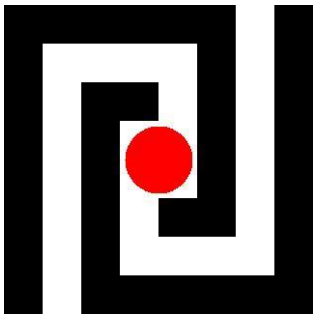
An Introduction to the Source Code of the Pyrrho DBMS

Malcolm Crowe, University of the West of Scotland

www.pyrrhodb.com



Version 7.0 (August 2020)

1. Introduction

For a general introduction to using the Pyrrho DBMS, including its aims and objectives, see the manual that forms part of the distribution of Pyrrho. The web site pyrrhodb.com includes a set of clickable pages for the SQL syntax that Pyrrho uses. This document is for programmers who intend to examine the source code, and includes (for example) details of the data structures and internal locks that Pyrrho uses.

All of the implementation code of Pyrrho remains intellectual property of the University of the West of Scotland, and while you are at liberty to view and test the code and incorporate it into your own software, and thereby use any part of the code on a royalty-free basis, the terms of the license prohibit you from creating a competing product or from reverse engineering the professional edition.

I am proud to let the community examine this code, which has been available since 2005: I am conscious of how strongly programmers tend to feel about programming design principles, and the particular set of programming principles adopted here will please no one. But, perhaps surprisingly, the results of this code are robust and efficient, and the task of this document is to try to explain how and why.

This document has been updated to version 7 (July 2019) and aims to provide a gentle introduction for the enthusiast who wishes to explored the source code, and see how it works. The topics covered include the workings of all levels of the DBMS (the server) and the structure of the client library PyrrhoLink.dll. Over the years some features of Pyrrho and its client library have been added and others removed. At various times there has been support for Rdf and SPARQL, for Java Persistence, for distributed databases, Microsoft's entity data models and data adapters, and MongoDB-style $ operators. These have been mostly removed over time: some support for Documents and Mandatory Access Control remains. In particular, the notion of multi-database connections is no longer supported.

Much of the structure and functionality of the Pyrrho DBMS is documented in the Manual. The details provided there include the syntax of the SQL2016 language used, the structure of the binary database files and the client-server protocol. Usage details from the manual will not be repeated here. In a few cases, some paragraphs from the user manual provide an introduction to sections of this document. The current version preserves the language, syntax and file format from previous versions and should be mostly[1] compatible with existing database files (whether from Open Source or professional editions). From this version, there is only one edition of Pyrrho, the executables are called Pyrrho… and use append storage. All of the binaries work on Windows and Linux (using Mono). The EMBEDDED option is for creating a class library called EmbeddedPyrrho, with an embedded Pyrrho engine, rather than a database server.

The basic structure of the engine is completely changed in version 7. Many of the low-level internal details follow the design of StrongDBMS (see strongdbms.com), and all of the upper layers of the Pyrrho engine have been redesigned to use shareable data structures similar to StrongDBMS. The implementation of roles is also completely redesigned, so that each role may have its own definition of tables, procedures, types and domains; the database schema role is used for operations on base tables.

The reader of this document is assumed to be a database expert and competent programmer. The DBMS itself has over 600 C# classes, spread over roughly 100 source files in 6 namespaces. The Excel worksheet Classes.xls lists all of the classes together with their location, superclass, and a brief description. The code itself is intended to be quite readable, with the 2019 version (C#7) of the language, notably including its ValueTuple feature.

This document avoids having a section for each class, or for each source file. Either of those designs for this document would result in tedious repetition of what is in the source. Instead, the structure of this document reflects the themes of design, with chapters addressing the role in the DBMS of particular groupings of related classes or methods.

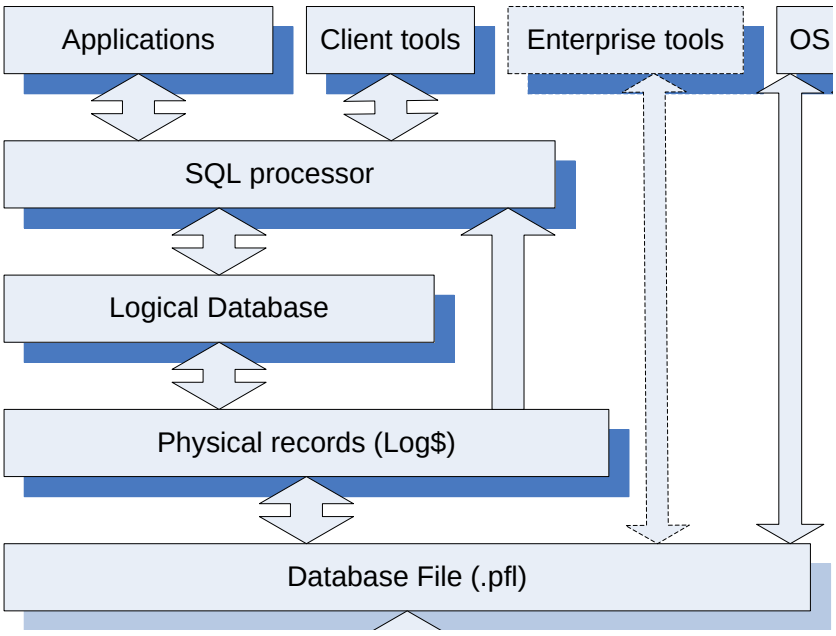
[1] An important exception is that from version 7, Pyrrho does not allow schema modifications to objects that contain data. This affects the use of databases from previous versions that record such modifications (see 2.5.6).

2. Overall structure of the DBMS

2.1 Architecture

The following diagram shows the DBMS as a layered design. There is basically a namespace for each of four of the layers, and two other namepsaces, Pyrrho and Pyrrho.Common.

Namespace	Title in the diagram	Description
Pyrrho		The top level contains only the protocol management files Start.cs, HttpService.cs and Crypt.cs
Pyrrho.Common		Basic data structures: Integer, TypedValue, BTree, and the lexical analyzer for SQL. All classes in Common, including Bookmarks for traversing them, are immutable and shareable.
Pyrrho.Level1	Database File, Database File Segments	Binary file management and buffering.
Pyrrho.Level2	Physical Records	The Physical layer: with classes for serialisation of physical records.
Pyrrho.Level3	Logical Database	Database.cs, Transaction.cs, Value.cs and classes for database objects. All classes in Level3 are immutable and shareable.
Pyrrho.Level4	SQL processing	RowSet.cs, Parser.cs etc. Cursors are immutable and shareable, and give access to the version of the RowSet that created them.



2.2 Key Features of the Design

The following features are really design principles used in implementing the DBMS. There are important modifications to these principles that apply from v7.

1. Transaction commits correspond one-to-one to disk operations: completion of a transaction is accompanied by a force-write of a database record to the disk. The engine waits for this to complete in every case. Some previous versions of Pyrrho had a 5-byte end-of-file marker which was overwritten by each new transaction, but from version 7, all physical records once written are immutable. Deletion of records or database objects is a matter for the logical database, not the physical database. This makes the database fully auditable: the records for each transaction can always be recovered along with details about the transaction (the user, the timestamp, the role of the transaction).

2. Because data is immutable once recorded, the physical position of a record in the data file (its "defining position") can be used to identify database objects and records for all future time (as names can change, and update and drop details may have a later file position). The transaction log threads together the physical records that refer to the same defining position but, from version 7, Pyrrho maintains the current state of base table rows in memory (using the TableRow class), and does not follow such non-scalable trails

3. Data structures at the level of the logical database (Level 3) are immutable and shareable. For example, if an entry in a list is to be changed, what happens at the data structure level is that a replacement element for the list is constructed and a new list descriptor which accesses the modified data, while the old list remains accessible from the old list descriptor. In this way creating a local copy or snapshot of the database (which occurs at the start of every transaction) consists merely to making a new header for accessing the lists of database objects etc. As the local transaction progresses, this header will point to new headers for these lists (as they are modified). If the transaction aborts or is rolled back, all of this data can be simply forgotten, leaving the database unchanged. With this design total separation of concurrent transactions is achieved, and local transactions always see consistent states of the database.

4. When a local transaction commits, however, the database cannot simply be replaced by the local transaction object, because other transactions may have been committed in the meantime. If any of these changes conflict with data that this transaction has read (read constraints) or is attempting to modify (transaction conflict), then the transaction cannot be committed. If there is no conflict, the physical records proposed in the local transaction are relocated onto the end of the database.

5. Following a successful commit, the database is updated using these same physical records. Thus all changes are applied twice – once in the local transaction and then after transaction commit – but the first can be usefully seen as a validation step, and involves many operations that do not need to be repeated at the commit stage: evaluation of expressions, check constraints, execution of stored procedures etc.

6. From version 7, database objects such as tables and domains cannot be modified if they hold data. The semantics of such changes in previous versions were not really manageable. There are necessarily several mutable structures: Reader, Writer, Context, and Physical (level 2). Physical objects are used only for marshalling serialisation and associated immutable objects replace Physicals in Level 3.

7. Data recorded in the database is intended to be non-localised (e.g. it uses Unicode with explicit character set and collation sequence information, universal time and date formats), and machine-independent (e.g. no built-in limitations as to machine data precision such as 32-bit). Default value expressions, check constraints, views, stored procedures etc are stored in the physical database in SQL2011 source form, and parsed to a binary form when the database is loaded.

8. The database implementation uses an immutable form of B-Trees throughout (note: B-Trees are *not* binary trees). Lazy traversal of B-Tree structures (using immutable bookmarks) is used throughout the query processing part of the database. This brings dramatic advantages where search conditions can be propagated down to the level of B-Tree traversal.

9. Traversing a rowset recovers rows containing TypeValues, and the bookmark for the current row becomes is accessible from the Context. This matches well with the top-down approach to parsing and query processing that is used throughout Level 4 of the code. In v7, the evaluation stack is somewhat flattened. A new Context is pushed on the context stack for a new procedure block or activation, or when there is a change of role. The new context receives a copy of the previous context's immutable tree structures, which are re-exposed when the top of the stack is removed.

10. The aim of SQL query processing is to bridge the gap between bottom-up knowledge of traversable data in tables and joins (e.g. columns in the above sense) and top-down analysis of value expressions. Analysis of any kind of query goes through a set of stages: (a) source analysis to establish where the data is coming from, (b) computation of the result data types, (c) conditions analysis which examines which search and join conditions can be handled in table enumeration, (d) ordering analysis which looks not only at the ordering requirements coming from explicit ORDER BY requests from the client but also at the ordering required during join evaluation and aggregation, and finally (e) RowSet construction, which chooses the best enumeration method to meet all the above requirements.

11. Executables and SqlValues are immutable level 3 objects that are constructed by the parser. They are not stored in the database, but reconstructed on creation or loading. Procedure bodies being read from the database can contain SqlValues with positions allocated according to the current Reader position. Objects constructed from the input stream of the transaction can use the position in the input stream provided that this accumulates from the start of the transaction rather than the start of the current input line. This is the responsibility of Server.Execute(sql) and SqlHTTPService parser calls.

2.3 Multi-threading, uids, and dynamic memory layout

In accordance with the above notes, each Connection has its own PyrrhoServer instance in a separate thread (Pyrrho has no other threads). There is a static set of immutable copies of databases (as committed) and filenames from which a new server instance will start with the committed version of the database it will work with. This set is initially empty accessible from all server threads and protected by the only lock used by Pyrrho. Initialisation also sets up the _system database, containing primitive types and system tables. Every database structure includes this immutable information. No other cross-thread access is possible in Pyrrho.

Unique identifiers are central to the v7 design of Pyrrho. At the database level (level 3) of the design, each object (including Database and Transaction) contains an association called mem indexed by 64-bit uids. Importantly, uids are also used at level 4 of the engine for run-time data structures, in Contexts and Activations, which manage a similar association called values. This section outlines a rationale for the allocation of uids and the significance of their ranges of values.

Databases contain committed data, which uses two ranges of uids. A fixed set of approximately 1000 uids<0 are used for a set of system objects (constants, tables, domains), and file positions in range $0..2^{62}$-1 are used to identify committed objects in the database. Some objects with uids in this range are indexed in the objects tree so they can be referenced elsewhere. The Role object allows object uids to be found by name (objects can be renamed by roles). Apart from such referencing, uids are used in evaluation contexts to manage values and object visibility, and to identify expressions that are equivalent, so the uniqueness of uids is very important in this design.

Transactions contain uncommitted objects, so the uids must be managed differently. A transaction allocates uids above 4×2^{60} for uncommitted objects (e.g. proposed new physical records). These uids are retained until the transaction is committed or rolled back and form a natural stack. On commit, the transaction's objects are serialised to the data file, whereupon the uids are replaced with the committed file positions.

This means that the transaction works with a mixture of committed and uncommitted database objects (table, column, domain etc). Any query processed by the transaction may contain multiple references to the same tables and columns, which may have different values[2] so that each reference gets a new uid.

So far so good. However, not all uncommitted objects have the same lifetime. Prepared statements are connection based and so persist beyond the end of a transaction, and triggers and stored procedures become dynamic objects with shorter lifetimes (command, statement, etc). If a server is to run for a long time, available memory would be exhausted. It is not attractive to intervene at various stages to avoid exhaustion of the available space, traversing the lists of uncommitted objects to compactify the lists. As an experiment, the current implementation seeks to stratify things by identifying different uid ranges for different treatment. We begin by limiting the range for uncommitted physical records to $4\times2^{60}..5\times2^{60}..-1$. The lifetime of objects in this range is the duration of the transaction.

During query analysis, transactions allocate space for objects local to the processing of the current Command. Uids in the range $5\times2^{60}..6\times2^{60}$-1are allocated based on the lexical position of objects in the command text (see worked example in sec 6). The highwatermark for this process is called nextId (in the Context). Thus, all identifiers that occur in the SQL are replaced during parsing with uids in this range as allocated on the first occurrence in the command text. Columns not referred to will be given temporary uids from the nextHeap range, described below: they cannot use their defining position as this might conflict via a separate reference to the table in the SQL (subqueries, views etc).

On Load(), it is not appropriate to use lexical uids, and the range $6\times2^{60}..7\times2^{60}$-1 is used instead (statement uids) when stored executable code is parsed. Uids in this range persist in the in-memory

[2] Obviously, column references will have different values in different rows. SqlValue evaluation uses the appropriate cursor to obtain the value for a row. During trigger operation the same is true for old/new tables.

database and are shared with future transactions for this database. The highwatermark is called db.nextStmt[3] and this is persisted in the Database structure.

The only place that physical column ids are used after parsing is for Insert and Update where TransitionRowSets will use the physical uids of columns in the affected table to construct the new physical records without using any role-based column ordering.

As a result of the above considerations, the replacement of identifier(chain)-based references with uids proceeds from left to right during parsing. When source identifiers are resolved to column references, a lexical id is given to the column reference. If the resulting DBObjects are serialised to the database (stored procedures, triggers), the source code only is saved in the transaction log, and instead of reparsing, each such lexical or heap uid is replaced by a physical or (respectively) statement uid based on the permanent file position of the lexeme. Finally, we note that several roles may be involved, so that columns in a table may be defined by different roles, and be differently referred to in stored procedures, triggers, and constraints.

A "connection" range of uids, $7\times2^{60}..8\times2^{60}-1$ is for prepared statements, as these accumulate and are shared with future transactions for this connection, but are not committed. Each transaction starts with the current database snapshot and this set of prepared statements (the highwatermark is called db.nextPrep). The temporary uids mentioned above work in reverse for prepared statements as it is the uids in the query analysis range that get relocated to the prepared statement range. In the Context there are twin functions called Unheap and UnLex that deal with these contrary relocations.

Schema changes cannot be introduced during such execution of stored procedures, and so the heap is local to the current Command: the execution context initialises cx.nextHeap using db.nextPrep.

System objects	$-8\times2^{60}..-1$	Basis._uid (downwards)	Global	
File positions	$0..4\times2^{60}-1$	0 (upwards)	Presisted in file	
New Physicals	$4\times2^{60}..5\times2^{60}-1$	Database.nextPos (up)	Local to Transaction	'
Query analysis	$5\times2^{60}..6\times2^{60}-1$	Database.nextIid (up)	Local to Statement	#
Executables	$6\times2^{60}..7\times2^{60}-1$	Database.nextStmt (up)	Local to Database	@
Prepared Stmts	$7\times2^{60}..$ nextPrep	Database.nextPrep (up)	Local to Connection	%
Heap storage	nextPrep.. $8\times2^{60}-1$	Context.nextHeap (up)	Local to Command	%

The boundaries of these ranges are subject to change in later versions, as they are internal to the engine and not relevant to durable file contents. Allocation of uids in each ranges need to be independent for the following reasons:

- File positions: audit requires asynchronous writing to the transaction log during a transaction. Such asynchronous writing cannot occur during the transaction commit as this process occupies the thread.
- New Physicals: can be created because of triggers at various points during a transaction step. There is a dependency field that helps to ensure that serialisation duing commit takes place in an orderly way.
- Query analysis objects such as cursors need to be managed within statement processing, but the statement Obey() can cause creation of new physicals.
- Storage for compiled statements is local to a database, is immutable, and can be used by successive steps in a transaction and successive transactions.
- The prepared statement storage is semi-persistent and shared among sequential transactions in a single connection independently of commit. This storage can be shared with heap storage.

Contexts form a tree-like stack of frames, providing an easy support for recursive procedure execution, and in the SQL programming language, dynamic structures are accessible only by direct reference.

2.4 The folder and project structure for the source code

The src folder contains

[3] When debugging, it is useful to know that the uids allocated on Load() will be the same every time and will match those allocated on Commit().

- Folders for the Pyrrho applications: PyrroCmd (including PyrrhoStudio for preparing embedded databases), PyrrhoJC, and PyrrhoSQL.
- The Shared folder contains the sources for the PyrrhoDBMS engine and the Pyrrho API and this arrangement is described next.

The Shared folder contains files and folders for the 3 currently supported overlapping solutions EmbeddedPyrrho (EP), PyrrhoLink (PL), and PyrrhoSvr (PS).

- The Properties folder handles Visual Studio project stricture. Unusually, it has subfolders for isolating the AssemblyInfo for each of the 3 solutions.
- The Common, Level1, Level2, Level3, and Level4 folders contain the real code base for the DBMS..
- Any change to a Connection requires a Transaction instance. It will have the client's Connection instance as its parent. When the transaction is configured a Participant instance is created for each Database in the connection, with a VirtBase wrapping its Level 2 PhysBase. This gives a set of snapshots of the database at the start of the transaction.
- Transaction instances are also created to validate rename, drop and delete operations before these are executed.

3. Basic Data Structures

In this chapter we discuss some of the fundamental data structures used in the DBMS. Data structures selected for discussion in this chapter have been chosen because they are sufficiently complex or unusual to require such discussion. All of the source code described in this section is in the Pyrrho.Common namespace: all of the classes in this namespace are immutable and shareable.

3.1 B-Trees and BLists

Almost all indexing and cataloguing tasks in the database are done by BTrees. These are basically sorted lists of pairs (key,value), where key is comparable. In addition, sets and partial orderings use a degenerate sort of catalogue in which the values are not used (and are all the single value **true**).

There are several subclasses of BTree used in the database: Some of these implement multilevel indexes. BTree itself is a subclass of an abstract class called ATree. The BTree class provides the main implementation. These basic tree implementations are generic, and require a type parameter, e.g. BTree<long,bool> . The supplied type parameters identify the data type used for keys and values. BTree is used when the key type is an IComparable value type. If the key is a TypedValue, CTree is used instead.

See 3.1.5 for a list of related B-Tree classes. All except RTree are immutable and shareable.

3.1.1 B-Tree structure

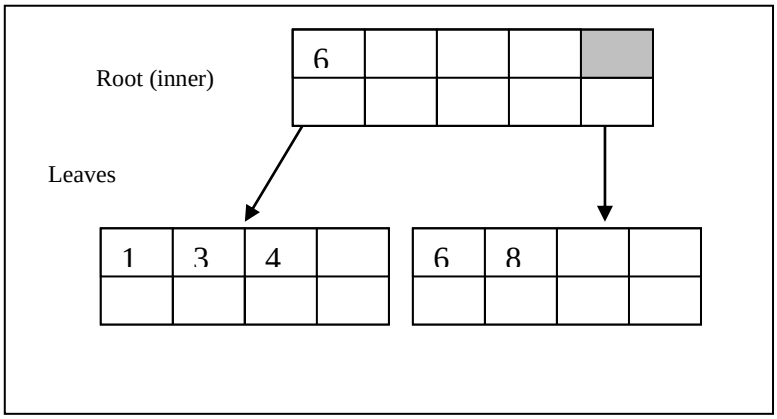
The B-Tree is a widely-used, fully scalable mechanism for maintaining indexes. B-Trees as described in textbooks vary in detail, so the following account is given here to explain the code.

A B-Tree is formed of nodes called Buckets. Each Bucket is either a Leaf bucket or an Inner Bucket. A Leaf contains up to N KeyValuePairs. An Inner Bucket contains key-value pairs whose values are pointers to Buckets, and a further pointer to a Bucket, so that an Inner Bucket contains pointers to N+1 Buckets altogether ("at the next level"). In each Bucket the KeyValuePairs are kept in order of their key values, and the key-value pairs in Inner buckets contain the first key value for the next lower-level Bucket, so that the extra Bucket is for all values bigger than the last key. All of these classes take a type parameter to indicate the key type.

The value of N in Pyrrho is currently 8: the performance of the database does not change much for values of N between 4 and 32. For ease of drawing, the illustrations in this section show N=4.

The BTree itself contains a root Bucket and some other data we discuss later.

The BTree dynamically reorganizes its structure so that (apart from the root) all Buckets have at least N/2 key-value pairs, and at each level in the tree, Buckets are either all Inner or all Leaf buckets, so that the depth of the tree is the same at all values.



3.1.2 ATree<K,V>

The basic operations on B-Trees are defined in the abstract base class ATree<K,V>:, ATree<K,V>.Add, ATree<K,V>.Remove etc, and associated operators + and -.

For a multilevel index, Key can be an array or row (this is implemented in MTree and RTree, see section 3.2).

The following table shows the most commonly-used operations:

Name	Description
long Count	The number of items in the tree

object this[key]	Get the value for a given key
bool Contains(key)	Whether the tree contains the given key
ABookmark<K,V> First()	Provides a bookmark for the first pair in the B-tree
static Add(ref T, Key, Value)	For the given tree T, add entry Key,Value .
static Remove(ref T, Key)	For the given tree T, remove the association for Key.

However, in version 7 these fundamental operations are made protected, and modifications to B-Trees uses + and – operators. So, to add a new (key,value) pair to B-Tree t, we write code such as

```
t += (key,value);
```

and to remove a key we write t -= key; . The current version of Visual Studio colours the operator brown to indicate the use of a custom method.

Some B-Trees have values that are also B-Trees, and for these it is convenient to define addition and removal operators for different tuple types (such as triples).

3.1.3 TreeInfo

There are many different sorts of B-Tree used in the DBMS. The TreeInfo construct helps to keep track of things, especially for multilevel indexes (which are used for multicolumn primary and foreign keys).

TreeInfo has the following structure:

Name	Description
Ident headName	The name of the head element of the key
Domain headType	The data type of the head element of the key
Domain kType	Defines the type of a compound key.
TreeBehaviour onDuplicate	How the tree should behave on finding a duplicate key. The options are Allow, Disallow, and Ignore. A tree that allows duplicate keys values provides an additional tree structure to disambiguate the values in a partial ordering.
TreeBehaviour onNullKey	How the tree should behave on finding that a key is null (or contains a component that is null). Trees used as indexes specify Disallow for this field.
TreeInfo tail	Information about the remaining components of the key

3.1.4 ABookmark<K,V>

Starting with version 6.0 of Pyrrho, we no longer use .NET IEnumerator interfaces, replacing these with immutable and thread-safe structures. Bookmarks mark a place in a sequence or tree, and allow moving on to the next item. Every B-Tree provides a method First() that returns a bookmark for the first element of the tree (or null if the tree is empty).

ABookmark<K,V> has a method Next() which returns a bookmark for the next element if any.

Name	Description
ABookmark<K,V> Next()	
K key()	The key at the current position
long position()	The current position (stars at 0)
V value()	The value at the current position

Cursors follow a similar pattern (see section 3.6.10)

3.1.5 ATree<K,V> Subclasses

Other implementations provide special actions on insert and delete (e.g. tidying up empty nodes in a multilevel index).

The main implementation work is shared between the abstract BTree<K,V> and Bucket<K,V> classes and their immediate subclasses.

There are just 5 ATree subclasses, all sharing the same base implementation::

Name	BaseClass	Description
BList<V>	BTree<K,V>	Same as BList<V> with a shortcut for adding to the end
CList<V>	BList<V>	Same as BList<V> where V is IComparable, allows

		comparison of lists
BTree<K,V>	ATree<K,V>	The main implementation of B-Trees, for a one-level key that is IComparable
CTree<K,V>	BTree<K,V>	A similar class where the key is a TypedValue
SqlTree	CTree<TypedValue, TypedValue>	For one-level indexes where the keys and values have readonly strong types
Idents	BTree<string, (DBObject,Idents)>	This behaves like a lookup tree for Ident->DBObject where the comparison of Idents is as identifierchains.

The following related immutable classes are contained in the Level3 and Level4 namespaces. Neither of these is a subclass of ATree.

MTree	For multilevel indexes where the value type is long?
RTree	For multilevel indexes where the value type is SqlRow:

3.2 Other Common Data Structures

3.2.1 Integer

All integer data stored in the database uses a base-256 multiple precision format, as follows: The first byte contains the number of bytes following.

#bytes (=n, say)	data0	data1	…	data(n-1)

data0 is the most significant byte, and the last byte the least significant. The high-order bit 0x80 in data0 is a sign bit: if it is set, the data (including the sign bit) is a 256s-complement negative number, that is, if all the bits are taken together from most significant to least significant, that data is an ordinary 2s-complement binary number. The maximum Integer value with this format is therefore 2^{2039}-1 .

Some special values: Zero is represented as a single byte (0x00) giving the length as 0. -1 is represented in two bytes (0x01 0xff) giving the length as 1, and the data as -1. Otherwise, leading 0 and -1 bytes in the data are suppressed.

Within the DBMS, the most commonly used integer format is long (64 bits), and Integer is used only when necessary.

With the current version of the client library, integer data is always sent to the client as strings (of decimal digits), but other kinds of integers (such as defining positions in a database, lengths of strings etc) use 32 or 64 bit machine-specific formats.

The Integer class in the DBMS contains implementations of all the usual arithmetic operators, and conversion functions.

3.2.2 Decimal

All numeric data stored in the database uses this type, which is a scaled Integer format: an Integer mantissa followed by a 32-bit scale factor indicating the number of bytes of the mantissa that represent a fractional value. (Thus strictly speaking “decimal” is a misnomer, since it has nothing to do with the number 10, but there seems no word in English to express the concept required.)

Normalisation of a Decimal consists in removing trailing 0 bytes and adjusting the scale.

Within the DBMS, the machine-specific double format is used.

With the current version of the client library, numeric data is always sent to the client in the Invariant culture string format.

The Decimal class in the DBMS contains implementations of all the usual arithmetic operations except division. There is a division method, but a maximum precision needs to be specified. This precision is taken from the domain definition for the field, if specified, or is 13 bytes by default: i.e. the default precision provides for a mantissa of up to 2^{103}-1 .

3.2.3 Character Data

All character data is stored in the database in Unicode UTF8 (culture-neutral) format. Domains and character manipulation in SQL can specify a "culture", and string operations in the DBMS then conform to the culture specified for the particular operation.

The .NET library provides a very good implementation of the requirements here, and is used in the DBMS. Unfortunately .NET handles Normalization a bit differently from SQL2011, so there are five low-level SQL functions whose implementation is problematic.

3.2.4 Documents

From v.5.1 Pyrrho includes an implementation of Documents similar to MongoDB, however the $ operators of MongoDB are not provided from v7.

Document comparison is implemented as matching fields: this means that fields are ignored in the comparison unless they are in both documents (the $exists operator modifies this behaviour). This simple mechanism can be combined with a partitioning scheme, so that a simple SELECT statement where the where-clause contains a document value will be propagated efficiently into the relevant partitions and will retrieve only the records where the documents match. Moreover, indexes can use document field values.

3.2.5 Domain

Strong types (Domains) are used internally for all processing. ObInfos are role objects used in the Database layer, in the sense that the role contains details of object names, column ordering, and accessibility for the role: see section 3.5.7. In the Database layer (level 3), objects have uids but not names. In the Transaction layer (level 4), SqlValues and Queries have names for the current role (the current role and user are maintained by the Context).

The Domain provides methods of input and output of data, parsing and formatting of value strings, coercing, checking assignability etc. Domain representations allow for sub-columns defined by uid and Domain, but do not specify a column ordering.

At the Transaction level, ad-hoc domains are supported, and defining position of a Domain is ignored. Domains are equal if they have the same properties (other than the defpos), and there is an arbitrary comparison method based on properties.

On the other hand, all level 3 (committed) objects have domains that have defining positions. In particular user-defined Types are implemented as Domains.

The following well-known standard types are defined by the Domain class:

Name	Description
Null	The data type of the null value
Wild	The data type of a wildcard for traversing compound indexes
Bool	The Boolean data type (see BooleanType)
RdfBool	The iri-defined version of this
Blob	The data type for byte[]
MTree	Multi-level index (used in implementation of MTree indexes)
Partial	Partially-ordered set (ditto)
Char	The unbounded Unicode character string
RdfString	The iri-defined version of this
XML	The SQL XML type
Int	A high-precision integer (up to 2048 bits)
RdfInteger	The iri-defined version of this (in principle unbounded)
RdfInt	`value>=-2147483648 and value<=2147483647`
RdfLong	`value>=-9223372036854775808 and value<=9223372036854775807`
RdfShort	`value>=-32768 and value<=32768`
RdfByte	`value>=-128 and value<=127`
RdfUnsignedInt	`value>=0 and value<=4294967295`
RdfUnsignedLong	`value>=0 and value<=18446744073709551615`
RdfUnsignedShort	`value>=0 and value<=65535`

RdfUnsignedByte	`value>=0 and value<=255`
RdfNonPositiveInteger	`value<=0`
RdfNegativeInteger	`value<0`
RdfPositiveInteger	`value>0`
RdfNonNegativeInteger	`value>=0`
Numeric	The SQL fixed point datatype
RdfDecimal	The iri-defined version of this
Real	The SQL approximate-precision datatype
RdfDouble	The iri-defined version of this
RdfFloat	Defined as Real with 6 digits of precision
Date	The SQL date type
RdfDate	The iri-defined version of this
Timespan	The SQL time type
Timestamp	The SQL timestamp data type
RdfDateTime	The iri-defined version of this
Interval	The SQL Interval type
Collection	The SQL array type
Multiset	The SQL multiset type
UnionNumeric	A union data type for constants that can be coerced to numeric or real
UnionDate	A union of Date, Timespan, Timestamp, Interval for constants

See also sec 3.5.3.

3.2.6 TypedValue

A TypedValue has a Domain and an ordering of columns, and a tree values. TypedValues are immutable, even TArray, TMultiset and TDocument. As with all immutable objects operators such as + provide a new TypedValue. The following lists the subclasses of TypedValue:

Cursor
Delta
TArray
TBlob
TBool
TChar
TContext
TDateTime
TDocArray
TDocument
TInt
TInterval
TMTree
TMultiset
TNull
TNumeric
TPartial
TPeriod
TReal
TRegEx
TRvv
TTimeSpan
TTypeSpec
TUnion
TXml

3.2.7 Ident

An Ident is a dotted identifier chain, and is used to support the analysis of SQL queries during parsing. This construct appears in multiple places in the syntax (see below). Ident is immutable.

CompareTo(ob)	Support alphanumeric comparison of Ident
string ident	The head portion of the Ident

Ident(…)	*Numerous constructors*
long iix	A unique uid, usually obtained from the lexer position
int Length	The number of segments in the Ident
Ident sub	The tail of the Ident
string ToString()	A readable version of the Ident

There is a special tree structure Ident.Idents for handling definitions during parsing. Formally it is a subclass of BTree<string,(long, Ident.Idents)>. It contains SqlValues and Queries indexed by name for the current role (not ObInfo, Domain, or any sort of TypedValues). During parsing, subobject information is added in the Ident.Idents part to deal with query aliases (but not internal structure of SqlValues). The idea is as follows:

Given an Ident chain, there are three possibilities: (a) the chain identifies a unique SqlValue or query, (b) the first part of the chain identifies a query, document or structured object and the rest of the chain leads to a field or child object, (c) the chain is a reference to something that the parse has not yet reached.

There are two lookup this[] functions: one that takes an Ident and returns the DBObject associated, and another that works on the first part of an ident chain. It takes a pair (Ident, int) and retains a pair (DBObject, Idents) giving the object reached and the subtree from that point. There is also a this[] function that takes a string, inherited from the BTree<(DBObject, Ident.Idents)> superclass.

During join processing, column names that are ambiguous and not referenced in the query may get renamed with a dotted notation, similar to a chain. In this case, the aliased column name is treated a s string containing a dot, not a chain. See an example of this process in section 6.1.

3.3 File Storage (level 1)

At this level, the class IOBase manages FileStreams, with ReaderBase and WriterBase for the encoding the data classes defined above. The Reader and Writer classes are for reading from and writing to the transaction log, and contain instantaneous snapshots of the database as it evolves during these operations. At the conclusion of Database.Load(), and Transaction.Commit the final version of the database is recorded in a static database list.

The locking required for transaction management is limited to locking the underlying FileStream during Commit(). The FileStream is also locked during seek-read combinations when Readers are created. The binary file transaction log format is almost unchanged since the earliest versions of Pyrrho: every edition of the user manual has documented the file format as a sequence of physical records[4]. It uses 8-bit bytes and Unicode UTF8 for strings, but otherwise is independent of machine architecture, operating system, or location.

There are full details of the file format in the Pyrrho Manual, together with a brief outline of the client-server protocol. Some further details are given below.

3.3.1 Client-server protocol

In auto-commit mode (implicit transactions) there is generally no acknowledgement of a successful end of the transaction. If an acknowledged service is important, use the Trace requests, or use explicit transactions: note the options of CommitAndReport and CommitAndReportTrace.
Not all services have a response byte.
Note that the request and response bytes are followed by data (for example, DoneTrace, or Schema). See the Manual.

Request	Intermediate	Final response	
(Connect/Open)		Primary	
(Error)		Exception	
		FatalError	
Authority		*Done*	
BeginTransaction			unacknowledged
CloseConnection			unacknowledged
CloseReader			unacknowledged
Commit	{Warning}	Done	

[4] Confusingly, one of the Physical subclasses, for inserting data in the database, is called Record.

CommitAndReport CommitAndReport1	{Warning}	TransactionReport	
CommitAndReportTrace CommitAndReportTrace1	{Warning}	TransactionReportTrace	
CommitTrace	{Warning}	DoneTrace	
Execute	{Warning}	Done	
		Schema	
ExecuteTrace	{Warning}	DoneTrace	
		Schema	
ExecuteNonQuery	{Warning}	Done	
ExecuteNonQueryTrace	{Warning}	DoneTrace	
ExecuteReader	{Warning}	Done	
		Schema	
Get Get1	{Warning}	Schema	
		Done	
		NoData	
Get2	{Warning}	Schema1	
		Done	
		NoData	
GetFileNames		Files	
GetInfo	{Warning}	NoData	
		Columns	
Post	{Warning} Schema	Done	
Put	{Warning} Schema	Done	
Prepare		Done	
ReaderData		NoData	
		ReaderData	
ResetReader		Done	
Rest	{Warning}	Schema	
Rollback		Done	
TypeInfo		(data)	

3.4 Physical (level 2)

Physical is the base class used for actual items stored in the database file. Physical subclasses are identified by the Physical.Type enumeration, whose values are actually stored in the database. The defining position of a Physical is given by the Reader or Writer position when reading or writing a database file, and for uncommitted objects has a uid exceeding 4×2^{60}. Uncommitted objects are those created during parsing, when the parser creates new Physical structures: and adds them to the Transaction. Since Transaction is immutable this means that each Physical gets installed in a new Transaction with a uid given by the lexical position in the source read by the transaction. This object defpos is replaced on Commit by its position in the transaction log. Every thread has its own sequence of uncommitted uids (see sec 2.3), restarting at 4×2^{60} after Commit. For ease of reading, the resulting temporary defpos are rendered in ToString() as '0, '1, '2 etc.

During Commit, the sequence of Physical records prepared by a Transaction is actually written (serialised) to durable media, and the uncommitted uids are replaced by the file positions.

Each Physical type contributes a part of the serialization and deserialization implementation. For example, an Update Physical contributes some fields, and calls its base class (Record) to continue the serialization, and finally Record calls Physical's serialization method. The Physical layer is level 2 of Pyrrho.

In version 7 many of the so-called Physical classes in memory are subclasses of Compiled, and these have a framing field structures belonging to level 4: for example, expressions and executable

statements. These objects are not serialised to or from disk: as in previous versions of Pyrrho stored procedures, queries, triggers etc are recoded in the log in source (string) form. During database Load these source strings are compiled (see sec 3.4.2 below) into an immutable form that is simply cached in the Context when required.

3.4.1 Physical subclasses (Level 2)

The type field of the Physical base class is an enum Physical.Type, as shown here:

Code	Class	Base class	Description
0	EndOfFile	Physical	Checksum record at end of file
1	PTable	Physical	Defines a table name
2	PRole	Physical	A Roleand description
3	PColumn	Physical	Defines a TableColumn
4	Record	Physical	Records an INSERT to a table
5	Update	Record	Records an UPDATE of a record
6	Change	Physical	Renaming of non-column objects
7	Alter	PColumn2	Modify column definition
8	Drop	Physical	Forget a database object
9	Checkpoint	Physical	A synchronization point
10	Delete	Physical	Forget a record from a table
11	Edit	PDomain	ALTER DOMAIN details
12	PIndex	Physical	Entity, unique, references
13	Modify	Physical	Change proc, method, trigger,check,view
14	PDomain	Physical	Define a Domain
15	PCheck	Physical	Check constraint for something
16	*PProcedure*	*Physical*	*Stored procedure/function (deprecated, see below)*
17	PTrigger	Physical	Define a trigger
18	PView	Physical	Define a niew
19	PUser	Physical	Record a user name
20	PTransaction	Physical	Record a transaction
21	Grant	Physical	Grant privileges to something
22	Revoke	Grant	Revoke privileges
23	PRole1	Physical	Record a role name
24	PColumn2	PColumn	For more column constraints
25	PType	PDomain	A user-defined structured type
26	PMethod	PProcedure	A method for a PType (deprecated, see below)
27	PTransaction2	PTransaction	Distributed transaction support
28	Ordering	Physical	Ordering for a user-defined type
29	(NotUsed)		
30	PDateType	PDomain	For interval types
31	*PTemporalView*	*Physical*	*A View for a Temporal Table (obsolete)*
32	PImportTransaction	PTransaction	A transaction with a source URI
33	Record1	Record	A record with provenance URI
34	PType1	PType	A user-defined type with a reference URI
35	PProcedure2	Physical	(PProcedure2) Specifies return type information
36	PMethod2	PProcedures	(PMethod2) Specifies return type information
37	PIndex1	PIndex	Adapter coercing to a referential constraint
38	Reference	Physical	Adapter coerces to a reference constraint
39	Record2	Record	Used for record subtyping
40	Curated	Physical	Record curation of the database
41	*Partitioned*	*Physical*	*Record a partitioning of the database*
42	PDomain1	PDomain	For OWL data types
43	Namespace	Physical	For OWL data types
44	PTable1	PTable	For OWL row types
45	Alter2	PColumn2	Change column names
46	AlterRowIri	PTable1	Change OWL row types
47	PColumn3	PColumn2	Add new column constraints

48	Alter3	PColumn3	Alter column constraints
49	*PView1*	*Pview*	*Define update rules for a view (obsolete)*
50	Metadata	Physical	Record metadata for a database object
51	PeriodDef	Physical	Define a period (pair of base columns)
52	Versioning	Physical	Specify system or application versioning
53	PCheck2	PCheck	Constraints for more general types of object
54	*Partition*	*Physical*	*Manages schema for a partition*
55	*Reference1*	*Reference*	*For cross-partition references*
56	ColumnPath	Physical	Records path selectors needed for constraints
57	Metadata2	Metadata	Additional fields for column information
58	Index2	Index	Supports deep structure
59	*DeleteReference1*	*Reference*	
60	Authenticate		Credential information for web-based login
61	RestView	View	Views defined over HTTP
62	TriggeredAction		Distinguishes triggered parts of a transaction
63	*RestView1*	*RestView*	*deprecated*
64	Metadata3	Metadata	Additional fields for column information in views
65	RestView2	RestView	Support for GET USING

3.4.2 Compiled Objects

This heading includes Triggers, Constraints, Views, Procedures and Methods. Procedures and Methods use the SQL stored persistent modules language as described in the SQL standard, including the handling of conditions (exceptions). When compiled code is invoked, it runs in the definer's role, as specified by the SQL standard.

Following the design outlined in this document, the transaction log contains only the source form of compiled objects, while the in-memory database contains the compiled version. From version 7, parsing is done only on definition, and following parsing everything is referred to by uid, not by using string identifiers. As their name implies, uids are unique in the database, but they are private to the implementation, and are subject to change is later versions of the DBMS.

> There are differences in operation of the different versions, however. Up to version 6.3 of the DBMS (file format 5.1) the source code contained database object positions instead of the definer's name for database objects. This approach is supported in version 7 of the DBMS for database files created with previous versions. Databases created with version 7 or later (file format >5.1) will contain the source code exactly as given by the definer. This is generally supported by previous versions of the DBMS, but objects will display differently in the Log$ system tables.

There is a subclass of Physical for the associated Level2 objects, to provide helper methods for the compilation process. The in-memory data structures resulting from parsing include SqlValues, Queries, and Executables. Compiled objects exist in one of three forms, as follows:

- OnLoad(), recreates the Framing (see sec 3.5.18) for the compiled object from deserialised source code. so that the compiled data structures naturally use uids based on lexical positions of source identifiers in the database file.
- At the end of parsing, the Framing field of the Compiled class receives the generated DBObjects and RowSets, with any heap uids relocated to transaction locations above 5×2^{60}. Other uids will be either those of physical objects, or locations in the source code (from the database file, or the SQL command).
- On Commit(), when the Compiled Physical is being relocated to its final (file) position, the compiled object uids are also relocated within the range of file positions occupied by the new Physical record. This enables the new compiled object to activate in the current Database.

This design means that the uids of the objects in the framing field will be different depending on which of these three stages applies. The uids that differ are not generally visible, and the in-memory compiled code structures are otherwise identical.

3.5 Database Level Data Structures (Level 3)

3.5.1 Basis

To ensure shareability and immutability of database objects, the Basis class is used as the base class for most things in Level3 of Pyrrho. Basis contains a flexible way of maintaining object properties in a in a per-object BTree<long,object> called mem. Recall that BTree and its subclasses are themselves immutable.

```
// negative keys are for system data, positive for user-defined data
public readonly BTree<long, object> mem;
```

Database, DBObject and Record are direct subclasses of Basis, as are helper level 3 classes such as OrderSpec, WindowBound etc.. All properties and subclasses of Basis are immutable and shareable.

The "user-defined data" for the positive part of mem is defined for the following subclasses of Basis:

Basis subclass	User-defined data
Database, Transaction	DBObjects
Role	ObInfos
Table	TableRows
TableRow	Fields
Framing	for Compiled objects

Negative uids are for named properties of the different subclasses of Basis and for predefinied system objects (such as standard types, system tables, and their columns). The same API pattern is used for Basis and all its classes. Each subclass defines a set of properties, which are assigned negative uids during engine initialisation. Basis itself defines the Name property.

Key	Property	Type	Uid
Name	name	string	-50

In the code a key is defined as follows:

```
internal const long Name = --_uid; // string
```

The name property is then accessed by

```
public string name => (string)mem[Name]??"";
```

This defines name as a readonly property of Basis[5].

Basis subclasses typically have just two constructors, one public, the other protected: both derive from the single constructor in the abstract class Basis:

```
protected Basis (BTree<long,object> m) { mem = m; }
```

Each Basis subclass must define a New method with signature `New(BTree<long,object> m)`. Each Basis subclasses (eg XX) defines an operator for "adding" a property value by creating a new instance:

```
public static XX operator+(XX b,(long,object)x)   {
    return b.New(b.mem + x);
}
```

(See sec 3.1.2 for the definition of ATree's + operator). For example, given a basis object b, to change its name to 'xyz', we can write

```
b += (Basis.Name, "xyz");
```

The above coding pattern is used throughout version 7. Some classes define further operators in the same way.

The following sections list other subclasses of Basis. All such must be immutable and shareable.

Key	Property	Type	Subclass	Uid
Val	val	long	UpdateAssignment	-237
Vbl	vbl	long	UpdateAssignment	-238
FDConds	conds	BTree<long,bool>	FDJoinPart	-211
FDIndex	index	long	FDJoinPart	-212

[5] For renamable objects, the role's name for the object is held in an ObInfo structure stored in the Role.

FDRefIndex	rindex	long	FDJoinPart	-213
FDRefTable	rtable	long	FDJoinPart	-214
FDTable	table	long	FDJoinPart	-215
Reverse	reverse	bool	FDJoinPart	-216
Items	items	CList<long>	OrderSpec	-217
_Generation	generation	Generation	GenerationRule	-273
GenExp	exp	SqlValue	GenerationRule	-274
GenString	gfs	string	GenerationRule	-275

3.5.2 Database

Database is a subclass of Basis. The database class manages all of the database objects and all transactional proposals for updating them. The base class (Database) is used to share databases with a number of connections. Like other level 3 objects, Database is immutable and shareable, so that Reader, Writer and Transaction all have their own version of the database they are working on. The current committed state of each database can be obtained from the Database.databases list, and the current state of the transaction logs can be obtained from the Database.databases list. Both of these structures are protected, and accessed using locking in just 2 or 3 places in the code.

The subclasses of Database are described in this section, and are as follows

Class	Base Class	Description
Database	Basis	
Transaction	Database	

The level 3 Database structure maintains the following data:

- Its name, usually just the name of the database file (not including the extension), but see above in this section
- The current position (loadpos) in the associated database file (level 1): at any time this is where the next Commit operation will place a new transaction.
- The id of the user who owns the database
- The list of database objects defined for this database.

All committed DBObjects accessible from the Database class have a defining position given by a fixed position in the transaction log. In v7, the Transaction subclass additionally allows access to its thread-local uncommitted objects, where the defining position is in the range for uncommitted objects, above 2^{62}, and this is derived from the lexical position in all SQL read from the client since the start of the transaction.[6]

In v7.0, many DBObject subclasses are for Query and SqlValue objects that do not correspond to physical records, but have been constructed by the Parser. For the Database class this happens with ViewDefinitions and in stored procedures. In such cases the physical records contain source strings for the definitions, and parsing occurs once only for each definition. The defining position of the Query and SqlValue objects is given by the position of the first lexical token in the definition string, and so is still a fixed position in the transaction log. To achieve this, all non-protected SqlValue constructors are passed the current Lexer.

The properties defined by the Database class are as follows:

Key	Property	Type	Comments	Uid
Cascade	cascade	bool	only for Transaction	-227
Curated	curated	long	Point at which the database was archived	-53
_ExecuteStatis	executeStatus	ExecuteStatus	Parse/Obey/Drop.Rename	-54
Format	format	int	50 for Pyrrho v5-6, 51 for v7	-392
Guest	guestRole	Role		-55
Levels	levels	BTree<Level,long>	classification	-56

[6] Transactions running in different threads cannot see each other's data, so concurrent transactions will use the same range of defining positions.

LevelUids	cache	BTree<long,Level>		-57
NextPos	nextPos	long[7]	Proposed new Physicals	-395
NextPrep	nextPrep	long	Connection uids, dynamic	-394
NextStmt	nextStmt	long	Uncommitted compiled code	-393
NextId	nextId	long	Offset of next transaction step	-58
Owner	owner	long		-59
Roles	roles	BTree<string,long>		-60
SchemaKey	schemakey	long	for POCO	-286
Types	types	BTree<Domain,Domain>	nameless types	-61
User	uswe	long		-277

The main functionality provided by the Database class is as follows:

Name	Description
Install	Install a Physical record in the level 3 structures
Load	Load a database from the database file
databases	A static list of Databases by name
dbfiles	A static list of FileStreams by database name
loadpos	The current position in the file

3.5.3 Transaction

Transaction is implemented as a subclass of Database. It is immutable to facilitate condition handling, but its collection of Physical records, prepared for a possible commit, means that it is not shareable..

The Transaction maintains the following additional data:

- The current role and user.
- For checking Drop and Rename, a reference to a DBObject that is affected.
- A list of the physical data items prepared for Commit
- Information for communication with the client, described next

If a client request results in an exception (other than a syntax error) the transaction will be aborted. Otherwise, following each round-trip from the client, the transaction gives private access to its modified version of the database together with some additional items available for further communication with the client before any further execution requests are made:

- A rowset resulting from ExecuteQuery (RowSet is immutable)
- A list of affected items resulting from ExecuteNonQuery, including versioning information
- A set of warnings (possibly empty) generated during the transaction step
- Diagnostic details accessible using the SQL standard GET DIAGNOSTICS syntax

An ExecuteQuery transaction step isvolves at least two sever round trips. In the first, the RowSet is constructed by the Parser using an ad hoc Query and Domain, and then further round trips progressively compute and return batches of rows from the resulting RowSet.

The database server also has a private long called nextTid that will be used to start the next step of the transaction: this is a number used for generating unique uids within the transaction. At the start of each transaction the generator for this number (tid) is initialised to 2^{62}, and the server increments this for each transaction step by the length of the input string (used as described in section 3.5.2 to provide defining positions when parsing). This ensures that each uncommitted object referred to in the transaction has a unique defining position. During the writing of physical records during Commit, all of the corresponding DBObjects are reconstructed and reparsed so that following commjit the resulting Database has only the transaction-log-based defining positions described in 3.5.2. This mechanism ensures that tids do not accumulate from one transaction to the next.

The very strong form of transaction isolation used in Pyrrho means that no transaction can ever see uncommitted objects in another transaction. Similarly although each role can have its own domain for objects granted to it, the role cannot see the domain for another role although the defpos is the same.

[7] Many of these properties have type long: such things are uids, and identify DBObjects, SqlValues, Queries, Executables etc. Comments in the source code generally indicate what type of objects is indicated.

The properties defined by the Transaction class are as follows:

Key	Property	Type	Comments	Uid
AutoCommit	autoCommit	bool		-278
Deferred	deferred	BList<TriggerActivation>		-279
Diagnostics	diagnostics	BTree<Sqlx.TypedValue>	For GET DIAGNOSTICS	-280
_Mark	mark	Transaction	For UNDO	-281
Physicals	physicals	BTree<long,Physical>	For Commit	-282
ReadConstraint	readConstraint	BTree<long,ReadConstraint>		-283
Step	step	long		-276
StartTime	startTime	long		-287
TriggeredAction	triggeredAction	long	role boundary	-288
Warnings	warnings	BList<Exception>		-289

3.5.4 Role

In v7, each Role manages ObInfo, the role-based information about DBObjects. In Pyrrho, most objects can be renamed on a per-role basis, and accessibility of objects depends on the role. The Role provides a way of looking up objects by name (such as tables, role, types).

However, query processing and procedure execution works at the compiled level, where the definier's permissions are already built in. So SqlValues and Queries already have the right names and rowTypes.

Key	Property	Type	Comments	Uid
DBObjects	dbobjects	BTree<string,long>	domains, tables, views, triggers, (not TableColumns)	-248
	infos	BTree<long,ObInfo>	stored in mem	
Procedures	procedures	BTree<string, BTree<int,long>>	name and arity	-249

User is a subclass of Role:

Key	Property	Type	Comments	Uid
Password	pwd	string	hidden	-303
InitialRole	initalRole	long		-304
Clearance	clearance	Level		-305

Before the database log is accessed, the static _system environment for the database is set up. It becomes common to all databases and contains predefined domains and system tables. If a database uses a predefined domain it acquires a new version of the domain, with a defining position in the database log. However, access to system tables is normally restricted to the database owner.

Before the engine starts to load an existing database, the schemaRole and Owner are given default values of -61 $Schema and -63 the engine account[8]. There is also a -57 $Guest role which is empty. If roles and users have not been defined the schemaRole continues to operate the database, but access to the database is limited to the account that started up the server (the engine account). PTransaction markers are not placed in the transaction log until a role and user have been loaded from the database log.

Normally the schemaRole is defined first and then the owner role is defined and given privileges over it. The first role to be defined becomes the SchemaRole (so that the default schema role is forgotten) and inherits all of the system objects. (In previous versions of Pyrrho the schemaRole always had position 5 and had the same name as the database file.) The first user to be defined becomes the Owner of the database, with privileges of Usage and AdminRole on the schemaRole. Access to the database is henceforth determined by these new arrangements, as subsequently modified by grant and revoke.

From this point the Reader uses each PTransaction record to set the defining role and user for installing the details from the log.

[8] Such negative numbers are assigned more or less arbitrarily in the source code for properties in the lists shown above and elsewhere in Ch 3. For example, the current value of Database.Schema is -61.

3.5.5 DBObject

Many Physical records define database objects (e.g. tables, columns, domains, user-defined types etc). For convenience, there is a base class that gathers together three important aspects common to all database objects: (a) a definer and defining position, (b) the classification (for mandatory access control) (c) dependency relationships between objects created during parsing, and (d) the depth of such dependency. We explain these aspects briefly later in this section, but the main discussion of these topics must wait for a later chapter (see Sections 8 and 5 of this document).

Roles can be granted access to many DBObject types, including roles, tables (excluding system tables), views, columns, fields, procedures, methods, domains and user-defined types, so that in v7 the Role object maintains a list (infos) that gives access privileges. The effective row-type of a Table depends on which columns have been granted to the role. In Pyrrho this facility was extended to allow role-based metadata and names[9], so that in v7 the Role becomes responsible for all name lookups for level 3 objects. Level 4 objects contains names directly.

If a DBObject is being renamed or dropped in a role, some action needs to be taken in all the role-based catalogues structures that refer to this object.

Defining positions of objects are all 64-bit longs and have several ranges as described in sec 2.3. Defining positions are allocated by the Reader, and Transaction objects are relocated on Commit.

Basis properties for DBObject are as follows

Key	Property	Type	Comments	
_Alias	alias	string		-62
Classification	classification	Level		-63
CompareContext	compareContent	Context	for user-defined types	-250
Definer	definer	long	Role	-64
Dependents	dependents	BTree<long,bool>	Objects that must be serialised before the current one	-65
Depth	depth	int	Max depth of Dependents (>=1)	-66
Description	desc	string		-67
_Domain	domain	Domain	Not for Domains	-176
Framing	framing	BTree<long,DBObject>	For compiled objects	-167
LastChange	lastChange	long		-68
Sensitive	sensitive	bool		-69

DBObject has the following subclasses documented in later sections below:

Class	Base Class	Description
Domain	DBObject	Level 3 domain information: checks, defaults etc
Query	DBObject	
Table	DBObject	Level 3 table information: columns, rows, indexes
Index	DBObject	Entity, references and unique constraints
Procedure	DBObject	Procedure/function parameters and execution
Role	DBObject	Level 3 roles have privileges and DBObject lists
Check	DBObject	
Trigger	DBObject	
UDType	Domain	
Method	Procedure	
PeriodDef	DBObject	
Query	DBObject	
RowSet	DBObject	
Executable	DBObject	

The following subclasses define further key uids:

[9] One of the examples in the Pyrrho manual sees a role adding a generated column to a table. This is an extreme case of table metadata!

Key	Property	Type	Subclass	Uid
GroupKind	kind	Sqlx	Grouping	-232
Groups	groups	BList<Grouping>	Grouping	-233
Members	members	BList<long>	Grouping	-234
DistinctGp	distinct	bool	GroupSpecification	-235
Sets	sets	BList<long>	GroupSpecification	-236
Current	current	bool	WindowBound	-218
Distance	distance	TypedValue	WindowBound	-219
Preceding	preceding	bool	WindowBound	-220
Unbounded	unbounded	bool	WindowBound	-221
Exclude	exclude	Sqlx	WindowSpecification	-222
High	high	WindowBound	WindowSpecification	-223
Low	low	WindowBound	WindowSpecification	-224
Order	order	OrderSpec	WindowSpecification	-225
OrderWindow	orderWindow	string	WindowSpecification	-226
OrdType	ordType	Domain	WindowSpecification	-227
Partition	partition	int	WindowSpecification	-228
PartitionType	partitionType	Domain	WindowSpecification	-229
Units	units	Splx	WindowSpecification	-230
WQuery	query	Query	WindowSpecification	-231
Checks	checks	BTree<long,Check>	TableColumn	-268
Generated	generated	GenerationRule (see below)	TableColumn	-269
Table	tabledefpos	long	TableColumn	-270
UpdateAssignments	update	BList<UpdateAssignment>	TableColumn	-271
UpdateString	updateString	string	TableColumn	-272
Prev	prev	TableColumn	ColumnPath	-321
StartCol	startCol	long	PeriodDef	-387
EndCol	endCol	long	PeriodDef	-388
ParamMode	paramMode	Sqlx	ParamInfo	-98
Result	result	Sqlx	ParamInfo	-99

3.5.6 ObInfo

Role-based information about an object (identified by uid) includes its name, security information and other metadata. The ordering of columns is also a type of metadata, called the rowType as in query processing. All database objects apart from SqlValues have associated ObInfo. (SqlValues always target a single Role, and so directly contain name and column information as appropriate.)

Key	Property	Type	Comment	Uid
Columns	columns	Clist<long>		-251
Methods	methods	BTree<string,BTree<int,long>>	lookup by name and arity	-252
Privilege	priv	Grant.Privilege		-253
Properties	properties	BTree<string,long>	see below	-254

Metadata properties have case-sensitive string names and types as follows:

Key	Property	Type	Comment
Attribute	Attribute	BTree<string,bool>	Columns can be flagged as attributes for Xml output. BTree placed somewhere in mem
Caption	Caption	string	string placed somewhere in mem
Csv	Csv	bool	
_Description	Description	string	Role-based, not necessarily the same as DBObject.description. string placed somewhere in mem
Entity	Entity	bool	
Histogram	Histogram	bool	
Id	Id	long	
Iri	Iri	string	string placed somewhere in mem
Json	Json	bool	
Legend	Legend	string	string placed somewhere in mem
Line	Line	bool	

Points	Points	bool	
Pie	Pie	bool	
X	X	int	column for X axis
Y	Y	int	column for Y axis

3.5.7 Domain

In v7, the Common level SqlDataType has been removed, and instead there is much greater use internally of the Domain and ObInfo classes. Scalar values are described by a Domain, and the representation information allows rowTypes and other structures to be organised.

The properties of Domain (see sec 3.2.5) are as follows:

Key	Property	Type	Comments	Uid
Abbreviation	abbrev	string		-70
Charset	charSet	CharSet	default is UCS	-71
Constraints	constraints	BTree<long,bool>	Evaluate to bool	-72
Culture	culture	CultureInfo	default InvariantCulture	-73
Default	defaultValue	TypedValue		-74
DefaultString	defaultString	string		-75
Descending	AscDesc	Sqlx	ASC or DESC	-76
Display	display	int	default is rowType.Length	-177
Element	elType	Domain	For derived types	-77
End	end	Sqlx	DAY etc	-78
Iri	iri	string		-89
NotNull	notNull	bool		-81
NullsFirst	Nulls	bool	Affects ordering	-82
OrderCategory	orderCatgory	OrderCategory		-83
OrderFunc	orderFunc	long	DBObjects	-84
Precision	prec	int		-85
Provenance	provenance	string	For imports	-86
Representation	representation	BTree<long,Domain>	→obs For anything with columns (even TRow)	-87
RowType	rowType	CList<long>	SqlValue or TableColumn	-187
Scale	scale	int		-88
Start	start	Sqlx (YEAR etc)		-89
Structure	structdef	BList<long>	DBObjects	-391
Under	super	long	Domains	-90
UnionOf	unionOf	BList<long>	Domains	-91

3.5.8 Table

The name of the table and its columns are role-specific, and are retrieved for a From instance (From is a subclass of Query). This slightly indirect mechanism works well since tables can occur multiple times in a query, and the From instances distinguish between them.

Key	Property	Type	Comments	Uid
ApplicationPS	appPS	long	PeriodSpecification	-262
Enforcement	enforcement	Grant.Privilege	For mandatory access control	-263
Indexes	indexes	BTree<CList<long>, long>	Key cols, Index	-264
SystemPS	sysPS	long	PeriodSpecification	-265
TableChecks	tableChecks	BTree<long,bool>	Checks	-266
TableCols	tblCols	BTree<long,bool>	TableColumns	-332
TableRows	tableRows	BTree<long,TableRow>	see note below	-181
Triggers	ptriggers	BTree<PTrigger.TrigType, BTree<long,bool>>	Triggers	-267

SystemTable has an additional property:

Key	Property	Type		Uid
Cols	tableCols	BTree<long,TableColumn>		-175

TableRow is immutable but is not a Basis subclass. It has some similarities to TRow but is role-independent and therefore has no Domain.

3.5.9 Query

From v7, Query is a subclass of DBObject, and it is immutable and shareable to facilitate being a component of SqlValues and Executables stored in the database. The nature of DBObject is inherited by Tables and Views.

Key	Property	Type	Comments	Uid
_Aggregates	aggregates	bool		-191
Assig	assig	BTree<UpdateAssignment,bool>	For update	-174
FetchFirst	fetchFirst	int		-179
Filter	filter	BTree<long,TypedValue>		-180
Matches	matches	BTree<long,TypedValue>		-182
Matching	matching	BTree<long, BTree<long,bool>>		-183
OrdSpec	ordSpec	OrderSpec		-184
Periods	periods	BTree<long,PeriodSpecl>		-185
_Repl	replace	BTree<string,string>		-186
SimpleQuery	simpleQuery	From		-189
Where	where	BTree<long,bool>	SqlValue	-190

Query subclasses define some additional properties:

Key	Property	Type	Subclass	Uid
RVQSpecs	rVqSpecs	BList<long>	CursorSpecification	-192
RestGroups	restGroups	BTree<string,int>	CursorSpecification	-193
RestViews	restViews	BTree<long,bool>	CursorSpecification	-194
_Source	_source	string	CursorSpecification	-195
Union	union	long	CursorSpecification	-196
UsingFrom	usingFrom	long	CursorSpecification	-197
Assigns	assigns	BList<UpdateAssignment>	From	-150
Source	source	long	From	-151
Static		long	From	-152
_Target	target	long	From	-153
_FDInfo	FDInfo	FDJoinPart	JoinPart	-202
JoinCond	joinCond	BTree<long,bool>	JoinPart	-203
JoinKind	kind	Sqlx	JoinPart	-204
LeftOrder	leftOrder	OrderSpec	JoinPart	-205
LeftOperand	leftOperand	long	JoinPart	-206
NamedCols	namedCols	BList<long>	JoinPart	-207
Natural	natural	Sqlx	JoinPart	-208
RightOrder	rightOrder	OrderSpec	JoinPart	-209
RightOperand	rightOperand	long	JoinPart	-210
Distinct	distinct	bool	QueryExpression	-243
_Left	left	long	QueryExpression	-244
Op	op	Sqlx	QueryExpression	-245
_Right	right	long	QueryExpression	-246
SimpleTableQuery	simpleTableQuery	bool	QueryExpression	-247
Distinct	distinct	bool	QuerySpecification	-239
RVJoinType	rvJoinType	Domain	QuerySpecification	-240
Scope	scope	BTree<long,Domain>	QuerySpecification	-241
TableExp	tableExp	TableExpression	QuerySpecification	-242
_Table	table	From	SqlInsert	-154
Provenance	provenance	string	SqlInsert	-155
Value	value	SqlValue	SqlInsert	-156
From	from	long	TableExpression	-198
Group	group	long	TableExpression	-199
Having	having	BTree<long,bool>	TableExpression	-200
Needed	needed	BTree<long,bool>	TableExpression	-178

Windows	windows	BTree<long,bool>	TableExpression	-201

3.5.10 Index

Indexes are constructed when required to implement integrity constraints.

Key	Property	Type	Comments	Uid
Adapter	adapter	Procedure	Given an index key returns an index key	-157
IndexConstraint	flags	ConstraintType	primary, foreign , unique etc	-158
Keys	keys	CList<long>	The key columns	-159
References	references	BTree<long, BList< TypedValue>>	A list of keys formed by the adapter if any	-160
RefIndex	refindexdefpos	long	The index referred to by a foreign key	-161
RefTable	reftabledefpos	long	The table referred to by a foreign key	-162
TableDefPos	tabledefpos	long	The table whose rows are indexed	-163
Tree	rows	MTree	The multi-level index key->long	-164

3.5.11 SqlValue

From v7, SqlValue is a subclass of DBObject, whose defpos is assigned by the transaction or reader position, or during compilation.

Key	Property	Type	Comments	Uid
_Columns	columns	CList<long>	SqlValue	-299
_From	from	long	Directs to a Query	-306
Left	left	long	DBObject or SqlValue	-308
_Meta	meta	MetaData		-307
Right	right	long	SqlValue	-309
Sub	sub	long	SqlValue	-310

SqlValue has the following virtual Methods:

Method	Returns	Signatures	Description
RowSet		From	Installs a computed rowset for VALUES, subquery etc. The base implementation is for a singleton.
StartCounter		Query	Precedes an aggregation calculation
AddIn		Query	For the aggregation calculation
Compare	Int	Context	A base for join conditions
Constrain		Context, Domain	Used during query analysis
Check		Context, List of groups	For checking grouping constructs during query analysis
_Setup		Domain	Compute nominal types during query analysis
Conditions		Context	For computing joins etc in query analysis. Also used for searched case statements
MatchExpr		SqlValue	Compare SqlValues for structural equality
IsFrom		Query	For analysing joins
IsConstant			For optimising filters
JoinConidition	SqlValue	JoinPart	Gets for condition for a join
DistributeConditions		Query	For optimising filters in query analysis
WhereEquals	bool	List of exprs	Collecting AND wheres
HasDisjunction	Bool	SqlValue	Detecting ORs
LVal	Target	Context	Used for computing the left side of an assignment, e.g. subscript

Invert	SqlValue		For optimising predicates
Lookup	SqlValue	Ident	Field selector for Rvalue
FindType	Domain	Domain	Helper for nominal type analysis in exprs
Eval	TypedValue	Context	

The base SqlValue class has methods that can be used in aggregations (count, sum etc).

There are over 30 subclasses of SqlValue some of which provide machinery specific to particular syntactic constructs that can occur inside the SELECT clause (e.g. SqlMultiset, SqlValueArray, SqlDocArray).

SqlValue subclasses have the following additional properties:

Key	Property	Type	Subclass	Uid
Bits	bits	BList<long>	ColumnFunction	-333
Escape	escape	long	LikePredicate	-358
_Like	like	bool	LikePredicate	-359
Found	found	bool	MemberPredicate	-360
Lhs	lhs	long	MemberPredicate	-361
Rhs	rhs	long	MemberPredicate	-362
NIsNull	isnull	bool	NullPredicate	-364
NVal	val	long	NullPredicate	-365
All	all	bool	QuantifiedPredicate etc	-348
Between	between	bool	QuantifiedPredicate etc	-349
Found	found	bool	QuantifiedPredicate etc	-350
High	high	long	QuantifiedPredicate etc	-351
Low	low	long	QuantifiedPredicate etc	-352
Op	op	Sqlx	QuantifiedPredicate etc	-353
Select	select	long	QuantifiedPredicate etc	-354
Vals	vals	BList<long>	QuantifiedPredicate etc	-355
What	what	long	QuantifiedPredicate etc	-356
Where	where	long	QuantifiedPredicate etc	-357
QExpr	expr	long	QueryPredicate	-363
Call	call	long	SqlCall	-335
TableCol	tableCol	long	SqlTableCol	-322
CopyFrom	copyFrom	long	SqlCopy	-284
Sce	sce	SqlRow	SqlConstructor DqlDefaultConstructor	-336
Udt	ut	Domain	SqlConstructor SqlDefaultConstructor	-337
Spec	spec	long	SqlCursor	-334
Field	field	string	SqlField	-314
Filter	filter	long	SqlFunction	-338
Mod	mod	Sqlx	SqlFunction	-340
Monotonic	monotonic	bool	SqlFunction	-341
Op1	op1	long	SqlFunction	-342
Op2	op2	long	SqlFunction	-343
Query	query	long	SqlFunction	-344
_Val	val	long	SqlFunction	-345
Window	window	long	SqlFunction	-346
WindowId	windowId	long	SqlFunction	-347
KeyType	keyType	ObInfo	SqlHttp	-370
Mime	mime	string	SqlHttp	-371
Pre	pre	TRow	SqlHttp	-372
RemoteCols	remoteCols	string	SqlHttp	-373
TargetType	targetType	ObInfo	SqlHttp	-374
Url	url	SqlValue	SqlHttp	-375
GlobalFrom	globalFrom	From	SqlHttpBase	-255
HttpWhere	where	BTree<long,SqlValue>	SqlHttpBase	-256
HttpMatches	matches	BTree<SqlValue,TypedValue>	SqlHttpBase	-257

HttpRows	rows	RowSet	SqlHttpBase	-258
UsingCols	usC	BTree<string,long>	SqlHttpUsing	-259
UsingTablePos	usingtablepos	long	SqlHttpUsing	-260
_Val	val	TypedValue	SqlLiteral	-317
TransitionRowSet	trs	long	SqlOldTable, SqlOldRowCol	-318
Rows	rows	BList<long>	SqlRowArray	-319
ArrayValuedQE	aqe	long	SqlSelectArray	-327
TableRow	rec	TableRow	SqlTableRowStart	-311
TreatExpr	val	long	SqlTreatExpr	-313
TreatType	type	long	SqlTypeExpr	-312
Array	array	BList<long>	SqlValueArray	-328
Svs	svs	long	SqlValueArray	-329
Modifier	mod	Sqlx	SqlValueExpr	-316
Expr	expr	long	SqlValueSelect	-330
Source	source	string	SqlValueSelect	-331
Attrs	attrs	BTree<int, (XmlName,SqlValue)>	SqlXmlValue	-323
Children	children	BList<long>	SqlXmlValue	-324
Content	content	long	SqlXmlValue	-325
Element	element	XmlName	SqlXmlValue	-326

3.5.12 Check

Key	Property	Type	Comments	Uid
Condition	condition	SqlValue		-51
Source	source	string		-52

3.5.13 Procedure

Functions are procedures with a return type. The _Domain for the Procedure gives the return type.

Key	Property	Type	Comments	Uid
Body	body	long	Executable	-168
Clause	clause	string		-169
Inverse	inverse	long	For transformations	-170
Monotonic	monotonic	bool	For adapters	-171
Params	ins	Blist<long>	ParamInfo	-172

3.5.14 Method

Method is a subclass of Procedure. _Domain gives the return type. The owning UDT is TypeDef.

Key	Property	Type	Comments	Uid
MethodType	methodType	MethodType	Instance, Overriding, Static, Constructor	-165
TypeDef	udType	Domain	The user-defined type for this method	-166

3.5.15 Trigger

Key	Property	Type	Comments	Uid
Action	action	WhenPart		-290
NewRow	newRow	long	if specified	-293
NewTable	newTable	long	if specified	-294
OldRow	oldRow	long	if specified	-295
OldTable	oldTable	long	if specified	-296
TrigType	tgType	TrigType	insert/update before/after etc	-297
UpdateCols	updateCols	BList<long>	if specified	-298

3.5.16 Executable

From v7 Executable is a subclass of DBObject. It has dozens of subclasses as detailed below.

Many Executables can provide a result value, which is placed in the Context on execution, using Context's row or ret field depending on whether the result type is described by a Selection or a Domain. The desired result of an Executable is therefore provided to the runtime system as a (Domain,Selection) to cover these cases.

Key	Property	Type	Class	Uid
Label	label	string	Executable	-92
Stmt	stmt	string	Executable	-93
_Type	type	Executable.Type	Executable	-94
_Table	from	long	SqlInsert	-154
Provenance	provenance	string	SqlInsert	-155
Value	value	long	SqlInsert	-156
Val	val	long	AssignmentStatement	-105
Vbl	vbl	long	AssignmentStatement	-106
Parms	parms	BList<long>	CallStatement	-133
Proc	proc	long	CallStatement	-134
Var	var	long	CallStatement	-135
Stmts	stmts	BList<long>	CompoundStatement	-96
CS	cs	long	CursorDeclaration	-129
Cursor	cursor	long	FetchStatement	-129
How	how	Sqlx	FetchStatement	-130
Outs	outs	Blist<long>	FetchStatement	-131
Where	where	long	FetchStatement	-132
Cursor	cursor	string	ForSelectStatement	-124
ForVn	forvn	string	ForSelectStatement	-125
Loop	loop	long	ForSelectStatement	-126
Sel	sel	long	ForSelectStatement	-127
Stms	stms	BList<long>	ForSelectStatement	-128
List	list	BTree<long,Sqlx>	GetDiagnostics	-141
HDefiner	hdefiner	Activation	Handler	-103
Hdlr	hdlr	HandlerStatement	Handler	-104
HType	htype	Sqlx	HandlerStatement	-102
Conds	conds	BList<string>	HandlerStatement	-101
Action	action	long	HandlerStatement	-100
CredPw	pw	long	HttpREST	-143
CredUs	us	long	HttpREST	-144
Mime	mime	string	HttpREST	-145
Posted	data	long	HttpREST	-146
Url	url	long	HttpREST	-147
Verb	verb	string	HttpREST	-148
Where	wh	long	HttpREST	-149
Else	els	BList<long>	IfThenElse	-116
Elsif	elsif	BList<long>	IfThenElse	-117
Search	search	long	IfThenElse	-118
Then	then	BList<long>	IfThenElse	-119
Init	init	long	LocalVariableDec	-97
LhsType	lhsType	Domain	MultipleAssignment	-107
List	list	BList<long>	MultipleAssignment	-108
Rhs	rhs	long	MultipleAssignment	-109
Ret	ret	long	ReturnStatement	-110
CS	cs	long	SelectStatement	-95
Outs	outs	BList<long>	SelectSingle	-142
Else	els	BList<long>	SimpleCaseStatement, SearchedCaseStatement	-111
Operand	operand	long	SimpleCaseStatement	-112
Whens	whens	BList<long>	SimpleCaseStatement, SearchedCaseStatement	-113
Exception	exception	Execption	Signal	-136

Objects	objects	BList<object>	Signal	-137
_Signal	signal	string	Signal	-138
SetList	setlist	BTree<Sqlx,long>	Signal	-139
SType	stype	Sqlx	Signal	-140
Cond	cond	long	WhenPart	-114
Stms	stms	BList<long>	WhenPart, LoopStatement	-115
Loop	loop	long	WhileStatement, RepeatStatement, LoopStatement	-121
Search	search	long	WhileStatement, RepeatStatement	-122
What	what	BList<long>	WhileStatement	-123
Nsps	nsps	BTree<string,string>	XmlNameSpaces	-120
QMarks	qMarks	CList<long>	PreparedStatement	-396
Target	target	Executable	PreparedStatement	-397

3.5.17 View

Key	Property	Type	Comments	Uid
RemoteGroups	remoteGroups	GroupSpecification		-378
ViewDef	viewdef	string		-379
ViewQuery	viewQry	QueryExpression		-380

RestView is a subclass of View

Key	Property	Type	Comments	Uid
ClientName	nm	string	Deprecated	-381
ClientPassword	pw	string	Deprecated	-382
RemoteCols	joinCols	BTree<string,int>		-383
RemoteAggregates	remoteAggregates	bool		-384
UsingTablePos	usingTable	long		-385
ViewStructPos	viewStruct	long		-386

3.5.17 RowSet

RowSets are DBObjects that deliver the result of query processing. RowSets are immutable and shareable, and constructed in a Context when requested. There is an evaluation pipeline for rowsets, starting with the base tables, applying sorting and joins, aggregation, merging and selection etc, according to a strategy determined during parsing.

Some rowsets operate directly on base tables: TableRowSet, IndexRowSet or supplied values (TrivialRowSet, ExplicitRowSet). TransitionRowSets (or insert, update and delete) operate directly on base tables but allow for manipulation of column values by triggers.

Other rowset types have one or more source rowsets, traversed before or during traversal of the result. As far as possible, traversal of the resulting rowset proceeds recursively: a request for a row of a rowset recursively requests a row of the source from which it can be computed. This approach is worthwhile because it is very likely that not all rows will be traversed. JoinRowSets and MergeRowSets use possibly sorted rowset operands, which are built before traversal, but the columns are simple to compute. Aggregation and ordering require the evaluation pipeline to be broken up with Trivial or ExplicitRowSets constructed for intermediate results. Subqueries and window functions also require the construction of auxiliary source rowsets, and where sources are table-valued procedures or views, ExportedRowSets are constructed. In such cases, the Context keeps track of which auxiliary cursors should be used instead of re-evaluating the SqlValue itself.

Evaluation may take place in several stages owing to stored procedures or subqueries in the select list or source list. In some cases, the Selection is simply attached from the source Selection, that is, columns are evaluated from source columns (SelectedRowSet). In other cases, the Selection is into an auxiliary rowset of intermediate results as described above. Complex ordering or aggregation conditions can require the construction of auxiliary selected rowsets in lower levels of the pipeline.

A useful structure in the implementation is Finder. Formally this is a pair RowSet,Column, and associations (uid,Finder) identify the location of the current value of a base column by uid in the current set of cursors.

In all cases, auxiliary rowsets have their own rowType and keyType since their values are no longer taken from their source rowsets.

The abstract interface offered is as follows:

Cursor First()	Compute the first row of the rowSet
CList<long>| keyType	The keyType of the RowSet
int limit	
BTree<long,TypedValue> matches	A set of filters
Cursor PositionAt(PRow key)	Compute the row of the rowSet for the given key position
Query qry	The Query this rowSet delivers results for.
OrderSpec rowOrder	The specified ordering for the row set
CList<long> rowType	The nominalDataType of the rowSet is normally that of the query, but can differ for select-single subqueries and for RESTViews
RowSet(Database,Context,Query, *Selection,OrderSpec*)	Constructor. *The last two parameters are optional*
int skip	
BTree<long,SqlValue> where	A where condition on the rowset columns

There are numerous subclasses of RowSet. The columns in the table below indicate a subclass name, whether the rowset must be the "top level" result, has a new uid, what the source rowsets are and whether the cursors proceed one-to-one (source rowsets can in general come from subqueries and sorted rowsets in addition to base tables), whether new rows may be constructed, and whether new columns may be constructed. Each RowSet subclass has one or more associated Cursor subclass with a similar name. Each Cursor subclass has its own implementations of First and Next.

SubClass	Top	Uid	Sce	Row	Cols
DistinctRowSet		Y	N	N	N
DocArrayRowSet		Y	N	N	N
EmptyRowSet		Y	N	N	N
EvalRowSet		N	Y	Y	Y
ExplicitRowSet		Y	N	Y	N
ExportedRowSet		Y	Y	N	N
GroupingRowSet		Y	GB	Y	N
IndexRowSet		N	N	N	N
JoinRowSet		N	LR	N	Y
MergeRowSet		N	LR	N	N
OldTableRowSet		Y	Y	N	Y
OrderedRowSet		Y	Y	N	N
RoutineCallRowSet		Y	N	Y	N
RowSetSection	Y	Y	Y	N	N
SelectedRowSet		N	121	N	N
SqlRowSet		N	N	N	N
SystemRowSet		N	N	Y	N
TableRowSet		N	N	N	N
TransitionRowSet	Y	Y	121	N	N
TrivialRowSet		Y	121	N	N
WindowRowSet		Y	N	N	N

As with other subclasses of Basis, properties of these immutable classes have uids that allow them to be stored in the BTree<long,object> mem structure inherited from Basis. Many of these properties were first defined for Queries and other parsed entities, so many of the entries below are defined in earlier sections of this manual. For better readability and convenience, their names and descriptions are repeated here.

Name	Type	Definition	Uid

_tgt	PTrigger.TrigType	TransitionRowSet.TriggerType	-421
_trs	TransitionRowSet	TransitionTableRowSet.Trs	-431
actuals	BList<long>	RoutineCallRowSet.Actuals	-435
adapters	Adapters	TransitionRowSet._Adapters	-429
built	bool	RowSet.Built	-402
data	BTree<long,TableRow>	TransitionTableRowSet.Data	-432
defaults	BTree<long,TypedValue>	TransitionRowSet.Defaults	-415
distinct	bool	QuerySpecification.Distinct	-239
docs	BList<SqlValue>	DocArrayRowSet.Docs	-440
domain	Domain	DBObject._Domain	-176
explRows	BList<(long,TRow)>	ExplicitRowSet.ExplRows	-414
filter	PRow	FilterRowSet.IxFilter	-411
finder	BTree<long,Finder>	RowSet._Finder	-403
first	long	JoinRowSet.Jfirst	-447
from	From	TransitionRowSet,TrsFrom	-416
groupings	BList<long>	GroupingRowSet.Groupings	-406
groupSpec	GroupSpecification	TableExpression.Group	-199
having	BTree<long,bool>	TableExpression.Having	-200
index	long	IndexRowSet._Index	-410
indexdefpos	long	TransitionRowSet.IxDefPos	-420
join	JoinPart	JoinRowSet._Join	-446
keys	CList<long>	Index.Keys	-159
map	BTree<long,Finder>	TransitionTableRowSet.Map	-433
matches	BTree<long,TypedValue>	Query._Matches	-182
matching	BTree<long,BTree<long,bool>>	Query.Matching	-183
mtree	MTree	Index.Tree	-164
needed	BTree<long,Finder>	RowSet._Needed	-401
offset	int	RowSetSection.Offset	-438
proc	Procedure	RoutineCallRowSet.Proc	-436
ra	TriggerContext	TransitionRowSet.Ra	-424
rb	TriggerContext	TransitionRowSet.Rb	-422
result	RowSet	RoutineCallRowSet.Result	-437
ri	TriggerContext	TransitionRowSet.Ri	-423
rmap	BTree<long,Finder>	TransitionTableRowSet.RMap	-434
row	TRow	TrivialRowSet.Singleton	-405
rowOrder	CList<long>	RowSet.RowOrder	-404
rows	Blist<TRow>	RowSet._Rows	-407
rt	CList<long>	(domain.rowType)	
second	long	JoinRowSet.Second	-448
size	int	RowSetSection.Size	-439
source	long	From.Source	-151
sqlRows	BList<long>	SqlRowSet.SqlRows	-413
sQMap	BTree<long,Finder>	SelectedRowSet.SQMap	-408
ta	TriggerContext	TransitionRowSet.Ta	-426
table	long	IndexRowSet.IxTable	-409

tabledefpos	long	SqlInsert._Table	-154
targetAc	Activation	TransitionRowSet.TargetAc	-430
targetInfo	ObInfo	TransitionRowSet.TargetInfo	-417
targetTrans	BTree<long,Finder>	TransitionRowSet.TargetTrans	-418
tb	TriggerContext	TransitionRowSet.Tb	-425
td	TriggerContext	TransitionRowSet.Td	-428
transTarget	TriggerContext	TransitionRowSet.TransTarget	-419
tree	RTree	OrderedRowSet._RTree	-412
values	Tmultiset	WindowRowSet.Multi	-441
wf	SqlFunction	WindowRowSet.Window	-442
where	BTree<long,bool>	Query.Where	-190

3.5.18 Framing

Compiled objects have an immutable property called Framing, which holds the results of parsing the object definition. .

Key	Property	Type	Comment	Uid
Data	data	BList<long,RowSet>	RowSet	-450
Defs	defs	Ident.Idents	To enable name matching	-451
Obs	obs	BTree<long,DBObject>	DBObjects other than RowSet	-449
Resullt	result	long	RowSet	-452
Results	results	BTree<long,long>	Query->RowSet	-453

See discussion in 3.4.2.

3.6 Level 4 Data Structures

Level 4 handles transactions, including Parsing, the execution context, activations etc.

3.6.1 Context

A Context contains the main data structures for analysing the meaning of SQL. Activations are the only subclasses of Context. Context and its subclasses are mutable, but all of the other structures mentioned above are immutable.

During parsing and execution of a command, the Context caches database objects it needs, RowSets it is working on, the TypedValues of local variables and open Cursors, and the lookup tables used during parsing. All objects cached are for the context's role, so that the name and column properties of SqlValues and Queries are correct for the current role. Definitions for new objects are parsed with the help of an ad-hoc table of new ObInfos passed into the parsing routines. This architecture means that ObInfos are never required for cached objects.

During evaluation of expressions, Activations are added to the Context stack when procedure or trigger code is executed, and their cache initialised to the current one before the activation's schema objects are cached for the definer's role. This means that all data is passed in, but the schema objects are for the right Role. At the end of an Activation, the caller's local data is copied back into the calling context together with the return values and out parameters It is important that in SQL there is no concept of reference parameters, so, at any time during expression evaluation, only the top activation is accessible. An apparent exception is with a complex expression on the left-hand side of an assignment, such as f(x).y = z; or even f(x)[y]=z, but even with these, expression f(x) and z can be computed before the assignment is completed. Activations provide for complex Condition and Signal handling, similar to the operation of exceptions.

In Pyrrho, we use lazy traversal of rowsets, frequently delivering an entire rowset to the client interface before computing even one row. The client request for the first or next row begins the evaluation of rows. Each new row bookmark computes a list of (defpos,TypedValue) pairs called vals. While sorting, aggregating or DISTINCT result sets often requires computation of intermediate rowsets, many opportunities for deferring traversal remain and Pyrrho takes every opportunity. To assist this process,

Pyrrho uses immutable bookmarks for traversal instead of the more usual iterators. Window functions need the computation of adjacent groups of rows.

Procedural code can reference SqlValues, in complex select list expressions and conditions, in triggers, and in structured programming constructs such as FOR SELECT. Activations can return tables as rowsets: as mentioned above, these are immutable typedvalues.

The data maintained by any kind of Context (for any of the above sorts) is as follows:

- The current transaction snapshot **tr**.
- A set of DBObjects called **obs** consisting of the SqlValues and Queries in the current evaluation. During parsing, there are also (a) a set of definitions called **defs**, which helps with looking up identifier chains, and (b) a structure called **depths**, which organises the set of objects by nesting depth to help with the evolution of queries and sqlValues during parsing analysis.
- A set of RowSets called **data**, one of which is the current **result**, and an association from Queries to RowSets called **results**. The construction of these items completes the compilation process (see sec 3.4.2).
- Volatile lists of TypedValues by uids: Cursors (**cursors**), variables (**values**), a return value (**val**) a top-level Cursor (**rb**) and an association (**from**) between SqlValues and RowSets. This last is copied from a RowSet's **finder** during Cursor evaluation. These things enable SqlValues to be evaluated given the context.

Activations have additional structures to help with parameter passing and exception handling. They are initialised with the above structures from the calling context. When stored procedure code is being prepared for execution, the compiled version is then added to the activation's **objects** set. At the end of the activation, the results are slid down the stack to the calling context, and result and out parameters are updated. (In SQL there is no way to modify non-local data during the activation.)

TriggerActivations give access to the old row and table values for the calling transition rowset. This is a bit more complicated as a trigger-side version of the transition cursor is placed in the trigger activation (whose uids are file positions from the trigger definition's position) instead of the transition's starting context. Both versions of the transition cursor use the same ol row and old table uids, but the the new/current uids are different in the two versions: #0 and table column uids in the calling context, but SelectedRowSet defpos and SqlValue uids in the trigger activation version.

3.6.2 Context Subclasses

Context has the following subclasses, which are described more fully in later chapters:

Class	Base Class	Description
Activation	Context	A context for execution of procs, funcs, and methods: local variables, labels, exceptions etc
CalledActivation	Activation	An activation for a procedure or method call, managing a Variables stack
TriggerActivation	Activation	For trigger execution

3.6.3 Cursor

Previously called RowBookmark, this is an abstract and immutable class for traversing rowSets. All RowSets offer a First() that returns a Cursor at position 0, or null. Cursors are immutable. Unlike bookmarks they are unidirectional.

The Context remembers the current Cursor for each RowSet it defines: others can be remembered using the SqlCursor class. The construction of some rowsets (e.g. grouped and windowed) uses a temporary context.

The interface offered includes the following:

TRow key	*Abstract* The current key
Cursor Next()	*Abstract:* Returns a bookmark for the next row, or returns null if there is none
int _pos	The current position: starts at 0 for First() cursor in a traversal

Cursor PositionAt(pos)	Returns a bookmark for the given position, or null if there is none.
TableRow Rec()	*Abstract* The current table row if defined for this rowset
TRow row	*Abstract* The current row
RowSet _rs	The rowset as it was at the start of the traversal

There are numerous subclasses of Cursor, many of which are local to RowSets.

3.6.4 Activation

Activation is a subclass of Context. It has two subclasses: CalledActivation and TriggerActivation

Key	Property	Type	Comments
Break	brk	long	An Activation to break to
Continue	cont	long	An Activation to continue to
DynLink	dynLink	Activation	
Execptions	exceptions	BTree<string,Handler>	
ExecState	execState	ExecState	saved Transaction and Activation state
Return	ret	TypedValue	
Signal	signal	Signal	
Vars	vars	BTree<string,Variable>	

Activations form a stack, using the next field of Contexts. Local variables are held in the values tree (identified by uid) and the val field of Context holds the return value if any. Many statements are labelled, and these run in new Activation with a matching label.

The break statement allows execution to break out of a loop to a named Activation: local variables and return values if any are slid down the stack.

Signals also cause a change of Context, and the behaviour depends on the kind of Handler defined for that condition.

Thus the loop in a CompoundStatement will check the Context that results from Obeying and Executable. If the context has not changed, the next statement in the CompoundStatement is Obeyed.

Otherwise we break out of the loop.

When an Activation initializes, it starts with values and other information copied from the parent context. A CalledActivation will set up local variables corresponding to parameters (and, for methods, target fields).A TriggerActivation installs a cursor for the current row from the TransitionRowSet, adding cursors for oldRow and oldTable if these are defined.

At the end of the activation (return statement), the SlideDown method copies non-local values to the parent context.

4. Locks, Integrity and Transaction Conflicts

Pyrrho's transaction model uses an optimistic 3-stage Commit protocol, as follows:

- Commit1: For each connected database that has work to commit, we first check any recently committed information from the database file for conflicts with data that this transaction has accessed. Then we lock the DataFile and do this again. Conflicts at this stage may cause the transaction to fail (for example, alteration of a record we have accessed).
- Commit2: For each connected database, flush the physical records to the database (the datafile is still locked). The local transaction structure is now discarded.
- Commit3: For each connected database, get the base Level 3 Database instance to install the physicals that have just been committed to the DataFile, and unlock the datafile.

Pyrrho's optimistic transaction model means that the client is unable to lock any data. The database engine uses DataFile locks internally to ensure correct operation of concurrent transactions.

4.2 Transaction conflicts

This section examines the verification step that occurs during the first stage of Commit. For each physical record P that has been added to the database file since the start of the local transaction T, we

- check for conflict between P and T: conflict occurs if P alters or drops some data that T has accessed, or otherwise makes T impossible to commit
- install P in T.

Let D be the state of the database at the start of T. At the conclusion of Commit1, T has installed all of the P records, following its own physical records P': T=DP'P. But, if T now commits, its physical records P' will follow all the P records in the database file. The database resulting from Commit3 will have all P' installed after all P, ie. D'=DPP'. Part of the job of the verification step in Commit1 is to ensure that these two states are equivalent: see section 4.2.2.

Note that both P and P' are sequences of physical records: $P=p_0p_1\ldots p_n$ etc.

4.2.1 ReadConstraints (level 4)

The verification step goes one stage beyond this requirement, by considering what data T took into account in proposing its changes P'. We do this by considering instead the set P'' of operations that are read constraints C' or proposed physicals P' of T. We now require that DP''P = DPP'' .

The entries in C' are called ReadConstraints (this is a level 4 class), and there is one per base table accessed during T (see section 3.8.1). The ReadConstraint records:

- The local transaction T
- The table concerned
- The constraint: CheckUpdate or its subclasses CheckSpecific, BlockUpdate

CheckUpdate records a list of columns that were accessed in the transaction. CheckSpecific also records a set of specific records that have been accessed in the transaction. If all records have been accessed (explicitly or implicitly by means of aggregation or join), then BlockUpdate is used instead.

ReadConstraints are applied during query processing by code in the From class.

The ReadConstraint will conflict with an update or deletion to a record R in the table concerned if

- the constraint is a BlockUpdate or
- the constraint is a CheckSpecific and R is one of the specific rows listed.

This test is applied by Participant.check(Physical p) which is called from Commit1.

4.2.2 Physical Conflicts

The main job of Participant.check is to call p.Conflict(p) to see of two physical records conflict. The operation is intended to be symmetrical, so in this table the first column is earlier than the second in alphabetical sequence:

Physical	Physical	Conflict if
Alter	Alter	to same column, or rename with same name in same table
Alter	PColumn	rename clashes with new column of same name
Alter	Record	record refers to column being altered
Alter	Update	update refers to column being aletered
Alter	Drop	Alter conflicts with drop of the table or column
Alter	PIndex	column referred to in new primary key
Alter	Grant	grant or revoke for object being renamed
Change	PTable	rename of table or view with new table of same name
Change	PAuthority	rename of authority with new authority of same name
Change	PDomain	rename of domain with new domain of same name
Change	PRole	rename of role with new role of same name
Change	PView	rename of table or view with new view of same name
Change	Change	rename of same object or to same name
Change	Drop	rename of dropped object
Change	PCheck	a check constraint and a rename of the table or domain
Change	PColumn	new column for table being renamed
Change	PMethod	method for type being renamed
Change	PProcedure	rename to same name as new proc/func
Change	PRole	rename to same name as new role
Change	PTable	rename to same name as new table
Change	PTrigger	trigger for renamed table
Change	PType	rename with same name as new type
Change	PView	rename of a view with new view
Delete	Drop	delete from dropped table
Delete	Update	update of deleted record, or referencing deleted record
Delete	Record	insert referencing deleted record
Drop	Drop	drop same object
Drop	Record	insert in dropped table or with value for dropped column
Drop	Update	update in dropped table or with value for dropped column
Drop	PColumn	new column for dropped table
Drop	PIndex	constraint for dropped table or referencing dropped table
Drop	Grant	grant or revoke privileges on dropped object
Drop	PCheck	check constraint for dropped object
Drop	PMethod	method for dropped Type
Drop	Edit	alter domain for dropped domain
Drop	Modify	modify dropped proc/func/method
Drop	PTrigger	new trigger for dropped table
Drop	PType	drop of UNDER for new type
Edit	Record	alter domain for value in insert
Edit	Update	alter domain for value in update
Grant	Grant	for same object and grantee
Grant	Modify	grant or revoke for or on modified proc/func/method
Modify	Modify	of same proc/func/method or rename to same name
Modify	PMethod	rename to same name as new method
PColumn	PColumn	same name in same table
PDomain	PDomain	domains with the same name
PIndex	PIndex	another index for the same table
PProcedure	PProcedure	two new procedures/funcs with same name
PRole	PRole	two new roles with same name
PTable	PTable	two new tables with same name
PTable	PView	a table and view with same name
PTrigger	PTrigger	two triggers for the same table

PView	PView	two new views with the same name
Record	Record	conflict because of entity constraint
Record	Update	conflict because of entity or referential constraint
PeriodDef	Drop	Conflict if the table or column is dropped during period definition
Versioning	Drop	Conflict if the table or period is dropped during versioning setup

4.2.3 Entity Integrity

The main entity integrity mechanism is contained in Participant. However, a final check needs to be made at transaction commit in case a concurrent transaction has done something that violates entity integrity. If so, the error condition that is raised is "transaction conflict" rather than the usual entity integrity message, since there is no way that the transaction could have detected and avoided the problem.

Concurrency control for entity integrity constraints are handled by IndexConstraint (level 2). It is done at level 2 for speed during transaction commit, and consists of a linked list of the following data, which is stored in (a non-persisted field of) the new Record

- The set of key columns
- The table (defpos)
- The new key as an array of values
- A pointer to the next IndexConstraint.

During Participant.AddRecord and Participant.UpdateRecord a new entry is made in this list for the record for each uniqueness or primary key constraint in the record.

When the Record is checked against other records (see section 4.2.2) this list is tested for conflict.

4.2.4 Referential Integrity (Deletion)

The main referential integrity mechanism is contained in Participant. However, a final check needs to be made at transaction commit in case a concurrent transaction has done something that violates referential integrity. If so, the error condition that is raised is "transaction conflict" rather than the usual referential integrity message, since there is no way that the transaction could have detected and avoided the problem.

Concurrency control for referential constraints for Delete records are handled by ReferenceDeletionConstraint (level 2). It is done at level 2 for speed during transaction commit, and consists of a linked list of the following data, which is stored in (a non-persisted field of) the Delete record

- The set of key columns in the referencing table
- The defining position of the referencing table (refingtable)
- The deleted key as an array of values
- A pointer to the next ReferenceDeletionConstraint.

During Participant.CheckDeleteReference a new entry is made in this list for each foreign key in the deleted record.

When the Record is checked against other records (see section 4.2.2) this list is tested for conflict. An error has occurred if a concurrent transaction has referenced a deleted key.

4.2.5 Referential Integrity (Insertion)

The main referential integrity mechanism is contained in Participant. However, a final check needs to be made at transaction commit in case a concurrent transaction has done something that violates referential integrity. If so, the error condition that is raised is "transaction conflict" rather than the usual referential integrity message, since there is no way that the transaction could have detected and avoided the problem.

Concurrency control for referential constraints for Record records are handled by ReferenceInsertionConstraint (level 2). It is done at level 2 for speed during transaction commit, and consists of a linked list of the following data, which is stored in (a non-persisted field of) the Record record

- The set of key columns in the referenced table
- The defining position of the referenced table (reftable)
- The new key as an array of values
- A pointer to the next ReferenceInsertionConstraint.

During Participant.AddRecord a new entry is made in this list for each foreign key in the deleted record.

When the Record is checked against other records (see section 4.2.2) this list is tested for conflict. An error has occurred if a concurrent transaction has deleted a referenced key.

4.4 System and Application Versioning

With version 4.6 of Pyrrho versioned tables are supported as suggested in SQL2011. PeriodDefs are database objects that are stored in the Table structure similarly to constraints ad triggers. PeriodSpecs are query constructs that are stored in the Context: e.g. FOR SYSTEM_TIME BETWEEN .. and .. ,

The GenerationRule enumeration in PColumn allows for RowStart and RowEnd autogenerated columns (as required by SQL2011) and also RowNext, as required for Pyrrho's implementation of application-time versioning.

a system or application period is defined for a table, Pyrrho constructs a special index called versionedRows whose key is a physical record position, and whose value is the start and end transaction time for the record. This versionedRows structure is maintained during CRUD operations on the database. If a period is specified in a query, versionedRows is used to create an ad-hoc index that is placed in the Context (there is a BTree called versionedIndexes that caches these) and used in constructing the rowsets for the query.

If system or application versioning is specified for a table with a primary index, new indexes with special flags SystemTimeIndex and ApplicationTimeIndex respectively are created: these are primary indexes for the pseudotables T FOR SYSTEM_TIME and T FOR P which are accessible for and "open" period specification.

5. Parsing

Pyrrho uses LL(1) parsing for all of the languages that it processes. The biggest task is the parser for SQL2011. There is a Lexer class, that returns a Sqlx or Token class, and there are methods such as Next(), Mustbe() etc for moving on to the next token in the input.

From version 7, parsing of any code is performed only in the following circumstances[10]:

- The transaction has received SQL from the client.
- SQL code is found in a Physical being loaded from the transaction log (when the databse is opened, or after each commit)

The top-level calls to the Parser all create a new instance of the Transaction, containing the newly constructed database objects as described in 3.5.3 above. They begin by creating a Context for parsing.

Within the operation of these calls, the parser updates its own version of the Transaction and Context. The Context is not immutable, so update-assignments are mostly not required for the Context. The recursive-descent parsing routines return the new database objects constructed (Query, SqlValue, Executable) for possible incorporation into the transaction.

5.1 Lexical analysis

The Lexer is defined in the Pyrrho.Common namespace, and features the following public data:

- char[] input, for the sequence of Unicode characters being parsed
- pos, the position in the input array
- start, the start of the current lexeme
- tok, the current token (e.g. Sqlx.SELECT, Sqlx.ID, Sqlx.COLON etc)
- val, the value of the current token, e.g. an integer value,the spelling of the Sqlx.ID etc.

The Lexer checks for the occurrence of reserved words in the input, coding the returned token value as the Sqlx enumeration. These Sqlx values are used throughout the code for standard types etc, and even find their way into the database files. There are two possible sources of error here (a) a badly-considered change to the Sqlx enumeration might result in database files being incompatible with the DBMS, (b) the enumeration contains synonyms such as Sqlx.INT and Sqlx.INTEGER and it is important for the DBMS to be internally consistent about which is used (in this case for integer literal values).

The following table gives the details of these issues:

Sqlx id	Fixed Value	Synonym issues
ARRAY	11	
BOOLEAN	27	
CHAR	(37 recode)	Always recode CHARACTER to CHAR
CLOB	40	
CURSOR	65	
DATE	67	
INT	(128 recode)	Always recode INT to INTEGER
INTEGER	135	
INTERVAL	137	
MULTISET	168	
NCHAR	171	
NCLOB	172	
NULL	177	

[10] Versions of Pyrrho prior to v7 reparsed definitions for each role, since roles can rename objects. This was a mistake, since execution of any definition always occurs with the definer's role.

NUMERIC	179	
REAL	203	
TIME	257	
TIMESTAMP	258	
TYPE	267	
XML	356	

Apart from these fixed values, it is okay to change the Sqlx enumeration, and this has occurred so that in the code reserved words are roughly in alphabetical order to make them easy to find.

5.2 Parser

The parser retains the following data:

- The Lexer
- The current token in the Lexer, called tok (1 token look ahead)
- The current Context, including the current Database or Transaction

In addition there are lists for handling parameters, but these are for Java, and are described in chapter 11. Apart from parsing routines, the Parser class provides Next(), Match and Mustbe routines.

5.2.1 Execute status and parsing

Many database objects such a stored procedures and view contain SQL2011 source code, so that database files actually can contain some source code fragments. Accordingly parsing of SQL code occurs in several cases, discriminated by the execute status (see 3.10.1) of the transaction:

Execute Status	Purpose of parsing
Parse	Parse or reparse of stored procedure body etc. Execution of procedure body uses the results of the parse (Execute class)
Obey	Immediate execution, e.g. of interactive statement from client
Drop	Parse is occurring to find references to a dropped object (see sections 5.3-4)
Rename	Parse is occurring to find references affected by renaming (sections 5.3-4)

5.2.3 Parsing routines

There are dozens of parsing routines (top-down parsing) for the various syntax rules in SQL2011. The context is provided to the Parser constructor, to enable access to the execute status, Nearly all of these are private to the Parser.

The routines accessible from other classes are as follows:

Signature	Description
Parser(cx)	Constructor
void ParseSql(sql)	Parse an SqlStatement followed by EOF.
SqlValue ParseSqlValue(sql,type)	Parse an SQLTyped Value
SqlValue ParseSqlValueItem(sql)	Parse a value or procedure call
CallStatement ParseProcedureCall(sql)	Parse a procedure call
WhenPart ParseTriggerDefinition(sql)	Parse a trigger definition
SelectStatement ParseCursorSpecification(sql)	Parse a SELECT statement for execution
QueryExpression ParseQueryExpression(t,sql)	Parse a QueryExpression for execution

All the above methods with sql parameters set up their own Lexer for parsing, so that

new Parser(cx).ParseSql(sql)

and similar calls work.

6. Query Processing and Code Execution

In section 2.2 a very brief description of query processing was given in term of bridging the gap between bottom-up knowledge of traversable data in tables and joins (e.g. columns in the above sense) and top-down analysis of value expressions.

From v.7 Queries are fully-fledged DBObjects, and are only parsed once. During parsing the Context give access to objects created by the parse, which initially have transaction-local defining positions Some of these will be committed to disk as part of some other structure (e.g. a view or in a procedure body), when of course they will get a new defining position given by the file position.

I would like the context to have lists of queries, executables and rowsets (similarly to the lists of TypedValues). At every point starting a new transaction simply inherits the current context state, but the old context becomes accessible once more when the stack is popped.

6.1 Overview of Query Analysis

Pyrrho handles the challenge that identifiers in SQL queries are of several types and subject to different rules of persistence, scope and visibility. Some identifiers are names of database objects (visible in the transaction log, possibly depending on the current role), queries can define aliases for tables, views, columns and even rows, that can be referred to in later portions of the query, user defined types can define fields, documents can have dynamic content, and headers of compound statements and local variables can be defined in SQL routines. Added to all of this is the fact that ongoing transactions proceed in a completely isolated way so that anything they have created is hidden from other processes and never written to disk until the transaction commits.

In addition, Pyrrho operates a very lazy approach to rowSet traversal. RowSet traversal is required when the client requests the results of a query, and as required for source rowSets when an ordering or grouping operation is required by a result traversal. A significant number of rowSet classes is provided to manage these processes. RowSet traversal always uses bookmarks as described elsewhere in this booklet.

In previous versions of Pyrrho, the above considerations led to a lookup process in which identifiers were looked up at runtime, using lists created during a runtime parse of the relevant source codes. Version 7 and later handles this differently. All SQL query text, whether coming from the database file or from an interactive user, is immediately parsed into a Query structure, in which all the above identifiers have been replaced by a long defpos, which for database objects is (or is changed to) the defining position in the transaction log. It remains true that different roles can have different names for database objects, and this is handled by keeping the naming of objects in the role. The defining role owns this ObInfo, and it is added to other roles that are granted access to the object. So expressions and Query are database-objects (in db.objects) while objects also have role-based ObInfos (in role.infos), stored with the same defpos but in the Role's mem tree. Several non-immutable structures (Reader, Writer, Context) have their own local copies of the database to speed things up, and Query and SqlValue contain naming and ObInfo information for the current role.

The parsing process is recursive. During the lexical (left-to-right) phase of analysis the lexer supplies defining positions for unknown SqlValue it encounters. When a FROM clause in the query enables targets of any of these selectors to be identified, the defining position is updated to match the target. In previous versions of Pyrrho, this was referred to as the Sources stage of analysis. In v7 there is no separate analysis stage and the query is progressively rewritten during the parse, with identifiers and aliases being substituted by the query or sqlvalue they refer to.

QuerySpecifications have SqlValue select items and maintain a scope consisting of all referenceable DBObjects from the containing query or domain if any, and all references created within it to the left for the current position (for example, join operands and their columns). Select items may refer ahead to objects brought into scope by the from clause: until they are resolved they will simply be new SqlValues with the undefined domain Content.

For example, in a query such as "Select b+c as d from a" , the defining positions for the selectors b and c will be replaced by those of columns in a, and the query will have a single selector whose defining position is that of the + operator as the column in the row type for the query and name d. In "Select a.c from t a" the first occurrence of a in the dotted expression will be replaced by t, and c by the column of

t, so that the rowType will have a selector “a.c” whose defining position is that of the dotted expression[11].

Parsing proceeds from left to right, and objects are replaced one at a time when further information about them is known. For example, replacing a single select item by uid, affects all objects that refer to it including the containing queries (rowType and domain). Queries can also be replaced (with the same uid) when conditions and filters are moved within the resulting structure. The context uses a special structure called done that records the new version for each uid that has been scanned. If a part of the query contains more than one reference to the item being replaced, the enclosing structure may be replaced more than once (overwriting its entry in done), but when the scan reaches a reference to the changed uid it will not be rescanned.

As the name implies, these uids need to be unique within the thread. The uniqueness is guaranteed by using the lexical position of the starting token of the query or sqlvalue, offset from the transaction identifier tid. The first step in a transaction for a given thread starts at 0x4000000000000000. An explicit transaction may have several steps and one step can define uncommitted database objects that are used in later steps, so that each step gets a new tid (transaction-uid), incremented by the length of the sql used in the previous step.

Views complicate the picture further, since and each occurrence of a view will increase the tid-increment by the length of corresponding source. Expressions are given uids simply by their lexical position of the first character, or for complex expressions by the position of the top operator (e.g. + < or .).

Queries are more complicated for uids because of the way they are nested in SQL. CursorSpecifications always start with SELECT, so the SELECT position is used for its uid. QueryExpression will have as uid the position of the second character of the SELECT (or UNION etc) that introduces it and the QuerySpecification will use the last character of the corresponding token (so that it gets a different uid from its first selector). FROM introduces a TableExpression, and Joins have uids given by the position of the join operator. The use of SELECT * and other asterisk expressions will extend the source of the query (similarly to Views as mentioned above) by the list of identifiers represented by the asterisk.

Aliases are always simple strings, and identifier chains are always parsed as dot expressions: if a join operation needs dotted chains for disambiguation this is done by creating dot expressions: the default column name for a column containing a dotted expression is the identifier chain.

An example will help to explain the process. Consider the following (not clever SQL but has enough features to illustrate the process). Suppose the database defines (only)

```
create table author(id int primary key,aname char)
create table book(id int primary key,aid int references author,title char)
```

Then the following uids are defined in the database, as can be verified from the log:

22	AUTHOR	a Table
33	INTEGER	
47	ID	for 22
69		an Index (ID) for 22
83	CHAR	
96	ANAME	for 22
139	BOOK	a Table
148	ID	for 139
171		an Index (ID) for 139
187	AID	for 139
212		an Index (AID) referencing 22
230	TITLE	for 139

[11] The defining position of an expression is that of the top operator, such as + or . . In general, we can’t replace a dotted expression simply with its right hand operand, because the dotted notation is used for disambiguation, not just among columns with the same name, but often between multiple instances of the table (t here).

Now consider the following query:

```
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890123456789
SELECT b.title AS c FROM author a, book b WHERE a.id=b.aid AND c>'M'
```

Query processing always starts with a context that is “empty”, i.e. it has a snapshot of the database/transaction and knows the role and user for the current command, but nothing else. Parsing will cache or create some database objects and metadata, and maintain a table called defs that helps with translation from identifiers to uids. Once parsing is complete, all objects in the context will be identified by their uids, and not by identifiers. The uid for an object cached from the database will be its defining position in the database (position in the transaction log (e.g. 22 for AUTHOR), while the uid for a new object will be its lexical position in the command (e.g. #8 for the identifier b), or its transaction id, (e.g. '0 for the first object created in the transaction), or a heap uid (e.g. %0 for the first object not in any of these categories).

The keyword SELECT introduces a QuerySpecification which is given the uid #7. Following SELECT we start to parse the select items. When the parser reaches lexical position #8, it parses b.title as an identifier chain and looks it up in the context. As the context is empty, it is not known whether b is an object name or an alias, though the dot allows us to infer that it will have structure. ParseVarOrColumn() builds a SqlValueExp #9 whose operator is DOT, and #9 and #11 are unknown SqlValues whose names are b and title respectively. After also noting that alias C at position #19 refers to #9, ParseSelectItem returns the updated QuerySpecification #7 and the SqlValue #9 as a tuple. Here we will show the current state of the parse as

QuerySpecification #7 (#9(B #8.TITLE #10) as C) from ?

B and TITLE are unknown at this stage. At position 26, we lookup author in the database’s metadata for the context’s Role, find it is a table (with file position 22), and add the alias at position 33. The parser returns this as a From object with lexical position #26 in case the command later has another reference to this table. The From is a subclass of Query, and so has a row type. Its constructor is passed the select list, and assigns heap uids for any columns that are so far unreferenced, which is all of them in this case:

From AUTHOR #26 (%0,%1) target=22

The context notes the definitions of both AUTHOR and A as #26.

The comma at position #34 indicates a join. We look up the BOOK table as another From object, and this allows us to resolve the unknown B.TITLE as a simple column (to be copied from the table). So ParseTableReferenceItem returns

From BOOK #36 (#10,%2,%3) target=139

TITLE is now resolved as an SqlCopy and its query, domain and tableColumn are known:

SqlCopy TITLE #10 From #36 Domain: 83 CHAR copy from 230

and the context has noted that B refers to #36.

The next step in ParseFromClause is to construct the JoinPart for these two From objects, whose row type and domain are now known:

JoinPart #34 (#9) #26 CROSS join #36

Next, ParseWhereClause returns a disjunction list (#53,#65):

SqlValueExpr #53(#49=%3)

SqlValueExpr #65(#9>#66)

We have reached the end of the query (no grouping, or further joins) so ParseTableExpression returns

TableExpression #21 (#9) Where:(#53,#65) From: #34

and this completes the QuerySpecification:

QuerySpecification #7 (#9) from #21

As there are no unions/intesections in this query, it is a simple query expression

QueryExpression #2 (#9) Left: #7

there is no order clause, and the cursor specification is now complete:

CursorSpecification #1 (#9) union #2

The next stage is the construction of RowSets, which are gathered in the Context in a list called data::

69=IndexRowSet AUTHOR 69 (ID 47, ANAME 103) Table=22,

#26=SelectedRowSet AUTHOR (A.ID #49, ANAME 103),

180=IndexRowSet BOOK 180 (ID 156, AID 202, TITLE 247) Table=146,

#36=SelectedRowSet BOOK (C #10, B.ID 156, AID #54 AID)

#34=JoinRowSet (A.ID #49, ANAME 103, C #10, B.ID 156, AID #54) CROSS where #53(SqlCopy A.ID #49 (33 INTEGER) from 47= SqlCopy B.AID #54 (33 INTEGER) from 202) and #65(SqlCopy TITLE #10 Alias C (90 CHAR) from 247>M),

#21=TableExpRowSet (A.ID #49, ANAME 103, C #10, B.ID 156, AID #54 where #53(SqlCopy A.ID #49 (33 INTEGER) copy from 47= SqlCopy B.AID #54 (33 INTEGER) from 202) and #65(SqlCopy TITLE #10 Alias=C (90 CHAR) from 247>M),

#7=SelectRowSet (C #10)

The cursor supplied at the top level in this case will contain a single column e.g. {(#10=Nostromo)}. Further information about RowSets and Cursors follows in the next sections.

6.2 RowSets and Context

RowSets give access to a Cursor for traversing a set of rows. Unless an operation such as sorting requires all its data, the rowset rows are not computed until traversal begins. Recall that queries, rowsets and cursors are all immutable data structures, rowsets and queries are nested structures, with SQL commands and results at the top level, and base table references at the bottom: each level contains a reference to an immuable structure at the next level down. A cursor contains a reference to its own rowset. This arrangement requires rowsets to have uids, which for IndexRowSet and TableRowSet can be the same as the index and table, and contain nothing related to the current command, role or user. SystemRowSets are the same for all databases.

RowSet class	Comments
IndexRowSet	Primary or unique index for any user and role
SystemRowSet	System Tables as defined in Manual sec 8. Many Cursor types
TableRowSet	Accesses base table for any user and role

Some RowSets have uids given by the lexical position in the command, as shown in the following table.

Syntax	RowSet class	Comments
From	TrivialRowSet	Static results: rowType from command
From	SelectedRowSet	Accesses base table:
JoinPart	JoinRowSet	Cursor classes for different join types
QueryExpression	MergeRowSet	
QuerySearch	TransitionRowSet	TransitionRowSet has defpos #0
QuerySpecification	SelectRowSet	
QuerySpecification	EvalRowSet	EvalRowSet has just one row
QuerySpecification	GroupingRowSet	
SqlInsert	TransitionRowSet	TransitionRowSet has defpos #0
TableExpression	TableExpRowSet	
UpdateSearch	TransitionRowSet	TransitionRowSet has defpos #0
VALUES	ExplicitRowSet	

Other RowSets are given uids as required from the different volatile uid ranges (see section 2.3). GroupingRowSet is particularly complex and discussed later in this section.

The Context keeps track of the structures required for computing results. A new Context must be created for each procedure stack frame, but the new stack frame can start off with all the previous frame's values still visible. SQL does not allow values in statically enclosing frames to be updated. When a procedure returns, the upper context is removed exposing the previously accessible data even if the procedure used the same identifiers for its local data.

The context contains the result of parsing SQL, as a collection of DBObjects (Queries and SqlValues, arranged by depth). During rowset traversal the context contains a set of current rowsets (data) and

their cursors. During cursor evaluation, the context contains an association between SqlValues and the rowset used to access its current value. Each rowset contains two versions of this assocation, one (_finder) for constructing its cursors, and another (finder) for use by clients. These relate to the rowset hierarchy in an obvious way: the internal _finder at one level is the external finder for level below. For some rowsets such as OrderedRowSet, the finders are changed after building.

In aggregated, grouped and windowed operations, the context contains a set of Registers for accumulating aggregated values for each SqlFunction and any appropriate group or window keys. To facilitate this, the StartCounter and AddIn functions are called with their own Cursor, even through all such will use the same transaction and context.

To compute a join, it is often the case that join columns have been defined and the join requires equality of these join columns (inner join). If the two row sets are ordered by the join columns, then computing the join costs nothing (i.e. O(N)): a join bookmark simply advances the left and right rows returns rows where the join columns have matching values. If the join columns are not in the right order for the join, a sorted rowset is interposed during parsing. The cost of ordering is O(NlogN+MlogM) if both sets need to be ordered. Cross joins cost O(MN) of course.

The above transformations of row sets motivate what happens during query processing in Pyrrho.

6.2.1 Grouped aggregations

SQL(2008-) allows a query to specify groupings by columns (or more generally by SqlValue expressions built from columns). Each grouping can be on a single column or lists of columns (as with indexes), called grouping keys. Columns (or expressions) in the source rowset that are not directly used for grouping are called data columns. The select list of the grouped query can contain SqlValues built from grouping keys and aggregation expressions directly or in combination: data columns cannot be referenced directly, The grouped rowset is built by traversing the source rowset, to compute partial sums of data columns for each value of the grouping keys.

6.3 SqlValue vs TypedValue

As discussed in the above section, during Query analysis it is quite common for queries to be combined (e.g. in Joins) or have additional columns added (e.g. derived or computed columns in the results).

The parser creates SqlValues, which in v7 are immutable and only need to be constructed once. SqlValues do not have exposed values: the value of an SqlValue is only found by evaluation. During query processing all structured types (normally based on TypedValues) have SqlValue analogues built from SqlTypeColumn. To see the difference consider the following examples of an array of NUMERIC

[INTEGER(1),34.5,0] TypedValues (the first one here has a subtype of NUMERIC)

[A,B,36] SqlValues (36 is an SqlLiteral)

So for example we have Row and SqlRow etc. As has been mentioned, evaluation Contexts are used as general containers, and are important for creating scopes during Parsing.

During rowset traversal, each Cursor computes the TypedValue for its current row. In complex queries, some query columns may depend on derived tables defined later in the query syntax, complicating the normal process of left-to-right evaluation. The evaluation of an SqlValue in these circumstances may temporarily result in a null TypedValue indicating the imcompleteness of evaluation (not a TNull.Value indicating a computed SQL NULL value), and causes the computation of such columns to be retried. This code is contained in Cursor.Value().

Activations form stacks during execution, local variables are defined by giving their activation and name (this is a Target). Variables identified by a dotted sequence of names will be searched for by unwinding the stack to find the appropriate context.

6.4 TransitionRowSet operation

The TransitionRowSet is the bridge between query processing and construction of physical records, and is governed by the rather complex description in the SQL standard. It is important to remember that TrasitionRowSet operations can be nested (e.g. a trigger can call an SQL Insert), so cannot use TableColumn uids directly for its cursor values. For this reason a TransitionCursor contains an extra

TRow for trigger execution, called targetRow, wjich is used to create a TargetCursor for use in the TriggerActivation.

6.5 Persistent Stored Modules

A big change in this version of Pyrrho is that code modules (triggers, stored procedures etc) are only parsed once per cold start. The uids given to DBObjects generated by the parser during Load are all in the range of file positions occupied by the corresponding source code in the transaction log. As with all parsing in this version, during the parsing process the resulting objects are added to the parser's context. These generated objects are not added to the Database objects tree, but stored (in Framing) in the trigger or procedure DBObject. When execution is required, these objects can simply be added to the activation context (by the line of code cx.obs += ob.framing) to make them available to the execution engine. The latter does not need access to the other transient parsing structures (such as defs), since ObInfo information is now added to each SqlValue and Query.

This brings a need to manage cx.obs during the construction of such frames. Each new Parser instance creates a Context where these structures are empty. A complex SQL statement might result in a number of Compiled objects (for example, a table whose column definitions include inline Check constraints), but we cannot simply construct a new Context for each to accumulate the framing for each one. But we can and should control the growth of cx.obs: once the framing property of the compiled object has been collected, we should restore cx.obs (and cx.defs amd cx.depths) to their values at the start of parsing the compiled object. Other parts of the context, such as cx.db, cx.nextPos, cx.nextId continue to accumulate the parsing results.

6.6 Trigger Implementation

As elsewhere in Pyrrho, parsing of executable SQL takes place on definition or grant of a database object. Definition is within the command processing that creates the object, or the database load on server start-up, and the associated private uids of compiled objects will differ for these two cases. The compiled objects are placed in the framing property of the associated DBObject, and these are added to the context when required.

Trigger code can refer to new row and table, or old row and table. For simplicity, the new row and table use the same uid as the target table, while different uids refer to the old row and table, which are cached at the start of trigger execution.

This section presents a worked example, based on Test16 in the PyrrhoTest program. This test has a table XA with three triggers defined, which modify two other tables XB and XC. The working below will deal with the second modification to XA, which is an update. The relevant declarations in the test are:

```
create table xa(b int,c int,d char)
create table xb(tot int)
insert into xb values (0)
[create trigger ruab before update on xa referencing old as mr new as nr
for each row begin atomic update xb set tot=tot-mr.b+nr.b;
set nr.d='changed' end]
```

At this point, the log contains:

Pos	Record Type	Contents
22	PTable XA	
29	PDomain INTEGER	
43	PColumn3 B	
64	PColumn3 C	
86	PDomain CHAR	
99	PColumn3 D	
138	PTable XB	
145	PColumn3 TOT	
186	Record 264[138]	145=0[-509]
217	Trigger RUAB	Trigger RUAB Update, Before, EachRow on 22,MR=old row, NR=new row : begin atomic update xb set tot=tot-mr.b+nr.b; set nr.d='changed' end
618	Record 618[22]	43=7[29],64=10[29],99=inserted[86]

656	Update 264[138]	145=7[29] Prev:264

The Trigger definition is stored in the log in source form, but an initial version of the compiled contents are now in memory in a field called framing on the Trigger object. The following "readable" version of this field is from the debugger, and contains the items that will be cached in the context when the trigger is used. As can be seen, the uids for the compiled objects are taken from the empty range 217-618 in the log above. The trigger will run in the definer's role, independently of the caller's identity[12].

```
{(22=Table Name=XA 22 Definer=-64 Domain: Domain: -535 TABLE([43, Domain: 29 INTEGER],[64,
Domain: 29 INTEGER],[99, Domain: 86 CHAR])  Enforcement=Select, Insert, Delete, Update,
  43=TableColumn 43 Definer=-64 Domain: Domain: 29 INTEGERTable=22,
  64=TableColumn 64 Definer=-64 Domain: Domain: 29 INTEGERTable=22,
  99=TableColumn 99 Definer=-64 Domain: Domain: 86 CHARTable=22,
  138=Table Name=XB 138 Definer=-64 Sensitive Domain: Domain: -535 TABLE([145, Domain: 29
INTEGER])  Enforcement=Select, Insert, Delete, Update,
  145=TableColumn 145 Definer=-64 Domain: Domain: 29 INTEGERTable=138,
  219=SqlCopy Name=B 219 From:220 Domain: 29 INTEGER copy from 43,
  220=From Name=XA 220 RowType:(219,221,222) Assigs: Target=22,
  221=SqlCopy Name=C 221 From:220 Domain: 29 INTEGER copy from 64,
  222=SqlCopy Name=D 222 From:220 Domain: 86 CHAR copy from 99,
  223=SqlNewRow 223 Domain: -536 ROW([219, Domain: 29 INTEGER],[221, Domain: 29 INTEGER],[222,
Domain: 86 CHAR]) ,
  224=SqlValueExpr Name=NR.D 224 From:86 Left:223 Domain: 86 CHAR Right:222 224(223.222),
  225=SqlOldRow 225 Domain: -536 ROW([219, Domain: 29 INTEGER],[221, Domain: 29 INTEGER],[222,
Domain: 86 CHAR]) ,
  226=SqlValueExpr Name=MR.B 226 From:29 Left:225 Domain: 29 INTEGER Right:219
226(225.219),227=227 CompoundStatement(228,229),
  228=228 UpdateSearch230 set UpdateAssignment Vbl: 231 Val: 232,229=224=234,
  230=From Name=XB 230 RowType:(231) Assigs: Assigns:(0=UpdateAssignment Vbl: 231 Val: 232)
Target=138,
  231=SqlCopy Name=TOT 231 From:230 Domain: 29 INTEGER copy from 145,
  232=SqlValueExpr Name= 232 From:29 Left:233 Domain: 29 INTEGER Right:224 232(233+224),
  233=SqlValueExpr Name= 233 From:29 Left:231 Domain: 29 INTEGER Right:226 233(231-226),
  234=changed,
  235=235 WhenPart_ Stms: (227))}
```

In the test, there are definitions of an Insert trigger for XA, and an insert in XA that triggers an insert in XB. After these steps, the following have been added to the log:

The next step is the following command

```
update xa set b=8,d='updated' where b=7
```

We take up the story in Table.Update, which is about to apply the update trigger for table 22.

```
(tr,vals) = trb.UpdateRB(tr, cx, vals);
```

The current context has just computed updated values for the row in XA:

cx.values: {(43=8,64=10,99=updated,@8=Row(B=7,C=10,D=inserted))}

The trigger execution TransitionRowSet.Exec() takes place in the trigger activation context #2 that was set up by the TransitionRowSet. TriggerActivation.Exec places a copy of the old row for table 22 in its values:

ta.values {(22=Row(B=7,C=10,D=inserted),43=8,64=10,99=updated,
@8=Row(B=7,C=10,D=inserted))}

and calls Obey for the trigger action, leading to UpdateSearch XB.Obey(). This recursively calls Table.Update, which has its own transition rowset but no triggers have been defined for XB. This time the evaluation of the trigger's update assignment for XB is interesting. The first update assignment is

Vbl: TOT Col: 145 Val: 567(562(TOT Col: 145- Old B Col: 43 TRS:22)+ B 568 Col: 43)

Evaluating the right-hand side here proceeds as follows: 567 left.Eval() calls 562 Eval() 562.right is SqlOldRow[43]-> 7. 562.left is TOT so is also 7. 567.right.Eval() retrieves the new value 8, so that TOT:=8.

An Update record is generated for XB setting TOT:=8.

The next statement executed for the trigger is the assignment statement

[12] The caller's identity is important for mandatory access control if this feature is in use.

{ D 99 Col: 99=changed}

and this results in the new row for table XA to have contents {(43=8,64=10,99=changed)}.

The log entries for the command execution are committed as

Pos	Record Type	Contents
677	PTransaction for 3	
694	Update 618[22]	43=8[29],64=10[29],99=changed[86] Prev:618
730	TriggeredAction 506	
737	Update 264[138]	145=8[29] Prev:656

The PTransaction and TriggeredAction records are for identifying the user and role that made the changes. The triggered action will take place using the trigger definer's role, so that the boundaries of responsibility are made clear in the log.

7. Permissions and the Security Model

This chapter considers only the security model of SQL2011, i.e. machinery for access control (GRANT/REVOKE). Pyrrho's model for database permissions follows SQL2011 with the following modifications

- GRANT OWNER TO has been added to allow ownership of objects (or the database) to be transferred
- REVOKE applies to effective permissions held without taking account of how they were granted

The last point is documented in the manual as a set of proposed changes to the SQL2011 standard. The manual provides a significant amount of helpful information about the security model, e.g. in sections 3.5, 4.4, 5.1, 5.5, 13.5.

7.1 Roles

Each session in Pyrrho uses just one Role, as defined in the SQL2011 standard. Tables and columns can be granted to roles. However, Pyrrho allows Roles to have different (conceptual, data) models of the database: tables and columns can be renamed, and new generated columns can be defined. Generated columns can be equipped with update rules. Moreover, XML metadata such as declaring a column as an attribute or directing that a foreign key should be auto-navigated, can be specified at the role level.

With this model, database administrators are encouraged to have roles for different business processes, and given these roles privileges over tables etc, and grant roles to users, rather than granting privileges to users direct.

Since in general a Pyrrho session involves connections to several databases, there is at any time a set of effective roles in operation, one for each database in the connection. This set is called the authority for the transaction. The authority is dynamic, since procedures, checks,. triggers etc. run under definer's rights, as discussed below.

7.1.1 The schema role for a database

To get things started, a new database comes with a schema role (with the same name as the database) that can do anything and grant anything, and the database owner can use and administer this role. At the moment there is no mechanism for modifying the name of the schema role. Because this is the very first object to be created in a new database its defining position is the start of data in the database (this is currently 4).

It is good practice to create other roles for all operations on database tables and other objects and to transfer privileges to these roles rather than use the schema authority for all operations. Once this is done, it is recommended to revoke privileges from the schema authority. Note that without proper care it is easy to lose all ability to update (or even access) the database.

7.1.2 The guest role (public)

The other predefined role is "guest" or PUBLIC, which has a defining position of -1. This role has access to all public data, and is the owner of standard types.

The guest role is the default one, and cannot be granted, revoked, or administered. Standard types cannot be modified (for example, you cannot ALTER DOMAIN int), since this would require administration of the guest role.

All new public objects are added to the namespace of all roles.

7.1.3 Other roles

On creation, a new role is owned and administered by the creating user (who must have had administrative permission in the session role in order to create a role). All public objects are automatically added to the namespace of the new role.

All new data objects are added to the definer's role only.

7.2 Effective permissions

The current state of the static permissions in a database can be examined using the following system tables: (Only the Database owner can examine these.) By default the defining role is the owner of any orbject.

Name	Description
Role$Domain	Shows the definers of domains
Role$Method	Shows the definers of methods of user-defined types
Role$Privilege	Shows the current privileges held by any grantee on any object
Role$Procedure	Shows the definers of procedures and functions
Role$Trigger	Shows the definers of triggers
Role$Type	Shows the definers of user defined types
Role$View	Shows the definers of views

The session role gives only the initial authority for the transaction. During the transaction, the effective permissions are controlled by a stack. Stored procedures, methods, and triggers execute using the authority of their definers (and so can make changes to the database schema only if the definer has been granted the schema authority).

The Database owner is the only user allowed to examine database Log tables; and the owner of a Table is the only user allowed to examine the ROWS(..) logs for a table. The transaction user becomes the owner of a new database or new table. The GRANT OWNER syntax supports the changing of ownership on procedure and tables, and even of the whole database.

7.3 Implementation of the Security model

As is almost implied by the table in section 7.2,

- Each Database has an owner
- Each Table, Procedure or Method has an owner
- Each DBObject (including TableColumns) has an ATree which maps grantee to combinations of Grant.Privilege flags.
- Each Role has an ATree which maps database object (defining position) to combinations of Grant.Privilege flags.

7.3.1 The Privilege enumeration

These values are actually stored in the database in the Grant/Revoke record, so cannot be changed. This is a flags enumeration, so combinations of privileges are represented by sums of these values:

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

7.3.2 Checking permissions

The main methods and properties for this are as follows:

- CheckPermissions(DBObject ob, Privilege priv) is a virtual method of the Database class, whose override in Participant calls the ob.ObjectPermissions method. There is a shortcut version called CheckSchemaAuthority() for this specific purpose.
- ObjectPermissions(Database db, Privilege priv) has an implementation in DBObject that checks that the current user or the current authority holds the required privilege.
- There is an override of ObjectPermissions in the Table class, which deals with system and log tables: in the open source edition these tables are read-only and public.
- Table.AccessibleColumns computes the columns that the current user and authority can select.

7.3.3 Grant and Revoke

The main methods and properties for this are as follows:

- Access (Database db, bool grant, Privilege priv) grants or revokes a privilege on a DBObject. This requires the creation of a copy of the DBObject with a different users tree (a virtual helper method NewUsers handles this).
- Table.AccessColumn applies a Grant or Revoke to a particular column
- Transaction..DoAccess creates a Grant or Revoke record, and is called by Transaction.Access… routines that are called by the Parser.

An annoying aspect of the implementation is that Grant Select (or Insert, Update, or References) or Usage on a Table or Type implies grant select/usage of all its columns or methods, while a Grant on a Column or method implies grant select/usage on the table/type (but only the explicitly granted columns/methods). This behaviour is as specified by the SQL2011 standard (section 12.2 GR 7-10).

7.3.4 Permissions on newly created objects

As mentioned above, creation of new database objects requires use of the schema authority. Privileges on the new object are assigned to the schema authority as follows:

Object Type	Initial privileges for schema authority	with option
Table	Insert, Select, Update, Delete, References	Grant
Column	Insert, Select, Update	Grant
Domain	Usage	Grant
Method	Execute	Grant
Procedure	Execute	Grant
View	Select	Grant
Authority	UseRole	Admin
Role	UseRole	Admin

It is good practice to limit use of the schema authority, for example to a database administrator, and only for changes to the schema.

7.3.5 Dropping objects

The main methods and properties for this are as follows:

- When a database object is dropped it must be removed from any Roles that have privileges on it. This is done by Role.RemoveObject.
- Similarly when a grantee is dropped there is a DBObject.RemoveGrantee method to remove all references to it in grantees lists help by database objects.
- All late-parsed data in the schema (view definitions, procecure bodies, triggers, default values etc. are parsed in a DropTransaction: objects holding references to the dropped object either prevent the drop occurring (RESTRICT), or are dropped in a CASCADE, depending on the action specified.

8. The Type system and OWL support

Pyrrho supports SQL2011 type system and OWL types. It does so using a integrated data type system using the Domain class. The many subclasses of Domain used in previous versions of Pyrrho have been removed as the system had become unworkably complex with the addition of support for OWL classes.

There is now just one Domain class which deals with a large number of situations. In the simplest case, an Domain might be just new Domain(Sqlx.INTEGER), and other versions add precision and scale information, etc. Array and multiset types refer to their element type. Row types are constructed for any result set. Such types can be constructed in an ad hoc way, and may be added to the database by means of a generated domain definition if required when serialising a record.

In a user-defined type, defpos gives the Type definition, which can be accessed for method definitions etc, while the structType field gives the structure type. However, this information is only available for objects in the local database. A feature of Pyrrho is to allow multi-database connections and distributed databases, so during query processing data types exist that may have no counterpart in any database, or conversely may occur in more than one. We will discuss later in this section the rather subtle mechanisms Pyrrho needs to use to keep such volatility under control.

In the latest versions of Pyrrho, the type system has been greatly strengthened, with Level 2 (PhysBase) maintaining a catalogue of dataTypes and role-based naming. All data values are typed (the Value class). The Domain has a domaindefpos and a PhysBase name (a PhysBase pointer is not possible because of the way transactions are implemented). There are two local type catalogues maintained: for anonymous types such as INTEGER (or INTEGER NOTNULL etc), and for role-based named domains.

One further complicating aspect in the requirements for the type system is that SQL is very ambiguous about the expected value of different kinds of Subquery. In the SQL standard, subqueries can be scalar, row-valued, or multiset-valued, and row value constructors and table value constructors can occur with the ROW and TABLE keywords. The SQL standard discusses syntactic ambiguities between different kinds of subqueries (Notes 211 and 391). The starting point for such discussion is that users expect to be able to include a scalar query in a select list, and a rowset valued subquery in an insert statement, and both sorts in predicates. From v4.8 of Pyrrho, the type system is strengthened to enable us to distinguish different cases. We introduce the following usage:

1) CursorSpecifications result in a TABLE whose column names are given by the select list.
2) Insert into T values … the data following values will be of type TABLE where the named columns are contextually supplied by T, whereas the column names in the subquery's select list if any are only used within the subquery.
3) Subqueries can be used as selectors or in simple expressions, in which case we expect a scalar value (the target Domain is initially Null, so to be more specific we can require Sqlx.UnionDateNumeric or Sqlx.CHAR etc)
4) Subqueries can be used as a table reference, in which case the type will be of type TABLE and the column names will be those of the subquery, or the select list has a single element of the table's row type.
5) Subqueries can be used in predicates that expect them (e.g. IN, EXISTS etc) in which case the type will be MULTISET.
6) Select single row should have type ROW.

In general the type of a subquery result is a UNION of all the possible returned types. There is a step in query processing to Limit the resulting type.

As far as I can tell, all the usages of subqueries in SQL are provided for in Pyrrho's design. It is an error for such a subquery to have more than one column (unless it is prefixed by ROW so that the column will contain a single row, or TABLE where more than one row is possible) or more than one row (unless the subquery is prefixed by ARRAY, MULTISET, or TABLE as above).

During query analysis of the select lists, selectors have unique SqlName identifiers, and the identity of the corresponding data and types is propagated up and down the query parse tree, using the SqlValue

Setup method to supply constraining type information. Selectors are either explicitly in the column lists in select statements, or implicit as in select * or aggregation operations.

Rows are structured values that can be placed in the datafile: the dataType refers to the Table that defines the structure of the Row from the ordering of columns in the Table, and the data is an array of Column, each is defined by a TableColumn in the defining Table. In query processing on the other hand, the columns of a Row type are just values of the appropriate type for the column. The Row type is coerced to the type required for insertion as a row of the table during serialisation: just as when any value (structured or not) is inserted into a column of a table.

From the above discussion we see that Context notions are required at several points in the low levels parts of the DBMS, for parsing, formatting and ordering user-defined types. We need facilities at these lower levels to create role-based parsing and ordering contexts.

9. The HTTP service

This section sets out to explain the implementation of the HTTP and HTTPS service aspects of the Pyrrho DBMS.

From version 4.5 this is replaced by a REST service, where the URLs and returned data are role-dependent. The Role is specified as part of the URL, and WS-* security mechanisms are used to ensure that the access is allowed. Pyrrho supports the Basic authentication model of HTTP, which is sufficient over a secure connection. Support for HTTPS however is beyond the scope of these notes.

To keep the syntax intuitive multiple database connections are not supported. Only constant values can be used in selectors or parameter lists. Joins and other advanced features should be implemented using generated columns, stored procedures etc, which can be made specific to a role. However, the default URL mapping allows navigation through foreign keys.

Most of the implementation described in this section is contained in file HttpService.cs .

9.1 URL format

All data segments in the URL are URLencoded so, for example, identifiers never need to be enclosed in double quotes. Matching rules are applied in the order given, but can be disambiguated using the optional $ prefixes. White space is significant (e.g. the spaces in the rules below must be a single space, and the commas should not have adjacent spaces).

http://*host*:*port*/d*atabase*/*role*{/Selector}{/Processing}[/Suffix]

Selector matches

[$table]*Table*_id
[**$procedure**]*Procedure*_id
[$where]*Column*_id=string
[**$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.

Processing matches:

orderasc *Column*_id{, *Column*_id}
orderdesc *Column*_id{, *Column*_id}
skip *Int*_string
count *Int*_string

Suffix matches

$edmx
$xml
$sql

The $xml suffix forces XML to be used even for a single value. $edmx returns an edmx description of the data referred to. The $sql suffix forces SQL format. Posted data should be in SQL or XML format.

The HTTP response will be in XML unless it consists of a single value of a primitive type, in which case the default (invariant, SQL) string representation of this type will be used.

For example with an obvious data model, GET http://Sales/Sales/Orders/1234 returns a single row from the Orders table, while http://Sales/Sales/Orders/1234/OrderItem returns a list of rows from the OrderItem table, and http://Sales/Sales/Orders/1234/Customer returns the row from the Customer table

A URL can be used to GET a single item, a list of rows or single items, PUT an update to a single items, POST a new item to a list, or DELETE a single row.

For example, PUT http://Sales/Sales/Orders/1234/DeliveryDate with posted data of 2011-07-20 will update the DeliveryDate cell in a row of the Orders table.

POST http://Sales/Sales/Orders will create a new row in the Orders table: the posted data should contain the XML version of the row. In Pyrrho the primary key can be left unspecified. In all cases the new primary key value will be contained in the Http response.

9.2 REST implementation

A described above, the first few segments of the URL are used to create a connection to a database, and set the role. Then the selectors are processed iteratively, at each stage creating a RowSet that will be processed iteratively, with iteration over the final RowSet taking place when sending the results.

It is much simpler to use the facilities in this section as follows:

A single SELECT statement can be send in a GET request to the Role URL,and the returned data will be the entire RowSet that results, together with an ETag as a cache verification token.

If a single row is obtained in this way, a PUT or DELETE with a matching ETag can be used to update the row, provided no conflicting transaction has intervened.

A group of SQL statements can be POSTed as data to the Role URL and will be executed in an explicit transaction. The HTTP request can be made conditional or the continuing validity of previous results by including a set of ETags in the request.

The ETag mechanism is defined in RFC 7232. Its use supports a simple form of transactional behaviour. Very few database products implement ETags at present. The above descriptions can still work in a less secure way in the absence of ETags.

9.3 RESTViews

RestViews get their data from a REST service. The parser sets up the restview target in the global From. During Selects, based on the enclosing QuerySpecification, we work out a remote CursorSpecification for the From.source and a From for the usingTable if any, in the usingFrom field of the From.source CS.

Thus there are four Queries involved in RestView evaluation, here referred to as QS,GF,CS and UF. Where example columns such as K.F occur below we understand there there may in general be more than one of each sort.

All columns coming from QS maintain their positions in the final result rowSet, but their evaluation rules and names change: the alias field will hold the original name if there is no alias.

The rules are quite complicated:

If QS contains no aggregation columns, GF and FS have the same columns and evaluation rules: and current values of usingTable columns K are supplied as literals in FS column exps.

If there is no alias for a column expr, an alias is constructed naming K.

Additional (non-grouped) columns will be added to CS,GF for AVG etc.

GS will always have the same grouping columns as QS. For example

```
QS (AVG(E+K),F) group by F ->
   CS (SUM(E+[K]) as C_2,F COUNT(E+[K]) as D_2),
```

[K] is the current value of K from UF

```
   GF (SUM(C_2) as C2,F,COUNT(D_2) as D2)
      -> QS(C2/D2 as "AVG(E+K)", F)
```

Crucially, though, for any given QS, we want to minimise the volume D of data transferred. We can consider how much data QS 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 QS only refers to local data (no RESTViews) D is always zero, so that all of the following analysis is specific to the RESTView technology.

We will add a set of query-rewriting rules to the database engine aiming to reduce D by recursive analysis of QS and the views and tables it references. As the later sections of this document explain,

some of these rules can be very simple, such as filtering by rows or columns of V, while others involve performing some aggregations remotely (extreme cases such as select count(*) from V needs only one row to be returned). In particular, we will study the interactions between grouped aggregations and joins. The analysis will in general be recursive, since views may be defined using aggregations and joins of other views and local tables.

Any given QS might not be susceptible to such a reduction, or at least we may find that none of our rules help, so that a possible outcome of any stage in the analysis might be to decide not to make further changes. Since this is Pyrrho, its immutable data structures can retain previous successful stages of query rewriting, if the next stage in the recursion is unable to make further progress.

There are two types of RESTView corresponding to whether the view has one single contributor or multiple remote databases. In the simple exercises in this document, V is a RESTview with one contributor, and W has two. In the multiple-contributors case, the view definition always includes a list of contributors (the "using table", VU here) making it a simple matter to manage the list of contributors.

RESTView often refer to data from other RESTViews so everything needs to work recursively.

The simplest way of reducing the data transferreed is to identiy which columns in a remote view are actually needed by QS, either in the select list or in other expressions in QS such as join conditions.

The first aspect of rewriting we consider is filters. If there are some columns of the RESTView that are not used in the given query, there is an obvious reduction, and if a where-condition can be passed to the remote database, this will also reduce the number of rows returned.

There are two levels for filters in the Pyrrho engine. There is a low-level filter where particular requirements on defining positions and associated values can be specified: this is called match. There is a higher-level filter based on SQL value expressions, corresponding to SQL where and having conditions. To assist with optimisation Pyrrho works with lists of such conditions anded together.

Consider such a where condition E in QS. If both sides of the condition are remote, we can move the where condition to CS to filter the amount of data transferred. On the other hand, we cannot pass a where-condition to the remote database if it references data from a local table.

Similarly with aggregations: if all of the data being aggregated is remote, the aggregation can be done in CS. If some of the select items in QS contain local aggregation or grouping operations, it is often possibly to perform the operations in stages, with some of the agrregations can be done remotely, filtered where appropriate, so that the contributing databases provide partial sums and counts that can be combined in the GF. Most of the work for this is done in SqlValue.ColsForRestView, which is called by RestView.Selects. The base implementation of ColsForRestView examines the given group specification if any and builds lists of remote groups and GF columns. SqlValueExpr.ColsForRestView needs to consider many subcases according as left and right operands are aggregated, local or not, and grouped or not. SqlFunction.ColsForRestView performs complex rewriting for AVG and EXTRACT.

The above analyses and optimisations also need to be available when RESTViews are used in larger SQL queries. The analysis of grouping and filters needs to be applied top down so that as much as possible of the work is passed to the remote systems. Each RESTView target or subquery will receive a different request, and the results will be combined as rowsets of explicit values.

This contrasts with the optimisations used for local (in-memory) data, which instead aims to do as little work as possible until the client asks for successive rows of the results. In addition, detailed knowledge of table sizes, indexes and functional dependencies is available for local data, which helps with query optimisation.

10. References

Crowe, M. K. (2005-14): The Pyrrho Database Management System, University of the West of Scotland, www.pyrrhodb.com;

SQL2016: ISO/IEC 9075-2:2016 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, 2016)