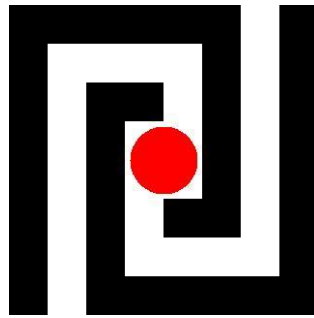


# **The Pyrrho Database Management System**

**Malcolm Crowe, University of the West of Scotland**  
**[www.pyrrhodb.com](http://www.pyrrhodb.com)**



Version 7.010 (September 2025)

## Contents

1. Introducing Pyrrho .....	7
1.1 Features of Pyrrho.....	7
1.2 Pyrrho of Elis.....	8
1.4 How to read this manual .....	8
1.5 About this version.....	8
2. Obtaining Pyrrho.....	10
2.1 Downloading the package.....	10
2.2 System requirements.....	10
2.3 Licensing and Copyright.....	10
2.4 Importing existing data .....	10
2.5 Converting existing database applications .....	11
3. Installing and starting the server .....	12
3.1 Command line options .....	12
3.2 Server account .....	13
3.3 Database folder .....	13
3.4 Security considerations .....	13
3.4.1 Sensitive data.....	14
3.4.2 Mandatory access control .....	14
3.5 Forensic investigation of a database .....	16
3.6 Role-based Data Models.....	17
3.7 Virtual Data Warehousing .....	18
3.8 HTTP and ADO.NET services .....	19
3.8.1 A Transacted REST Service.....	19
3.8.2 A URL-based HTTP service. ....	21
3.9 Localisation and Collations.....	23
3.10 Pyrrho DBMS architecture .....	23
4. Pyrrho client utilities.....	25
4.1 The Pyrrho Connection library .....	25
4.1.2 Localisation.....	25
4.2 Installing the client utilities.....	25
4.3 PyrrhoCmd.....	25
4.3.1 Checking it works .....	26
4.3.2 Accessing a server on another machine .....	26
4.3.3 Connecting to databases on the server .....	26
4.3.4 The QL> prompt .....	27
4.3.5 Multiline SQL statements .....	27
4.3.6 Adding data and blobs to a table .....	27
4.3.7 Retrieving data and blobs from the server.....	27
4.3.8 Command Line synopsis.....	28
4.3.9 Transactions and PyrrhoCmd.....	28
4.4 RESTClient.....	29
5. Database design and creation.....	30
5.1 Creating a Database .....	30
5.2 Creating database objects.....	30
5.2.1 Pyrrho's data type system .....	31
5.2.2 Indexes, Identity etc .....	32
5.2.3 Row versions.....	33
5.2.4 Typed Graph Data.....	33
5.3 Altering tables.....	35
5.4 Sharing a database with other users .....	36
5.5 Roles .....	36
5.6 Stored Procedures and Functions.....	37
5.6.1 Examples.....	38
5.7 Structured Types .....	38
5.8 Triggers.....	39
5.9 Subtype semantics .....	40
5.9.1 IRI references and subtypes .....	40

5.9.2 Row and table subtypes.....	41
5.9.3 Graph Types and Columns.....	41
5.9.4 Value binding in graph patterns .....	41
6. Pyrrho application development .....	43
6.1 Getting Started .....	43
6.2 Opening and closing a connection .....	43
6.3 The connection string.....	44
6.4 REST and POCO .....	45
6.5 DataReaders.....	48
6.6 Using PHP .....	49
6.7 Python.....	49
6.7.1 (AutoKeyAttribute).....	50
6.7.2 DatabaseError .....	50
6.7.3 (Date) .....	50
6.7.4 DocArray.....	50
6.7.5 Document.....	50
6.7.6 DocumentException.....	50
6.7.7 (ExcludeAttribute) .....	50
6.7.8 (Field Attribute) .....	51
6.7.9 PyrrhoArray .....	51
6.7.10 PyrrhoColumn.....	51
6.7.11 PyrrhoCommand .....	51
6.7.12 PyrrhoConnect .....	51
6.7.13 PyrrhoDbType.....	52
6.7.14 PyrrhoInterval .....	52
6.7.15 (PyrrhoParameter).....	52
6.7.16 (PyrrhoParameterCollection) .....	52
6.7.17 PyrrhoReader .....	52
6.7.18 PyrrhoRow .....	52
6.7.19 PyrrhoTable.....	53
6.7.20 PyrrhoTransaction.....	53
6.7.21 Versioned .....	53
6.7.22 WebCtrl.....	53
6.7.23 WebSvc .....	53
6.7.24 WebSvr .....	54
6.8 SWI-Prolog.....	54
7. SQL and GQL Syntax for Pyrrho .....	55
7.1 Statements.....	55
7.2 Data Definition .....	59
7.3 Access Control.....	64
7.4 Type.....	65
7.5 RowSet.....	67
7.6 Scalar Expressions .....	69
7.7 RowSet Expressions .....	71
7.8 Predicates.....	72
7.9 SQL Functions.....	73
7.10 Compliance with the SQL and GQL standards.....	75
7.10.1 SQL-sessions.....	75
7.10.2 SQL-transactions.....	76
7.10.3 Roles .....	76
7.10.4 Privileges.....	76
7.10.5 Drop statements.....	77
7.10.6 Integrity Constraints.....	77
7.10.7 Data Types .....	77
7.10.8 Tables.....	77
8. Pyrrho Reference .....	78
8.1 Diagnostics .....	78
8.1.1 GQLSTATUS .....	78
8.1.2 Get Diagnostics .....	84
8.1.3 Draft GQL compliance statement .....	85

Features where conformance is intended .....	85
Implementation-dependent elements .....	87
Implementation defined specifications.....	88
8.2 Sys\$ table collection .....	91
8.2.1 Sys\$Audit.....	91
8.2.2 Sys\$AuditKey .....	91
8.2.3 Sys\$Classification.....	91
8.2.4 Sys\$ClassifiedColumnData.....	91
8.2.5 Sys\$Enforcement .....	91
8.2.6 Sys\$Graph.....	92
8.2.7 Sys\$Role .....	92
8.2.8 Sys\$RoleUser.....	92
8.2.9 Sys\$ServerConfiguration .....	92
8.2.10 Sys\$User .....	92
8.3 Role\$ table collection .....	92
8.3.1 Role\$Class .....	92
8.3.2 Role\$Column .....	93
8.3.3 Role\$ColumnCheck .....	93
8.3.4 Role\$ColumnPrivilege.....	93
8.3.5 Role\$Domain .....	93
8.3.6 Role\$DomainCheck .....	94
8.3.7 Role\$EdgeType.....	94
8.3.8 Role\$GraphCatalog.....	94
8.3.9 Role\$GraphEdgeType.....	94
8.3.10 Role\$GraphInfo .....	94
8.3.11 Role\$GraphLabel .....	95
8.3.12 Role\$GraphNodeType .....	95
8.3.13 Role\$GraphProperty .....	95
8.3.14 Role\$Index.....	95
8.3.15 Role\$IndexKey .....	96
8.3.16 Role\$Java.....	96
8.3.17 Role\$Method.....	96
8.3.18 Role\$NodeType .....	96
8.3.19 Role\$Object .....	96
8.3.20 Role\$Parameter.....	97
8.3.21 Role\$PrimaryKey.....	97
8.3.22 Role\$Privilege.....	97
8.3.23 Role\$Procedure.....	97
8.3.24 Role\$Python.....	97
8.3.25 Role\$SQL .....	97
8.3.26 Role\$Subobject.....	98
8.3.27 Role\$Table.....	98
8.3.28 Role\$TableCheck.....	98
8.3.29 Role\$TablePeriod.....	98
8.3.30 Role\$Trigger .....	98
8.3.31 Role\$TriggerUpdateColumn.....	99
8.3.32 Role\$Type.....	99
8.3.33 Role\$View .....	99
8.4 Log\$ table collection .....	99
8.4.1 Log\$.....	99
8.4.2 Log\$Check.....	100
8.4.3 Log\$Classification .....	100
8.4.4 Log\$Clearance .....	101
8.4.5 Log\$Column .....	101
8.4.6 Log\$DateType .....	101
8.4.7 Log\$Delete.....	101
8.4.8 Log\$Domain .....	101
8.4.9 Log\$Drop.....	102
8.4.10 Log\$Enforcement.....	102
8.4.11 Log\$Grant.....	102

8.4.12 Log\$Index .....	102
8.4.13 Log\$IndexKey .....	102
8.4.14 Log\$Metadata .....	102
8.4.15 Log\$Modify .....	102
8.4.16 Log\$Ordering.....	103
8.4.17 Log\$Procedure .....	103
8.4.18 Log\$Record.....	103
8.4.19 Log\$RecordField .....	103
8.4.20 Log\$Revoke.....	103
8.4.21 Log\$Role.....	103
8.4.22 Log\$TablePeriod.....	103
8.4.23 Log\$Transaction .....	104
8.4.24 Log\$Trigger .....	104
8.4.25 Log\$TriggerUpdateColumn.....	104
8.4.26 Log\$TriggeredAction.....	104
8.4.27 Log\$Type .....	104
8.4.30 Log\$TypeMethod.....	104
8.4.31 Log\$Update.....	105
8.4.32 Log\$User.....	105
8.4.3 Log\$View .....	105
8.5 Table and Cell Logs.....	105
8.5.1 A Table Log .....	105
8.5.2 A Cell Log.....	106
8.6 Pyrrho Class Library Reference.....	106
8.6.1 AutoKeyAttribute.....	107
8.6.2 DatabaseError .....	107
8.6.3 Date .....	107
8.6.4 DocArray.....	107
8.6.5 Document.....	107
8.6.6 DocumentException.....	108
8.6.7 ExcludeAttribute .....	108
8.6.8 FieldAttribute .....	108
8.6.9 PyrrhoArray .....	108
8.6.10 PyrrhoColumn.....	108
8.6.11 PyrrhoCommand .....	108
8.6.12 PyrrhoConnect .....	109
8.6.13 PyrrhoDbType.....	110
8.6.14 PyrrhoInterval .....	111
8.6.15 PyrrhoList .....	111
8.6.16 PyrrhoParameter.....	111
8.6.17 PyrrhoParameterCollection .....	111
8.6.18 PyrrhoReader .....	111
8.6.19 PyrrhoRow .....	112
8.6.20 PyrrhoTable.....	112
8.6.21 PyrrhoTransaction .....	112
8.6.22 Versioned .....	112
8.6.23 WebCtrl.....	113
8.6.24 WebSvc .....	113
8.6.25 WebSvr .....	113
8.7 The Pyrrho protocol.....	114
8.7.1 Low level-communication .....	114
8.7.2 Sending the connection string .....	115
8.7.3 Protocol details.....	116
8.7.4 Schema.....	119
8.7.5 Column.....	119
8.7.6 Cell.....	119
8.7.7 Type .....	119
8.7.8 Exceptions.....	119
8.7.9 JsonData.....	120
9. Pyrrho Database File Format .....	121

9.1 Data Formats.....	121
9.1.1 Integer format.....	121
9.1.2 Numeric and Real format .....	121
9.1.3 Variant format .....	122
9.1.4 Array, Vector and Multiset format .....	122
9.1.5 Row and User Defined Type format .....	122
9.1.6 Blob format .....	122
9.1.7 Boolean format.....	122
9.1.8 String (Char) format.....	122
9.1.9 Date and TimeSpan formats .....	122
9.1.10 Interval format.....	122
9.2 Record formats.....	122
9.2.1 DataType.....	124
9.2.2 Drop Action .....	125
9.2.3 Fields information .....	125
9.2.4 Update information .....	125
9.2.5 Index flags.....	125
9.2.6 Method type .....	125
9.2.7 Privilege flags .....	125
9.2.8 Trigger type.....	126
9.2.9 Ordering type .....	126
9.2.10 Interval fields .....	126
9.2.11 Metadata flags.....	126
9.2.12 GenerationRule .....	127
9.2.13 Mandatory Access Control Label.....	127
9.2.14 Graph Flags .....	127
9.2.15 Graph Supertypes and Targets .....	127
10. Troubleshooting .....	128
10.1 Destruction and restoration .....	128
10.2 Hardware failure during commit.....	128
10.3 Alternative names for a database file .....	128
10.4 User identity and database migration.....	129
10.5 API Dependency on database history .....	129
11. End User License Agreement .....	130
References .....	131
Index to Syntax .....	132

# 1. Introducing Pyrrho

Pyrrho is a compact and efficient relational database management system for the .NET framework.

Pyrrho implements much of the ISO 9075 SQL standard and also includes other features including support for semantic web services, and data model integration techniques. Databases created by Pyrrho are platform independent, location-independent, and culture-independent.

Since version 4.0 all versions of Pyrrho are freeware: from version 7.0 the professional and open source editions are merged. The principal binaries consist of a stand-alone server (PyrrhoSvr) and a client library (PyrrhoLink). There is a build option for an Embedded edition that intended for the situation that a database is available to just one local application, so that a separate database server is not needed.

## 1.1 Features of Pyrrho

Pyrrho is a rigorously developed relational database management system that can run on small computers but can also scale up to large enterprise uses. It is built for .NET, which is available on Windows, and on Linux and Apple systems with .NET. For best results, the server's main memory should be at least eight times the size of the database. Instead of encouraging large single-database systems, Pyrrho supports view-mediated integration of data from heterogeneous servers in a loose federation.

Pyrrho has strong transactions, designed for business uses. It is most suited to data that includes a regular stream of new information that is to be kept indefinitely, for example, customer data, orders or accounting transactions<sup>1</sup>.

Pyrrho supports the SQL database language, largely compatible with the SQL2023 standard<sup>2</sup>. It is stricter than SQL2023 in some areas: for example, integrity constraints can be deferred to the end of a transaction but cannot be disabled, and the only possible transaction isolation level is `SERIALIZABLE`. In Pyrrho the default is that data types are variable-length and independent of platform and locale. There are practical limits, e.g. integers can be up to 2040 bits, and data uids are limited to 60 bits. For division of non-integer quantities Pyrrho sets a default precision of 13 decimal digits, but higher precision is used if specified. If the specified precision of reals or actual values of integers are sufficiently small, hardware arithmetic is used.

From early 2024, Pyrrho adds support for the GQL database language<sup>3</sup>. The philosophy of GQL is different from SQL, adding horizontal data aggregation and query composition for linked data. Pyrrho's implementation of GQL supports inference of graph types and multiple inheritance. Pyrrho uses the defining position of a node for references if no primary key or identity column has been specified, may supply non-constraining indexes in pattern matching if their absence would have a significant impact on performance, and adds syntax for specification (if desired) of constraints before or after graph creation.

Use cases of graph pattern matching systems today include detection of undesirable behaviour such as plagiarism or fraud. In this context it seems necessary to stress that while Pyrrho supports international character sets and rich data types, it does not contain language models or interpret the meaning of text and uses naïve matching when equality is specified.

The client-server configuration uses a robust TCP-based protocol for communication with clients. The client library is designed as a thread-safe version of the ADO.NET architecture, Pyrrho supplies its own ADO.NET-like classes such as `PyrrhoReader`, `PyrrhoCommand` etc.

Optimistic execution is used as this is more suitable for wide-area operations. Pyrrho supports role, user and timestamp recording for all changes to the database. Transaction log information, including the above details, is recorded permanently in the database file using "append storage" so that deleted or modified data can always be recovered if required, and the physical database file is the transaction log.

---

<sup>1</sup> This is because the transaction log is persistent and contains the complete history of the database (see note later in this section 1.1).

<sup>2</sup> Throughout this manual, SQL2023 denotes the most recent full version of the SQL standard ISO 9075 at the time of writing, including later updates of individual volumes of the standard.

<sup>3</sup> This support includes and extends the support for the Typed Graph Model dating from version 7.03. GQL (ISO DIS 39075) defines additional reserved words (such as `INT32`), which now are treated as reserved in Pyrrho's SQL parser.

The implementation of Pyrrho is in the C# language. Because the database file is the transaction log, Pyrrho typically writes to the disk just once per transaction and performs well in standard benchmark tests.

## 1.2 Pyrrho of Elis

This database management system is named after an ancient Greek philosopher, Pyrrho of Elis (360-272BC), who founded the school of Scepticism. We know of this school from writers such as Diogenes Laertius and Sextus Empiricus, and several books about Pyrrhonism (e.g. by Floridi) have recently appeared.

And their philosophy was called investigatory, from their investigating or seeking the truth on all sides.

*(Diogenes Laertius p 405)*

Pyrrho's approach was to support investigation rather than mere acceptance of dogmatic or oracular utterance.

Accordingly in this database management system, care is taken to preserve any supporting evidence for data that can be gathered automatically, such as the record of who entered the data, when and (if possible) why; and to maintain a complete record of subsequent alterations to the data on the same basis. The fact and circumstances of such data entry and maintenance provide some evidence for the truthfulness of the data, and, conversely, makes any unusual activity or data easier to investigate. This additional information is available, normally only to the database owner, via SQL queries to system tables, as described in Chapter 8 of this manual. It is of course possible to use such automatically-recorded data in databases and applications.

In other ways Pyrrho supports investigation. For example, in SQL2023, renaming of objects requires copying of its data to a new object. In Pyrrho, by contrast, tables and other database objects can be renamed, so that the history of their data can be preserved. Object naming is role-based (see section 3.6).

The logo on the front cover of this manual combines the ancient "Greek key" design, used traditionally in architecture, with the initial letters of Pyrrho, and suggests security in its interlocking elements.

## 1.4 How to read this manual

Each chapter begins with a "getting started" section, and most will have sections towards the end intended for developers or advanced users. The reader is advised to skip over the later sections of chapters on a first reading.

The typographical conventions are as follows: Courier New font is used to indicate computer input or output. Bold face type is used for input, and normal for output, and italic font to indicate items that vary depending on user choices, as in

```
PyrrhoCmd -h:host database  
QL> select * from table
```

The current version of the .NET framework on Linux requires the above command to be given as

```
dotnet PyrrhoCmd.dll -h:host database
```

Similar incantations are needed at present for every .NET executable under Linux. This will not be mentioned every time in this manual, which will generally give the short (Windows) version of commands. Some versions of Linux can be configured with add-ins so that the "dotnet" prefix is not required.

## 1.5 About this version

All databases developed under previous versions of Pyrrho should still work with the latest version of the server<sup>4</sup>. However, when versions change, client applications should be recompiled so that their version of PyrrhoLink matches the server. There is now just one version of the Pyrrho engine:

---

<sup>4</sup> The assumed process is one of migration, with an automatic compatibility mode for migrated databases. Databases created or modified with the latest server version generally cannot be used with previous versions.



PyrrhoSvr.exe: the former open source versions have been merged with these, and applications that formerly used Pyrrho's embedded edition can simply spawn their own local server<sup>5</sup>.

A number of features that Pyrrho once offered have been removed over the years. These have included support for Microsoft technology such as DataAdapters and the Entity Framework, for Java Persistence, SPARQL, OWL, RDF and even Mongo. Some previous editions were linked for use in mobile phones and web servers and allowed multi-database connections.

Version 7 of Pyrrho is a major re-implementation of the database engine, and the architecture modifications are described in the SourceIntro document. The protocol changed slightly in v7.03, when support for the Typed Graph Model, and so older applications should be recompiled with the v7 PyrrhoLink class library.

From version 7.09, we support the new international standard for GQL (ISO/IEC 39075), so that many SQL keywords are still recognized but no longer reserved. Most GQL options are available, with usability extensions that include graph type inference. The GQL query language features the notion of identifier binding, so that unbound identifiers play a special role in the syntax. The prompts in the PyrrhoCmd client now show QL instead of SQL. The Reference sections in this document reflects these changes: the illustrations and worked examples in this document will be updated over time. See section 5.2.4.

Version 7.010 differentiates lists and (sparse) arrays: this has required changes to the Pyrrho protocol.

---

<sup>5</sup> In that case, any databases accessed by the server should be owned by the same account as the application or contain no user names.

## 2. Obtaining Pyrrho

Pyrrho is available as a free and very small download for the .NET framework. Later sections of this chapter discuss issues associated with moving an existing database to Pyrrho.

### 2.1 Downloading the package

The source and binary code of Pyrrho is available from <https://pyrrhodbms.uws.ac.uk> in a single download<sup>6</sup>. Provided the .NET framework (or .NET for Linux) has been installed, it is possible to extract all of the files in the distribution to a single folder and start to use Pyrrho in this folder without making any system or registry changes. Pyrrho currently targets .NET9.0.

You are allowed to view and test the code, and redistribute any of the files available on the Pyrrho website in their entirety. With suitable acknowledgement, you may embed the dlls or re-use any of the source code in any application. Any uses other than those described here requires a license from the University of the West of Scotland (see below).

### 2.2 System requirements

The .NET version 9.0 or .NET for Linux is required. Database files are machine-independent and can be transferred between Windows and Linux or between different machine architectures, provided only that the .NET framework or Mono is installed first<sup>7</sup>.

PyrrhoSvr.exe itself is currently just over 1.5MB, and a minimum of 14MB of memory is required for the server process. However, if the database holds xMB of data then at least 8xMB of main memory is recommended.

### 2.3 Licensing and Copyright

Pyrrho is intellectual property of Malcolm Crowe and the University of the West of Scotland, United Kingdom. The associated documentation and source code, where available, are copyright of Malcolm Crowe and the University of the West of Scotland. Your use of this intellectual property is governed by a standard end-user license agreement, which permits the uses described above without charges. All other use requires a license from the University of the West of Scotland.

Pyrrho depends on the .NET class libraries, which are royalty-free. Pyrrho conforms to the extent described herein to the SQL2023 standard, which is available from the standards bodies (ISO and national bodies).

### 2.4 Importing existing data

When importing tables from an existing database, it is good to take the opportunity for some minor redesign. For example, additional integrity constraints can be added, or data types can be simplified, for example by relaxing field size constraints. Keywords that imply such sizes, e.g. DOUBLE PRECISION, BIGINT etc are not supported in Pyrrho, which provides maximum precision by default. National character sets are deprecated since they make data locale-specific: universal character sets are used by default.

A more important area for attention is Pyrrho's security model, described fully in chapter 5. This offers an opportunity for improving the security of the business process. For simplicity during migration, the current user should initially use the server's account, as this will generally allow all desired operations to be performed with system privileges.

The first thing to note is that Pyrrho expects the operating system to handle user authentication so that there is no way for a user to pretend to be someone else: a custom encryption of the connection string is used to ensure this. There is an implicit business requirement to know which staff took the responsibility for data changes (corresponding to initials in former paper-based systems), and Pyrrho's approach is that

---

<sup>6</sup> At the time of writing, and Pyrrho v7 (alpha) is available at <https://github.com/MalcolmCrowe/ShareableDataStructures> .

<sup>7</sup> Databases that contain user identities are obviously less portable: see section 10.4.

it is undesirable for the database management system to force anonymity on such operations by disguising the staff responsible behind a faked-up application identity.

This means that users of the database should be identified and granted permissions (normally to use one or more database roles). Where the number of authorised staff is large, mechanisms for authorising new users can be automated. It is useful to use the role mechanism to simplify the granting of groups of permissions to the users.

Existing users and roles can be imported from the existing database: assuming users are identified in the existing database by their login identities. Where applications have been given user identities in the legacy system, this should generally be replaced by roles. Ideally each business process should have a role to enable associated database changes to be tracked. Each connection to Pyrrho is for a role, and this can enable a good record of the reasons for changes to data.

## ***2.5 Converting existing database applications***

Stored procedures and view definitions will need to be converted in general since Pyrrho uses the SQL2023 convention whereby identifiers are converted to upper case (not case-sensitive) unless they are enclosed in double quotes. Double-quoted identifiers can include layout and special characters. The use of square brackets instead of double quotes is not supported. Stored procedures must conform to the syntax specified in “SQL2023 – Persistent stored modules”, and are detailed in Chapter 7.

Pyrrho supports the SQL language for coding stored procedures, and a simple version of the ADO.NET application programming interface described in Chapter 6: this allows SQL statements to be used as parameters. Pyrrho v7 has a prepared statement API, described in section 8.7.12. Other ways of embedding SQL into program coding are not supported.

The biggest conceptual hurdle in developing applications for Pyrrho is the use of optimistic transactions. It is very important for programmers to accept this approach as a fact of life, explained in the following paragraphs, and not try to imitate a locking model.

All good database architectures today support the ACID<sup>8</sup> properties of transactions (atomicity, consistency, isolation and durability). Database products that use pessimistic locking (such as SQL Server or Oracle) acquire these locks on behalf of transactions by default, and it is not usually necessary for an application to deal with these issues directly. In a pessimistic locking product, transactions can be delayed (blocked) while waiting for the required locks to become available.

A transaction can fail because it conflicts with another transaction. For example, with pessimistic locking, the server may detect that two (or more) transactions have become deadlocked, that is, all of the transactions in the group is waiting for a lock that is held by another transaction in the group. In these circumstances, the server will abort one of the transactions, and reclaim its locks, so that other transactions in the group can proceed.

With pessimistic locking, if a transaction reaches its commit point, the commit will generally succeed. If it does not complete, it retains locks on database resources until it is rolled back. With SQL Server, for example, once a transaction T begins, it acquires locks on data that it accesses. If it updates any data, it acquires an exclusive lock on the data. Until T commits or is rolled back, no other transaction can access any data written by T or make any change to data read by T.

With optimistic locking, the first sign of failure may well be when the transaction tries to commit. A transaction will fail if it tries to make a change that conflicts with a change made by another transaction and if any data it has read has been changed. Except for syntax errors, any exception will abort the current transaction, unless the exception occurs inside a stored procedure that handles the signal.

In the classic transaction example of withdrawing money from a bank account, a transaction for making a transfer might include an SQL statement of the form “update myaccount set balance=balance-100” or “update myaccount set balance=3456”. Writing SQL statements in the first form makes them apparently easier to restart, but the point being made here is that it should be the client application’s responsibility to decide if the statements should simply be replayed on restart. The server should not simply make assumptions about the business logic of the transaction. Pyrrho transaction checking includes checking that data read by the transaction has not been changed by another transaction.

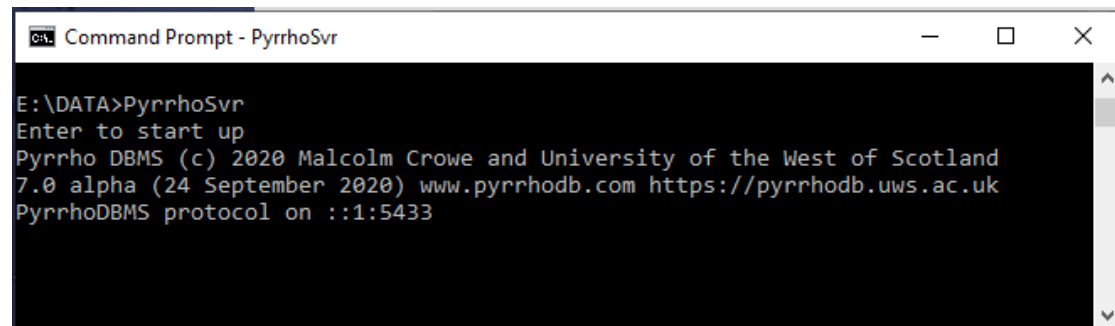
---

<sup>8</sup> In this document, it is assumed that ACID implies in particular that transaction isolation is conflict serializable.

### 3. Installing and starting the server

The server `PyrrhoSvr.exe` is normally placed in the folder that will also contain the database files. Then `PyrrhoSvr` can be started from the command line, by the user who owns this folder. It is a good idea to run the server in a command window, because occasionally this window is used for diagnostic output.

After you start the server, it echoes the command line arguments for you to confirm startup (with the Enter key). If there are no arguments, you should then get confirmation that Pyrrho has started its services:

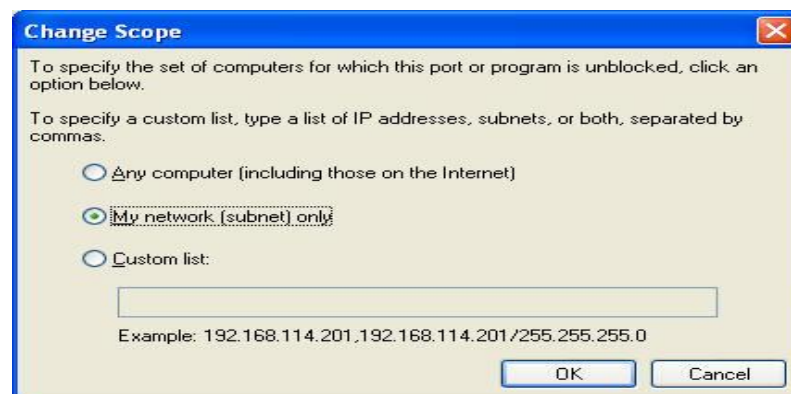


```

Command Prompt - PyrrhoSvr
E:\DATA>PyrrhoSvr
Enter to start up
Pyrrho DBMS (c) 2020 Malcolm Crowe and University of the West of Scotland
7.0 alpha (24 September 2020) www.pyrrhodb.com https://pyrrhodb.uws.ac.uk
PyrrhoDBMS protocol on ::1:5433

```

If Windows announces that it is blocking this program as a precaution, you will need to click the “Unblock” button on this security dialogue if you want to use the server. However, you should configure your firewall to make this service local to your subnet or local machine. The following dialogue box is from Windows XP:



See detailed instructions for Windows Firewall at <http://www.pyrrhodb.com/firewall.htm> .

In Windows 10 there are generally options such as Unblock or Show more to allow full operation of the software.

Under Linux, the command is **dotnet PyrrhoSvr.dll** .

You can stop the server by closing the window, since all committed transactions are already saved to persistent storage.

#### 3.1 Command line options

The command line syntax is as follows:

```

PyrrhoSvr [-d:path] [-h:host] [-p:port] [-r:port] [+s[:port]]
[+S[:port]] [-M:port] [-R] [-E] [-H] [-T] [-V]

```

The `-h` and `-p` arguments are used to set the TCP host name and port number to something other than `::1` and `5433`<sup>9</sup> respectively. This can be a useful and simple security precaution, as any client access must specify the same host address. The host argument should be a valid IP address (IPv4 or IPv6), not a

<sup>9</sup> Port 5433 belongs to PyrrhoDBMS: see <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt> (accessed on 16 November 2024)

computer name (see section 3.4)<sup>10</sup>. The `-d` flag can be used to specify the server's database folder, if the server is placed in another location.

The `+s` and `+S` flags are for starting Pyrrho's HTTP service (see section 3.8). On Windows 7 systems and later, if you get Access denied, you can either run the program as administrator, or you can fix the http url reservations. To do this open a command prompt as administrator and issue the following commands (with your full user name where shown):

```
netsh http add urlacl http://127.0.0.1:8180/ user=DOMAIN\user
netsh http add urlacl https://127.0.0.1:8133/ user=DOMAIN\user
```

If you get other error messages, try using different ports.

Other flags are for instructional use and troubleshooting. The `-T` flag (Tutorial mode) is useful for demonstrating the steps taken by the server for distributed transactions and is less useful in the default situation where this feature is disabled. The `-E` flag can be used to display the per-command execution strategy after optimisation, and the `-V` flag can be used to display the syntax transformations applied to support renaming of database objects. The `-H` flag gives some feedback on the number of rows returned by HTTP requests in the RESTView system.

## 3.2 Server account

PyrrhoSvr.exe, the folder that contains it, and all the database files in this folder are normally owned by the same user, called the server account in the following notes. Note that the logical "database owner" is different – as described in this section.

The server account can always be used to create new databases. Other users who can access the server over the network can generally create databases (but see section 5.1), and naturally become the owner of any databases they create.

From version 7, if the client account matches the server account, the database will not initially contain user or role identities and can be accessed locally by the server account. The first user to be defined in the database becomes the owner, who then can access the database (e.g. over the network). This facility is useful in an educational context where a tutor wishes to create a database for students to copy to their own servers.

In enterprise contexts it is good practice under Linux for the server account to be a server identity such as `S_PYRRHO`, ie. a user identity created on the system, whose only system privileges are to be able to create, delete, read and write files in the server folder, and provide a TCP service on the Pyrrho port (see section 3.1). Things should be set up so that PyrrhoSvr.exe runs under this account, and no other account should have access to the database folder.

The server operates its own security policies (controlled in the usual SQL way by GRANT and REVOKE) on who is allowed to create and access database files. On Windows the client library uses the Windows.Security package to identify the client user ID (Windows login name) and construct an encrypted connection string to pass this to the server.

## 3.3 Database folder

By default, the server will create databases in a folder specified on server startup. You can inspect the database folder from time to time to check everything is in order. A database file path can be used if the server account is able to create and access the given path.

Normal file copying utilities can be used for the database: for example, the server account can copy a database created on another machine into its folder. There is one file per database which is the transaction log. Database files are all owned by the server account<sup>11</sup>.

## 3.4 Security considerations

Pyrrho is a TCP server, and the Internet is generally not a secure place. The Pyrrho DBMS server should be configured behind a firewall, and then accessed from within the firewall by web servers and possibly

<sup>10</sup> The corresponding client-side argument can be a computer name (see chapters 4-8).

<sup>11</sup> The server account can be changed provided file and server accounts continue to match.

local users. This precaution guards against denial-of-service and other attacks. Further instructions for firewall configuration are given at the very start of this chapter.

Within such a firewall, the client-server usage of Pyrrho as described in this booklet should conform to the following levels of security.

1. The security of the database file itself. Naturally, access to the database folder (section 3.3) should be limited to the server and operations staff, and strong password policies should be in place.

To protect against loss, copies of this file should be taken periodically and placed in a secure location. It is good practice to compare successive copies of the database: the database should always match the backup copy over the entire length of the latter. These features facilitate the creation of very secure systems.

2. The security of communication with the server. For all editions of Pyrrho, the connection string is encrypted using a custom encryption technique<sup>12</sup>. In a secure environment, access to the ports would be limited to authorised users, and the port numbers could be changed periodically.

3. The security of user identity for each transaction. The client library obtains the user identity from the operating system and encrypts it in the connection string for secure transmission to the server. Web applications should be configured so that the remote user's identity is correctly passed through using headers (anonymous access should be discouraged).

Within the database, all objects have owners, initially the user that created them (the definer). There are two predefined roles for a database: the default role, with the same name as the database, initially with all privileges, and the guest (public) role, initially allowed to use the default role. The normal SQL grant/revoke mechanisms can be used to modify these permissions (see also section 3.6 and 7.12). From v7 of Pyrrho, in databases that have no users defined, the account that starts the server has all privileges.

See also section 5.1 on the question of permissions for users to create new databases. (This is not really a question of database security.)

### **3.4.1 Sensitive data**

Inspired by the EU's General Data Protection Regulations, Pyrrho now supports the concept of sensitive data, for which any access is auditable. Columns, domains and types can be declared SENSITIVE<sup>13</sup>. Sensitive values are not assignment-compatible with anything that is not sensitive, and there is a sensitive property inherited by any object that contains a sensitive data type. This means for example that the sum of sensitive data is still sensitive. The transaction log will contain a record of every access to sensitive values (apart from by the database owner), even if the transaction is rolled back. Auditing uses the Sys\$Audit system table (see section 8.3.1-2).

### **3.4.2 Mandatory access control**

From December 2018 the DBMS also offers a simulation of Bell-LaPadula security based on clearance and classification levels D to A: the database owner is the security administrator (see section 7). The support is quite extensive, so this section includes some sample discussion. Some aspects of the Bell-LaPadula system are found in current DBMS: essentially the idea that rows of a table can have hidden multi-level security labels that control who can access the rows (and different rows in a table can have different labels).

The access control system is based on the concept of security levels, which are conventionally labelled D to A. Level D is the default and corresponds to no access control beyond the permissions described in the above sections. In the US Department of Defense Orange Book, Levels B and C have subdivisions based on the level of auditing available: since Pyrrho always audits levels above D, its levels C and B roughly equate to levels C2 and B3. Level A requires mathematical proof, which would probably be possible, but is not further discussed here. In addition a security label can contain two lists of identifiers here called groups and references, that are visible only to the security administrator (SA), for the purpose of fine-tuning the authorisations of individual users in individual tables.

---

<sup>12</sup> After connection, further client-server communication is not encrypted by Pyrrho. The use of transport-layer security or alternative ports should be considered if security is an issue.

<sup>13</sup> SENSITIVE is a reserved word in SQL that normally applies to cursor sensitivity. The usage in Pyrrho described here is quite different, and the keyword comes at the end of a type clause (see section 7.4).

A user can be assigned a range of levels<sup>14</sup> called *clearance*, and tables and data records in the database can be assigned a level called *classification*. Initially all users have clearance level D (to D). As mentioned above, both clearance and classification can have lists of groups and references (see syntax below). The clearance and classification labels include the level and two sets of identifiers called here groups and references.

The database owner plays the role of security administrator SA for all objects and users of the database. The database owner has special privileges: to consult all system tables including logs, to access and modify the clearance and classification of users and tables and data records, and to specify the enforcement of these rules for tables in the database. By default, all operations on a table are enforced, but these can be limited to some combination of read/insert/update/delete.

The access rules for users other than the database owner are as follows (where the levels are ordered so that D is the lowest and A the highest). Subject to the normal SQL permissions and the enforcement policy

- A user with clearance x can access data with classification y iff  $x \geq y$
- A user with clearance x can change or create data only with classification x

In addition, the list of references in the user's clearance must include all the references mentioned in the object's classification (if any); and the list of groups in the user's clearance must include at least one of the groups mentioned in the object's classification (if any). The second bullet point above means for example that some users will be able to see objects they are not allowed to modify. If a user inserts a new record in a table where insert is subject to enforcement, the new record will have a classification with the user's minimum level, the subset of the user's groups that are in the table's classification, and all the table's references (which must be a subset of the user's references).

The database owner (as security administrator) is exempt from these access rules. The database owner can specify the classification label for a new table or record. By default a new row will have the same classification as the table that receives it. When called directly or indirectly by the SA, triggers and stored procedures follow the usual (definer's) rules. The SA can also determine for each table whether to apply the access rules just for some combination of read, update, insert and delete operations (by default they are applied for all operations).

The SA can use syntax for level and enforcement descriptors: (as usual [] indicate optional, {} a sequence).

**Level** = LEVEL id ['-id] [GROUPS {id}] [REFERENCES {id}] .

**Enforcement** = SCOPE [READ] [UPDATE] [INSERT] [DELETE]

where the level id is one of the letters D to A.

The SA can add Level and Enforcement to a CREATE or ALTER for tables, specify Level in an INSERT statement or when defining columns, and use SECURITY as a pseudo column in SELECT, UPDATE and DELETE statements.

The SA can assign a clearance level to a user with the following extension to the GRANT statement:

**GRANT Level TO user\_id**

where the user id normally requires to be enclosed in double quotes. The clearance level takes effect immediately on commit, but because of Pyrrho's approach to transaction isolation ongoing transactions will not be affected.

Where a user is unable to access some data because of classification, such data is silently excluded from any direct or indirect computation by that user. If specifically requested information is thus hidden, the requestor will be told that the objects are undefined or that the data is not found. Other exceptions raised by the operation of these rules contain only the information "access denied" (e.g. if a user has been prevented from updating something they have successfully accessed).

There are several system tables that allow the SA to monitor the operation of the above mechanisms. Actions by the SA are visible in the Log\$ table and there are separate tables (Log\$Clearance,

---

<sup>14</sup> A range of levels as a user clearance means that the user is free to read material at a high level and trusted to create at a lower level of security (the minimum they can access), and they can update an object whose classification is in their range (its classification does not change).

Log\$Classify and Log\$Enforcement) that allow SQL access to details of the direct and indirect actions taken by the SA to alter clearance or classification. The current status of all clearances, classified rows, classified columns, and enforcement is available to the SA in the Sys\$Clearance, Sys\$Classification, Sys\$ClassifiedColumnData and Sys\$Enforcement table, respectively, where such status is different from the default.

### 3.5 Forensic investigation of a database

Pyrrho supports two kinds of investigation of a database.

First, full log tables are maintained. These are accessible to the owner of the database. The log files allow tracing back to discover the full history of any object: when it was created, what changes to it were made, and when it was dropped. In each case, full transaction details are recorded: user, role and timestamp. Since objects can be renamed, logs use numeric identifiers to refer to objects in the database. Full details of the log tables are given in chapter 8. Using these tables, it is always possible to obtain details of when and by whom entries were made in the database. The system log refers to columns and tables by their uniquely identifying number rather than by name.

One extension to SQL2023 syntax which assists with forensic investigation is the pseudo-table ROWS(n) where n is the “Pos” attribute of the table concerned in “Role\$Table” (see section 8.3). For example, suppose we want a complete history of all insert, update and delete operations on table BOOK. We first lookup BOOK in Role\$Table:

```
select "Pos" from "Role$Table" where "Name"='BOOK'
```

If this yields 149, then the required history is

```
select * from rows(149)
```

These can of course be combined:

```
select * from rows((select "Pos" from "Role$Table" where "Name"='BOOK'))
```

```

Command Prompt - pyrrhocmd ab

SQL> select "Pos" from "Role$Table" where "Name"='BOOK'
---|
Pos|
---|
149|
---|

SQL> select * from rows(149)
---|-----|-----|-----|-----|
Pos|Action|DefPos|Transaction|Timestamp|
---|-----|-----|-----|-----|
365|Insert|365|347|04/10/2020 11:19:16|
430|Insert|430|412|04/10/2020 11:19:40|
483|Insert|483|465|04/10/2020 11:20:13|
540|Update|483|522|06/10/2020 10:41:07|
605|Delete|605|587|06/10/2020 10:41:30|
---|-----|-----|-----|-----|

SQL> select * from rows((select "Pos" from "Role$Table" where "Name"='BOOK'))
---|-----|-----|-----|-----|
Pos|Action|DefPos|Transaction|Timestamp|
---|-----|-----|-----|-----|
365|Insert|365|347|04/10/2020 11:19:16|
430|Insert|430|412|04/10/2020 11:19:40|
483|Insert|483|465|04/10/2020 11:20:13|
540|Update|483|522|06/10/2020 10:41:07|
605|Delete|605|587|06/10/2020 10:41:30|
---|-----|-----|-----|-----|

```



The second set of parentheses is needed in SQL2023 here to force a scalar subquery. The Log\$ table gives a semi-readable account of all transactions:, and Log\$RecordField enables programmatic access to the data values of Insert and Update records.. Most of the System and log tables have a column called “Pos” which gives the defining position of the relevant entry.<sup>15</sup>

```
SQL> table "Log$"
```

Pos	Desc	Type	Affects
5	PTransaction for 4 Role=-291 User=-292 Time=10/04/2020 11:17:22	PTransaction	5
23	PTable AUTHOR	PTable	23
34	Domain INTEGER	PDomain	34
48	PColumn3 AID for 23(0)[34]	PColumn3	48
71	PIndex AUTHOR on 23(48) PrimaryKey	PIndex	71
92	Domain CHAR	PDomain	92
105	PColumn3 ANAME for 23(1)[92]	PColumn3	105
131	PTransaction for 6 Role=-291 User=-292 Time=10/04/2020 11:18:07	PTransaction	131
149	PTable BOOK	PTable	149
159	PColumn3 BID for 149(0)[34]	PColumn3	159
184	PIndex BOOK on 149(159) PrimaryKey	PIndex	184
206	PColumn3 AUTH for 149(1)[34]	PColumn3	206
233	PIndex2 on 149(206) ForeignKey refers to [71]	PIndex2	233
252	PColumn3 TITLE for 149(2)[92]	PColumn3	252
280	PTransaction for 2 Role=-291 User=-292 Time=10/04/2020 11:18:34	PTransaction	280
298	Record 298[23]: 48=1,105=Dickens	Record	298
323	Record 323[23]: 48=2,105=Conrad	Record	323
347	PTransaction for 1 Role=-291 User=-292 Time=10/04/2020 11:19:16	PTransaction	347
365	Record 365[149]: 159=10,206=1,252=A Tale of Two Cities	Record	365
412	PTransaction for 1 Role=-291 User=-292 Time=10/04/2020 11:19:40	PTransaction	412
430	Record 430[149]: 159=11,206=2,252=Nostromo	Record	430
465	PTransaction for 1 Role=-291 User=-292 Time=10/04/2020 11:20:13	PTransaction	465
483	Record 483[149]: 159=12,206=1,252=Dombe & Son	Record	483
522	PTransaction for 1 Role=-291 User=-292 Time=10/06/2020 10:41:07	PTransaction	522
540	Update 483[149]: 159=12,206=1,252=Dombe and Son Prev:483	Update	483
587	PTransaction for 1 Role=-291 User=-292 Time=10/06/2020 10:41:30	PTransaction	587
605	Delete Record 365[149]	Delete1	365

```
SQL>
```

The normal way for ownership of a Pyrrho database to be changed is for the database owner to invoke the Pyrrho-specific GRANT OWNER statement. This is implemented as part of the normal database service, and it is good practice to ensure that owners of database objects (see section 7.13) are user identities that are still available in the operating system.

### 3.6 Role-based Data Models

At any time, a database connection in Pyrrho has a user id and a role. On Windows systems, the user is obtained from Windows, and the default role has the same name as the database. Another role that the user is allowed to use can be specified in the connection string, or specified by the SET ROLE statement. Pyrrho allows database objects to be renamed or altered by holders of the appropriate permissions: but from Pyrrho 4.5 such renaming and alteration applies to the current role, so that a database object can have different names in different roles.<sup>16</sup>

By default, all roles in a Pyrrho database have a default data model based on the base tables, their columns, and using foreign keys as navigable properties. Composite keys use the list notation for values e.g. (3,4) and the name is the reserved word key, which can be suffixed by the property name of the key component. The default data model can be modified on a per-role basis to provide more user-friendly entity and

<sup>15</sup> “Pos” and many other columns in the system tables have the integer subtype Position, which is specially handled in Pyrrho. See sec 8.1.3.

<sup>16</sup> In Pyrrho versions 4.5 to 6.3, this mechanism was implemented by modifying source SQL contained in view, trigger and procedure definitions to contain defining positions instead of object names before storing the definition in the database. This behaviour was detectable in system tables such as Log\$. In version 7 and later, the source SQL is stored unchanged. For reasons of compatibility, databases created by previous versions will continue to use the database format of the older version, even for new objects.

column names, and user-friendly descriptions of these entities and properties. Tables and columns can be flagged as entities and attributes as desired.

For example, roles could be defined for users in different countries, using entity names, property names and descriptions appropriate to the language of the country, giving access to localised columns or views. The localisation of columns is facilitated by the Pyrrho-specific UPDATE clause for generated columns which can perform lookups or casts behind the scenes. These defined views or generated columns could even have specific data types targeting specific roles, since they impose no overhead unless they are explicitly used.

Roles that are granted usage of an object will not see any subsequent name changes applied in the parent role, but the role administrator can define new names. Stored procedures, view definitions, generation rules etc use the definer's permissions for execution.

Apart from object names, only the owner of an object can modify objects. This includes changes to object constraints and triggers, and inevitably such modifications can disrupt the use of the object by other roles, procedures etc. References in code in other roles can introduce restrictions on dropping of objects, but as usual, cascades override restrictions, and in Pyrrho, revoking privileges always causes a cascade. Granting select on a table must include at least one non-null column. Granting insert privileges for a role must include any non-null columns that do not have default values, and cannot include generated columns.

Metadata is an added feature in Pyrrho. Role administrators can modify object metadata as viewed from their role, and this is useful primarily for data output over HTTP.

### 3.7 Virtual Data Warehousing

Normally, data warehousing involves creating central data repositories (using extract-transform-load technologies) to enable analytic processing of a combined data set. There are several situations where this is undesirable, for example where the resulting data protection responsibility at the central repository is excessive, where the data is volatile and it becomes expensive to maintain all of the centrally-held data in real time, or where it is better to leave the data at its sources where the responsibility lies [Crowe et al. 2017]. With database technology, a View (if defined but not materialised) allows access to data defined in other places. The virtual data warehouse concept exploits this notion, and endeavours to avoid the central accumulation of data. Pyrrho uses HTTP to collect data from the remote DBMS using a simple REST interface[Fielding, 2000], and so the resulting technology here is called RESTView.

Thus, with RESTView, a Pyrrho database allows definition of views where the data is held on remote DBMS(s), and is accessible via SQL statements sent over HTTP with Json responses. Pyrrho itself provides such an HTTP service (see the next section) and the distribution includes suitable interface servers (RestIf, see sec 4.6) to provide such a service for remote MySQL and SqlServer DBMS.

In the use cases considered here, where a query  $Q$  references a RESTView  $V$ , we assume that (a) materialising  $V$  by Extract-transform-load is undesirable for some legal reason, and (b) we know nothing of the internal details of contributor databases. A single remote select statement defines each RESTView: the agreement with a contributor does not provide any complex protocols, so that for any given  $Q$ , we want at most one query to any contributor, compatible with the permissions granted to us by the contributor, namely grant select on the RESTView columns.

Crucially, though, for any given  $Q$ , we want to minimise the volume  $D$  of data transferred. We can consider how much data  $Q$  needs to compute its results, and we rewrite the query to keep  $D$  as low as possible. Obviously many such queries (such as the obvious select \* from  $V$ ) would need all of the data. At the other extreme, if  $Q$  only refers to local data (no RESTViews)  $D$  is always zero, so that all of this analysis is specific to the RESTView technology.

During query processing  $Q$  is transformed by replacing views by the tables they reference, filters are applied at the lowest level of the query (e.g. directly on a remote table), and the JSON representation of the result of selection is slightly enhanced to add the registers used to compute any remote aggregations.

There are two types of RESTView syntax (see section 7.2): corresponding to whether the view has one single contributor or multiple remote databases, as we will now see.

```
ViewDefinition = [ViewSpec] AS (RowSetSpec | GET [USING Table_id]) {Metadata} .
```

The RowSetSpec option here is the normal syntax for defining a view. The REST options both contain the GET keyword. The simplest kind of RESTView is defined as GET from a url defined in the Metadata. The types of the columns need to be specified in a slightly extended ViewSpec syntax (see sec 7.2). If

there are multiple remote databases, the GET USING table\_id option is available. The rows of this table describe the remote contributions: the last column supplies the metadata for the contributor including a url<sup>17</sup>, and data in the other columns (if any) is simply copied into the view. There are simple examples of this mechanism in the Pyrrho blog and website.

Depending on how the remote contributions are defined, RESTViews may be updatable, and may support insert and delete operations. In v7 of Pyrrho, a transaction can make alterations to the base tables of at most one physical database (no matter where it is hosted).

### 3.8 HTTP and ADO.NET services

Pyrrho's internal HTTP server is enabled using the +s +S -h -p flags of the PyrrhoSvr command line respectively to switch on the http and https service and optionally change the port from the defaults 8180 and 8133, to provide a hostname other than localhost, and to provide a RCP address other than ::1. You can supply your own server certificate for transport layer security and/or specify different ports.

Pyrrho requires Basic authentication as specified in RFC 7617, and this can provide a satisfactory level of security when used with transport-layer security (https).

The Authorization header should have a blank password element if the user is the owner of the database (or matches the server account of the database defines no users). If Pyrrho receives an unauthenticated request it will seek authentication, identifying its realm as "Pyrrhodb". Pyrrho does not store passwords.

One reason for using Pyrrho's HTTP service is to capture database input from, or obtain database output in, JSON, HTML, or CSV as alternatives to SQL (whose MIME type is text/plain). Selection of these formats is by use of the HTTP headers as usual (Accept for output format selection, Content-Type for input format selection). This mechanism is used by RESTView (see section 8.7.9 for extra details).

Metadata flags can be supplied for database objects for special HTML output formats such as charts and graphs. See section 7.2. Metadata flags can also request that SQL output use an alternative format.

As described in section 3.7 and below in this section, Pyrrho becomes a client of HTTP services (its own and others) in its implementation of RESTViews. See also the HttpFunction in sec 7.9.

The HTTP service (and the REST service it hosts) offers one important way to query and manipulate another database. Another way allows a script to be constructed for execution via ADO.NET on another database: the keyword to access this mechanism is currently FETCH and its syntax is given in section 7.1. Here is a trivial example that creates two database za and zb:

```
C:\PyrrhoDB70\Pyrrho>pyrrhocmd za
QL> create table a(b int,c char)
QL> insert into a values(78,'old record'),(52,'weeks in year')
2 records affected in za
QL> fetch (select * from a) for 'zb' first 'create table d(p char,q int)' do 'insert into d values($C,$B)'
QL> ^C
C:\PyrrhoDB70\Pyrrho>pyrrhocmd zb
QL> table d
```

p	q
old record	78
weeks in year	52

```
QL>
```

Note the use of capitals \$C, \$B. The statements in single string quotes are executed in the new database zb.

#### 3.8.1 A Transacted REST Service

Pyrrho's HTTP server supports RFC 7232 [Fielding, 2014] and uses it to offer a transacted SQL service over HTTP/1.1.

The URL used has the form `http://hostname:port/database/role[/table]`, the default verb is POST and the Content-Type of the request is text/plain, and the body of the request is a semicolon-separated sequence of SQL statements.

The transacted service supports the verbs HEAD and POST, and supplies ETag information in its responses. (For the other verbs, see section 3.8.2.)

<sup>17</sup> In v7, the declared type of this column must be METADATA.

The RFC 7232 header If-Unmodified-Since enables the session to be started conditionally, giving a date in the recent past (e.g. the start of another transaction in progress). The header If-Match allows the service to continue if a given list of ETags is still valid, so that typically ETags accumulate during a transaction.

Etags are normally opaque double-quoted strings, but Pyrrho's ETags have the form of a double-quoted sequence of comma-separated big integers

ETag = "ttt,ddd,ppp{,ddd,ppp}" {,ETag}

where ttt is a table uid, ddd a row uid (or -1 to indicate all rows), and ppp the latest data position for the table or row.<sup>18</sup>

Both If-Unmodified-Since and If-Match can be used with a HEAD request, respectively to synchronise with an ongoing transaction on another database, or to validate a read-only transaction.

Selects do POST in the read part of the transaction and HEAD at the end. It is the other way round for INSERT, UPDATE and DELETE. The rules for validation and ETag upating are shown in the table below:

Statement	When	Verb	If-Unmodified-Since	If-Match	Returned ETag recorded
Select	Read	POST	If * Validate		Rows read*
Select	Write	HEAD		Validate	
Insert	Read	HEAD	If * Validate		
Insert	Write	POST			Rows inserted
Update	Read	HEAD	If * Validate		Rows to be updated
Update	Write	POST		Validate	Rows updated
Delete	Read	HEAD	If * Validate		Rows to be deleted
Delete	Write	POST		Validate	

Validation failure for If-Unmodified-Since is 40084, for If-Match 40082.

\* If we read the whole table, then any change will be a conflict, so we record -1, lastData; otherwise we override a previous -1,lastData entry with specific information.

The Pyrrho server will be a client of such a service when explicit transactions are used with RESTViews, and the ETAG metadata flag is specified in the View definition, Pyrrho automatically generates the required headers for remote databases, and defers REST requests to the commit point of the explicit transaction. It is strongly recommended that only one remote database is modified in any such transaction. The conditions 400084 and 400082 are raised on precondition failure for If-Modified-Since and If-Match.

The If-Match mechanism is based on a string cookie that is returned with HTTP results. RFC7232 returns this in the response header. RFC7232 states that a wild-card ETag is the string "\*"; there is no point in using this for validation. Otherwise the ETag can be used in an If-Match header in subsequent HTTP calls, to verify that the data that was returned has not changed in the meantime. This mechanism has obvious limitations from the point of view of transaction control, where HTTP results take the form of data tables. Importantly, it only applies to a single HTTP response, and not to a transaction history. Its guarantees can apply to a small number of base table rows, or a single table, or a single database (the full database test applies if RFC 7232's Weak W/ flag is specified in If-Match). This has been implemented for Pyrrho for REST Views that have the ETAG metadata flag, and Pyrrho will raise the "ETag validation fails" error for an If-Match ETag that no longer matches the database state.<sup>19</sup>

The If-Unmodified-Since header has a standard date format (RFC 7231) that does not support time intervals less than one second. Pyrrho will allow the header to include a fractional milliseconds part (of form .ddd) if the metadata flag MILLI is provided with the URL.

Clients of PyrrhoDBMS can use a RESTful interface provided by the PyrrhoConnect class as described in section 8.7.8 and 8.3.4.

The service-specific request headers for this service are as follows:

Header name	Syntax	Comments
Accept	text/plain   text/html   text/csv   application/json	Mime strings as specified in RFC2616
Authorization	user	As specified in GRANT

<sup>18</sup> The above format has been adopted in PyrrhoV7 for compatibility with RFC 7232.

<sup>19</sup> The mechanism has some similarities to the RVV mechanism proposed by Laiho and Laux (2010).

If-Match	"*"   "t,d,c{,d,c}"{"t,d,c{,d,c}"}	*, or ETags previously returned by the server cf. RFC 7232 and below
If-Unmodified-Since	HTTP-date	The transaction start time as in RFC 7231 sec 7.1.1.1
Url	proto://host[:port]/dbname/role[Details]	For Details see below

The service-specific response headers are as follows (for Status 200):

Header name	Syntax	Comments
Description	Unicode string	If specified in table metadata
Classification	/[{g{,g}}][[r{,r}]]	Level A-D, group and reference are optional sequences of strings separated by commas, groups enclosed in {}, references enclosed in [].
LastData	Unsigned integer	Highwatermark in the log for tables included in results. The LAST_DATA property is available in Pyrrho's SQL
ETag	"t,d,c{,d,c}"{"t,d,c{,d,c}"}	The row-version validator (CHECK) for the first row returned. Separated by semicolons: table uid, record uid (or -1 for all rows), log position, separated by commas. See sec 5.2.3.

### 3.8.2 A URL-based HTTP service.

This service supports GET, PUT, POST and DELETE (for HEAD see below), where the url has the form `http://hostname:port/database/roleDetails`

where Details is as follows:

Details: {'/'Selector}{'/'Processing}{'?'Metadata]

Selector matches<sup>20</sup>:

```
[table ]Table_id
[procedure ]Procedure_id ['( Parameters ')']
[where ]Column_id('='|'>'|'<'|'<='|'>='|'<=>')string
[select ]Column_id{,Column_id}
[key ]string
```

Appending another selector is used to restrict a list of data to match a given primary key value or named column values, or to navigate to another list by following a foreign key, or supply the current result as the parameters of a named procedure, function, or method (see the examples below).

Parameters matches a comma separated list of constant values.

Processing matches:

```
distinct [Column_id{, Column_id}]
ascending Column_id{, Column_id}
descending Column_id{, Column_id}
skip Int_string
count Int_string
```

For Metadata, see section 7.2.

The Http/https Accept and Content-Type headers control the formatting used. At present the supported formats are JSON (application/json), HTML (text/html, only for responses) and SQL (text/plain). The Pyrrho distribution includes a REST client which makes it easier to use PUT, POST and DELETE. A URL can be used to GET a single item, a list of rows or single items, PUT an update to a list of items, POST one or more new rows for a table, or DELETE a list of rows. Thus GET and POST are very different operations: for example, in this service POST does not even return data. All tables referenced by selectors must have primary keys. See section 4.5.

For the key selector, the parser knows the datatypes of the table's columns so it is quite flexible about the format, and in particular single quotes around a single string value are optional. If the selector has several components, they should be separated by commas.

<sup>20</sup> The optional keywords here are less restrictive than might appear: In this syntax views and tables can be used interchangeably, so that the keyword **table** if present may be followed by a view (unlike SQL). Similarly, the keyword **procedure** if present may be followed by a function call.

Some navigation is possible with this URL model. For example for a database D with role Sales, GET<sup>21</sup> /D/Sales/Orders/1234 returns a single row from the Orders table, GET /D/Sales/Orders/Total>50.0/OrderDate/distinct returns a list of dates when large orders were placed, GET /D/Sales/Orders/OrderDate,Total returns just the dates and totals, GET /D/Sales/Orders/1234/of OrderItem returns a list of rows from the OrderItem table, and GET /D/Sales/Orders/1234/CUST/Customer/NAME returns the name of the customer who placed order 1234. The response will contain a list of rows: if HTML has been requested it will display as a table (or a chart if specified by the Metadata flags, sec 7.2). Using HTML will also localise the output for dates etc for the client.

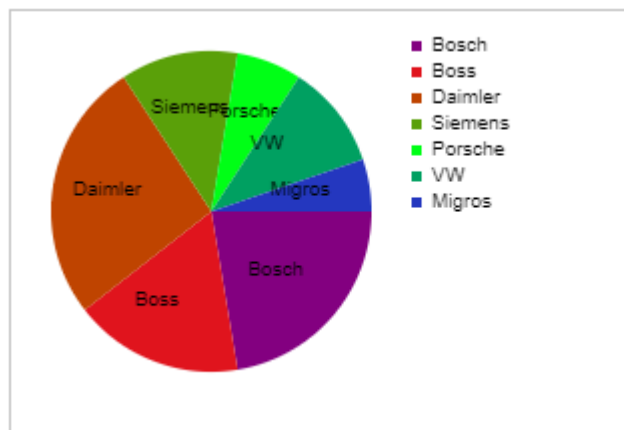
For example, with the database E created by

```
create table sales (cust char(12) primary key, custSales numeric(8,2));
insert into sales values ('Bosch', 17000.00), ('Boss', 13000.00), ('Daimler', 20000.00);
[insert into sales values ('Siemens', 9000.00), ('Porsche', 5000.00), ('VW', 8000.00),
('Migros', 4000.00);]
create role E;
grant E to "usermachine\username";
```

if the browser is asked for

[http://localhost:8180/E/E/SALES/?PIE\(CUST,CUSTSALES\)LEGEND](http://localhost:8180/E/E/SALES/?PIE(CUST,CUSTSALES)LEGEND)

it will display the output shown.



PUT <http://D/Sales/Orders/1234/DeliveryDate> with text/plain content of ((date'2011-07-20')) will update the DeliveryDate cell in a row of the Orders table. PUT content consists of a list of rows, whose type must match the rowset returned by the URL. If the list has more than one row, commas can be used as separators. JSON format is also supported.

POST <http://D/Sales/Orders> will create one or more new rows in the Orders table. In Pyrrho an integer primary key can be left unspecified. In SQL (text/plain) format, column names can be included in the row format, e.g. (NAME:'Fred','DoB':date'2007-10-22'): if no names are provided, all columns are expected. Remember that the REST service is case-sensitive for database object names. JSON can be used with the obvious format. A mime type of text/csv has been added to facilitate import from Excel spreadsheets.

Pyrrho supplies ETag information with responses, and one or more of these can be submitted in an If-Match header for conditional HTTP processing. Using this approach ACID behaviour can be guaranteed for a sequence of HTTP requests where all except the last are GETs. However, it is generally better to send a transacted sequence of SQL statements using POST, as described in section 3.8.1 above.

See also sections 4.5 and 4.6. The implementation of GET in the server is also used for the Versioned library (sec 6.4).

Pyrrho will become a client of this service in the implementation of a RESTView that specifies the URL metadata flag with its url, and then PUT, POST, and DELETE operations are sent immediately to the remote server even if an explicit transaction is in progress.

With a database yc [after Francis,2023] created by

```
[CREATE (p1:Person{name:'Aretha'})-[:owns]->(a1:Account{isBlocked:false}),
(p2:Person{name:'Jay'})-[:owns]->(a2:Account{isBlocked:false}),
(p3:Person{name:'Mike'})-[:owns]->(a3:Account{isBlocked:true}),
(p4:Person{name:'Scott'})-[:owns]->(a4:Account{isBlocked:false}),
(a1)-[:Transfer{amount:2000000}]->(a2)-[:Transfer{amount:2500000}]->(a3),
(a3)-[:Transfer{amount:3000000}]->(a4)-[:Transfer{amount:3500000}]->(a1),
(p2)-[:Member]->(c1:YachtClub{name:'Ankh-Morpork Yacht Club',address:'Cable Street'}),
```

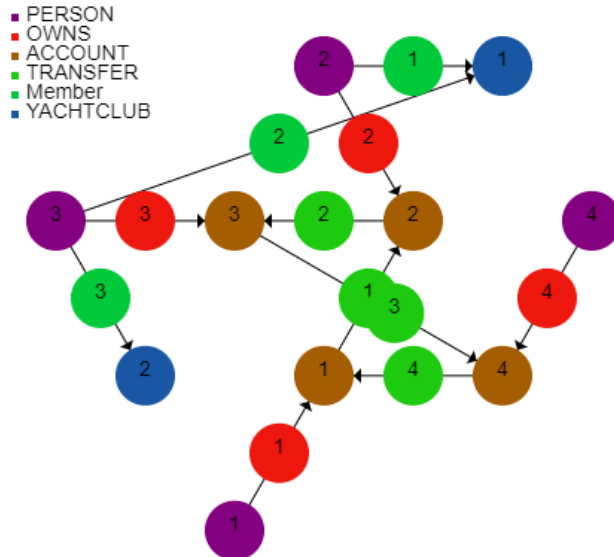
<sup>21</sup> Here the GET URL will be prefixed by <http://host:port> as usual.

```
(c1)-[: "Member"]-(p3)-[: "Member"]->(c2:YachtClub {name:'Emerald City Yacht
Club',address:'Yellow Brick Rd'})]
create role YC
grant YC to "usermachine\username"
```

if the browser is asked for

<http://localhost:8180/yc/YC/PERSON/ID=1?NODE>

it will display



### 3.9 Localisation and Collations

Pyrrho's database files are intended to be locale-neutral: they use universal time and UTF-8 encoding with standard case-sensitive collation. Localisation and regional settings are applied in the API library (PyrrhoLink.dll) and by default use the regional settings of the client (see chapter 4). This design makes it easier for databases to be accessed from or copied to different locales, and is consistent with the locale-neutral SQL language.

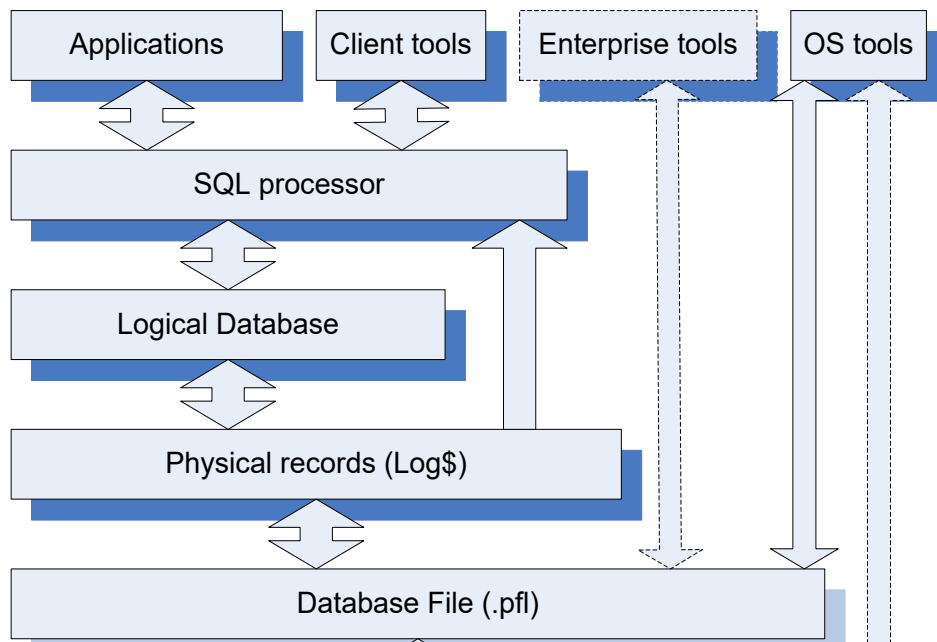
Pyrrho also supports most localisation facilities available in the SQL standard. For example, it uses the character set names as specified in SQL2023. Specifying a character set restricts the values that can be used, not the format of those values. By default, the UCS character set is used. By default, the UNICODE collation is used, and all collation names supported by the .NET framework are supported by Pyrrho. CHAR uses standard culture-independent Unicode. NCHAR is no longer supported and is silently converted to CHAR. UCS\_BINARY is supported.

The SQL2023 standard specifies a locale-neutral interface language to the server, notably for dates and times.

In addition, views and updatable generated columns provide opportunities for localisation, which can be targeted by defining roles for specific locales.

### 3.10 Pyrrho DBMS architecture

The structure of the Pyrrho DBMS is shown in the drawing below (the .pfl extension is no longer used).





## 4. Pyrrho client utilities

The main client utility at present is a traditional command-line interpreter PyrrhoCmd. There is a Windows client called PyrrhoSQL. As with all Pyrrho clients, the PyrrhoLink.dll assembly is also required. We discuss these first. The distribution also contains a REST client and a transaction profiling utility.

### 4.1 The Pyrrho Connection library

PyrrhoLink.dll (or the Java package org.pyrrhodb.\*) is used by any application that wishes to use the Pyrrho DBMS. This library includes support for client applications. The simplest possible approach is simply to place PyrrhoLink.dll in the same folder as the application that is using it.

In Chapter 6 we will see that PyrrhoLink.dll is also needed to be at hand when compiling applications.

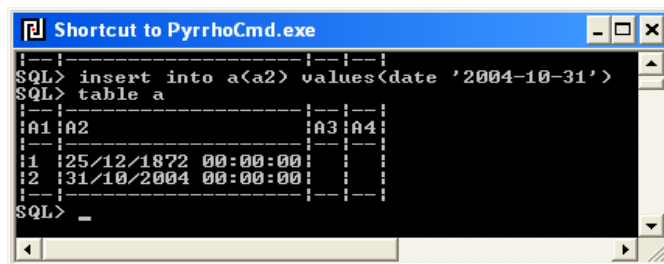
Since version 5.4 thread-safety is enforced in client-side programming. Connections can be shared among threads. But there can be at most one transaction or command active in a connection, and transactions, commands and readers cannot be shared between threads.

#### 4.1.2 Localisation

In the current version, PyrrhoLink supplies error messages in English. Localisation to other languages was provided in previous editions and the mechanism to do this is still available in the code.

Locale-independent data from the database, such as dates and times, can be rendered by PyrrhoLink.dll according to the regional settings on the client machine. The database may be in a different country or timezone from the client.

However, SQL2023 itself is invariant (details of data formats are given in section 7). Thus the following behaviour is correct for a client machine in the UK:



```

Shortcut to PyrrhoCmd.exe
SQL> insert into a(a2) values(date '2004-10-31')
SQL> table a
+-----+-----+-----+-----+
|A1|A2|A3|A4|
+-----+-----+-----+-----+
|1|25/12/1872 00:00:00| | |
|2|31/10/2004 00:00:00| | |
+-----+-----+-----+-----+
SQL>
  
```

### 4.2 Installing the client utilities

The distribution currently contains PyrrhoCmd.exe, and PyrrhoSQL.exe, and the PyrrhoLink module PyrrhoLink.dll. These can be placed anywhere in the file system so long as the dll is in the same folder as the executable.

Since the client executables are so small (currently 140KB including the DLL) it is generally easier to copy them where they are required rather than using load-paths or registry entries.

It is usually convenient for database administration to install them on the server (in addition to client machines if any), but the client utility do not have to be on the server machine. If the server is not on 127.0.0.1 the `-h:` command line option can be used to specify a different host (e.g. `-h:fred`, or `-h:192.168.1.3`).

### 4.3 PyrrhoCmd

PyrrhoCmd is a console application for simple interaction with the Pyrrho server. Basically it allows SQL statements to be issued at the command prompt and displays the results of SELECT statements in a simple form. Insert, update, and delete operations will generally cause a response indicating the number of rows affected<sup>22</sup>.

<sup>22</sup> For operations involving table constraints that specify cascade or other side effects, the response will be simply OK.

It has some additional features. It supports upload and download of blobs (binary large objects) through use of the escape character ~. It also supports the sort of multi-database connection described in section 2.7. See section 4.1 for locale issues.

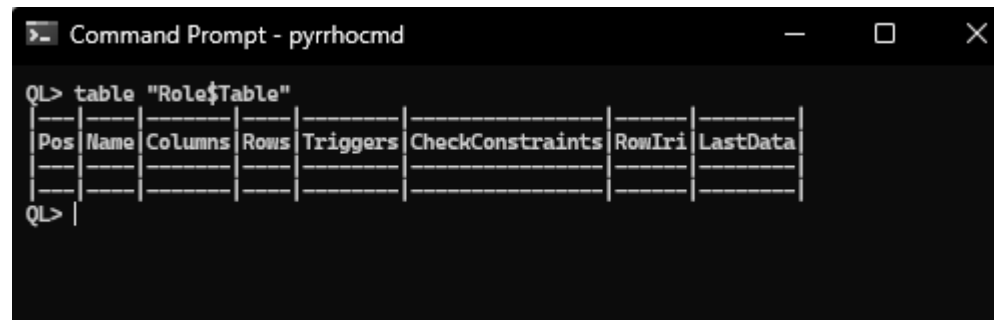
### 4.3.1 Checking it works

On the same machine as the server, open a command window and use `cd` to navigate to the same folder as the client executable.

**PyrrhoCmd**

```
QL> table "Role$Table"
```

In SQL2023 `table id` is the same as `select * from id` for base tables and system tables.



PyrrhoCmd will respond with the list of ables in the current database. The default database Temp is created if necessary by the command processor. To create or use another database, specify it on the command line. The above response from the server merely gives information about the tables in the database that are accessible from the current role (you are the database owner in this case, but it contains no tables). If you look in the folder: you will see a file called Temp (it was not there before).

You can use control-C, or close the window, to exit from PyrrhoCmd.

Note that database names in Pyrrho are case sensitive. (Windows is relatively careless.)

### 4.3.2 Accessing a server on another machine

If the client is running on a different machine from the server, you will need to specify the host in the command line, as in:

**PyrrhoCmd -h:address**

Normally, PyrrhoCmd is used to connect to a particular database, specified as an argument in the command line. If no argument is supplied, then as indicated above, the Temp database is used.

If the database is new, and the client account is not the same as the server account, a default role is created with the same name as the database, the client account is granted use of it, and becomes the database owner. If the client account is the same as the server account, no role or user information is initially placed in the database, allowing roles and users to be defined later. This makes it easier to create databases for students. In both cases, the first role to be created becomes the default role for the database, and the first user to be granted this role becomes the database owner, and these can access any database objects created up to that point.

### 4.3.3 Connecting to databases on the server

For example, if there is a database called Book, use

**PyrrhoCmd Book**

to connect to it. Note that case is significant in database names (since these are parts of actual file names). If more than one database name is given on the command line, a connection is established that opens a list of databases in the order given. See section 2.7.

### 4.3.4 The QL> prompt

PyrrhoCmd is normally used interactively. At the QL> prompt you can give a single SQL statement. There is no need to add a semicolon at the end. There is no maximum line length either, so if the command wraps around in PyrrhoCmd's window this is okay.

```
QL> set role ranking
```

Be careful not to use the return key in the middle of an SQL statement as the end of line is interpreted by PyrrhoCmd as EOF for the SQL statement. If you want to use multiline SQL statements, see section 4.3.5.

At the SQL command prompt, instead of giving an SQL statement, you can specify a command file using *@filename*. Command files are ordinary text files containing an SQL statement on each line.

### 4.3.5 Multiline SQL statements

If wraparound annoys you, then you can enclose multi-line SQL statements in [ ] . [ and ] must then enclose the input, i.e. be the first and last non-blank characters in the input.

```
QL> [ create table directors ( id int primary key,
> surname char,
> firstname char, pic blob ) ]
```

Note that continuation lines are prompted for with > . It is okay to enclose a one-line statement in [ ] .

Note that Pyrrho creates variable length data fields if the length information is missing, as here. This seems strange at first: a field defined as CHAR is actually a string.

### 4.3.6 Adding data and blobs to a table

Binary data is actually stored inside the database table, and in SQL such data is inserted using hex encoding. But PyrrhoCmd supports a special syntax that uses a filename as a value:

```
QL> [ insert into directors (id, surname, firstname) values (1,
'Spielberg', 'Steven', ~spielberg.gif) ]
```

The above example shows how PyrrhoCmd allows the syntax *~source* as an alternative to the SQL2023 binary large object syntax X'474946...' . PyrrhoCmd searches for the file in the current folder, and embeds the data into the SQL statement before the statement is sent to the server.

As this behaviour may not be what users expect, the first time Pyrrho uploads or downloads a blob, a message is written to the console, e.g.:

Note: the contents of *source* is being copied as a blob to the server  
*source* can be enclosed in single or double quotes, and may be a URL, i.e. *~source* can be *~"http://something"*.

A textfile containing rows for a table can similarly be added using a command such as

```
insert into directors values ~rowsfile
```

Simple data can be provided in a csv or similar file. The first line containing column headings and exposed spaces in the file are ignored. Data items in the given file are separated by exposed commas or tabs. Rows are parenthesized groups (optionally separated by commas), or provided without parentheses but separated by exposed semicolons or newlines. Characters such as commas etc are not considered to be separators if they are within a quoted string or a structure enclosed in braces, parentheses, brackets, or pointy brackets.

### 4.3.7 Retrieving data and blobs from the server

Data is retrieved from the database using TABLE or SELECT statements, as indicated in 4.2.1.

If data returned from the server includes blobs, by default PyrrhoCmd puts these into files with new names of form *Blobnn* . Again PyrrhoCmd will alert the user to this process on the first occasion (unless *-s* flag has been set, see section 4.3.8, or the above message has been shown). To prevent downloads, use the *-b* flag, see section 4.3.8.

```

D:\mkc\NRTech\PyrrhoCmd\bin\Debug\PyrrhoCmd.exe
SQL> select * from directors
Note: blob(s) from database copied to file(s)
-----
ID|SURNAME|FIRSTNAME|PIC
-----
1 |Spielberg|Steven|@Blob1
-----
SQL>

```

Blobs retrieved to the client side by this method end up in PyrrhoCmd's working directory (which is usually different from the database folder). To view them it is usually necessary to change the file extension, e.g. to Blob1.gif.

For ways to retrieve data and blobs using an application, see Chapter 6.

### 4.3.8 Command Line synopsis

When starting up PyrrhoCmd, the following command line arguments are supported:

<code>database ...</code>	One or more database names on the server. The default is ab.
<code>-h:hostname</code>	Contact a server on another machine. The default is 127.0.0.1
<code>-p:nnnn</code>	Contact the server listening on this port number. The default is 5433
<code>-s</code>	Silent: suppress warnings about uploads and downloads
<code>-e:command</code>	Use the given command instead of taking console input. (Then the QL> prompt is not used.)
<code>-f:file</code>	Take SQL statements from the given file instead of from the console.
<code>-c:locale</code>	Specify a language for the user interface, overriding .NET defaults. Localised versions of the error messages will be used if available. See section 4.1.2.
<code>-b</code>	No downloads of Blobs
<code>-U</code>	Set CaseSensitivity to true. The default is false, for standard SQL identifier syntax (double-quotes typically enclose identifiers such as columns that use upper and lower case, and unquoted identifiers are changed to upper case). If set to true, double quotes can be used instead of single quotes, and identifiers are not converted to upper case.
<code>-v</code>	Show version and readCheck information for each row of data
<code>-?</code>	Show this information and exit.

Whether the command prompt (console) window is able to display the localised output will depend on system installation details that are outside Pyrrho's control. Localisation is more effective with Windows Forms or Web Forms applications.

### 4.3.9 Transactions and PyrrhoCmd

Transactions in Pyrrho are mandatory, and are always serializable. By default, each command is committed immediately unless an error occurs. Alternatively, you can start an explicit transaction at the QL> prompt with BEGIN TRANSACTION or START TRANSACTION:

```
QL> begin transaction
```

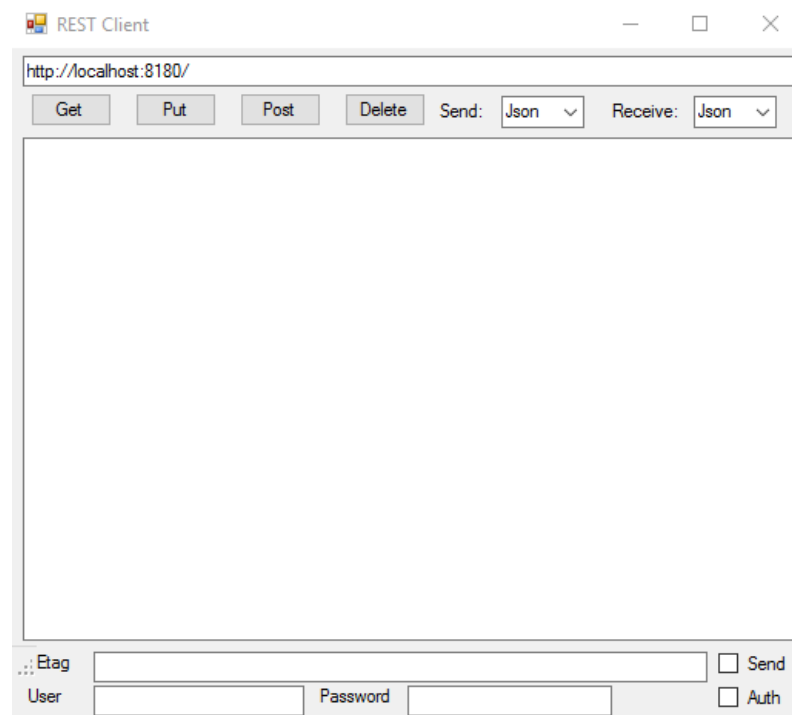
Then the command line prompt changes to QL-T> to remind you that a transaction is in progress. This will continue until you issue a **rollback** or **commit** command at the SQL-T> prompt. If an error is reported by the database engine during an explicit transaction, you may get an additional message saying that the transaction has been rolled back followed by a normal QL> prompt, or another SQL-T> prompt as an invitation to try to continue the transaction by means of another SQL command.

This continue behaviour is similar to the support offered by SQL's CONTINUE handler. The PyrrhoCmd client examines the TRANSACTION\_ACTIVE diagnostic after an exception to see if the transaction can continue.

With explicit transactions, there appears to be a difference in the feedback provided by the server on completion of an insert, delete or update statement: the number of rows reported to affected accumulates during the transaction. In both cases the report is of the number of changes to be committed. The difference is that with implicit transactions, the commit happens immediately (implicitly), resetting the count of rows affected; while in the explicit transaction the running total of rows affected continues to grow until the explicit COMMIT command occurs, the changes are abandoned with ROLLBACK, or an error occurs that means the transaction cannot be committed.

## 4.4 RESTClient

This Windows Forms program is useful for using the REST interface described in section 3.7. It is not Pyrrho-specific and uses Windows authentication to the server:



It offers a choice of send and receive formats (SQL and HTML). It is important to remember the role must be specified in addition to the database name, and URLs are case-sensitive.

The drop-down lists offer alternative formats for request and response: Json, SQL, and String,

If an ETag is returned by the service, it is displayed. The Send checkbox is used to send the contents of the ETag box as an If-Match condition with the HTTP request.

The Auth checkbox is used to supply the given User as credentials to the service. Passwords are not stored but are local to the HTTP interaction.

## 5. Database design and creation

This chapter assumes that the reader is familiar with the general principles of relational databases including normal form and integrity constraints. For simplicity, we will document the use of the command line utility to carry out the steps discussed in this chapter.

Many activities could of course be automated using command scripts or application programs. For the latter, see Chapter 6.

### 5.1 Creating a Database

As mentioned in the last section, by default Pyrrho will create a database if necessary. To create a database, simply issue the command

**PyrrhoCmd *dbname***

The file *dbname.osp* will be created in the database folder, and owned by the server account. The database will not be completely empty: it will have two initial records. The first of these will be a User record identifying the client account as the owner of the database. The second will be a default Role (with the same name as the database) which permits all actions on the database. The User will be the client's login ID. These two records specify the database owner and the schema role for the database.

The remainder of this subsection can be skipped on a first reading.

For example, suppose the Pyrrho service account on VANCOUVER is PYR\_USR, and user LONDON\Fred issues the command

```
PyrrhoCmd -h:VANCOUVER MyLibrary
```

This command assumes that Fred has access to the client program, and to port 5433 on VANCOUVER where PyrrhoSvr is already running. If database MyLibrary already exists on host VANCOUVER, and LONDON\Fred is allowed to access it, the command line utility will start up on the client computer with a connection to this database. If MyLibrary does not exist on VANCOUVER, the PyrrhoSvr will create a new database file MyLibrary.pfl in the database folder, which will be owned by PYR\_USR. MyLibrary.pfl will have an initial User record for user 'LONDON\Fred' of type owner, and a Role called 'MyLibrary'. In both cases, the PyrrhoCmd utility will now give the command prompt

QL>

for SQL commands such as creation of the first few objects in this new database.

It is entirely reasonable for administrators to wish to limit the ability to create databases in the database folder. A better solution on a corporate network will be for databases to be initially created by their owners on their local machines but using their network login, and then copied by an administrator to the database folder on the server host. On the server host, the database folder should have permissions such that the server account cannot create new files. This approach would have the added advantage that the database file would actually continue to be owned by the client user.

### 5.2 Creating database objects

When using CREATE TABLE and other SQL statements at the command prompt, bear the following points in mind:

- SQL2023 syntax is somewhat different from many legacy systems. In particular:
- Identifiers are not case-sensitive unless they are enclosed in double quotes
- Double-quoted identifiers can be used to avoid confusion with reserved words and for identifiers that contain special characters
- By default, variable length data types can be used, e.g. CHAR instead of CHAR(16). If size and precision are specified, values are truncated. Precision specification for numeric types, if specified, is in decimal digits
- Single quotes are still used for string literals.

- Date, time, timestamp, and interval literals have a fixed syntax (e.g. DATE '2005-07-20') and the formats are not locale-sensitive.

In the current version the SQL2023 Timezone feature is not implemented (since it impedes moving a database between timezones), so time and timestamp are displayed for the local time zone on the computer in question, but are stored in the database in universal time.

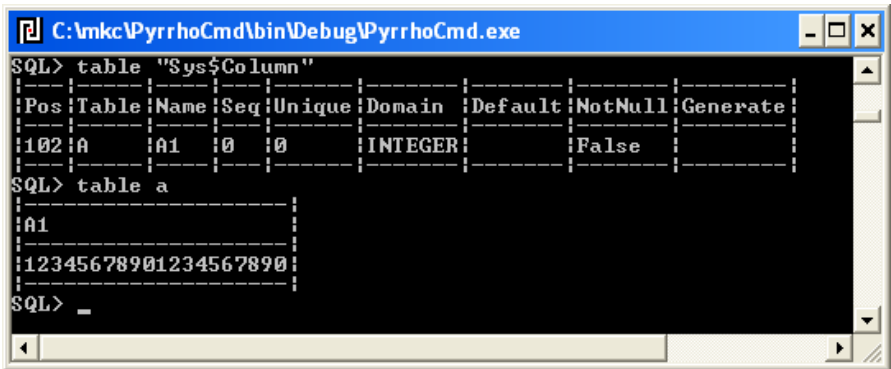
For example the SQL statement

`Insert into Winner ("YEAR",Rep) values (2005,'Fred')`

will create a new record in an already-existing table WINNER(YEAR,REP) of form

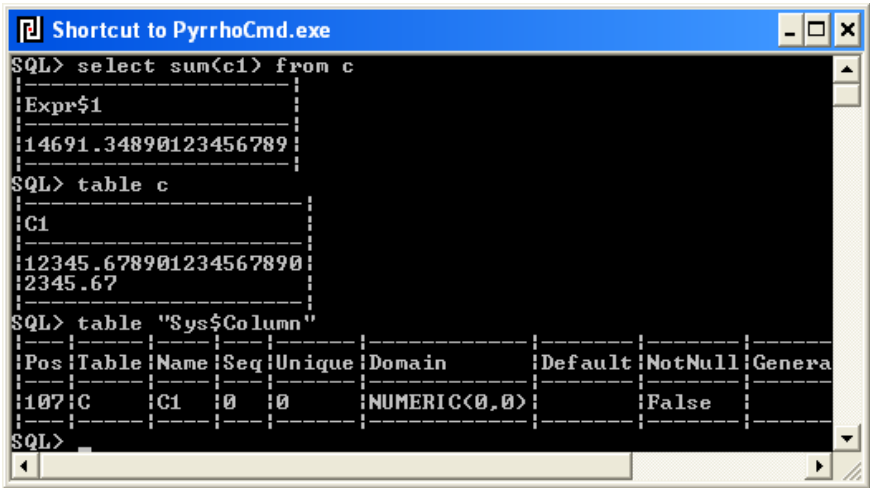
YEAR	REP
2005	Fred

The double quotes are needed since YEAR is a reserved word in SQL2023. The single quotes are needed since Fred would otherwise be interpreted as an identifier (e.g. a column name). These requirements come from SQL2023.



```
C:\vnc\PyrrhoCmd\bin\Debug\PyrrhoCmd.exe
SQL> table "Sys$Column"
+-----+-----+-----+-----+-----+-----+-----+-----+
|Pos|Table|Name|Seq|Unique|Domain|Default|NotNull|Generate|
+-----+-----+-----+-----+-----+-----+-----+-----+
|102|A|A1|0|0|INTEGER|False|False|
+-----+-----+-----+-----+-----+-----+-----+-----+
SQL> table a
+-----+
|A1|
+-----+
|12345678901234567890|
+-----+
SQL> _
```

The illustration above shows an integer value (larger than “long”) in an “ordinary” integer field. The following example shows precision greater than double precision:



```
Shortcut to PyrrhoCmd.exe
SQL> select sum(c1) from c
+-----+
|Expr$1|
+-----+
|14691.34890123456789|
+-----+
SQL> table c
+-----+
|C1|
+-----+
|12345.678901234567890|
|2345.67|
+-----+
SQL> table "Sys$Column"
+-----+-----+-----+-----+-----+-----+-----+-----+
|Pos|Table|Name|Seq|Unique|Domain|Default|NotNull|Genera|
+-----+-----+-----+-----+-----+-----+-----+-----+
|107|C|C1|0|0|NUMERIC<0,0>|False|False|
+-----+-----+-----+-----+-----+-----+-----+-----+
SQL> _
```

### 5.2.1 Pyrrho’s data type system

This now includes the GQL predefined data types, and constructed value types of array, list, multiset set and record types are supported<sup>23</sup>.

Pyrrho’s type system also includes modified versions of SQL’s data types, as follows: (a) char and char(0) indicate unbounded strings, i.e. the GQL string type; int, int(0), integer and integer(0) indicate 2048-bit integers (see example in the section above); and real has a mantissa of 2048 bits by default; (b) GQL’s

<sup>23</sup> From May 2025 and v7.010 list format is used for rowsets: this is a breaking change, as array format now consists of pairs (long,data).

size specific types `int8` etc are supported; (c) persisted data is not changed by subsequent changes to column datatypes as long as it is coercible to the new type.

To explain the last point, suppose a table has a column of type `numeric`, and contains values with (say) up to 5 significant digits. Suppose now the table is altered so that the column is `numeric(3)`. At this point all new values will be truncated on insertion, and all existing values will be truncated on retrieval, so that the table appears to contain values with a maximum of 3 significant digits. Now suppose the data type is changed back to `numeric`. Now the old data with 5 significant digits is visible once more, but of course the data inserted when the data type was `numeric(3)` will only have 3 significant digits. A case could be made that this behaviour is incorrect. But it helps to avoid accidental loss of data.

User-defined types may specify supertypes (confusingly using the `UNDER` clause in type definitions). The supertype will implement all columns and indexes required to support its subtypes, and columns are inherited from supertypes. Inserted rows are added to the base-tables that define their columns (thus, the target type and its supertypes), but the log shows a single record with all fields. Update and delete behave similarly.

For any `UDType t`, `t.rowType` includes all columns of all supertypes of `t`, but need not include all columns of subtypes. If a subtype `s` of `t` contains a `tableRow r`, `r` is also a `tablerow` of `t`. If this `tableRow` defines a GQL node, it will also belong to every node type in its key label set. This means that (a) `tableRows` can belong to more than one table, (b) if similarly named columns exist in two direct or indirect subtypes one of these columns must be equal to or a subtype of the other.

Edge connections can be handled if desired using SQL-style foreign keys, but the default mechanism uses system references based on the defining positions of records. A column `C` in a table `T` declared to have the pseudo-data type `(TO|FROM|WITH) NodeType` declares `T` to be a node type (if it is not already) and defines a directed or undirected simple edge type, also called `C`, connecting `T` and the referenced node type<sup>24</sup>. The deletion of a node causes a cascade of deletions of edges connected to it. Drop of a node or edge type `T` will not delete nodes or edges of a supertype `S` of `T`.

From version 7.04 (and from 7.03 in the case of typed graph types) there are built-in base tables directly implementing user-defined types, and these can be targets of selection, insertion etc. The specified target of insert etc determines the type of row that is inserted, while selecting from a type will return all rows that are of that type or its subtypes. The most specific type of a row is provided by a standard SQL function `SPECIFICTYPE()`, and the defining position of a node or edge is a pseudocolumn `POSITION`<sup>25</sup>. We adopt the SQL convention that if `N` is “of” a type `T`, as in the above paragraph, this means that `N`’s most specific type is `T` or a subtype of `T` (SQL allows `OF ONLY` to limit to the specific type).

## 5.2.2 Indexes, Identity etc

Indexes are not database objects in standard SQL. Pyrrho supports primary and unique keys as in SQL, but users need to understand that such constraints are inherited by subtypes<sup>26</sup>. Integrity, uniqueness and referential constraints imply their use within the database engine, and behind the scenes Pyrrho uses indexes built in this way for automatic query optimisation.<sup>27</sup>

Pyrrho extends the notion of referential constraint to allow adapter functions: this behaviour is helpful when working with semantic inference systems. To illustrate the concept of an adapter function, consider this example:

```
foreign key(rdate, regionid) references t using (extract (year from rdate), regionid)
```

If the `USING` value is non-null, this should be a key in the referenced table. When this extended behaviour is used, the value of the computed foreign key is recorded along with the transaction.

<sup>24</sup> Values in the new column `C` have the predefined data type `POSITION`. Edge types that have properties and multiple connections are best defined separately with the help of the `EDGETYPE` metadata construct. Both of these mechanisms are compatible with GQL’s insert and match statements and have automatic system indexes.

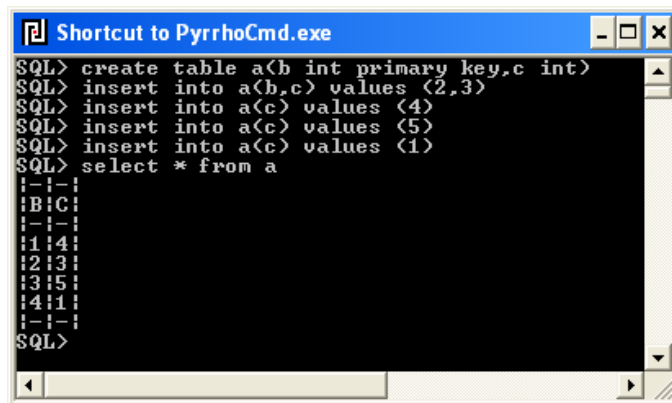
<sup>25</sup> The `RVV.version` is now given by the `VERSION` pseudo-column.

<sup>26</sup> For example, if `Employee` is a subtype of `Person`, and `Person` has a primary key, the primary key for `Person` will also be the primary key for `Employee`, and so each person has at most one `Employee` record.

<sup>27</sup> A syntax for `CREATE INDEX` has however been added to Pyrrho so support the MongoDB service.



Pyrrho does not have “identity”, “autonumber”, “sequence”, or “generator” features found in other databases. Instead, it has the following automatic feature, which it claims is better. If insert is proposed and a key component is missing, Pyrrho will find a suitable value: this behaviour is similar to POST in REST-based systems.



```

Shortcut to PyrrhoCmd.exe
SQL> create table a(b int primary key,c int)
SQL> insert into a(b,c) values (2,3)
SQL> insert into a(c) values (4)
SQL> insert into a(c) values (5)
SQL> insert into a(c) values (1)
SQL> select * from a
+---+
|B|C|
+---+
|1|4|
|2|3|
|3|5|
|4|1|
+---+
SQL>

```

If multiple rows are used in a single INSERT, as in “insert into a(c) values (4),(5),(1)”, the actual order of insertion will not necessarily seem to be the obvious one.

### 5.2.3 Row versions

Pyrrho supplies pseudocolumns in all base tables for row versioning purposes and security. There are four of these: SECURITY, CHECK, and its components POSITION and VERSIONING. CHECK is a Rvv cookie, and POSITION and VERSIONING are integers.

As the name implies, SECURITY is reserved for use by the database owner as security administrator, and has a special type called Level. It is assignable to a value of type Level For further information see section 3.4.2..

The information in CHECK includes the defining position of the table(s), defining position of the row(s) and current offset of the row version<sup>28</sup>. When retrieved it refers to the version valid at the start of the transaction, but it can be used at any time subsequently (inside or outside the transaction) to see if the row has been updated by this or any other transaction (this is the only violation of transaction isolation in Pyrrho).

The normal use of this data in application programming is to check that information previously read by the application is still valid<sup>29</sup>. For example, if the application reads a row of data including the VERSIONING pseudocolumn and saves the version in a local variable called (say) rvv, a subsequent UPDATE of this row could specify WHERE VERSIONING=rvv, so that the application could check the number of rows affected.

Transaction behaviour complicates this picture considerably, as clients can retrieve rows during a transaction that updates them. If the client application has programmed an explicit transaction and has made a copy of versioning information, it is the client’s responsibility to include a set of versioned objects to be updated when calling Commit (See sections 6.8 and 8.7.12).

### 5.2.4 Typed Graph Data

This section explains the relationship between the GQL and SQL aspects of the implementation. The database objects can include both relational tables and graph data<sup>30</sup>. The SourceIntro document in the distribution explains the data structures that achieve this: there are subclass relationships between the implementation classes that are additional to the semantics of both SQL and GQL.

The execution model described in the GQL specification ISO/IEC 39075 is very different from SQL, with the current working table, record, graph, and schema tracked between adjacent GQL statements that can modify any or all of these. It has been a surprise in the implementation that this model is so

<sup>28</sup> CHECK is one of Pyrrho’s primitive types from v7. See section 7.4.

<sup>29</sup> The discussion in this paragraph is about additional checks the application programmer wishes to include. The server will prevent read-write and write-write conflicts in any case.

<sup>30</sup> Both GQL and SQL are sublanguages of the effective language of this implementation.

completely orthogonal to the imperative SQL execution model that their implementations do not conflict. Managing the execution sequence for aggregations and composite queries is however not simple, as some statements (Match, For) create additional rows, while on the other hand grouping operations and composite queries such as EXCEPT need binding rowsets to be built already. Accordingly, the compound (or nested) statements become lists of lists: (a) each SQL step, terminated by a semicolon or END<sup>31</sup> may contain a GQL sequence, which adds rows to the result and/or changes to the GQL-data or catalogue, and whose bindings are then undone before the next step, (b) Result, Insert or further Match statements following a Match are executed on each binding unless a composite query statement follows. Full details are in the GQL specification.

### The RDBMS view of graph data

A NodeType (or EdgeType) corresponds to a single database object that defines both a base Table in the database and a user-defined type for its rows. This UDT is managed by the database engine by default, but the usual ALTER operations are available for both Table and UDT. Columns are provided in the node type for any properties that are defined for a node of this type. By default, Pyrrho will use the defining position of records as the uids of nodes and edges<sup>32</sup>, but a primary key can also be used for referencing them if it has been specified.

An EdgeType additionally specifies NodeType for connection relationships (a directed edge is said to leave one or more node(s) and arrive at other(s), and an undirected edge connects to one or more nodes). The edge type presumes a potential collection of edges connecting a given set of nodes. In a simple case, an edge can link two nodes. In a more complex example, a transfer between bank accounts authorised by a given mechanism represent an edge type linking 3 nodes (source of funds, destination of funds, and a credit card). It is easy to imagine more complex transactions (e.g., in property conveyancing). As with foreign keys, the engine maintains multisets for the reverse relationships (edges leaving from or arriving at the node).

TNode and TEdge are TypedValues whose dataType is a NodeType (resp EdgeType). A TGraph is a collection of node and edge uids.

Nodes and edges are represented by rows in the tables thus defined, and these can be updated and deleted using SQL in the usual ways, while ALTER TYPE, ALTER DOMAIN and ALTER TABLE statements can be applied to node and edge types.

In CREATE TYPE statements, metadata is available to declare a new type as a node type or an edge type, possibly specifying ID, LEAVING and ARRIVING columns and constraints: see further notes on this in section 5.9 below, and the Metadata syntax in section 7.2. However, a more convenient mechanism for defining or adding to typed graphs is provided by the CREATE syntax in this section 5.2.4 and illustrated below.

There is a natural interpretation of subtypes in Pyrrho, extending the UNDER feature in SQL by allowing subtypes to contain additional data. Fields in a type allow subtype values by default, and subtypes of edge types can specify subtypes for connected nodes. For example, an edgetype that connects LegalEntity to Account can have a subtype that connects Person to Loan (if Person is a subtype of LegalEntity and Loan is a subtype of Account). There is no requirement for column names to be unique in the subtype hierarchy: searching for any schema entry or record begins with a namespace determined by the context.

### Creating graph data in the RDBMS

A Neo4j-like syntax can be used to add one or more nodes and zero or more edges using the CREATE statement defined in section 7.2 below:

```
Create: CREATE GraphExp {THEN Statement}.
GraphExp: Node {Edge Node} {',' Node { Edge Node }} .
Node: '(' [id] [Label] [doc] ')'.
Edge: '-[' [id] [Label] [doc] ']->' | '<-' [id] [Label] [doc] ']-' .
Label: ':' id [Label].
```

In this syntax we see new diglyph and triglyph tokens for indicating the start and end of directed edges. In this syntax id is an SQL identifier for later reference in the statement, not a node ID: node and edge identities are specified in the JSON document doc. Pyrrho will supply a default value for ID if not specified.

<sup>31</sup> Braces {...} can be used as synonyms for BEGIN..END, and NEXT is a synonym for semicolon.

<sup>32</sup> GQL supports creation of nodes with multiple type labels (label sets).

The Label identifies a node or edge type (with an optional subtype), which may be new. As suggested above, the columns of new node and edge types are inferred from supplied property values and automatically modified as needed. All nodes and edges by default have the special property ID of type INT. The syntax connects up the edges: it is not permitted to specify leaving and arriving nodes explicitly.

As indicated, the syntax can contain a comma-separated list of graph fragments. The engine endeavours to combine these, verifying or modifying the available node and edge types, and defining new nodes and edges.

### Retrieving graph data from the RDBMS

The Match statement has the following syntax:

```
Match: MATCH MatchExp [Where] [Statement].
```

The given graph fragments are evaluated in a recursive process that finds sets of values for unbound identifiers, for which the graph fragments are all found in the database. The result is thus a set of successful assignments of unbound identifiers to TypedValues. The Statement if supplied is executed for each row of this set. To be unbound, an identifier should not match any top-level database object (table, view, domain, type, procedure) or any identifier defined earlier in the current SQL statement.

In Pyrrho, unbound identifiers can be used in the MatchExp not only as path, node, or edge identifiers, but also as labels, field names, or field values (not however as operands in expressions), allowing direct references to the bound value in later parts of the Match statement.

### The Graph view of graph data

The database is considered to contain a (possibly empty) set of disjoint TGraphs. Every Node in the database belongs to exactly one graph in this set.

The nodes of a graph are totally ordered by the order of insertion in the database, but this is not the traversal ordering: the first node in a graph is the first in both orderings. The traversal ordering starts with this first node but preferentially follows edges: the leaving edges ordered by their edge types and edge uids followed by arriving edges ordered similarly, while not visiting any node or edge more than once.

The set of graphs is (internally) totally ordered by the defining position of their first node.

In the data management language, an SqlNode is an SqlRow whose domain is a Node type. Evaluation of the SqlNode gives an explicit rowset of TGraph values. A TGraph specified in the above ways may match a subgraph of one of the graphs in this set, in which case we say the TGraph is found in the database.

## 5.3 Altering tables

SQL2023 allows for tables to be altered by adding, altering, or dropping columns, and adding and dropping constraints.

Tables can also be dropped. Pyrrho supports the renaming of objects, with the following syntax for renaming tables:

```
alter table oldname to newname
```

and similar syntax for renaming other objects. Renaming columns is a special case:

```
alter table tname alter [ column ] oldname to newname
```

The position of a column can also be changed. (Column positions have little semantic value but it is convenient to have a known ordering of columns in `select *` results.)

```
alter table tname alter [ column ] cname position n
```

Renaming of database objects is role-specific: renaming applies to the current role (see sec 5.5 below), and requires appropriate privileges. The database file (transaction log) uses numeric identifiers instead of names. The Log\$... system tables show these, while the Role\$... system tables show the current names in the current Role. The following screenshot shows these numeric identifiers in the log:

Pyrrho reconstructs compiled representations of database objects as required to reflect schema changes. The -V flag for the server allows this compilation process to be verified. Internally, objects not yet written

to the database are given temporary numeric identifiers starting with 0x400000000001. For convenience these are abbreviated to '1, '2 etc.

Pyrrho does not modify database data when column types are changed: however, it does check that the database data can be coerced into the new column type.

## 5.4 Sharing a database with other users

One of the first uses for the client utilities should be to create the base tables of the database and grant permissions on them to users. The best ways of doing this are explained in the next section.

The database creator initially is the only user known to the database. If there are no users defined, the only user known to the database is the one that started the server, and all objects are considered defined by the system. The first role created in the database takes over all of these objects, and the first user to be granted this role becomes the database owner, with administrative privileges on it. Users who have not been granted any permissions are guests, and by default have no privileges.

Thus, under Windows, Linux, or MacOS, if a database has no users as yet, but a role ADMIN has been defined, the creator of the database can share it with anyone by the following GRANT statement:

```
grant admin to public
```

This allows anyone to access or alter the data in any way. For a specific user mary on computer JOE on Windows, grant admin to "JOE\mary". The double quotes are needed because of case-sensitivity for user names; use `select user` to check the format for users on your system. To let mary alter the role she will need to be granted the admin option too. Only specifically-granted users are allowed to access the database over HTTP.

```
grant select atable to public
```

This allows any user to read the table ATABLE (and all its current columns). Other grant statements can be used to apply specific privileges to specific database objects. The full syntax for the grant and revoke statements is specified in the SQL standard and summarised in section 7.

There are some special cases in Pyrrho. At any time, a user has the privileges of at most one role (select `current_role` to see what it is), but can set the role to any role they have been granted. Domains and types are public. Views, stored procedures, triggers and constraints execute using their definer's role (set role is not a valid statement in such code). The database owner is able to access all of the system tables and profiles. A role is allowed to access the Role\$ system tables.

For best results only grant permissions to Roles: these are described next.

## 5.5 Roles

For example, suppose a small sporting club (such as squash or tennis) wishes to allow members to record their matches for ranking purposes:

```
Members: (id int primary key, firstname char)
```

```
Played: (id int primary key, winner int references members, loser int references members,
        agreed boolean)
```

For simplicity we give everyone select access to both these tables.

```
Create role admin
Grant select on members to public
Grant select on played to public
```

Although Pyrrho records which user makes changes, it will save time if users are not allowed to make arbitrary changes to the Played table. Instead we will have procedure `Claim(won,beat)` and `Agree(id)`, so that the Agree procedure is effective only when executed by the loser. With some simple assumptions on user names, the two procedures could be as simple as:

```
Create procedure claim(won int,beat int)
    insert into played(winner,loser) values(claim.won,claim.beat)

Create procedure agree(p int)
    update played set agreed=true
    where winner=agree.p and
    loser in (select m.id from members m
```

```
where current_user like '%'||firstname escape '!')
```

We want all members of the club to be able to execute these procedures. We could simply grant execute on these procedures to public. However, it is better practice to grant these permissions instead to a role (say, `membergames`) and allow any member to use this role:

```
Create role membergames 'Matches between members for ranking purposes'
Grant execute on procedure claim(int,int) to role membergames
Grant execute on procedure agree(int) to role membergames
Grant membergames to public
```

This example could be extended by considering the actual use made of the `Played` table in calculating the current rankings, etc.

In SQL2023, although a user may be entitled to use roles, he/she can only use one at a time, and the current role determines the permissions available. This is established in the connection string or using `SET ROLE`, and can be referred to as `SESSION_ROLE`.

Apart from the owner privilege (which can be held by just one user), granting privileges directly to users is deprecated. It is recommended to grant roles to users instead. Similarly, attempting to create a hierarchy of roles is also deprecated, and in Pyrrho the grant of role A to role B has the effect only of granting role A to all users authorised to use role B at the time of the grant: it does not create a permanent relationship between the roles; revoking a role from a role does nothing, and all roles are in the root namespace. This behaviour appears to be a departure from SQL2023 (see section 7.11 below).

Similarly, a grant of privileges does not create any permanent relationship between roles. For example, granting `Select` on a Table implies granting `select` on all of the *current* columns. The grant can be repeated later if new columns are added, or the new columns can be granted. Similarly in Pyrrho, access to a column can be revoked even though the role was previously granted access to the whole table (again see section 7.11).

A user who has been granted the `admin` option for a role can define new tables, procedures, constraints, types, etc in that role, and can grant privileges on these objects to other roles. All SQL code, if it is executable by the current role, executes with the permissions of the owner of the code (definer's rights). A user entitled to administer a role can modify metadata (including the object name, but excluding the `iri`) of objects visible from their role: other defining properties of the object can only be changed by the owner or schema role. All standard types are `PUBLIC` and all roles remain in the root namespace. Other objects can be prefixed with the name of the role if this is helpful for disambiguation.

On creation a database has a default role with the same name as the database, and the owner of the database can use this “schema” role to create the starting set of objects for the database.

The System tables can be used to ascertain the privileges held at any time: from v4.5 these are accessible by the database owner, or by using the schema role.

## 5.6 Stored Procedures and Functions

Pyrrho supports stored procedures and functions following the SQL2023 syntax (volumes 2 and 4). The programming model offered in this way is computationally complete, so the use of external code written in other programming languages is not supported.

Following SQL2023 the syntax `:v` is not supported for variable references, and instead variables are identified by qualified identifier chains of form `a.b.v`. The syntax `?` for parameters is not supported either.

Following SQL2023-2-11.60, procedures never have a `returns` clause (functions should be used if a value is to be returned), and procedure parameters can be declared `IN`, `OUT` or `INOUT` and can be `RESULT` parameters. Variables can be `ROW` types and collection types. For functions, `TABLE` is a valid `RETURNS` type (it is strictly speaking a “multiset” in SQL2023 terminology). From SQL2023-2-6.45 we see that `RETURN TABLE` (`RowSetSpec`) is valid syntax for a return statement.

The operation of the security model for routines in SQL2023 is rather subtle. All routines operate with definer's rights by default, but access to them is controlled according to the current role.

Pyrrho allows some metadata properties to be set for functions. `MONOTONIC` (order-preserving) functions used in join conditions can allow Pyrrho to speed up joins by sorting the table operands provided `USING` syntax specifies the use of an adapter function. The `INVERTS` metadata property

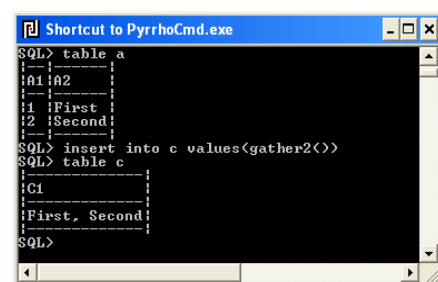
establishes a pair of mutually inverse functions and this information means that views and joins defined USING such functions can be updatable depending on the availability of keys.

### 5.6.1 Examples

The following functions perform the same task. The first uses a handler, while the second uses a for statement.

```
create function gather1() returns char
begin
  declare c cursor for select a2 from a;
  declare done Boolean default false;
  declare continue handler for sqlstate '02000' set done=true;
  declare a char default '';
  declare p char;
  open c;
  repeat
    fetch c into p;
    if not done then
      if a = '' then
        set a = p
      else
        set a = a || ', ' || p
      end if
    end if
  until done end repeat;
  close c;
  return a
end
```

```
create function gather2() returns char
begin
  declare b char default '';
  for select a2 from a do
    if b='' then
      set b = a2
    else
      set b = b || ', ' || a2
    end if
  end for;
  return b
end
```



## 5.7 Structured Types

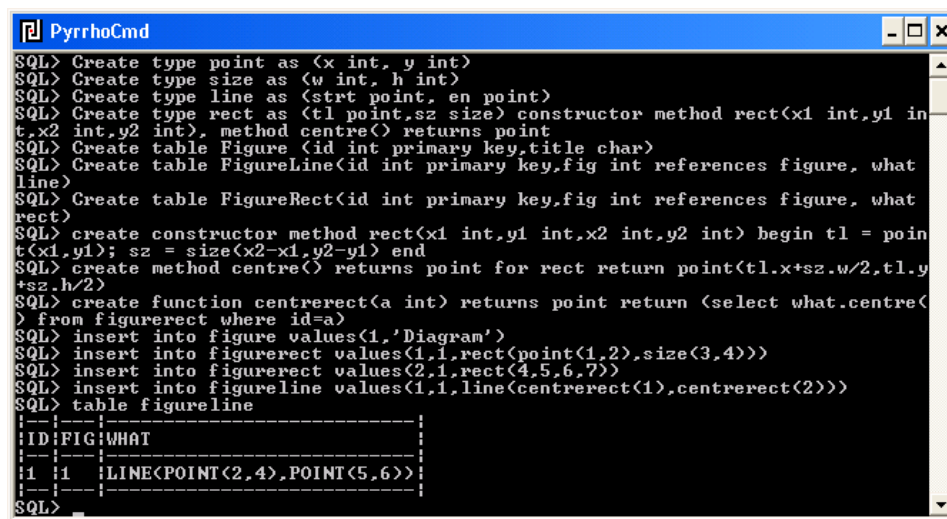
SQL2023 supports structured types with single inheritance. Structured types, multisets, sets, lists and arrays can be stored in tables. There is a difference between (say) a table with certain columns, a multiset of rows with similarly named fields and a multiset of a structured type with similarly named attributes, even though in an element of each of these the value of a column, field or attribute respectively is referenced by syntax of the form a.b. Some constructs within SQL2023 overcome these differences: for example, the INSERT statement uses a set of values of a compatible row type to insert data into a table,

and TABLE v constructs a table out of a multiset v. The type model in Pyrrho allows user-defined types to be simple or structured, with optional prefixes and suffixes for input and output, and constraints.

To use structured types, it is necessary to CREATE TYPE for the structured type: this indicates the attributes and methods that instances of the type will have. Then a table (for example) can be defined that has a column whose values belong to this type. At this stage the table could even be populated since there is an implicit constructor for any structured type; but before any methods can be invoked they need to be given bodies using the CREATE METHOD construct. Note that you cannot have a type with the same name as a table or a domain (since a type has features of both).

Values of a structured type can be created (using NEW), assigned to variables, used as parameters to suitably declared routines, used as the source of methods, and placed in suitably declared fields or columns.

GQL supports multiple inheritance since node insertion in a graph can specify a label expression that is a set of labels. Accordingly, node and edge types in Pyrrho are allowed to have multiple supertypes (other user-defined types are not), and there are some modifications to syntax rules to permit this. Note, however, that unpredictable behaviour can result from badly defined update triggers on supertypes.



```

PyrrhoCmd
SQL> Create type point as <x int, y int>
SQL> Create type size as <w int, h int>
SQL> Create type line as <strt point, en point>
SQL> Create type rect as <t1 point, sz size> constructor method rect<x1 int, y1 int, x2 int, y2 int>, method centre<> returns point
SQL> Create table Figure <id int primary key, title char>
SQL> Create table FigureLine<id int primary key, fig int references figure, what line>
SQL> Create table FigureRect<id int primary key, fig int references figure, what rect>
SQL> create constructor method rect<x1 int, y1 int, x2 int, y2 int> begin t1 = point<x1, y1>; sz = size<x2-x1, y2-y1> end
SQL> create method centre<> returns point for rect return point<t1.x+sz.w/2, t1.y+sz.h/2>
SQL> create function centrerect<a int> returns point return <select what.centre<> from figurerect where id=a>
SQL> insert into figure values<1, 'Diagram'>
SQL> insert into figurerect values<1, 1, rect<point<1, 2>, size<3, 4>>>
SQL> insert into figurerect values<2, 1, rect<4, 5, 6, 7>>
SQL> insert into figureline values<1, 1, line<centrerect<1>, centrerect<2>>>
SQL> table figureline
+-----+
| ID | FIG | WHAT |
+-----+
| 1 | 1 | LINE<POINT<2, 4>, POINT<5, 6>> |
+-----+
SQL>

```

Notes: 1) The coordinates have been declared as int, so the first point here is not (2.5, 4). 2) In v7, the last method here must refer to a as id=centrerect.a, not id=a. This is because in the SQL standard (2011, sec 6.6), unqualified names need to lie in the context of a range variable or table name, to which they refer, and so an identifier chain is required in this example.

Arrays, sets, lists, and multisets of known types do not need explicit type declaration. Their use can be specified by the use of the keyword ARRAY, LIST, SET or MULTISSET following the type definition of a column or domain. Note that in GQL v1.0, LIST and ARRAY are synonyms but this is not the case in Pyrrho<sup>33</sup>.

The VECTOR type represents points in an n-dimensional space. The coordinates are typically numbers or strings, one for each dimension 1,..., and the GQL specification defines a set of standard vector distance functions whose values in this implementation are of type REAL. The names of these functions are listed in section 7.9.

## 5.8 Triggers

SQL2023 supports triggers.

Pyrrho has built-in facilities to do activity logging (see section 3.5 and 8.2). However, triggers allow for a more customizable approach as the following example shows:

```

create table test1(id int primary key, val char)
create table test2(id int primary key, ent char, val char)

```

<sup>33</sup> In this implementation lists can be accessed using square bracketed subscripts 0,.. (The current GQL specification does not specify such an operation.)

```

create procedure log(g char,h char) insert into test2(ent,val)
values(g,h)
create trigger logininsert after insert on test1 referencing new row as
a for each row call log('inserted',a.val)
create trigger logupdate before update on test1 referencing old row as
a new row as b for each row call log(a.val,b.val)
create trigger logdelete before delete on test1 referencing old row as
a for each row call log('deleted',a.val)
insert into test1 values(1,'First'),(2,'Second')
table test2
update test1 set val='New One' where id=1
table test2
delete from test1 where id=2
table test2

```

The screenshot shows a PyrrhoCmd window with the title 'PyrrhoCmd'. The command prompt is 'SQL> @trig.sql'. The output shows three tables: test1, test2, and test1 again. The first table test1 has two rows: (1, inserted, First) and (2, inserted, Second). The second table test2 has three rows: (1, inserted, First), (2, inserted, Second), and (3, First, New One). The third table test1 has four rows: (1, inserted, First), (2, inserted, Second), (3, First, New One), and (4, deleted, Second).

ID	ENT	VAL
1	inserted	First
2	inserted	Second

ID	ENT	VAL
1	inserted	First
2	inserted	Second
3	First	New One

ID	ENT	VAL
1	inserted	First
2	inserted	Second
3	First	New One
4	deleted	Second

## 5.9 Subtype semantics

All records have a default identity given by their defining position in the database log. This facilitates some aspects of Pyrrho's implementation of subtypes, since subtype instances whose properties coincide in the supertype remain distinguishable in the supertype. Prefixes and Suffixes are supported for currency and physical types. The Metadata concept provides the main support for this additional information in SQL DDL. Like domains, types can be altered using the ALTER syntax subject to access privileges and restrict/cascade semantics depending on existing database contents.

Subtypes inherit columns, methods, and constraints from their supertype, and overriding of methods is supported. Changes to types affect subtypes, so any change to a supertype is also subject to access and semantic conditions on its subtypes.

Subtypes of node/edge types are node/edge types (the metadata does not need to be repeated). In the physical database, properties of the nodes/edges of these types are placed in the base table(s) for the type and supertype(s) that specify them (the partial records in each base table all have the same defining position, corresponding to the position of the record in the transaction log). Thus the transaction log shows the record, and the supertype table will contain rows from all its subtypes (the SQL function SPECIFICTYPE() gives the name of the subtype), while selecting from the subtype gives each of its rows, including columns inherited from the supertype. The order and visibility of columns is determined by the current role.

### 5.9.1 IRI references and subtypes

Semantic information can be used to define a subtype, for example, a Pyrrho extension to SQL2023 allows declarations such as:

```
CREATE DOMAIN ukregno 'uri://carregs.org/uk'
```



The final string constant is stored by the database as metadata. Subtype information can be examined using the standard SQL2023 type predicate, for example

```
SELECT * FROM cars WHERE reg IS OF (ukregno)
```

## 5.9.2 Row and table subtypes

In an INSERT operation, whole rows or tables can be assigned a subtype using the TREAT function. A structured type can be declared with additional metadata uri information, such as

```
CREATE TYPE t AS (c CHAR, b INT) 'http://a.com'
```

Then supposing table A had been created with a compatible row type, such as CREATE TABLE(c CHAR,b INT), we could write

```
INSERT INTO A TREAT (VALUES ('Ex',1)) AS t
```

The type of a row can be tested using the ROW keyword, e.g. ROW IS OF(*type*) . Following the SQL standard, the row type for a table or view T is REF(T).

## 5.9.3 Graph Types and Columns

The GQL standard says very little about how node and edge types are implemented, but for efficiency in searching any implementation must consider how to speed up the process using indexes. By default, Pyrrho uses index-like mechanisms based on the defining position of records (uids) for edge connections and primary key indexes for an id field if defined but also supports primary and foreign keys. The GQL standard allows the use of label sets such as A&B and the overloading of edge type names. In Pyrrho, graph types can be created and modified using SQL create/alter type together with metadata, or during graph creation using INSERT/CREATE<sup>34</sup>.

The implementation of graph types in Pyrrho is intended to be intuitive, so the rest of this section is for reference: it is about the algorithms used when constructing a new graph type. There are some resulting extensions to SQL, allowing multiple inheritance to be permitted for graph types in addition to nesting. Pyrrho also permits n-ary edge types (see the Connection syntax in sec 7.2). If instead a graph type is constructed by an inline insert/create statement it must have a label without & . Pyrrho does not have a distinguished ID column: if such a column is declared it has no special status<sup>35</sup>.

In simple cases, simple labels are used. As an extension in Pyrrho, subtype labels can be constructed using => , and then the subtype automatically has all the columns (properties) and constraints of its supertype. Another sort of inheritance occurs for edge types where the same label (and associated properties) is used for edges linking different node types. For example, suppose an edge label IsLocatedIn is used to locate countries in continents as well as cities in countries, but city and country are unrelated types. Then there will be IsLocatedIn columns in both city and country and two corresponding columns in the implementation of the edge type of type Position.

Other columns are simply inherited and added in declaration order.

## 5.9.4 Value binding in graph patterns

Match statements can bind property names and values: such bindings have separate columns in the binding table. In path pattern matching, such names and values are grouped (into lists named by the binding name) and can be aggregated. On the other hand, the path binding names a list of the named lists of nodes and edges matched. Nested paths will give nested lists. During the path matching process, the instantaneous state of these lists is accessible, and this is useful for creating predicates (monotonicity, distinctness, truncation) to control the pattern matching process.

<sup>34</sup> Any resulting schema changes are transacted within the relevant statement. Inferred changes are only made for a simply labelled graph type: if a node is being created with labels A&B it will be committed as a single record with properties drawn from types A and B but neither node type will be altered, and no additional types or indexes will be created.

<sup>35</sup> For example, ID columns in node types are not automatically made primary keys, unless an explicit primary key constraint is added to the element type specification (such as primary key(id)), and then the primary key constraint is applied to any subtypes. Edge type specifications should not specify primary keys, as particular edges should be searched using the graph structure (with MATCH).

The execution models for SQL and GQL differ substantially, and this affects the accumulation of bindings in a sequence of query clauses<sup>36</sup>. GQL supports chains of MATCH statements without separators, optionally followed by RETURN/YIELD (in Pyrrho, DML statements are also allowed); otherwise, the possibly modified current record and current query result pass to the following statement. This means that the occurrence of a separator (semicolon or NEXT) unbinds the identifiers bound in MATCH statements in the current nested/compound statement<sup>37</sup>.

Window functions are not currently mentioned in the GQL specification. Any defined on the match statement are also applied during the matching process, based on the current state of the lists and the nodes and edges being considered for addition.

---

<sup>36</sup> In the current GQL specification, aspects of this are discussed in sections 20.12 and 22.6.

<sup>37</sup> The bindings (uid to value) are retained in the binding table. If the identifiers are re-used later, they will receive new uids.

## 6. Pyrrho application development

This section contains technical information required by database application programmers. For many purposes the first few subsections are sufficient.

For simplicity, it is assumed in sections 6.1-6.5 that the application programmer is writing in C#. Later sections discuss the APIs available for Python, Java, PHP, and SWI-Prolog, all available on Windows and Linux.

### 6.1 Getting Started

Application programming with Pyrrho can be carried out using C#, Java, Python, and even SWI-Prolog, and the source code for all of the libraries is available in the distribution.

The best support is available with C#, where there are two programming models available: ADO.NET and “Plain Old C# Objects” POCO, which is more like an entity framework. As with many other DBMS, Pyrrho provides its own version of both in the PyrrhoLink.dll file in the distribution and the directive

```
using Pyrrho;
```

For both models, the connection to the server is PyrrhoConnect (sec 6.2-3), and the resulting effective API is documented in section 8.7 of this manual. The Pyrrho API supports a simple kind of prepared statements, see sec. 8.7.12, but in a different way from the SQL standard, as the prepared statements are stored in the current connection and not in the database.

Unless the dll is installed in the global assembly cache, it should be copied to the same folder as the application executable. If you are using a tool such as Visual Studio to develop your application, ensure that the project references PyrrhoLink.dll. You may need to browse to the location where Pyrrho has been installed. Visual Studio will then make information from PyrrhoLink.dll available during compilation and place a copy of PyrrhoLink.dll in the same folder as the executable.

### 6.2 Opening and closing a connection

The database connection is provided using an extension to the standard ADO.NET IDbConnection interface:

```
var db = new PyrrhoConnect(connectionstring);
```

See section 6.4 for details of the connection string. A sample is provided below.

The connection must be opened before it can be used:

```
db.Open();
```

Connections should be closed when no longer required:

```
db.Close();
```

An application may use this cycle many times during its operation, as connections may be opened for different databases, groups of databases, or using different roles. By default, the connection operates in autocommit mode where every (possibly multiline) command from the client is immediately committed. If explicit transactions are used, any uncommitted transactions are silently rolled back when a connection is closed (see section 8.7.20). Two functions in this interface described below are CreateCommand (section 6.5) and BeginTransaction (section 6.7).

As usual with ADO.NET, at most one IDataReader can be open for any connection. Remember to close the IDataReader before calling another ExecuteReader.

For example, the following console program connects to a database Movies on the local server, and lists the TITLES found in table MOVIE:

```
using Pyrrho;
class Test
{
    public static void Main(string[] args)
    {
        var db = new PyrrhoConnect("Files=Movies");
        db.Open();
        var cmd = db.CreateCommand();
```

```

        cmd.CommandText = "select title from Movie";
        var rdr = cmd.ExecuteReader();
        while (rdr.Read())
            Console.WriteLine((string)rdr["TITLE"]);
        rdr.Close();
        db.Close();
    }
}

```

Note that SQL is not normally case sensitive: see section 5.2. If you want SQL identifiers to be case sensitive, you will need to double-quote them, and in C# strings, the double-quote will need to be escaped. For more details of the ADO.NET and similar functionality, see section 8.6.

POCO technology is also available. Pyrrho will supply class definitions to paste into your application program, either using the REST interface, or from the Role\$Class system table. If this has been done for the MOVIE class here, the above code can be simplified to:

```

using Pyrrho;
class Test
{
    public static void Main(string[] args)
    {
        var db = new PyrrhoConnect("Files=Movies");
        db.Open();
        var obs = db.Get("/MOVIE");
        foreach(MOVIE m in obs)
            Console.WriteLine(m.TITLE);
        db.Close();
    }
}

```

As suggested by format of the Get parameter, this mechanism uses the new role-based REST features. See section 6.5 below.

## 6.3 The connection string

ConnectionString = [Files=]filename {';'Setting} .

Setting = id='val'{';'val} .

If the connection string begins with Files=, this portion is ignored for reasons of backward compatibility for single-database connections. Note that a database file name cannot contain = or ; .

The possible fields in Settings are as follows:

Field	Default value	Explanation
Base		Used by server-server communication to create a new partition remotely. Not for client-server use.
BaseServer		Used by server-server communication to create a new partition remotely. Not for client-server use.
Coordinator		Used in server-server communications: the transaction coordinator server
CaseSensitive	false	The default false is for standard SQL identifier syntax (double-quotes typically enclose identifiers such as columns that use upper and lower case, and unquoted identifiers are changed to upper case). If set to true, double quotes can be used instead of single quotes, and identifiers are not converted to upper case.
Files		(Despite the name, there can only be one.)
Graph	databasename	The name of the home graph.
Host	127.0.0.1	The name of the machine providing the service.
Length		Used in server-server communication to notify clients of a new master file length. Not for client-server use. The connection is closed immediately.
Locale		The default locale is given by the regional settings for the client.

Modify		The default value is true for the first file in the connection, and false for others. If the value true is specified then it applies to all of the Files in the current connection string.
Port	5433	The port on which the server is listening
Provider	PyrrhoDBMS	
Role		A role name selected as the session role. If this field is not specified, the session role will be the default database role if the user is the database owner or has been granted this role (it has the same name as the database), or else the guest role, which can access only PUBLIC objects.
Schema		The name of the home schema
Stop		<i>If a value is specified, this means that Pyrrho is to load the database as it was at some past time.</i>
User		<i>This field is supplied by infrastructure</i>

## 6.4 REST and POCO

POCO stands for Plain Old CLR Object. In addition to the HTTP REST service in section 3.8, Pyrrho has a RESTful API that supports row-versioning (cp. Laiho and Laux, 2010). The Role\$Class system table (see sec 8.4.1) supplies a set of class definitions that can be pasted into a C# application<sup>38</sup>. Similar tables Role\$Java (8.4.9) and Role\$Python (8.4.16) provide class definitions for Java and Python.

If a role contains the metadata flag ENTITY for a table, Pyrrho applies some object-oriented features similar to structured types. For example, the versioning mechanism is available in PyrrhoConnect, together with navigation properties similar to those for URL-based web access (see sec. 3.8.2).

Suppose that the database t64 contains four base tables as follows, all with ENTITY flag:

Customer (Id, Name)

Item (It, Name, Price)

Order (Id, Cust references Customer, OrderDate, Total)

OrderItem (Id, Oid references Order, Item references Item, Qty)

Suppose further that the database manager has defined the database and inserted some sample data using the following SQL:

```
create role "Sales"
grant "Sales" to "machine\user"
set role "Sales"

create table "Customer" (id int primary key,"NAME" char, unique("NAME")) entity

[create table "Order" (id int primary key,cust int references "Customer", "OrderDate"
date,"Total" numeric(6,2)) entity]

create table "Item" (id int primary key,"NAME" char, price numeric(6,2)) entity

[create table "OrderItem" (it int,oid int references "Order" on delete cascade,
item int references "Item",qty int,primary key(oid,it)) entity]

insert into "Customer" values (10,'John'),(11,'Fred'),(12,'Mary')

[insert into "Order" values (1230,10,date'2022-05-10',34.56),(1231,12,date'2022-05-
11',67.89),(1234,11,date'2022-06-04',56.78)]

[insert into "Item" values(71,'Pins',0.78),(72,'Pump',67.0),(73,'Crisps',0.89),
(74,'Rug',56.78),(75,'Bag',33)]

[insert into "OrderItem" values (100,1230,75,1),(101,1230,71,2),(102,1231,73,1),
(103,1231,72,1),(103,1234,74,1)]
```

<sup>38</sup> It is important to note that these class definitions should always be generated from the database and not copied from definitions used in another database, not even a database with the same structure and objects.

Note that NAME is enclosed in double quotes because NAME is a reserved word in SQL

Then the Role\$Class system table will contain C# class definitions including fragments similar to the following:

```
using Pyrrho;
using System;

/// <summary>
/// Class Customer from Database t64, Role Sales
/// PrimaryKey(ID)
/// Unique(NAME)
/// </summary>
[Table(97,191)]
public class Customer : Versioned
{
    [Field(PyrrhoDbType.Integer)]
    [AutoKey]
    public Int64? ID;
    [Field(PyrrhoDbType.String)]
    public String? NAME;
    public Order[]? orders => conn?.FindWith<Order>(("CUST", ID));
}

/// <summary>
/// Class Order from Database t64, Role Sales
/// PrimaryKey(ID)
/// ForeignKey, RestrictUpdate, RestrictDelete(CUST) Customer
/// </summary>
[Table(242,416)]
public class Order : Versioned
{
    [Field(PyrrhoDbType.Integer)]
    [AutoKey]
    public Int64? ID;
    [Field(PyrrhoDbType.Integer)]
    public Int64? CUST;
    [Field(PyrrhoDbType.Date)]
    public Date? OrderDate;
    [Field(PyrrhoDbType.Decimal, 373, "Domain NUMERIC Prec=6 Scale=2")]
    public Decimal? Total;
    public Customer? customer => conn?.FindOne<Customer>(("ID",CUST));
    public OrderItem[]? orderItems => conn?.FindWith<OrderItem>(("OID", ID));
}
```

There will be similar entries for the OrderItem class and any class it references. The doubled parentheses are used where the parameters are pairs, in all these cases the column name and column value for a where clause. Taking the highlighted pieces in order:

- (b) we are told that the NAME field is indexed as it will be unique
- (a) The Table attribute gives the defining proposition of the base table and the file position of the last schema change for the class: this is used as a check during connection to the database;
- (c) the primary key of the Customer table is ID, and by default integer keys that are not also part of a foreign key are declared AutoKey because the server will supply a non-null integer key value if the programmer does not do so,
- (d) the Order table contains a foreign key referencing an entity Customer, so a customer instance has a navigation property listing all the orders referencing that customer, whose name is obtained (sometimes awkwardly) simply by adding an s.
- (e) Total have datatype information, and the foreign key CUST mentioned above is associated with a navigation property whose value is the referenced Customer instance. The programmer can omit details of classes and properties they do not need in their application and refactor the fieldnames: the class names and attributes should not be changed.
- (f) A database connection conn to the database is required. PyrrhoConnect conforms to normal ADO.NET/ODBC rules<sup>39</sup>: it is opened for a database and role and may have a current transaction that can be committed or rolled back. A suitable static variable conn needs to be declared in the class containing the above class definitions.

Then instances of these classes can be retrieved, and new and modified instances of these classes committed, as described below. For exceptions that can occur, see the documentation in section 8.1.

<sup>39</sup> Threading safeguards are applied by the runtime.

The above example shows navigation defined implicitly by foreign key relationships directly referencing entities creating one-many and many-one relationships. One-one relationships are recognized by Pyrrho when the referencing columns in a foreign key are a key in the referenced entity and a many-many relationship is recognized when a foreign key reference in one entity table is to a non-entity table (the auxiliary table) with a foreign key to another entity. A many-many relationship uses `FindIn<C>()` to retrieve keys from the auxiliary table. Integer suffixes are added to field names if necessary to disambiguate them.

The Versioned base class contains the following data<sup>40</sup>.

```
public class Versioned
{
    public PyrrhoConnect conn41;
    public string entity = "";
    public string version = "";
}
```

This leads to a very tidy RESTful API, consisting of the following methods for the PyrrhoConnect (or Connection) class, where E is Versioned or a subclass of Versioned defined by code obtained from the Role\$Class (or Role\$Java or Role\$Python) system table in Pyrrho. Classically, REST uses the HTTP 1.1 verbs of GET, POST, PUT, and DELETE, and the strongly typed Get and FindXXX methods below are recommended over Get(..). With auto-committed transactions, POST always silently updates the entity field, and other fields may also be updated by auto-key or triggers<sup>42</sup>.

Class PyrrhoConnect:

Method	Explanation
C#: E[]? FindAll<E>() Java: Versioned[] FindAll(E.class) Python: E[] findAll(E)	Retrieve all entities of the given Versioned type.
C#: E[]? FindIn<E>(string sql) Java: Versioned[] FindIn(sql) Python: E[] findIn(E,sql)	sql should be a SELECT statement that returns a list of keys for E
C#: E? FindOne<E>(params (string?,IComparable[]? w) Java: FindOne(E.class,Object[] w) Python: E findOne(E,w)	Retrieve a single entity of a given Versioned type E with key fields w. w is a comma-separated set of conditions of form <i>field=value</i> , using programming language format.
C#: E[]? FindWith<E>(string? w) Java: Versioned[] FindWith(E.class,String w) Python: E[] findWith(E,w)	Retrieve a set of Versioned entities satisfying a given condition. Field names are case sensitive and values are in SQL format.
C#: E[]? FindWith(params (string,IComparable[]? w)	Retrieve a set of Versioned entities satisfying a given condition. w is a comma-separated set of conditions of form <i>field=value</i> , using programming language format.
C#: E[] Get<E>(string rurl) Java: Versioned[] Get(E.class,String w) Python: E[] get(E,rurl)	The relative url provided should be compatible with the Versioned subclass E.
C#: void Post(E) Java: void Post(E ob) Python: post(ob)	The object should be a new entity. An integer key field will be autopopulated with a suitable value, but otherwise it is the caller's responsibility to find a suitable key for the new object. Invokes triggers if any.

Methods for any Versioned subclass E:

<sup>40</sup> The versioning is remembered and will be checked even in a later Connection. Explicit transactions should be kept as short as possible since they must run exclusively in one thread.

<sup>41</sup> The application may use several database connections. If so, these use different threads and may see different versions of the database.

<sup>42</sup> Triggers may create new versions of other entities in the database, but the client will need to use Get to discover such side effects.

Method	Explanation
C#: void Delete() Java: void Delete() Python: delete()	Delete the given entity from the database table E, calls triggers if any.
C#: void Get() Java: void Get() Python: get()	Overwrites the fields of this with the latest version of the entity from its PyrrhoConnect.
C#: void Put(E ob) Java: void Put(E ob) Python*: put(ob)	Make a new version of the entity. With auto-commit, this will update the version field and possibly other fields depending on integrity constraints and triggers.

## 6.5 DataReaders

The PyrrhoReader interface is very similar to IDataReader as described in the ADO.NET documentation. To get a PyrrhoReader, call the ExecuteReader() method of PyrrhoCommand, e.g.:

```
var rdr = cmd.ExecuteReader();
```

The columns that will be returned in the rows of the DataReader can be accessed using the following methods (among others):

Property or Method signature	Explanation
int FieldCount	Gets the number of fields returned per row
string GetName(int i)	Returns the name of the ith field (the first field is field 0)
Type GetFieldType(i)	Returns the System.Type of the ith field

Before a PyrrhoReader can access any data, the Read() method must be called. Each time it is called, it moves on to the next row of the results if there is one. This function returns a Boolean value: which is true if Read() has succeeded in moving to the next row of data, and false if there is no more data.

Assuming that Read() has returned true, the fields in the returned row can be obtained by indexing the DataReader object. Fields can be indexed by ordinal position or by name. The value returned is a System.Object. If the corresponding value might be a null value, then it can be checked against DBNull.Value (or for being DBNull) before being cast to the expected System.Type.

For example:

```
if (!(rdr[1] is DBNull)) then Console.WriteLine((string)rdr[1]);
```

For languages where casting to different types is awkward, the DataReader interface has a range of functions of form GetByte(i), GetInt64(i) etc. For integers and numerics whose precision cannot fit into the standard types, Pyrrho returns a string representation. If this is expected, then you should test if the value is string.

SQL basic type	.NET data type
Boolean	System.Boolean
Int, integer	System.Int64
Real, Float	System.Double
Char, CLOB	System.String
BLOB	System.Byte[]
Date, Timestamp	System.DateTime
Row, Interval, Array, Multiset	See section 8.6

If indexing by name is used, remember that strings in the programming language are case-sensitive, even though SQL (unquoted) identifiers are not, so you will probably need to ensure your field names are in upper case letters.

The client library uses the DataReader interface with as few added classes as possible. The only added classes are PyrrhoRow, PyrrhoArray, and PyrrhoInterval. Dates and Timestamps use the DateTime class in the common language runtime, Times use the TimeSpan class for a simple time of day, but Intervals are handled using PyrrhoInterval. The three new classes are documented in section 6.8.

The routines ExecuteReaderCrypt and ExecuteNonQueryCrypt send the SQL string to the server using Pyrrho's encryption algorithm.



## 6.6 Using PHP

There is an extra class ScriptConnect in PyrrhoLink.dll which is very useful for use with the scripting language PHP.

PHP can be used for building web applications, and then the same considerations as in the last section apply for the user identity of the web server and ownership of the databases.

To enable PHP support for Pyrrho under Windows, an administrator needs to issue the following two commands from the folder that contains PyrrhoLink.dll:

```
gacutil -i PyrrhoLink.dll
regasm PyrrhoLink.dll -tlb:PyrrhoLink.tlb
```

You need to ensure that your PHP installation is 32-bit and has the php\_com\_dotnet extension.

The following steps can be used to access Pyrrho databases from PHP:

To create a connection to a Pyrrho database:

```
$conn = new COM("OSPLink"); // "PyrrhoLink" for the Pro version
$conn->ConnectionString = ...;
$conn->Open();
```

Once a connection is open as above, an SQL statement can be sent to the database as follows

```
$rdr = $conn->Execute(...);
```

The result returned will be a ScriptReader in the case that the SQL statement returns data.

Then

```
$row = $rdr->Read();
```

can be used to return successive rows of the data as variant arrays. If there are no more rows then the value returned is -1, which can be tested using `is_int($row)`:

```
$row = $rdr->Read();
while(!is_int($row))
{
    print($row[0].': '.$row[1].<br/>'); // or "\r\n"
    $row = $rdr->Read();
}
```

`$rdr->Close();` should be called when the reader is no longer required.

`$conn->Execute(...);` can also be used for other types of SQL statements.

## 6.7 Python

PyrrhoLink.py is available in the distribution and enables the open-source Pyrrho server PyrrhoSvr to be accessed from Python 3.4 clients. The API has similarities to Pyrrho's version of ADO.NET as documented in section 8.7, and the following subsections are numbered similarly to those of section 8.7 in a conscious attempt to show the relationship.

Since version 5.4 of Pyrrho, thread-safety is enforced by the PyrrhoLink.py library. The connection object can be shared between threads. But a connection can have at most one transaction and/or command active at any time, and these cannot be shared between threads. As a result, the methods noted below will block until the connection is available.

To use PyrrhoLink.py, place it in the same folder as your Python script.

For example:

```
from PyrrhoLink import *
from builtins import print

conn = PyrrhoConnect("Files=Temp;User=Fred")
conn.open()
try:
```

```

conn.act("create table a(b date)")
except DatabaseError as e:
    print(e.message)
conn.act("insert into a values(current_date)")
com = conn.CreateCommand()
com.CommandText = 'select * from a'
rdr = com.ExecuteReader()
while rdr.read():
    print(rdr.val(0))
rdr.close()
print("Done")

```

### 6.7.1 (AutoKeyAttribute)

There is no analogue to C# attributes/Java annotations in Python.

### 6.7.2 DatabaseError

Attribute	Explanation
<i>dict</i> info	Information placed in the error: see section 8.1.2
<i>str</i> message	The message text: see section 8.1.1
<i>str</i> sig	The SQLSTATE

### 6.7.3 (Date)

PyrrhoLink.py uses the Python *date* class.

### 6.7.4 DocArray

Attribute	Explanation
build(ob)	Append the attributes of ob not starting with <code>_</code> to this document; the process recursively builds embedded Documents and DocArrays for structured values
bytes()	Create the Bson representation of this DocArray
<i>cls</i> [] extract(cls)	Construct an array of cls objects from this
fromBson(bytes)	Append the given Bson data to an empty DocArray
<i>list</i> items	The items of the DocArray
parse(str)	Append items from the given string to this DocArray
str()	Create the Json representation of this DocArray

### 6.7.5 Document

Attribute	Explanation
build(ob)	Append the attributes of ob not starting with <code>_</code> to this document; the process recursively builds embedded Documents and DocArrays for structured values
bytes()	Create the Bson representation of this document
<i>cls</i> _extract(cls)	Construct an object of type cls from this
fromBson(bytes)	Append the given Bson data to this document
<i>list</i> fields	Each field is a pair (key,value)
parse(str)	Append fields from the given string into this document
str()	Create the Json representation of this document

### 6.7.6 DocumentException

This subclass of Exception is used to report parsing errors in the Document.parse method.

### 6.7.7 (ExcludeAttribute)

There is no analogue to C# attributes/Java annotations in Python.

### 6.7.8 (Field Attribute)

There is no analogue to C# attributes/Java annotations in Python.

### 6.7.9 PyrrhoArray

Attribute	Explanation
<i>str</i> kind	The domain name if defined
<i>list</i> data	The items in the array

### 6.7.10 PyrrhoColumn

Attribute	Explanation
<i>str</i> columnName	The name of the column
<i>str</i> caption	The name of the column
<i>str</i> datatypeName	The domain or type name of the column
<i>int</i> type	The PyrrhoDbType of the column (see sec 6.8.13)

### 6.7.11 PyrrhoCommand

Attribute	Explanation
<i>str</i> commandText	The SQL statement for the Command
PyrrhoConnect conn	The connection
PyrrhoReader ExecuteReader()	Initiates a database SELECT and returns a reader for the returned data (as in IDataReader). Will block until the connection is available.
int ExecuteNonQuery()	Initiates some other sort of Sql statement and returns the number of rows affected. Will block until the connection is available.

### 6.7.12 PyrrhoConnect

Attribute	Explanation
<i>int</i> Act(sql)	Convenient shortcut to construct a PyrrhoCommand and call ExecuteNonQuery on it. Will block until the connection is available.
PyrrhoTransaction BeginTransaction()	Start a new isolated transaction (like IDbTransaction). Will block until the connection is available.
<i>bool</i> Check(ch) <i>bool</i> Check(ch, rc)	Check to see if a given Versioned rowCheck string is still current, i.e. the row has not been modified by a later transaction. (See sec 5.2.3 and 8.7.21). The second version shown also tests the readCheck. (There is no need to perform a check unless the Versioned data is from a previous transaction.)
Close()	Close the channel to the database engine
<i>str</i> connectionString	Get the connection string for the connection
PyrrhoCommand CreateCommand()	Create an object for carrying out an Sql command (as in IDbCommand).
Delete(ob)	Delete (drop) a Versioned object from the database. Will block until the connection is available.
<i>list</i> FindAll(cls)	Retrieve all of the instances of the given Versioned class. Will block until the connection is available.
<i>object</i> FindOne(cls,key)	Retrieve the single instance of the given Versioned class with the given key (key is a list) Will block until the connection is available.
<i>list</i> FindWith(cls,cond)	Retrieve a list of instances of the given Versioned class that satisfy the given SQL condition. Will block until the connection is available.
<i>list</i> Get(cls,rurl)	The rurl should be the portion of a REST url following the Role component, targeting class cls in the client application. Will block until the connection is available.
void Open()	Open the channel to the database engine
Post(ob)	The object should be a new Versioned object to be entered in a base table. If autoKey is set key field(s) containing default values (0,"" etc) in ob are overwritten with suitable new value(s). Will block until the connection is available.

Put(ob)	The given object is an updated Versioned object that should be used to update the database. Will block until the connection is available.
PyrrhoConnect(cs)	Create a new PyrrhoConnect with the given connection string. Documentation about the connection string is in section 6.3, except that for Python you should supply the User field.
<i>list</i> Update(cls,w,u)	Specifies a Document update operation on a Versioned class containing documents. Documents matching w are updated according to the operations in u, and the set of modified objects is returned. Will block until the connection is available.

### 6.7.13 PyrrhoDbType

member	int
DBNull	0
Integer	1
Decimal	2
String	3
Timestamp	4
Blob	5
Row	6
Array	7
Real	8
Bool	9
Interval	10
Time	11
Date	12
UDType	13
Multiset	14
Document	16

### 6.7.14 PyrrhoInterval

Attribute	Explanation
<i>int</i> years	The years part of the time interval
<i>int</i> months	The months part of the time interval
<i>long</i> ticks	The ticks part of the time interval

### 6.7.15 (PyrrhoParameter)

Not implemented.

### 6.7.16 (PyrrhoParameterCollection)

Not implemented.

### 6.7.17 PyrrhoReader

Attribute	Explanation
close()	Close the reader
<i>object</i> col(nm)	Get the value in the column with name nm in the current row
<i>bool</i> read()	Get the next row of data into the reader. Return False if none.
PyrrhoRow row	Get the current row
PyrrhoTable schema	Get the schema for the rows
<i>str</i> type(i)	Get the subtype name of val(i)
<i>object</i> val(i)	Get the value in the ith column of the current row

### 6.7.18 PyrrhoRow

Attribute	Explanation
<i>object</i> col(nm)	Get the value in the column with name nm

<i>str</i> check	Get the check string if any
<i>int</i> version	Get the row version if any
<i>str</i> type(i)	Get the subtype name of the value in the ith column
<i>object</i> val(i)	Get the value in the ith column

### 6.7.19 PyrrhoTable

Attribute	Explanation
PyrrhoColumn[] columns	A set of columns
<i>dict</i> cols	Maps column names to column positions
<i>str</i> connectionString	The connection string
PyrrhoReader getReader()	Used for structured values
PyrrhoColumn[] primaryKey	The columns that form the primary key if any
<i>str</i> selectString	The select string that retrieved the table
<i>str</i> tableName	The name of the table

### 6.7.20 PyrrhoTransaction

Attribute	Explanation
commit()	Commit the transaction
rollback()	Roll back the transaction

### 6.7.21 Versioned

Attribute	Explanation
<i>str</i> rowCheck	A string giving the server's row version validator. For Pyrrho this is a comma-separated list of form <i>dbname:depos:lasttrans</i>
<i>str</i> readCheck	A validator to check that the query used to retrieve the data would still return the same results. This is conservative: the validation will fail if the server is unable to provide this guarantee. The server takes account of all data read during the transaction that gave the validator.

### 6.7.22 WebCtrl

This class is similar to WebCtrl in the AWebSvr library. Your controllers will derive from this class. The base class implementations of get, post, put, and delete do nothing and return an empty string.

Attribute	Explanation
<i>bool</i> allowAnonymous()	The base implementation returns false, but anonymous logins are always allowed if no login page is supplied (Pages/Login.htm or Pages/Login.html).
<i>str</i> delete(ws, ps)	Do a Delete for the given WebSvc and parameters
<i>str</i> get(ws, ps)	Do a Get for the given WebSvc and parameters
<i>str</i> post(ws, ps)	Do a Post for the given WebSvc and parameters ([0] is the posted data)
<i>str</i> put(ws, ps)	Do a Put for the given WebSvc and parameters ([0] is the posted data)

### 6.7.23 WebSvc

This class is similar to WebSvc in the AWebSvr library. In this library it is a subclass of BaseHTTPHandler. Your custom web server/service instance(s) will indirectly be subclasses of this class, so will have access to its protected fields and methods documented here. Your subclass will typically organise connection(s) to the DBMS being used. The connection can be for the service or for the request, and so should be set up in an override of the open method, using server or client credentials respectively. (The normal case with the AWebSvr library is to use an embedded DBMS, but this Python API currently supports only OSPSvr, the server edition of Pyrrho.)

Field	Explanation
<i>bool</i> authenticated()	Is called to enforce authentication, if there is a login page and there is no controller for the request or the

	controller's allowAnonymous() returns false. The default implementation populates the WebSvc's user and password and your override can look up the credentials supplied.
close()	Can be overridden to release request-specific resources.
str getData()	Extracts the HTTP data supplied with the request: a URL component beginning with { will be converted to a Document.
log(verb, url, postData)	Write a log entry for the current controller method. The default implementation appends this information to Log.txt together with the user identity and timestamp.
open ()	Can be overridden by a subclass, e.g. to choose a database connection for the current request. The default implementation does nothing.
str password	The client's claimed credentials. See authenticated()
serve()	<i>Calls the requested method using the above templates. Don't call this method directly.</i>
str user	The client's claimed credentials. See authenticated()

### 6.7.24 WebSvr

This class is similar to WebSvr in the AWebSvr library. Your custom web server should be a subclass of WebSvr, and WebSvr is a subclass of WebSvc and hence of BaseHTTPHandler. It defines the URL address (hostname and port number) for the service. If your service is multi-threaded, you can override the Factory method to return a new instance of your WebSvc subclass. Finally, call the Server method to start the service loop.

Field	Explanation
WebSvc factory ()	Can be overridden by a subclass to create a new service instance. The default implementation returns self (for a single-threaded server).
server( address,port)	Starts the server listening on the given address and port.

## 6.8 SWI-Prolog

Pyrrho also comes with some support for SWI-Prolog. This is contained in a module pyrrho.pl which is part of the distribution. The code is at an early stage, so comments are welcome. The following documentation uses the conventions of the SWI-Prolog project.

The interface with SWI-Prolog is implemented by providing SWI-Prolog support for the Pyrrho protocol (section 8.9). The following publicly-visible functions are currently supported:

<b>connect</b> ( -Conn, +ConnectionString )	Establish a connection to the Open Source Pyrrho server. Conn has the form <b>conn</b> (InStream,InBuffer,OutStream,OutBuffer). Codes in OutBuffer are held in reverse order.
<b>sql_reader</b> (+Conn0, -Conn1, +SQLString, -Columns)	Like ExecuteReader on the connection. Conn0. Conn1 is the updated connection. Columns is a list of entries of form <b>column</b> (Name,Type) .
<b>read_row</b> (+Conn0,-Conn1,+Columns,-Row)	Reads the next row (fails if there is no next row) from the connection Conn0. Conn1 is the updated connection. Columns is the column list as returned from sql_reader. Row is a list of corresponding values for the current row.
<b>close_reader</b> (+Conn)	Closes the reader on connection Conn.
<b>field</b> (+Columns,+Row,+Name,-Value)	Extracts a named value from a row. The atom null is used for null values.

## 7. SQL and GQL Syntax for Pyrrho

The following details are provided here for convenience. The syntax shown is merely suggestive in relation to semantics. Full details may be found in SQL2023 and GQL2024, but not all of the details in these specifications are relevant to Pyrrho. In addition, many statements below, such as GRANT OWNER, ALTER .. TO, and SET statements, are Pyrrho specific. To support both SQL and GQL, there are often alternative key words in the syntax rules below (BEGIN|START below is the first such pair). The set of reserved words has been reduced to those of GQL: this makes a big difference to the procedural language from SQL. If newly non-reserved words are given some other meaning, some SQL syntax will not be usable in that context.

In this section capital letters indicate key words: those that are reserved words are shown in a sans-serif font. Tokens such as id, int, string are shown as all lower case words. Mixed case is used for grammar symbols defined in the following productions. The characters = . [ ] { } are part of the production syntax. Characters and multicharacter tokens that appear in the input are enclosed in single quotes, thus '. White space is not allowed within such tokens. Where an identifier representing an object name is required, and the type of object is not obvious from the context, locutions such as *Role\_id* are used.

In SQL and GQL all string literals are enclosed in single quotes, case-sensitive identifiers or containing special characters are enclosed in double quotes.

### 7.1 Statements

There is a fundamental difference between SQL and GQL in the handling of sequences of statements, as occur in procedure specifications. An SQL procedure body typically contains a compound statement that specifies a sequence of statements executed one after another. But in GQL a statement (for example, MATCH) can create a set of bindings that are added to or removed from the current working table: subsequent statements then apply to each row of the current working table. This results in an execution model similar in some ways to Prolog (the FINISH keyword below plays a role similar to the cut in Prolog).

Activity = [AT *Schema\_id*] {[BINDING] BindingVar} Statements .

Statements = Statement { (['|NEXT[YieldClause]]<sup>43</sup> Statement } .

Statement =  
| Alter  
| Assignment  
| (BEGIN|START) TRANSACTION | COMMIT<sup>44</sup>  
| BREAK  
| Call  
| CaseStatement  
| Close  
| NestedStatement  
| CreateClause  
| CursorSpecification  
| Declaration  
| Delete  
| DropStatement  
| Fetch  
| FilterStatement  
| FINISH  
| ForStatement  
| GetDiagnostics  
| Grant  
| IfStatement

<sup>43</sup> Separators are required between SQL procedure statements but often must be omitted between GQL statements. See section 5.2.4. For compliance with GQL see section 7.10 below.

<sup>44</sup> Currently BEGIN/START transaction and COMMIT are supported by the command processor, and available via the PyrrhoConnect API, as ADO.NET and ODBC. Pyrrho does not support nested transactions, and accordingly the only transaction control statement permitted within an SQL statement is ROLLBACK.

```

|      Insert
|      INTERACT Value SET id
|      ITERATE label
|      LEAVE label
|      LetStatement
|      LoopStatement
|      MatchStatement { MatchStatement }[Return]
|      OPTIONAL 45 NestedStatement
|      Open
|      OrderByStatement
|      Rename
|      Repeat
|      Return
|      Revoke
|      ROLLBACK46
|      SelectSingle
|      SET AUTHORIZATION '=' CURATED
|      SET ROLE id
|      SET TIMEOUT '=' int
|      Signal
|      Update
|      WhenStatement
|      While .

```

SET AUTHORIZATION = CURATED is only available to the database owner, and makes all further transaction log information PUBLIC (it is not reversible).

```

Assignment =   SET Target '=' Scalar | Document
               |   SET NodeOrEdge_Target (':'|S) Label_id [Document] .

```

The keyword SET can be omitted for the simplest assignments. Setting the label for a node or edge reference is additional to any labels it already has. The Document options in the syntax allow merging with and updates to property or field values (combining the Document option with setting a label is a Pyrrho extension to GQL and is intended to allow setting of some properties related to the added label).

```

Target =       id { (',' id) | '[' Scalar ']' }
               |   '(' Target { ',' Target } ')' . .

```

Targets which directly contain parameter lists are not supported.

```

Call =         CALL Procedure_id '(' [ Scalar { ',' Scalar } ] ')' [YieldClause]
               |   MethodCall [YieldClause] .

```

Inside a procedure declaration the CALL keyword can be omitted.

```

CaseStatement = CASE Scalar { WHEN Values THEN Statements }[ ELSE Statements ]END CASE
|               CASE { WHEN SearchCondition THEN Statements } [ ELSE Statements ] END CASE .

```

There must be at least one WHEN in the forms shown above.

```

Close =        CLOSE id .

```

```

NestedStatement = Label '('|BEGIN) Statements ('|END) .

```

```

Declaration =  DECLARE id { ',' id } Type
               |   DECLARE id CURSOR FOR CursorSpecification

```

<sup>45</sup> OPTIONAL behaviour is defined for Call and MatchStatement.

<sup>46</sup> By design in Pyrrho, the execution of ROLLBACK causes immediate exit of the current transaction with SQLSTATE 40000. See the previous footnote.



| DECLARE HandlerType HANDLER FOR ConditionList Statement .

Declarations of identifiers, cursors, and handlers are specific to a scope in a SQL routine.

HandlerType = CONTINUE | EXIT | UNDO .

ConditionList = Condition { ',' Condition } .

Condition = ConditionCode | SQLEXCEPTION | SQLWARNING | (NOT FOUND) .

The ConditionCode not\_found is acceptable as an alternative to not found.

Signal = SIGNAL ConditionCode [ SET CondInfo '=' Scalar {'CondInfo'= 'Scalar' }]  
| RESIGNAL [ConditionCode ] [ SET CondInfo '=' Value{'CondInfo'= 'Scalar' } ] .

ConditionCode = *Condition\_id* | SQLSTATE string .

CondInfo = CLASS\_ORIGIN|SUBCLASS\_ORIGIN|CONSTRAINT\_CATALOG|  
CONSTRAINT\_SCHEMA|CONSTRAINT\_NAME|CATALOG\_NAME|SCHEMA\_NAME|  
TABLE\_NAME|COLUMN\_NAME|CURSOR\_NAME|MESSAGE\_TEXT .

GetDiagnostics = GET DIAGNOSTICS Target '=' ItemName { ',' Target '=' ItemName } .

ItemName = NUMBER | MORE | COMMAND\_FUNCTION | COMMAND\_FUNCTION\_CODE | DYNAMIC\_FUNCTION  
| DYNAMIC\_FUNCTION\_CODE | ROW\_COUNT | TRANSACTIONS\_COMMITTED |  
TRANSACTIONS\_ROLLED\_BACK | TRANSACTION\_ACTIVE | CATALOG\_NAME | CLASS\_ORIGIN |  
COLUMN\_NAME | CONDITION\_NUMBER | CONNECTION\_NAME | CONSTRAINT\_CATALOG |  
CONSTRAINT\_NAME | CONSTRAINT\_SCHEMA | CURSOR\_NAME | MESSAGE\_LENGTH |  
MESSAGE\_OCTET\_LENGTH | MESSAGE\_TEXT | PARAMETER\_MODE | PARAMETER\_NAME |  
PARAMETER\_ORDINAL\_POSITION | RETURNED\_SQLSTATE | ROUTINE\_CATALOG | ROUTINE\_NAME |  
ROUTINE\_SCHEMA | SCHEMA\_NAME | SERVER\_NAME | SPECIFIC\_NAME | SUBCLASS\_ORIGIN |  
TABLE\_NAME | TRIGGER\_CATALOG | TRIGGER\_NAME | TRIGGER\_SCHEMA COMMIT\_COMMAND |  
CREATE\_GRAPH\_STATEMENT | CREATE\_GRAPH\_TYPE\_STATEMENT |  
CREATE\_SCHEMA\_STATEMENT | DELETE\_STATEMENT | DROP\_GRAPH\_STATEMENT |  
DROP\_GRAPH\_TYPE\_STATEMENT | DROP\_SCHEMA\_STATEMENT | FILTER\_STATEMENT |  
FOR\_STATEMENT | INSERT\_STATEMENT | LET\_STATEMENT | MATCH\_STATEMENT |  
ORDER\_BY\_AND\_PAGE\_STATEMENT | REMOVE\_STATEMENT | ROLLBACK\_COMMAND |  
SESSION\_CLOSE\_COMMAND | SESSION\_RESET\_COMMAND |  
SESSION\_SET\_BINDING\_TABLE\_PARAMETER\_COMMAND |  
SESSION\_SET\_PROPERTY\_GRAPH\_COMMAND |  
SESSION\_SET\_PROPERTY\_GRAPH\_PARAMETER\_COMMAND | SESSION\_SET\_SCHEMA\_COMMAND |  
SESSION\_SET\_TIME\_ZONE\_COMMAND | SESSION\_SET\_VALUE\_PARAMETER\_COMMAND |  
SET\_STATEMENT | START\_TRANSACTION\_COMMAND .

SQLSTATE strings are 5 characters in length, comprising a 2-character class and a 3 character subclass. See the table in section 8.1.1.

Fetch = FETCH [How] *Cursor\_id* INTO VariableRef { ',' VariableRef }

| FETCH (' Query ') FOR '*d\_string*' [FIRST '*b\_string*'] DO '*r\_string*' [LAST '*e\_string*'] .

The second syntax here allows specification of a Query for the current database whose results are used to construct a command script for another database on this server. The FOR part specifies the name of this database, the FIRST part an optional initial command, the DO part is the template for a statement where column names preceded by \$ (case-sensitive) are replaced by the corresponding values in the row, and the LAST part an optional final command (see section 3.8).

How = NEXT | PRIOR | FIRST | LAST | ((ABSOLUTE | RELATIVE) Value ) .

In Pyrrho, binding variables are inferred in the MatchStatement by usage of an undefined identifier, but may also be defined by BindingVar, LetStatement and the second form of the ForStatement below.

BindingVar = GraphVar | TableVar | ValueVar .

GraphVar = BindType '=' (Value|CURRENT\_GRAPH|CURRENT\_PROPERTY\_GRAPH) .

TableVar = TABLE *Unbound\_id* BindType '=' Rowset\_Value .

ValueVar = VALUE *Unbound\_id* BindType '=' Value .

BindType = [[:'|TYPED] Type] .

```

ForStatement = Label FOR [ For_id AS ][ id CURSOR FOR ] RowSetSpec DO Statements END FOR
              [ Label_id ]
              |      FOR Unbound_id IN RowSet_Value [WITH (ORDINALITY|OFFSET) id ] .

```

$$\text{LetStatement} = \text{LET LetDef} [\text{' , ' LetDef}].$$

```
LetDef =      Unbound_id '=' Value
           | ValueVar .
```

FilterStatement = FILTER (WhereClause|*Boolean\_Value*) .

```

OrderByStatement= OrderByClause [OffsetClause][LimitClause]
                |      OffsetClause
                |      LimitClause .

```

These three statements operate on the current rowset result.

IfStatement = IF BooleanExpr THEN Statements { ELSEIF BooleanExpr THEN Statements }  
[ ELSE Statements ] END IF .

WhenStatement = WHEN BooleanExpr THEN Statement [ ELSE Statement ] .

Label = [ label ':' ] .

LoopStatement = Label LOOP Statements END LOOP .

$$\text{MatchStatement} = [\text{UseGraph}] (\text{MATCH}|\text{WITH}) [\text{Schema}] [\text{Truncation}] \text{Match} \{', \text{Match}\} [\text{WhereClause}].$$

The `Match` statement computes a rowset of bindings for `unbound` for which the `Match` is found in all or a selected set of graphs in the database (see `MatchMode` below). Bindings have effect only within the scope of the current statement and its dependents and are removed before the next statement. The binding table does not contain duplicate rows. Nodes, edges, element type labels, property names, and property values can be bound. See also the `INSERT Graph` statement in section 7.5.

$$\text{Truncation}^{47} = \text{TRUNCATING } (' \text{TruncationSpec} \{ ' , ' \text{TruncationSpec} \} ') .$$
$$\text{TruncationSpec} = [EdgeType \text{ id}] \text{ '(' } Ordering \text{ Value} \text{ ')} \text{ '=' } int \text{ Value} .$$

The Truncation clause defines an upper bound for the number of edges to be traversed from a node in a step of the match process. The limit can be applied differently to specific edge types and for a specified ordering of possible edges. Limits specified for supertypes of selected edges are also applied, as is the unnamed limit if present. The ordering part is a string expression parsed at run time with syntax `OrderSpec{'', 'OrderSpec'}`.

```
Match = (MatchMode [id '='] MatchNode) {'| Match} .
```

$$\text{MatchNode} = (' \text{MatchItem} ') \{ (\text{MatchEdge} | \text{MatchPath}) \text{MatchNode} \}.$$

```
MatchEdge = '-' MatchItem '-'> | '<-' MatchItem ']-'.
```

$$\text{MatchItem} = [\text{id} \mid \text{Node Value}] [\text{GraphLabel}] [\text{Document} \mid \text{Where}] .$$

MatchPath = '[' Match ']' MatchQuantifier .

$$\text{MatchQuantifier} = '?' | '*' | '+' | '\{int, [int]\}'.$$

MatchMode = [TRAIL|ACYCLIC|SIMPLE] [SHORTEST|LONGEST|ALL|ANY].

The MatchMode<sup>48</sup> controls how repetitions of path patterns are managed in the graph matching mechanism. A MatchPath creates lists of values of bound identifiers in its Match. By default, binding rows that have already occurred in the match are ignored<sup>49</sup>, and paths that have already been listed in a

<sup>47</sup> Truncation is a Pyrrho addition inspired by the LDBC Financial Benchmark.

<sup>48</sup> The GQL specification specifies many path modes. This selection is similar to those in [Francis 2023].

<sup>49</sup> This is a different convention from the behaviour of SELECT.

quantified graph are not followed. The MatchMode modifies this default behaviour: TRAIL omits paths where an edge occurs more than once, ACYCLIC omits paths where a node occurs more than once, SIMPLE looks for a simple cycle. The last three options apply to MatchStatements that do not use the comma operator, and select the shortest match, all matches or an arbitrary match.

```

Open =          OPEN id .

Repeat =        Label REPEAT Statements UNTIL BooleanExpr END REPEAT .

Return =        YIELD id {',' id}
                |      RETURN Value [AS id] {',' Value [AS id]} .

SelectSingle =  SELECT [ALL|DISTINCT] SelectItems INTO TargetList TableExpression .

TargetList =    VariableRef {',' VariableRef } .

While =         Label WHILE SearchCondition DO Statements END WHILE .

UserFunctionCall = id '(' [ Scalar {',' Scalar } ] ')'.

MethodCall =    Scalar '.' Method_id '(' [ Scalar {',' Scalar } ] ')'
                |      '(' Scalar AS Type ')' '.' Method_id '(' [ Scalar {',' Scalar } ] ')'
                |      Type '::' Method_id '(' [ Scalar {',' Scalar } ] ')'.

```

## 7.2 Data Definition

As is usual for a practical DBMS, Pyrrho's Alter statements are richer than SQL2023. In executable code, Value can often replace **id** or tokens in the syntax defined in this section.

```

Alter =         ALTER DOMAIN id AlterDomain
                |      ALTER FUNCTION id '(' Parameters ')' RETURNS Type AlterBody
                |      ALTER PROCEDURE id '(' Parameters ')' AlterBody
                |      ALTER Method AlterBody
                |      ALTER TABLE id AlterTable
                |      ALTER TYPE id AlterType
                |      ALTER VIEW id AlterView .

```

```
Method = MethodType METHOD id '(' Parameters ')' [RETURNS Type] [FOR id].
```

```
Parameters = Parameter {',' Parameter } .
```

```
Parameter = id Type .
```

The specification of IN, OUT, INOUT and RESULT is not (yet) supported.

```
MethodType =    [ OVERRIDING | INSTANCE | STATIC | CONSTRUCTOR ] .
```

The default method type is INSTANCE. All OVERRIDING methods are instance methods.

```

AlterDomain =   SET DEFAULT Default
                |      DROP DEFAULT
                |      TYPE Type
                |      AlterCheck .

```

```
AlterBody =     AlterOp {',' AlterOp } .
```

```

AlterOp =       TO id
                |      Statement
                |      [ADD|DROP] { Metadata } .

```

```

Default =       Literal | DateTimeFunction | CURRENT_USER | CURRENT_ROLE | NULL |
                ARRAY('') | MULTISET('') .

```

```
AlterCheck =      (ADD|DROP) CheckConstraint
                  |      [ADD|DROP] { Metadata }
                  |      DROP CONSTRAINT id .
```

Note that anonymous constraints can be dropped by finding the system-generated id in the Role\$TableCheck, Role\$ColumnCheck or Role\$DomainCheck table (see section 8.1).

CheckConstraint = [ CONSTRAINT id ] CHECK '(' SearchCondition ')'

```
AlterTable =      TO id
                  |      ADD ColumnDefinition
                  |      ALTER [COLUMN] id AlterColumn
                  |      DROP [COLUMN] id RemoveAction
                  |      (ADD|DROP) (TableConstraintDef | VersioningClause)
                  |      SET Cols REFERENCES id [ Cols ] [ USING (id|('Values')) ] ReferentialAction
                  |      ALTER PERIOD id TO id
                  |      DROP TablePeriodDefinition RemoveAction
                  |      Classification | Enforcement
                  |      AlterCheck
                  |      [ADD|DROP] Metadata { Metadata } .
```

```
AlterColumn =     TO id
                  |      POSITION int
                  |      SET ((NOT NULL)|ColumnConstraint )
                  |      DROP ((NOT NULL)|ColumnConstraint ) RemoveAction
                  |      AlterDomain
                  |      Classification
                  |      GenerationRule
                  |      [ADD|DROP RemoveAction] { Metadata } .
```

When columns are renamed, Pyrrho cascades the change to SQL referring to the columns.

```
AlterType =       TO id
                  |      ADD ( Field | Method )
                  |      DROP ( Field_id | Routine) RemoveAction
                  |      Classification
                  |      Representation ReferentialAction
                  |      (ADD|SET) UNDER Type_id{'Type_Id'}50 ReferentialAction
                  |      DROP UNDER RemoveAction
                  |      [DROP] { Metadata }
                  |      ALTER Field_id AlterField .
```

Other details of a Method can be changed with the ALTER METHOD statement (see Alter above). A sensitive type cannot be altered to a non-sensitive type.

Field = id Type [DEFAULT Value] Collate {Metadata} .

```
AlterField =      TO id
                  |      [DROP] RemoveAction { Metadata }
                  |      TYPE Type
                  |      SET DEFAULT Value
                  |      DROP DEFAULT .
```

---

<sup>50</sup> Multiple inheritance added for GQL.

```
AlterView =      SET SOURCE TO RowSet
                |      TO id
                |      [ADD|DROP] { Metadata }.
```

```
Metadata =CAPTION | LEGEND | X | Y | JSON | CSV | ETAG | ID | LEAVING | ARRIVING
          | MILLI | MONOTONIC | SCHEMA | ((INVERTS|KEY) id)
          | ([URL | MIME | SQLAGENT | USER] string) | | |
          | ((HISTOGRAM | LINE | PIE | POINTS | NODE) ['(' id ',' id ')']) | iri |
          | ((PREFIX|SUFFIX) id) | NODETYPE |
          | (EDGETYPE '(' Connections ')') |
          | ((CARDINALITY|MULTIPLICITY) '('(int|'*'| (int '..' (int|'*'))')' ) |
          | SENSITIVE | SECURITY Level.
```

The Metadata syntax is a Pyrrho extension. Many of the options affect query output for a role in Pyrrho's Web service and most are not reserved words. By default, query output in the Web service is an HTML table. Histogram, Legend, Line, Points, Pie, Node (for table, view or function metadata, can optionally supply column ids for X and Y), Caption, X and Y (for column or subobject metadata) specify JavaScript added to HTML output to draw the data visualisations specified. The string is usually for a description, and for X and Y columns is used to label the axes of charts. For RestViews, url and other properties<sup>51</sup> for the view are given as string literals. For INVERTS the id should be the name of the function being inverted<sup>52</sup>, and KEY is an assertion about the object schema key (for v.7.01). PREFIX and SUFFIX are metadata for subtype declarations: in SQL the id is supplied as prefix/suffix to a value of the supertype to construct a value of the subtype and is used without double-quotes to decorate output.

NODETYPE and EDGETYPE are metadata for user defined types and can be specified only in CREATE TYPE or ALTER statements. CARDINALITY can be specified as a constraint for array, set, multiset, and edge types, and MULTIPLICITY for REFERENCES in foreign key definitions. \* here indicates no limit (this is the default): the default multiplicity for edge connections is 1..\*, while for REFERENCES and OPTIONAL edge connections it is 0 .. \* <sup>53</sup>.

Classification levels (*level\_id* = D,C,B or A) can only be specified by the database owner: D is the default. When applied to users or permissions, these are clearance levels, to database objects, classification levels. See section 3.4.2. A Type can be declared sensitive: this property is silently inherited by values, columns, tables, and views. A non-sensitive object cannot receive a sensitive value.

```
Level = LEVEL level_id [ '-' level_id ] [GROUPS {id}] [REFERENCES {id}] .
```

```
AddPeriodColumnList = ADD [COLUMN] Start_ColumnDefinition ADD [COLUMN]
End_ColumnDefinition .
```

```
Create = CREATE ROLE id [Description_string]
        | CREATE DOMAIN id [AS] DomainDefinition [Classification]
        | CREATE FUNCTION id '('Parameters')' RETURNS Type {Metadata} Statement54
        | CREATE [OR REPLACE] [PROPERTY] GRAPH [SchemaDetails] [IF NOT EXISTS] GraphDetails55
        | CREATE [OR REPLACE] [PROPERTY] GRAPH TYPE [IF NOT EXISTS] GraphTypeDetails
        | CREATE ORDERING FOR UDType_id (EQUALS ONLY|ORDER FULL) BY Ordering
        | CREATE PROCEDURE [catref]56 id '(' Parameters ')'Statement
        | CREATE Method Statement
```

<sup>51</sup> ETAG means RFC 7232, MILLI means RFC 7232 but with a 3-digit fractional part for seconds (i.e. not quite RFC 7231 format). RESTViews can be declared ENTITY.

<sup>52</sup> Pyrrho uses such information automatically in the implementation of updatable views and joins.

<sup>53</sup> If the minimum number of edges is specified >0, this is a constraint on the connected node type (like a non-null foreign key).

<sup>54</sup> Functions that return tables have an explicit row type, so the table value returned by the Statement should explicitly alias columns to match the returns clause in case table columns are changed later.

<sup>55</sup> CREATE GRAPH/GRAPH TYPE/SCHEMA syntax is available in Pyrrho as an alternative to direct graph creation using INSERT/CREATE and/or creating types with graph metadata.

<sup>56</sup> GQL catalog reference. See under GraphDetails later in this subsection.

```

| CREATE SCHEMA [IF NOT EXISTS] SchemaDetails .
| CREATE TABLE id TableContents [Classification][Enforcement] {Metadata}
| CREATE TRIGGER id (BEFORE|AFTER) Event ON id [ RefObj ] Trigger
| CREATE TYPE id ((UNDER id{'id'}57)|AS Representation) {CheckConstraint} [Classification]
| [ Method {' Method' } ]
| CREATE VIEW id ViewDefinition
| [UseGraph] CREATE Graph {'Graph'}58
| CREATE GRAPH [SCHEMA] [CONTENT TYPES Contents][NODE TYPES Nodes][EDGE TYPES Edges] .

```

Method bodies in SQL2023 are specified by CREATE METHOD once the type has been created...In Pyrrho types UNDER or Representation must be specified (not both). Classification and Enforcement can only be set by the database owner (see section 3.4.2).

Enforcement = SCOPE [READ] [INSERT] [UPDATE] [DELETE] .

Representation = (StandardType|Table\_id|(' Field {' Field }')) .

DomainDefinition = StandardType [DEFAULT Default] { CheckConstraint } Collate .

Ordering = (RELATIVE|MAP) WITH Routine  
STATE .

TableContents = (' TableClause {' TableClause } ') { VersioningClause }  
OF Type\_id [ (' TypedTableElement {' TypedTableElement } ') ] .

VersioningClause = WITH SYSTEM VERSIONING .

TableClause = ColumnDefinition {Metadata} | TableConstraint | TablePeriodDefinition .

```

ColumnDefinition = id Type [DEFAULT Default] {ColumnConstraint|CheckConstraint} Collate
{Metadata}
| id (TO|FROM|WITH) Type_id59
| id GENERATED ALWAYS AS ('Value')
| id GENERATED ALWAYS AS ROW (START| END) .

```

ColumnConstraint = [CONSTRAINT id ] ColumnConstraintDef .

```

ColumnConstraintDef = NOT NULL
| PRIMARY KEY
| REFERENCES id [ Cols ] [USING (id|('Values'))] { ReferentialAction }
| UNIQUE
| DEFAULT Value
| Classification .

```

The Using expression here is an extension to SQL2023 behaviour, allowing a row expression or the name of an adapter function. See section 5.2.2. A column default value overrides a domain default value.

TableConstraint = [ CONSTRAINT id ] TableConstraintDef .

```

TableConstraintDef = UNIQUE Cols
| PRIMARY KEY Cols
| FOREIGN KEY Cols REFERENCES id [Cols] [USING (id|('Values'))]
{ ReferentialAction } .

```

The Cols of a foreign key are allowed to be SET types. The Using expression here is an extension to SQL2023 behaviour allowing a row expression or the name of an adapter function. See section 5.2.2.

<sup>57</sup> Multiple inheritance added for GQL.

<sup>58</sup> The syntax of Graph is in section 7.5: the CREATE Graph syntax is an alternative to INSERT Graph and has the same semantics.

<sup>59</sup> See section 5.2.1 and footnote to that section.

TablePeriodDefinition= PERIOD FOR PeriodName '(' *Column\_id* ',' *Column\_id* ')'

PeriodName = SYSTEM\_TIME | id .

TypedTableElement = ColumnOptionsPart | TableCnstraint .

ColumnOptionsPart = id WITH OPTIONS '(' ColumnOption {' ColumnOption } )'

ColumnOption = (SCOPE *Table\_id*) | (DEFAULT Value) | ColumnConstraint .

Values = Value {' Value } .

Cols = '(' ColRef { ' ColRef } ' PERIOD *ApplicationTime\_id* ] )'.

The period syntax here can only be used in a foreign key constraint declaration where both tables have application time period definitions, and allows them to be matched up.

ColRef = *Column\_id* { ' *Field\_id* [AS Type]}.

The *Field\_id* syntax is Pyrrho specific and can be used to reference fields of structured types or documents.

ReferentialAction = ON (DELETE|UPDATE) (CASCADE| SET (DEFAULT|NULL)|RESTRICT) .

The default ReferentialAction is RESTRICT.<sup>60</sup>

ViewDefinition = [ViewSpec] AS RowSetSpec {Metadata} .

The resulting view may be updatable using UPDATE, DELETE and INSERT statements.

ViewSpec = Cols | OF *Type\_id* | OF Representation .

The third syntax here is to define the contents of RESTViews.

TriggerDefinition = TRIGGER id (BEFORE|(INSTEAD OF)| AFTER) Event ON id [RefObj] Trigger .

Event = INSERT | DELETE | (UPDATE [ OF id { ' id } ] ) .

RefObj = REFERENCING { (OLD|NEW)[ROW|TABLE][AS] id } .

In this syntax, the default is ROW; TABLE cannot be specified for a BEFORE trigger; OLD cannot be specified for an INSERT trigger; NEW cannot be specified for a DELETE trigger.

Trigger = FOR EACH (ROW|STATEMENT [DEFERRED]) [TriggerCond] (Statement | (BEGIN ATOMIC Statements END)) .

TriggerCond = WHEN '(' SearchCondition )' .

DropStatement = DROP DropObject RemoveAction .

```
DropObject =  ObjectName
              |  ORDERING FOR id
              |  TRIGGER id ON id
              |  ROLE id
              |  SCHEMA [IF EXISTS] Path_Value id
              |  TRIGGER id .
```

RemoveAction = | RESTRICT | CASCADE .

The default RemoveAction is RESTRICT.

Rename = SET ObjectName TO id .

---

<sup>60</sup> The SQL standard specifies that the default should be NO ACTION, but such an option is not available in Pyrrho.

UseGraph = USE [SCHEMA] [catref] id.

SCHEMA if present in a use clause indicates that the following statement is focussed at the graph schema level. A GQL catalog reference (catref) is an unquoted string chain with separators '/' and '.' and no embedded spaces. Pyrrho extends this by allowing any url string<sup>61</sup>. The keyword HOME\_SCHEMA is replaced in Pyrrho by the empty string, and the HOME\_PROPERTY\_GRAPH or HOME\_GRAPH is the database. If catref is present it sets the CURRENT\_GRAPH and CURRENT\_SCHEMA.

SchemaDetails = [catref] id.

GraphDetails = [catref] id (OpenGraphType|OfGraphType)[AS COPY OF Graph] .

GraphTypeDetails = [catref] id [AS] COPY OF id | LIKE Graph | [AS id] .

OpenGraphType = [::]|TYPED] ANY [[PROPERTY] GRAPH] .

OfGraphType = ANY [[PROPERTY] GRAPH]  
| LIKE Graph  
| ([::]|TYPED] (GraphType\_id[[PROPERTY] GRAPH] '{ ElementList }')).

GraphTypeDef = '{ ElementList }' .

ElementList = (NodeTypeDetails|EdgeTypeDetails) {Metadata}<sup>62</sup> '{ ElementList}' .

NodeTypeDetails = [Node [TYPE] id] (' Filler')  
| Node [TYPE] Filler [AS id].

Filler = [Alias\_id] [Labels] ['=>' [Labels]]['{ Properties}'] .

Labels = LABEL id | (LABELS[':|IS]) id '{&' id} .

Properties = (id [::]|TYPED] Type) '{, Properties}' .

Node = NODE | VERTEX .

EdgeTypeDetails = [Direction Edge [TYPE] id] EdgePattern  
| Direction Edge [TYPE] (id|Filler) EndPoints.

EndPoints = CONNECTING '(' NodeTypeDef ('<-'|>|TO|~) NodeTypeDef) ' .

EdgePattern = ('(NodeTypeDef|Filler)' '-[Filler]->' ('(NodeTypeDef|Filler)'  
| ('(NodeTypeDef|Filler)' '<-[' Filler ']-' ('(NodeTypeDef|Filler)'  
| ('(NodeTypeDef|Filler)' '~[' Filler '~' ('(NodeTypeDef|Filler)') .

NodeTypeDef = NodeType\_id {Metadata} '{| NodeType\_id {Metadata} }' .

EndPoints = CONNECTING '(' Connections ')' .

Connections = [FROM Connectors ][WITH Connectors][TO Connectors] .

Connectors = Connector '{, Connector}' .

Connector = [Connector\_id=] Type\_id '{| Type\_id } [SET] Metadata .

For Connector\_id see below.

Edge = EDGE | RELATIONSHIP .

Direction = DIRECTED | UNDIRECTED .

Contents = id ['=>' id] [Contents] .

Nodes = '(' id ')' ['|, Node]' .

Edges = ('(NodeType\_id) ([-Connector\_id->] (<-[Connector\_id-]) | (~[Connector\_id~])) ('(NodeType\_id)' [Edges] .

The optional Connector\_id supports identifiers for connectors. FROM, TO and WITH connectors are distinguished by arrow tokens and their default identifiers have form FROMnn (resp., TOnn, WITHnn): connectors are also distinguishable when they target distinct node types. Connector identifiers become property names in edge types. Also see section 7.5.

## 7.3 Access Control

Grant = GRANT Privileges TO GranteeList [ WITH GRANT OPTION ]  
| GRANT Role\_id {', Role\_id } TO GranteeList [ WITH ADMIN OPTION ]

<sup>61</sup> HTTP access using such a url depends on privileges and content on the HTTP server referenced.

<sup>62</sup> For an edge type CARDINALITY specifies the maximum number of edges that can connect to a given pair of endpoints (default is no limit).



| GRANT Level TO *user\_id* .

Grant can only be used in single-database connections (section 3.4). For roles see section 5.5. Clearance levels (D to A) can only be applied to users by the database owner (D is the default).

Revoke = REVOKE [GRANT OPTION FOR] Privileges FROM GranteeList  
 | REVOKE [ADMIN OPTION FOR] *Role\_id* { ',' *Role\_id* } FROM GranteeList .

Revoke can only be used in single-database connections. Revoke withdraws the specified privileges in a cascade, irrespective of the origin of any privileges held by the affected grantees: this is a change to SQL2023 behaviour. (See also sections 5.5 and 7.13.)

Privileges = ObjectPrivileges [ON] ObjectName .

ObjectPrivileges = ALL PRIVILEGES | Action { ',' Action } .

Action = SELECT [ '(' id { ',' id } ')' ]  
 | DELETE  
 | INSERT [ '(' id { ',' id } ')' ]  
 | UPDATE [ '(' id { ',' id } ')' ]  
 | REFERENCES [ '(' id { ',' id } ')' ]  
 | USAGE  
 | TRIGGER  
 | EXECUTE  
 | METADATA  
 | GRAPH  
 | SCHEMA  
 | OWNER .

The graph and schema privileges can be granted to a role by their owner: the nominated graph/schema becomes the home graph/schema for the role (accessibility is a type privilege). The owner privilege (Pyrrho-specific) can only be granted by the owner of the object (or the database) and results in a transfer of ownership of that object to a single user or role (not PUBLIC). Ownership always implies grant option for the owner privilege. References here can be to columns, methods, fields or properties depending on the type of object referenced by the objectname (usage is for domains).

ObjectName = TABLE id  
 | DOMAIN id  
 | TYPE id  
 | Routine  
 | VIEW id .

GranteeList = PUBLIC | Grantee { ',' Grantee } .

Grantee = [USER] id  
 | ROLE id .

See section 5.5 for the use of roles in Pyrrho.

Routine = PROCEDURE id [DataTypeList]  
 | FUNCTION id [DataTypeList]  
 | [ MethodType ] METHOD id [DataTypeList] [FOR id ]  
 | TRIGGER id .

DataTypeList = '('Type, {',' Type } )' .

## 7.4 Type

Type = StandardType | DefinedType | *Domain\_id* | *Type\_id* | CollectionType | SimpleRference .

StandardType = (BOOL|BOOLEAN) | CharacterType | LobType | NumericType | FloatType | [SIGNED|UNSIGNED] IntegerType | DateTimeType | IntervalType | VECTOR | POSITION | DOCUMENT | DOCARRAY | CHECK.

The last three types are Pyrrho-specific: Document is as in <http://bsonspec.org>, DocArray is for the array variant used in Bson. See also sec 7.6. Documents and DocArrays are transmitted to clients as subtypes of byte[] data, using Bson format. Check is an Rvv cookie transmitted to clients as a string, such cookies are IComparable and can be merged using +. All four types have automatic conversion from strings: Json to Bson for Document and DocArray. Documents are considered equal if corresponding fields match<sup>63</sup>.

CharacterType = (STRING | ([NATIONAL] CHARACTER) | CHAR | NCHAR | VARCHAR) [VARYING] ['(int ') [CHARACTER SET id ] Collate .

The keywords are synonyms in Pyrrho for a variable length Unicode string<sup>64</sup>.

Collate = [ COLLATE id ] .

There is no need to specify COLLATE UNICODE, since this is the default collation. COLLATE UCS\_BASIC is supported but deprecated. Other CultureInfo strings (in double quotes) are supported depending on the current version of the .NET libraries: since Windows 10 any valid BCP-47 language tag can be used. This determines comparison of strings and conversion from dates etc.

FloatType = (FLOAT | FLOAT16 | FLOAT32 | FLOAT64 | FLOAT128 | FLOAT256 | REAL|DOUBLE PRECISION) ['(int','int')] .

IntegerType = INT | INT8 | INT16 | INT32 | INT64 | INT128 | INT256 | INTEGER | INTEGER8 | INTEGER16 | INTEGER32 | INTEGER64 | INTEGER128 | INTEGER256 | BIGINT | SMALLINT .

By default, Pyrrho does not enforce length or radix-2 precision for intermediate results: an exception is only raised for overflow when data is to be returned to the client.

LobType = ([NATIONAL] CHARACTER |BINARY) LARGE OBJECT | BINARY | BYTES | VARBINARY | BLOB | CLOB | NCLOB .

National is ignored, the character large object types are regarded as equivalent to STRING since they represent unbounded character strings, and BLOB and BINARY LARGE OBJECT is the same as BINARY, BYTES, and VARBINARY (which are also equivalent).

NumericType = (NUMERIC|DECIMAL|DEC) ['(int','int')] .

The names here are regarded as equivalent in Pyrrho.

DateTimeType = (DATE | TIME | TIMESTAMP) ([IntervalField [ TO IntervalField ]] | ['( int ')]).

The use of IntervalFields when declaring DateTimeType is an addition to the SQL standard.

IntervalType = INTERVAL IntervalField [ TO IntervalField ] .

IntervalField = YEAR | MONTH | DAY | HOUR | MINUTE | SECOND ['( int ')'] .

DefinedType = (ROW|TABLE) Representation

| DataTypeList .

The TABLE alternative here is a Pyrrho extension to SQL2023, corresponding to the difference between a row and a rowset. DataTypeList is an anonymous row type (no column names), also specific to Pyrrho.

CollectionType = Type (LIST|VECTOR|ARRAY|SET|MULTISET).

<sup>63</sup> This is extremely useful though counter-intuitive, as the empty document is “equal” to every other document!

<sup>64</sup> The length integer is a constraint during query processing, but strings in the physical database are not truncated.

LIST and SET are added to the standard types; and expressions whose type is a list, vector, array, set or multiset of scalar type are scalars.

SimpleReference = (FROM|TO|WITH) *Type\_id* .

This is a special syntax for defining simple edges in a NodeType. See the syntax for ColumnDefinition.

## 7.5 RowSet

Query = Match | LetStatement | ForStatement | FilterStatement | OrderByStatement  
 | CallStatement | SelectStatement | TABLE *id* | Query OTHERWISE Query  
 | Query (UNION|INTERSECT|EXCEPT) [DISTINCT|ALL] Query .

DISTINCT is the default and discards duplicates from both operands.

RowSet = TableReference  
 | DEFAULT VALUES  
 | GET [USING *Table\_id*] .

The domain of DEFAULT VALUES and GET must be constrained by the context. The GET syntax here is for the RestView feature of Pyrrho<sup>65</sup>.

Insert = INSERT INTO *Table\_id* [ Cols ] RowSet [Classification]  
 | [UseGraph] INSERT [SCHEMA [SchemaDetails]] Graph {'Graph'} [THEN Statement]  
 | [UseGraph] INSERT SCHEMA [SchemaDetails] '['GraphItem']' .

In the first version of the Insert statement, the VALUES keyword is mandatory if you are providing an explicit TableValue (see section 7.7). Only the database owner, as security manager, is permitted to provide a classification: otherwise, if the insert succeeds, the classification of the row is determined by the clearance of the current user, and may differ from the classification of other rows in the table. The column list if present names the columns from the table for which values are provided: otherwise values must be provided for all columns. The second version of INSERT adds one or more nodes or edges to a graph according to the given pattern, creating new base tables (as node and edge types) and table rows (as nodes and edges) and their properties as required; id aliases for new elements are bound for the duration of the current statement and may be referenced in the dependent statement if present. The keyword SCHEMA if present implies that the graphs will create or modify element types<sup>66</sup>, and the third version of the syntax allows creation of new edge labels.

Graph = [Node] Path {'Node Path' } .

Path = { Edge Node } .

Node = '(' GraphItem ')' .

Edge = arrow GraphItem arrow .

The arrow tokens come in pairs for before and after a GraphItem and may not contain embedded spaces. The tilde arrows are for undirected edges. Arrows can be optionally labelled with an id as shown.

Arrow before GraphItem	Arrow after GraphItem
'-' [ id '-' ]	[ '-' id ]' -> '
'<-' [ id '-' ]	[ '-' id ]' - '
'~' [ id '~' ]	[ '~' id ]' ~ '
'~' [ id '~' ]	[ '~' id ]' ~> '
'<~' [ id '~' ]	[ '~' id ]' ~ '

GraphItem = [Node\_Value] ['GraphLabel'] [Document] .

<sup>65</sup> For AS GET url, the url string is supplied in the Metadata syntax. Explicit column names can be specified using the extended ViewSpec in this section. For AS GET USING id the specified USING table gives some data identifying contributing servers including a primary key and the URL of the contribution as the last column. The row type of the Representation should consist of the columns of the USING table (except the last), and the remaining columns must match the contributed data.

<sup>66</sup> In a graph pattern at schema level, {a:b} specifies a property whose name is a and has *data type* b.

GraphLabel = ['!'] (id | *Label\_Value* | '%' | (' GraphLabel ')) {LabelOp GraphLabel} .

LabelOp = '&' | '|' | ('=>' | ':') | ':'<sup>67</sup>.

In Insert/Create Graph statements the only LabelOps permitted are ':' and '&', and the use of '|' and parentheses is not allowed. ':' is not allowed *within* a GraphLabel in MatchStatements.

Update = UPDATE *Target\_id* Assignment [WhereClause] [ReferentialAction]  
| (UPDATE|SET) Assignment [WhereClause] [ReferentialAction].

The first version requires a table or rowset reference to give the context. The second version is used in the context of a Match statement.

Delete = DELETE FROM *Target\_id* [WhereClause] [RemoveAction]  
| [DETACH|NODETACH] (DELETE|REMOVE) *Node\_Value* [WhereClause] .

The second version is used in the context of a Match statement.

In these four definitions *Target* can be a table or view.

RowSetSpec = SELECT [ ALL | DISTINCT ] SelectList TableExpression [FOR UPDATE] .

FOR UPDATE is ignored by Pyrrho, and is allowed in the syntax only for compatibility with other DBMS.

SelectList = SelectItem { ',' SelectItem } .

SelectItem = [Col '.']\* | Scalar [AS id ] | RowValue '.' '\*' [AS Cols].

Alias = [[AS] id [ Cols ]].

The id is an alias for the referenced table, and the column list if present selects columns from it.

TimePeriodSpecification = AS OF Scalar  
| BETWEEN [ASYMMETRIC|SYMMETRIC] Scalar AND Scalar  
| FROM Scalar TO Scalar .

This syntax is slightly more general than in SQL2023.

JoinedTable = TableReference CROSS JOIN TableFactor  
| TableReference NATURAL [JoinType] JOIN TableFactor  
| TableReference[JoinType]JOIN TableReference ((USING '(Cols'))(ON SearchCondition)) .

JoinType = INNER | ( LEFT | RIGHT | FULL ) [OUTER] .

SearchCondition = BooleanExpr .

OrderByClause = ORDER BY OrderSpec { ',' OrderSpec } .

OrderSpec = Scalar [ ASC | DESC ] [ NULLS ( FIRST | LAST )].

The default order is ascending, nulls first.

FetchFirstClause = FETCH FIRST [ int ] (ROW|ROWS) ONLY .

OffsetClause = (OFFSET|SKIP) *int\_Value* .

LimitClause = LIMIT *int\_Value*.

---

<sup>67</sup> The use of : as an operator within label expressions is a Pyrrho extension to GQL, with similar semantics to =>. There should be no white space between = and > in the implies symbol.

## 7.6 Scalar Expressions

Value = Scalar | RowValue<sup>68</sup> | ListValue | ArrayValue | TableValue | Treatment .

Treatment = TREAT '(' Value AS Type ')' .

The SQL standard requires the target of a TREAT expression to be a structured type. Pyrrho does not.

Scalar =

- Literal
- Scalar BinaryOp Scalar
- '-' Scalar
- '(' Scalar ')'
- Scalar Collate
- Scalar '[' Scalar ']'
- Scalar AS Type
- ColumnRef
- VariableRef
- Scalar\_Subquery
- (SYSTEM\_TIME|Period\_id|(PERIOD('Scalar','Scalar')))
- VALUE
- Scalar '.' Field\_id
- Scalar\_MethodCall
- NEW Constructor\_MethodCall
- Scalar\_FunctionCall
- Document
- DocArray
- Graph
- (MULTISET|SET|ARRAY) (('Value { ',' Value } '])| Table\_Subquery)
- ARRAY '[' int\_Value '=' Value { ',' int\_Value '=' Value } ']'
- CASE Value {WHEN Value {',' Value} THEN Value } [ELSE Value ] END
- CASE {WHEN BooleanExpr THEN Value } [ELSE Value ] END
- USER
- CURRENT\_ROLE .

The VALUE keyword is used in Check constraints, A scalar subquery must have exactly one column and return a single value. The explicit list option for multiset, set, and array cannot directly contain table expressions. The first syntax for ARRAY uses default subscripts 0,1,..., while the second allows the subscript values to be given explicitly. A scalar MethodCall or FunctionCall does not return a table. Collate if specified applies to an immediately preceding expression, affecting comparison operands etc. The AS syntax in Scalar AS Type is allowed only in parameter lists and methodcalls.

Collate = [Schema\_id '.' ] COLLATE id .

Document =

- '{ [ id ':' DocValue { ',' id ':' DocValue } ] }'
- '{ id (::'|TYPED) Type { ',' id (::'|TYPED) Type } }' .

The first syntax is not legal in INSERT SCHEMA: keynames are case-sensitive and should be enclosed in single or double quotes. Fields can be extracted from and added to document nodes using dot or subscript notation (to delete a field update the parent node). The second syntax is for INSERT SCHEMA only.

DocArray = '[' [ DocValue { ',' DocValue } ] ]' .

DocValue = Scalar | doublequotedstring .

<sup>68</sup> This means that SELECT and MATCH queries that return a single row with a single column can be used to provide a scalar value.

To avoid being parsed as a doublequotedstring, in a DocValue a double-quoted identifier needs to be part of a larger expression such as a dotted identifier chain.<sup>69</sup>

BinaryOp = '+' | '-' | '\*' | '/' | '||' | MultisetOp .

|| is used in array and string concatenation.

VariableRef = { Scope\_id '.' } Variable\_id .

ColumnRef = [ TableOrAlias\_id '.' ] ColRef  
               | TableOrAlias\_id '.' CHECK  
               | SECURITY .

The use of the SECURITY and CHECK pseudo-columns is a change to SQL2023 behaviour. CHECK is a row versioning cookie accessible by anyone with select permission for the table.. SECURITY is reserved to the database owner (security administrator) and can be set to a value of type Level (see below).

MultisetOp = (MULTISET|SET) ( UNION | INTERSECT | EXCEPT ) ( ALL | DISTINCT ) .

ListValue = (SET|MULTISET|LIST) '[' Value {',' Value } ']' .

ArrayValue = ARRAY '[' int\_Value '=' Value {',' int\_Value '=' Value } ']' .

Literal =  
           | int  
           | float  
           | string  
           | TRUE | FALSE  
           | 'X' { hexit }  
           | DATE *date\_string*  
           | TIME *time\_string*  
           | TIMESTAMP *timestamp\_string*  
           | INTERVAL ['-' ] *interval\_string* IntervalQualifier  
           | Level .

Strings are enclosed in single quotes. Two single quotes in a string represent one single quote. Hexits are hexadecimal digits 0-9, A-F, a-f and are used for binary objects. Level literal can only be used by the database owner.

Dates, times and intervals use string (single quoted) values and are not locale-dependent. For full details see SQL2023: e.g.

- a date has format like DATE 'yyyy-mm-dd' ,
- a time has format like TIME 'hh:mm:ss' or TIME 'hh:mm:ss.sss' ,
- a timestamp is like TIMESTAMP 'yyyy-mm-dd hh:mm:ss.ss',
- an interval is like e.g.
  - INTERVAL 'yyy' YEAR,
  - INTERVAL 'yy-mm' YEAR TO MONTH,
  - INTERVAL 'm' MONTH,
  - INTERVAL 'd hh:mm:ss' DAY(1) TO SECOND,
  - INTERVAL 'sss.ss' SECOND(3,2) etc.

The SQL2023 standard specifies that intervals cannot have a mixture of year-month and date-second fields.

<sup>69</sup> In Mongo, keynames and strings starting with \$ have special meanings and can be used to refer to values in the current context (e.g. "\$a.b" ). Some Mongo usages are available as an option in the source code.

IntervalQualifier = StartField TO EndField

| DateTimeField .

StartField = IntervalField ['(' int ')'] .

EndField = IntervalField | SECOND ['(' int ')'] .

DateTimeField = StartField | SECOND ['(' int [' int ')'] ] .

The ints here represent precision for the leading field and optionally for seconds the fraction part.

IntervalField = YEAR | MONTH | DAY | HOUR | MINUTE .

## 7.7 RowSet Expressions

TableValue = VALUES '(' Scalar { ',' Scalar } ')' { ',' '(' Scalar { ',' Scalar } ')' } [AS *Type\_id*]  
| RowSetSpec  
| *Table\_Subquery* .

RowValue = [ROW] '(' Scalar { ',' Scalar } ')'   
| Scalar .

The Scalar option here constructs a row with a single column whose value is the given scalar value.<sup>70</sup>

TableExpression = [FromClause] [WhereClause] [GroupByClause] [HavingClause] [WindowClause] .

GroupByClause and HavingClause are used with aggregate functions. WindowClause is used with window functions. From v7 the FromClause can be omitted.

FromClause = FROM TableReference { ',' TableReference } .

WhereClause = WHERE BooleanExpr | (CURRENT OF *Cursor\_id*).

GroupByClause = GROUP BY [DISTINCT|ALL] GroupingSet { ',' GroupingSet } .

DISTINCT is the default.

GroupingSet = OrdinaryGroup | GroupingSpec | '()'.

OrdinaryGroup = ColumnRef [Collate] | '(' ColumnRef [Collate] { ',' ColumnRef [Collate] } ')' .

GroupingSpec = GROUPING SETS '(' GroupingSet { ',' GroupingSet } ')' .

HavingClause = HAVING BooleanExpr .

PartitionClause = PARTITION BY OrdinaryGroup .

WindowFrame = (ROWS|RANGE) (WindowStart|WindowBetween) [ Exclusion ] .

WindowStart = ((Scalar | UNBOUNDED) PRECEDING) | (CURRENT ROW) .

WindowBetween = BETWEEN WindowBound AND WindowBound .

WindowBound = WindowStart | ((Scalar | UNBOUNDED) FOLLOWING ) .

Exclusion = EXCLUDE ((CURRENT ROW)|GROUP|TIES|(NO OTHERS)) .

TableReference = TableFactor Alias | JoinedTable .

TableFactor = *Table\_id* [FOR SYSTEM\_TIME [TimePeriodSpecification ]]  
| *View\_id*  
| ROWS '(' int [ ',' int ] ')'   
| *Table\_FunctionCall*  
| *Table\_Subquery*  
| '(' TableReference ')'

<sup>70</sup> This syntax is called <row value constructor> in the SQL standard (section 7.1).

```

|      TABLE '(' Scalar ')'
|      UNNEST '(' Scalar ')'
|      DocArray .

```

ROWS(..) is a Pyrrho extension (for table and cell logs), and the last option above is also Pyrrho-specific and allows a specific list of documents to be supplied. The value in UNNEST is normally an array of rows, but DocArray values are interpreted in the obvious way.

## 7.8 Predicates

BooleanExpr = BooleanTerm | BooleanExpr OR BooleanTerm .

BooleanTerm = BooleanFactor | BooleanTerm AND BooleanFactor .

BooleanFactor = [NOT] BooleanTest .

BooleanTest = Predicate | '(' BooleanExpr ')' | *Boolean\_Value* .

Predicate = Any | Between | Comparison | Contains | Every | Exists | In | Like | Member | Null | Of | PeriodBinary | Some | Unique | [ColumnRef '.'] *Document\_Value* .

The use of a Document as a predicate is considered to be an equality condition consisting of a conjunction of equality conditions for its field names and values.

Any = ANY '(' [DISTINCT|ALL] Value ')' FuncOpt .

The qualifier DISTINCT|ALL has no effect for ANY.

Between = Value [NOT] BETWEEN [SYMMETRIC|ASYMMETRIC] Value AND Value .

Comparison = Scalar CompOp Scalar .

CompOp = '=' | '<' | '<=' | '>' | '>=' | '>' .

Contains = PeriodPredicand CONTAINS (PeriodPredicand | *DateTime\_Value*) .

Every = EVERY '(' [DISTINCT|ALL] Value ')' FuncOpt .

The qualifier DISTINCT|ALL has no effect for EVERY.

Exists = EXISTS NestedStatement.

The result of EXISTS is a boolean value indicating whether the NestedStatement had a non-empty result.

FuncOpt = [FILTER '(' WHERE SearchCondition ')'] [OVER WindowSpec] .

The presence of the OVER keyword makes a *window function*. In accordance with SQL2023-02 section 6.10 and 4.16.3. Window functions can only be used in the select list of a QuerySpec or SelectSingle or the order by clause of a simple table query. Thus window functions cannot be used within expressions or as function arguments.

In = RowValue [NOT] IN '(' *Table\_Subquery* | ( Scalar { ',' Scalar } ) ')' .

Like = Scalar [NOT] LIKE *Char\_Scalar* [ ESCAPE *Char\_Scalar* ] .

LIKE\_REGEX and SIMILAR can be supported using directives in the source code.

Member = RowValue [ NOT ] MEMBER OF *Multiset\_Scalar* .

Null = Scalar IS [NOT] NULL .

Of = Value IS [NOT] ( OF '(' [ONLY] Type { ',' [ONLY] Type } ')' | (CONTENT | DOCUMENT | VALID) ) .

Some = SOME '(' [DISTINCT|ALL] TableValue ')' FuncOpt .



The qualifier DISTINCT|ALL has no effect for SOME.

Unique = UNIQUE *Table\_Subquery* .

PeriodBinary = PeriodPredicand (OVERLAPS | EQUALS | [IMMEDIATELY] (PRECEDES | SUCCEEDS)) PeriodPredicand .

See also Contains above.

PeriodPredicand = { id '.' } id | PERIOD '(' Scalar ',' Scalar ')' .

## 7.9 SQL Functions

FunctionCall = NumericValueFunction | StringValueFunction | DateTimeFunction | GraphFunctions | TypeCast | HTTPFunction | VersioningFunction | VectorFunctions | UserFunctionCall | MethodCall .

All FunctionCalls are considered Scalars unless the returned type is TABLE.

NumericValueFunction = AbsoluteValue | Avg | Ceiling | Coalesce | Count | Exponential | Extract | Floor | Grouping | Last | LengthExpression | Maximum | Minimum | Modulus | NaturalLogarithm | Next | Nullif | Position | PowerFunction | RowNumber | Schema | SquareRoot | Sum .

AbsoluteValue = ABS '(' Scalar ')' .

Avg = AVG '(' [DISTINCT|ALL] Scalar ')' FuncOpt .

ALL is the default.

Ceiling = (CEIL|CEILING) '(' Scalar ')' .

Coalesce = COALESCE '(' Scalar '{',' Scalar }' ')' .

Count = COUNT '(' '\*' ')'
   
| COUNT '(' [DISTINCT|ALL] Scalar ')' FuncOpt
   
| COUNT '(' ColumnRef ')' OVER WindowSpec .

ALL is the default.

Exponential = EXP '(' Scalar ')' .

Extract = EXTRACT '(' ExtractField FROM Value ')' .

ExtractField = YEAR | MONTH | DAY | HOUR | MINUTE | SECOND.

First = FIRST\_VALUE '(' ColumnRef ')' OVER WindowSpec .

Floor = FLOOR '(' Scalar ')' .

Grouping = GROUPING '(' ColumnRef { ',' ColumnRef } ')' .

HttpFunction = HTTP '(' verb\_Value ',' url\_Value ',' content\_Value ')' .

In the HttpFunction added for Pyrrho v7.01, verb and url are string values, content is a possibly empty Json Document and the return value is a possibly empty Json Document. (See section 7.4)

Last = LAST\_VALUE '(' ColumnRef ')' OVER WindowSpec .

LastData= LAST\_DATA .

Table/derived-table function added for Pyrrho v7: the log position of the last table change (or 0 if no relevant tables).

LengthExpression = (CHAR\_LENGTH|CHARACTER\_LENGTH|OCTET\_LENGTH) '(' Scalar ')' .

Maximum = MAX '(' [DISTINCT|ALL] Scalar ')' FuncOpt .

The qualifier DISTINCT|ALL has no effect for MAX.

Minimum = MIN '(' [DISTINCT|ALL] Scalar) ')' FuncOpt .

The qualifier DISTINCT|ALL has no effect for MIN.

Modulus = MOD '(' Scalar ',' Scalar ')' .

NaturalLogarithm = LN '(' Scalar ')' .

Next = NEXT [(' ColumnRef ') OVER WindowSpec ] .

Nullif = NULLIF '(' Scalar ',' Scalar ')' .

WindowSpec = '(' [ PartitionClause ] [ OrderByClause ] [ WindowFrame ] ')' .

WithinGroup = WITHIN GROUP '(' OrderByClause ')' .

Position = POSITION [('Scalar IN TableValue ')]

PowerFunction = POWER '(' Scalar ',' Scalar ')' .

RowNumber = ROW\_NUMBER '('")' OVER WindowSpec .

Schema = SCHEMA '(' ObjectName [ COLUMN id ]')' .

Added for Pyrrho: returns a number identifying the most recent schema change affecting the specified object (including any change to this object by another name in another role). Note the syntax of ObjectName given in sec 7.4 above above uses keyword prefixes such as TABLE. The COLUMN syntax shown can only be used with tables.

SquareRoot = SQRT '(' Scalar ')' .

Sum = SUM '(' [DISTINCT|ALL] Scalar) ')' FuncOpt .

DateTimeFunction = CURRENT\_DATE | CURRENT\_TIME | LOCALTIME |  
CURRENT\_TIMESTAMP | LOCAL\_DATETIME | LOCAL\_TIMESTAMP .

StringValueFunction = Substring.

Normalize= NORMALIZE '(' Scalar ')' .

Substring = SUBSTRING '(' Scalar FROM Scalar [ FOR Scalar ] ')' .

SetFunction = Cardinality | Collect | Element | Fusion | Intersect | Set .

Collect = COLLECT '(' [DISTINCT|ALL] Scalar) ')' FuncOpt .

Fusion = FUSION '(' [DISTINCT|ALL] Scalar) ')' FuncOpt .

Intersect = INTERSECTION '(' [DISTINCT|ALL] Value) ')' FuncOpt .

Cardinality = CARDINALITY '(' Scalar ')' .

Element = ELEMENT '(' Scalar ')' .

Set = SET '(' Scalar ')' .

GraphFunctions = GraphLabels | GraphType .

GraphLabels = LABELS '(' Value ')' .

GraphType = TYPE '(' Value ')' .

Typecast = CAST '(' Scalar AS Type ')' | TREAT '(' Scalar AS *Sub\_Type* ')' .

VectorFunctions = VectorConstructor | VectorDimensionCount | VectorDistance | VectorSerialize .

VectorConstructor = VECTOR '(' String\_Value ',' Int\_Value ',' Type ') ' .

VectorDimensionCount = VECTOR\_DIMENSION\_COUNT '(' Value ') ' .

VectorDistance = VECTOR\_DISTANCE '(' Value ',' Value Metric ') ' .

VectorNorm = VECTOR\_NORM '(' Value ',' (EUCLIDEAN|MANHATTAN) ') ' .

EUCLIDEAN is the square root of the sum of squares of the coordinates, MANHATTAN is the sum of their absolute values.

VectorSerialize = VECTOR\_SERIALIZE '(' Value ') ' [RETURNING Type] .

Type is a string type: the vector serialize function amounts to JSON serialization.

Metric = EUCLIDEAN | EUCLIDEAN\_SQUARED | MANHATTAN | COSINE | DOT | HAMMING .

EUCLIDEAN distance is the square root of the sum of square differences of corresponding coordinates, MANHATTAN is the sum of absolute values of their differences, COSINE is the sum of products of corresponding coordinates divided by the product of the Euclidean norms of the vectors, DOT is the sum of absolute values of differences, and HAMMING is the number of coordinates where the components differ.

VersioningFunction = VERSIONING '(' RowSet ') ' .

Added for Pyrrho v7, returns a CHECK cookie for the versioning state of the rows in the rowSet.

## 7.10 Compliance with the SQL and GQL standards

This version of Pyrrho supports GQL syntax. GQL has many more reserved words than SQL, shown in the above syntax rules in a sans-serif font, a simpler security model and transaction mechanism. There is a draft compliance statement for ISO39075 in the diagnostics section 8.1.3 below.

The standards have many features in common, and Pyrrho continues to support SQL2023 features, too many to list here. Importantly, the set of reserved words in Pyrrho does not include all SQL's reserved words. If a database defines SQL keywords that are not reserved in GQL to have any other meaning, SQL syntax containing such keywords may be unavailable.

In addition, some syntax specified earlier in this section is not allowed in either standard. The notes in this section identify aspects that are known to be non-compliant.

### 7.10.1 SQL-sessions

Each SQL-connection<sup>71</sup> corresponds to a single SQL-session. An SQL-session has a single user. The name of the current user is established by the operating system before the connection opens, and may match the account that has started the Pyrrho server (the server account). An SQL-session is established for the current user by the HTTP service using a URL specifying a database name and role name, or by an application opening a PyrrhoConnect using a connection string, which must specify a database and may specify a role.

If no database with the specified name is found and the current user is the server account, an empty database is created that is owned by the server account: in that case the database contains no users or roles, and the server account is the owner of the database and of the default role, whose name matches the database name.

There are thus three cases (a) the set of database objects defines no users, and the user is the server account, (b) the user matches a User in the database, (c) the user is Guest.

The SQL-connection persists until the connection is closed by the client or the underlying transport is broken. This will also terminate the SQL-session. At any point in time, many SQL-sessions may be in progress for the database, and it is possible for many such sessions to have the same current user. A single user can have many SQL-connections in progress at any time.

---

<sup>71</sup> Pyrrho does not have feature F771 "Connection management" or F321 "User authorization". Pyrrho does have features T331 "Basic roles" and T332 "Extended roles", modified as described below.

## 7.10.2 SQL-transactions

An SQL-session can have at most one SQL-transaction in progress at any time. At the end of an SQL session any transaction in progress is rolled back.

In this section S is any SQL or GQL statement, whether it affects schema, security, or database contents. If there is no transaction in progress at the start of execution of S, an auto-commit transaction is initiated, otherwise S begins a transaction step in T. If S is found to have a syntax error, an exception occurs, the execution of S does not proceed, and any changes associated with the transaction step are rolled back. If any other kind of exception arises during the execution of S and is not handled, transaction T is rolled back. In both cases, the exception is reported to the client. At the end of the execution of S, if T is an auto-commit transaction, it is committed.

The isolation level for all transactions is `SERIALIZABLE` and cannot be changed. Changes made by a transaction (including schema changes) are not visible to other transactions until the transaction is committed.

## 7.10.3 Roles

At any point in the SQL session, there is just one current role. A value for `CURRENT_ROLE` can be specified in the connection string (or URL). Otherwise, the server will select a suitable value from among the Roles that for which the current user has the `USAGE` permission. If there is no such role, the `CURRENT_ROLE` is `PUBLIC`.

The `SET ROLE` statement can be used during an SQL-session to change the value of `CURRENT_ROLE`: it must specify a role for which the user has `USAGE` permission.

*Execution uses definer's role:* If the current role R allows an operation on an object O (e.g. a table, a view, a constraint, a procedure, or trigger) defined by role R', then during any computations defined by O, the current role is temporarily set to R'. If O is (or evaluates to) a row or table, access to its columns is again determined by R.

## 7.10.4 Privileges

If there are no users or roles in the database, there are no privilege descriptors in the database. In this state, provided the current user is the server account, execution proceeds as if they have all privileges on all objects. In this state, the first Role to be created becomes the default role for the database and the definer's role for all user-defined objects in the database and privilege descriptors are added for all privileges, and the first User to be named as a grantee in a grant statement G becomes the database owner and the owner of all of its objects in the database at this point, and a `USAGE` privilege descriptor is added to the current role (in addition to the permissions granted by G).

At any time thereafter, (a) the database owner has `SELECT` permission on the log tables and usage permission on the default role, and the default role has `SELECT` permission on the system tables; (b) if a user-defined database object has no remaining privilege descriptors it is dropped in a cascade<sup>72</sup>.

Grant of `SELECT` on a table by default allows `SELECT` on all of its columns. All data types are implicitly `PUBLIC` i.e. all roles implicitly have the `USAGE` privilege.

Objects including types can be altered or dropped only by their owner (a user).

The `REFERENCES` privilege is not supported: a role has effectively the same privilege if it has `SELECT` on the column.

`WITH HIERARCHY OPTION` is not supported.

Grant to a user of anything other than ownership or role usage is deprecated. Grant of ownership can only be done by the object's owner.

Grant of usage of a role only creates privilege descriptors for the user for the role. The current privileges of the user on any database object are determined by their current role.

Grant to a role of ownership or role usage is deprecated.

---

<sup>72</sup> It is relatively unusual practice to remove the access privilege of an object's creator.

Predefined data types are considered to be owned by SYSTEM. All roles are considered to have the USAGE privilege on all data types including tables and graph types but can define alternative names for them and their attributes.

### **7.10.5 Drop statements**

NO\_ACTION is not allowed.

### **7.10.6 Integrity Constraints**

NO\_ACTION is not allowed. DEFERRED defers operation of the constraint to the end of the transaction.

### **7.10.7 Data Types**

There are numerous departures from the SQL standard, see section 7.4.

Predefined types can have user defined scalar types as subtypes. User defined row types have an associated table of instances. Graph types are implemented as user defined types with multiple inheritance.

### **7.10.8 Tables**

There are effectively only two types of table: base table and derived table. A viewed table is treated as a derived table (resulting from the view definition). Transition tables (SQL) and binding tables (GQL) are derived table whose lifetimes are limited to execution of the statement containing them. The view syntax has been extended to allow access to remote tables, and such views can specify column types and integrity constraints.

## 8. Pyrrho Reference

There are five collections of system tables in Pyrrho. The Sys\$ collections contain the current system information set, the Role\$ collection is the schema for the current role, and the Log\$ collection accesses the transaction log.

All these collections consist of virtual tables, whose data is constructed as required from the Pyrrho engine's data structures. From version 5.0 it is possible to see uncommitted details in the current transaction, so that "defining positions" in these system tables are no longer Integer but String data: the fields contain the string version of the Integer defining position if it is committed, and otherwise contain a numeric identifier preceded by a single character. This sort of column is called Position below. There is a further special case for Domains, whose defining position is shown if the database defines it, while standard Domains are given by their possible truncated name.

The fourth kind of system table is for reviewing data operations on an individual table. See section 8.5.

All these tables and their attributes are case-sensitive, and the table-names contain the character \$, so all SQL statements will need to use double-quoted (delimited) identifiers, as in

```
Select * from "Sys$Role" where "Name" like 'Sales%'
```

### 8.1 Diagnostics

Pyrrho implements basic diagnostics management as defined in GQL, with a single diagnostics area. The NOT\_FOUND condition is signalled if there is a handler for it (it is not an error).

#### 8.1.1 GQLSTATUS

Pyrrho defines the following condition codes, shown here with the message formats for the invariant culture (these can be localised in the client library).

Pyrrho treats many things as errors that appear in the GQL standard as warnings and imposes fewer restrictions: see comments below. Additional error messages below (in category 40) relate to transaction conflicts caused by schema changes. Those in class 08 cannot be handled by a GQL-procedure.

Databases may define, raise and handle other condition codes.

Number	Message Template	GQL	Pyrrho	Comments
00000	Successful completion	y	n	Not raised as signal
00001	Omitted result	y	n	Not raised as signal
01000	Warning	y	n	
01004	Warning – string data, right truncation	y	n	Default max length is 2 <sup>31</sup> -1
01G03	Warning – graph does not exist	y	n	Silently added if possible
01G04	Warning – graph type does not exist	y	n	Silently added if possible
01G11	Warning – null value eliminated in set function	y	n	
02000	No data	y	y	Raised if defined
03000	Informational	y	n	
08000	Connection exception	y	n	Raised by client
08001	Client unable to establish connection	n	n	Reported by client applications
08004	Server unavailable	n	n	Reported by client applications
08007	Connection exception – transaction resolution unknown	y	n	Cannot occur
08C00	Client side threading violation for reader	n	n	Reported by client applications
08C01	Client side threading violation for command	n	n	Reported by client applications
08C02	Client side threading violation for transaction	n	n	Reported by client applications
08C03	An explicit transaction is already active in this thread and connection	n	n	Reported by client applications
08C04	A reader is already open in this thread and connection	n	n	Reported by client applications
08C06	Cannot change connection properties during a transaction	n	n	Reported by client applications
21000	Cardinality violation	y	y	

22000	Data exception	y	y	
22001	String data, right truncation	y	y	
22003	Numeric value out of range	y	y	
22004	Null value not allowed	y	y	
22007	Invalid datetime format: ?	y	y	Diagnostic info added
22008	Datetime field overflow: ?	y	y	Diagnostic info added
22011	Substring error	y	y	
22012	Division by zero	y	y	
22015	Interval field overflow	y	n	Cannot occur
22018	Invalid character value for cast	y	n	
22019	Invalid escape character	y	y	
2201E	Invalid argument for natural logarithm	y	n	
2201F	Invalid argument for power function	y	n	
22025	Invalid escape sequence	n	y	
22027	Trim error	y	n	
2202F	Array data, right truncation	y	n	Default max size is 2^63-1
2202G	Invalid repeat argument in a sample clause	y	n	
22042	Unknown schema key	n	y	
22102	Type mismatch on concatenate	n	y	
22103	Multiset element not found	n	y	
22104	Incompatible types for union	n	y	Generalized from multisets
22105	Incompatible types for intersection	n	y	Generalized from multisets
22106	Incompatible types for except	n	y	Generalized from multisets
22107	Exponent expected	n	y	
22108	Type error in aggregation operation	n	y	
22109	Too few arguments	n	y	
22110	Too many arguments	n	y	
22111	Circular dependency found	n	y	
22203	Loss of precision on conversion	n	y	
22204	Rowset expected	n	y	Changed from Query
22205	Null value found in table ?	n	y	
22206	Multiplicity ? min value not reached	n	y	
22207	Multiplicity ? ,ax value exceeded	n	y	
22208	Cardinality ? min value not reached	n	y	
22209	Cardinality ? max value exceeded	n	y	
22211	Check constraint fails	n	y	
22G02	Negative limit value	y	y	
22G03	Invalid value type	y	y	
22G04	Values not comparable	y	y	
22G05	Invalid date, time, or datetime function field name	y	n	Reported as syntax error
22G06	Invalid datetime function value	y	n	Cannot arise
22G07	Invalid duration field name	y	n	Reported as syntax error
22G0B	List data, right truncation	y	n	Cannot arise
22G0C	List element error	y	n	
22G0F	Invalid number of paths or groups	y	n	
22G0H	Invalid duration format	y	n	See 22007
22G0K	Multi-sourced or multi-destined edge	n	y	SQL/PGQ
22G0L	Incomplete edge ?	n	y	SQL/PGQ
22G0M	Multiple assignments to a graph element property	y	n	Otherwise reported
22G0N	Number of node labels below supported minimum	y	n	Cannot arise
22G0P	Number of node labels exceeds supported maximum	y	n	..
22G0Q	Number of edge labels below supported minimum	y	n	..
22G0R	Number of edge labels exceeds supported maximum	y	n	..

22G0S	Number of node properties exceeds supported maximum	y	n	..
22G0T	Number of edge labels exceeds supported maximum	y	n	..
22G0U	Record fields do not match	y	n	See 22G03
22G0V	Reference value, invalid base type	y	n	Cannot arise
22G0W	Reference value, invalid constrained type	y	n	Cannot arise
22G0X	Record data, field unassignable	y	y	
22G0Y	Record data, field missing	y	y	
22G0Z	Malformed path	y	n	
22G10	Path data, right truncation	y	n	Cannot arise
22G11	Reference value, referent deleted	y	n	
22G12	Invalid value type	y	n	Cannot arise
22G13	Invalid group variable name	y	n	
22G14	Incompatible temporal instant unit groups	y	n	
22G21	Edge connection ? missing	n	y	
23000	Integrity constraint violation	y	y	
23001	RESTRICT: ? referenced in ?	y	y	A referenced object cannot be deleted
23002	RESTRICT: Index is not empty	n	y	
23101	Integrity constraint on referencing table ? (delete)	n	y	
23102	Integrity constraint on referencing table ? (update)	n	y	
23103	This record cannot be updated	n	y	Usually integrity violation
24000	Invalid cursor state	n	y	
24101	Cursor is not open	n	y	
25000	Invalid transaction state	y	y	
25G01	Active GQL-transaction	y	y	
25G02	Catalog and data statement mixing not supported	y	n	Mixing is supported
25G03	Read-only GQL transaction	y	n	Cannot arise
25G04	Accessing multiple graphs not supported	y	n	Such access is supported
26000	Invalid SQL statement name	n	y	
28000	Invalid authorization specification	n	y	No role ? in database ?
28101	Unknown grantee kind	n	y	
28102	Unknown grantee ?	n	y	
28104	Users can only be added to roles	n	y	
28105	Grant of select: entire row is nullable	n	y	
28106	Grant of insert must include all notnull columns	n	y	
28107	Grant of insert cannot include generated column ?	n	y	
28108	Grant of update : column ? is not updatable	n	y	
2D000	Invalid transaction termination	y	y	
2E104	Database is read-only	n	y	
2E105	Invalid user for database ?	n	y	
2E106	This operation requires a single-database session	n	y	
2E108	Stop time was specified, so database is read-only	n	y	
2E110	Unauthorized HTTP access	n	y	
2E111	User ? can access no columns of table ?	n	y	
2E201	Connection is not open	n	y	See also 080nn
2E202	A reader is already open	n	y	
2E203	Unexpected reply	n	y	
2E204	Bad data type ? (internal)	n	y	
2E205	Stream closed	n	y	
2E206	Internal error: ?	n	y	



2E208	Badly formatted connection string ?	n	y	
2E209	Unexpected element ? in connection string	n	y	
2E210	LOCAL database server does not support distributed or partitioned operation	n	y	
2E300	<i>The calling assembly does not have type ?</i>	n	y	
2E301	<i>Type ? doesn't have a default constructor</i>	n	y	
2E302	<i>Type ? doesn't define field ?</i>	n	y	
2E303	Types ? and ? do not match	n	y	
2E304	Get rurl should begin with /	n	y	REST service
2E305	No data returned by rurl ?	n	y	REST service
2E307	Obtain an up-to-date schema for ? from Role\$Class	n	y	
2F003	Prohibited SQL-statement attempted	n	y	
2H000	Invalid collation name	n	y	
33000	Invalid SQL descriptor name	n	y	
33001	Error in prepared statement parameters	n	y	
34000	Invalid cursor name	n	y	
3D000	Invalid catalog specification	y	y	
3D001	Database ? not open	n	y	
3D005	Requested operation not supported by this edition of Pyrrho	n	y	
3D006	Database ? incorrectly terminated or damaged	n	y	
3D007	Database is not append storage	n	y	Server is append storage version
3D008	Database is append storage	n	y	Server is not for append storage
40000	Transaction rollback	y	y	
40001	Transaction Serialisation Failure	y	y	
40002	Transaction rollback – integrity constraint violation	y	n	
40003	Transaction rollback – statement completion unknown	y	y	
40004	Transaction rollback – triggered action exception	y	n	
40005	Transaction rollback – new key conflict with empty query	n	y	
40006	Transaction conflict: Read constraint for ?	n	y	
40007	Transaction conflict: Read conflict for ?	n	y	
40008	Transaction conflict: Read conflict for table ?	n	y	
40009	Transaction conflict: Read conflict for record ?	n	y	
40010	Object ? has just been dropped	n	y	
40011	Supertype ? has just been dropped	n	y	
40012	Table ? has just been dropped	n	y	
40013	Column ? has just been dropped	n	y	
40014	Record ? has just been deleted	n	y	
40015	Type ? has just been dropped	n	y	
40016	Domain ? has just been dropped	n	y	
40017	Index ? has just been dropped	n	y	
40021	Supertype ? has just been changed	n	y	
40022	Another domain ? has just been defined	n	y	
40023	Period ? has just been changed	n	y	
40024	Versioning has just been defined	n	y	
40025	Table ? has just been altered	n	y	
40026	Integrity constraint: ? has just been added	n	y	
40027	Integrity constraint: ? has just been referenced	n	y	
40029	Record ? has just been updated	n	y	
40030	A conflicting table ? has just been defined	n	y	
40031	A conflicting view ? has just been defined	n	y	

40032	A conflicting object ? has just been defined	n	y			
40033	A conflicting trigger for ? has just been defined	n	y			
40034	Table ? has just been renamed	n	y			
40035	A conflicting role ? has just been defined	n	y			
40036	A conflicting routine ? has just been defined	n	y			
40037	An ordering now uses function ?	n	y			
40038	Type ? has just been renamed	n	y			
40039	A conflicting method ? for ? has just been defined	n	y			
40040	A conflicting period for ? has just been defined	n	y			
40041	Conflicting metadata for ? has just been defined	n	y			
40042	A conflicting index for ? has just been defined	n	y			
40043	Columns of table ? have just been changed	n	y			
40044	Column ? has just been altered	n	y			
40045	A conflicting column ? has just been defined	n	y			
40046	A conflicting check ? has just been defined	n	y			
40047	Target object ? has just been renamed	n	y			
40048	A conflicting ordering for ? has just been defined	n	y			
40049	Ordering definition conflicts with drop of ?	n	y			
40050	A conflicting namespace change has occurred	n	y			
40051	Conflict with grant/revoke on ?	n	y			
40052	Conflicting routine modify for ?	n	y			
40053	Domain ? has just been used for insert	n	y			
40054	Domain ? has just been used for update	n	y			
40055	An insert conflicts with drop of ?	n	y			
40056	An update conflicts with drop of ?	n	y			
40057	A delete conflicts with drop of ?	n	y			
40058	An index change conflicts with drop of ?	n	y			
40059	A constraint change conflicts with drop of ?	n	y			
40060	A method change conflicts with drop of type ?	n	y			
40068	Domain ? has just been altered, conflicts with drop	n	y			
40069	Method ? has just been changed, conflicts with drop	n	y			
40070	A new ordering conflicts with drop of type ?	n	y			
40071	A period definition conflicts with drop of ?	n	y			
40072	A versioning change conflicts with drop of period ?	n	y			
40073	A read conflicts with drop of ?	n	y			
40074	A delete conflicts with update of ?	n	y			
40075	A new reference conflicts with deletion of ?	n	y			
40076	A conflicting domain or type ? has just been defined	n	y			
40077	A conflicting change on ? has just been done	n	y			
40078	Read conflict with alter of ?	n	y			
40079	Insert conflict with alter of ?	n	y			
40080	Update conflict with alter of ?	n	y			
40081	Alter conflicts with drop of ?	n	y			
40082	ETag validation failure	n	y			
40083	Secondary connection conflict on ?	n	y	Remote connection snapshots differ		
40084	Transaction start conflict	n	y			
40085	An update conflicts with delete of ?	n	y			

42000	Syntax error or access rule violation at ?	y	y	
42001	Invalid syntax	y	n	
42002	Invalid reference	y	n	
42004	Use of visually confusable identifiers	y	n	
42006	Number of edge labels below supported minimum	y	n	Cannot arise
42007	Number of edge labels exceeds supported maximum	y	n	..
42008	Number of edge properties exceeds supported maximum	y	n	..
42009	Number of node labels below supported minimum	y	n	..
42010	Number of node labels exceeds supported maximum	y	n	..
42011	Number of node properties exceeds supported maximum	y	n	..
42012	Number of node type key labels below supported minimum	y	n	..
42013	Number of node type key labels exceeds supported maximum	y	n	..
42014	Number of edge type key labels below supported minimum	y	n	..
42015	Number of edge type key labels exceeds supported maximum	y	n	..
42101	Illegal character ?	n	y	
42102	Name cannot be null	n	y	
42103	Key must have at least one column	n	y	
42104	Proposed name conflicts with existing database object (e.g. table already exists)	n	y	
42105	Access denied ?	n	y	
42107	Table ? undefined	n	y	
42108	Procedure ? not found	n	y	
42109	Assignment target ? not found	n	y	
42111	The given key is not found in the referenced table	n	y	
42112	Column ? not found	n	y	
42113	Multiset operand required, not ?	n	y	
42115	Unexpected object type ? ? for GRANT	n	y	
42116	Role revoke has ADMIN option not GRANT	n	y	
42117	Privilege revoke has GRANT option not ADMIN	n	y	
42118	Unsupported CREATE ?	n	y	
42119	Domain ? not found in database ?	n	y	
4211A	Unknown privilege ?	n	y	
42120	Domain or type must be specified for base column ?	n	y	
42123	NO ACTION is not supported	n	y	
42124	Colon expected ..	n	y	
42125	Unknown Alter type ?	n	y	
42126	Unknown SET operation	n	y	
42127	Table expected	n	y	
42128	Illegal aggregation operation	n	y	
42129	WHEN expected	n	y	
42131	Invalid POSITION ?	n	y	
42132	Method ? not found in type ?	n	y	
42133	Type ? not found	n	y	
42134	FOR phrase is required	n	y	
42135	Object ? not found	n	y	
42138	Field selector ? not defined for ?	n	y	
42139	:: on non-type	n	y	

42140	:: requires a static method	n	y	
42142	NEW requires a user-defined type constructor	n	y	
42143	? specified more than once	n	y	
42146	OLD specified on insert trigger or NEW specified on delete trigger	n	y	
42147	Cannot have two primary keys for table ?	n	y	
42148	FOR EACH ROW not specified	n	y	
42149	Cannot specify OLD/NEW TABLE for before trigger	n	y	
42150	Malformed SQL input (non-terminated string)	n	y	
42151	Bad join condition	n	y	
42152	Non-distributable where condition for update/delete	n	y	
42153	Table ? already exists	n	y	
42154	Unimplemented or illegal function ?	n	y	
42156	Column ? is already in table ?	n	y	
42157	END label ? does not match start label ?	n	y	
42158	? is not the primary key for ?	n	y	
42159	? is not a foreign key for ?	n	y	
42160	? has no unique constraint	n	y	
42161	? expected at ?	n	y	
42162	Table period definition for ? has not been defined	n	y	
42163	Generated column ? cannot be used in a constraint	n	y	
42164	Table ? has no primary key	n	y	
42166	Domain ? already exists	n	y	
42167	A routine with name ? and arity ? already exists	n	y	
42168	AS GET needs a schema definition	n	y	
42169	Ambiguous column name ? needs alias	n	y	
42170	Column ? must be aggregated or grouped	n	y	
42171	A table cannot be placed in a column	n	y	
42172	Identifier ? already declared in this block	n	y	
42173	Method ? not defined	n	y	
42174	Unsupported rowset modification attempt	n	y	
42175	Alternative match expressions must bind the same identifiers	n	y	Graphical Query extensions
44000	With check option violation	n	y	
44001	Domain check ? fails for column ? in table ?	n	y	
44002	Table check ? fails for table ?	n	y	
44003	Column check ? fails for column ? in table ?	n	y	
44004	Column ? in Table ? contains null values, not null cannot be set	n	y	
44005	Column ? in Table ? contains values, generation rule cannot be set	n	y	

### 8.1.2 Get Diagnostics

From version 4.8, Pyrrho supports the GET DIAGNOSTICS statement, giving useful information for the following keys. When an exception condition is handled in an SQL routine or reported to the client, information from this collection is included in the DatabaseError.

CATALOG_NAME	
CLASS_ORIGIN	This is ISO 9075 for conditions whose class is defined in SQL2023
COLUMN_NAME	
COMMAND_FUNCTION	From Table 32 of the SQL standard
COMMAND_FUNCTION_CODE	From Table 32 of the SQL standard

CONDITION_NUMBER	
CONNECTION_NAME	This is the Files part of the connection string
CONSTRAINT_NAME	
CURSOR_NAME	
MESSAGE_LENGTH	Computed from MESSAGE_TEXT
MESSAGE_OCTET_LENGTH	Computed from MESSAGE_TEXT
MESSAGE_TEXT	By default, this is formatted when an exception occurs
RETURNED_SQLSTATE	The condition code
ROUTINE_NAME	
ROW_COUNT	
SERVER_NAME	The host part of the connection string
SUBCLASS_ORIGIN	This is ISO 9075 if the whole condition code is defined in SQL2023
TABLE_NAME	
TRANSACTIONS_COMMITTED	The number of transactions committed for this connection
TRANSACTIONS_ROLLED_BACK	The number of rollbacks for this connection
TRIGGER_NAME	
TYPE*	The target type
VALUE*	The value type
WITH*	Additional information for transaction conflicts (version 5.4)

\*Pyrrho specific.

### 8.1.3 Draft GQL compliance statement

Pyrrho aims to be conformant to the GQL specification **except for the points in red text below**:

- 1) It supports GG01, GG02, GG20, GG201 and GG22.
- 2) It supports the Unicode standard via Microsoft's .NET framework 8.0.
- 3) The value types include STRING, BOOL, INT, FLOAT.

### Features where conformance is intended

Feature IDs etc are as in the GQL specification (ISO 39075).

GQL ref	Feature ID	Feature Name
3	G004	Path variables
8	G011	Advanced path modes: TRAIL
9	G012	Advanced path modes: SIMPLE
10	G013	Advanced path modes: ACYCLIC
12	G015	All path search: explicit ALL keyword
13	G016	Any path search
14	G017	All shortest path search
15	G018	Any shortest path search
16	G019	Counted shortest path search
20	G032	Path patter union
21	G033	Path pattern union: variable length path operands
22	G035	Quantified paths
23	G036	Quantified edges
24	G037	Questioned paths
25	G038	Parenthesized path pattern expression
35	G050	Parenthesized path pattern: WHERE clause
37	G060	Bounded graph pattern quantifiers
38	G061	Unbounded graph pattern quantifiers
50	GA01	IEEE 754 floating point operations
53	GA05	Cast specification
54	GA06	Value type predicate
55	GA07	Ordering by discarded binding variables
56	GA08	GQL status objects with diagnostic records
58	GB01	Long identifiers
61	GC01	Graph schema management
64	GC04	Graph management

66	GD01	Updatable graphs
67	GD02	Graph label set changes
68	GD03	DELETE statement: subquery support
69	GD04	DELETE statement: simple expression support
70	GE01	Graph reference value expressions
71	GE02	Binding table reference value expressions
72	GE03	Let-binding of variables in expression
80	GF02	Trigonometric functions
81	GF03	Logarithmic functions
86	GF10	Advanced aggregate functions: general set functions
87	GF11	Advanced aggregate functions: binary set functions
88	GF12	CARDINALITY functions
89	GF13	SIZE functions
90	GF20	Aggregate functions in sort keys
91	GG01	Graph with an open data type
92	GG02	Graph with a closed graph type
93	GG03	Graph type inline specification
94	GG04	Graph type like a graph
95	GG05	Graph from a graph source
96	GG20	Explicit element type names
97	GG21	Explicit element type key label sets
98	GG22	Element type key label set inference
105	GL01	Hexadecimal literals
108	GL04	Exact number in common notation without suffix
111	GL07	Approximate number in common notation or as decimal integer with suffix
116	GL12	SQL datetime and interval formats
117	GP01	Inline procedure
118	GP02	Inline procedure with implicit anested variable scope
120	GP04	Named procedure calls
121	GP05	Procedure-local value variable definitions
122	GP06	Procedure-local value variable definitions: valuie variables based on simple expressions
123	GP07	Procedure-local value variable definitions: value variable based on subqueries
124	GP08	Procedure local binding table variable definitions
125	GP09	Procedure local binding table variable definitions: binding table variables based on simple expressions or references
126	GP10	Procedure local binding table variable definitions: binding table variables based on subqueries
132	GP16	AT schema clause
134	GP18	Catalog and data statement mixing
135	GQ01	USE graph clause
136	GQ02	Composite query: OTHERWISE
137	GQ03	Composite query: UNION
138	GQ04	Composite query: EXCEPT DISTINCT
139	GQ05	Composite query: EXCEPT ALL
140	GQ06	Composite query: INTERSECT DISTINCT
141	GQ07	Composite query: INTERSECT ALL
142	GQ08	FILTER statement
143	GQ09	LET statement
144	GQ10	FOR statement: list value support
145	GQ11	FOR statement: WITH ORDINALITY
146	GQ12	ORDER BY and page statement: OFFSET clause
147	GQ13	ORDER BY and page statement: LIMIT clause
148	GQ14	Complex expressions in sort keys
149	GQ15	GROUP BY clause
150	GQ16	Pre-projection aliases in sort keys
151	GQ17	Element-wise group variable operations
152	GQ18	Scalar subqueries
153	GQ19	Graph pattern YIELD clause
154	GQ20	Advanced linear composition with NEXT
155	GQ21	OPTIONAL: Multiple MATCH statements
156	GQ22	EXISTS predicare: multiple MATCH statements
158	GQ24	FOR statement: WITH OFFSET
174	GT01	Explicit transaction commands

176	GT03	Use of multiple graphs in a transaction
177	GV01	8 bit unsigned integer numbers
178	GV02	8 bit signed integer numbers
179	GV03	16 bit unsigned integer numbers
180	GV04	16 bit signed integer numbers
181	GV05	Small unsigned integer numbers
182	GV06	32 bit unsigned integer numbers
183	GV07	32 bit signed integer numbers
184	GV08	Regular unsigned integer numbers
185	GV09	Specified integer number precision
186	GV10	Big unsigned integer numbers
187	GV11	64 bit unsigned integer numbers
188	GV12	64 bit signed integer numbers
189	GV13	128 bit unsigned integer numbers
190	GV14	128 bit signed integer numbers
191	GV15	256 bit unsigned integer numbers
192	GV16	256 bit signed integer numbers
193	GV17	Decimal numbers
194	GV18	Small signed integer numbers
195	GV19	Big signed integer numbers
196	GV20	16 bit floating point numbers
197	GV21	32 bit floating point numbers
198	GV22	Specified floating point number precision
199	GV23	Floating point type name synonyms
200	GV24	64 bit floating point numbers
201	GV25	128 bit floating point numbers
202	GV26	256 bit floating point numbers
203	GV30	Specified character string minimum length
204	GV31	Specified character string maximum length
205	GV32	Specified character string fixed length
210	GV39	Temporal types: date, local datetime and local time support
211	GV40	Temporal types: zoned datetime and zoned time support
212	GV41	Temporal types: duration support
213	GV45	Record types
214	GV46	Closed record types
216	GV47	Nested record types
217	GV50	List value types
221	GV65	Dynamic union types
222	GV66	Open dynamic union types
225	GV70	Immaterial value types
226	GV71	Immaterial value types: null type support
228	GV90	Explicit value type nullability

## Implementation-dependent elements

UA001 The GQL-environment in any GQL-session is isolated from from any other GQL-session and works on a snapshot of the GQL-environment as it stood at the start of the transaction. Views can be created of objects in other GQL-environments that have granted appropriate privileges to the local principal (see section 5.5 in this document), and subject to those privileges can be accessed and/or modified using HTTP. Such access does not create any congoing link between GQL-environments, and any condition raised by the remote system will cause an exception. See section 6.4 in this document.

UA002 Conditions other than warnings are handled individually: if not handled, any exception condition is reported to the GQL-agent and the current transaction is terminated.

UA004 When expressions are evaluated, only the parts of the expression that contribute to calculating its current result are permitted to raise an exception. This is the usual distinction between compile-time (syntax) errors and run-time errors.

UA005 Path bindings beyond the required number are discarded.

UA006 No additional path bindings are examined.

UA007 Rollback occurs in all cases where a GQL-transaction blocked, cannot complete with causing semantic inconsistency, or the resources available for its execution are insufficient.

US001 The sequence of records in an unordered binding table is determined (in order of priority) by any ordering operations implicitly or explicitly required for the current query, the left-to-right ordering within path and graph expressions, and the timestamp of the first definition of referenced nodes or edges.

- US005 In the absence of ordering of the binding table, the sequence of path bindings is determined by as specified in US001.
- US006 See US001.
- US007 NULLS FIRST is the default, NULLS LAST can be specified. Otherwise see US001.
- US008 The actual order of evaluation of any unparenthesised expression is left to right.
- US009 The request timestamp is set as the start of execution of the implicit or explicit transaction. Security auditing uses the current\_time and is unaffected by the transaction mechanism.
- UV001 The value of an object identifier (a 64-bit integer) is accessible using the keyword POSITION (unless this non-reserved word has been defined to be something else).
- UV003 Such a value expression is inaccessible to the GQL-client.
- UV004 G100 is not supported.
- UV005 The implementation has a class called Common.TypedValue to represent instances of all data types and intermediate values. It has subclasses for records (rows), nodes, and edges.
- UV007 See UV005.
- UV009 See IA017 below.
- UV014 Depends on the .NET runtime library but should not matter for any practical purpose.
- UW001 The first unhandled exception condition will be reported.

## Implementation defined specifications

Codes IA001 etc are as in the GQL specification (ISO 39075).

- IA001 The declared type of a result value is exposed via the PyrrhoReader API: The GQL type name is returned by GetDataTypeName.
- IA002 GQL-status objects are not chained.
- IA003 Unnormalized strings are forbidden. Unicode Normalization C is used.
- IA004 Internally, Pyrrho uses a base-256 representation called **Integer** to achieve high precision numerical accuracy (up to 256 256-complement digits, so that the maximum value of **Integer** exceeds  $2^{2000}$ ). **Numerics** comprise an **Integer** mantissa with a 32-bit scale, and/or a 32-bit exponent. By default, these are the limits for primitive numeric types, with one exception: division guarantees to preserve only 12 decimal digits of precision unless a greater precision is specified. Most clients use or specify much more restrictive arithmetic so apart from Integer division Pyrrho will always be more accurate. For data structures and file positions, Pyrrho uses longs, and conversions to and from Integer for such internal representations are implicit. Pyrrho's implementation size limits and file positions are all 60 bits (see IL001-IL023 below). INT is allowed to contain large integers (**Integer**).
- IA005 See IA004.
- IA006 Does not arise, see IA004.
- IA007 See IA004.
- IA010 See IA004.
- IA011 See IA004.
- IA012 A GQL Flagger for Pyrrho is not currently available. Pyrrho has the following reserved words in addition to the list in ISO39075: CURSOR, INTERVAL0, MULTiset, NCHAR, NCLOB, NUMERIC, REAL0. (See 24.5.3. Keywords that are reserved in SQL but not in GQL are no longer reserved in Pyrrho.)
- IA013 Processing of any transaction is terminated when an exception condition is raised. (As with other SQL syntax, the exception-handling syntax is available provided the SQL keywords have not been overloaded by names of graphs and other persistent objects.)
- IA014 NULL can be cast to any type, but if the type declared NOT NULL another exception will be raised.
- IA015 Strings are not padded for comparison.
- IA016 Strings **do not have final null bytes**.
- IA017 In the situation described in 13.3 GR7, **all possible assignments are made**.
- IA019 The Unicode standard is followed. String literals can contain any Unicode character provided the single-quote character is doubled. (The server does not process escape sequences, octal representations etc. except for its HTTP service, which supports URL encoding for URLs.)
- IA020 Identifiers if double-quoted can contain any Unicode characters except the double quote symbol. **If not double quoted, they are case-insensitive and limited to letters, digits and \_, but \_ is not allowed as the first character of an unquoted identifier.**
- IA021 No exception is raised when truncation occurs when applying the declared precision or length of a value. Otherwise see IA004.
- IA023 GQL source text should **not contain comments**. Newline is treated as white space.
- IA025 Pyrrho relies on Microsoft's .NET 8.0 for IEEE754 floating point operations and does not supply any non-standard values. Exceptions reported under IEEE754 will result in an exception condition in the data value class "22".
- IA026 **Leap seconds are not supported by .NET and therefore are ignored by Pyrrho.**
- ID001 The principal must be identified by the operating system (environment) of the application, or through use of standard HTTP authentication mechanisms. Communication with the server encrypts connection strings in the



open-source PyrrhoLink library. (The server itself runs in an ordinary account that owns the transaction logs. If no users are defined in a transaction log, it is the only authorised user.)

ID002 The principal must be the owner of the home graph or be granted suitable privileges on it. Stored procedures and internal operations of declared typed have the permissions of the principal who defined them. See also ID049.

ID003 The authentication and privileges machinery follows ISO 9075, with two changes (a) ownership of the database can be granted and (b) granting privileges to users on objects other than roles is deprecated (users can be granted usage rights on roles).

ID004 By default, the list element type is determined by the primitive data type of the first element of the list (that is, its declared type, or one of the standard types **String**, **Integer**, **Numeric**, **Real**). If the first element is an *undeclared* array, multiset, set, row, document or table type, the type of its sub-elements defaults to **Content**.

ID005 See ID004.

ID006 The default transaction mode is determined by the privileges of the current user (see ID001). Subject to those privileges, the results of data-modifying statements are committed to persistent storage at the end of a transaction. The condition code NODATA can be handled by a declared exception handler as in SQL. If a condition code indicating an error or exception is raised during execution, and not handled as in SQL, the transaction is terminated (with an implicit rollback command), and diagnostic information (condition code text and contained strings) is returned to the client. The PyrrhoLink API allows the client to request the contents of the diagnostic information as strings. Transaction control statements other than rollback are not allowed in procedure code.

ID016 Condition texts can be localized on installation of Pyrrho. Localization files is done by the client library using client-side localization files and regional settings, however, the current distribution supports only English condition texts. By default, the neutral culture is used.

ID017 See ID006.

ID022 If no users are defined in the database, the default Unicode collation is used. If at least one user has been defined, the default collation is that of the current user. The current string expression can specify a collation (locale) as in SQL.

ID023 STRING and CHAR are synonyms, are used interchangeably and have maximum length  $2^{31}-1$ . Other character string types with unspecified length, including VARCHAR, NCHAR, CLOB etc., are also synonyms for STRING. BLOB is available, using HEX representation as in SQL, and is a synonym for VARBINARY.

ID028 For intermediate results of arithmetic expressions, see IA004. Values declared INT are not truncated. Truncation to INT8, INT16, INT32, INT64, or decimal precision, is applied on assignment.

ID034 For intermediate results of arithmetic expressions, see IA004. Values declared NUMERIC are not truncated or rounded (except on division see IA004). Rounding to a specified decimal scale and precision is applied on assignment.

ID037 For intermediate results of arithmetic expressions, see IA004. Values declared REAL are not truncated or rounded (except on division see IA004). Rounding to FLOAT, FLOAT16, FLOAT32, FLOAT64, FLOAT128, FLOAT256, REAL or DOUBLE or a specified decimal scale and precision is applied on assignment.

ID048 Universal time is used internally. If a user has been defined in the database, the current user's time zone is applied on assignment.

ID049 If no user has been defined in the database, the current user is the account that started the server, and no network access is allowed. Otherwise, the authorization identifier and principal are obtained from the operating system for the current user (**users may not simply say who they are**). Their time zone, locale and regional settings as advised by the operating system are used for default settings. The authorization principal may grant (role based as in SQL) access privileges to specified users as authenticated by the operating system. The current user can only use one role at a time. Otherwise, **session parameters cannot be set or accessed by users**.

ID057 INT64.

ID058 INT64.

ID059 See IA004.

ID061 See ID023.

ID062 See IA004.

ID063 See IA004.

ID064 See IA004.

ID065 See IA004.

ID066 See IA004.

ID067 See IA004.

ID068 INT64.

ID069 These values are supplied by NET 8.0.

ID070 INT64.

ID074 See IA004, ID028 and ID034.

ID075 See IA004 and ID037.

ID076 INT64. Nodes committed to the database have file positions in the range **0..2<sup>60</sup>-1**. Numbers above this range are used for intermediate or uncommitted results.

ID079 See IA004.

ID085 There is a special value called TNull.Value whose domain is Domain.Null. These are singleton values.

ID086 The Match mode is REPEATABLE ELEMENTS.

ID089 GRAPH.

ID090 NODE.

- ID091 EDGE.
- ID095 See IA004.
- ID096 See IA004.
- ID097 See IA004.
- ID098 See IA004.
- ID099 See IA004.
- IE001 An external object reference conforming to RFC7230 can identify an HTTP service endpoint.
- IE002 See ID006. Transaction isolation is **SERIALIZABLE** and cannot be changed.
- IE003 Identifiers containing characters other than letters and digits, beginning with underscore, or matching a GQL reserved word **or one of those mentioned in IA012**, must be double-quoted.
- IE004 No exceptions to serializable behavior are permitted. Deferred triggers are performed at the validation step at the end of a transaction.
- IE005 See ID006.
- IE006 All modifications are under transaction control. Transactions can include catalog-modifying operations and data modifying operations in any order subject only to reference dependency (e.g., nodes must be inserted before edges that connect them).
- IE007 See IE006.
- IE008 None.
- IE009 See Pyrrho manual section 8.1.1 SQLSTATE.
- IE010 None.
- IL001  $0..2^{60}-1$ .
- IL002  $2^{60}-1$ .
- IL003  $2^{60}-1$ .
- IL009  $2^{31}-1$ .
- IL010 See IA004.
- IL011 See IA004.
- IL013  $2^{31}-1$ .
- IL015  $2^{60}-1$ .
- IL018  $2^{60}-1$ .
- IL020  $2^{60}-1$ .
- IL023  $-(2^{31}-1)..(2^{31}-1)$ .
- IL024 **Limited by the precision of the duration**, for which see IA004.
- IS001 NULLS FIRST.
- IV001 Unicode.
- IV002 I have no idea what **“essentially comparable”** means. Custom ordering functions are available as in SQL.
- IV003 As in <node type pattern>.
- IV008 The normal forms of all types are the same as their declared types.
- IV010 I have no idea what **“universally comparable”** means. See IV002.
- IV011 Closed union types are supported. There is a type **Content** for open dynamic types: it does not support arithmetic or comparison. In both cases there is an attempt to infer a specific type from the value.
- IV012 See IV011.
- IV014 See IA004.
- IV015 See ID049.
- IV016 See Pyrrho manual section 8.1.1.
- IV023 Unicode.
- IW001 See the PyrrhoConnect API in the Pyrrho manual section 8.6.12. The connection follows normal TCP protocols.
- IW002 As SQL.
- IW003 Termination of the client connection.
- IW004 See IW001 and ID006.
- IW005 See the PyrrhoConnect API (Pyrrho manual section 8.6.12) and DatabaseError (section 8.6.2).
- IW006 See the PyrrhoConnect API (section 8.6.12) for the Prepare and Execute methods.
- IW007 See the DatabaseError API (section 8.6.2).
- IW010 External procedures are not supported.
- IW011 See the PyrrhoReader API (section 8.6.17) for the GetDataTypeName and GetSubtypeName methods.
- IW012 see IW011.
- IW014 No attempt is made to determine if unequal strings are visually confusable.
- IW015 GQL-directories are recorded as persistent objects in the database (i.e. transaction log). They do not correspond to anything else in the operating system.
- IW016 GQL-schemas are recorded as persistent objects in the database (i.e. transaction log). They do not correspond to anything else (any other files) in the operating system.
- IW017 Unnormalized strings are not supported.
- IW018 If CAST(VE AS VT) succeeds, the result is of type VT. For value types that have supplied precision and/or scale constraints the CAST can alter these. Numeric types can be cast to other numeric types. If VT is STRING the string representation of VE that might be returned to the client is generated: e.g. where VE is subtype of a standard type such as INT, REAL, or DATE, or a collection type, or graph type.

IW019 The common supertype of a set of value types will relax precision as required and set scale to zero unless there is a common value for the scale.

IW021 **Not implemented.**

IW022 All values are nullable unless specified otherwise (by NOT NULL).

IW023 The canonical name of an identifier is its string representation without quotes.

IW025 See IE006.

## 8.2 Sys\$ table collection

Sys\$Audit, Sys\$Role, and Sys\$User list all of the corresponding objects in the current database. The Sys\$ tables are read-only and available only to the database owner: the only way to change anything in a database is by means of the APIs provided e.g. SQL or REST.

### 8.2.1 Sys\$Audit

Field	Data Type	Description
Pos	Position	The location of this access record in the transaction log
User	Char	The name of the accessing user
Table	Position	The defining position of the sensitive or classified object
Timestamp	Int	The time of the access in ticks

Audit records are only for committed sensitive data. Entries come from physical Audit records, and are added immediately on access (do not wait for transaction commit).

### 8.2.2 Sys\$AuditKey

Field	Data Type	Description
Pos	Position	The location of the access record in the transaction log
Seq	Int	The ordinal position of the key (0 based)
Col	Position	The defining position of the key column
Key	Char	A string representation of the key value at this position

Key information for audit records comes from the filters used to access a sensitive object. For example, if a record is inserted in a table, there is no applicable filter, the audit record will apply to the whole table, and there will be no key information here.

### 8.2.3 Sys\$Classification

Field	Data Type	Description
Pos	Position	The defining position of this record in the transaction log
Type	Char	The object type
Classification	Char	Readable version of Level as in 7.2
LastTransaction	Position	The most recent transaction for this object

This table contains information for all current objects and data records with classification different from D. The order is not specified. Rows are not included unless the whole row is classified (see Sys\$ClassifiedColumnData). The key in this table is Pos.

### 8.2.4 Sys\$ClassifiedColumnData

Field	Data Type	Description
Pos	Position	The defining position of the record in the transaction log
Col	Position	The Column's defining position
Classification	Char	Readable version of Level as in 7.2 for the contents
LastTransaction	Char	The most recent transaction for this record

This table contains information for current records affecting columns whose classification is different from D, excluding records contained in Sys\$Classification. The order is not specified. The key in this table is (Pos,Col).

### 8.2.5 Sys\$Enforcement

Field	Data Type	Description
Name	char	The Table name
Scope	char	Enforcement flags

By default classification is enforced for all operations: there will be entries in this table only for tables with specified enforcement levels. There may also be an entry for the table in Sys\$Classification.

### 8.2.6 Sys\$Graph

Field	DataType	Description
Uid	char	The uid of the representative node of a graph in the database
Id	char	The id of the representative node of a graph in the database
Type	char	The node type of this node

If a database defines graph data, the database manages a list of the disjoint graphs it contains. There is a row in this system table for each of these disjoint graphs, arbitrarily selecting a representative node in each. Pos and Id uniquely identify nodes in the database (though they may be the same string). There is a base table with the same name as the node type, and Id is a primary key for this table. Rows in such a table are TNodes, each giving access to multisets of leaving and arriving TEdges.

The NodeType defines its possible leaving and arriving edge types, and has a base table whose rows are TEdges, each with a primary key **id** and columns giving access to its **leaving** TNode and **arriving** TNode.

### 8.2.7 Sys\$Role

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Name	Char	The Role identifier
Details	Char	A readable description of the intended use of the role

(Pos) and (Name) are keys in this table.

### 8.2.8 Sys\$RoleUser

Field	DataType	Description
Role	Char	The Role identifier
User	Char	A User identifier allowed to use this role

(Role,User) is the key in this table.

### 8.2.9 Sys\$ServerConfiguration

Field	DataType	Description
Property	Char	Currently one of AllowDatabaseCreation (true), SegmentationBits (35), ValueRowSetLimit (0=no limit), IndexLimit (0=no limit)
Value	Char	The value of this configuration setting.

### 8.2.10 Sys\$User

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Name	Char	The User identifier
Initial Role	Char	The initial Role for the user
Clearance	Char	Readable version of Level as in 7.2

Users are created in the database the first time they are assigned privileges. (There is no CREATE USER in SQL2023.) Users cannot be renamed. (Pos) and (Name) are keys in this table.

## 8.3 Role\$ table collection

These tables give information about objects seen from the current role.

The Role\$ tables (like all other system tables) are read-only: the only way to change anything in a database is by means of the APIs provided e.g. SQL or REST.

### 8.3.1 Role\$Class

Field	DataType	Description
-------	----------	-------------

Name	Char	The name of a base table or view, with the same name as the class
Key	Char	A comma separated list of the key columns of this object if any
Definition	Char	A C# class definition suitable for receiving rows of this object. See also Role\$Java below.

Dots in top-level column names coming from views are automatically replaced by underscores.

### 8.3.2 Role\$Column

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Table	Char	The current name of the Table or View
Name	Char	The current name of the Column
Seq	Int	The current position in the row (there may be gaps in the sequence here due to columns inaccessible from the current role)
Domain	Domain	The data type for the Column (Position if user-defined)
DefaultValue	Char	String representation of the default value
NotNull	Boolean	Whether the column has been defined NOT NULL
Generated	Boolean	Whether the column is GENERATED ALWAYS
Update	Char	The update statement for a generated column

(Pos) and (Table,Name) are keys.

### 8.3.3 Role\$ColumnCheck

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Table	Char	The current name of the Table
Name	Char	The current name of the Column
CheckName	Char	The current identifier for the CHECK (unique per domain)
Select	Char	The RowSetSpec used to check the VALUE

(Pos) and (Table,Name,CheckName) are keys

### 8.3.4 Role\$ColumnPrivilege

Field	DataType	Description
Table	Char	The current name of the table in the current role
Name	Char	The current name of the column in the current role
Grantee	Char	The Grantee name (a Role)
Privilege	Char	The privilege granted

(Table,Name,Grantee) is the key.

### 8.3.5 Role\$Domain

For further information about domains that are tables, see 8.3.21 and 8.3.7.

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Name	Char	The current identifier for the DOMAIN. May have forms such as CHAR(6), U(5005) if not user-defined (see note below).
DataType	Char	The data type
DataLength	Int	The data length (precision for DECIMAL, REAL, INTEGER)
Scale	Int	The scale (for Numeric type)
StartField	Char	The start field (for Interval type)
EndField	Char	The end field (for Interval type)
DefaultValue	Char	String representation of the default value
Struct	Char	Type string for MULTISSET or ARRAY or ROW element
Definer	Char	The owning role

Pyrrho creates a new domain for each new type in the database (e.g. CHAR(6) ), and makes a special domain for evaluating generated columns. (Pos) and (Name) are keys.

### 8.3.6 Role\$DomainCheck

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
DomainName	Char	The current identifier for the DOMAIN
CheckName	Char	The current identifier for the CHECK (unique per domain)
Select	Char	The RowSetSpec used to check the VALUE

(Pos) and (DomainName, CheckName) are keys in this table

### 8.3.7 Role\$EdgeType

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
Name	Char	The current identifier for the EDGETYPE
LeavingNodeType	Char	The current identifier for the leaving node type
ArrivingNodeType	Char	The current identifier for the arriving node type
IdName	Char	The current identifier for the identity column if any
LeavingName	Char	The current identifier for the leaving column
ArrivingName	Char	The current identifier for the arriving column

(Pos) is the key in this table. Where edge types have the same name, they are disambiguated in GQL by the attached noode types: in SQL the name refers to the first, and the numeric Pos can be used to refer to the others. See also table 8.3.21, 8.3.22, and 8.3.27.

### 8.3.8 Role\$GraphCatalog

Enumerates specified schemas and directories.

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
PathOrName	Char	The catalog parent and name
Type	Char	one of Directory, Schema, Graph, GraphType
Owner	Char	The owner of the schema

(Pos) and (PathOrName) are keys in this table: the latter is used for ordering the rows.

### 8.3.9 Role\$GraphEdgeType

Enumerates specified edge types of a closed graph or graph type.

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
Parent	Position	The defining position of the parent graph or graph type.
Name	Char	The current name of this edge type in the role
Owner	char	The owner of the edge type

(Pos) and (PathOrName) are keys in this table. The edges in a given edge type can be enumerated using select \* from *edgetype*.

### 8.3.10 Role\$GraphInfo

Field	Data Type	Description
Name	Char	A graph info name (see table below)
Value	Int	

The available items are as follows:

Name	Meaning
schemas	Total number of schemas in the catalog
directories	Total number of directories in the catalog
graphs	Total number of graphs in the catalog
graph-types	Total number of graph types specified in the current graph*

nodes	Total number of node instances in the current graph
edges	Total number of edge instances in the current graph
properties	Total number of different property names in the current graph
labels	Total number of distinct labels in the current graph
label-sets	Total number of distinct label sets specified in the current graph

\* This is not the same as the number of different data types of nodes and edges in the current graph. See section 5.9.3.

### 8.3.11 Role\$GraphLabel

Enumerates the members of the label set of a closed node type or edge type.

Field	DataType	Description
Pos	Position	System key (position information) for the label
Graph	Position	The defining position of the graph or graph type
Parent	Position	The defining position of the closed node type or edge type
Label	Char	The current name of this label in the role

### 8.3.12 Role\$GraphNodeType

Enumerates the specified node types of a closed graph or graph type.

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Parent	Position	The defining position of the parent graph
Name	Char	The current name of this edge type in the role
Owner	Char	The owner of this node type

### 8.3.13 Role\$GraphProperty

Enumerates the specified properties of a closed node type or edge type.

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Label	Position	The defining position of the label
Name	Char	The current name of this property in the role

### 8.3.14 Role\$Index

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Table	Char	The current name of the table
Name	Char	The name of the index (see note)
Flags	Int	Sum of values in table below
RefTable	Char	Name of referenced table (or null)
RefIndex	Char	Name of referenced index (or null)
Distinct	Int	Number of distinct values
Adapter	Char	Name of adapter function or method (or null)
Rows	Int	The number of rows in the index

User indexes are not supported in SQL2023. Pyrrho builds indexes automatically for all primary, unique, and foreign keys (there is no CREATE INDEX) in order to enforce integrity and referential constraints. They have names like U(67). (Pos) is the key for this table

Flag	Meaning
1	Primary Key
2	Foreign Key
4	Unique
8	Descending
16	Restrict Update

32	Cascade Update
64	Set Default Update
128	Set Null Update
256	<i>Restrict Delete</i>
512	Cascade Delete
1024	Set Default Delete
2048	Set Null Delete

The Restrict flags are currently unused, since RESTRICT is the default and is only overridden if CASCADE or SET NULL has been set.

### 8.3.15 Role\$IndexKey

Field	Data Type	Description
IndexName	Char	The name of the index
TableColumn	Char	The current name of the column
Position	Int	Zero-based column position in the index
Flags	Char	<i>Blank except for Mongo</i>

(IndexName, TableColumn) and (IndexName, Position) are keys in this table.

### 8.3.16 Role\$Java

Field	Data Type	Description
Name	Char	The name of a base table or view, with the same name as the class
Key	Char	A comma separated list of the key columns of this object if any
Definition	Char	A Java class definition suitable for receiving rows of this object. See also Role\$Class

Dots in top-level column names coming from views are automatically replaced by underscores.

### 8.3.17 Role\$Method

Field	Data Type	Description
Name	Char	The identifier for the type
Method	Char	The name of the method
MethodType	Char	Instance, Constructor, Static, or Overriding
Definition	Char	The method body
Definer	Char	The owning role

### 8.3.18 Role\$NodeType

This table contains all the node types accessible from this role. (If there several edge types with the same name, there will also be a node type that has the columns they share.)

Field	Data Type	Description
Pos	Position	Defining position (System key) for this object
Name	Char	The identifier for the node type
IdName	Char	The name of the identifier column if any

(Pos) is the key in this table. The nodes of a given node type can be enumerated by select \* from *nodetype*.

### 8.3.19 Role\$Object

Field	Data Type	Description
Pos	Position	Defining position (System key) for this object
Type	Char	The type of database object, e.g. Table, Role etc
Name	Char	The current name of the database object in this role
Description	Char	The object description (at creation)
Iri	Char	The object iri if defined (for Domain, at creation)
Metadata	Char	Metadata for the current role



(Pos) and (Type, Name) are keys in this table. For the available Metadata flags see section 7.2 (page 51 at the last count).

### 8.3.20 Role\$Parameter

Field	DataType	Description
Pos	Position	System key (position information) for the procedure or method
Seq	Int	The ordinal of the parameter
Name	Char	The name of the parameter
Type	Char	The name of the parameter's type
Mode	Char	In or None, Out, InOut, Result

(Pos, Seq) is the key in this table.

### 8.3.21 Role\$PrimaryKey

Field	DataType	Description
Pos	Position	System key (position information) for the table or type
Table	Char	The current name of the table
Ordinal	Int	The position of the column in the primary key
Column	Char	The current name of the column

(Pos,Ordinal) and (Table, Ordinal) are keys in this table.

### 8.3.22 Role\$Privilege

Field	DataType	Description
Pos	Position	System key (position information) for the object
ObjectType	Char	The kind of object for which the privilege is granted
Name	Char	The name of the object for which the privilege is granted: for columns, methods etc this may have form id.id..
Grantee	Char	The Grantee name
Privilege	Char	The privilege granted
Definer	Char	The owning role of the granted object

(Pos, Name, Grantee) is the key in this table. Tables can have delete permission in this table, but Select, Insert and Update apply to columns.

### 8.3.23 Role\$Procedure

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Name	Char	The current name of the Procedure or Function
Returns	Char	The return type (Null for a Procedure)
Definition	Char	The string containing the current procedure or function definition
Inverse	Char	The name of the inverse function if any
Monotonic	Boolean	Whether the function has been declared monotonic
Definer	Char	The owning role (body will run as this role)

The Definition starts from the beginning of the parameter list. (Pos) is the key in this table. For the signature information, see 8.3.21.

### 8.3.24 Role\$Python

Field	DataType	Description
Name	Char	The name of a base table or view, with the same name as the class
Key	Char	A comma separated list of the key columns of this object if any
Definition	Char	A Python class definition suitable for receiving rows of this object. See also Role\$Class

### 8.3.25 Role\$SQL

Field	DataType	Description
-------	----------	-------------

Name	Char	The name of a base table or view
Key	Char	A comma separated list of the key columns of this object if any
Definition	Char	A table definition and schema key suitable for inclusion in a RESTView definition

### 8.3.26 Role\$Subobject

Field	DataType	Description
Type	Char	The type of database object, e.g. Table, Role etc
Name	Char	The current name of the database object
Seq	Int	The ordinal position of the column
Column	Char	The name of the column
Output	Char	Metadata
Description	Char	Metadata
Iri	Char	Metadata

The primary key in this table is (Type, Name, Seq). For the available Metadata flags see section 7.2. TableColumns are found in the Role\$Object table: the Role\$Subobject table is for columns in views and the tables returned from functions.

### 8.3.27 Role\$Table

Field	DataType	Description
Pos	Position	System key (position information) for the current object
Name	Char	The name of the Table
Columns	Int	The number of columns
Rows	Int	The number of rows
Triggers	Int	The number of triggers
CheckConstraints	Int	The number of Table check constraints
RowIri	Char	The Iri type constraint for rows if defined

(Pos) and (Name) are keys in this table.

### 8.3.28 Role\$TableCheck

Field	DataType	Description
Pos	Position	System key (position information) for the current object
TableName	Char	The current identifier for the Table
CheckName	Char	The identifier for the CHECK (unique per table)
Select	Char	The RowSetSpec used to check the VALUE

(Pos) and (TableName,CheckName) are keys in this table

### 8.3.29 Role\$TablePeriod

Field	DataType	Description
Pos	Position	System key (position information) for the current object
TableName	Char	The current name of the table
Period Name	Char	The name of the Period (e.g. SYSTEM_TIME)
PeriodStartColumn	Char	The name of the system time period start column
PeriodEndColumn	Char	The name of the system time period end column
Versioning	Boolean	Whether period versioning has been specified

### 8.3.30 Role\$Trigger

Field	DataType	Description
Pos	Position	System key (position information) for the trigger
Name	Char	The name of the Trigger
Flags	Char	Before/After, Insert/Delete/Update
TableName	Char	The current name of the table concerned
Definer	Char	The definer role for the Trigger

(Pos) and (Name) are keys in this table. Use Log\$Trigger to see the defining code.

### 8.3.31 Role\$TriggerUpdateColumn

Field	Data Type	Description
Pos	Position	System key (position information) for the trigger
Name	Char	The name of the Trigger
ColumnName	Char	Column for Update

(Pos) and (Name,ColumnName) are keys in this table

### 8.3.32 Role\$Type

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
Name	Char	The identifier for the type
Supertype	Char	The name of the supertype
OrderFunc	Char	The name of the ordering function if specified
OrderCategory	Char	The string representation of the order category (see 9.2.8)
Subtypes	Char	The number of subtypes of this type
Definer	Char	The owning role
Graph	Char	NodeType or EdgeType if appropriate.

(Pos) and (Name) are keys in this table.

### 8.3.33 Role\$View

Field	Data Type	Description
Pos	Position	System key (position information) for the current object
View	Char	The current VIEW identifier
Select	Char	The current corresponding query expression if any
Struct	Char	The structure type (OF)
Using	Char	The name of the GET USING table if any
Definer	Char	The owning role

(Pos) and (View) are keys in this table. See also the Role\$Object table for metadata.

## 8.4 Log\$ table collection

These tables give access to records in the transaction log. They retain their system key (defining position) throughout their lifetime, but all other details including their name are subject to modification by later entries in the transaction log. In particular, Types can later become NodeTypes or EdgeTypes as a result of Metadata.

The Log\$ tables identify all objects by (long) integer values, shown in the Sys\$ tables as Pos, and in the Log\$ tables as DefPos (the defining position of the object, i.e. the log entry which records the creation of the object).

The Log\$ table itself shows full details of each physical record in the database transaction log. Other tables in this collection break out much of the information in tables that are easier to access from SQL.

Tables in this collection are read-only. They are always available to the database owner.

### 8.4.1 Log\$

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Desc	char	A semi-readable version of the log information
Type	Char	The type of log entry (see table below)
Affects	Char	The object affected*
Transaction	Char	The transaction this record belongs to

\*Affects is a single defining position added to this table for convenience (it is not actually in the log file). For log entries that cause cascading changes, there is no attempt to provide details of the consequential

actions, which occurred at the time the log entry was laid down and will occur again when the database loads.

Type	Further information in	Comments
Alter, Alter2, Alter3	Log\$Alter	Alter column properties
Authenticate	Log\$Authenticate	
Change	Log\$Change	Rename object
Delete, Delete1, Delete2	Log\$Delete	
DeleteReference1		
Drop, Drop1	Log\$Drop	
Edit	Log\$Edit	Alter domain properties
Grant	Log\$Grant	
Index	Log\$Index, Log\$IndexKey	
Metadata, Metadata2, Metadata3	Log\$Metadata	
Modify	Log\$Modify	Alter proc/func/method
Namespace		
Ordering	Log\$Ordering	
PCheck, PCheck2	Log\$Check	
PColumn, PColumn2, PColumn3	Log\$Column	
PDateType	Log\$DateType	
PeriodDef	Log\$PeriodDef	
PProcedure, PProcedure2	Log\$Procedure	
PRole, PRole1	Log\$Role	
PSchema		
PTable, PTable1	Log\$Table, Log\$Column	
PTransaction	Log\$Trigger, Log\$TriggerUpdateColumn	
PType, PType1, PType2	Log\$Type	See also PDomain
PUser	Log\$User	
PView	Log\$View	
Record, Record1, Record2, Record3, Record4	Log\$Record, Log\$RecordField	
RestView1, RestView2		See PView
Revoke	Log\$Revoke	
Transaction	Log\$Transaction	
Update, Update1, Update2	Log\$Update, Log\$RecordField	
Versioning	Log\$Versioning	

### 8.4.2 Log\$Check

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Ref	Char	The object (table, column etc)referred to
ColRef	Char	The column referred to (Check2) or -1
Name	Char	The original name of the constraint (possibly system supplied)
Check	Char	The source code for the check condition

### 8.4.3 Log\$Classification

This table contains all log entries for database objects that change the classification. For Records see Log\$Update.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Obj	Position	The defining position of the object affected
Classification	Char	D to A

### 8.4.4 Log\$Clearance

This table contains all log entries that change the clearance of users.

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
User	Position	The defining position of the user affected
Clearance	Char	D to A

### 8.4.5 Log\$Column

This table also provides information for the subclasses of Column such as Column3 and Alter3. The details represent a snapshot of this column when this log entry is installed.

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Defpos	Position	The defining position of the column (will be different for Alter)
Table	Position	The defining position of the table
Name	Char	The name of the column in this log entry
Seq	Int	The ordinal position of the column
Domain	Char	The specified domain
Default	Char	The string if specified for a default value
NotNull	Boolean	Whether the column must have a non-null value
Generated	Boolean	Whether GENERATED ALWAYS
Update	Char	The update assignment rule for a generated column
Flags	GraphFlags	Extra information for columns in node and edge types
RefIndex	Position	The defining position of a simply referenced index
ToType	Position	The defining position of a simply referenced node type

### 8.4.6 Log\$DateType

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The original name for the date type domain
Kind	Char	The base type of the date type (e.g. INTERVAL)
StartField	Char	The start field for the date type
EndField	Char	The end field for the date type

### 8.4.7 Log\$Delete

Field	Data Type	Description
Pos	Position	System key (position information) for the delete operation
DelPos	Char	The defining Pos for the record

### 8.4.8 Log\$Domain

This table also provides details for Edit, Type, and DateType log entries.

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Kind	Char	Domain, Edit, or Type
Name	Char	The name of the domain or type
Data Type	Int	Describes the data type
DataLength	Int	Length of the data type
Scale	Int	Scale factor for numerics
Charset	Char	Character set identifier
Collate	Char	The collation identifier
Default	Char	String representation of default value
StructDef	Char	Domain reference for MULTiset or ARRAY element, or Table reference for ROW or TYPE element

### 8.4.9 Log\$Drop

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
DelPos	Position	The defining position of the object being deleted

### 8.4.10 Log\$Enforcement

Field	Data Type	Description
Pos	Position	System key (position information) for the Alter Domain operation
Table	Position	The defining position of the table
Flags	Int	Enforcement flags (read,insert,update,delete) see 9.2.7

### 8.4.11 Log\$Grant

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Privilege	Int	Describes the privilege granted see 9.2.7
Object	Position	The object for which the grant is made
Grantee	Position	The object gaining the privilege

### 8.4.12 Log\$Index

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The name of the index (system generated, e.g. U(nnn))
Table	Position	The table on which this index is defined
Flags	Char	Describes this index, see 9.2.5
Reference	Position	Identifies the referenced index
Adapter	Position	Identifies the adapter function if any

### 8.4.13 Log\$IndexKey

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
ColNo	Int	The ordinal position of the column in the key
Column	Position	Identifies the key column by defining position

### 8.4.14 Log\$Metadata

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
DefPos	Position	The defining position of the database object
Name	Char	The new name of the object as viewed from this role
Description	Char	The object description for this role
RefPos	Position	The defining position referred to (if any)
Detail	Char	Web metadata for this role.

This table will not include NODETYPE and EDGETYPE metadata. See Log\$Type.

### 8.4.15 Log\$Modify

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
DefPos	Position	The defining position of the proc/func/method being modified
Name	Char	The new name of the object; or update assignments for Column; for View, one of Name,Query,Update,Insert,Delete
Body	Char	The modified source code of the proc/func; for View Name, the new name

### 8.4.16 Log\$Ordering

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
TypeDefPos	Char	The defining position of the type being ordered
FuncDefPos	Char	The defining position of the function or method
OrderFlags	Int	The ordering category flags (see 9.2.8)

### 8.4.17 Log\$Procedure

This table also provides information for Methods.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The original name of the procedure
Arity	Int	Number of parameters
RetDefPos	Position	The defining position of the return type
Proc	Char	The original source code of the proc/func (including the formal params)

### 8.4.18 Log\$Record

This table also provides information for Updates.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Table	Position	The defining position of the table for the insert
SubType	Char	The defining position of the subtype if specified
Classification	Char	D to A

### 8.4.19 Log\$RecordField

This table also provides information for Updates.

Field	DataType	Description
Pos	Position	System key (position information) for the current log entry
ColRef	Position	Identifies the column
Data	Char	String version of the data*

\* As interpreted using the Domain that applied at the time.

### 8.4.20 Log\$Revoke

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Privilege	Int	Identifies the privilege being revoked
Object	Position	The object to which the privilege relates
Grantee	Position	The grantee from whom the privilege is being withdrawn

### 8.4.21 Log\$Role

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The Name of the role
Details	Char	The description of the intended use of the role

### 8.4.22 Log\$TablePeriod

This table records details of period type definitions.

Field	DataType	Description
Pos	Position	System key (position information) for this log entry
Table	Position	The defining position of the table
PeriodName	Char	The original name of the period (or SYSTEM_TIME)
Versioning	Boolean	Whether system versioning is specified

StartColumn	Position	The defining position of the system time period start column
EndColumn	Position	The defining position of the system time period end column

### 8.4.23 Log\$Transaction

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
NRecs	Int	The number of log entries following
Time	TimeStamp	A timestamp
User	Position	Identifies the current user
Role	Position	Identifies the current role

### 8.4.24 Log\$Trigger

The table records trigger definitions.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The original name of the trigger
Flags	Char	Before/After, Insert/Delete/Update
Table	Char	The identifier of the table concerned
OldRow	Char	Referencing identifier for old row
NewRow	Char	Referencing identifier for new row
OldTable	Char	Referencing identifier for old table
NewTable	Char	Referencing identifier from new table
Def	Char	The original code for the trigger including WHEN if defined

### 8.4.25 Log\$TriggerUpdateColumn

This table provides details for trigger definitions.

Field	DataType	Description
Pos	Position	System key (position information) for the trigger
Column	Position	Column for Update

### 8.4.26 Log\$TriggeredAction

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Trigger	Position	Identifies the defining position of the trigger that is starting

Entries of this type in the log show a change of responsibility from the user and role starting the transaction to the defining user and owning role of the trigger.

### 8.4.27 Log\$Type

Field	DataType	Description
Pos	Position	System key (position information) for the Domain log entry
Name	Char	The name of the type
SuperType	Position	Identifies the defining log entry for the supertype
Graph	Char	NodeType or EdgeType if appropriate (see also Log\$EdgeType)

The type name is given in the Log\$Domain table. The list of methods is in the Log\$TypeMethod table. The list of members is in the Log\$Table table. The method bodies are in the Log\$Modify table.

### 8.4.30 Log\$TypeMethod

This table records method declarations.

Field	DataType	Description
Pos	Position	System key (position information) identifying this method
MethodType	Int	See coding below
Name	Char	The original name of the method



Value	MethodType
0	Instance
1	Overriding
2	Static
3	Constructor

Method bodies are given in the Log\$Modify table.

### 8.4.31 Log\$update

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
DefPos	Position	Identifies the defining log entry for the record
Table	Position	Identifies the table for the update
SubType	Position	Identifies the subtype if any
Classification	Char	D to A

### 8.4.32 Log\$user

This table records the first occurrence of a user identity in the database.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The name of the user

### 8.4.3 Log\$view

This table records view definitions.

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Name	Char	The original name of the View
Select	Char	The original query expression defining the view
Struct	Position	The defining position of the structure tpe (OF) if any
Using	Position	The defining position of the GET USING table if any

## 8.5 Table and Cell Logs

In auditing databases (section 3.5), it is convenient to be able to review all insert, update, and delete operations for a specific table, or for a specific cell. Pyrrho provides table and cell log facilities to do this, provisionally referred to as ROWS(nnnn) and ROWS(rrr,ccc) where nnnn is the numeric identifier of the table in question, rrr the defining position of the desired row, and ccc that of the desired column.

### 8.5.1 A Table Log

Pyrrho provides a table log facility, provisionally referred to as ROWS(nnnn) where nnnn is the numeric identifier of the table in question. ROWS(nnnn) is a table with the following fields:

Field	DataType	Description
Pos	Position	System key (position information) for the log entry
Action	Char	“Insert”, “Update”, or “Delete”*
Transaction	Int	The transaction this log entry belongs to
cccc	Cell	The value specified for the column with identifier ccccc

\*Entries from cascading updates and deletes are not included in this table.

This feature allows data to be recovered even where columns have been removed (by ALTER TABLE or even DROP TABLE).

## 8.5.2 A Cell Log

Pyrrho provides a cell log facility, provisionally referred to as ROWS(rrr,ccc) where rrr is the defining position of the row containing the cell, and ccc the defining position of the column in question. ROWS(rrr,ccc) is a table with the following fields:

Field	Data Type	Description
Pos	Position	System key (position information) for the log entry
Value	Cell	The value
StartTransaction	Int	The transaction responsible for placing this value
StartTimestamp	Timestamp	The timestamp for the StartTransaction
EndTransaction	Int	The transaction responsible for replacing this value
EndTimestamp	Timestamp	The timestamp for the EndTransaction

This feature allows data to be recovered even where the row and/or even the column or table has been removed (by DELETE, or ALTER TABLE, or DROP TABLE).

## 8.6 Pyrrho Class Library Reference

Any application using Pyrrho should include PyrrhoLink.dll. The API is designed to be similar to ADO.Net.

Except where noted, all of these dlls define (export) the following classes, which are described in the following subsections:

SQL2023 API:

Class	Subclass of	Description
Date		Data type used for dates.
PyrrhoArray		
PyrrhoColumn		Helps to describe the columns of a table or structured type
PyrrhoConnect	System.Data.IDbConnection	Establishes a connection with a Pyrrho DBMS server, and provides additional methods and properties.
PyrrhoDocument		This class allows editing of embedded Documents (in the sense of MongoDB)
PyrrhoInterval		This class is used to represent a time interval
PyrrhoRow		Data type used for ROW fields in a database table, a column of type ROW can be added to the table. (SQL2023)

Exceptions:

Class	Subclass of	Description
DatabaseError	System.Exception	Used for “user” exceptions, e.g. a specified table or column does not exist, an attempt is made to create a table or column that already exists, incorrect SQL etc. The message property gives a readable explanation. see section 8.1.
TransactionConflict	DatabaseError	The action attempted has conflicted with a concurrent transaction, e.g. two users have attempted to update the same cell in a table. The changes proposed by the current transaction have been rolled back, because the database contents have been changed by the other transaction.

Class	Subclass of	Description
PyrrhoTable		
PyrrhoTable<T>	PyrrhoTable	

PHP support:

Class	Subclass of	Description
-------	-------------	-------------

ScriptConnect		Provided for PHP support (section 6.7)
ScriptReader		Provided for PHP support (section 6.7)

### 8.6.1 AutoKeyAttribute

Class definitions obtained from Role\$Class may have fields marked [AutoKey].

Attribute form	Explanation
[AutoKey]	An integer or string key field that can be left as null for a new row so that it can be filled in by the server

### 8.6.2 DatabaseError

The methods and properties of DatabaseError are:

Method or Property	Explanation
Dictionary<string,string> info	Information placed in the error: the keys specified in the SQL standard are CLASS_ORIGIN, SUBCLASS_ORIGIN, CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, CATALOG_NAME, SCHEMA_NAME, TABLE_NAME, COLUMN_NAME, CURSOR_NAME, MESSAGE_TEXT. Pyrrho adds PROFILE_ID if profiling is enabled.
String Message	The reason for the exception (inherited from Exception): this can be localised as described in section 3.8.
String SQLSTATE	The signal sent from the DBMS: usually a five character string beginning with a digit such as "2N000". Many of these codes are defined in the SQL standard.

### 8.6.3 Date

The methods and properties of Date are:

Method or Property	Explanation
DateTime date	The underlying DateTime value
Date(DateTime d)	Constructor.
string ToString()	Overridden: Formats the date using DateTime.ToShortDate() which is locale-specific

### 8.6.4 DocArray

Property	Explanation
DocArray(string s)	Create a DocArray from JSON.
C[] Extract<C>(params string[] p)	Extract instances of C from a DocArray. C must have a public parameterless constructor. P is a path of fields in the documents of the array.
List<object> fields	A document array consists of an array of documents

### 8.6.5 Document

PyrrhoConnect.Get/Post/Put/Delete can be used for whole Documents and BSON and Json formats are supported. This class can be used to access fields within Documents and to convert to and from Json  
Note: this class remembers the connection to the database if any, and all these changes are transacted in the database unless the Document is detached or the connection is closed.

Method or Property	Explanation
bool Contains(string k)	Tests if there is a field k in the top level of the document
Document()	Constructor: a new empty Document
Document(object)	Constructor: reflection is used to build a Document based on the public fields of the given parameter
Document(string)	Constructor: the string should be JSON.
C[] Extract<C>(params string[])	Reflection using class C is used recursively to extract instances of C from this document, starting at a place indicated by the given path of keys.

List<KeyValuePair<string,object>> fields	The content of the Document (accessed using this[])
object this[string]	Access a field of the document.
string ToString()	Convert a document to Json

### 8.6.6 DocumentException

This subclass of Exception is used to report parsing errors in Document parameters.

### 8.6.7 ExcludeAttribute

Mark a public field of a Versioned class with the [Exclude] attribute to avoid its use in Put/Post.

### 8.6.8 FieldAttribute

Class definitions obtained from Role\$Class have some fields marked [Field..]

Attribute form	Explanation
[Field(PyrrhoDbType t)]	Pyrrho's data type is t
[Field(PyrrhoDbType t,string i)]	Pyrrho's data type is t, domain info i
[Field(PyrrhoDbType t, string i,long st)]	Pyrrho's data type is t, domain info i, cookie st of dependent field type (e.g. array element type)
[Field(PyrrhoDbType t,long d, string i)]	Pyrrho's data type is t, domain cookie d, domain info i
[Field(PyrrhoDbType t,long d, string i,long st)]	Pyrrho's data type is t, domain cookie d, domain info i, cookie st of dependent field type (e.g. array element type)

### 8.6.9 PyrrhoArray

Method or Property	Explanation
PyrrhoArray(object[])	
object[] data	The values of the array or multiset. Note that the ordering of multiset values is non-deterministic and not significant.

### 8.6.10 PyrrhoColumn

The methods and properties of PyrrhoColumn are:

Method or Property	Explanation
bool AllowDBNull	Whether the column can contain a null value
string Caption	The name of the column
long domain	Pyrrho's domain cookie
string desc	Pyrrho's description of the domain
bool ReadOnly	Whether the column is read-only

### 8.6.11 PyrrhoCommand

PyrrhoCommand implements IDbCommand or imitates it.

From version 5.4, thread-safety is enforced for client-side programming. PyrrhoCommand cannot be shared among threads because methods of the IDbCommand class might be used in another thread to modify the command. PyrrhoConnect can be shared among threads, but there can be at most one command active at any time per connection. As a result, methods such as ExecuteReader will block until the connection is available.

Method or Property	Explanation
string CommandText	The SQL statement for the Command
IDbDataParameter CreateParameter()	The returned object is a PyrrhoParameter.
PyrrhoReader ExecuteReader()	Initiates a database SELECT and returns a reader for the returned data (as in IDataReader). Will block until the connection is available.
(int,PyrrhoReader) ExecuteMatch()	Initiates a database MATCH and returns the number of records added OR (if this number is 0) a reader for the returned data (as in IDataReader). Will block until the connection is available.

object ExecuteScalar()	Initiates a database SELECT for a single value. Will block until the connection is available.
int ExecuteNonQuery(params Versioned[])	Initiates some other sort of Sql statement and returns the number of rows affected. If the transaction automcommits, the given versioned objects have versions updated if affected. Will block until the connection is available.

## 8.6.12 PyrrhoConnect

PyrrhoConnect imitates the IDbConnection interface from ADO.NET and supplies some additional functionality. Additional methods described here provide a RESTful interface\*: Get, Post, Put and Delete.

From version 5.4, thread-safety is enforced for client-side programming. Although the PyrrhoConnect can be shared among threads, there can be at most one transaction and/or command active at any time per connection, and transactions, commands, and readers cannot be shared with other threads. As a result, methods such as BeginTransaction will block until the connection is available.

From v7 the Prepare and Execute methods of the connection manage and use a set of named SQL statements with ? as placeholders for zero or more Literal values. The Execute method provides these SQL literals as actual parameters for each such placeholder. The named statements are available in the PyrrhoConnect instance that defined them.

Method or Property	Explanation
int Act(string sql)	Convenient shortcut to construct a PyrrhoCommand and call ExecuteNonQuery on it. Will block until the connection is available.
Activity activity	(AndroidOSP) Set only. Set the Activity into PyrrhoConnect. This must be done before the connection is opened. E.g. in Activity.OnCreate(bundle) use code such as <pre>conn = new PyrrhoConnect("Files=mydb"); conn.activity = this; conn.Open();</pre> Note that mydb (without the osp extension) needs to be an AndroidAsset to be copied to the device.
PyrrhoTransaction BeginTransaction()	Start a new isolated transaction (like IDbTransaction). Will block until the connection is available. [In Java, PyrrhoJC.Connection does this automatically if autoCommit has been set false.]
PyrrhoReader Call(string name, Document doc)	Call a named procedure or function, supplying the document contents as named parameters. Each Read() gives a Document version of a result row for a function or a single empty Document on successful execution of a procedure.
bool Check(string ch) bool Check(string ch, string rc)	Check to see if a given Versioned check string is still current, i.e. the row has not been modified by a later transaction. (See sec 5.2.3 and 8.7.21). The second version shown also tests the readCheck. (There is no need to perform a check unless the Versioned data is from a previous transaction.)
void Close()	Close the channel to the database engine
string ConnectionString	Get the connection string for the connection
PyrrhoCommand CreateCommand()	Create an object for carrying out an Sql command (as in IDbCommand).
void Delete(Versioned ob)	Delete the row corresponding to this object.* Will block until the connection is available.
int Execute (string name, params string[] actuals)	Execute the named prepared statement with the given actual parameters (given as SQL Literals). (For the more familiar ExecuteReader and ExecuteNonQuery, see PyrrhoCommand, sec 8.7.11). See Prepare() below.
PyrrhoReader ExecuteReader (string name, params string[] actuals)	
E[] FindAll<E>()	Retrieve all Versioned entities of a given type.* Will block until the connection is available.
E FindOne<E>(params IComparable[] w)	Retrieve a single entity of a given Versioned type E with key fields w.* The Role\$Class table generates classes that provide a static With method that uses this generic function.

E[] FindWith<E>(string w)	Retrieve a set of Versioned entities satisfying a given condition. w is a comma-separated set of conditions of form <i>field=value</i> . Field names are case sensitive and values are in SQL format (single quotes on strings are optional in the absence of ambiguity).* Will block until the connection is available.
E[] Get<E>(string rurl)	The rurl should be a partial REST url (the portion following the Role component), that targets a class E in the client application.* Will block until the connection is available.
string[] GetFileNames	Returns the names of accessible databases.
BTree<string,int> GetInfo()	Returns information about graphs. See section 8.3.8.
PyrrhoColumn[] GetInfo(string dataType)	Get information about a datatype: the string must exactly match the datatype name of a table or type.
void Open()	Open the channel to the database engine
void Post(Versioned ob)	The object should be a new row for a base table.* If autoKey is set key field(s) containing default values (0,"" etc) in ob are overwritten with suitable new value(s). Will block until the connection is available.
void Prepare(string name, string sql)	Prepare a named statement. The sql can contain ? placeholders for actual parameters, which are supplied as SQL fragments in Execute. Prepared statements are local to the current PyrrhoConnect. The Prepare function should not be called within a transaction.
void Prepare(string e, string s, string d, string n)	Prepare a rule to avoid conflicts in graph input: edge type e from source node type s to destination node type d should use edge type n.
void Put(Versioned ob)	The object should be an updated version of an entity retrieved from or committed to the database.* Will block until the connection is available.
PyrrhoConnect(string cs)	Create a new PyrrhoConnect with the given connection string. Documentation about the connection string is in section 6.3.
void ResetReader()	Repositions the IDataReader to just before the start of the data
void SetRole(string s)	Set the session role for the connection
E[] Update<E> (Document w, Document u)	Specifies a Document update operation on a Versioned class containing documents. Documents matching w are updated according to the operations in u, and the set of modified objects is returned. (See 8.8.4)* Will block until the connection is available.
DatabaseError[] Warnings	Warnings for the most recent operation on the connection

\* The Find., Get, Put, Post, Delete and Update methods assume that the Version subclasses corresponding to the relevant database tables have been installed in the application, for example using the sources provided by the Role\$Class system table (sec 8.4.1), so that the base table name matches the class name. These methods use .NET Reflection machinery to access public fields in the supplied object. If you add other public fields and properties to these classes, consider marking them with the [Exclude] attribute.

### 8.6.13 PyrrhoDbType

DbType in System.Data is used for DbParameters and is rather specific for SQL Server. Pyrrho's version of this is as follows:

Value	DomainAttribute provided?
DBNull	
Integer	
Decimal	
String	
Timestamp	
Blob	yes
Row	yes
Array	yes
Real	
Bool	
Interval	yes

Time	
Date	yes
UDType	yes
Multiset	yes
Document	

### 8.6.14 PyrrhoInterval

The methods and properties of PyrrhoInterval are:

Method or Property	Explanation
int years	The years part of the time interval
int months	The months part of the time interval
long ticks	The ticks part of the time interval
static long TicksPerSecond	Gets the constant number of ticks per second
static string ToString()	Formats the above data as e.g. (0yr,3mo,567493820000ti)

### 8.6.15 PyrrhoList

### 8.6.16 PyrrhoParameter

This class is Pyrrho's implementation of IDbDataParameter and IDataParameter. The only change introduced is that the native field type is publicly accessible. See PyrrhoDbType in 8.8.9 above.

### 8.6.17 PyrrhoParameterCollection

This is Pyrrho's implementation of DbParameterCollection.

### 8.6.18 PyrrhoReader

This class is Pyrrho's implementation of IDataReader. The only additional members of PyrrhoReader are DataSubtypeName, Description, GetEntity, Output, and Url:

Method or Property	Explanation
void Close()	
string DataSubtypeName(int i)	Returns the domain or type name of the actual type of the ith column in the current row. (Usually this will be the same as DataTypeName.)
string Description(int i)	Returns the description metadata of the ith column
void Dispose()	
bool GetBoolean(int i)	Gets the value of the specified column as a Boolean.
byte GetByte(int i)	Gets the 8-bit unsigned integer value of the specified column.
int GetBytes(int i,int fieldoff,byte[] buf,int off, int length)	Reads a stream of bytes from the specified column offset into the buffer as an array, starting at the given buffer offset.
char GetChar(int i)	Gets the 1-character value of the specified column.
int GetChars(int i,int field,byte[] buf,int off,int length)	Reads a stream of characters from the specified column offset into the buffer as an array, starting at the given buffer offset.
string GetDataTypeName(int i)	Gets the data type information for the specified field.
DateTime GetDateTime(int i)	Gets the date and time data value of the specified field.
decimal GetDecimal(int i)	Gets the fixed-position numeric value of the specified field
double GetDouble(int i)	Gets the double-precision floating point number of the specified field.
T GetEntity<T>()	Used in strongly-typed PyrrhoReaders (as in ExecuteTable<T>)
System.Type GetFieldType(int i)	Gets the Type information corresponding to the type of Object that would be returned from GetValue(Int32).
float GetFloat(int i)	Gets the single-precision floating point number of the specified field.
short GetInt16(int i)	Gets the 16-bit signed integer value of the specified field.
int GetInt32(int i)	Gets the 32-bit signed integer value of the specified field.
long GetInt64(int i)	Gets the 64-bit signed integer value of the specified field.
string GetName(int i)	Gets the name for the field to find. (the caption for the column)

int GetOrdinal(string n)	Return the index of the named field.
PyrrhoTable GetSchemaTable()	Returns a DataTable that describes the column metadata of the IDataReader.
string GetString(int i)	Gets the string value of the specified field.
object GetValue(int i)	Return the value of the specified field
void GetValues(object[])	Populates the array with the values in the current row
bool IsDBNull(int i)	Return whether the specified field is set to null.
string Output(int i)	Returns the output flag of the ith column
bool Read()	Advances to the next row: returns false if there are no more rows
string Url(int i)	Returns the web metadata url of the ith column

### 8.6.19 PyrrhoRow

PyrrhoRow is used only when required for values of structured types. The methods and properties of PyrrhoRow are:

Method or Property	Explanation
CellValue[] row	The values of the fields
int state	0=Original, 1=Current, 2=Proposed
PyrrhoTable schema	The table that specifies the row structure

### 8.6.20 PyrrhoTable

A PyrrhoTable is constructed internally by every invocation of ExecuteReader. As in ADO.NET DataTable there are properties called Rows and Columns, and an array of PrimaryKey columns.

### 8.6.21 PyrrhoTransaction

This class imitates IDbTransaction, but provides an extra method: CommitAndReport()

Method or Property	Explanation
int Commit (params Versioned[])	Commit the transaction and optionally fill in version information for a set of objects. Returns the number of records affected by deferred triggers.
bool Conflict	Gets whether a conflicting transaction has been committed since the start of this transaction. (Requires a round trip to the transaction master server.) If Conflict is true, a subsequent Commit will fail, but the transaction is not closed.
void Rollback()	Roll back the transaction

### 8.6.22 Versioned

Versioned is the base class for Pyrrho's entities as generated by Role\$Class, and supports the REST additions to PyrrhoConnect. Subclasses of Versioned model records in the database, and all fields are marked nullable in order to support the POST operation.

Field	Explanation
PyrrhoConnect conn	A copy of the PyrrhoConnect used to create this
string version	The value is the latest row version validator for the entity, which is a string returned by the server. Do not modify this field.
string entity	A validator to check that the query used to retrieve the data would still return the same results. Do not modify this field.

Method	Explanation
void Delete()	Advise conn that this object is to be deleted: invokes triggers as side effects.
void Get()	Overwrites this with the latest version from conn
void Put()	Advise conn that this object is to be updated: invokes triggers as side effects.



### 8.6.23 WebCtrlr

This class is from the AWebSvr library. Derived classes (e.g. XXController) should provide one of more of the standard HTTP methods GetXX, PutXX, PostXX, DeleteXX according to one or both of the following templates:

```
public static string VERBXX(WebSvc ws,Document d)
public static string VERBXX(WebSvc ws,params object data)
```

The value returned should be the response string for sending to the client.

Field	Explanation
virtual bool AllowAnonymous()	Can be overridden by a subclass. The default implementation returns false, but anonymous logins are always allowed if no login page is supplied (Pages/Login.htm or Pages/Login.html).

### 8.6.24 WebSvc

This class is from the AWebSvr library. Your custom web server/service instance(s) will indirectly be subclasses of this class, so will have access to its protected fields and methods documented here. Controllers should be added in a static method, e.g. in Main()

Derived classes typically organise a connection to the DBMS being used. The connection can be for the service or for the request, and so should be set up in an override of the Open method.

Field	Explanation
static void Add(WebCtrlr wc)	Install a controller for the service.
virtual bool Authenticated()	Override this to discriminate between users. By default the request will be allowed to proceed if AllowAnonymous is set on the controller or there is no login page. Get user identities etc from the context.
virtual void Close()	Can be overridden to release request-specific resources.
System.Net.HttpListenerContext context	Gives access to the current request details.
<i>dict</i> controllers	The controllers for the service. Make sure you add controller to this dictionary.
static System.Collections.Generic.Dictionary <string,WebCtrlr> controllers	The controllers defined for the service.
string GetData()	Extracts the HTTP data supplied with the request: a URL component beginning with { will be converted to a Document.
virtual void Log(string verb, System.Uri u, string postData)	Write a log entry for the current controller method. The default implementation appends this information to Log.txt together with the user identity and timestamp.
virtual void Open (System.Net.HttpListenerContext cx)	Can be overridden by a subclass, e.g. to choose a database connection for the current request. The default implementation does nothing.
<i>Serve()</i>	<i>Calls the requested method using the above templates. Don't call this method directly.</i>

### 8.6.25 WebSvr

This class is from the AWebSvr library. Your custom web server should be a subclass of WebSvr, and WebSvr is a subclass of WebSvc. It defines the URL prefixes (including hostnames and port numbers) for the service. If your service is multi-threaded, you can override the Factory method to returning a new instance of your WebSvc subclass. Finally, call either of the two Server methods to start the service loop.

Field	Explanation
virtual WebSvc Factory ()	Can be overridden by a subclass to create a new service instance. The default implementation returns this (for a single-threaded server).
void Server(params string[] prefixes)	Starts the server listening of a set of HTTP prefixes (up to the appName), with anonymous authentication.
void Server(System.Net.AuthenticationSchemes au, params string[] prefixes)	Starts the server listening of a set of HTTP prefixes (up to the appName), with the given authentication scheme(s).

## 8.7 The Pyrrho protocol

The "Pyrrho protocol" defines the binary traffic between the client and server. (Note that this is different from the "PyrrhoDb protocol" mentioned in section 6.13, which is actually implemented on the client side by class `PyrrhoWebRequest` in file `PyrrhoDbClient.cs`).

In the following discussion, ints are coded in 4 octets as signed 32-bit quantities, most significant octet first, and longs are 8 octets. A String is always coded in UTF8 invariant-culture Unicode, prefixed by an int giving the number of octets in the string data.

Localisation is handled by the client library.

### 8.7.1 Low level-communication

As soon as the TCP connection to the server is established, the server sends a long to the client. This is a nonce used for encrypting the connection string.

Client replies with octet 0x0 .

Since version 1.0, the low-level communication uses asynchronous buffering, with the help of the class `AsyncStream`. All communication between client and server uses 2048-octet buffers, which normally contain in the first two octets (octets 0 and 1) the count of valid bytes that follow in the buffer (i.e. this count is in range 0..2046.)

Since version 2.0, this mechanism has been modified to provide better support for exceptions reported by the server during transmission of data (e.g. during `PutRow()`). If the count appears to be 2047, the buffer contains an exception record instead, in which the next two octets (octets 2 and 3) the count of octets used to transmit the exception details. On the server side, this exception mechanism is supported by `AsyncStream.StartException()`. On the client side, there is a corresponding `AsyncStream.GetException()`.

The following protocol bytes are supported (enumeration `PyrrhoBase.Protocol`).

Protocol Name	Byte
Authority	13
BeginTransaction	6
Check	42
CheckConflict	32
ClientAnswer	81
CloseConnection	9
CloseReader	5
Commit	7
<i>CommitAndReport</i>	43
CommitAndReport1	77
<i>CommitAndReportTrace</i>	75
CommitAndReportTrace1	78
CommitTrace	74
Delete	48
<i>DetachDatabase</i>	15
Execute	55
ExecuteNonQuery	2
ExecuteNonQueryCrypt	39
ExecuteNonQueryTrace	73
ExecuteReader	21
ExecuteReaderCrypt	28
Get	33
Get1	47
Get2	56
GetFileNames	10
GetInfo	54
<i>Mark</i>	23
Prepare	11
Post	45

Put	46
ReaderData	16
ResetReader	14
Rollback	8
TypeInfo	19

The following response bytes are defined (enumeration PyrrhoBase.Responses)

Acknowledged	0
ReaderData	10
Done	11
Schema	13
CellData	14
NoData	15
Files	18
Prepare	55
Primary	60
Begin	62
TransactionReport	65
Warning	67
PostReport	68
Columns	71
Schema1	72
DoneTrace	76
TransactionReportTrace	77
Entity	79
AskClient	80

## 8.7.2 Sending the connection string

For .NET implementation, the user name is supplied by the operating system (not by the user). Not all fields in the connection string are sent to the server: provider: host and port are already used in establishing the connection to the server. For the reference for the connection string, see section 6.3.

All traffic in this section is encrypted including the protocol octets. Recall that the encryption algorithms in PyrrhoLink.dll and OSPLink.dll are different. Note that Locale is handled by these DLLs and not sent to the server.

Connecting Octet	Octet value	Further data	Description
<i>Base</i>	29		unused
<i>BaseServer</i>	31		unused
<i>Coordinator</i>	30		unused
Done	24		signals end of the connection string data
Files	22	String	a comma-separated list of databases*
<i>Host</i>	26		
Length	33		
Modify	32	true or false	Allow modification (default true for the first database in the connection)
<i>Password</i>	20		
<i>Port</i>			
<i>Provider</i>			
Role	23	String	the Role for the connection
Stop	25	String	the stop time
<i>User</i>	21		

\*The Files keyword is now a misnomer as only one database is permitted per connection.

On successful completion of this phase, non-encrypted communication resumes, and the server responds as follows:

Response Octet	Octet value	Further data	Description
----------------	-------------	--------------	-------------

Primary	60		<i>Preserved for compatibility</i>
---------	----	--	------------------------------------

### 8.7.3 Protocol details

This list is complete in the interests of backward compatibility. Italics indicates the protocol octet is no longer supported by the server.

Protocol octet	Request	Response
-1	EoF	
0		<i>Acknowledged</i>
1		<i>OobException</i>
2	ExecuteNonQuery	
3	<i>SkipRows</i>	
4	<i>GetRow</i>	
5	CloseReader	
6	BeginTransaction	
7	Commit	
8	Rollback	
9	CloseConnection	
10	GetFileNames	ReaderData
11	Prepare	Done
12	<i>Request</i>	Exception
13	Authority	Schema
14	ResetReader	CellData
15	<i>Detach</i>	NoData
16	ReaderData	FatalError
17	<i>Fetch</i>	TransactionConflict
18	<i>DataWrite</i>	Files
19	TypeInfo	
20		<i>RePartition</i>
21	ExecuteReader	
22	<i>RemoteBegin</i>	
23	<i>Mark</i>	
24	<i>DbGet</i>	
25	<i>DbSet</i>	
26	<i>Physical</i>	
27	<i>GetMaster</i>	
28	<i>ExecuteReaderCrypt</i>	
29	<i>DirectServers</i>	
30	<i>RePartition</i>	<i>RePartition</i>
31	<i>RemoteCommit</i>	
32	<i>CheckConflict</i>	
33	Get	
34	<i>CheckSerialisation</i>	
35	<i>IndexLookup</i>	
36	<i>CheckSchema</i>	
37	<i>GetTable</i>	
38	<i>IndexNext</i>	
39	<i>ExecuteNonQueryCrypt</i>	
40	<i>TableNext</i>	
41	<i>Mongo</i>	
42	<i>Check</i>	<i>Fetching</i>
43	CommitAndReport	<i>Written</i>
44	<i>RemoteCommitAndReport</i>	
45	Post	<i>Master</i>
46	Put	<i>NoMaster</i>
47	Getl	<i>Servers</i>
48	Delete	<i>IndexCursor</i>
49	<i>Update</i>	<i>LastSchema</i>

50	Rest	<i>TableCursor</i>
51	<i>Subscribe</i>	<i>IndexData</i>
52	<i>Synchronise</i>	<i>IndexDone</i>
53	<i>SetMaster</i>	<i>TableData</i>
54	GetInfo	<i>TableDone</i>
55	Execute	<i>Prepare</i>
56	Get2	<i>Request</i>
57		<i>Committed</i>
58		<i>Serialisable</i>
59	ExecuteMatch	<i>MatchDone</i>
60		<i>Primary</i>
61		<i>Secondary</i>
62		<i>Begin</i>
63		<i>Valid</i>
64		<i>Invalid</i>
65		<i>TransactionReport</i>
66		<i>RemoteTransactionReport</i>
67		<i>PostReport</i>
68		<i>Warning</i>
69		<i>TransactionReason</i>
70		<i>DataLength</i>
71		<i>Columns</i>
72		<i>Schema1</i>
73	ExecuteNonQueryTrace	
74	CommitTrace	
75	CommitAndReportTrace	
76	ExecuteTrace	<i>DoneTrace</i>
77	CommitAndReport1	<i>TransactionReportTrace</i>
78	CommitAndReportTrace1	
79		<i>Entity</i>
80		<i>AskClient</i>
81	ClientAnswer	
82	Continue	
83	GraphInfo	<i>GraphInfo</i>

Normal traffic consists of client requests and server replies, using formats described in the following subsections ( braces { } indicate repetition prefixed by an int count ):

Protocol Octet	Further data	Description	Response Octet†	Further data	Description
Authority	String	Session role name	Done		
BeginTransaction					
<i>Check</i>	<i>String</i>		<i>Valid</i>		
			<i>Invalid</i>		
<i>CheckConflict</i>				<i>int</i>	<i>1 if transaction conflict has occurred</i>
CloseConnection					
CloseReader					
Commit			Done		
CommitAndReport	{check}	ignored	Transaction Report	{check}	Updates the check information
CommitAndReport1	{check}	ignored	Transaction Report	int, {check}	No of records affected, check info
CommitAndReportTrace	{check}	ignored	Transaction ReportTrace	long,long, {check}	oldloadpos,loadpos, check info
CommitAndReportTrace1	{check}	ignored	Transaction ReportTrace	int, long,long, {check}	no of records affected, oldloadpos,loadpos, check info

CommitTrace			DoneTrace	long,long	oldloadpos,loadpos
Delete	int, string	Schema key, entity	Done		
<i>DetachDatabase</i>	<i>String</i>	<i>Database name</i>	<i>Done</i>		
Execute	String, {String}	Prepared statement name, actual params	Done	int	number of records affected
ExecuteMatch	String	SQL Match statement	MatchDone	int	number of records affected, or
			TableData Schema	schema	or otherwise
ExecuteNonQuery	String	SQL statement	Done	int	number of records affected
ExecuteNonQuery Trace	String	SQL statement	DoneTrace	long,long, int	oldloadpos,loadpos, number of records affected
ExecuteReader	String		Schema	schema	
			Done		if not a select statement
<i>ExecuteReaderCrypt</i>	<i>String</i>	<i>Encrypted SQL</i>	<i>Schema</i>	<i>schema</i>	
			<i>Done</i>		<i>if not a select statement</i>
ExecuteTrace	String, {String}	Prepared statement name, actual params	DoneTrace	long,long, int	oldloadpos,loadpos, number of records affected
Get	String	rurl	Schema	schema	
			Done		no data
Get1	long, String	Schema key, rurl	Schema	schema	
			Done		no data
Get2	String	rurl	Schema1	long, schema	Schema key, schema
			Done		No data
GetFileNames			Files	{string}	Names of databases in folder
GetInfo	String	typeName	Columns	{column}	Details for a structured type
GraphInfo			GraphInfo	{String, int}""	See Role\$GraphInfo
<i>Mark</i>					<i>Allows error recovery (uses TRANSACTION_ACTIVE)</i>
Prepare	2 Strings	name, SQL parametrised statement	Done		
Post	int, sql	Schema key, insert stmt	Entity	{column, cell} string	changed columns if any followed by entity/ version
Put	int, string, row	Schema key, entity, values data			
ResetReader			Done		
Rollback			Done		
ReaderData		Verb, url, Jsondata	ReaderData	{cell}	cells*
Rest	String, url, String	Data type name	Done		
TypeInfo	String			string	

† Octet 68 (Warning) may precede any reply, followed by string, {string} for signal and parameters.

§ In explicit transactions instead of Commit use CommitAndReport to update entity information.

\* A single large cell may take more than one physical block. Otherwise, the ReaderData call returns the number of cells that will fit into a physical block, which may include data from subsequent rows if any.

## 8.7.4 Schema

The Schema reply consists of 0xb if the table is empty. Otherwise, it consists of 0xd, followed by the number of columns, the name of the table, and then for each column, the caption and type data as described below (sec. 8.7.5).

## 8.7.5 Column

A Columns reply consists of the number of columns, followed by the caption for the column and a type. The caption is a String. The type information consists of a type name followed by an int constructed as follows:

Mask	Description
0x00f	Base Data Type (see below)
0x0f0	0 if not a primary key column, otherwise primary key ordinal+1
0x100	Not Null
0x200	Generated Always
0x400	Reverse order (internal)

## 8.7.6 Cell

The number of columns was provided beforehand, so a row consists of CellData for each of the columns.

CellData may be optionally preceded by octet 3 and a row version validator string and/or octet 4 and a readCheck string. Then octet 0 if the column contains null, octet 1 followed by the cell value if the value type matches the column's typecode (followed by the value), octet 2 otherwise (followed by subtype name and value).

Typecode	Data Type	Value format
0	null	0 for null
1	Integer	String
2	Numeric	String
3	String	String
4	Timestamp	a long: ticks
5	Blob	{ Octet }
6	NestedRow	{ Field }
7	Array	ARRAY { Cell Cell }
8	Real	String
9	Boolean	int
10	Interval	3 longs: years, months, ticks
11	Time	long: ticks
12	Field	String, Cell
13	Date	long: ticks
14	Table, Type, NodeType, EdgeType	Schema { Cell }
15	List, Set, Multiset	(LIST SET MULTISET) { Cell }

## 8.7.7 Type

Type information is given as an XML string.

## 8.7.8 Exceptions

These are exception replies during the normal traffic sequence. Since version 2.0, these are reported in a special exception block, as follows. If the count appears to be 2047, the buffer contains an exception record instead, in which the next two octets (octets 2 and 3) contain the count of octets used to transmit the exception details.

Server Octet	Further data	Description
0xc	String, Strings, StringPairs*	Database Exception
0x11	String	Transaction Conflict
0x10	String	Other exception

\* added in version 4.8 for diagnostics information.

### 8.7.9 JsonData

Structured data is returned in JSON format in the API and by the Pyrrho's HTTP service. For the RESTView implementation, an additional field is added to the Json document returned by the HTTP service for every aggregation function in each row, containing the Register contents accumulated during computation of that function for that row. These extra fields enable the aggregation of such results from a number of remote servers by the REST USING feature described in this document. It is hoped that other DBMS will support this extension.

The extra fields have names of form \$#nnn (a dollar sign, a number sign, and a deminal integer string). The number nnn distinguishes the aggregation function in the SQL request that generated this number: these are assigned in ascending order from left to right, and should be the same for each row of the data returned.

The value of the extra field is a Json document depends on the kind of aggregation function containing a sequence of fields with decimal integer names 0, 1, etc and optional values in the following order:

- The value of COUNT
- The string value accumulated by the function if any
- One of the following:
  - The value of MAX, MIN, FIRST, LAST, ARRAY
  - A document containing numbered fields for a multiset value (e.g. INTERSECT)
  - The value of a typed SUM (used in several functions e.g. AVG)
- A sum of squares (e.g. STDDEV\_POP)

The Register class contains other fields that are used for window functions; but window functions are not aggregation functions and so these fields are not used for JsonData.



## 9. Pyrrho Database File Format

The Pyrrho database file begins with a key (777) and version number (e.g. 50) encoded using Pyrrho's integer format 9.1.1. The rest of the file consists of a sequence of variable length records, whose type is given by the opening byte, and whose contents are variable length. Each record is made up of a set of data fields: some have fixed format, and some have variable format. The record committed by a transaction are placed together, prefaced by a PTransaction record that declares the user and role for the commit and the number of following records in the commit; and all of the record in the commit contain a reference to this PTransaction record.

Once any data has been written to the file it stays unchanged at the position it was written (append storage). Database files larger than 32GB are physically divided into 32GB segments. The data is continued logically from one file to the next without any additional formatting.

### 9.1 Data Formats

Byte and Unicode are the only predefined formats. It is assumed that all data files are dealt with by the operating system as a sequence of bytes. In particular, Pyrrho has its own way of encoding integers, floats etc, which are described below.

Pyrrho constructs a small set of data types from these, as follows:

Code	Data Type	Format as
0	Null	
1	Time	1 Integer (UTC ticks)
2	Interval	3 Integers (year,month, ticks)
3	Integer	1 byte (bytelength), bytelength bytes: see 9.1.1
4	Numeric	2 Integers (mantissa, scale: see 9.1.2)
5	String	1 Integer (bytelength), bytelength UTF-8 bytes
6	Date	1 Integer (UTC ticks)
7	TimeStamp	1 Integer (UTC ticks)
8	Boolean	1 byte: T=1,F=0
9	DomainRef	Structured: 2 Integers (typedefpos,els), els variants: see 9.1.3 Otherwise: 1 Integer (domaindefpos)
10	Blob	1 Integer (bytelength), bytelength bytes
11	Row	2 Integers (typedefpos,cols), cols pairs(coldefpos,variant: see 9.1.3)
12	Multiset	2 Integers (typedefpos,els), els (variant,count)s: see 9.1.3, 9.1.4
13	Array	2 Integers (typedefpos,els), els (int,variant)s: see 9.1.3, 9.1.4
14	Vector	2 Integers (typedefpos,els), els (int,variant)s: see 9.1.3, 9.1.4
15	List	2 Integers (typedefpos,els), els variants: see 9.1.3

#### 9.1.1 Integer format

Zero is encoded as 0 bytes. An integer that fits in a signed byte is encoded as 1 byte (i.e. -127.. 127). Otherwise integers are encoded in unsigned bytes (radix 256), using as many as are required to ensure the first byte has a sign bit (0x80) if and only if the integer is negative.

Unless otherwise specified, unbounded precision is used for integer arithmetic. A string representation is used if required to return a very large integer value to the client.

#### 9.1.2 Numeric and Real format

Numeric format has one Integer for the mantissa, and 1 for the scale. If these are m and s respectively, then the value of the decimal is  $m \cdot 10^{-s}$ . This format is used for both numeric/decimal and real quantities.

Unless constrained by precision specifications, addition and multiplication of numeric quantities uses 2040-bit precision, while division uses a default precision of 13 decimal digits. If greater precision is required for division, it can be specified. It should be obvious that there are resource implications to using very large precision values.

### 9.1.3 Variant format

This consists of

- a 1-byte code for the data type (the code in the above table 9.1),
- if this byte is 9 (DomainRef), the defining position of the type
- data in the corresponding format.

### 9.1.4 Array, Vector and Multiset format

Two Integers (9.1.1), namely the defining position of the element type, the number of elements  $n$ , followed by  $n$  pairs: for Array and Vector these are (long,variant), for Multiset (variant,count).

### 9.1.5 Row and User Defined Type format

Two Integers (9.1.1), namely the defining position of the row type, the number of non-null fields  $n$ , then for each, an Integer (9.1.1) for the defining position of the field (a column), and an element of that type.

### 9.1.6 Blob format

An Integer (9.1.1), namely the number of bytes  $n$ , followed by  $n$  bytes.

### 9.1.7 Boolean format

1 byte (1 for true, 0 for false).

### 9.1.8 String (Char) format

An Integer (9.1.1), namely the number of bytes  $n$  of actual data, followed by  $n$  bytes in UTF8 encoding. (The fieldsize is not used).

### 9.1.9 Date and TimeSpan formats

An Integer (9.1.1) namely the number of ticks in the date or timespan.

### 9.1.10 Interval format

Three Integers (9.1.1), namely years, months, and ticks.

## 9.2 Record formats

The record formats are as follows (note that many are now deprecated for all new transaction data as indicated below):

Code	Record type	Format as 1 byte for Code and then
	Physical	1 integer (transaction id)
0	EndOfFile	4 bytes (validation). Not used with append storage.
1	Table	1 string (name), Physical
2	Role	2 strings (name, details), Physical
3	Column	1 integer (table id), 1 string (name), 2 integer (position, domain id), Physical. <i>Deprecated – see Column3</i>
4	Record (Insert)	1 integer (table id), Fields (see 9.2.2), Physical
5	Update	2 integers (replaced record id, other fields: see 9.2.3), Record
6	Change	1 integer (object id), Table (no longer used)
7	Alter	1 integer (prev), Column. <i>Deprecated – see Alter3</i>
8	Drop	1 integer (object id), Physical
9	Checkpoint	(no data), Physical
10	Delete	1 integer (record id), Physical <i>Note: deprecated: use Delete1 instead</i>
11	Edit	1 integer (replaced domain id), Domain
12	Index	1 string (name), 2 integers (table id, ncols), ncols integers ( $\pm$ column id), 2 integers (flags, reference, see 9.2.5), Physical. Negative column id indicates reverse ordering
13	Modify	1 integer (replaced id), 2 strings (name, body), Physical

14	Domain	1 string (name), 3 integers (dataType: see 9.2.1, dataLength, scale), 3 strings (charset, collate, default), 1 integer (element domain or table id), Physical
15	Check	1 integer (object id), 2 string (name, check source), Physical
16	Procedure	1 string, 1 integer, 1 string (name, arity, proc source), Physical - <i>deprecated: see Procedure2</i>
17	Trigger	1 string (name), 3 integers (table id, triggertype, position, see 9.2.8), 1 string (definition), Physical
18	View	2 strings (name, view source), Physical
19	User	1 string (name), Physical
20	Transaction	4 integers (nrecs, role id, user id, time)
21	Grant	3 integers (privilege, see 9.2.7, object id, grantee id), Physical
22	Revoke	Grant
23	Role1	1 string (name), Physical. <i>Deprecated – use Role instead</i>
24	Column2	1 string (default), 1 boolean (notNull), 1 GenerationRule, Column <i>Deprecated</i>
25	Type	1 integer (under type id) <sup>73</sup> , Table
26	Method	2 integers (type id, methodtype: see 9.2.6), Procedure - <i>deprecated: see Method2</i>
27	<i>Not Used</i>	
28	Ordering	3 integers (type def, func def, flags: see 9.2.9), Physical
29	<i>NotUsed</i>	
30	DateType	2 integers (start field, end field, see 9.2.10). Domain (dataLength and scale are for seconds precision), Physical
31	<i>Not Used</i>	
32	<i>Not Used</i>	
33	<i>Not Used</i>	
34	<i>Type1</i>	<i>1 string (with uri), Type</i>
35	Procedure2	1 string, 2 integers, 1 string (name, arity <sup>74</sup> , ret type id, proc source), Physical
36	Method2	2 integers (type id, methodtype: see 9.2.6), Procedure2
37	Index1	1 integer (adapter), Index
38	Reference	2 integers (index defpos, referrer pos), Fields (see 9.2.2), Physical: only used when a coercion or adapter function creates a reference. The Fields give the computed foreign key.
39	Record2	1 integer (subtype), Record
40	Curated	Physical. Prevents further change, subsequent log entries are PUBLIC
41	<i>NotUsed</i>	
42	<i>Domain1</i>	<i>2 strings(typeiri,abbrev),Domain</i>
43	<i>Namespace</i>	<i>2 strings(prefix,iri)</i>
44	<i>Table1</i>	<i>1 string(rowiri), Table</i>
45	<i>Alter2</i>	<i>1 long (prev), Column2</i>
46	<i>AlterRowIri</i>	<i>1 long (prev), Table1</i>
47	Column3	1 string, 3 ints (flags, reindex, toType), Column2 For flags and toType see 9.2.14
48	Alter3	1 long (prev), Column3
49	Type2	extra supertypes see 9.2.15, Type ( <i>from Feb 2024</i> )
50	Metadata	3 strings (name, details, iri), 3 ints (seq, objid, flags – see 9.2.11), Physical. Seq is nonzero only for view and function columns.
51	PeriodDef	1 integer (table), 1 string (periodname), 2 integers (start, end)
52	Versioning	1 integer (period) only for system versioning
53	Check2	1 integer (columndefpos), Check
54	<i>Not Used</i>	

<sup>73</sup> See also Type2.<sup>74</sup> From file format 52, arity is no longer used and is given as 0. Procedure and method names are no longer modified by adding \$arity. The source field supplies the signature.

55	<i>Not Used</i>	
56	ColumnPath	1 integer (column defpos), 1 string (the path, starting with .), 1 integer (the domain definition).
57	Metadata2	1 int (maxDocuments) 1 long (storageSize), Metadata not used
58	Index2	1 integer (metadata), Index1 not used
59	DeleteReference1	Reference1
60	Authenticate	1 string (password), 1 int (defrole), User deprecated
61	RestView	1 integer (struct), View. The URL is provided in metadata as the desc field.
62	TriggeredAction	1 integer (trigger defpos) introducing an embedded set of changes
63	RestView1	Name,password,RestView deprecated: provide any credentials in URL
64	Metadata3	refpos, Metadata2
65	RestView2	usingtable, RestView
66	Audit	3 integers (user, table, ticks) {integer} {string} (cols,keys), Physical
67	Clearance	1 integers (user), Label (clearance, see 9.2.13), Physical
68	Classify	2 integers (object), Label (classification, see 9.2.13), Physical
69	Enforcement	2 integers (table, flags see 9.2.7 Privilege below), Physical
70	Record3	Level (classification, see 9.2.13), Record2
71	Update1	Level (classification, see 9.2.13), Update
72	Delete1	1 integer (table), Delete
73	Drop1	1 integer (RemoveAction), Drop
74	RefAction	2 integers (defpos, flags) Physical
75	Post	Not serialised. For building transacted REST requests
76	NodeType	Type2
77	EdgeType	2 integers (leaving, arriving), NodeType
78	EditType	1 integer (unused), Type
79	AlterIndex	1 integer (index), Physical
80	AlterEdgeType	3 integers (qlx, reftype, edgetype), Physical
81	Record4	extra tables see 9.2.15, Record3
82	Update2	extra tables see 9.2.15, Update1
83	Delete2	extra tables see 9.2.15, Delete1
84	PSchema	1 string (directoryPath), Physical
85	PGraph	1 string, {integer} {integer} (iri,node and edge types, records), Physical
86	PGraphType	1 string {integer} {iri. node and edge types}, Physical*

### 9.2.1 DataType

Code	DataType
11	ARRAY
27	BOOLEAN
37	CHAR
40	CLOB
65	CURSOR
67	DATE
135	INTEGER
136	INT
137	INTERVAL0
152	INTERVAL
168	MULTISET
171	NCHAR
172	NCLOB
177	NULL
179	NUMERIC
199	REAL0
203	REAL
218	PASSWORD
255	SET

257	TIME
258	TIMESTAMP
267	TYPE
297	TABLE
461	EDGETYPE
534	NODETYPE

These codes are used in the PColumn and PDomain records.

### 9.2.2 Drop Action

Code	Drop Action
3	Cascade
2	Default
1	Null
0	Restrict (default)

### 9.2.3 Fields information

The sequence of fields defining a record is formatted as 1 integer (nfields), nfields x (1 integer (column id), 1 variant (value)) see 9.1.3. Fields not defined by a record are not supplied.

### 9.2.4 Update information

The Update record contains in the base class (Record) part the fields that are updated. The other fields integer identifies the most recent previous Record or Update record with field information that remains current. The replaced record id is the original record that subsequent updates have altered.

### 9.2.5 Index flags

The reference field is the id of a reference object (a table or type).

Flag	Meaning
0	NoType
1	Primary Key
2	Foreign Key
4	Unique
8	Descending (all key columns) <i>Deprecated</i>
16	Restrict Update
32	Cascade Update
64	Set Default Update
128	Set Null Update <i>Deprecated</i>
256	Restrict Delete
512	Cascade Delete
1024	Set Default Delete
2048	Set Null Delete
4096	TemporalKey <i>Deprecated</i>

Not all flags are permitted or required: Restrict is a default, and Set Null is not permitted.

### 9.2.6 Method type

Value	Meaning
0	Instance
1	Overriding
2	Static
3	Constructor

### 9.2.7 Privilege flags

Flag	Meaning	Flag	Meaning
0x1	Select	0x400	Grant Option for Select
0x2	Insert	0x800	Grant Option for Insert

0x4	Delete	0x1000	Grant Option for Delete
0x8	Update	0x2000	Grant Option for Update
0x10	References	0x4000	Grant Option for References
0x20	Execute	0x8000	Grant Option for Execute
0x40	Owner	0x10000	Grant Option for Owner
0x80	Role	0x20000	Admin Option for Role
0x100	Usage	0x40000	Grant Option for Usage
0x200	Handler	0x80000	Grant Option for Handler

### 9.2.8 Trigger type

Flag	Meaning
1	Insert
2	Update
4	Delete
8	Before
16	After
32	Each row
64	Instead
128	Each statement
256	Deferred

### 9.2.9 Ordering type

Flag	Meaning
0	None
1	Equals
2	Full
4	Relative
8	Map
16	State

### 9.2.10 Interval fields

Flag	Meaning
0	SECOND
1	MINUTE
2	HOUR
3	DAY
4	MONTH
5	YEAR

### 9.2.11 Metadata flags

Flag	Meaning
0x0	Unspecified
0x1	ENTITY
0x2	ATTRIBUTE
0x4	PIE
0x8	<i>SERIES</i>
0x10	POINTS
0x20	X
0x40	Y
0x80	HISTOGRAM
0x100	LINE
0x200	CAPTION
0x400	<i>CAPPED</i>
0x800	<i>USEPOWEROF2SIZES</i>
0x1000	<i>BACKGROUND</i>

0x2000	<i>DROPDUPS</i>
0x4000	LEGEND
0x8000	URL
0x10000	MIME
0x20000	SQLAGENT
0x40000	USER
0x80000	PASSWORD
0x100000	IRI
0x200000	ETAG
0x400000	MILLI

In HTML output from a table, a chart is generated if the table is a pie, series, or points, one column has x and at least one column has y, histogram or line. Some of the deprecated entries here were for MongoDB. Url, mime, sqlagent, etag. Milli and user are for RESTViews.

### 9.2.12 GenerationRule

Flag	Meaning
0	No
1	Generated AS expression
2	Generated AS ROW START
3	Generated as ROW NEXT
4	Generated AS ROW END

### 9.2.13 Mandatory Access Control Label

There are two formats depending on whether the label is in the cache. The record begins with an Integer flag, and determines the format of what follows.

Flag	Rest of Record
0	1 Integer (defining position of the Label in the transaction log)
1	2 Integers (minLevel, maxLevel) {id} (groups) {id} (references)

### 9.2.14 Graph Flags

Special columns for NodeType and EdgeType are always of type CHAR, and their values are unique in the database.

Flag	Mnemonic	Default Name	Meaning
0	None		Not a special graph column
1	IdCol	ID	Node/Edge identity column
2	LeaveCol	LEAVING	Edge Leaving column
4	ArriveCol	ARRIVING	Edge Arriving column
8	SetValue		Value is a set of node identifiers

### 9.2.15 Graph Supertypes and Targets

Graph types are allowed to have more than one supertype. If PType is desired and there is more than one supertype, PType2 should be used for the supertypes other than the first. Serialization uses one int for the number n of further supertypes (i.e. n = total number of supertypes -1) and n longs.

Similarly, Records for node and edge types are allowed to have more than one target table. If a node or edge is created (resp updated, deleted) with more than one label, Record4 (resp Update2, Delete2) is used in place of Record (resp Update1, Delete1) or Record3 and serialises one int for the number n of extra tables and n longs.

## 10. Troubleshooting

This section reviews a number of circumstances in which a database can become unusable. The safeguards that cause a database to be marked unusable are there to protect business operations as far as practicable against hardware errors or malicious activity.

Databases should not become unusable during normal operation. Any performance issue of this sort should be notified immediately to malcolm@pyrrhodb.com, so that this issue can be resolved.

Suggested additions to this section will be very welcome. The following checklist is intended for use where a correctly installed Pyrrho installation ceases to work.

Symptom	Possible causes	Section
Application crashes or malfunctions	The PyrrhoLink.dll it uses needs to be updated to match the PyrrhoSvr	10.7
A database will not load	The database file may have been removed, renamed, or damaged	10.1-3
An application reports an invalid schema key	A user has updated the database schema and the Role\$Class, Role\$Java or Role\$Python system table should be used to regenerate the database class.	10.5
A user can no longer access or modify data	The user may be accessing the data from another user's account, or from an environment that reports the user name differently	10.4
	The user's (or role's) permissions have been modified	5.5

### 10.1 Destruction and restoration

It is fundamental to database design that transactions are durable once committed, with results that can only be changed by subsequent transactions. There are some interventions at the operating system level that violate this principle, which are possible even with Pyrrho.

- Destruction of the entire database through deletion of the database file, formatting or disposing of the storage media etc.
- Restoration of a database from a backup copy

These actions will result in some or all work recorded in the database to be lost. Restoration from backup can restore transactions up to the time of the backup, but transactions committed after the last backup will be permanently lost.

There are other interventions that can make the database temporarily inaccessible: such as stopping the server, or altering access permissions on the file or the network. These are not regarded as changing the durability of the transaction. The notes in this section assume that such matters can be resolved in the usual ways, such as restoring the accessibility of the database file, restoring network connectivity, etc.

Some hardware failures can cause a single transaction being committed at the time of the failure to be lost (section 10.2).

### 10.2 Hardware failure during commit

If a hardware failure occurs during the commit phase of a transaction, the client or application will be told that the connection has been broken but may not know whether the transaction commit was completed before communication with the server was broken.

When the database is reloaded, it is very likely that either (a) the transaction will have been forgotten (rolled back) or (b) the transaction will be found in its entirety. If a part of the transaction data was actually written to physical media, then recovery is required.

### 10.3 Alternative names for a database file

The database name can be the pathname of the file. Databases can be renamed in this version of Pyrrho, provided the connection strings in all applications are modified to reflect the change.



## ***10.4 User identity and database migration***

It is deliberately made difficult in Pyrrho for a user to pretend to be someone else: the user's name is supplied by the operating system. If a database file is installed in a new context, or a user's identity is changed, it may be difficult for an application to have the correct user identity for contacting the database.

Unless the database has withdrawn privileges from the system role, the server account can be used to access the database.

If any user identity in the database is still available, and has suitable admin privileges, it can be used to grant permissions to the new user identities.

Otherwise, use investigation of the log files to find out the user identities configured in the database, and temporarily install a user identity that is recognised by the database (preferably that of the database owner) and grant the permissions that the new user identities require.

## ***10.5 API Dependency on database history***

Section 6.4 discussed the API for object-oriented access to the database. It is important to remember that the class definitions (for C#, Java, or Python) used by this API must match the database schema. Each class and structured type has a schema key and this must match the position in the database file of the last schema change affecting the class or type.

Following such a change (or reconstruction of the database by another user) the affected schema keys must be updated in the application program.

## 11. End User License Agreement

You may use and redistribute the client libraries (PyrrhoLink.dll and/or PyrrhoJC.jar) in any product. You may copy and distribute this booklet in its entirety.

You are hereby granted a non-transferable, royalty-free license to use the software described in this manual in accordance with its provisions, and to view and test the source code, including modifications or incorporation in other software. Under no circumstances will Malcolm Crowe or the University of the West of Scotland be liable for any loss or damage however caused.

This software is and remains intellectual property of the University of the West of Scotland, protected by copyright. You are permitted to redistribute and include any of the code in any product, provided its ownership and copyright status is suitably acknowledged.

## References

- Crowe, M. K. (2007): An introduction to the source code of the Pyrrho DBMS. *Computing and Information Systems Technical Reports*, **40**, University of Paisley. 2<sup>nd</sup> ed. available on [github.com/MalcolmCrowe/ShareableDataStructures/tree/master/PyrrhoV7alpha](https://github.com/MalcolmCrowe/ShareableDataStructures/tree/master/PyrrhoV7alpha)
- Crowe, M., Begg, C., Laux, F., Laiho, M (2017): Data Validation for Big Live Data, *DBKDA 2017, The Ninth International Conference on Advances in Databases, Knowledge and Data Application*, Barcelona, Spain, May 21-26 2017. ISBN 978-1-61208-558-6, p. 30-36.
- Fielding, R.T. (2000): Architectural Styles and the Design of Network-Based Software Architectures, PhD Thesis, University of California, Irvine.
- Fielding, R. T , Reschke, J (eds) (2014): RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, IETF.org
- Floridi, L.: Sextus Empiricus: The transmission and recovery of Pyrrhonism (American Philological Association, 2002) ISBN 0195146719
- Francis, N. et al (2023): A Researcher's Digest of GQL, <https://hal.science/hal-04094449v1/document>
- GQL: ISO/DIS 39075 Database Languages: GQL (International Standards Organisation, 2024).
- Laertius, Diogenes (3<sup>rd</sup> cent): The Lives and Opinions of Eminent Philosophers, trans. C. D. Yonge (London 1895)
- Laiho, M., Laux, F. (2010): Implementing Optimistic Concurrency Control for Persistence Middleware Using Row Version Verification, *Advances in Databases Knowledge and Data Applications (DBKDA)*, 2010 Second International Conference on, IEEE, ISBN 978-1-4244-6081-6 p. 45-50, DOI: 10.1109/DBKDA.2010.25.
- Miller, C., McFadyen, R. (n.d.): Relational Databases and Microsoft Access, PressBooks.
- SQL2023: ISO/IEC 9075-2:2023 Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation); ISO/IEC 9075-4:2016: Information Technology – Database Languages – SQL – Part 4: Persistent Stored Modules; (International Standards Organisation, 2023)
- SWI-Prolog: [www.swi-prolog.org](http://www.swi-prolog.org)

## Index to Syntax

AbsoluteValue .....	74	Collect.....	75
Action .....	66	CollectionType.....	67
Adapter function.....	33	ColRef.....	64
Alias .....	69	Cols .....	64
Alter.....	60	COLUMN_NAME .....	58
AlterBody .....	60	Column2.....	124
AlterCheck.....	61	ColumnConstraint.....	63
AlterColumn.....	61	ColumnConstraintDef.....	63
AlterDomain.....	60	ColumnDefinition .....	63
AlterField.....	61	ColumnOption .....	64
AlterOp.....	60	ColumnOptionsPart.....	64
AlterTable.....	61	ColumnRef.....	70
AlterType.....	61	COMMAND_FUNCTION .....	58
AlterView .....	62	COMMAND_FUNCTION_CODE .....	58
Any .....	73	COMMIT_COMMAND.....	58
ASC .....	69	Comparison.....	73
Assignment.....	57	CompOp.....	73
Authority .....	46	CompoundStatement.....	57
Avg .....	74	CondInfo .....	58
BEGIN.....	56, 57	Condition .....	58
Between .....	73	CONDITION_NUMBER .....	58
BETWEEN .....	72	ConditionCode.....	58
BinaryOp .....	70	ConditionList .....	58
BindingVar .....	58	CONNECTION_NAME.....	58
BindType.....	58	Connections .....	65
BLOB .....	67	Connector.....	65
BooleanExpr.....	72	Connectors.....	65
BooleanFactor .....	72	CONSTRAINT .....	61
BooleanTerm.....	72	CONSTRAINT_CATALOG .....	58
BooleanTest.....	73	CONSTRAINT_NAME .....	58
BREAK.....	56	CONSTRAINT_SCHEMA.....	58
Bson.....	67	CONSTRUCTOR .....	60
Call .....	57	Contains .....	73
CAPTION.....	62	CONTENT .....	73
Cardinality .....	75	Count.....	74
CARDINALITY.....	62	Create .....	62
CASCADE .....	64	CREATE_GRAPH_STATEMENT.....	58
Cascade Delete .....	97, 126	CREATE_GRAPH_TYPE_STATEMENT ..	58
Cascade Update .....	96, 97, 126	CREATE_SCHEMA_STATEMENT.....	58
CaseStatement .....	57	CROSS.....	69
Cast.....	75	CSV.....	62
CATALOG_NAME .....	58	CURRENT .....	75
Ceiling .....	74	CURSOR_NAME.....	58
CHAR_LENGTH.....	74	DatabaseError .....	107
CharacterType .....	67	DataReader.....	49
CHECK .....	34, 66, 70	DataTypeList .....	66
CheckConstraint .....	61	Date.....	108
CLASS_ORIGIN.....	58	DateTimeField .....	71
classification.....	68	DateTimeFunction .....	75
Classification .....	61	DateTimeType .....	67
clearance.....	68	DBNull.....	49
Clearance .....	66, 93	Declaration.....	57
CLOB .....	67	Default .....	60
Close.....	57	DEFAULT .....	60
Coalesce.....	74	DEFERRED.....	64
Collate .....	67	DefinedType .....	67

Delete.....	69	GetName.....	49
DELETE_STATEMENT.....	58	Grant.....	65
DESC.....	69	Grantee.....	66
Direction.....	65	GranteeList.....	66
DocArray.....	70	Graph.....	68
DOCARRAY.....	66	GraphDetails.....	65
Document.....	70	GraphFunctions.....	75
DOCUMENT.....	66, 73	GraphItem.....	68
DocValue.....	70	GraphLabel.....	68
DomainDefinition.....	63	GraphLabels.....	75
DROP_GRAPH_STATEMENT.....	58	GraphType.....	75
DROP_GRAPH_TYPE_STATEMENT.....	58	GraphTypeDetails.....	65
DROP_SCHEMA_STATEMENT.....	58	GraphVar.....	58
DropObject.....	64	GroupByClause.....	72
DropStatement.....	64	Grouping.....	74
DYNAMIC_FUNCTION.....	58	GroupingSet.....	72
DYNAMIC_FUNCTION_CODE.....	58	GroupingSpec.....	72
Edge.....	65, 68	HandlerType.....	58
EdgePattern.....	65	HavingClause.....	72
EDGETYPE.....	62	HISTOGRAM.....	62
EdgeTypeDetails.....	65	Host.....	45
Element.....	75	How.....	58
ElementList.....	65	HttpFunction.....	74
EndField.....	71	IDataReader.....	49
EndPoints.....	65	IfStatement.....	59
EndTimestamp.....	107	In.....	73
EndTransaction.....	107	INNER.....	69
Enforcement.....	63	Insert.....	68
ETAG.....	62, 128	INSERT_STATEMENT.....	58
Event.....	64	INSTANCE.....	60
Every.....	73	IntegerType.....	67
Exclusion.....	72	Intersect.....	75
Execute.....	110	INTERSECT.....	71
Exists.....	73	INTERVAL.....	71
Exponential.....	74	IntervalField.....	67, 71
Extract.....	74	IntervalQualifier.....	71
ExtractField.....	74	IntervalType.....	67
Fetch.....	58	INVERTS.....	62
FetchFirstClause.....	69	ItemName.....	58
Field.....	61	ITERATE.....	57
FieldCount.....	49	JoinedTable.....	69
Filler.....	65	JoinType.....	69
FILTER_STATEMENT.....	58	JSON.....	62
FilterStatement.....	59	Label.....	59
First.....	74	Labels.....	65
FIRST.....	69	Last.....	74
FIRST_VALUE.....	74	LAST.....	69
FloatType.....	67	LAST_VALUE.....	74
Floor.....	74	LastData.....	74
FOR_STATEMENT.....	58	LEAVE.....	57
Foreign Position.....	126	LEFT.....	69
ForStatement.....	59	LEGEND.....	62
FromClause.....	72	LengthExpression.....	74
FULL.....	69	LET_STATEMENT.....	58
FuncOpt.....	73	LetDef.....	59
FunctionCall.....	73	LetStatement.....	59
Fusion.....	75	Level.....	62
GetDiagnostics.....	58	LEVEL.....	62
GetFieldType.....	49		
GetFileNames.....	111		

Like.....	73	OfGraphType.....	65
LimitClause.....	69	Open.....	60
LINE.....	62	OpenGraphType.....	65
Literal.....	71	ORDER_BY_AND_PAGE_STATEMENT.....	58
LobType.....	67	OrderByClause.....	69
LOCALTIME.....	75	OrderByStatement.....	59
LONGEST.....	59	Ordering.....	63
LoopStatement.....	59	OrderSpec.....	69
Match.....	59	OrdinaryGroup.....	72
MATCH_STATEMENT.....	58	OUTER.....	69
MatchEdge.....	59	OVERRIDING.....	60
MatchItem.....	59	OWNER.....	66
MatchMode.....	59	Parameter.....	60
MatchNode.....	59	PARAMETER_MODE.....	58
MatchPath.....	59	PARAMETER_NAME.....	58
MatchQuantifier.....	59	PARAMETER_ORDINAL_POSITION.....	58
MatchStatement.....	59	Parameters.....	60
Maximum.....	74	PartitionClause.....	72
Member.....	73	Path.....	68
MESSAGE_LENGTH.....	58	PeriodBinary.....	73
MESSAGE_OCTET_LENGTH.....	58	PeriodName.....	64
MESSAGE_TEXT.....	58	PeriodPredicand.....	73
Metadata.....	62	PIE.....	62
METADATA.....	66	POINTS.....	62
Method.....	60	Port.....	46
MethodCall.....	60	Position.....	75
MethodType.....	60	PowerFunction.....	75
Metric.....	75	Predicate.....	73
MILLI.....	62, 128	PREFIX.....	62
MIME.....	62	Prepare.....	111
Minimum.....	74	Privileges.....	66
Modulus.....	74	Properties.....	65
MongoDB.....	67	Provider.....	46
MONOTONIC.....	38	PyrrhoArray.....	107, 109
MORE.....	58	PyrrhoConnect.....	107
MULTIPLICITY.....	62	PyrrhoInterval.....	107, 112
MULTISET.....	70	PyrrhoRow.....	107, 112, 113
MultisetOp.....	71	Query.....	68
NATURAL.....	69	REAL.....	67
NaturalLogarithm.....	74	ReferentialAction.....	64
navigation properties.....	46	RefObj.....	64
NCLOB.....	67	REMOVE_STATEMENT.....	58
Next.....	74	RemoveAction.....	64
Node.....	65, 68	Rename.....	65
NODETYPE.....	62	Repeat.....	60
NodeTypeDetails.....	65	Representation.....	63
Normalize.....	75	ResetReader.....	111
Null.....	73	RESTRICT.....	64
Nullif.....	74	<i>Restrict Delete</i> .....	97, 126
NULLS.....	69	Restrict Update.....	96, 126
NUMBER.....	58	RESTView.....	64
NumericType.....	67	Return.....	60
NumericValueFunction.....	74	RETURNED_SQLSTATE.....	58
ObjectName.....	66	Revoke.....	66
ObjectPrivileges.....	66	RIGHT.....	69
OCTET_LENGTH.....	74	ROLLBACK.....	57
Of.....	73	ROLLBACK_COMMAND.....	58
OffsetClause.....	69	Routine.....	66
		ROUTINE_CATALOG.....	58

ROUTINE_NAME .....	58	TableClause .....	63
ROUTINE_SCHEMA .....	58	TableConstraint .....	63
ROW_COUNT .....	58	TableConstraintDef .....	63
RowNumber .....	75	TableContents .....	63
RowSet .....	68	TableExpression .....	72
RowSetSpec .....	69	TableFactor .....	72
RowValue .....	72	TablePeriodDefinition .....	64
Scalar .....	69	TableReference .....	72
Schema .....	75	TableValue .....	71
SCHEMA_NAME .....	58	TableVar .....	58
SchemaDetails .....	65	Target .....	57
SearchCondition .....	69	TargetList .....	60
SECURITY .....	34, 62, 70	TicksPerSecond .....	112
SelectItem .....	69	TimePeriodSpecification .....	69
SelectList .....	69	TIMESTAMP .....	71
SelectSingle .....	60	TRANSACTION_ACTIVE .....	58
SENSITIVE .....	62	<i>Transaction2</i> .....	124
SERVER_NAME .....	58	TransactionConflict .....	107
SESSION_CLOSE_COMMAND .....	58	TRANSACTIONS_COMMITTED .....	58
SESSION_RESET_COMMAND .....	58	TRANSACTIONS_ROLLED_BACK .....	58
SESSION_SET_BINDING_TABLE_PARAMETER_COMMAND .....	58	TREAT .....	75
SESSION_SET_PROPERTY_GRAPH_COMMAND .....	58	Treatment .....	69
SESSION_SET_PROPERTY_GRAPH_PARAMETER_COMMAND .....	58	Trigger .....	64
SESSION_SET_SCHEMA_COMMAND .....	58	TRIGGER_CATALOG .....	58
SESSION_SET_TIME_ZONE_COMMAND .....	58	TRIGGER_NAME .....	58
SESSION_SET_VALUE_PARAMETER_COMMAND .....	58	TRIGGER_SCHEMA .....	58
Set .....	75	TriggerCond .....	64
Set Default Delete .....	97, 126	TriggerDefinition .....	64
Set Default Update .....	97, 126	Truncation .....	59
Set Null Delete .....	97, 126	TruncationSpec .....	59
Set Null Update .....	97, 126	Type .....	66
SET_STATEMENT .....	58	TypedTableElement .....	64
SetAuthority .....	111	UNBOUNDED .....	72
SetFunction .....	75	UNICODE .....	67
Signal .....	58	UNION .....	71
SIGNAL .....	58	Unique .....	73
Some .....	73	UNNEST .....	72
SPECIFIC_NAME .....	58	Update .....	69
SQLAGENT .....	62	URL .....	62
SquareRoot .....	75	UseGraph .....	65
StandardType .....	66	USER .....	62
START_TRANSACTION_COMMAND .....	58	UserFunctionCall .....	60
StartField .....	71	VALID .....	73
StartTimestamp .....	107	Value .....	69
StartTransaction .....	107	Values .....	64
Statements .....	56	VALUES .....	71
STATIC .....	60	ValueVar .....	58
StringValueFunction .....	75	VariableRef .....	70
SUBCLASS_ORIGIN .....	58	VectorConstructor .....	75
Substring .....	75	VectorDimensionCount .....	75
SUFFIX .....	62	VectorDistance .....	75
Sum .....	75	VectorFunctions .....	75
System.Type .....	49	VectorNorm .....	75
TABLE_NAME .....	58	VectorSerialize .....	75
		Versioned .....	111
		VERSIONING .....	76
		VersioningClause .....	63

<b>ViewDefinition</b> .....	19, 64	<b>WindowFrame</b> .....	72
<b>ViewSepecification</b> .....	64	<b>WindowSpec</b> .....	74
<b>WhenStatement</b> .....	59	<b>WindowStart</b> .....	72
<b>WhereClause</b> .....	72	<b>WithinGroup</b> .....	74
<b>While</b> .....	60	<b>X</b> .....	62
<i>window function</i> .....	73	<b>Y</b> .....	62
<b>WindowBetween</b> .....	72		
<b>WindowBound</b> .....	72		