An Introduction to the Source Code of the Pyrrho DBMS

Malcolm Crowe, University of the West of Scotland

www.pyrrhodb.com

Version 7.0 (December 2019)

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).

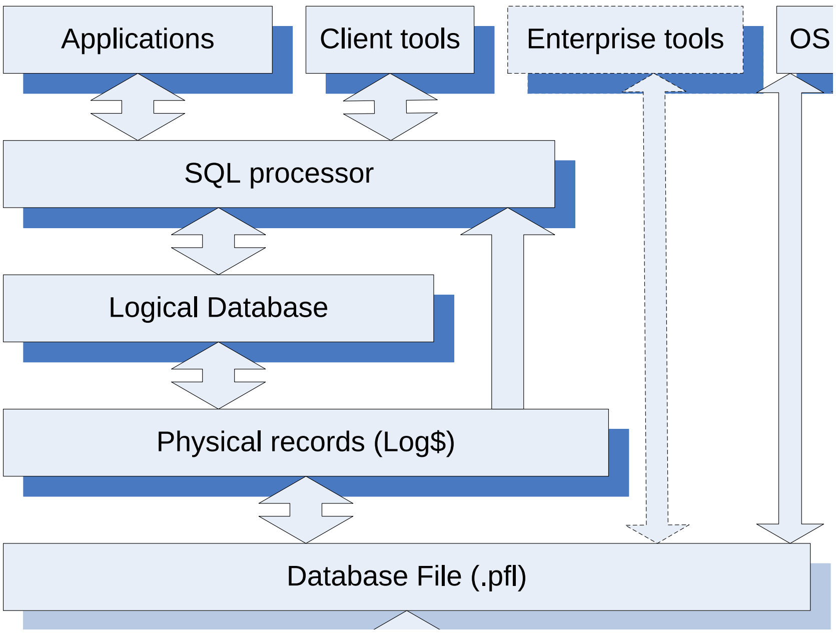
2. Overall structure of the DBMS

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. RowBookmarks are immutable and shareable, and give access to the version of the RowSet* that created them.

* RowSets are not immutable but are not modified once the first RowBookmark is constructed.



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) creation of ad-hoc data types for the results, (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 in many cases can choose 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. 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 come 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). There is a complication with the processing of syntax such as select * which implicitly introduces a set of identifiers for columns of a table, and these cannot simply all have the uid of the lexical position of the *, nor can their uids be the defining position of the associated table-column. The handling of * involves allocating a new range of uids after the end of the transaction command which is used during the transaction.

[2] Obviously, column references will have different values in different rows. During trigger operation the same is true for tables.

Executable statements (triggers, constraints, procedures etc) are contained in the database in both source and compiled forms. The compiled form is not written to the database file but constructed when the object is defined or granted to a role[3]. When committed, all uids in such compiled code will be file positions, except for select * identifiers, which consist of uids in the range range $6\times2^{60}..7\times2^{60}$-1 and accumulate for each committed transaction. (These will persist for the lifetime of the in-memory database, and an option would be to use negative uids (e.g. < -1000) for these instead.)

There is finally a "connection" range of uids, $7\times2^{60}..8\times2^{60}$-1 for prepared statements, as these accumulate and are shared with other transactions for this database. In the current version of Pyrrho. Each transaction starts with this set of prepared statements. This connection range can be used for local dynamic storage independently by transactions. Schema changes cannot be introduced during such execution of stored procedures, and so the heap is local to the current Command.

System objects	$-8\times2^{60}..-1$	Basis._uid (downwards)	Shared by Connections	
File positions	$0..4\times2^{60}-1$	0 (upwards)	Shared by Transactions	
New Physicals	$4\times2^{60}..5\times2^{60}-1$	Database.nextPos (up)	Local to Command	'
Query analysis	$5\times2^{60}..6\times2^{60}-1$	Database.nextTid (up)	Local to Statement	#
Executables	$6\times2^{60}..7\times2^{60}-1$	Database.nextPid (up)	Shared by transactions	@
Connection	$7\times2^{60}..8\times2^{60}-1$	Database.NextHeap (up)	Shared on a connection	%

The boundaries of these ranges are subject to change in later versions, as they are not relevant to durable file contents. It seems that these ranges need to be independently managed for the following reasons:

- File positions: audit requires asynchronous writing to the transaction log during a transaction. This is not required during the transaction commit.
- New Physicals: are created because of triggers at various points during a transaction step.
- Query analysis objects such as cursors need to be managed within statement operator, but statement operation can involve creation of new physicals.
- Storage for compiled executables is local to a database, is immutable, and can be used by successive transactions and connections.
- The prepared statement storage is semi-persistent and shared among sequential transactions in a single connection. 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

- 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 ConsoleApplication1 and UnitTestProjects provide some unit testing for the Pyrrho engine, and are configured to be compiled in the EmbeddedPyrrho solution.
- 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

[3] Different roles may have different names for renamable objects such tables and columns.

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

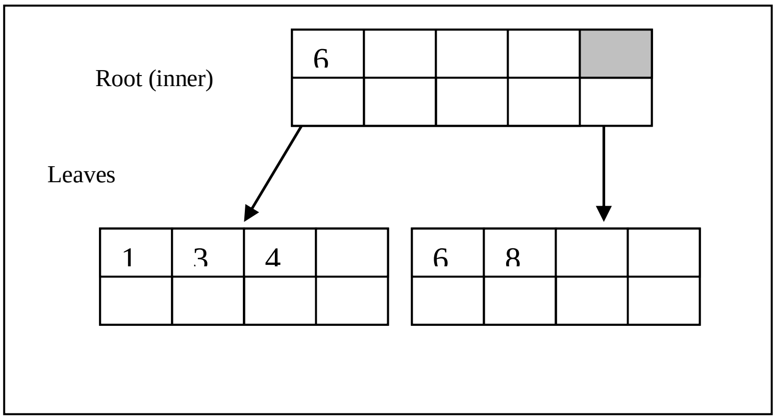
The B-Tree is a widely-used, fully scalable mechanism for maintaining indexes. B-Trees as described in textbooks vary in detail, so the following account is given here to explain the code.

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

RowBookmarks 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 4 ATree implementations:

Name	BaseClass	Description
BList<V>	BTree<K,V>	Same as BList<V> with a shortcut for adding to the end
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 classes are contained in the Level3 and Level4 namespaces. Neither of these is a subclass of ATree. MTree is immutable and shareable. RTree is used for multisets and rowSet implementation.

MTree	For multilevel indexes where the value type is long?
RTree	For multilevel indexes where the value type is SqlRow: It also contains a list of Rows, and so the RTree is not shareable.

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 as in MongoDB. The same mechanism is implemented for Update records in the database, so Document fields in Update records normally contain these operators, and Pyrrho computes and caches the updated document when the Update is installed in the database.

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.

Document matching recurses down to matching of fields. A number of MongoDB-style patterns including $ operators and aggregation pipelines were implemented and some of this code is visible in the sources.

3.2.5 Domain

Strong types are used internally for all transaction-level processing. The main mechanism for this is the Domain, which is a database object. It provides methods of input and output of data, parsing, coercing, checking assignability etc.

From version 7 ad-hoc ObInfos are used in query processing for row types: see section 3.5.7.

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 is a Domain and an object giving the value. TypedValues are generally immutable: the exceptions are TArray, TMultiset and TDocument. The following lists the subclasses of TypedValue:

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

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). From v7, objects are identified in the engine uniquely by their defining positions (see 3.5.5). Idents are constructed by the Lexer.

The Context has a special tree structure called Idents, for looking up DBObjects by their Ident.

Some parts of the Ident are immutable as shown below, but since the Ident is not immutable it must not be a part of any Basis subclass (see section 3.5).

CompareTo(ob)	Support alphanumeric comparison of Ident
Ident Final()	The last segment of the Ident
string ident	The head portion of the Ident
Ident(...)	*Numerous constructors*
readonly long iix	A unique uid, usually obtained from the lexer position
int Length()	The number of segments in the Ident
Lexer lexer	The Lexer that created this Ident if any

int lxrpos	The end of the Ident in the Lexer if any
int lxrstt	The start of the Ident in the Lexer if any
readonly Ident sub	The tail of the Ident
string ToString()	A readable version of the Ident

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.

3.4 Physical (level 2)

Physical is the base class used for actual items stored in the database file. There is an enumeration called Physical.Type, whose values are actually stored in the database, that discriminates between the different sorts of Physical record. The defining position of a Physical is given by the Reader or Writer position when reading or writing a database file, and by the current Transaction uid in Transactions. The Transaction case happens during parsing, as the parser creates new Physical structures: each one is installed in 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 transaction has its own sequence of uids (see sec 2.3) starting at 0x4000000000000000 for the first transaction in a thread, and for ease of reading, the resulting temporary defpos are rendered in ToString() as ‘1, ‘2 etc.

During Commit, the sequence of Physical records prepared by a Transaction is actually written (serialised) to durable media.

Each Physical type contributes a part of the serialization and deserialisation 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. During serialization relocation is required since Physicals in a Participant refer to each other through their defining position, and this defining position is not known until serialization begins. The Writer class also deals with the reassignment of defining positions from the file position during Commit.

In version 7 many of the so-called Physical classes contain structures belonging to the logical database levels: for example, expressions and executable statements. In the transaction log, these are stored in string form. During database Load these the parser from level 4 is used to construct this data, while newly defined database object data in such physical records is relocated during Commit.

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

3.5 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.

```
// 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	Domains (by DBObject defpos)
Table	TableRows
TableRow	Fields

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 readonly static long Name = --_uid; // string
```

For Basis subclasses with a unique name, the name property is then accessed by

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

This defines name as a readonly property of Basis[4].

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 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 new XX(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.

This pattern also enables the class of an object to be modified, within certain limits. For example an SqlValue subclass XX will have a constructor of form XX(SqlValue ob,BTree<long,object> m) allowing the creation of a new XX that replaces the previous ob (which might have been a different subclass of SqlValue), but retaining the same defining position defpos. Similarly, a DBObject subclass OB will have a constructor of form OB(long defpos, BTree<long,object> m) allowing the creation of an OB from its properties that replaces the object with that defpos. For more details about the use of the defining position concept, see the next two sections 3.5.3 and 3.5.3.

The following table shows all of the Keys defined for Basis subclasses. For details see that articles relating to the class.

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

The predefined property keys are all negative long constants. This means that positive keys are available for recording data. This is most suitable for when the positive keys are database file positions or other unique ids such as transaction uids, or sqid..

In the future we might take advantage of this for objects (Role) and Context[5] typevals (Sqid), Table rows and columns (Seq,Defpos). But there are good reasons for not adopting this idea, For example, the Table instance with the rows and indexes gets placed in the schemaRole. But having a separate ATree for each table is useful for traversal.

The following section list all subclasses of Basis. All such must be immutable and shareable.

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
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	
Guest	guestRole	Role		-55

[5] Context is not currently a subclass of Basis.

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

Levels	levels	BTree<Level,long>	classification	-56
LevelUids	cache	BTree<long,Level>		-57
NextHeap	heap	long	Connection uids, dynamic	-393
NextPid	nextPid	long	Aliases for compiled code	-394
NextPos	nextPos	long	Proposed new Physicals	-395
NextTid	nextTid	long	Offset of next transaction	-58
Owner	owner	long		-59
Roles	roles	BTree<string,Role>		-60
Types	types	BTree<Domain,Domain>	nameless types	-61

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.

The properties defined by the Transaction class are as follows:

Key	Property	Type	Comments	Uid
AutoCommit	autoCommit	bool		-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
Role	_role	long	role is roles[_role]	-285
SchemaKey	schemaKey	long	For POCO	-286
StartTime	startTime	long		-287
User	_user	long	user is (User)roles[_user]	-288
Warnings	warnings	BList<System.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
	obinfos	BTree<long,ObInfo>	stored in mem	
Procedures	procedures	BTree<string, BList<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[7]. 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.

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) and (c) dependency relationships between objects created during parsing. We explain these

[7] 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.

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 (obinfos) 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[8], so that in v7 the Role becomes responsible for all name lookups. Columns, fields and procedures use a two-stage lookup: columns are listed in the table's ObInfo, fields in the UDType's ObInfo, and the procedure lookup depends on the arity of the procedure. To simplify things a bit, rowType information is held in ObInfo rather than the Domain class, and the facility is extended to SqlValues and Queries in the current transaction.

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, in the following ranges:

Defpos Range	Objects	Interpretation	Notation
< -1	Properties and system objects		
-1	Ad hoc objects		
>= 0 and < 2^{62}	Committed objects	Position in transaction log	
>= 2^{62}	Transaction objects	Transaction offset + lexical position	'1, '2, ..

Defining positions are allocated by the Reader, and Transaction objects are relocated on Commit.

Basis properties for DBObject are as follows

Key	Property	Type	
_Alias	alias	string	-62
Classification	classification	Level	-63
Definer	definer	long	-64
Dependents[9]	dependents	BTree<long,bool>	-65
Depth[10]	depth	int	-66
Description	desc	string	-67
_Domain	domain	Domain	-176
Kind	kind*	Sqlx	-80
LastChange	lastChange	long	-68
Sensitive	sensitive	bool	-69

* kind is delegated to domain except for Domain and SqlValue classes.

DBObject has the following subclasses

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	
TableColumn	DBObject	TableColumns have check constraints
Trigger	DBObject	
UDType	Domain	
Method	Procedure	
PeriodDef	DBObject	

[8] 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!
[9] This field holds the defpos of objects that must be dropped if this object is dropped in a cascade.
[10] The maximum depth of dependent objects (depth is 1 if this objects has no dependents)

3.5.6 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 rowTypes (including user-defined types) by an ObInfo (see section 3.5.7).

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,Check>	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
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	Procedure		-84
Precision	prec	int		-85
Provenance	provenance	string	For imports	-86
Representation	representation	Domain	For user defined types	-87
Scale	scale	int		-88
Start	start	Sqlx (YEAR etc)		-89
Structure	structdef	long	Table reference for row and table types	-391
Under	super	Domain		-90
UnionOf	unionOf	BList<Domain>		-91

3.5.7 ObInfo

ObInfo is used for rowTypes and depends on the Role, so that a Role has a BTree of obinfos by each granted object’s defpos. ObInfos usually have a list of named columns, and a map to enable columns to be found by name. ObInfos have a domain property that may be a simple indication such as Domain.Content, Domain.Row, or Domain.TableType, but can also indicate a user-defined type. The result of Select is always described by an ObInfo, which may be constructed on an ad-hoc basis.

Key	Property	Type	Comments	Uid
Columns	columns	BList<DBObject>		-250
Map	map	BTree<string,int>		-251
Names	names	BList<string>		-252
Privilege	privilege	Grant.Privilege		-253
Properties	properties	BTree<string,long>	Property values in mem	-254

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

3.5.8 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. The Rows are in the mem BTree.

Key	Property	Type	Comments	Uid
ApplicationPS	appPS	long	For version tables	-262
Enforcement	enforcement	Grant.Privilege	For mandatory access control	-263
Indexes	indexes	BTree<CList<TableColumn>, long>		-264
SystemPS	sysPS	long	For version tables	-265
TableChecks	tableChecks	BTree<long,Check>	Evaluates to bool	-266
	tableRows	BTree<long,TableRow>	in mem	
Triggers	ptriggers	BTree<PTrigger.TrigType, BTree<long,Trigger>>		-267

TableRow has the following properties.

Fields	fields	BTree<long,TypedValue>		-227
Prev	prev	long	Previous version	-276
PrevKeys	prevKeys	BTree<long,PRow>	Index keys for prev	-277
Time	time	long	For versioning	-278

SystemTable has an additional property:

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

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
Display	display	int		-177
Enc	enc	Query	Nesting of queries in SQL	-178
FetchFirst	fetchFirst	int		-179
Filter	filter	BTree<long,TypedValue>		-180
_Import	import	BTree<SqlValue,SqlValue>		-181
Matches	matches	BTree<SqlValue,TypedValue>		-182
Matching	matching	BTree<SqlValue, BTree<SqlValue,bool>>		-183
OrdSpec	ordSpec	OrderSpec		-184
Periods	periods	BTree<long,PeriodSpec>		-185
_Replace	replace	BTree<string,string>		-186
RowType	rowType	ObInfo		-187

SimpleQuery	simpleQuery	Table		-189
Where	where	BTree<long,SqlValue>		-190

Query subclasses define some additional properties:

Key	Property	Type	Subclass	Uid
RVQSpecs	rVqSpecs	BList<QuerySpecification>	CursorSpecification	-192
RestGroups	restGroups	BTree<string,int>	CursorSpecification	-193
RestViews	restViews	BTree<long,RestView>	CursorSpecification	-194
_Source	_source	string	CursorSpecification	-195
Union	union	QueryExpression	CursorSpecification	-196
UsingFrom	usingFrom	Query	CursorSpecification	-197
FDConds	conds	BTree<long,SqlValue>	FDJoinPart	-211
FDIndexDefPos	indexDefPos	long	FDJoinPart	-212
FDRefIndexDefPos	rindexDefPos	long	FDJoinPart	-213
FDRefTableDefPos	rtableDefPos	long	FDJoinPart	-214
FDTeableDefPos	tableDefPos	long	FDJoinPart	-215
Reverse	reverse	bool	FDJoinPart	-216
Assigns	assigns	BList<UpdateAssignment>	From	-150
Source	source	Query	From	-151
Static		long	From	-152
_Target	target	long	From	-153
GroupKind	kind	Sqlx	Grouping	-232
Groups	groups	BList<Grouping>	Grouping	-233
Members	members	BList<SqlValue>	Grouping	-234
DistinctGp	distinct	bool	GroupSpecification	-235
Sets	sets	BList<Grouping>	GroupSpecification	-236
_FDInfo	FDInfo	FDJoinPart	JoinPart	-202
JoinCond	joinCond	BTree<long,SqlValue>	JoinPart	-203
JoinKind	kind	Sqlx	JoinPart	-204
LeftInfo	leftInfo	OrderSpec	JoinPart	-205
LeftOperand	leftOperand	Query	JoinPart	-206
NamedCols	namedCols	BList<long>	JoinPart	-207
Natural	natural	Sqlx	JoinPart	-208
RightInfo	rightInfo	OrderSpec	JoinPart	-209
RightOperand	rightOperand	Query	JoinPart	-210
Items	items	BList<SqlValue>	OrderSpec	-217
Distinct	distinct	bool	QueryExpression	-243
_Left	left	Query	QueryExpression	-244
Op	op	Sqlx	QueryExpression	-245
_Right	right	QueryExpression	QueryExpression	-246
SimpleTableQuery	simpleTableQuery	bool	QueryExpression	-247
Distinct	distinct	bool	QuerySpecification	-239
RVJoinType	rvJoinType	Domain	QuerySpecification	-240
Star	selectStar	bool	QuerySpecification	-241
TableExp	tableExp	TableExpression	QuerySpecification	-242
_Table	table	Table	SqlInsert	-154
Provenance	provenance	string	SqlInsert	-155
ValuesDataType	valuesdataType	Domain	SqlInsert	-156
From	from	SelectQuery	TableExpression	-198
Group	group	GroupSpecification	TableExpression	-199
Having	having	BTree<long,SqlValue>	TableExpression	-200
Windows	windows	BTree<long, WindowSpecification>	TableExpression	-201
Current	current	bool	WindowBound	-218
Distance	distance	TypedValue	WindowBound	-219
Preceding	preceding	bool	WindowBound	-220
Unbounded	unbounded	bool	WindowBound	-221
Val	val	SqlValue	UpdateAssignment	-237
Vbl	vbl	Selector	UpdateAssignment	-238
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

3.5.10 TableColumn

TableColumn is a subclass of DBObject. It is not an SqlValue: the SqlCol class manages the evaluable references to TableColumns in a Query.

Key	Property	Type	Uid
Checks	checks	BTree<long,Check>	-268
Generated	generated	GenerationRule (see below)	-269
Table	tabledefpos	long	-270
UpdateAssignments	update	BList<UpdateAssignment>	-271
UpdateString	updateString	string	-272

GenerationRule has the following properties:

Key	Property	Type	Uid
_Generation	generation	Generation	-273
GenExp	exp	SqlValue	-274
GenString	gfs	string	-275

TableColumn has the following subclasses:

Key	Property	Type	Class	Uid
Prev	prev	TableColumn	ColumnPath	-321
StartCol	startCol	long	PeriodDef	-387
EndCol	endCol	long	PeriodDef	-388

3.5.11 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	BList<TableColumn>	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->TableRow	-164

3.5.12 SqlValue

From v7, SqlValue is a subclass of DBObject, whose defpos is assigned by the transaction or reader position. As explained in sec 3.5.1, the subclass can be modified on resolution.
There are two base constructors: the one with Context as first parameter gives a new SqlValue, while the other allows an existing SqlValue to be modified (possibly to a new subclass, see 3.5.1).

Key	Property	Type	Comments	Uid
_From	from	From		-306
_Info	info	ObInfo		-307
Left	left	SQLValue		-308
Right	right	SQLValue		-309
Sub	sub	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	

3.5.13 SqlValue subclasses

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).

An SqlValue that has been defined in the top level of a Query select list is entered in the selects list and map for that Query. Before analysis it is entered as SqlName, SqlValueExpr, SqlFunction or SqlMethodCall depending on the syntax. All of these items can appear deeper inside an SqlValueExpr or ArgumentList. Instances of the SqlVariable subclass are used in an Activation to contain variables whose initial state is set when LocalVariable statements are executed.

In general, for any rowSet of the query there is a an TRow, whose value depends on which row we are on. The default value is the one in the final rowSet of the home query. There are special variable behaviours associated with triggers. An SqlVariable is a special kind of named SqlValue associated with a LocalVariable that may have been declared in a Routine. For any Activation of the query there is a corresponding SqlVariable, whose value can be updated.

SqlValue subclasses have the following additional properties:

Key	Property	Type	Subclass	Uid
Bits	bits	BList<long>	ColumnFunction	-333
GroupInfo	groupInfo	BList<SqlValue>	GroupRow	-320
Escape	escape	SqlValue	LikePredicate	-358
_Like	like	bool	LikePredicate	-359
Found	found	bool	MemberPredicate	-360
Lhs	lhs	SqlValue	MemberPredicate	-361
Rhs	rhs	SqlValue	MemberPredicate	-362
NIsNull	isnull	bool	NullPredicate	-364
NVal	val	SqlValue	NullPredicate	-365
All	all	bool	QuantifiedPredicate etc	-348

Between	between	bool	QuantifiedPredicate etc	-349
Found	found	bool	QuantifiedPredicate etc	-350
High	high	SqlValue	QuantifiedPredicate etc	-351
Low	low	SqlValue	QuantifiedPredicate etc	-352
Op	op	Sqlx	QuantifiedPredicate etc	-353
Select	select	QuerySpecification	QuantifiedPredicate etc	-354
Vals	vals	SqlValue[]	QuantifiedPredicate etc	-355
What	what	SqlValue	QuantifiedPredicate etc	-356
Where	where	Query	QuantifiedPredicate etc	-357
QExpr	expr	Query	QueryPredicate	-363
Call	call	CallStatement	SqlCall	-335
TableCol	tableCol	TableColumn	SqlCol	-322
CopyFrom	copyFrom	SqlValue	SqlCopy	-284
Sce	sce	SqlRow	SqlConstructor DqlDefaultConstructor	-336
Udt	ut	UDType	SqlConstructor SqlDefaultConstructor	-337
Spec	spec	CursorSpecification	SqlCursor	-334
Field	field	string	SqlField	-314
Filter	filter	SqlValue	SqlFunction	-338
Mod	mod	Sqlx	SqlFunction	-340
Monotonic	monotonic	bool	SqlFunction	-341
Op1	op1	SqlValue	SqlFunction	-342
Op2	op2	SqlValue	SqlFunction	-343
Query	query	Query	SqlFunction	-344
_Val	val	SqlValue	SqlFunction	-345
Window	window	WindowSpecification	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	Query	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<SqlRow>	SqlRowArray	-319
ArrayValuedQE	aqe	QueryExpression	SqlSelectArray	-327
TableRow	rec	TableRow	SqlTableRowStart	-311
TreatExpr	val	SqlValue	SqlTreatExpr	-313
TreatType	type	Domain	SqlTypeExpr	-312
Array	array	BList<SqlValue>	SqlValueArray	-328
Svs	svs	SqlValueSelect	SqlValueArray	-329
Modifier	mod	Sqlx	SqlValueExpr	-316
Expr	expr	Query	SqlValueSelect	-330
Source	source	string	SqlValueSelect	-331
TargetType	targetType	Domain	SqlValueSelect	-332
Attrs	attrs	BList<(XmlName,SqlValue)>	SqlXmlValue	-323
Children	children	BList<SqlXmlValue>	SqlXmlValue	-324
Content	content	SqlValue	SqlXmlValue	-325
Element	element	XmlName	SqlXmlValue	-326

3.5.14 Check

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

3.5.15 Procedure

Functions are procedures with a return type.

Key	Property	Type	Comments	Uid
Arity	arity	int	The number of parameters	-167
Body	body	Executable		-168
Clause	clause	string		-169
Inverse	inverse	long	For transformations	-170
Monotonic	monotonic	bool	For adapters	-171
Params	ins	Blist<ProcParameter>		-172
RetType	retype	Domain		-173

3.5.16 Method

Method is a subclass of Procedure.

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

3.5.17 Trigger

Key	Property	Type	Comments	Uid
Action	action	WhenPart		-290
Def	def	string	The source form of the action	-291
NewRow	newRow	string	if specified	-292
NewTable	newTable	string	if specified	-293
OldRow	oldRow	string	if specified	-294
OldTable	oldTable	string	if specified	-295
Table	tabledefpos	long		-296
TrigType	tgType	TrigType	insert/update before/after etc	-297
UpdateCols	updateCols	long[]	if specified	-298

3.5.18 UDType

Key	Property	Type	Comments	Uid
Methods	methods	BTree<long,bool>		-299
TableDefPos	tanledefpos	long		-300
UnderDefPos	underdefpos	long	if specified	-301
WithUri	withuri	string	if specified	-302

3.5.20 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 ObInfo or a Domain. The desired result of an Executable is therefore provided to the runtime system as a DBObject to cover there two cases.

Key	Property	Type	Class	Uid
Label	label	string	Executable	-92
Stmt	stmt	string	Executable	-93
_Type	type	Executable.Type	Executable	-94

Val	val	SqlValue	AssignhmentStatement	-105
Vbl	vbl	SqlValue	AssignmentStatement	-106
Parms	parms	BList<SqlValue>	CallStatement	-133
Proc	proc	Procedure	CallStatement	-134
Var	var	SqlValue	CallStatement	-135
Stmts	stmts	BList<Executable>	CompoundStatement	-96
CS	cs	SqlCursor	CursorDeclaration	-129
Cursor	cursor	SqlCursor	FetchStatement	-129
How	how	Sqlx	FetchStatement	-130
Outs	outs	Blist<SqlValue>	FetchStatement	-131
Where	where	SqlValue	FetchStatement	-132
Cursor	cursor	string	ForSelectStatement	-124
ForVn	forvn	string	ForSelectStatement	-125
Loop	loop	long	ForSelectStatement	-126
Sel	sel	CursorSpecification	ForSelectStatement	-127
Stms	stms	BList<Executable]	ForSelectStatement	-128
List	list	BTree<SqlValue,Sqlx>	GetDiagnostics	-141
HDefiner	hdefiner	Activation	Handler	-103
Hdlr	hdlr	HandlerStatement	Handler	-104
HType	htype	Sqlx	HandlerStatement	-100
Conds	conds	BList<string>	HandlerStatement	-101
Action	action	Executable	HandlerStatement	-102
CredPw	pw	SqlValue	HttpREST	-143
CredUs	us	SqlValue	HttpREST	-144
Mime	mime	string	HttpREST	-145
Posted	data	SqlValue	HttpREST	-146
Url	url	SqlValue	HttpREST	-147
Verb	verb	SqlValue	HttpREST	-148
Where	wh	SqlValue	HttpREST	-149
Else	els	BList<Executable>	IfThenElse	-116
Elsif	elsif	BList<Executable>	IfThenElse	-117
Search	search	SqlValue	IfThenElse	-118
Then	then	BList<Executable>	IfThenElse	-119
Init	init	TypedValue	LocalVariableDec	-97
LhsType	lhsType	Domain	MultipleAssignment	-107
List	list	BList<long>	MultipleAssignment	-108
Rhs	rhs	SqlValue	MultipleAssignment	-109
ParamMode	paramMode	Sqlx	ProcParameter	-98
Result	result	Sqlx	ProcParameter	-99
Ret	ret	SqlValue	ReturnStatement	-110
CS	cs	CursorSpecification	SelectStatement	-95
Outs	outs	BList<SqlValue>	SelectSingle	-142
Else	els	BList<Executable>	SimpleCaseStatement, SearchedCaseStatement	-111
Operand	operand	SqlValue	SimpleCaseStatement	-112
Whens	whens	BList<WhenPart>	SimpleCaseStatement, SearchedCaseStatement	-113
Exception	exception	Execption	Signal	-136
Objects	objects	object[]	Signal	-137
_Signal	signal	string	Signal	-138
SetList	setlist	BTree<Sqlx,SqlValue>	Signal	-139
SType	stype	Sqlx	Signal	-140
Cond	cond	SqlValue	WhenPart	-114
Stms	stms	BList<Executable>	WhenPart, LoopStatement	-115
Loop	loop	long	WhileStatement, RepeatStatement, LoopStatement	-121

Search	search	SqlValue	WhileStatement, RepeatStatement	-122
What	what	BList<Executable>	WhileStatement	-123
Nsps	nsps	BTree<string,string	XmlNameSpaces	-120

3.5.21 View

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

3.5.22 PeriodDef

Key	Property	Type	Comments	Uid
StartCol	startCol	long		-387
EndCol	endCol	long		-388

3.6 Level 4 Data Structures

Level 4 handles transactions, including Parsing, the execution context, result sets etc. All of the Query subclasses are derived from Context.

3.6.2 Context

A Context contains the main data structures for analysing the meaning of SQL. Query and Activation are the main subclasses of Context, and differ widely because of the different handling of identifiers: specifically, queries have mainly table and column identifiers and aliases that accumulate left to right, activations have variables that accumulate from enclosing static contexts, and activations form a stack during recursive procedure and method execution. Queries are analysed using a recursive structure as complex queries are formed from query-like clauses and subqueries. Following analysis, a query has an associated rowType, and can be evaluated to obtain a rowSet whose traversal usually involves recursive traversals of the rowSets in this structure. Queries can form the context for activation of functions called from referenced values, and activations can form the context for procedure and method calls. In v7 SqlValues no longer have a home Context: SqlValues are immutable objects and Contexts are modified during rowSet traversal and procedure execution.

This caused some confusion in earlier designs, but currently the overrides of Lookup in the Query and Activation classes try to keep things separate. The bottom Activation in the Activations stack will be a routine call from a Query or a CALL statement. Different stages in the Activation stack can use different Roles as procedure and trigger execution uses the definer’s role.

It is important that in SQL there is no concept of reference parameters, so at any time during expression evaluation only the most recent activations are 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, and such rowSets remain traversible even though the Activation is no longer on the stack. Such Activations therefore remain accessible (as a snapshot), generally until the end of the current Transaction.

Activations and Queries provide separate Push and Pop methods for stack management. All Context information is volatile. SQL procedure invocation makes a snapshot of the associated activation, as this may no longer be on the stack when rowset traversal begins.

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

- The field names of structured types give read access to the current TypedValues for the corresponding fields.
- A set of local variables (for activations) that can be the targets of assignment. CalledActivations and TriggerActivations are strongly typed and the defpos and domain of corresponding variables are known. Method activations etc have actual parameters with known defpos and values.[11].
- A set of contexts whose data has been imported into this one (queries) or are statically accessible (activations). For example, the identifiers in the target of a FROM are accessible in select * or within subqueries in the select list.

Assignment replaces the current context or their fields (not to query columns except in TransitionActivation as mentioned). Such assignment is non-destructive (we make a new copy of the current context).

SqlValues are best thought of as the select list of a Query. They can be evaluated on a context: e.g. each row of the query rowset is constructed by evaluating the select list on the RowBookmark. (RowBookmarks have a Value function that gives a TypedValue for each column defpos.)[12]

SqlValues can also form structured objects of different type, and methods and constructors use StructuredActivation. SqlValues are constructed for every data reference in the SQL source of a Query or Activation. Many of these are SqlNames constructed for an identifier in a query: during analysis all of these must be changed to a corresponding data reference. Obviously some SqlValues will be rows or arrays:

SqlValues can be *evaluated* at a given point in a transaction, using the Eval(tr,cx) method. The parameters here are a Transaction and Context, for accessing the database structures and runtime context respectifvely. The value will a TypedValue. There is a TNull class with TNull.Value implementing the logical SQL Null valueEval(tr) will return a binary null value if the result is not yet available (see footnote 6 above).

Data values and local variables can be of structured types (row, array, multiset, user-defined, documents). In normal use such complex data can be in a class of TypedValue (containing other TypedValues). Each TypedValue has a Domain and an immutable object as value.

3.6.3 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

[11] Update statements also have SET for query columns but the mechanism is different (it uses a TransitionRowSet which can be accessed by TriggerActivations).

[12] In complex cases a left to right order of evaluation of columns sis not possible. Pyrrho repeatedly scans the columns until all are evaluated or a circular dependency is detected.

Query	Context	A base class for query processing. Has a cached RowSet. Uses left-to-right accumulation of columns and aliases (defs)
TriggerActivation	Activation	For trigger execution

Activations have BTree<string,TypedValue> variables, mapping identifiers to structures, queries and subdocuments; and Queries have BTree<Ident, SqlValue> defs for tracking aliases.

Context has the following virtual Methods:

Method	Returns	Signatures	Description
Assign	Void	Ident, Value	Enters variables, overridden in TranisitionContext
SetMatch	Void	PColumn, SqlValue	Adds filters when overridden in the Query class
Lookup	SqlValue	Ident	Finds a SqlValue by name/alias in the current context. Columns in Queries can be defined using complex expressions which are then referenced by aliases.
LVal	Target	Ident	Finds a Target (useful for left hand side of assignment, e.g. for update or local variables
Eval	TypedValue	Ident	
Eval	TRow		Evaluates a query or this for a UDT,

3.6.6 RowSet

RowSets deliver the result of query processing. RowSets are immutable and shareable. Once analysed, a Query has a RowSet for its result. The abstract interface offered is as follows:

bool building	Some rowsets (grouped, ordered etc) require building
RowBookmark First()	Compute the first row of the rowSet
Domain keyType	The keyType of the RowSet
int limit	
BTree<long,TypedValue> matches	A set of filters (used when the columns have defpos information)
RowBookmark 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
Domain rowType	The nominalDataType of the rowSet is normally that of the query, but can differ for select-single subqueries and for RESTViews
RowSet(Query,Domain,OrderSpec)	Constructor
Schema(Domain,int[])	Compute client-side flags for the rowSet
TargetName(Participant)	Return the client-sident rowset name (e.g. the table name)
int skip	

There are numerous subclasses of RowSet.

3.6.10 RowBookmark

This is an abstract and immutable class for traversing rowSets. All RowSets offer a First() that returns an RowBookmark at position 0, or null. RowBookmarks are immutable. The interface offered includes the following:

RowBookmark Next()	*Abstract:* Returns a bookmark for the next row, or returns null if there is none
RowBookmark PositionAt(pos)	Returns a bookmark for the given position, or null if there is none.
TRow Value()	The value at the current position. *Uses the indexer below to compute the whole row and caches the resulting TRow.*
TypedValue this[int i]	*Abstract:* The value of the ith column of the row.

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

3.6.4 Activation

Activation is a subclass of Context.

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>	

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[13]:

- 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	

[13] 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 Parsing

The parser retains the following data:

- The Lexer
- The current token in the Lexer, called tok (1 token look ahead)

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.

During parsing many routines record the positions where Ident lexemes occur, in a special tree called refs in the root context. This is because role-based conceptual models can rename database objects. For example, the View object maintains a snapshot of the owner's role at the time of definition, and this is used in the From constructor when the view definition is applied: once the From has constructed the required RowSet the names are updated to reflect the current metadata. System tables such as Role$View modify the display of SQL view definitions etc to reflect role-based renaming of objects that may have occurred since object definition.

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 (inline View, or Table expression)
QueryExpression ParseQueryExpression(t,sql)	Parse a QueryExpression for execution (for a View)

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.obinfos), 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.

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[14].

The query rewriting uses a Replace algorithm that performs a left-to-right scan of the query context. This scan is for replacing a single object reference (uid), and affects both the expression objects and the rowType. Queries can also be replaced (with the same uid) when conditions and filters are moved within the resulting structure. Either way, all other parts of the query will be updated as and when queries and expressions are changed. 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	AUTHOR	an Index (ID) for 22
90	CHAR	
103	ANAME	for 22
146	BOOK	a Table
156	ID	for 146
180	BOOK	an Index (ID) for 146
202	AID	for 146
228		an Index (AID) referencing 22
247	TITLE	for 146

[14] In general, we can't replace this dotted expression simply with c, 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'
```

When the parser reaches b.title, it parses it 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. The parser builds a SqlValueExp structure whose operator is DOT, and left and right are undefined references. After also noting the alias. at position 24 the context has

#7=QuerySpecification #7 ObInfo _ (#9(SqlValue B #8.SqlValue TITLE #10) as C) from ?

The context notes the definitions of both identifier chains B.TITLE and C as #9.

At position 26, we lookup author in the Role, and find it is a table (with file position 22).

#26=From AUTHOR #26 ObInfo (SqlCol ID 47(33 INTEGER) Col: 47, SqlCol ANAME 103 (90 CHAR) Col: 103) Target=22

We add a From reference to the query, and it is followed by an alias which is noted in the context (not in the parse tree). The context notes the definitions of both AUTHOR and A as #26.

The comma at position #35 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:

#36=From BOOK #36 ObInfo (SqlCol ID 156 (33 INTEGER) Col: 156, SqlCol AID 202 (33 INTEGER) Col: 202, SqlCol B.TITLE #9 Alias C Col: 247) Target=146)

The columns that are not referenced in the query retain their file uids. The comma means a table expression is constructed to join the tables:

#34= {JoinPart #34 ObInfo _ (SqlCol A.ID 47 (33 INTEGER) Col: 47, SqlCol ANAME 103 (90 CHAR) Col: 103, SqlCol B.ID 156 (33 INTEGER) Col: 156, SqlCol AID 202 (33 INTEGER) Col: 202, SqlCol B.TITLE #9 Alias C (90 CHAR) Col: 247) #26 CROSS join #36

Parsing moves on to the WHERE clause, so after identifying the columns we have the conjunction id=aid and b.title>'M', as:

{(#53= #53(SqlCol ID #50 (33 INTEGER) Col: 47= SqlCol AID #55 (33 INTEGER) Col: 202),

#65= #65(SqlCopy B.TITLE #64 from #9>M))}

and the above from and joinPart structures are updated to note that A.ID is #50 and AID is #55. The combination for the join with the where condition is a TableExpression

#21= {TableExpression #21 ObInfo #21 (SqlCol B.TITLE #9 Alias C (90 CHAR) Col: 247) Where:(#53(SqlCol ID #50 (33 INTEGER) Col: 47= SqlCol AID #55 (33 INTEGER) Col: 202) AND #65(SqlCopy B.TITLE #64 (90 CHAR) copy from #9>M)) From: #34 }

The combination of the select list and the TableExpression is a QuerySpecification:

#7 {QuerySpecification #7 ObInfo #7 (SqlCol B.TITLE #9 Alias C (90 CHAR) Col: 247) from #21}

During traversal of the final result, the typedvalues corresponding to all of the column references are temporarily placed in the context, in order to compute the columns in the result: and in this case there is only one per row, for example {(#9=Nostromo)}

The target of the FROM might be a table, a view, or subquery. In all cases we first compute the resulting rowType, first by applying access permissions, and then by using the correlation if any., in order to resolve references during parsing.

As this example indicates, once the parser reaches the end of the query, the query structure is replaced with a new one in which all select items, while retaining their unique uid, now contain their target and domain details. This stage was formerly called the Selects stage of analysis. The new select item objects can have some name and alias properties that can be useful in diagnostic information, but in v7 these are never referred to by the database engine. The query structure resulting from the parse is retained, so that any query string is only ever examined once (with the definer's role). Sql objects and their uids and targets are not stored in the database file.

There are several further stages of analysis:

Stage	Description
Conditions	conditions analysis which examines which search and join conditions can be pushed down towards the base tables.
Ordering	ordering analysis which looks not only at the ordering requirements coming from explicit ORDER BY requests in the SQL but also at the ordering required during join evaluation and aggregation. Actual ordering is done by the RowSet when traversal begins.
RowSets	When query definitions from views are incorporated, significant rebuilding of the overall query can occur,

All analysis routines have a return value consisting of the new version of the query. Some result in rebuilding of the overall query, with the help of the Context structure, which contains a mutable catalogue of immutable Queries and SqlValues for the current parse, arranged in order of depth. Any change is propagated through this structure so that all of the parts are kept consistent.

RestViews add many further requirements. See Section 9.3.

6.2 RowSets

RowSets give access to a RowBookmark for traversing a set of rows. From June 2016 all row sets and bookmarks are immutable data structures. From November 2017, data is not collected for RowSets until traversal begins. From v7, the RowBookmark and its TRow value cannot be stored in the Query, and must be accessed from the Context (and the Context cannot be a base class for Query).

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 the 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 behaviour of nested row traversals is a different, and entirely local, matter, but in version 7 their handling is facilitated by the use of the SqlColumn class since a table column can occur in more than one current rowset. Obviously, the set of values visible in this top context is not neecessarily the same as any query rowType, as it is the union of the current rows and local variables of ongoing traversals and computations.

In aggregated and windowed traversals, 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 RowBookmark, even through all such will use the same transaction and context.

Selection using a search condition can be done by the context (e.g. a From or Query) holding a where condition: so that typically the implementation of Next() uses a loop to moving not to the next physical row, but to the next row satisfying the search condition. If the search condition involves only the values of the primary index (or some other index), then whether a particular is included or not can be determined from the index, without even the need of examining the actual data values in the database file.

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 order, a new rowset is constructed in memory for the data during traversal. 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.

6.3 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 code makes extensive use of partial evaluation, so that for example Eval() expressions (e.g. are affected by movement of an enumerator (e.g. enu.MoveNext()). The effect of this relatively unusual programming technique is to mimic the behaviour of SQL expressions: thus a RowSet is a SqlRow (a set of Columns) and has an Enumerator (which modifies the values of the row columns as it moves).

This matches well with the top-down approach to parsing and query processing that is used throughout Level 4 of the code.

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 RowBookmark 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 RowBookmark.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 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)
[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]
```

Previous steps in the test have added a row each in these two tables. In the transaction log these objects have been recorded as

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	
264	Record 264[138]	145=0[29]
506	Trigger RUAB	From=513 On=22 Action:(0=534 (0=UpdateSearch XB SET Vbl: TOT Col: 145 Val: 567(562(TOT Col: 145- Old B Col: 43 TRS:22)+ B 568 Col: 43), 1= D Col: 99=changed)) OldRow=MR NewRow=NR)
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 (the text is given above), and the compiled contents now in memory are shown above from a debugger display.

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

{ 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)