered query on a single table or view with no expressions in the output columns. Changes to records and data in this type of active query are more efficiently detected.

2. A query of the above type on a table (not view) where the select restriction precisely defines the columns in a unique index or primary key using "=" and no more. This type of query can return at most one record. While an active query matches one record, changes to different records in the queried table will not cause the active query to expend resources. Hence this
form of active query is more efficient when monitoring a number of records in a table where there are many more records being updated than viewers.

Note that the optimisations described here may change in future versions.

### 4.3.4 Schema changes and active queries

An active query will fail if a change to the database schema is made that would cause the active query to either error or return different columns if it was run again after the schema is changed.

Schema changes that will fail an active query are:

- Dropping or renaming a table that is referenced by the active query.
- Dropping or renaming a column of a table that is referenced by the active query.
- Adding a column to a table or changing a column from hidden to not hidden or not hidden to hidden that is referenced by an active query that retrieves all columns, i.e. select * from …

### 4.4 Transactions

A transaction is the only way for a client to change data in the server. A transaction is always atomic; clients either see the complete set of changes submitted in a transaction, or in the event of failure no change at all.

Clients may submit transactions in two ways: as a set of data changes applied through an active query, or as an SQL procedure. Both methods may be combined into a single transaction in which case the procedure is executed first. We shall consider these methods separately below.

#### 4.4.1 Data modification through SQL procedures (Standard only)

A client may send exactly one SQL procedure to be executed as a single transaction.

#### 4.4.2 Data modification through active queries

A client may use an active query to modify data in the server. The change is always done in an atomic transaction. To send the transaction to the server, the client sends all the differences between the query data set at the start and at the end of the client’s transaction. Changes may be any mixture of

- new records (inserts)
- changes to existing records (updates)
- removal of existing records (deletes)

Once the modifications have been applied, the query, whether ordinary or polled, is re-fetched and its deltas re-evaluated. The new set of deltas is used to handle data modification failure (see section 4.4.3).

A client changing data in the server through an active query does so by specifying changes to the query data set. When the server receives the data set changes, it attempts to interpret them in a sensible way, applying the new values to attributes in the database. This can only be done unambiguously if the data set can be mapped directly onto values in the database.

The server is conservative in doing this, and only allows direct updates on active queries selecting columns directly (i.e. no SQL expressions) from a single database table. If the server detects an ambiguity, the transaction fails.

#### 4.4.3 Data modification failure
For all transactions that fail, an error message is sent back to the client indicating the nature of the error.

If the transaction was started by data modifications on an active query, the server additionally sends a set of deltas with any failure. The deltas are to realign the client's data set with the data in the server. For example, consider a client executing a delete through an active query. The client defines the transaction by deleting a record from its active query data set. Should the transaction fail, the corresponding delete fails to occur in the server. The server therefore sends an insert delta to tell the client to reinstate the deleted record.

It is the server that is the final arbiter of what the correct data is.

### 4.4.4 Conflict detection

Conflict detection prevents two clients altering the same piece of data at the same time. It is carried out by the server whenever two or more clients attempt to modify the same data at the same time through active queries, assuming the clients have requested conflict detection.

Conflict detection works by comparing the sets of transactional changes and applying the following rules:

<table>
<thead>
<tr>
<th>Actions on the same object</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete + update</td>
<td>conflict</td>
</tr>
<tr>
<td>delete + delete</td>
<td>conflict</td>
</tr>
<tr>
<td>update + update</td>
<td>conflict if the same columns have been updated</td>
</tr>
</tbody>
</table>

The rules embody a form of ‘optimistic concurrency’.

Conflict anywhere in a transaction causes the whole transaction to fail.
4.5 Locks

Locks can be obtained by clients to provide improved concurrency control. The data they protect in the RTRDB can be defined to the row and field level. Both optimistic and pessimistic locks are supported and a client can obtain any combination of them.

A request for a lock is either immediately accepted or rejected. It is not possible for a client to request that it waits until the data server can satisfy a pessimistic lock request. Further, if a lock request cannot be satisfied, all locks already held by that client are released. This approach avoids the potential for a deadlock or deadly embrace to occur.

Typically locks once obtained by a client are retained by the RTRDB until that client next submits a transaction, although they may be failed before that by the RTRDB if for instance an optimistic lock detects a change to the data it is protecting or SQL DDL is executed.

Due to the potential for pessimistic locks to prevent other clients from executing transactions, the RTRDB provides configuration resources for disabling their use entirely or associating a time-out with all pessimistic locks. The configuration resource `sql_pessimistic_lock_disable` is used to disable pessimistic locks and `sql_pessimistic_lock_timeout` is used to specify a time-out. The use of the time-out avoids pessimistic locks being held for extended periods of time and degrading system performance. Specifying a time-out is therefore recommended.

4.6 Multi-Threaded Execution

The RTRDB employs operating system threads to stop large queries or file access from blocking the database thus preventing the RTRDB from performing any other activities or requests. The RTRDB allows multiple concurrent queries to run in conjunction with an individual transaction. This is achieved by using:

- A database thread pool
- Client connection threads
- Separate threads for journalling
- A separate thread for CL

The locking strategy operates as follows. Note that the locking describe here is low-level table locking within the RTRDB and is entirely separate from the high-level locking described in section 4.5. A transaction will obtain an exclusive lock for each table it accesses at the point it first accesses that table. A query will obtain a shared lock for each table it accesses and will do this as an atomic operation before it is executed (after the preparation stage). Any transactions containing DDL exclusively lock the entire database. The RTRDB allows a static query to contain a sequence of DML statements before the SELECT statement. This is supported by the table locking mechanism by ensuring that the exclusive table locks obtained by the DML statements are released and the shared locks for the SELECT statement obtained as an atomic operation after the DML statements have been executed and before the SELECT statement is executed.

A pool of threads is used by the RTRDB to process client requests. The number of threads in the pool is configurable. By default it will initially contain one thread and will increase to match the maximum number of concurrent client connections. It is possible to specify the initial number and to limit the maximum number of threads in the pool. Thus, it is possible to specify a fixed size thread pool where all threads are created at start-up and no others are subsequently created. The configuration resources `thread_pool_initial`, `thread_pool_limit` and `thread_pool_size` are used to configure the thread pool. These are documented in section 7 on resources.

The number of threads created for client communications is transport specific. For the OSE transport a single thread (process) is created to handle all client communications for a single data service. For the TCP transport a thread is created for each client connection.
5. Running the RTRDB

This section describes how to run the RTRDB.

5.1 Configuration

Before running, the RTRDB must first be configured to fit the user environment and to run the application code. Configuration is achieved through

- **configuration parameters** (used to describe the environment in which the RTRDB will be executed.)
- **CL and SQL** (used to define objects in the system. Refer to the Control Language Reference and SQL Reference documents.)

Configuration parameters are specified as resources on the command line and/or in a configuration file. For details on the format of resources refer to the Polyhedra User's Guide document. For details on the resources used by the RTRDB refer to section 7.

5.2 Starting the RTRDB

The RTRDB is a server and must be started up before any client. The following command line invokes the RTRDB:

```
    rtrdb { <option> } <module>
```

where `<module>` is the name of the resource module within the configuration file. The following options are supported:

- `-c <configfile>` is the name of the configuration file (see the Polyhedra User's Guide).
- `-r <resource>` is a resource used by the RTRDB (see section 6) and has the format:
  
  "resource" = value 

- `-help` displays help message.
- `-v` print version only.

5.3 RTRDB Initialisation

On start up, the RTRDB initialises in the following order:

- The database is initialised. It is optionally loaded from a .dat file and in the Polyhedra Flash DBMS case creates and uses a poly.fldb file if no file is specified by resource.
- The Historian, if included, and constructs tables in the database if they do not already exist. Further information regarding the Historian facility can be found in the Historic Data Logging Manual.
- The CL manager, if included, initialises, loading and compiling any scripts.
- The server opens the ports for client connections. If the port cannot be opened, the RTRDB terminates immediately.

At this point the RTRDB is running and available for connections from Polyhedra clients.

Setting the **ready_port** resource to a numeric value will send a 4-byte message on the UDP port number, which is specified by that value on the local machine. This indicates that it is able to accept client connections.
5.4 System Clock

The RTRDB (and other Polyhedra components) use timers and other devices which depend on the passage of time, and their correct operation is dependent on the system clock reporting time as increasing in a monotonic fashion. The behaviour of Polyhedra is undefined if the system clock is changed while any component is running.
6. Limitations

This section describes the restrictions on the RTRDB. If an attempt is made to exceed any of the limits described then an appropriate error will be generated (see Section 21).

6.1 Records

A maximum of $2^{32}$ records per table are allowed. Note that the number of records in a table includes records in derived tables.

6.2 Tables

A maximum of 8192 levels of inheritance are permitted in each hierarchy, and a maximum of $2^{32}$ tables may be derived from a single table. There is no restriction on the number of inheritance hierarchies.

6.3 Columns

A maximum of 65536 columns (whether shared, virtual or not) may be defined in one table. Derived tables may also define 65536 columns.

6.4 Connections

On 64-bit platforms, a maximum of 65534 simultaneously-open statements (queries or transactions) are allowed for each client connection. On 32-bit platforms, there is no arbitrary limit but, for maximum portability, it is recommended that applications do not exceed this limit of 65534 statements.

6.5 Polyhedra Flash DBMS File Size (Flash DBMS only)

The Polyhedra Flash DBMS database file may be up to 2GB in size. It may be necessary to increase the page size used in order to reach this size.
7. Resources

This section describes the resources specific to the RTRDB. The RTRDB also uses general resources for Polyhedra, CL, and the Historian. Refer to the Polyhedra User's Guide, CL Reference, and Historic Data Logging manuals for details of these. Where marked (Standard only) the resource is applicable to the standard Polyhedra product. Where marked (Flash DBMS only), the resource is only applicable to the Polyhedra Flash DBMS product. Otherwise, the resource is applicable to both products.

It should be noted that both RTRDBs in a fault tolerant configuration should be configured the same. Resources should therefore be given the same values. The only exceptions to this are resources which provide the names of services or directories.

Note that the data_service resource must always be supplied.

**arbitrator_protocol**

Specifies the protocol to be used between the RTRDB and arbitrator.

If arbitrator_protocol is supplied, then so must arbitrator_service.

- **poly**
  The default standard Polyhedra client-server protocol. In this case the arbitrator will be implemented as a Polyhedra database.

- **ose**
  A simple OSE signal based protocol that does not require the arbitrator to be a Polyhedra database or require a TCP/IP stack to be available. This is only supported on OSE.

- **tcp**
  A simple TCP/IP based protocol that does not require the arbitrator to be a Polyhedra database.

- **linx**
  A simple LINX based protocol that does not require the arbitrator to be a Polyhedra database.

**arbitrator_service**

Specifies the service by which the RTRDB connects to the arbitrator.

If arbitrator_service is supplied, then so must arbitrator_protocol.

- **poly**
  arbitrator_service contains either a standard Polyhedra data service name or a comma-separated list of data service names. This is the default; this protocol will be used if no value is supplied for the arbitrator_protocol resource.

- **ose**
  arbitrator_service contains the name of the arbitrator process.

- **tcp**
  arbitrator_service contains a TCP/IP port number.

- **linx**
  arbitrator_service contains a LINX endpoint name.

This resource is only required if the RTRDB is running as part of a fault tolerant configuration.
**cl_memory_limit**

The maximum amount of memory in bytes that may be allocated by the CL sub-system. If this limit is exceeded then the RTRDB will attempt to recover by resetting the CL sub-system. This memory will be included in the **memory_limit** resource if that is set. See the section on Memory Exhaustion Recovery for more details. A zero value indicates the default behaviour of no CL memory limit.

**char_character_set**

Specifies the character set used by the SQL CHAR and VARCHAR and CL STRING data types. The allowable values are ASCII and UTF-8. The default value is ASCII.

**dataconnection_class**

A string resource which conveys the name of an alternative table to be used for storing client connection details (see dataconnection class in the Utility Classes manual).

**dataport_class**

A string resource which conveys the name of an alternative table to be used for storing data service listening port details (see dataport class in the Utility Classes manual).

**dataquerymonitor_class**

A string resource which conveys the name of an alternative table to be used for defining criteria for monitoring active queries (see the dataquerymonitor class in the Utility Classes manual).

**dataquery_class**

A string resource which conveys the name of an alternative table to be used to show current active queries (see the dataquery class in the Utility Classes manual).

**default_table_max**

*(Flash DBMS only)* The default maximum number of pages that may be allocated for a table. A value of 0 specifies that the number of pages is unlimited. The default value is 0.

The value may be overridden using the WITH MAX table constraint.

**default_table_min**

*(Flash DBMS only)* The default number of pages allocated for a table when it is created. The default value is 10.

The value may be overridden using the WITH MIN table constraint.

**enable_journalling**

*(Standard only)* A boolean resource which if set activates the journalling capability of the RTRDB. The default value is **false** unless **journal_service** is specified in which case the default is **true**. See section 11.

**enable_parallel_allocator**

A Boolean resource that when set (the default is true) activates the parallel memory allocation scheme. This scheme improves performance by efficiently providing memory to client sessions within the RTRDB.

Setting this resource to false may reduce memory usage at the expense of performance.

The parallel memory allocation scheme requires the allocator library (liballoc.a on Linux) to be linked into the RTRDB. The default RTRDB executable includes this library.

**error_text_detail**

Specifies the levels of detail in the error text supplied with a transaction or query failure. The default value is 1.
**flash_cache_size**  
*(Flash DBMS only)* The amount of RAM reserved for the database cache and other necessary runtime information in bytes. The default value is 20480. The RTRDB will not start-up if the cache size specified is too small.

**flash_convert_file**  
*(Flash DBMS only)* This resource optionally specifies a standard Polyhedra load file. The Flash DBMS database file (as specified normally with `load_file`) is initialised with the contents of this file. There are a number of unsupported configurations that will cause the RTRDB to terminate:

- The RTRDB is in FT mode.
- The load file contains any functionality not supported by Flash DBMS (e.g. domains, arrays).
- The load file contains journalled records (i.e. the load file must be created by a “SAVE INTO”).
- The load file contains CL and CL is enabled in the RTRDB (if CL is disabled, the CL in the load file is ignored).

This conversion functionality requires the optional Fault Tolerance module. Please see the linking instructions for the relevant platform for how to link in this module.

**flash_file_extent**  
*(Flash DBMS only)* The number of pages by which the size of the database file is increased when it is required to grow. A value of 0 specifies that the file cannot grow in size. The default value is 100.

The resource is used when a database file is created or an existing database file is loaded.

**flash_file_max**  
*(Flash DBMS only)* The maximum size of the database file in pages. A value of 0 specifies that the size of the file is unlimited. The default value is 0.

**flash_file_size**  
*(Flash DBMS only)* The size of the database file in pages. The default value is 100.

The resource is used when a database file is created and when the `flash_shrink_file` resource is set.

This resource is ignored when an existing database file is used.

**flash_page_size**  
*(Flash DBMS only)* The page size of the created database file in bytes. Acceptable values are 512, 1024, 2048, 4096 and 8192. The default value is 1024.

The RTRDB will not start-up if an invalid page size is specified. For optimal performance, the page size must be a multiple of the block size used by the Flash memory.

This resource is ignored when an existing database file is used.
**flash_shrink_file**  
*(Flash DBMS only)* A Boolean resource specifying whether to shrink the database file. A value of “true” specifies that an attempt is made to reduce the size of the database file when it is loaded. The default value is ‘false’. The resource can be used in conjunction with the **flash_file_size** resource to shrink a database file to a given size. Note that this functionality depends on a file system's ability to truncate files and may not work correctly on all file systems (for example some OSE file systems).

**flash_swap_file**  
*(Flash DBMS only)* A string resource that defines a file to receive database changes in swap file mode. The setting of this resource enables swap file mode. The value is affected by the **load_file_directory** resource. When set the load file is not written to, changes are written to the swap file in a non-durable fashion and the number of writes to the file are minimised.

**journal_connect_count**  
Specifies the number of times an attempt is made by a replica RTRDB to connect to the journal service of a master or replica RTRDB. A value of 0 specifies an unlimited number of attempts. The default value is 10. Once a replica RTRDB has established its first connection, it will make an unlimited number of attempts to connect and reconnect to a standby or second replica RTRDB, irrespective of the setting of this resource.

**journal_connect_interval**  
Specify in milliseconds the interval between connection attempts to a master or standby RTRDB made by a replica RTRDB. The default is 1000.

**journalconnection_class**  
A string resource which conveys the name of an alternative table to be used for storing journal connection details (see journalconnection class in the Utility Classes manual).

**journalport_class**  
A string resource which conveys the name of an alternative table to be used for storing journal service listening port details (see journalport class in the Utility Classes manual)

**journal_connect_timeout**  
Specifies in milliseconds the timeout for each connection attempt made by a standby or replica RTRDB. The default is 10000.

**journal_failover_timeout**  
Specifies in seconds the timeout that a replica RTRDB uses when waiting for a standby RTRDB to be promoted to master. The default value is 60. A replica RTRDB terminates if it loses connection to the replication service whilst obtaining its initial snapshot.

**journal_heartbeat_check_interval**  
Specifies in milliseconds the interval a journal server uses between checking its journal connections for heartbeat failure. A value of 0 disables checking. The default value is 1000. A journal server is aware of the heartbeat settings used by a journal client and if no heartbeat or other message is received within twice the specified interval, the server assumes the connection has failed and disconnects the client.
### journal_heartbeat_interval
Specifies in milliseconds the interval between journal heartbeat messages. When a journal client is connected, it sends heartbeat messages to the server at this interval. A value of 0 disables heartbeats. The default value is 1000.

### journal_heartbeat_timeout
Specifies in milliseconds a period within which a journal client expects a server to reply to a heartbeat message. If no reply or other message is received within this period, the client assumes the connection has failed and disconnects from the server. A value of 0 disables heartbeats. The default value is 5000.

### journal_load_timeout
Specifies a timeout in milliseconds. A standby or replica database will wait for twice this period for new data before terminating when loading the initial snapshot from the master RTRDB. It will also wait for twice this period for the connection to master to be broken before terminating. The default value is 60000.

### journal_queue_size
Specifies the number of transaction data buffers (each of size 4K bytes) that can be queued for journaling. If the queue fills up (due to not being emptied quickly enough), the next transaction will be blocked until there is room in the queue. The default value is 1000.

When increasing the queue size you must be aware of the extra memory requirements. Each slot in the queue takes up 8 bytes on 32-bit platforms and 16 bytes on 64-bit platforms. An extra 4K bytes is used for each occupied slot.

### journal_replication_depth
Specifies the maximum replication depth permitted for a particular configuration. The resource is only read by a standalone or master RTRDB and is communicated to and acted upon in any connected replica RTRDB. The default value is 2.

A replica RTRDB will error and terminate if it reports a replication depth equal to the maximum replication depth and it is providing a journal service.

### journal_service
Specifies the names of the data services on which the RTRDB listens for journal connections as a comma separated list. Note that setting **journal_service** also enables the journaling capabilities of the RTRDB (see **enable_journalling**).

### journal_session_queue_size
Specifies the number of transaction data buffers (each of size 4K bytes) that can be queued for sending to each standby and replica database. By default if the queue fills up (due to not being emptied quickly enough), the connection to the standby or replica will be disconnected. The default value is 1000.

A value of 0 specifies that the queue blocks until the data has been sent.

When increasing the queue size you must be aware of the extra memory requirements. Each slot in the queue takes up 8 bytes on 32-bit platforms and 16 bytes on 64-bit platforms. An extra 4K bytes is used for each occupied slot.

### journal_transaction_timeout
Specifies the timeout in milliseconds that a master RTRDB waits for transactions to be acknowledged by a standby RTRDB. If this timeout is exceeded, the master RTRDB closes the connection to the standby RTRDB and outputs a warning message. The default value is 60000. See section 13.5.2.1.
**load_file**

This resource specifies a `.dat` file to load on start up. In the standard product, this file is overwritten whenever the content of the database is saved to disk.

In the Flash DBMS product, if the file does not exist when the **RTRDB** starts-up, it is created. If it already exists, and is a valid database file, it is used. If no value is supplied for this resource, a default name of ‘poly.fdb’ is used, but any existing file named ‘poly.fdb’ is ignored and overwritten. This behaviour emulates that of the standard product as much as possible.

**load_file_directory**

Specifies the directory containing the journal files (and the Flash DBMS swap file if applicable). Defaults to the current directory.

**memory_limit**

The memory usage limit (in bytes) above which the RTRDB will attempt to reduce memory usage as defined in the section on Memory Exhaustion Recovery. A zero value indicates the default behaviour of no memory limit.

**msg_queue_size**

An integer resource specifying the number of entries allocated in each internal message queue used by the RTRDB. Changing this value will affect the memory allocated by the RTRDB for client sessions. Note that setting the value too low may degrade the performance of the RTRDB. The default value is 1000. It is an error to set **msg_queue_size** to less than 2.

**other_journal_service**

Specifies the name of the journal service on which the other RTRDB listens for connections. This resource is only required when configuring a fault tolerant system. Note that **other_journal_service** is ignored when using the OSE signal based arbitrator protocol. Instead, the journal service of the master database is supplied in the message from the arbitrator.

**password_security_level**

Specifies the method by which passwords are protected. If security is not enabled then the resource has no discernible effect. The default value is 0. This is described more fully in section 9.9

**query_delta_pk**

A Boolean resource that changes the way active queries are handled by the database. Normally, when updates are reported to a client, the server only sends the changed attributes. When this resource is set to **true**, the primary key attributes are also transmitted, regardless of whether they are changed. This affects the behaviour observed by all clients, and so should be used with caution, and only when use is being made of the callback API. The default is **false**.

**query_delta_all**

A Boolean resource that changes the way active queries are handled by the database. Normally, when updates are reported to a client, the server only sends the changed attributes. When this resource is set to **true**, all the attributes are transmitted, regardless of whether they are changed. This affects the behaviour observed by all clients, and so should be used with caution, and only when use is being made of the callback API. The default is **false**.

**query_disable_integer64**

A Boolean resource provided for backwards compatibility. If set to **true**, all queries return integer expressions as 32-bit. The default is **false**.

**ready_port**

Specifies a UDP port to which a single four-byte message (“POLY”) is sent when the database is ready to accept client connections. This resource is optional.
replica_listen_mode **(Standard only)** A Boolean resource (with default false) indicating whether an RTRDB running in replica mode initiates or listens for connections to obtain its journal stream.

replication_service **(Standard only)** If replica_listen_mode is false, specifies the name of the journal service(s) to which an RTRDB running in replica mode connects to obtain its journal stream.

If replica_listen_mode is true, specifies a single service on which an RTRDB running in replica mode listens for connections from which it obtains its journal stream.

Use of the resource implies the RTRDB is running in replica mode.

replica_queue_size Specifies the number of transaction data buffers (each of size 4K bytes) that can be queued in the replica. If the queue fills up (due to it not being emptied quick enough), data from the server will be blocked until there is room in the queue, which can cause data to queue up at the server and eventually the replica connection to be closed. The default value is 1000. Values less than 1000 are ignored.

When increasing the queue size you must be aware of the extra memory requirements. Each slot in the queue takes up 4 bytes which when occupied also uses an extra 4K per slot.

safe_commit A Boolean resource (with default false) indicating whether transactions performed through the internal connection or using a dataservice object should be submitted in safe-commit mode.

session_memory_limit The number of bytes that an individual client session may use before the RTRDB takes steps to reduce the memory consumption of the session – as defined in the section on Memory Exhaustion Recovery. The memory accounted for by a client session is included in the total limited by the memory_limit resource. A zero value indicates the default behaviour of no session memory limit.

snapshot_on_shutdown A Boolean resource which if true, and journalling is enabled, forces the RTRDB to write a snapshot on shutdown. If journalling is disabled, the resource has no effect. The default is false.

sql_default_transient_tables A Boolean resource which if true sets the default table column persistence to transient rather than persistent. The default is false.

sql_disable_v8_3_reserved_words A Boolean resource which if true disables the SQL reserved words LIMIT and OFFSET that were introduced in Polyhedra 8.3. Note that the functionality associated with these reserved words is also disabled.

This resource is provided as a migration aid and may be withdrawn in later versions of Polyhedra.

sql_disable_v8_7_reserved_words A Boolean resource which if true disables the SQL reserved words CONDITION, LOCK, OPTIMISTIC, PESSIMISTIC and WITHOUT that were introduced in Polyhedra 8.7. Note that the functionality associated with these reserved words is also disabled.

This resource is provided as a migration aid and may be withdrawn in later versions of Polyhedra.
### Resources

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sql_pessimistic_lock_disable</code></td>
<td>A Boolean resource which if <strong>true</strong> disables the use of pessimistic locks. If the use of pessimistic locks is disabled, the RTRDB rejects all pessimistic lock requests. The default is <strong>false</strong>.</td>
</tr>
<tr>
<td><code>sql_pessimistic_lock_timeout</code></td>
<td>An integer resource specifying, in milliseconds, how long the RTRDB will retain pessimistic locks before effectively converting them into optimistic locks. The timeout is measured from the first pessimistic lock obtained when a session has no existing pessimistic locks. If pessimistic locks are in place or obtained beyond the timeout, they operate as optimistic locks and fail if a locking conflict occurs. A value of 0 disables the use of any timeout and specifies that pessimistic locks can be held indefinitely. The default value is 0.</td>
</tr>
<tr>
<td><code>suppress_cl</code></td>
<td>A Boolean resource which if <strong>true</strong> prevents CL source being loaded and executed. The default is <strong>false</strong>.</td>
</tr>
<tr>
<td><code>suppress_log</code></td>
<td>(Standard only) A boolean resource which if <strong>true</strong> disables the Historian subsystem. The default value is <strong>false</strong>. See section 17.</td>
</tr>
<tr>
<td><code>suppress_standby_startup_snapshot</code></td>
<td>(Standard only) A boolean resource which if <strong>true</strong> prevents a standby RTRDB from writing a snapshot when it connects to the master RTRDB. The resource only has any effect if journalling is disabled. The default value is <strong>false</strong>.</td>
</tr>
<tr>
<td><code>tcp_keepalive</code></td>
<td>A Boolean resource which if <strong>true</strong> enables the setting of the SO_KEEPALIVE socket option on client TCP socket connections accepted by the RTRDB. The default value is <strong>false</strong>.</td>
</tr>
<tr>
<td><code>tcp_message_cache_size</code></td>
<td>An integer resource specifying the maximum number of message buffers held in a cache ready for use on each TCP (or SSL) connection. The default value is 10. When a connection requires a message buffer it either retrieves one from the cache or allocates a new buffer when none are available. Initially the cache starts off with no buffers. Once a buffer is finished with it is placed in the cache ready to be re-used unless the cache already contains the maximum number in which case the buffer is released. If <code>tcp_message_cache_size</code> is set to zero then each connection will only allocate message buffers as and when it needs them.</td>
</tr>
<tr>
<td><code>thread_pool_initial</code></td>
<td>An integer resource specifying the initial number of database threads. The default value is 0. It is an error to set <code>thread_pool_initial</code> to greater than <code>thread_pool_limit</code>.</td>
</tr>
<tr>
<td><code>thread_pool_limit</code></td>
<td>An integer resource specifying the maximum number of database threads. The default value is 1024. It is an error to set <code>thread_pool_limit</code> to less than <code>thread_pool_initial</code>.</td>
</tr>
<tr>
<td><code>thread_pool_size</code></td>
<td>An integer resource specifying the number of database threads. This specifies a fixed size thread pool and is an alternative to setting both <code>thread_pool_initial</code> and <code>thread_pool_limit</code> to the same value. It is an error to set <code>thread_pool_size</code> and either <code>thread_pool_initial</code> or <code>thread_pool_limit</code>.</td>
</tr>
<tr>
<td><code>type</code></td>
<td>This is used to specify the type of Polyhedra component to be executed. It must be set to <code>rtrdb</code>. This resource is compulsory.</td>
</tr>
</tbody>
</table>
8. Schema tables

In common with most RDBMS, the RTRDB uses special tables to hold information on the structure of the database. These tables are collectively known as schema tables. The schema tables comprise the following:

- tables
- views
- attributes
- indexes
- indexattrs

A client may investigate the database structure using standard SQL to query the schema tables. However, the schema tables may not be directly modified.

8.1 The Tables table

This table has a record for each table, domain, view or built-in datatype in the database.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char</td>
<td>Name of table/domain/datatype</td>
</tr>
<tr>
<td>depth</td>
<td>integer</td>
<td>Depth in inheritance hierarchy</td>
</tr>
<tr>
<td>type</td>
<td>integer</td>
<td>Type of table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0 = table, 1 = built-in type, 2 = domain, 3 = view)</td>
</tr>
<tr>
<td>derived_from</td>
<td>char</td>
<td>Name of parent table/domain if this is derived</td>
</tr>
<tr>
<td>islocal</td>
<td>bool</td>
<td>Is local data</td>
</tr>
<tr>
<td>pk_index</td>
<td>char</td>
<td>The name of the index defining the primary key</td>
</tr>
<tr>
<td>persistence</td>
<td>bool</td>
<td>Default persistence for all attributes in table</td>
</tr>
<tr>
<td>system</td>
<td>bool</td>
<td>Whether the table is a system schema table</td>
</tr>
<tr>
<td>owner</td>
<td>char</td>
<td>The name of the user who created the table</td>
</tr>
</tbody>
</table>
8.2 The Views table

This table contains a record for each view defined in the database. It is derived from the `tables` table and has the following additional column:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>char</td>
<td>Text of the views select expression</td>
</tr>
</tbody>
</table>

8.3 The Attributes table

This table contains a record for each attribute in a table, domain or view.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>char</td>
<td>Table where the attribute is found</td>
</tr>
<tr>
<td>name</td>
<td>char</td>
<td>Name of attribute</td>
</tr>
<tr>
<td>type</td>
<td>char</td>
<td>Type of attribute</td>
</tr>
<tr>
<td>length</td>
<td>integer</td>
<td>Length in bytes if attribute is limited in length</td>
</tr>
<tr>
<td>isexternal</td>
<td>bool</td>
<td>Is stored external to database object</td>
</tr>
<tr>
<td>not_null</td>
<td>bool</td>
<td>Does attribute allow null values</td>
</tr>
<tr>
<td>persistence</td>
<td>bool</td>
<td>Is persistent</td>
</tr>
<tr>
<td>isarray</td>
<td>bool</td>
<td>Is an array of object references</td>
</tr>
<tr>
<td>ishidden</td>
<td>bool</td>
<td>Is hidden from default selection (select *)</td>
</tr>
<tr>
<td>isprimary</td>
<td>bool</td>
<td>Is part of the table's primary key</td>
</tr>
<tr>
<td>default_value</td>
<td>char</td>
<td>Default value for attribute</td>
</tr>
<tr>
<td>islocal</td>
<td>bool</td>
<td>Is local data</td>
</tr>
<tr>
<td>issshared</td>
<td>bool</td>
<td>Is shared attribute</td>
</tr>
<tr>
<td>isvirtual</td>
<td>bool</td>
<td>Is virtual attribute</td>
</tr>
</tbody>
</table>
## 8.4 The Indexes table

This table contains a record for each index in the database. The table contains user-defined indexes and also indexes on primary keys (prefixed by PK-) or foreign keys (prefixed by FK-).

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char</td>
<td>Name of index</td>
</tr>
<tr>
<td>table_name</td>
<td>char</td>
<td>Table index is defined on</td>
</tr>
<tr>
<td>type</td>
<td>integer</td>
<td>Foreign key(0), Unique hash(1) or View Index (2)</td>
</tr>
<tr>
<td>constraint_type</td>
<td>integer</td>
<td>Unique (0), Primary Key (1), Foreign Key (2) and Non-Unique index (3)</td>
</tr>
<tr>
<td>is_unique</td>
<td>bool</td>
<td>Does index enforce uniqueness</td>
</tr>
<tr>
<td>foreign_table</td>
<td>char</td>
<td>Name of foreign table if index is foreign key</td>
</tr>
<tr>
<td>update_rule</td>
<td>integer</td>
<td>Referential action applied to foreign key when target is updated, if index is foreign key: Cascade (0), Restrict (1) and Set Null (2)</td>
</tr>
<tr>
<td>delete_rule</td>
<td>integer</td>
<td>Referential action applied to foreign key when target is deleted, if index is foreign key: Cascade (0), Restrict (1) and Set Null (2)</td>
</tr>
</tbody>
</table>

## 8.5 The IndexAttrs table

This table contains a record for each attribute included in an index.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_name</td>
<td>char</td>
<td>Name of index</td>
</tr>
<tr>
<td>index_order</td>
<td>integer</td>
<td>Order of attribute in index (start at 0)</td>
</tr>
<tr>
<td>attr_table</td>
<td>char</td>
<td>Table of attribute</td>
</tr>
<tr>
<td>attr_name</td>
<td>char</td>
<td>Name of attribute</td>
</tr>
<tr>
<td>expression</td>
<td>char</td>
<td>Optional expression for index column. e.g. “lower(col)”</td>
</tr>
</tbody>
</table>
8.6 The schema definition

The SQL definition of the schema tables is defined below:

```sql
create schema
create table tables
(
    name large varchar not null primary key,
    depth integer,
    type integer,
    derived_from large varchar references tables,
    pk_index large varchar references indexes,
    persistence bool,
    system bool,
    owner large varchar not null,
    islocal bool
)
create table views
(
    derived from tables,
    value large varchar not null
)
create table attributes
(
    primary key (table_name, name),
    table_name large varchar not null references tables,
    name large varchar not null,
    type large varchar references tables,
    length integer,
    isexternal bool,
    not_null bool,
    persistence bool,
    isarray bool,
    ishidden bool,
    default_value large varchar,
    islocal bool,
    isshared bool,
    isvirtual bool
)"
```
create table `indexes`
(
    `name` large varchar not null primary key,
    `table_name` large varchar references `tables`,
    `type` integer,
    `constraint_type` integer,
    `is_unique` bool,
    `foreign_table` large varchar references `tables`,
    `update_rule` integer,
    `delete_rule` integer
)
create table `indexattrs`
(
    primary key(`index_name`, `index_order`),
    `index_name` large varchar references `indexes`,
    `index_order` integer,
    `attr_table` large varchar,
    `attr_name` large varchar,
    `expression` large varchar,
    foreign key `(attr_table, attr_name)`
    references `attributes`(table_name, name)
);
9. Security

The RTRDB supports SQL security features. It supports the concepts of a user, user privileges and table ownership, and can be configured to enable password security. The statements GRANT and REVOKE are supported for managing privileges.

Each client connection establishes a session with the database as a user. In order to perform a particular operation on a table, domain or view the user must possess the relevant privileges.

Each table, domain or view is owned by the user who created it. The owner of a table, domain or view possesses all privileges on it. Privileges may be granted from one user to another with the GRANT statement and revoked with the REVOKE statement.

9.1 Enabling security

The RTRDB can be configured to enable user security and so control access to the database and the data held in it. User authentication by password can be enabled.

Security in the RTRDB is enabled by including a definition for the users table in the .dat file loaded on start up. The table definition must follow that in section 9.3.

9.2 Connections and sessions

Before a client can query or modify the RTRDB it must first establish a session with the database. If security is enabled, sessions can only be established if the client authenticates the connection as a particular user.

9.3 Users

The concept of users is supported through the users table. The presence of this table in the database schema when the RTRDB initialises causes security to be enabled.

It is up to the configurer to supply the table definition. The table definition must be based on the following:

```sql
create table users
(
    persistent,
    name large varchar primary key,
    password large varchar,
    login_name large varchar unique not null
);
```

The name and password columns must be present and their definitions must match those given above. name must be the only primary key column. The login_name column is optional, but if present its definition must match that given above.

The users table may be defined with additional columns or they may be added to it while the RTRDB is running using the SQL statement ALTER TABLE.

The RTRDB uses the users table to control access the database. Inserting a record registers a new user. The name column contains the user's name. The login_name column, if present, contains the login name to be used when establishing a session and the password column the password to be used.
If the login_name column is not present, the name column is used as the login name. The name and login_name should not contain spaces and should be valid identifiers i.e., something that would be acceptable as the name of a table, although this is not enforced. A null password means that no password authentication is carried out.

Two special users are always registered on the system, SYSTEM, and PUBLIC. The RTRDB automatically inserts their records into the users table at start-up if none already exist. These records cannot be deleted. The password field may be modified; this allows the passwords to be changed from their initial value of NULL.

It is possible to establish a session as either SYSTEM or PUBLIC. The significance of the SYSTEM and PUBLIC users is explained in sections 9.4 and 9.5.

All login names are case-insensitive when using the name column, but case-sensitive when using the login_name column. If security is enabled, values in the name column are converted to lowercase when stored in the users table, except for SYSTEM and PUBLIC, which are kept in upper case. Values in the login_name are stored unmodified and may be updated.

Once a session is established as a particular user, all actions performed during the session are done as that user.

### 9.4 Ownership

Each table, domain or view in the database is owned by a user. The name of the owner of each table, domain or view is stored in the owner column of the tables table.

The user executing a create table statement is the owner of that table. System tables and other tables created internally by the RTRDB are owned by the SYSTEM user. The user executing a create domain statement is the owner of that domain. The user executing a create view statement is the owner of that view.

A user cannot be deleted from the system if it owns any tables, domains or views; any tables, domains or views it owns must first be dropped.

### 9.5 Privileges

If security is enabled, a user can only perform an operation on a table, domain or view if they have the necessary privileges on it. Privileges are SQL’s mechanism for providing data security. The operations protected by privileges include: select, insert, update and delete.

The owner of a table, domain or view has all privileges on it. That is, the user who created a table, domain or view can perform any operation on it.

A user may grant any privilege it holds to other users.

All privileges held by the special PUBLIC user are implicitly granted to all other users.

Privileges previously granted by one user to another can be later revoked by the original user.

The two SQL statements GRANT and REVOKE are used to grant and revoke privileges. These statements are described in detail in the statements section of the SQL manual.

Explicitly granted privileges are recorded by the RTRDB in two tables. These are table_privileges and column_privileges.
The `table_privileges` table stores details of privileges explicitly granted on tables, domains and views. It has the following definition:

```sql
CREATE TABLE table_privileges
(
    persistent,
    table_name large varchar,
    grantor large varchar references users,
    grantee large varchar references users,
    alter_priv bool default false not null,
    delete_priv bool default false not null,
    execute_priv bool default false not null,
    index_priv bool default false not null,
    insert_priv bool default false not null,
    lock_priv bool default false not null,
    references_priv bool default false not null,
    select_priv bool default false not null,
    update_priv bool default false not null,
    alter_grantable bool default false not null,
    delete_grantable bool default false not null,
    execute_grantable bool default false not null,
    index_grantable bool default false not null,
    insert_grantable bool default false not null,
    lock_grantable bool default false not null,
    references_grantable bool default false not null,
    select_grantable bool default false not null,
    update_grantable bool default false not null,

    PRIMARY KEY (table_name, grantor, grantee)
) ;
```
The `column_privileges` table stores details of privileges explicitly granted a column. It has the following definition:

```sql
create table column_privileges
(
  persistent,
  table_name  large varchar,
  column_name large varchar,
  grantor     large varchar references users,
  grantee     large varchar references users,
  insert_priv bool default false not null,
  lock_priv   bool default false not null,
  update_priv bool default false not null,
  insert_grantable bool default false not null,
  lock_grantable bool default false not null,
  update_grantable bool default false not null,
  primary key (table_name, column_name, grantor, grantee)
);
```

Both tables are persistent and so granted privileges are maintained across re-start of the RTRDB.

### 9.6 Configuration

By default, security is disabled within the RTRDB. See section 9.1 for information on enabling security.

Client sessions that do not explicitly log in are assigned the default user. By default, this is a notional null user that has no privileges. It cannot access any tables, create, alter or drop any tables, domains and views, or shutdown the database. It is possible to override this and specify any registered user to be the default user. The configuration resource `default_user` is provided for this purpose. If supplied with a valid user name, that user will be assigned to client sessions that have not logged-in.

### 9.7 Configuration for ODBC

Enabling security in the RTRDB has implications for using the Polyhedra ODBC driver. Some ODBC API functions require access to the RTRDB schema. For this reason it is advisable that the RTRDB is configured so that all users connecting to it via ODBC have `select` privilege on the five schema tables, `tables`, `views`, `attributes`, `indexes` and `indexattrs`. A user connecting to the RTRDB via ODBC from MS Access must have `select` privilege on the schema. In addition, they must have at least `select` privilege on each table to which they want to attach.

### 9.8 Object queries

The RTRDB enforces security privileges for object queries as well as SQL queries. To perform an object query on a table or view a user must hold `select` privilege on that table or view. To perform an active object query a user must also hold insert, update and delete privileges on the table or view. These checks are made when the query is prepared. In the case of an active object query, if changes are made to privileges the user holds that directly relate to the query, such as revoking `select` or `update` privilege on the table or view, the query remains in operation and is not failed.
9.9 Password security

When a client logs into a server, the user and password information is transmitted over the local network. If the password is sent as clear-text then the security system is vulnerable to attack. Polyhedra supports the ability for clients to avoid sending the password as clear-text. The security measures are performed by the Polyhedra client libraries (or driver) in response to information sent by the server, and are transparent to the end-user.

9.9.1 Configuration

The level of password protection is controlled with the password_security_level resource, which supports the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This is the default value. Clients send passwords as clear-text</td>
</tr>
<tr>
<td>1</td>
<td>Clients encrypt passwords sent with login messages</td>
</tr>
<tr>
<td>2</td>
<td>Challenge-response with passwords stored as salted hashes</td>
</tr>
<tr>
<td>3 and above</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>

9.9.2 Level 1 protection

This level of password security encrypts the password before sending it to the server, which compares it with the encrypted value stored in the users table. This avoids an attack whereby the password is observed as it is transmitted over the network as clear-text, and, since the server does not handle the unencrypted password, also protects against an attacker obtaining the database load file. It does not protect against an attack whereby the authentication message is captured and replayed, nor does it protect against an attack whereby a clone of the Polyhedra protocol stack is constructed to mimic client messages.

The encryption algorithm used is the Secure Hash Algorithm 1 (SHA-1), as defined by US Federal Information Processing Standards (FIPS) Publication 180-1. The algorithm produces a 160 bit digest from an input of any length, and does not provide a way of decrypting the digest. Its strength and security are well documented elsewhere. Although implementations of this algorithm are, in some countries, subject to export controls, this restriction does not usually apply where it is used for solely password protection.

9.9.3 Level 2 protection

This level of password security uses a challenge-response protocol between the client and server. The client does not transmit either the password or the encrypted value stored in the users table as clear-text. The challenge-response protocol ensures that every challenge-response exchange is unique and therefore protects against a replay attack.

The challenge-response mechanism is based on the main algorithm used by the Salted Challenge Response Authentication Mechanism (SCRAM) defined by RFC 5802. The hash function used is SHA-1. A new challenge is generated for each authentication request. A new salt is generated for each password and the salted hash of the password and the salt itself are both stored in the users table.

All challenge and salt values are 160 bits of random data. This random data should ideally be generated using a Secure Pseudo-Random Number Generator (CSPRNG). A CSPRNG is typically platform specific so a user-supplied Embedded API function poly_random_data is provided for implementing a suitable CSPRNG. Implementations of poly_random_data are provided for POSIX and Win32 platforms. On other platforms the random data is generated using the POSIX rand function and a timestamp as input to the SHA-1 hash function.

9.9.4 Setup
In order to setup a secure system, a client must update the users table with the encrypted passwords. The callback and ODBC APIs provide encryption functions whose operation is dependent on the configuration of the server to which the client is connected. In order to set the values in the server, a client must therefore connect to a server which is configured with the `password_security_level` resource set to a non-zero value, login as a user which has a NULL password and which has update privileges on the users table, and then update the password attributes with the result of an API encryption routine.

This procedure is common to both new systems and upgrades of existing systems.

### 9.10 Summary of security tables

#### users persistent

<table>
<thead>
<tr>
<th>Datum</th>
<th>Type</th>
<th>Constraint</th>
<th>User Access</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>name of user</td>
</tr>
<tr>
<td>password</td>
<td>large varchar</td>
<td>read write</td>
<td></td>
<td>password for user</td>
</tr>
<tr>
<td>login_name</td>
<td>large varchar</td>
<td>unique not null</td>
<td>read write</td>
<td>login name for use</td>
</tr>
</tbody>
</table>

#### column_privilege persistent

<table>
<thead>
<tr>
<th>Datum</th>
<th>Type</th>
<th>Constraint</th>
<th>User Access</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>name of table</td>
</tr>
<tr>
<td>column_name</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>name of column</td>
</tr>
<tr>
<td>grantor</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>user granting privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>user receiving privilege</td>
</tr>
<tr>
<td>insert_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>insert privilege</td>
</tr>
<tr>
<td>lock_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>lock privilege</td>
</tr>
<tr>
<td>update_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>update privilege</td>
</tr>
<tr>
<td>insert_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>insert privilege has grant option</td>
</tr>
<tr>
<td>lock_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>lock privilege has grant option</td>
</tr>
<tr>
<td>update_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>update privilege has grant option</td>
</tr>
</tbody>
</table>
### table_privileges persistent

<table>
<thead>
<tr>
<th>Datum</th>
<th>Type</th>
<th>Constraint</th>
<th>User Access</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>large varchar</td>
<td>primary key</td>
<td>read only</td>
<td>name of table</td>
</tr>
<tr>
<td>grantor</td>
<td>large varchar</td>
<td>foreign key</td>
<td>read only</td>
<td>user granting privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>large varchar</td>
<td>foreign key</td>
<td>read only</td>
<td>user receiving privilege</td>
</tr>
<tr>
<td>alter_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>alter privilege</td>
</tr>
<tr>
<td>delete_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>delete privilege</td>
</tr>
<tr>
<td>execute_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>execute privilege</td>
</tr>
<tr>
<td>index_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>index privilege</td>
</tr>
<tr>
<td>insert_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>insert privilege</td>
</tr>
<tr>
<td>lock_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>lock privilege</td>
</tr>
<tr>
<td>references_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>reference privilege</td>
</tr>
<tr>
<td>select_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>select privilege</td>
</tr>
<tr>
<td>update_priv</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>update privilege</td>
</tr>
<tr>
<td>alter_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>alter privilege has grant option</td>
</tr>
<tr>
<td>delete_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>delete privilege has grant option</td>
</tr>
<tr>
<td>execute_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>execute privilege has grant option</td>
</tr>
<tr>
<td>index_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>index privilege has grant option</td>
</tr>
<tr>
<td>insert_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>insert privilege has grant option</td>
</tr>
<tr>
<td>lock_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>lock privilege has grant option</td>
</tr>
<tr>
<td>references_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>references privilege has grant option</td>
</tr>
<tr>
<td>select_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>select privilege has grant option</td>
</tr>
<tr>
<td>update_grantable</td>
<td>bool</td>
<td>not null</td>
<td>read only</td>
<td>update privilege has grant option</td>
</tr>
</tbody>
</table>
10. RTRDB support for developing CL scripts

The RTRDB supports the use of Control Language (CL) executed in the server itself. CL may be used to define a set of methods on a table, defining a behaviour for data contained in the tables. Details of using CL in the RTRDB are in a separate section of the CL Reference manual. It is assumed the reader is familiar with this material.

The RTRDB offers additional support for developing CL scripts for execution in the database. This support allows a developer to control CL execution, reload CL without terminating the RTRDB, and to perform various debugging functions such as setting breakpoints, single stepping through CL threads, and so on.

The additional support is provided through special tables in the RTRDB. These tables and how to use them are described in this section.

10.1 Loading CL

The RTRDB offers two ways of loading CL for execution. Normally, CL scripts are loaded from their source files which are listed in the cl_library resource. Alternatively, if no cl_library resource is defined, the RTRDB automatically loads CL from the .dat file, if any has been saved there. Note that CL in the .dat file is ignored if the cl_library resource is defined. Also note that a message of the form ‘Loading CL from loadfile’ does not mean that there is any cl in a particular load file, only that an attempt to load cl is being made.

CL already loaded into the RTRDB may be saved into a .dat file using the SQL save statement.

10.2 CL execution control

Execution of CL can be disabled and enabled while the RTRDB is running. A table clcontrol is defined in the RTRDB for this purpose. It has the following definition:

```
create table clcontrol
(
  local, transient,
  id integer primary key,
  disable bool
);
```

The RTRDB maintains a single record in the clcontrol table. Updating the value of the disable field to true disables the execution of CL. Updating the value to false enables CL execution.

No records can be inserted into or deleted from the clcontrol table.

Note that the resource variable cl_disable is provided to specify the initial state of the disable field. cl_disable is a boolean resource. A value of true disables CL execution once the RTRDB has started up. A value of false means that CL execution is enabled on start up.
11. Persistence (Standard only)

The RTRDB maintains the database in volatile memory. This enables it to provide fast access and manipulation of the data. However, it does mean that the values are not automatically maintained across a restart of the RTRDB. This section describes how the RTRDB is configured to handle persistent data.

Any user defined table, or columns within a table may be specified as either persistent or transient. Persistent, which is the default, indicates that the values are required to be maintained across a restart of the database. Columns of a table inherit the persistence of the table unless they are declared to the contrary.

Records in a table are preserved if there is at least one persistent column. Records are also preserved if there is a persistent column that references them. Otherwise, records are discarded. All the transient columns of a preserved record are set to NULL.

If a domain column is part of a table, then how the persistence of that domain column is defined in the table definition determines whether the associated column values are maintained across a restart of the database. The persistence of the domain itself is not relevant.

The RTRDB can be configured with journaling disabled or enabled. The handling of these persistent values depends on whether journaling is disabled or enabled within the RTRDB.

11.1 Journaling Disabled

The default configuration for the RTRDB is non-journaling. In this configuration, the application is responsible for maintaining a sufficient level of persistence by periodically issuing snapshot requests to the RTRDB. The SQL statement SAVE initiates a snapshot. On a snapshot, the RTRDB saves all data marked as persistent to the currently specified load file. Using this mechanism if the RTRDB terminates unexpectedly, any changes to persistent data made since the last snapshot was requested is lost.

When the RTRDB is shutdown, it automatically performs a snapshot to the current load file. Persistent data is therefore maintained across controlled restarts.

11.2 Journaling Enabled

The RTRDB may be configured to append details of changes to persistent data to the load file. These appended entries are referred to as journal entries. To avoid the effect of file writes blocking the RTRDB, the journal entries are not written immediately but instead queued for writing by a separate thread of execution within the RTRDB. This means they are written after the transaction in which the changes were made has completed. A change to data marked as persistent is not persistent until the corresponding journal entry has been written to the load file. Modifications to the database schema using SQL DDL statements also generate journal entries to be appended to the load file. A client may use the safe-commit mode to cause the return from a transaction to be held off until the changes have been written to the load file and made “safe”, or until an error has occurred. The precise meaning of when a transaction is safe depends on the configuration of the database.

The RTRDB must have read-write access to the file if journaling is enabled.

Changes to domain records which are not part of a table are not journalled.

The journaling functionality requires the optional Fault Tolerance module. Please see the linking instructions for the relevant platform for how to link in this module.
In order to allow applications to control and track the journalling process there is a built-in table `journalcontrol` is maintained by the RTRDB providing information about the transaction for which a journal entry has been written. The table has the following definition:

```sql
create table journalcontrol
(
  transient, local,
  id integer primary key,
  filename large varchar,
  created datetime,
  disable bool,
  mode integer not null,
  replication_depth integer not null,
  first_transaction integer,
  last_transaction integer,
  current_transaction integer,
  file_size integer
);
```
The meaning of each column is as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Unique id for the record.</td>
</tr>
<tr>
<td>filename</td>
<td>Name of the current database load file.</td>
</tr>
<tr>
<td>created</td>
<td>Date and time the database load file was created.</td>
</tr>
<tr>
<td>disable</td>
<td>Indicates whether journaling is enabled.</td>
</tr>
<tr>
<td>mode</td>
<td>The fault mode in which the RTRDB is running.</td>
</tr>
<tr>
<td>replication_depth</td>
<td>The replication depth.</td>
</tr>
<tr>
<td>first_transaction</td>
<td>Transaction id of the snapshot written to the load file.</td>
</tr>
<tr>
<td>last_transaction</td>
<td>Transaction id of the last transaction written to the load file.</td>
</tr>
<tr>
<td>current_transaction</td>
<td>Transaction id of the current transaction.</td>
</tr>
<tr>
<td>file_size</td>
<td>The current size of the load file in bytes.</td>
</tr>
</tbody>
</table>

There is only ever one record in the table maintained by the RTRDB, corresponding to the current load file. It is not possible to insert records into or delete records from the table. The disable column can be updated to disable and enable the generation of journal entries. No other columns can be updated.

If the disable column is set to true by the user, journaling will cease. Should the flag be set back to false, a new load file is created as a snapshot of the current state of the database and journaling commences to that load file. If the database is started with a load_file resource specifying a non-existent file and the enable_journalling resource is set to false, setting the disable column to false will cause a new load file of the name specified in the load_file resource to be created and journaling to commence.

The mode column can have one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Standalone or master without standby</td>
</tr>
<tr>
<td>1</td>
<td>Master with standby</td>
</tr>
<tr>
<td>2</td>
<td>Standby</td>
</tr>
<tr>
<td>3</td>
<td>Replica</td>
</tr>
</tbody>
</table>

The replication_depth column will have the value 0 if the RTRDB is running in standalone, master or standby mode and will have value will be 1 or greater if the RTRDB is running in replica mode.

If the RTRDB unexpectedly terminates while it is writing a journal entry, the final journal entry in the load file may be inconsistent. This is detected by the use of checksums upon reloading of the load file and any incomplete journal entries are discarded. The database reports that the load file ended with an invalid journal entry and starts normally. The transactional integrity of the database is maintained.

With journaling enabled the load file increases in size each time a new journal entry is appended. This has two effects: using up disk space and increasing the time the RTRDB takes to restart from the load file. To avoid these problems periodic snapshots must be requested by user code to cause the file to be rewritten from scratch. The size of the file can be monitored from user code by using the file_size field in the journalcontrol table.

Note that on embedded platforms, the calls to manipulate the load file on the file system can be intercepted by user-defined code - see the Polyhedra on Embedded Systems for details.

Journalling is enabled by setting the enable_journalling boolean resource to true.

11.3 Memory mapped file persistence
As an addition to the previous Journalling processes when running on Windows the RTRDB uses memory mapped files to hold data and historical buffers. These memory mapped files can be recovered by using the existing warm restart facility. Checksum support is also available and when enabled it checks the validity of a memory mapped file prior to its recovery. Any historian data structures holding logged values not flushed to disk are stored in memory mapped files. Refer to the *Polyhedra on Windows* user manual for details of how to configure the RTRDB to use memory-mapped files.
The Flash DBMS RTRDB also supports persistent and transient data. Table and column persistent data is defined in the same manner as with the Standard RTRDB, however storage is treated differently.

Transient data is stored in the database file in the same manner as persistent data. However, on restart, any transient records are deleted and any transient columns are reset to their default values.

Note that as with the Standard RTRDB, the tables are by default persistent.

The journalcontrol table, as mentioned in section 11.2, is also present in the Flash DBMS RTRDB when Fault Tolerance is linked in. In this case, there are a number of changes compared to the standard version:

- The disable field is set to false and is read-only. The Flash DBMS RTRDB is in effect journaling changes to the database file at all times.
- The file_size field does increase with each transaction as changes to the data are written within the database file rather than appended to the end of the file. Hence, it is not necessary, as it is with the Standard RTRDB, to execute a periodic SAVE in order to reduce the size of the file.
- The filename field will be set to ‘poly.fdb’ if no load_file is set.
- The first_transaction and created fields indicate when the file was initially created, be that when the RTRDB started or when then the file was created by a “SAVE INTO” statement.
13. Fault Tolerance

In applications that handle critical information and need high availability, it is not acceptable for the system to stop for any considerable period of time. Such applications require a second standby database that is ready to take over system control if the first fails or stops.

We define a fault tolerant system as a system that provides a specified service in the presence of a bounded number of failures. Starting with this, we must further define two areas: the service being maintained by Polyhedra in the presence of a fault; and the faults tolerated. This is the purpose of sections 13.1 and 13.2; the rest of this section describes how these are provided in Polyhedra.

The fault tolerance functionality requires the optional Fault Tolerance module. Please see the linking instructions for the relevant platform for how to link in this module.

13.1 Service Provided

The service maintained by Polyhedra in the presence of a fault is the availability of the RTRDB. That is, the service provided to its clients is maintained across a failure. To be more precise, it is only a defined subset of the RTRDB service that is fault tolerant; the main constituents being queries, transactions and control messages.

13.1.1 Queries

Queries are either fixed or active.

In this list of maintained service, the term failure implicitly refers to a failure that the system is defined to tolerate, as defined in section 13.2.

- A request for a fixed query is satisfied in the presence of a failure.
- A request to register an active query is satisfied in the presence of a failure.
- The servicing of a registered active query with deltas is maintained in the presence of a failure.
- A request to terminate an active query is satisfied in the presence of a failure.
- A request to update the data retrieved by an active query in the presence of a failure may fail or succeed. There is no indication of whether the transaction has succeeded or failed.

13.1.2 Transactions

A transaction consists of the client issuing SQL DML statements to the RTRDB.

- A request to perform a transaction in the presence of a failure may fail or succeed. There is no indication of whether the transaction has succeeded or failed.

13.1.3 Control Messages

There is currently only one control message supported by the system.

- A shutdown request is satisfied in the presence of a failure.
13.2 Faults Tolerated

The fault tolerance support provided by Polyhedra is designed to tolerate only a single point of failure. We now list the faults tolerated. These are categorised as either software, hardware or system faults.

13.2.1 Software Faults

The following software faults are tolerated by the system:

- RTRDB terminates unexpectedly.
- RTRDB hangs.

Certain instances of the following faults are tolerated:

- RTRDB exhibits an erroneous internal state.

13.2.2 Hardware Faults

The following hardware faults are tolerated by the system:

1. Processor halts.
2. Disk error.
3. Network error.

13.2.3 System Faults

The following system faults may be tolerated by the system:

1. Dynamic storage exhaustion - insufficient memory.
2. Stable storage exhaustion - insufficient disk space.

These faults can only be tolerated if they are transient faults, and, in particular, do not manifest themselves immediately on the standby system.

If a file write issued by the journalling subsystem blocks, this will prevent updates going to the standby and replicas. Such updates are queued. If the queues become full, then the RTRDB may stop processing transactions. Users may wish to make use of the flash filing system interface that can be implemented to fail a Polyhedra call if the underlying system call takes too long.

The RTRDB updates a file if it is the load file or it is used in the SQL `save` statement.

13.3 Clients

The following standard Polyhedra clients support fault tolerant connections to the RTRDB:

- ODBC driver
- JDBC driver
- SQL Client
- CLC Client

Please refer to the appropriate manuals for details of how to configure each of these clients for fault tolerance.
13.3.1 Call-back and ODBC API

The ODBC and Call-back API's are the published interface for implementing Polyhedra clients. The fault tolerant features added to Polyhedra client components are made available through these API's. These facilities are provided in a flexible manner. Memory usage by the API is an important consideration on some platforms and some features of Fault Tolerance require added memory overhead. For this reason the Fault Tolerant features supported by the API are optional. Fault Tolerant functionality is provided is on a per (logical) client connection basis. The API supports optional fail-over in various forms.

Please refer to the ODBC and Call-back API manuals for details.

13.4 System Architecture

The required fault tolerance is provided by a dual redundant database system. A second database reflects the state of the main one and is ready to take over system operation at any time. The terms active and standby are used to refer to these databases. The roles of these databases may switch. A standby database can become active at any time.

Two instances of the Polyhedra software component, RTRDB, are configured, each controlling its own RTRDB. The master RTRDB communicates with the standby RTRDB feeding it the journal entries. The standby RTRDB performs the transactions contained in the journal entries it receives from the RTRDB, replicating the state of the original RTRDB. Figure 13.1 illustrates the configuration.

![Figure 13.1 Fault Tolerant Configuration](image-url)
13.5 System Operation

This section describes the interaction of the various processes comprising a fault tolerant system. It assumes a Polyhedra database is being used as the arbitration mechanism to determine which server of an FT pair is to be the master. Different arbitration mechanisms are supported, as described in section 13.6, which will have the same general principles of operation though the details will vary.

13.5.1 Start-up sequence

The RTRDB performs the following start-up sequence when initiated:

1. It reads the configuration file which contains the following information:
   - The journal service on which the RTRDB listens for standby and/or replica (Standard Only) connections.
   - The service it should use to connect to the arbitrator.
   - An indication of whether it is logging.
   - If logging, the directory for writing log files.
   - The journal service of the other RTRDB.

2. It connects to the arbitrator. Failure to connect means the failure of the RTRDB. The RTRDB registers an active query on the arbitrator to determine which mode it is running in. Once this is known, it completes the start-up appropriately.

3. In master mode, the RTRDB starts up as follows:
   a) The RTRDB loads up from its load file (if specified) and becomes master.

4. In standby mode, the RTRDB starts up as follows:
   a) The RTRDB connects to the master RTRDB. If it is unable to do so, it terminates immediately.
   b) The RTRDB requests the current snapshot from the master RTRDB. The master RTRDB responds by collecting and sending a snapshot. The standby RTRDB, if logging, writes the log. The resource `journal_load_timeout` is provided to prevent the standby (or a replica) waiting forever for the initial snapshot.

13.5.2 Normal operation

Once the RTRDB has completed its start-up sequence it enters normal operation. Its behaviour is dependent upon whether it is in master or standby mode.

13.5.2.1 Master RTRDB

The normal operation of a master RTRDB is as follows:

1. The RTRDB sends a journal entry to the standby RTRDB.
2. If logging, the RTRDB filters out transient changes from the journal entry and writes a record of persistent data to the log (Standard only).
3. When the entry has been processed, the standby RTRDB sends an acknowledgement to the master RTRDB.

It should be noted that the master RTRDB does not maintain a heartbeat to the standby RTRDB and can only detect its absence by a closure of the underlying connection or a failure either to send a journal entry, or to receive an acknowledgement within the interval specified by the resource `journal_transaction_timeout`. If this timeout is exceeded, the master RTRDB closes the connection to the standby and outputs a warning message.
13.5.2.2 Standby RTRDB

The normal operation of a standby RTRDB is as follows:

1. The RTRDB receives a journal entry from the master RTRDB.
2. If logging, the RTRDB filters transient changes from the journal entry and writes a record of persistent data to the log (Standard only).
3. When the standby RTRDB has processed the entry, it sends an acknowledgement to the master RTRDB.

13.5.3 Activation

When a RTRDB running in standby mode switches to master mode, it performs the following sequence:

1. The RTRDB ensures that its connection to the previously master RTRDB is closed.
2. The RTRDB is now master and commences normal operation.

13.5.4 Deactivation

When a master RTRDB is instructed to deactivate, it performs the following sequence:

1. The RTRDB terminates.

13.6 The Arbitrator

This section describes the arbitrator. The arbitrator is a key component of Polyhedra fault tolerant configurations. The arbitrator is not an off-the-shelf part of Polyhedra. Instead, it is a component provided by the user.

The arbitrator must be configured to provide a defined behaviour. This behaviour involves

- monitoring RTRDB tasks
- controlling the operation mode (standby or master) of RTRDB tasks
- ensuring only one RTRDB is master
- instructing a standby RTRDB to take over master in the event of failure of the current master RTRDB.

A number of arbitration mechanisms are possible, including the use of a Polyhedra server, a process handling TCP messages, or other platform-specific methods.

13.6.1 Database arbitrator

The database arbitrator is implemented using an RTRDB, maintaining a table called \texttt{jcpcontrol} to a defined specification (see later).

The arbitrator must define a \texttt{jcpcontrol} table and maintain one record in it for each RTRDB, keeping it up to date with the RTRDB's current operation mode. RTRDB tasks both master and standby connect to the arbitrator and register a single active query on this table and operate accordingly. If the arbitrator changes the records, the RTRDBs will respond appropriately, as described in sections 13.5.3 and 13.5.4.
The table has the definition:

```sql
create table jcpcontrol
(
    name large varchar primary key,
    active bool,
    heartbeat_interval datetime,
    heartbeat datetime,
    transaction_no integer
);
```

The columns of `jcpcontrol` are defined as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Uniquely identifies a record. It contains the journal service name on which a RTRDB listens for connections.</td>
</tr>
<tr>
<td>active</td>
<td>Indicates whether the RTRDB is in master (active) or standby mode. A value of true indicates that the RTRDB should run in master mode, false indicates standby mode. Null indicates that the state of the RTRDB is unknown. If the RTRDB detects a value of null for this column, it updates the column with its current state. This can be used by a re-started arbitrator to obtain the current state of a RTRDB.</td>
</tr>
<tr>
<td>heartbeat_interval</td>
<td>Specifies an interval of time. The RTRDB once connected will update the values of the <code>heartbeat</code> and <code>transaction_no</code> columns every heartbeat interval. This can be used by the arbitrator to detect RTRDB tasks that have terminated or hung.</td>
</tr>
<tr>
<td>heartbeat</td>
<td>Updated by the RTRDB with the current date and time at interval specified by the <code>heartbeat_interval</code> column. This can be used by the arbitrator to detect RTRDB tasks that have terminated or hung.</td>
</tr>
<tr>
<td>transaction_no</td>
<td>An integer updated by the RTRDB with the current transaction every heartbeat interval.</td>
</tr>
</tbody>
</table>

A RTRDB’s connection to the arbitrator is fault tolerant and will automatically reconnect to a re-started arbitrator. Shutting down the arbitrator will cause the fault tolerant system to shut down.
13.6.2 TCP Arbitrator Protocol

The following sections describe the format and use of TCP messages that the arbitrator and the RTRDB exchange.

13.6.2.1 RTRDB messages

All messages from the RTRDB are of the form:

\[ J<\text{mode}><\text{trans}><\text{name}> \]

where

- \(<\text{mode}>\) 4 byte Current mode
- \(<\text{trans}>\) 4 byte Transaction number
- \(<\text{name}>\) RTRDB journal_service resource value

\(<\text{mode}>\) is a 32-bit value that indicates whether the RTRDB is running in active or standby mode or waiting to be told by the arbitrator which mode to start-up in.

0 Unknown
1 Master
2 Standby

\(<\text{trans}>\) is a 32 bit value containing the transaction number. It will be 0xFFFFFFFF (-1 if considered as a signed value) until the RTRDB has started-up. Once the RTRDB has successfully started-up and is ready to accept connections, the value of \(<\text{trans}>\) will be the current value of the current_transaction field in the journalcontrol table.

\(<\text{name}>\) contains the journal_service resource value used by the RTRDB. It is a null terminated string and the message includes the terminating null byte.

All 32-bit values are encoded with the least significant byte first.

13.6.2.2 Arbitrator messages

All messages from the arbitrator are of the form:

\[ A<\text{mode}><\text{trans}><\text{interval}> \]

where

- \(<\text{mode}>\) 4 byte Mode to set
- \(<\text{trans}>\) 4 byte Transaction number
- \(<\text{interval}>\) 4 byte Interval (micro-seconds)

\(<\text{mode}>\) is a 32-bit value that indicates whether the RTRDB is to run in active or standby mode.

1 Master
2 Standby

\(<\text{trans}>\) is a 32-bit value containing the transaction number. Although not currently used by the RTRDB it is recommended that the value should be 0 in the first message sent to an RTRDB. In subsequent messages it should be the last value received from the RTRDB it is sending to.

\(<\text{interval}>\) is a 32-bit value specifying the heartbeat interval, in micro-seconds, that the RTRDB should use.

All 32-bit values are encoded with the least significant byte first.

13.6.2.3 Message use

When the RTRDB starts up it will connect to the port specified by the arbitrator_service resource. It will then send a message to the arbitrator (with mode Unknown and a transaction number of -1).

The RTRDB will expect a response from the arbitrator, which will inform it of its mode and heartbeat interval.
At any time the arbitrator can send a message to change the mode or to change the interval.
The heartbeat message from the RTRDB will be the same format as all other messages, but the
transaction number should be filled in correctly.

13.6.2.4 Controlled Failover

If the TCP arbitrator wants to perform an intentional failover between master and standby it should perform the following steps:

1. Send an arbitrator message with mode = 2 (Standby) to the master RTRDB.
2. Wait until a subsequent message from the master RTRDB reports that it is now in standby mode (it will then close the connection) and note the transaction number in that message.
3. Wait for the original standby to report the last transaction sent in step 2.
4. Send an arbitrator message with mode = 1 (Master) to the original standby.

13.6.2.5 Uncontrolled Failover

If the connection from the TCP arbitrator to the master RTRDB is lost or the TCP arbitrator closes the connection, the RTRDB will perform the following steps:

1. Close all its connections to clients (causing fault tolerant clients to attempt to fail-over to the standby).
2. Perform a snapshot to the load file (if one if configured).
3. Terminate.

It is the TCP arbitrator's responsibility to decide what to do when the connection is lost, but typically it would request that the standby RTRDB is promoted to become the new master.

Note that as this is an uncontrolled failover there may be some transactions saved in the load file that were not applied by the standby. As long as the clients perform transactions in safe commit mode they can be guaranteed that any transactions that have completed will have been committed on both the master and standby. For transactions that fail during a failover, with a “lost connection” error, it is the client's responsibility to check that the transaction was applied and if not apply the transaction again if required.
13.7 Configuration

The RTRDB is configured for fault tolerance by the provision and specification of an arbitration service and a method of communicating with the other RTRDB in the fault tolerant pair.

The following resources may be used for the configuration. See section 7 for further details.

- arbitrator_protocol
- arbitrator_service
- journal_service
- enable_journalling
- load_file_directory
- other_journal_service
14. Data Replication (Standard only)

The RTRDB supports data replication using a fault tolerant pair, standalone RTRDB, another replica or replica pair as the source. The replicated database journals data to a replica database in the same way as a master database does to a standby database. Any number of replica databases may connect to the same source. A replica RTRDB, unlike a standby, does not run under the control of the arbitrator and can never be promoted to a master.

14.1 Configuration

The RTRDB is configured for data replication by specifying values for the following resources:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>journal_service</td>
<td>Specifies the names of the data services on which an RTRDB listens for journal connections.</td>
</tr>
<tr>
<td>replica_listen_mode</td>
<td>A Boolean resource (with default false) indicating whether an RTRDB running in replica mode initiates or listens for connections to obtain its journal stream.</td>
</tr>
</tbody>
</table>
| replication_service| If replica_listen_mode is false, specifies the name of the journal service(s) to which an RTRDB running in replica mode connects to obtain its journal stream.  
If replica_listen_mode is true, specifies a single service on which an RTRDB running in replica mode listens for connections from which it obtains its journal stream. Use of the resource implies the RTRDB is running in replica mode. |

How the connection between a replica RTRDB and its replication source(s) is established depends on the setting of the replica_listen_mode resource. The standard configuration is for a replica to initiate connection to its replication source(s). This is the behaviour when the replica_listen_mode resource is unset or set to false. Alternatively, if the replica_listen_mode resource set to true, the replica will listen for connection from its replication source(s).

If a value is supplied for the journal_service resource, the RTRDB acts as a replication source allowing a replica RTRDB to connect to it to obtain its journal stream. If a value is supplied for the replication_service resource, the RTRDB attempts to run in replica mode. If values are supplied for both the journal_service and replication_service resources, the RTRDB is both a replica RTRDB and a replication source to other replica RTRDBs.

14.2 Standard Replica Configuration

With standard replica configuration the value supplied for the replication_service resource can be either a single replication service or two replication services separated by a comma. If it’s a single replication service, the RTRDB runs in a non-fault tolerant replica mode. That is, it obtains its initial snapshot followed by a journal stream of transactions from the specified replication service. If it’s two replication services, the RTRDB runs in fault tolerant replica mode. That is, it attempts to connect to both replication services, which it verifies are providing a journal stream for the same database (i.e. from same standalone RTRDB or fault tolerant pair), but is able to function when only connected to a single RTRDB running in master or replica mode. It obtains its initial snapshot from the first RTRDB.
running in master or replica mode to which it connects. The order in which the two replication services are specified is not significant.

14.3 Replica Listen Mode

In replica listen mode the value supplied for the replication_service resource must be a single service on which the RTRDB listens for connections from its replication source(s). The RTRDB runs in fault tolerant replica mode and only accepts connection from a fault-tolerant pair, i.e. master and standby. It first waits for connection from a master RTRDB, rejecting all other attempted connections. Once a master is connected, it obtains its initial snapshot followed by a journal stream of transactions and goes ready. It then accepts connection from the associated standby, again rejecting all other attempted connections.

The master and standby initiate the connection to the replica using the JournalReplicaConnection table, which is described in the Utility Classes manual.

14.4 Connection

A replica RTRDB determines the mode in which an RTRDB providing its replication service is running when connection to/from that RTRDB is attempted. If the mode is unacceptable, it immediately disconnects and behaves as if the connection had failed.

A non-fault tolerant replica RTRDB will connect to an RTRDB running in standalone, master or replica mode. It will not connect to an RTRDB running in standby mode.

A fault tolerant replica RTRDB will connect to two RTRDBs either running as a fault-tolerant pair (i.e. master and standby) or, when using standard configuration, both running in replica mode. For both running replica mode they must be replicating the same database and be at the same replication depth.

A fault tolerant replica RTRDB rejects connection to/from a standby RTRDB if not already connected to a master RTRDB. It also rejects connection to/from a master RTRDB if already connected to a master. With standard configuration it also rejects connection to a master RTRDB if already connected to a replica RTRDB and rejects connection to a replica RTRDB if already connected to a master RTRDB.

Once a fault tolerant replica RTRDB has successfully connected to an RTRDB (running in master or replica mode) it obtains its initial snapshot and starts processing transactions from that RTRDB. With standard configuration it then repeatedly attempts to connect to a second RTRDB. In listening mode it then accepts connection from a second (matching standby) RTRDB.

Note that a fault tolerant replica RTRDB can be connected to a replica pair of any byte order. However a fault tolerant replica cannot be connected to a master and standby database with different byte orderings. In this case the replica will reject the standby connection.

14.5 Journal Service

It is possible to configure an RTRDB with values for both the replication_service and journal_service resources.

If a replica RTRDB is configured with a value for the journal_service resource, it attempts to open and start listening for connections on those services when it has successfully connected to a replication service. If it fails to open any of the services, it immediately terminates.

14.6 Replication Depth

When a replica RTRDB successfully connects to its replication service, irrespective of whether or not it is in listening mode, it reports its replication depth indicating the number of times the data has been replicated. A replica RTRDB obtaining data directly from an RTRDB running in standalone or master mode reports a replication depth of 1.
The `journal_replication_depth` resource can be used to override the default maximum replication depth allowed.

### 14.7 Loss of Connection

If a replica RTRDB loses connection to its replication service, or in the case of a fault tolerant replica, to both of its replication services, it terminates.

A fault tolerant replica RTRDB continues to run when only connected to a master or single replica RTRDB and, when not in listening mode, repeatedly tries to re-establish connection to the standby or second replica RTRDB. However it will terminate when only connected to a standby RTRDB and that standby fails to promote to master within a timeout period. This timeout is configurable using the `journal_failover_timeout` resource.

### 14.8 Shutdown

If a standalone or master RTRDB is shutdown, any replica RTRDB connected to it, either directly or indirectly, is also shutdown. Any fault tolerant clients connected to a replica RTRDB shutdown in this way do not attempt to fail-over.

It is possible to shutdown an individual replica RTRDB by connecting directly to it and issuing an SQL SHUTDOWN statement. Any replica RTRDB connected to it does not shutdown and any fault tolerant client connected to it attempts to fail-over.

### 14.9 Snapshot

It is possible to request a replica RTRDB generate a snapshot by using the SQL SAVE statement. The SAVE statement used must contain the name of the file and cannot request journalling, i.e. it must contain an INTO clause and cannot contain a LOGGING clause.

### 14.10 Historian

A replica RTRDB replicates historical data if its historian sub-system is enabled and the replication service is providing historical data. If a replica RTRDB has its historian sub-system enabled but the replication service is not providing historical data, it errors on start-up.

### 14.11 Security

A replica RTRDB only accepts settings for the `default_user` resource where the user specified does not have a password. If the user specified does have a password, the setting is ignored. This prevents any security restrictions imposed in the database from being overridden by a replica RTRDB.
15. Data Subscription (Standard Only)

The RTRDB supports a facility that can subscribe to (or mirror) a subset of tables from another RTRDB. Changes to data in the source tables are applied to the corresponding tables in the destination. This facility is in contrast to the data replication feature where the entire database must be replicated.

- The subscription can be for one or more tables with enforced referential integrity.
- A subset of columns can be specified and non-subscribed columns in the destination may be updated freely.
- A subscription may flatten table hierarchy in the source to a single table in the destination.
- While the subscription is active, the subscribed data in the destination may not be modified, however when the subscription is disabled, the data remains and is writeable. Moreover, when the subscription is enabled again, the source and destination are reconciled such that the destination is brought back into synchronisation with the source.
- An enabled subscription configuration may be saved into a load file. On startup, the RTRDB will not wait for the subscription to become active before becoming ready. The status of the subscription may be monitored using the configuration tables.
- A subscription may be configured to a fault tolerant pair or an FT pair of replicas. In these cases, the subscription will failover.
- A subscription may be configured with heartbeats such that connection failures will be detected and reconnection to the same or other servers attempted.
- Only data pertaining to the subscribed tables and transactions that affect that data is sent from source to destination.
- On reconnection after a communication failure or failover, reduced data is sent to the destination if the subscribed data has not changed in the meantime.
- Two way, bi-directional subscriptions can be configured, allowing data to be subscribed to, as well as from, tables in the server RTRDB.
- Record-level partial subscription is available where records in a subscribed table are sent to the subscriber based on the value in a specified criteria column. This facility is subject to additional referential integrity restrictions when using tables which have foreign key or array columns.

15.1 Configuration.

Data subscription requires configuration on both the client side and the server (passive) side. The facility is provided using a journal connection. Hence the server RTRDB must provide a journal connection as if it were providing data to a replica. See the journal_service resource. Note that setting journal_service also enables the journalling capabilities of the RTRDB. If you do not wish to journal changes to a load file see the enable_journalling resource. A bi-directional subscriber facility on the server RTRDB is configured using the JournalConnection table which controls the available connection sessions and allows the server RTRDB to configure subscriptions to data in the client RTRDB. Without this bi-directional configuration subscriptions can only be made at the client RTRDB to replicate data from the server.

Subscriptions are configured using the JournalSession and JournalSubscription tables. The JournalSession table is used to manage the journal connection to the connecting database. The JournalSubscription table configures the mapping of tables and columns in the source database to
those in the destination. For bi-directional subscription, these same tables are used in the server RTRDB to subscribe to data in the client RTRDB (configuration for subscriptions is always at the end that receives data).

For more information on these tables see the “Polyhedra Utility Classes” manual.

15.2 Data Aggregation

Normally a table may be the destination of a single subscription from a single session. This restriction is relaxed to allow multiple subscriptions from a single session where the source subscriptions share a primary key. It is also possible for a table to be the destination for subscriptions from multiple sources using data aggregation.

When data aggregation is configured for a table, the table is partitioned by extending the primary key of the table with an integer column that is related to the JournalSession record. In this way, the data coming from multiple sessions can be uniquely identified.

15.3 Example

There is an example of read only data subscription and data aggregation in the “demo_subscription” example in the release kit.
16. Local Data

The RTRDB supports the LOCAL constraint, which only has significance in either a fault tolerant configuration or configurations with deployed replicas. Local data are not journalled by the master to the standby or replica databases and it is possible to perform DML on local data on the standby or replica databases. The standby and replica have the same schema definition as the master.

The standby and replica databases will load and run any CL that is defined for local data. Further details can be found in the "CL Reference" manual.

The timer table is not, by default, constrained to be LOCAL, but it may be so constrained.
17. Historian (Standard only)

It is a requirement of many systems implemented on Polyhedra to be able to save away changes of selected items in the database. These changes may be recalled at a later time, either for historic trend graphing or statistical analysis. This feature has been incorporated into the RTRDB as a Historian subsystem. The subsystem is configured through predefined tables in the database. Entries in these tables are used to determine what data should be logged and how, as well as providing a means of accessing historic data.

The Historian allows an application writer to specify data whose history is to be saved. Such data may be stored raw, or compressed into slots of fixed duration, or in both formats simultaneously. Different data lifetimes may be applied to each format.

Historic data may be accessed either by querying a set of user-defined tables in the database which appear to the query be populated with historic data, or through a special interface that returns the requested data packed into a single binary field. The binary value may be unpacked into C structures using a supplied utility library; this function is done by the client process. Using this technique, the Historian is able to provide extremely fast logging and retrieval. Active queries are also supported using either technique.

The Historian also offers the ability to archive historic data to files that can be stored and brought back on-line for querying at a later date. Files and disk space are managed transparently on behalf of the application.

The historian may be used in a fault tolerant situation. A fault-tolerant Historian ensures that the Historian in the standby database holds precisely the same data as the active database and that, on fail-over, the Historian holds precisely the same data as if no fault had occurred and only the transactions surviving the fault had run.
18. Unicode

The RTRDB supports storage of Unicode character data using the UTF-8 encoding. UTF-8 is a variable-width multi-byte encoding of Unicode that is compatible with 7-bit ASCII.

Two alternative character sets are supported for the CHAR and VARCHAR data types.

The default character set is ASCII. This is equivalent to the single character set previously supported. Strictly, it supports 7-bit ASCII encoding of characters. However, it is supported to provide compatibility for those applications that are storing 8-bit (top-bit set) values in a CHAR or VARCHAR.

A UTF-8 character set is also supported. This uses the UTF-8 encoding, as defined by ISO 10646. Hence, it supports the storage of Unicode and 7-bit ASCII.

The RTRDB resource `char_character_set` specifies the character set used by the SQL CHAR and VARCHAR and CL STRING data types. The only acceptable values are ASCII and UTF-8. The default value is ASCII.

The column length attribute optionally specified in a CHAR or VARCHAR column definition specifies a number of characters. In Polyhedra SQL the length attribute is only significant when used with the VARCHAR data type, when it specifies the maximum number of characters for the column. This is the case for both the ASCII and UTF-8 character sets. Hence, if the ASCII character set is used, each character will be allocated one byte, whereas if the UTF-8 character set is used, each character will be allocated 4 bytes (this being the maximum space a Unicode character can occupy using the UTF-8 encoding).

(Flash DBMS only) The `char_character_set` resource is used when creating a database storage file. When opening an existing database storage file, the character set with which the file was created is used, overriding the resource setting.

18.1 Load and Save

The database load file format supports the storage of character data in either the ASCII or UTF-8 character set.

A database load file has a single character set associated with it, either ASCII or UTF-8. This is the character set used by the RTRDB that generated the load file. A load file contains an identification of its character set.

If the RTRDB is configured to use the ASCII character set, it will be possible to start from a load file that uses the ASCII character set, but it will not be possible to start from a load file that uses the UTF-8 character set.

If the RTRDB is configured to use the UTF-8 character set, it is possible to start from a load file that uses either the ASCII or UTF-8 character set. If the load file uses the ASCII character set, the character data it contains will be translated to UTF-8 as it is loaded. However, the load file itself is left unaltered.

18.2 Journalling

The Journal sub-system supports journalling of character data using the either the ASCII or UTF-8 character set.

When journalling is enabled it is not be possible to start from a load file that uses a different character set from the RTRDB.
Both RTRDB components in a fault tolerant configuration must be configured to use the same character set. An error will result when a standby attempts to connect to a master that is using a different character set.

An RTRDB component running as a replica must be configured to use the same character set as any RTRDB to which it connects. An error will result when a replica attempts to connect to an RTRDB that is using a different character set.

18.3 Security

The Security sub-system supports usernames and passwords in either the ASCII or UTF-8 character set.

18.4 DVI

The DVI sub-system supports character data in either the ASCII or UTF-8 character set. The GDI Protocol indicates which character set is being used by the RTRDB.

18.5 Historian

The Historian sub-system supports logging character data in either the ASCII or UTF-8 character set. An historian log file has a single character set associated with it, either ASCII or UTF-8. This is the character set used by the RTRDB that generated the log file. A log file contains an identification of its character set.

It is not possible to load a log file that uses a different character set from the RTRDB.

A conversion utility (FILEUTIL) is available on request that converts a log file that uses the ASCII character set into one that uses the UTF-8 character set. It does this without modifying the character data stored in the log file. It checks each character string stored in the log file to ensure that it does not contain any top-bit set characters. If it does not contain any, it changes the character set associated with the file to UTF-8. If it does contain any top-bit set characters, it errors and terminates without converting the file.

If a log file is converted from ASCII to UTF-8, subsequently updating a character value that was logged before the file was converted may result in the new value being truncated to a smaller number of characters than either the value originally logged or that specified by the length attribute for the column. This only occurs if the new value contains non-ASCII characters.
19. Memory Exhaustion Recovery

The RTRDB tracks the amount of heap memory used by the RTRDB in total, by the CL sub-system and by each client session. The RTRDB will attempt to recover when an attempt to allocate heap memory fails. Recovery may involve the termination of a client operation, client session, CL threads, or if recovery is not possible then the RTRDB is terminated.

19.1 Conditions That Will Trigger Recovery

The following conditions will cause an attempt to recover:

- A request to allocate heap memory is refused by the operating system.
- The amount of heap memory allocated is greater than that specified by the memory_limit resource. By default, there is no limit.
- The amount of heap memory allocated by a client session is greater than that specified by the session_memory_limit resource. By default, there is no limit. This value includes memory for prepared statements, execution of queries, storage of active query data and transactions but not the actual table data.
- The amount of heap memory allocated by the CL sub-system is greater than that specified by the cl_memory_limit resource. By default, there is no limit.

19.2 Recovery

The RTRDB will recover by terminating as little as possible. The following recovery actions are attempted:

- The client’s operation is terminated and an error returned to the client if:
  - A static query was being evaluated.
  - A transaction was in progress (INSERT/UPDATE/DELETE/SEND)
  - A statement was being prepared.
- A client’s session is terminated if:
  - An active query was being re-evaluated.
- A SAVE statement causes an error to be printed if:
  - The SAVE operation was in progress in the background (normally when the client issues the statement without setting safe-commit)
- A CL thread is terminated if:
  - The CL thread was being executed.
- All CL threads are killed, the poly_component_error function is called with error code 132 and the activate handlers invoked on all objects if:
  - A CL timer was incremented.
  - A thread is resumed after a delay or sleep.
- Otherwise, the RTRDB will be terminated.

19.3 Limitations
Memory exhaustion recovery is intended to protect the database during normal operation. The following operations are not protected from memory exhaustion:

- Database start-up.
- DDL operations (e.g. CREATE, DROP, GRANT, REVOKE).
- Procedure definition.
- Re-compilation of CL.
- Use of the CL TCP and UDP communication classes.

Certain sub-systems are not compatible with the memory exhaustion recovery and the RTRDB will not recover from a heap allocation failure by these sub-systems:

- The Historian sub-system.
- The Device Interface (DVI) sub-system.
- The Debugger sub-system.
- Journalling sub-system (including fault tolerance, data replication and subscription)
20. Performance Metrics

The performance metrics feature provides visibility into the performance and resource usage of a running RTRDB.

To enable the performance metrics table, the `performancemetrics` table (as defined below) must exist on database startup. In general, a user creates a `performancemetrics` record that indicates the measurement (or metric) that is to be observed. The record is then enabled. When a reading is triggered, the record is updated with the latest value of the measurement along with the maximum and minimum values seen since the metric was last enabled or read.

Observable measurements, or “Metrics” are grouped together by “Provider”, usually by feature such as SQL execution, Historian etc. There may be different versions of the metric, for example transactions executed by different data connections. These versions are known as “Instances”. There is usually a global instance denoted by “+” which gives the value for the metric for the whole system and there may be others that give values for a specific aspect such as a client session.

There are two types of metric. Firstly, a metric with a name starting with “Current” shows the level of some resource. All other metrics summarise information about events in the system. For example, the “CurrentActiveQueries” metric of the “SQL” provider shows the number of active queries currently being requested (and kept up to date) whereas the “ActiveQueries” metric of the same provider shows the number of active queries requested since the last measurement of the metric. So if 10 active queries were requested and 3 closed before a subsequent reading of the metrics, “CurrentActiveQueries” would show a value of 7 and “ActiveQueries” a value of 10.

20.1 Configuration

```sql
CREATE TABLE performancemetric (
    provider       large varchar not null,
    metric         large varchar not null,
    instance       large varchar not null,
    value          integer64,
    timestamp      datetime,
    maxvalue       integer64,
    maxtimestamp   datetime,
    minvalue       integer64,
    mintimestamp   datetime,
    enabled        bool default false not null,
    trigger        bool default false not null,
    description    large varchar,
    PRIMARY KEY (provider, metric, instance)
);
```

In order for the performance metrics component to be active, a table compatible with the above definition must exist on database startup. The table may be local, transient or persistent. The table may be derived from, may contain extra columns and may have CL defined on it. If the table is not
defined correctly, a warning is output and the performance metrics functionality is disabled.

The columns have the following behaviour:

<table>
<thead>
<tr>
<th>Provider</th>
<th>A component or functionality that groups metrics together. Provider names are case-sensitive.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
<td>An entity that can be measured. Metric names are case-sensitive.</td>
</tr>
<tr>
<td>Instance</td>
<td>A specific instance of a metric. System-wide metrics will have a value of “+”. Auto-generation metrics will have a value of “*”. Otherwise the value must match an instance defined in the documentation of the provider and metric. Where the instance is an integer, the column value must contain no spaces or leading zeros.</td>
</tr>
<tr>
<td>Value</td>
<td>The last read value of the metric. The value may denote a count, bytes or microseconds depending on the metric.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>The time that the metric was last read.</td>
</tr>
<tr>
<td>MaxValue</td>
<td>The maximum value that the metric attained. This is either the highest level that a “Current” metric attained or, for other metrics, the highest value of an event.</td>
</tr>
<tr>
<td>MaxTimestamp</td>
<td>The first time that the maximum value was recorded since the metric was last read.</td>
</tr>
<tr>
<td>MinValue</td>
<td>The minimum value that the metric attained. This is either the lowest level that a “Current” metric attained or, for other metrics, the lowest value of an event.</td>
</tr>
<tr>
<td>MinTimestamp</td>
<td>The first time that the minimum value was recorded since the metric was last read.</td>
</tr>
<tr>
<td>Enabled</td>
<td>Set to true to start measuring the metric. The transaction will be failed if a metric with incorrect Provider, Metric or Instance is enabled. Set to false to stop measuring the metric.</td>
</tr>
<tr>
<td>Trigger</td>
<td>Set to true to fill in the value and timestamp columns with the latest measured data for the metric. The column will be set back to false as part of the reading process.</td>
</tr>
<tr>
<td>Description</td>
<td>The description of the metric, which is set when the metric is either enabled or auto-generated.</td>
</tr>
</tbody>
</table>

20.2 Operation

20.2.1 Finding Available Metrics

A record with “*” in both Provider and Metric columns will, when enabled, be replaced by records for all metrics of all providers that are available in the RTRDB. The Instance column must be set to something as it is part of the primary key, however the value is ignored except in the following case. If the Instance column is “*” then only metrics that can auto-generate instances are created, and those metrics will have the Instance column set to “*”. Otherwise all metrics are found and the Instance column will be set to the “+” global value.

A record with “*” in the Metric column and a valid value for the Provider column will, when enabled, be replaced by records for all metrics of the specified Provider. The Instance column must be set to something as it is part of the primary key, however the value is ignored except in the
following case. If the Instance column is ‘*’ then only metrics that can auto-generate instances are created, and those metrics will have the Instance column set to ‘*’. Otherwise all metrics are found and the Instance column will be set to the “+” global value.

The Description of the metric will be filled in on all created records.

The set of providers in an RTRDB will depend on what features have been included in the RTRDB. Please note that the set of metrics included in the RTRDB may be altered in future versions.

### 20.2.2 Reading a Metric

In order to get a value of a metric, first the metric record must be inserted into the table. The metric must be enabled by setting the Enabled column to true. At this point the metric is being measured but the value is not shown in the record. To read the metric, set the Trigger column to true. The value and timestamp columns will be filled in as follows and the Trigger column set back to false.

For a metric starting with “Current” the value and timestamps columns have the following meaning:

- **Value** – The current level of the entity being measured.
- **Timestamp** – The time that the value was read.
- **MaxValue** – The maximum value that the entity attained since the measurement was started or the metric was last read.
- **MaxTimestamp** – The time that the entity first attained the maximum value.
- **MinValue** – The minimum value that the entity attained since the measurement was started or the metric was last read.
- **MinTimestamp** – The time that the entity first attained the minimum value.

For other metrics – those related to events that have occurred e.g. “TransactionTime”.

- **Value** – The sum of all the event values that occurred since the metric was last read. The value is 0 if no events have been recorded.
- **Timestamp** – The time that the value was read.
- **MaxValue** – The maximum value of an event since the measurement was started or the metric was last read. The value is NULL if no events have been recorded.
- **MaxTimestamp** – The time of the event that had the maximum value. The time is NULL if no events have been recorded.
- **MinValue** – The minimum value of an event since the measurement was started or the metric was last read. The value is NULL if no events have been recorded.
- **MinTimestamp** – The time of the event that had the minimum value. The time is NULL if no events have been recorded.

Note that for counts of events (e.g. “Transactions”) since the value of each event is 1, the Value will be a count of the events and the min and max values and events will not be useful.

If a transaction that performs a reading of a metric fails for some reason, the reading may be lost.

If the instance that a metric is recording data for goes away (e.g. a client data connection disconnects) then the next read trigger will update the values with the last known data and then disable the record by setting the Enabled field to false. The metric record will not be deleted.

### 20.2.3 Auto-generation of Metrics

It is possible to configure a metric to create a new enabled metric when a linked instance is created. For example, a “TransactionTime” metric from the “SQL” provider can be automatically created
whenever a new client connection is seen in the `DataConnection` table.

To configure auto-generation of a metric, enable the desired metric with a value of "*" for `Instance`. The "*" record will remain and will not be updated by a read trigger. Delete the metric to stop auto-generating those metrics.

An auto-generation will silently overwrite an existing metric record with a primary key that matches the auto-generated metric. Any newly created record will be of the type of the auto-generating record.

### 20.2.4 Startup and Fault Tolerant Behaviour

When a database starts up with enabled records existing in a persistent `PerformanceMetric` table the enabled metrics will be honoured if possible. If it is not possible to start measuring a metric (e.g. the related instance does not now exist), a warning will be output and that record will be disabled.

A local `PerformanceMetric` table will allow the measurement of metrics on standby and replica RTRDBs.

A non-local `PerformanceMetric` table will show the data from the master RTRDB. For this reason it is recommended that the `Trigger`, `Value`, `MaxValue`, `MinValue`, `Timestamp`, `MaxTimestamp` and `MinTimestamp` columns be made local unless you require the metric readings from the master to be available on the standby or replicas.

A non-local `PerformanceMetric` table on a standby will become active when the standby becomes master. All enabled records will be activated if possible. If it is not possible to activate a record (e.g. the related instance does not exist on the standby) then a warning will be output and the record will be disabled.

### 20.3 The Providers

#### 20.3.1 The SQL Provider

The "SQL" provider contains metrics concerned with SQL query and transaction execution. A "+" instance provides system wide information. Otherwise the instance must match the `Id` of the `DataConnection` table and the metric is restricted to that data client connection.

- **Transactions** – The number of transactions attempted.
- **TransactionTime** – The duration (in microseconds) of a transaction execution.
- **TransactionFailures** – The number of transactions that did not complete successfully.
- **TransactionInserts** – The number of rows inserted by SQL commands in a transaction.
- **TransactionUpdates** – The number of rows updated by SQL commands in a transaction.
- **TransactionDeletes** – The number of rows deleted by SQL commands in a transaction.
- **CurrentActiveQueries** – The number of active queries currently being monitored.
- **ActiveQueries** – The number of active queries requested.
- **ActiveQueryRefetches** – The number of examinations of an active query to determine if information is to be sent to the client.
- **ActiveQueryRefetchTime** – The time taken (in microseconds) to examine an active query to determine if information is to be sent to the client.
- **ActiveQueryDeltas** – The number of active query updates sent to the client.
- **Queries** – The number of SQL queries executed.
- **QueryFailures** – The number of SQL queries that failed.
Performance Metrics

- **QueryTime** – The time taken (in microseconds) to execute an SQL query.
- **QueryRecords** – The number of records returned by an SQL query.
- **CurrentOptimisticLocks** – The number of optimistic locks held by the client.
- **CurrentPessimisticLocks** – The number of pessimistic locks held by the client.
- **OptimisticLockFailures** – The number of transactions failed by an optimistic lock. For a connection-specific instance, the connection with the failed transaction counts the failure.
- **PessimisticLockFailures** – The number of transactions failed by a pessimistic lock. For a connection-specific instance, the connection with the failed transaction counts the failure.

### 20.3.2 The Client Provider

The “Client” provider contains metrics concerned with data client connections to the database. A “+” instance provides system wide information. Otherwise the instance must match the Name of an enabled record in the DataPort table and the metric is restricted to the data service described in that dataport.

- **CurrentConnections** – The number of clients maintaining a connection to the data service.
- **Connections** – The number of successful client connections to the data service.
- **ConnectionFailures** – The number of unsuccessful connection attempts to the data service.
- **LoginFailures** – The number of refused user login attempts on the data service.

### 20.3.3 The Memory Provider

The “Memory” provider contains metrics concerned with the memory usage of the RTRDB. A “+” instance provides system wide information. Otherwise the instance must match the Id of the DataConnection table and the metric is restricted to that data client connection.

In order to enable this provider for “+” instances, the memory_limit resource must be set to a suitable value and to enable the provider for client instances, the session_memory_limit resource must be set.

The values provided by this provider are for net memory requests and do not include memory used by some sub-systems such as the historian or DVI.

- **CurrentMemoryUsed** – The amount of memory (in bytes) in use (allocated but not freed).
- **MemoryAllocated** – The amount of memory allocated (in bytes).
- **MemoryFreed** – The amount of memory freed (in bytes).

### 20.3.4 The Thread Provider

The “Thread” provider contains metrics concerned with the number of operating system threads being used by the RTRDB. The only valid instance is “+”.

- **CurrentThreads** – The number of operating threads created by the RTRDB and still in use. This does not include the starting thread.
- **Threads** – The number of new operating system threads created by the RTRDB.

### 20.3.5 The Journal Provider

The “Journal” provider contains metrics concerned with the connections to standbys and replicas. The
only valid instance is “+”.

- **CurrentStandbys** – The number of standbys connected to the RTRDB.
- **CurrentReplicas** – The number of replicas connected to the RTRDB.
- **CurrentSubscribers** – The number of subscription clients connected to the RTRDB.
- **Connections** – The number of connection attempts to the RTRDB.
- **ConnectionFailures** – The number of connection attempts that failed.
- **CurrentGlobalPacketsQueued** – The RTRDB sends transactions first via a single global queue and then onto separate queues to each journal connection. This metric is the length in packets of the global queue.
- **CurrentPacketsQueued** – The RTRDB sends transactions first via a single global queue and then onto separate queues to each journal connection. This metric is the length in packets of the individual journal queues. Since this metric shows the information of a number of queues, the **MaxValue** will show if any individual journal connection is backing up.
- **SnapshotTime** – The time (in microseconds) taken to generate and send a snapshot of the database (or subset of) for a standby, replica or subscriber. This time includes time taken for any historian data.
- **ConnectionReadyTime** – The time (in microseconds) from a master receiving a connection from a standby until the standby indicates that it is ready. Note that this metric will not show anything for replica or subscriber connections.

### 20.3.6 The CL Provider

The “CL” provider contains metrics concerned with the CL language. The only valid instance is “+”.

- **CurrentThreads** – The number of CL threads executing or suspended.
- **CurrentMemoryUsed** – The amount of memory (in bytes) currently in use by the CL system. The **cl_memory_limit** resource must be set to something appropriate for this metric to be operational.
- **UnhandledExceptions** – The number of unhandled CL exceptions.
- **Transactions** – The number of transactions attempted by CL – including those caused by a CL handler on the **timer** table.
- **TransactionTime** – The time (in microseconds) taken by a transaction started by CL.
- **TransactionFailures** – The number of transactions started by CL that failed to complete successfully.
- **Activations** – The number of times that CL is triggered by an on set, on delete, or on create handler, an activation event or an SQL SEND statement. Recursive activations are not counted.

### 20.3.7 The DVI Provider (Standard only)

The “DVI” provider contains metrics concerned with the DVI sub-system. A “+” instance provides system wide information. Otherwise the instance must match the **device_name** of an enabled record in the **dvi_device** table and the metric is restricted to that device.

- **MessagesIn** – Number of messages received from a DVI device.
- **MessagesOut** – Number of messages sent to a DVI device.
- **Transactions** – Number of transactions attempted by the DVI. The only valid instance is “+”.

79
Performance Metrics

- **TransactionTime** – The time (in microseconds) taken by a transaction executed by the DVI. The only valid instance is “+”.
- **TransactionFailures** – The number of transactions started by the DVI that fail to complete successfully. The only valid instance is “+”.

### 20.3.8 The Historian Provider (Standard only)

The “Historian” provider contains metrics concerned with the historian sub-system. A “+” instance provides system wide information. Otherwise the instance must match the id of an enabled record in the `logcontrol` table and the metric is restricted to that historical stream.

- **CurrentTags** – The number of tags for a historical stream. A “+” instance is not valid for this metric.
- **CurrentEnabledObjects** – The number of objects that are enabled to log with a historical stream. A “+” instance is not valid for this metric.
- **CurrentRawBlockHeader** – The size (in bytes) of the block header for a raw log file of the historical stream. A “+” instance is not valid for this metric.
- **CurrentCompressedBlockHeader** – The size (in bytes) of the block header for a compressed log file of the historical stream. A “+” instance is not valid for this metric.
- **Flushes** – The number of flushes to disk for a historical stream.
- **FlushTime** – The time (in microseconds) taken to flush a historical stream to disk.
- **Archives** – The number of archive files written to disk.
- **ArchiveTime** – Time (in microseconds) taken to flush unarchived data to the archive file.
- **ArchiveLoads** – The number of general and extract archive files brought online.
- **ArchiveLoadTime** – The time taken (in microseconds) to bring a general or extract archive file online.
- **Samples** – The number of samples of fields made.
21. Errors

This section details the meaning of the errors generated by the RTRDB. These may be generated in response to a request, such as a query or transaction from a client, and comprise a numeric code and descriptive text.

21.1 Error Code Format

A Polyhedra error code is a 32-bit value. Encoded into the value is: the severity of the error (warning, error, fatal); the sub-system generating the error (Relational Object Store (ROS), SQL, CL, Historian, etc); and sub-system specific code; and a general code. The format of the error code is as follows:

<table>
<thead>
<tr>
<th>Section</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>0-1</td>
<td>0 = RESERVED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = WARNING</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = ERROR (transaction aborted)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = FATAL (RTRDB terminated)</td>
</tr>
<tr>
<td>Sub-System</td>
<td>2-7</td>
<td>0 = Communications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Client</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = ROS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 = CL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 = DVI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 = Historian</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 = Journal</td>
</tr>
<tr>
<td>Sub-System Specific Code</td>
<td>8-15</td>
<td>Value defined by sub-system</td>
</tr>
<tr>
<td>General Error Code</td>
<td>16-31</td>
<td>General error</td>
</tr>
</tbody>
</table>

A complete list of codes is given in section 21.3.

21.2 Error Text

The error text gives more information about the generic error described by the error code.

The RTRDB supports different levels of detail in the error text to avoid impacting the performance of production systems. This is controlled with the integer resource `error_text_detail`. The following values are supported:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This is the default value. Error text contains a brief description of the type of error (this may include the name of the table affected).</td>
</tr>
<tr>
<td>2</td>
<td>Errors contain text identifying, if applicable, the record and column causing the error.</td>
</tr>
<tr>
<td>3</td>
<td>In addition to the text generated for level 2, the error will contain text identifying, for example, the record and foreign key column containing the invalid reference.</td>
</tr>
</tbody>
</table>

The default value for the resource may change in a future release.

21.3 Error Codes

The following table lists all currently generated general error codes, which are encoded into the 32 bit error code as described in section 21.1:
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>Table not found</td>
</tr>
<tr>
<td>0x0002</td>
<td>Table already exists</td>
</tr>
<tr>
<td>0x0003</td>
<td>Table required</td>
</tr>
<tr>
<td>0x0004</td>
<td>Cannot insert into table – read-only</td>
</tr>
<tr>
<td>0x0005</td>
<td>Cannot update table – read-only</td>
</tr>
<tr>
<td>0x0006</td>
<td>Cannot delete from table – read-only</td>
</tr>
<tr>
<td>0x0007</td>
<td>Cannot send to table</td>
</tr>
<tr>
<td>0x0008</td>
<td>Cannot alter, rename or drop table – read-only</td>
</tr>
<tr>
<td>0x0009</td>
<td>Table name already exists (in a statement)</td>
</tr>
<tr>
<td>0x000A</td>
<td>Table alias already exists as a table name (in a statement)</td>
</tr>
<tr>
<td>0x000B</td>
<td>Table alias already exists as an alias (in a statement)</td>
</tr>
<tr>
<td>0x000C</td>
<td>Table definition has no primary key</td>
</tr>
<tr>
<td>0x000D</td>
<td>The primary key of a table definition is a partial foreign key</td>
</tr>
<tr>
<td>0x000E</td>
<td>Invalid array type for column definition</td>
</tr>
<tr>
<td>0x000F</td>
<td>More than one DERIVED clause in a table definition</td>
</tr>
<tr>
<td>0x010</td>
<td>More than one PERSISTENCE clause in a table definition</td>
</tr>
<tr>
<td>0x012</td>
<td>Column already exists in a table definition</td>
</tr>
<tr>
<td>0x013</td>
<td>Column already exists in a parent table definition</td>
</tr>
<tr>
<td>0x014</td>
<td>Unknown type for column definition</td>
</tr>
<tr>
<td>0x015</td>
<td>DEFAULT clause not allowed for column</td>
</tr>
<tr>
<td>0x016</td>
<td>DEFAULT clause is invalid</td>
</tr>
<tr>
<td>0x017</td>
<td>Domain cannot contain a unique constraint</td>
</tr>
<tr>
<td>0x018</td>
<td>No columns in a UNIQUE clause</td>
</tr>
<tr>
<td>0x019</td>
<td>No columns in a PRIMARY KEY clause</td>
</tr>
<tr>
<td>0x01A</td>
<td>Unknown constraint</td>
</tr>
<tr>
<td>0x01B</td>
<td>More than one SHARED or VIRTUAL clause for a column definition</td>
</tr>
<tr>
<td>0x01C</td>
<td>More than one PERSISTENCE clause for a column definition</td>
</tr>
<tr>
<td>0x01D</td>
<td>More than one HIDDEN clause for a column definition</td>
</tr>
<tr>
<td>0x01E</td>
<td>More than PRIMARY KEY clause in table definition</td>
</tr>
<tr>
<td>0x01F</td>
<td>Parent table already defines PRIMARY KEY</td>
</tr>
<tr>
<td>0x020</td>
<td>Primary key column not found</td>
</tr>
<tr>
<td>0x021</td>
<td>Primary key column cannot be shared or virtual</td>
</tr>
<tr>
<td>0x022</td>
<td>Primary key column cannot be an array</td>
</tr>
<tr>
<td>0x023</td>
<td>Primary key column cannot be a domain</td>
</tr>
<tr>
<td>0x024</td>
<td>Primary key column cannot be hidden</td>
</tr>
<tr>
<td>0x025</td>
<td>Column cannot reference a system table</td>
</tr>
<tr>
<td>0x026</td>
<td>Foreign key column not found</td>
</tr>
<tr>
<td>0x027</td>
<td>Column already foreign key</td>
</tr>
<tr>
<td>0x028</td>
<td>Foreign key cannot be an array</td>
</tr>
<tr>
<td>0x029</td>
<td>Foreign key column cannot have a DEFAULT clause</td>
</tr>
<tr>
<td>0x02A</td>
<td>More primary key columns specified than foreign in table</td>
</tr>
<tr>
<td>0x02B</td>
<td>Foreign key column not in the primary key of the target table</td>
</tr>
<tr>
<td>0x02C</td>
<td>Foreign key mismatch</td>
</tr>
<tr>
<td>0x02D</td>
<td>Foreign key column already used</td>
</tr>
<tr>
<td>0x02E</td>
<td>Foreign key column has different status (shared, etc) from other columns</td>
</tr>
<tr>
<td>0x02F</td>
<td>Foreign key column has different null rule from other columns</td>
</tr>
<tr>
<td>0x030</td>
<td>Foreign key column has different persistence from other columns</td>
</tr>
<tr>
<td>0x031</td>
<td>Missing columns for foreign key</td>
</tr>
<tr>
<td>0x032</td>
<td>Unique constraint column not found</td>
</tr>
<tr>
<td>0x033</td>
<td>Unique constraint column cannot be shared or virtual</td>
</tr>
<tr>
<td>0x034</td>
<td>Unique constraint column cannot be an array</td>
</tr>
<tr>
<td>0x035</td>
<td>Unique constraint column cannot be a domain</td>
</tr>
<tr>
<td>0x036</td>
<td>Unique constraint column cannot be hidden</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0037</td>
<td>Identical unique constraint placed on table</td>
</tr>
<tr>
<td>0x0038</td>
<td>Unique constraint contains partial foreign key</td>
</tr>
<tr>
<td>0x0039</td>
<td>Modification not present, part of foreign key</td>
</tr>
<tr>
<td>0x003A</td>
<td>Column changed twice</td>
</tr>
<tr>
<td>0x003B</td>
<td>Primary key column cannot allow nulls</td>
</tr>
<tr>
<td>0x003C</td>
<td>Primary key column cannot be hidden</td>
</tr>
<tr>
<td>0x003D</td>
<td>Array column cannot be hidden</td>
</tr>
<tr>
<td>0x003E</td>
<td>Foreign key column cannot be NOT NULL and have SET NULL referential action</td>
</tr>
<tr>
<td>0x003F</td>
<td>Null value in column which is to be made NOT NULL</td>
</tr>
<tr>
<td>0x0040</td>
<td>Cannot drop table because another table is derived from it</td>
</tr>
<tr>
<td>0x0041</td>
<td>Cannot drop because a view is defined on it</td>
</tr>
<tr>
<td>0x0042</td>
<td>Cannot drop because an index is defined on it</td>
</tr>
<tr>
<td>0x0043</td>
<td>Cannot alter an inherited columns</td>
</tr>
<tr>
<td>0x0044</td>
<td>Cannot drop column added in same transaction</td>
</tr>
<tr>
<td>0x0045</td>
<td>Cannot drop primary key</td>
</tr>
<tr>
<td>0x0046</td>
<td>Cannot drop table; already dropped in same transaction</td>
</tr>
<tr>
<td>0x0047</td>
<td>Cannot drop partial index</td>
</tr>
<tr>
<td>0x0048</td>
<td>Column in use by index</td>
</tr>
<tr>
<td>0x0049</td>
<td>Table column limit exceeded</td>
</tr>
<tr>
<td>0x004A</td>
<td>Table inheritance limit exceeded</td>
</tr>
<tr>
<td>0x004B</td>
<td>Table record limit exceeded</td>
</tr>
<tr>
<td>0x004C</td>
<td>Transient NOT NULL column not allowed</td>
</tr>
<tr>
<td>0x004D</td>
<td>Table persistence: change not permitted</td>
</tr>
<tr>
<td>0x0050</td>
<td>Domain required</td>
</tr>
<tr>
<td>0x0051</td>
<td>Domain cannot have a primary key</td>
</tr>
<tr>
<td>0x0052</td>
<td>Domain column must be not null</td>
</tr>
<tr>
<td>0x0060</td>
<td>View not found</td>
</tr>
<tr>
<td>0x0061</td>
<td>View already exists</td>
</tr>
<tr>
<td>0x0062</td>
<td>View required</td>
</tr>
<tr>
<td>0x0063</td>
<td>View is not updateable</td>
</tr>
<tr>
<td>0x0064</td>
<td>View definition column mismatch</td>
</tr>
<tr>
<td>0x0065</td>
<td>View column has no name</td>
</tr>
<tr>
<td>0x0066</td>
<td>View column already exists</td>
</tr>
<tr>
<td>0x0067</td>
<td>View contains partial foreign key</td>
</tr>
<tr>
<td>0x0068</td>
<td>View contains a GROUP BY clause</td>
</tr>
<tr>
<td>0x0070</td>
<td>Table or domain required</td>
</tr>
<tr>
<td>0x0071</td>
<td>Table, domain or view required</td>
</tr>
<tr>
<td>0x0072</td>
<td>Table or view required</td>
</tr>
<tr>
<td>0x0080</td>
<td>Column not found</td>
</tr>
<tr>
<td>0x0081</td>
<td>Column already exists</td>
</tr>
<tr>
<td>0x0082</td>
<td>Cannot update column – read-only</td>
</tr>
<tr>
<td>0x0083</td>
<td>Non-domain column referenced</td>
</tr>
<tr>
<td>0x0084</td>
<td>Column name is ambiguous</td>
</tr>
<tr>
<td>0x0085</td>
<td>Column external reference not found</td>
</tr>
<tr>
<td>0x0086</td>
<td>Bad value for column</td>
</tr>
<tr>
<td>0x0090</td>
<td>Index not found</td>
</tr>
<tr>
<td>0x0091</td>
<td>Index already exists</td>
</tr>
<tr>
<td>0x0092</td>
<td>Index required</td>
</tr>
<tr>
<td>0x0093</td>
<td>Index has no columns</td>
</tr>
<tr>
<td>0x0094</td>
<td>Index column cannot be shared or virtual</td>
</tr>
<tr>
<td>0x0095</td>
<td>Index column cannot be an array</td>
</tr>
<tr>
<td>0x0096</td>
<td>Index column cannot be a domain</td>
</tr>
<tr>
<td>0x0097</td>
<td>Index column must be not null</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0098</td>
<td>Index contains partial foreign key</td>
</tr>
<tr>
<td>0x0099</td>
<td>Index creation failed</td>
</tr>
<tr>
<td>0x009A</td>
<td>Index column occurs more than once</td>
</tr>
<tr>
<td>0x00A0</td>
<td>Index column must be local</td>
</tr>
<tr>
<td>0x00A1</td>
<td>Unique index not found</td>
</tr>
<tr>
<td>0x00A2</td>
<td>User not found</td>
</tr>
<tr>
<td>0x00B1</td>
<td>Cannot delete PUBLIC or SYSTEM users</td>
</tr>
<tr>
<td>0x00B2</td>
<td>Insufficient privileges</td>
</tr>
<tr>
<td>0x00B3</td>
<td>No privileges to grant</td>
</tr>
<tr>
<td>0x00B4</td>
<td>Failure to authenticate user</td>
</tr>
<tr>
<td>0x00C0</td>
<td>Procedure not found</td>
</tr>
<tr>
<td>0x00C1</td>
<td>Procedure already exists</td>
</tr>
<tr>
<td>0x00C2</td>
<td>Procedure creation failed</td>
</tr>
<tr>
<td>0x00C3</td>
<td>Cannot redefine procedure</td>
</tr>
<tr>
<td>0x00C4</td>
<td>Cannot undefined procedure</td>
</tr>
<tr>
<td>0x00C5</td>
<td>SELECT not last statement in procedure</td>
</tr>
<tr>
<td>0x00C6</td>
<td>Only one DDL statement allowed in procedure</td>
</tr>
<tr>
<td>0x00C7</td>
<td>No SELECT in procedure</td>
</tr>
<tr>
<td>0x00C8</td>
<td>SELECT in procedure after DLL statement</td>
</tr>
<tr>
<td>0x00C9</td>
<td>DML statement in procedure after non-DML statement</td>
</tr>
<tr>
<td>0x00CA</td>
<td>Insert procedure contains non-DML statement</td>
</tr>
<tr>
<td>0x00CB</td>
<td>Update procedure contains non-DML statement</td>
</tr>
<tr>
<td>0x00CC</td>
<td>Delete procedure contains non-DML statement</td>
</tr>
<tr>
<td>0x00CD</td>
<td>Procedure parameter count mismatch</td>
</tr>
<tr>
<td>0x00CE</td>
<td>Procedure cannot contain DML</td>
</tr>
<tr>
<td>0x00D0</td>
<td>SELECT has no columns</td>
</tr>
<tr>
<td>0x00D1</td>
<td>invalid order by clause</td>
</tr>
<tr>
<td>0x00D2</td>
<td>invalid index in order by clause</td>
</tr>
<tr>
<td>0x00D3</td>
<td>SELECT ORDER BY clause has no columns</td>
</tr>
<tr>
<td>0x00D4</td>
<td>SELECT ORDER BY clause column out of range</td>
</tr>
<tr>
<td>0x00D5</td>
<td>Bad select list for aggregate SELECT</td>
</tr>
<tr>
<td>0x00D6</td>
<td>SELECT DISTINCT clause used on non-distinct query</td>
</tr>
<tr>
<td>0x00D7</td>
<td>Bad where clause for aggregate SELECT</td>
</tr>
<tr>
<td>0x00D8</td>
<td>SELECT processed rows limit exceeded</td>
</tr>
<tr>
<td>0x00DA</td>
<td>No matching columns found via CORRESPONDING clause in UNION/INTERSECT/EXCEPT</td>
</tr>
<tr>
<td>0x00DB</td>
<td>Column in CORRESPONDING clause of UNION/INTERSECT/EXCEPT not found</td>
</tr>
<tr>
<td>0x00DC</td>
<td>Number of columns in component queries of UNION/INTERSECT/EXCEPT does not match</td>
</tr>
<tr>
<td>0x00DD</td>
<td>A column in component queries of UNION/INTERSECT/EXCEPT does not match</td>
</tr>
<tr>
<td>0x00DE</td>
<td>GROUP BY expression contains an aggregate function</td>
</tr>
<tr>
<td>0x00E0</td>
<td>Column name already exists (in statement)</td>
</tr>
<tr>
<td>0x00E1</td>
<td>Too many values</td>
</tr>
<tr>
<td>0x00E2</td>
<td>Not enough values</td>
</tr>
<tr>
<td>0x00E3</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>0x00E4</td>
<td>Evaluation failed</td>
</tr>
<tr>
<td>0x00E5</td>
<td>Invalid foreign key</td>
</tr>
<tr>
<td>0x00E6</td>
<td>More than one record returned by query</td>
</tr>
<tr>
<td>0x00E7</td>
<td>Invalid foreign key for table</td>
</tr>
<tr>
<td>0x00E8</td>
<td>An Aggregate function may not be used as a value for INSERT/UPDATE</td>
</tr>
<tr>
<td>0x00E9</td>
<td>Incompatible type for unary plus</td>
</tr>
<tr>
<td>0x00FA</td>
<td>Incompatible type for unary minus</td>
</tr>
<tr>
<td>0x00F2</td>
<td>Incompatible types for arithmetic operation</td>
</tr>
<tr>
<td>0x00F3</td>
<td>Incompatible types for comparison operation</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>0x00F4</td>
<td>LIKE left-hand side is not a character expression</td>
</tr>
<tr>
<td>0x00F5</td>
<td>LIKE pattern is not a character literal</td>
</tr>
<tr>
<td>0x00F6</td>
<td>LIKE pattern is invalid</td>
</tr>
<tr>
<td>0x00F7</td>
<td>LIKE escape is invalid</td>
</tr>
<tr>
<td>0x00F8</td>
<td>IN clause: bad use of NULL</td>
</tr>
<tr>
<td>0x00F9</td>
<td>IN clause mismatch</td>
</tr>
<tr>
<td>0x00FA</td>
<td>IN clause type mismatch</td>
</tr>
<tr>
<td>0x00FB</td>
<td>Attempt to divide by zero</td>
</tr>
<tr>
<td>0x0100</td>
<td>Function not found</td>
</tr>
<tr>
<td>0x0101</td>
<td>Bad arguments for function</td>
</tr>
<tr>
<td>0x0102</td>
<td>Bad argument type for function</td>
</tr>
<tr>
<td>0x0103</td>
<td>Bad variable argument types for function</td>
</tr>
<tr>
<td>0x0104</td>
<td>Bad argument for aggregate function</td>
</tr>
<tr>
<td>0x0105</td>
<td>Bad argument type for SUM function</td>
</tr>
<tr>
<td>0x0106</td>
<td>Bad argument type for AVG function</td>
</tr>
<tr>
<td>0x0107</td>
<td>Bad argument type for MAX or MIN functions</td>
</tr>
<tr>
<td>0x0110</td>
<td>Parameter not found</td>
</tr>
<tr>
<td>0x0111</td>
<td>Parameter already exists</td>
</tr>
<tr>
<td>0x0112</td>
<td>Parameter has wrong type</td>
</tr>
<tr>
<td>0x0120</td>
<td>Active query does not include primary key</td>
</tr>
<tr>
<td>0x0121</td>
<td>Query columns not unique</td>
</tr>
<tr>
<td>0x0122</td>
<td>Direct insert through active query not allowed</td>
</tr>
<tr>
<td>0x0123</td>
<td>Direct update through active query not allowed</td>
</tr>
<tr>
<td>0x0124</td>
<td>Direct delete through active query not allowed</td>
</tr>
<tr>
<td>0x0125</td>
<td>Direct change through active query not allowed</td>
</tr>
<tr>
<td>0x0126</td>
<td>Cannot directly modify an expression</td>
</tr>
<tr>
<td>0x0127</td>
<td>Query has duplicate primary keys</td>
</tr>
<tr>
<td>0x0128</td>
<td>Failed foreign key constraint in direct insert</td>
</tr>
<tr>
<td>0x0129</td>
<td>Failed foreign key constraint in direct update</td>
</tr>
<tr>
<td>0x012A</td>
<td>Query has invalid SQL</td>
</tr>
<tr>
<td>0x012B</td>
<td>Column was not consistent with the definition</td>
</tr>
<tr>
<td>0x012C</td>
<td>Column already exists in query</td>
</tr>
<tr>
<td>0x012D</td>
<td>Invalid unique index given for object query</td>
</tr>
<tr>
<td>0x012E</td>
<td>Cannot insert into object query</td>
</tr>
<tr>
<td>0x012F</td>
<td>Table has been deleted</td>
</tr>
<tr>
<td>0x0130</td>
<td>Column has been dropped from table</td>
</tr>
<tr>
<td>0x0131</td>
<td>Insertion into an object query is not allowed</td>
</tr>
<tr>
<td>0x0132</td>
<td>Invalid procedure for query</td>
</tr>
<tr>
<td>0x0133</td>
<td>An expression must be named explicitly</td>
</tr>
<tr>
<td>0x0134</td>
<td>DML operations cannot be in the same transaction as a DDL procedure</td>
</tr>
<tr>
<td>0x0135</td>
<td>Query is fixed</td>
</tr>
<tr>
<td>0x0136</td>
<td>Query in a transaction</td>
</tr>
<tr>
<td>0x0137</td>
<td>Query not in a transaction</td>
</tr>
<tr>
<td>0x0138</td>
<td>Active query contains an OFFSET clause</td>
</tr>
<tr>
<td>0x0139</td>
<td>Query cannot be used to create a lock</td>
</tr>
<tr>
<td>0x0140</td>
<td>Cannot mix DML and DDL in transaction</td>
</tr>
<tr>
<td>0x0141</td>
<td>Database is read-only</td>
</tr>
<tr>
<td>0x0142</td>
<td>Shutdown failed</td>
</tr>
<tr>
<td>0x0143</td>
<td>Data conflict</td>
</tr>
<tr>
<td>0x0144</td>
<td>Statement has been invalidated</td>
</tr>
<tr>
<td>0x0145</td>
<td>Statement limit exceeded</td>
</tr>
<tr>
<td>0x0146</td>
<td>Statement timeout exceeded</td>
</tr>
<tr>
<td>0x0150</td>
<td>Lost connection to database</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>0x0151</td>
<td>Cannot apply active query delta</td>
</tr>
<tr>
<td>0x0160</td>
<td>Syntax error</td>
</tr>
<tr>
<td>0x0170</td>
<td>Duplicate key</td>
</tr>
<tr>
<td>0x0171</td>
<td>Not null column not set</td>
</tr>
<tr>
<td>0x0172</td>
<td>Cannot delete record because of referential integrity constraint</td>
</tr>
<tr>
<td>0x0173</td>
<td>Invalid constraint</td>
</tr>
<tr>
<td>0x0174</td>
<td>Cannot update primary key</td>
</tr>
<tr>
<td>0x0175</td>
<td>Cannot update record because of referential integrity constraint</td>
</tr>
<tr>
<td>0x0176</td>
<td>Transaction failed because of optimistic lock failure</td>
</tr>
<tr>
<td></td>
<td>Cannot create a lock because it conflicts with an existing pessimistic lock or an existing lock has already failed</td>
</tr>
<tr>
<td>0x0177</td>
<td>Transaction failed because of conflict with a pessimistic lock</td>
</tr>
<tr>
<td>0x0178</td>
<td>Transaction failed because of a pessimistic lock timing-out and conflicting</td>
</tr>
<tr>
<td>0x0180</td>
<td>Trigger failed</td>
</tr>
<tr>
<td>0x0181</td>
<td>Pre-commit trigger failed</td>
</tr>
<tr>
<td>0x0182</td>
<td>Constraint trigger failed</td>
</tr>
<tr>
<td>0x0190</td>
<td>Feature not implemented</td>
</tr>
<tr>
<td>0x01A0</td>
<td>Local persistent table not allowed</td>
</tr>
<tr>
<td>0x01A1</td>
<td>Local persistent column not allowed in local table</td>
</tr>
<tr>
<td>0x01A2</td>
<td>Local persistent column not allowed</td>
</tr>
<tr>
<td>0x01A3</td>
<td>Local primary key not allowed</td>
</tr>
<tr>
<td>0x01A4</td>
<td>Local data: foreign key column mismatch</td>
</tr>
<tr>
<td>0x01A5</td>
<td>Reference column to local table mismatch</td>
</tr>
<tr>
<td>0x01A6</td>
<td>Array column to local table mismatch</td>
</tr>
<tr>
<td>0x01A7</td>
<td>Local derived table: change not permitted</td>
</tr>
<tr>
<td>0x01A8</td>
<td>Table local setting: change not permitted</td>
</tr>
<tr>
<td>0x01A9</td>
<td>Column local setting: change not permitted</td>
</tr>
<tr>
<td>0x01B0</td>
<td>Local not null column not allowed</td>
</tr>
<tr>
<td>0x01C0</td>
<td>Memory exhausted</td>
</tr>
<tr>
<td>0x01C1</td>
<td>Rollback buffer limit exceeded</td>
</tr>
<tr>
<td>0x01C2</td>
<td>Rows modified limit exceeded</td>
</tr>
<tr>
<td>0x01D0</td>
<td>File open failed</td>
</tr>
<tr>
<td>0x01D1</td>
<td>File close failed</td>
</tr>
<tr>
<td>0x01D2</td>
<td>File read failed</td>
</tr>
<tr>
<td>0x01D3</td>
<td>File write failed</td>
</tr>
<tr>
<td>0x01D4</td>
<td>File seek failed</td>
</tr>
<tr>
<td>0x01D5</td>
<td>File flush failed</td>
</tr>
<tr>
<td>0x01D6</td>
<td>File rename failed</td>
</tr>
<tr>
<td>0x01D7</td>
<td>File delete failed</td>
</tr>
<tr>
<td>0x01D8</td>
<td>Filename required</td>
</tr>
<tr>
<td>0x0200</td>
<td>Historian: column structure of stream may still be changing</td>
</tr>
<tr>
<td>0x0201</td>
<td>Historian: invalid time</td>
</tr>
<tr>
<td>0x0202</td>
<td>Historian: invalid granularity</td>
</tr>
<tr>
<td>0x0203</td>
<td>Historian: negative granularity</td>
</tr>
<tr>
<td>0x0204</td>
<td>Historian: invalid log column</td>
</tr>
<tr>
<td>0x0205</td>
<td>Historian: invalid index method</td>
</tr>
<tr>
<td>0x0206</td>
<td>Historian: update failed</td>
</tr>
<tr>
<td>0x0207</td>
<td>Historian: delete failed</td>
</tr>
<tr>
<td>0x0208</td>
<td>Historian: worker not connected</td>
</tr>
<tr>
<td>0x0209</td>
<td>Historian: insert failed</td>
</tr>
<tr>
<td>0x020A</td>
<td>Historian: raw access</td>
</tr>
<tr>
<td>0x020B</td>
<td>Historian: compressed access</td>
</tr>
<tr>
<td>0x020C</td>
<td>Historian: unable to update name column of a table being logged</td>
</tr>
<tr>
<td>0x020D</td>
<td>Historian: table is being logged by the Historian</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>0x020E</td>
<td>Historian: invalid history DML</td>
</tr>
<tr>
<td>0x020F</td>
<td>Historian: unable to update history value to NULL</td>
</tr>
<tr>
<td>0x0210</td>
<td>Historian update attempted on variable length field</td>
</tr>
<tr>
<td>0x0211</td>
<td>Historian: insufficient data</td>
</tr>
<tr>
<td>0x0212</td>
<td>Historian: corrupt log file detected</td>
</tr>
<tr>
<td>0x1000</td>
<td>Internal errors</td>
</tr>
<tr>
<td>0x1012</td>
<td></td>
</tr>
</tbody>
</table>
Index

Active query, 15, 19
  restrictions, 19
arbitrator, 57
Attribute name, 35
Attributes table, 35
CL module, 10
cl_library, 47
clcontrol, 47
column_privilege persistent, 44
column_privileges, 42
Configuration parameters, 23
Conflict detection, 21
control message, 54
Data modification
  active query, 20
  direct update, 20
  SQL procedure, 20
Data sets, 18
DDL, 15, 22
DDL (Data Definition Language), 10
DDL restriction, 15
default_user, 42
Delta, 19
DML, 15, 22
enable_journalling, 51
Enabling security, 39
Error Codes, 81
  Codes, 81
  Format, 81
fault tolerance
  arbitrator, 58
Fixed query, 15, 19
  restrictions, 19
hardware faults, 55
Historian, 69, 70
Historian subsystem, 69
Historic Data Logging, 72, 81
IndexAttrs table, 36
Indexes table, 36
individual query, 22
JCP Control table, 58
jcpcontrol, 59
journalcontrol, 50
load_file, 31
Module
  CLR, 10
Object store, 10
Schema, 10
Server, 10
SQL, 10
Modules of the RTRDB, 10
Object store module, 10
Overview
  RTRDB, 9
Persistence, 49
Polled active queries, 19
Privileges, 40
Procedure
  SQL, 15
Queries, 54
Query, 15, 18
  active, 15, 19
  Delta, 19
  fixed, 15, 19
  Polled, 19
Resource
  load_file, 31
  suppress_cl, 33
type, 33
Resources, 26
RTRDB
  invoking, 23
overview, 9
RTRDB modules, 10
Running the RTRDB, 23
Schema definition, 37
Schema module, 10
Schema tables
  Attributes, 35
  IndexAttrs, 36
  Indexes, 36
  Tables, 34
  Views, 35
Security, 39
  GRANT, 39
  REVOKE, 39
Server module, 10
software faults, 55
SQL
  procedures, 15
  SQL module, 10
  suppress_cl, 33
table_privileges, 41
table_privileges persistent, 45
### Index

**Real-Time Relational Database**

<table>
<thead>
<tr>
<th>Tables</th>
<th>The Attributes table, 35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute table, 35</td>
<td>Transaction, 15, 20</td>
</tr>
<tr>
<td>IndexAttrs table, 36</td>
<td>conflict detection, 21</td>
</tr>
<tr>
<td>Indexes table, 36</td>
<td>failure, 21</td>
</tr>
<tr>
<td>Tables table, 34</td>
<td>type, 33</td>
</tr>
<tr>
<td>Views table, 35</td>
<td>users persistent, 44</td>
</tr>
<tr>
<td>Tables tables, 34</td>
<td>Views table, 35</td>
</tr>
</tbody>
</table>