An Introduction to the Source Code of the Pyrrho DBMS

Malcolm Crowe, University of the West of Scotland

www.pyrrhodb.com

Version 7.03 (March 2023)

1. Introduction

Wer immer strebend sich bemüht,
Den können wir erlösen.
(J. W. v. Goethe, Faust)

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.

I am proud to let the community examine this code, whose successive versions have 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 wherever possible. 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 2022 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.

The rest of this Introductory section is adapted from a tutorial given by Malcolm Crowe and Fritz Laux at IARIA's DBKDA 2021 conference.

---

[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).

1.1 ACID and Serializable Transactions

In this work we offer some general methods that can be used in DBMS implementations to enforce struct atomicity, consistency, isolation and durability. The Pyrrho experiment itself provides a proof of concept for these ideas.

Our starting point is that full isolation requires truly serializable transactions.

All database textbooks begin by saying how important serializability and isolation are, but very quickly settle for something much less.

If we agree that ACID transactions are good, then:

- First, for atomicity and durability we should not write anything durable until (a) we are sure we wish to commit and (b) we are ready to write the whole transaction.
- Second: before we write anything durable, we should validate our commit against the current database.
- Third, for isolation, we should not allow any user to see transactions that have not yet been committed.
- Fourth, for durability, we should use durable media – preferably write-once append storage.

From these observations, it seems clear that a database should record its durable transactions in a non-overlapping manner.

- If transactions in progress overlap in time, they cannot both commit if they conflict: and if they don't conflict, it does not matter which one is recorded first.
- The simplest order for writing is that of the transaction commit.
- If we are writing some changes that we prepared earlier, the validation step must ensure that it depends on nothing that has changed in the meantime, so that our change can seem to date from the time it was committed rather than first considered.
- Effectively we need to reorder overlapping transactions as we commit them.

These few rules guarantee actual serialization of transactions for a single transaction log (sometimes called a single transaction master). It obviously does not matter where the transactions are coming from.

But if a transaction is trying to commit changes to more than one transaction log, things are very difficult (the notorious two-army problem). If messages between autonomous actors can get lost, then inconsistencies are inevitable. The best solution is to have a DBMS act as transaction master for every piece of data For safety any transaction should have at most one remote participant, and more complex transactions can be implemented as a sequence of such one-way dependency transactions.

1.2 The transaction log

The Appendix contains four demonstrations that were included in the [tutorial](#) in the [DBKDA 2021 Program (iaria.org)](#). The first was about the transaction log. In Pyrrho the transaction log defines the content of the database (it is the only thing stored on disk).

In this demonstration, we will see how every Transaction T consists of a set of elementary operations e_1, e_2,.. , e_n .

Each operation corresponds to a Physical object written to the transaction log

Committing this transaction applies the sequence to the Database D. In reverse mathematical notation

$(D)T = (..((D)e_1)e_2)..)e_n$
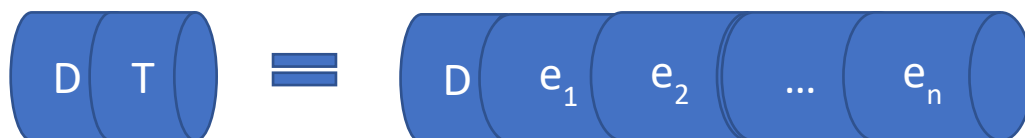
If we think of a transaction commit as comprising a set of elementary operations e, then the transaction log is best implemented as a serialization of these events to the append storage. We can think of these serialized packets as objects in the physical database. In an object-oriented programming language, we naturally have a class of such objects, and we call this class Physical.

So, at the commit point of a transaction, we have a list of these objects, and the commit has two effects (a) 7 appending them to the storage, (b) modifying the database so that other users can see. We can think of each of these elementary operations as a transformation on the database, to be applied in the order they are written to the database (so the ordering within each transaction is preserved).

And the transaction itself is the resulting transformation of the database.

Every Database D is the result of applying a sequence of transactions to the empty _system Database D0.

Any state of the database is thus the result of the entire sequence of committed transactions, starting from a known initial database state corresponding to an empty database.

The sequence of transactions and their operations is recorded in the log.

The first demonstration illustrates this process by using a debugger to show successive states during a transaction commit.

The Pyrrho manual gives details of the file format used the Pyrrho database and lists all of the Physical records used by Pyrrho. Full details are also provided in section 3.4 below.

1.3 DBMS implementation principles

- If we agree on a globalization strategy, then the DBMS should be neutral, and not specific to a particular machine, platform, or locale.
- A database created on one machine, platform or locale should be usable on another.
- The DBMS should not impose arbitrary size limits on strings, number of columns etc.
- Any that are imposed should be huge (Pyrrho uses 2^{60} as a useful size for such limits).
- If we agree that security is important, then we should use operating system authentication and no other options. Users should not be simply allowed to say who they are.

Pyrrho DBMS has always had these goals, and a full feature set including stored procedures, structured types, triggers, and views.

But it turned out that its consistency and isolation in its implementation were not good enough, and it was easily outperformed by StrongDBMS, a much simpler system. In an artificial test with high concurrency and serializable transactions, we found that all DBMS were outperformed by StrongDBMS.

So, in Pyrrho v7, the aim was to re-implement Pyrrho using a lesson learned from StrongDBMS, that in situations of high transaction concurrency it is best to use shareable, immutable data structures, for as many internal structures as possible.

The implementation has been progressing steadily, it is still in at the alpha stage, but anyone can see the progress that has been made, as the source code is on github.

1.4 Shareable data

Many programming languages (including Java, Python and C#) currently have shareable implementations of strings.

Specifically, strings in Java, Python and C# are immutable: if you make a change, you get a new string, and anyone who had a copy of the previous version sees no change.

In these systems, it is illegal to modify a string by assigning to a character position instead you need to use a library function.

The addition operator can be used in these languages to create a sum of strings. This is basically the model for shareable data structures.

For a class to be *shareable*, all fields must be read-only and shareable. Constructors therefore need to perform deep initialisation, and any change to an existing structure needs another constructor. Inherited fields need to be initialised in the base (or super) constructor, maybe with the help of a static method.

This is useful for databases because databases share so many things: predefined types, properties, system tables. For example, all databases can share the same starting state, by simply copying it from the _system database.
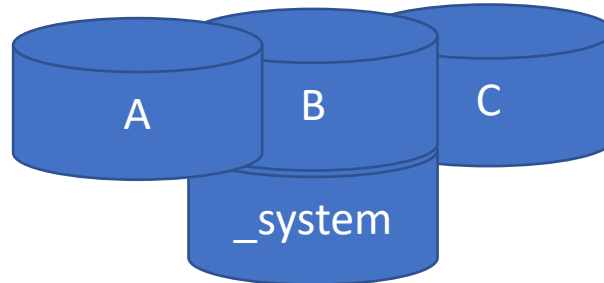
Even more importantly all transactions can start with the current state of the database, without cloning or separately copying any internal structures. When a transaction starts, it starts with the shared database state: as it adds physicals, it transforms. Different transactions will in general start from different states of the shared database.
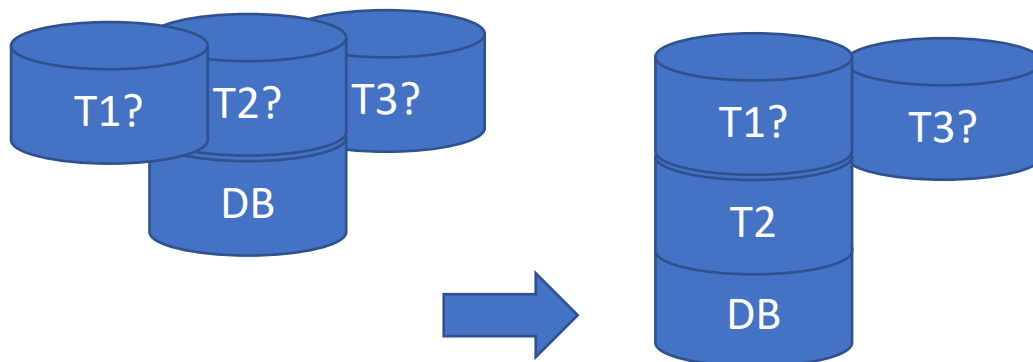
 In the above picture, we know what the database DB's state is. Each of concurrent transaction steps T1, T2, and T3 are, if committed, will create a new version of DB (for example (DB)T2.) Because of isolation, from the viewpoint of any of these transactions, they cannot know whether DB has already been updates by another transaction (in which case, they may no longer fit on the resulting database). In particular, after T2 commits, T1 and/or T3 will possibly no longer be able to commit.

However, if T1 was able to commit before, then it will still be able to commit provided it has no conflict with T2's changes.

1.5 Transaction conflict

The details of what constitutes a conflicting transaction are debatable. Most experts would agree with some version of the following rules:

- T1 and T2 will not conflict if they affect different tables or other database objects
    - And only read from different tables
- But we can allow them to change different rows in the same table
    - Provided they read different specified rows
- Or even different columns in the same row
    - Provided they read different columns

The first rule is sound enough, although the condition on reading is very important: we would need to include things like aggregation in our definition of reading. The first rule is also very easy to implement, especially if tables are shareable structures, as a simple 64-bit comparison is sufficient!

For the other rules, we would need to be very clear on what is meant by a "specified row", and the non-existence of a row might only be determined by reading the whole table.

1.6 Transaction Validation

In the second demonstration in the Appendix, we look in detail at the Commit() method for a transaction, and the detection of conflicts.

At the start of Transaction Commit, there is a validation check, to ensure that the transaction still fits on the current shared state of the database, that is, that we have no conflict with transaction that committed since our transaction started.

We will see that during commit of a Transaction T, we do a validation check. It ensures that the elementary operations of T can be validly relocated to follow those of any transaction T´ that has committed since the start of T . T planned
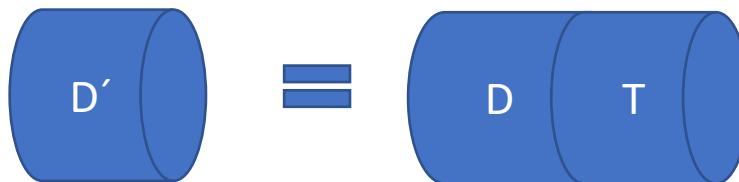
But now we have
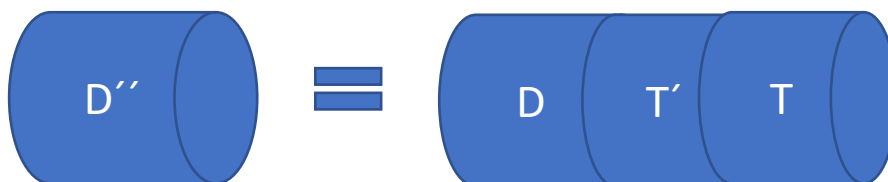
Relocation amounts to swapping the order of the elementary operations e_i that comprise transaction T and T'. Two such cannot be swapped if they conflict. E.g. They change the same object (write/write conflict). The tests for write-write conflicts involve comparing our list of affected physicals with those of the other transactions.

There are also tests for read/write conflicts between T and T´. For checking read-write conflicts, we collect "read constraints" when we are making Cursors. This algorithm will be simplified from September 2022 and will use two lists of uids accumulated during the transaction: one for columns that have been read, and the other for specific rows that have been read (indexed by table). For any table involved in the transaction we check that no column we have read has been updated by a more recent transaction; and if only a few specific rows have been read, we can limit the test to those rows. Rows and columns in base tables are identified by 64-bit uids that are unique in the database.

1.7 Shareable structures

The next question is how best to implement shareable data structures.

In a programming language based on references, such as Java, or C#, we can make all fields in our structure final, or readonly. Then any reference can be safely shared, and all fields might as well be public (unless there are confidentiality issues).

If all of the fields are, in turn, also known to be immutable, then there is no need to clone or copy fields: copying a pointer to the structure itself gives read-only access to the whole thing.

For example, if the Database class is shareable, and b is a database, then a:=b is a shareable copy of the whole database (we have just copied a single 64-bit pointer). Such an assignment (or snapshot) is also guaranteed to be thread-safe.

Pointers to shareable structures are never updated but can be replaced when we have a new version to reference. If this new version has some changed fields, it is perfectly fine to continue to use the same pointers for all the unchanged fields

1.8 Transformation: adding a node

When we add a field located deep in a shareable structure (e.g. a node to a shareable tree structure), we will need to construct a single new node at each level back to the top of the structure. But the magic is that all the other nodes remain shared between the old and new versions.

(a) the original shared tree,
the position of modification is marked

(b) the path to the position of modification is "deshared",
new nodes are thicker, shared nodes are shaded

The picture (from Krijnen and Mertens, Mathematics Centre, Amsterdam, 1987) shows a tree with 7 leaves (that is, a tree of size 7), and updating (that is, replacing) one leaf node has resulted in just 2 new inner nodes being added to the tree. This makes shareable B-Trees into extremely efficient storage structures.

In tests, we see that for a suitable minimum number of child nodes in the B-Tree, the number of new nodes required for a single update to a B-Tree of size N is O(logN), and experimentally, this means that for each tenfold increase in N, the number of new nodes per operation roughly doubles.

Note that we also get a new root node every time (this avoids wearing out flash memory).

1.9 The choice of programming language

It is helpful if the programming language supports:

- readonly directives (Java has final)
- Generics (Java has these)
- Customizing operators such as += (not Java)
  - Because a+=b is safer than Add(a,b)
  - Easy to forget to use the return value a=Add(a,b)
- Implies strong static typing (so not Java)
  - Many languages have "type erasure"
- Also useful to have all references nullable

So I prefer C#, which now has been around for 19 years. Java and Python have been with us for over 30 years.

However, C# provides no syntax for requiring a class to be shareable: specifically, there is no way of requiring a subclass of a shareable class to be shareable. It will cease to be shareable if it has even one mutable field.

1.10 Shareable database objects

What data structures in the DBMS can be made shareable?

- Database itself, and its subclass, Transaction.
- Database Objects such as Table, Index, TableColumn, Procedure, Domain, Trigger, Check, View, Role
- Processing objects such as Query, Executable, RowSet, and their many subclasses;
- Cursor and most of its subclasses.
- TypedValue and all its subclasses

All of these can be made shareable.

Context and Activation cannot be made shareable because in processing expressions we so often have intermediate values.

Also, something needs to access system non-shareable structures such as FileStreams, HttpRequest.

And Physical and its subclasses are used for preparing objects for the database file, so cursors that examine logs are not shareable.

1.11 An implementation library: first steps

A fundamental building block in many DBMS implementation is the B-tree. In Pyrrho BTree<K,V> is a sort of unbalanced B-tree. It has a += operator to add a (key,value) pair, and a -= operator to remove a key.

BList<V> is a subscriptable subclass where K is int. It is much slower than BTree, because it partially reindexes the list starting from 0 on insertion and deletion.

Both of these structures can be traversed up and down using shareable helper classes ABookmark<K,V> and methods First(), Last(). The ABookmark class implements key(). value(), Next() and Previous().

These classes and their subclasses are used throughout the DBMS implementation. They are shareable provided K and V are shareable. If the classes K and V are not themselves shareable, for example if one or both is the object class, a tree will be shareable provided all of its contents (the nodes actually added) are shareable. At least, it is easy to ensure that all public constructors only have shareable parameters.

For convenience, Pyrrho uses a potentially non-shareable base class **Basis**, whose only field is a **BTree<long,object>** called mem. It has an abstract method New which can be used to implement the += and -= as effectively covariant operators on subclasses, and these can be used to change the properties on a database (of course, by creating a new one).

1.12 DBObject and Database

DBObject is a subclass of **Basis**, with a public readonly field called defpos, of type long. The defpos (defining position) acts as a uid or unique identifier for every object in the database. As described in section 2.3 below, the range of values of long is divided into ranges for database objects depending on their lifetime: committed objects have the same lifetime as the database (e.g., the SQL delete statement can unlink one or more objects but they remain in the log). There are many subclasses of DBObject, described in this booklet.

Database is also a subclass of **Basis**, and it uses the mem field with positive uids to store all of database object it contains. (Entries in mem with negative uids are used for propert ies of the database, as described in section 3.5.2.) The Database class also has two static lists: databases, and dbfiles. The first is the set of databases indexed by database name, and the second is a corresponding this of **File** objects.

Then there is a method called Load() to build the database from the transaction log file, and Commit() which writes Physical record durably to the log.

1.13 Transaction and B-Tree

We can use the B-Tree concept to model how the transaction Commit process works.
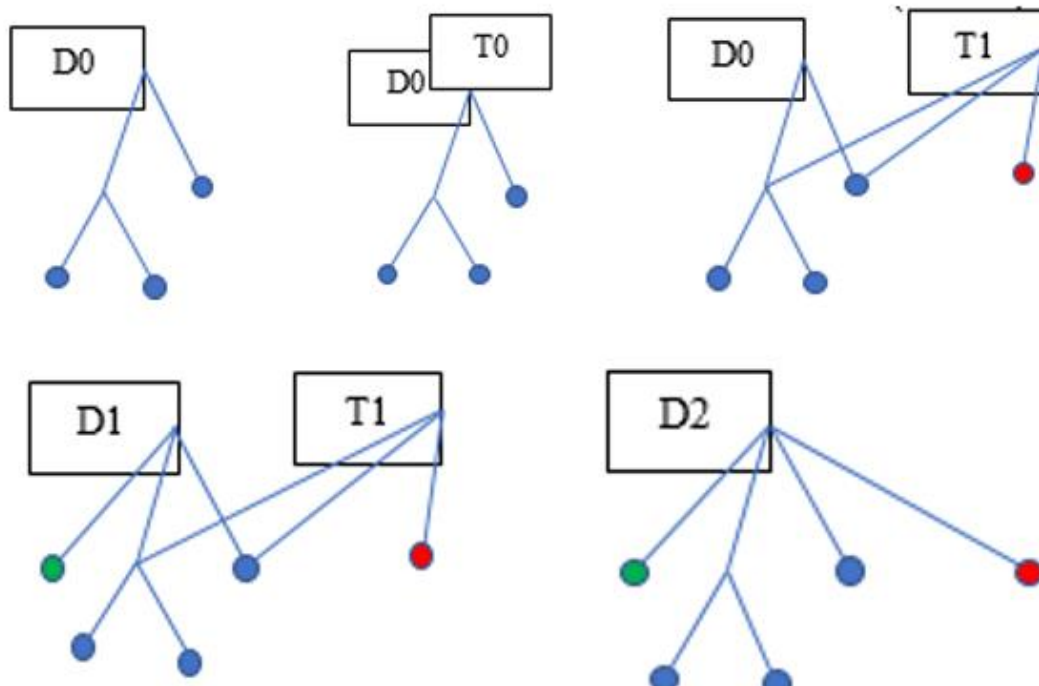
1. Suppose we have a database in any starting state D0.
2. If we start a transaction T0 from this state, initially the T0 is a copy of D0 (i.e. equal pointers).
3. As T0 is modified it becomes T1, which shares many nodes with D0, but not all.
4. D0 also evolves as some other transaction has committed, so D1 has a new root and some new nodes.
5. When T1 commits, we get a new database D2 incorporating the latest versions of everything.

1.14 RowSet review

Many other DBMS describe their process of "Query optimisation". The result of parsing data manipulation language in Pyrrho v7 is not an optimised Query but a possibly updatable RowSet. As with other immutable objects, RowSets must have a full set of properties on construction.

So instead of Query Optimisation, Pyrrho v7 has RowSet review. Changes to Rowset objects, or to the set of properties proposed for a new RowSet, are recursively applied to source rowsets. Naturally any change to a source gives a new source. Operations such as insert, update and delete are passed down to base local or remote tables, while ordering and filtering can often be performed more efficiently on source rowsets, and to help with this, equality filters ("matching") are also passed down to sources.

For example, ordering specifications can include complex expressions. Provided there is only one source, the order specification can be passed down to the source if it defines the operands of ordering expression. The source may already be known to have the correct order (for example, it may be the result of a merge or join). If the source is remote, the ordering specification can be passed to the remote source: while if the ordering is all on simple columns, integrity constraints on base tables may be able to supply the required ordering. Otherwise, an OrderedRowSet is constructed, to sort the rows prior to traversal.

As another example, during RowSet Review we replace selection from views and joins by selection from underlying tables. In the same way, inserts, updates and deletes can be applied to many rowsets by passing the operations down to individual table sources. Passing transformations down to sources is usually prevented where the columns are complex expressions.

In general, aggregations are already parsed before the from clause provides domain information for columns exposed by the referenced table(s) and/or view(s), where conditions are appended to the result, and grouping and ordering added. The process requires sufficient care to avoid ambiguity when importing

predefined constructs, and to take appropriate account of equivalences established by join conditions and similarity of expressions. See the worked examples in sections 6.8 and 6.9.

1.15 Integrity Constraints

The efficiency of a database depends on the maintenance of indexes to the rows of tables, and this leads to the requirement for the definition of table keys: keys are sets of columns that may be primary, unique or foreign. In SQL indexes are consequences of such key definitions and are not separately created.

Once a key is defined for a table, the database must maintain the index through any modifications to the table. Generally, a modification such as insert, update or delete will be restricted if it would lead to an error in the index, but the modification may instead request some automated action to maintain the consistency the index (for example, by modifying or deleting some other data). In this way, modifications may cause cascades of changes to the database, and these must be included in the current transaction. The transaction cannot be allowed to commit if any of these consequential actions cannot be committed at the same time.[2]

1.16 Roles, Security, Views and Triggers

SQL defines a security model that distinguishes the roles of definer (or owner), administrator and user of the database and its objects. Generally, the definer of an object has full permissions on a new object but may grant permissions (and/or administration rights) to another user.

A grant of permissions by one Role to another Role does not create any relationship between the Roles: and these can evolve independently. Similarly, revoking privileges from a Role takes no account of how these privileges were acquired.[3]

As a result of this security model, for example, the columns returned by a rowset will depend on the permissions of the current user on the objects being accessed. A consequence of the security model is that SQL supports the definition of views, which allow for the definition of rowsets granted to users, and triggers, that allow for the definition of actions consequential on a modification. Views and triggers must execute with the permissions of their definers.

If a full transaction log is being maintained, log entries must identify the current user for any modification to the database. Modifications to a database can be made subject to check constraints and triggers. constraints ensure the maintenance of desired logical relationships (uniqueness, etc). The PTransaction record in the log achieves this by recording the current user and role for adjacent log entries in a transaction.

In a large system the set of users may become large, and it becomes practical to define database roles that possess a set of permissions, so that database users can be granted the use of such a role, rather than acquiring permissions one by one on the (possibly large) set of database objects they need to use. This model is specified in the SQL standard and supported in Pyrrho subject to a simplification mentioned in the next paragraph.

To implement this, recent versions of the server manage a subclass of Physical called Defined. Defined objects have a definer, which is picked up from the PTransaction record mentioned above, and zero or more ObInfo structures to give the privileges of the definer and other users as specified on creation (subsequent changes to this list are managed using Grant and Revoke, which are not themselves Defined objects). Defined physical objects with no ObInfo list include Domain and UDType: this is a simplification of the SQL standard security model[4].

1.17 Compiled database objects

SQL allows database objects to contain programable check constraints and triggers and provides a programming language for these that supports structured data types, stored procedures, data conversions and condition handling. The DBMS implementor must provide for these.

---

[2] See the last two paragraphs of section 4.23.2 of the SQL standard ISO 9075-02 (2016).

[3] This is a departure from the SQL standard ISO9075 and documented in section 7.12 of the Pyrrho manual.

[4] The columns and methods of a UDType are nevertheless subject to ObInfo privilege information.

SQL defines a computationally complete programming language for this purpose, and this is implemented by allowing Contexts to form an execution stack, in which each Context on the stack has permissions depending on the definer of the statements they are executing, is dealing with a set of specific database objects, with its own set of intermediate results (a heap), and may have handlers for exception conditions.

It is an ambition of this version of Pyrrho that such programmable elements are compiled on definition or database load. Thus, many database objects, including views, procedures, check constraints, and triggers, are held as shareable *compiled* objects in memory, so that an instance of the compiled object can be quickly loaded into the execution context when the compiled object is referenced. See section 3.4.2.

Compiled is a subclass of Defined. Compiled physical objects contain an SQL version of the programmable elements, which are parsed on load.

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 the five layers (levels), 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 apart from Exception classes[5], including Bookmarks for traversing them, are immutable and shareable[6].
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. Physical objects are volatile so that shareable structures cannot contain them.
Pyrrho.Level3	Logical Database	Database.cs, Transaction.cs, Value.cs and classes for database objects. All classes in Level3 (except Scanner and Signal) are immutable and shareable.
Pyrrho.Level4	SQL processing	RowSet.cs, Parser.cs etc. All RowSets and most Cursors[7] are immutable and shareable, and give access to the version of the RowSet that created them.
Pyrrho.Level5	TypedGraph processing	Create and Match for graphs. TGraph, TNode and TEdge are immutable and shareable TypedValues.

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

---

[5] System.Exception is not shareable abd so all its subclasses are not Shareable.

[6] In general, Shareable is not a heritable property: shareable classes can have mutable subclasses. However, BTree<K,V> and BList<V> and their bookmarks are shareable if K and V are; otherwise they are shareable provided all objects placed in them are shareable. Non-shareability and non-immutability are heritable. Shareable classes with no internal subclasses could be made sealed, but this adds nothing for internal classes.

[7] Cursors used to examine the transaction log directly (LogSystemBookmark and its subclasses, and SysAuditBookmarks) are not shareable.

together the physical records that refer to the same defining position but, from version 7, Pyrrho maintains the current state of base table rows in memory (using the TableRow class), and does not follow such non-scalable trails

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

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

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

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

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

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

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

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

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

the current input line. This is the responsibility of Server.Execute(sql) and SqlHTTPService parser calls.

2.3 Data Formats

The data file format and SQL source formats are fully described in the Pyrrho manual. As noted above, both of these are independent of location, culture, operating system and machine architecture. In this section we discuss the implementation for the above formats and describe the other data formats used in the C# engine and for client-server communication.

2.3.1 Implementing the data file formats

From the Pyrrho manual, we recall that the Pyrrho data file uses a custom byte encoding for fixed format numeric data including integer, with up to 256 bytes (roughly 2040 bit) precision. The Integer class that implements this format is described in section 3.2.1 below, In the Pyrrho engine, the int data type is optimised so that integers that will fit in 64 bits are use the C# long format as implemented by the machine architecture in use. Numeric data is implemented as an Integer mantissa and short integer scale. Division of Numeric defaults to 12 decimal digits of precision.

For approximate (real) data format, the C# implementation of double is used.

All string information is implemented using Unicode, and UTF-8 encoding. In accordance with the above design principles, the data file uses the neutral culture for these strings and ignores any requirement to use national data formats (NCHAR etc), or normalize string representations. Explicit collation instructions in SQL code and for domains and columns are honoured for evaluation of expressions (e.g. in constraints).

Timestamps and other date and time formats (other than INTERVAL) are implemented using the C# DateTime, Timetamp, TimeSpan implementation. Intervals have a special class in the implementation, because in SQL the year-month and day-second fields cannot be mixed[8].

SQL code in the data file (constraints, routines, views etc) is represented in SQL source form as strings.

2.3.2 Implementing SQL formats

The SQL format from ISO9075 (2016) is followed strictly in the Lexer class and the Parse methods for Domain. This includes the definition of the format for binary data. This requirement gives a lot of difficulty for experts transferring from other database providers.

2.3.3 Formats in the in-memory data structures

The DBObject class and its subclasses are described in sections 4 and 5 of this document. During a transaction, the engine is working with a mixture of committed objects and uncommitted DBObjects. Some DBObjects such as Table, or TableColumn correspond to Physical objects such as PTable, PColumn and once committed have defining positions representing their location in the data file. Before commit, they are identified by uids as described in section 2.4.

Such DBObjects are shared with any transactions that use them: we note that a single query may have multiple references to a table, associated with different aliases and embellished with where-conditions, ordering etc. During computation of rows in rowset traversal, several rowsets will traverse the rows of shared tables. To deal with this, each reference to a Table is given a TableRowSet (an instance) whose defining position is given by a uid in the heap range, much as each invocation of a procedure results in an instance of its formal parameters and local variables. Then the Context object (section 3.6.1) keeps track of the current instances are their cursors.

Compiled objects such as views, routines, constraints etc generally contain DBObjects within them that do not correspond with Physical objects. are parsed on definition and on load from the database file: they are placed in the framing field of their parent DBObject, and their uids are handled quite differently (see section 3.4.2). During parsing, instead of using lexical positions, they are given sequential uids in a special range, and when instanced, these uids are replaced by heap uids.

---

[8] See section 4,6,1 of SQL 9075: An indication of whether the interval data type is a year-month interval or a day-time interval.

The database file is read by the engine on the first access by a client, and the Database object is built during this Load process. Transaction commits result in Physical objects being appended to the data file, and the associated committed objects installed into the shared Database. Transactions may also access the database file directly for (a) reading the log (b) appending audit records.

All these three forms of access lock the relevant database file. The server is multi-threaded, so that all transactions continue except for those wishing to access this particular database file in one of these three ways.

2.3.4 Client-server communications

The Pyrrho protocol is described in the Pyrrho manual, and the implementation use the same encodings as described above for the database file. The PyrrhoLink.dll API is also described in the Pyrrho manual: it is based on ADO.NET but enforces proper threading behaviour. Some of the API methods use a Document format based on JSON/BSON.

The server also provides a Web service, which follows REST protocols, and uses Http Basic authentication. There is some provision for HTTPS but this has not been tested.

2.4 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 types and system tables. Every database structure includes this immutable information. No other cross-thread access is possible in Pyrrho.

Unique identifiers[9] 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 $0..2^{62}-1$ (positions of physical records) 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 (proposed physical records), whose uids are in the range $2^{62}..5\times2^{60}$, and denoted !0,!1.. for convenience. Each transaction starts this range afresh (so that its first proposed object is always !0) because every transaction has its own context. These uids are retained until the transaction is committed or rolled back and form a natural stack[10]. On commit, the transaction's new physical records are serialised to the data file, whereupon their 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 so that each reference gets a new uid and new column uids[11].

So far so good. But for example, not all uncommitted objects have the same lifetime. The activation context for a procedure may have local variables. The execution heap is initialised on each transaction step. Prepared statements are connection based and so persist beyond the end of a transaction.

---

[9] Unique within the context. Different contexts may have different versions of the database, but always start with the latest committed version of the database.

[10] This stack is useful for recovery in the implementation of SQL exception handling.

[11] During rowset traversal, cursors associate column uids with their values in that row.

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). Thus, all identifiers that occur in the SQL are replaced during parsing with uids in this range as allocated on the first occurrence in the command text. Columns not referred to will be given temporary uids from the nextHeap range, described below: they cannot use their defining position as this might conflict via a separate reference to the table in the SQL (subqueries, views etc).

As a result of the above considerations, the replacement of identifier-based references with uids proceeds from left to right during parsing. When source identifiers are resolved to column references, a lexical id is given to the column reference. If the resulting DBObjects are serialised to the database (stored procedures, triggers), the source code only is saved in the transaction log, and instead of reparsing, each such lexical or heap uid is replaced by a physical or (respectively) statement uid based on the permanent file position of the lexeme.

Several database object types (e.g. Procedure, Check, Trigger, View) define executable code. The physical records in the transaction log record their definition in source form and are called Compiled objects (see sec 3.4.2). During the load phase the executable fragments are parsed, and the resulting executable structures (which have a mixture of lexical[12] and heap uids) are relocated so that their uids are in a contiguous block in the range $6\times2^{60}..7\times2^{60}$-1, denoted `0,`1.. for convenience. Such objects are immutable and shareable, so that any instances (with their view definitions, rowsets etc) will be given new uids in the heap range.

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

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

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

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

- File positions: audit requires asynchronous writing to the transaction log during a transaction. Such asynchronous writing cannot occur during the transaction commit as this process occupies the thread.

- New Physicals: can be created because of triggers and cascades at various points during a transaction step. There is a dependency field that helps to ensure that serialisation during commit takes place in an orderly way.

- Storage for compiled statements is local to a database, is immutable, and can be used by successive steps in a transaction and successive transactions.

- The prepared statement storage is semi-persistent and shared among sequential transactions in a single connection independently of commit.

- The heap range is for values and objects local to the current transaction step. Procedure activation extends the heap in the usual way with new local variables, and return values are moved to the previous heap.

---

[12] Lexical position is sometimes needed for resolving identifier chains, so is handled for SqlValue, ForwardReference and SelectRowSet.

- Since all shared objects may be referenced in several places in a command, instances are created for each reference to keep their properties separate. Instances are always placed on the heap.

- Compiled objects (such as constraints, procedures, views, prepared statements) have a framing field, which lists shared objects they reference and their result object, See section 3.4.2 below.

Replacement and review of a DBObject allows for more general changes, and therefore must use different algorithms. There are two of these:

- Replacement of an object during parsing uses a two-stage algorithm with an auxiliary catalogue called depths. The method is at Context.Replace(). In the first stage objects at each are scanned using _Replace, which adds a new version to cx.done. These are then installed in the Context.
- Review of the RowSets in a context is currently performed using a cascade based on rowset source. The method at Context.Review() calls Review() methods on each RowSet. Context.FixAll()is then called which calls Fix() on each object..

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.

2.5 The folder and project structure for the source code

The src folder contains

- Folders for the Pyrrho applications: PyrroCmd, PyrrhoTest, PyrrhoJC, and PyrrhoSQL.

- The Shared folder contains the sources for the PyrrhoDBMS engine and the PyrrhoLink API and this arrangement is described next.

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

- The Properties folder handles Visual Studio project stricture. Unusually, it has subfolders for isolating the AssemblyInfo for each of the 3 solutions.

- The Common, Level1 to Level5 folders contain the real code base for the DBMS.

- 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[13].

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 type parameters, 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 a IComparable. If the values are also IComparable, CTree is used instead (CTree is then IComparable). In order to report the non-existence of an entry, we use BTree<X_long?> instead of CTree<X,long>.

There are also BList<V> and CList<V>, based on BTree<int,V> and CTree<int,V> respectively, but the indexes are constrained to be 0,1,2,,. so that the mutation operators are O(N) instead of O(logN).

See 3.1.5 for a list of related B-Tree classes used in Pyrrho. All are immutable and shareable. BTree<K,V> is shareable provided K and V are, or provided no nonshareable objects have been inserted.

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.
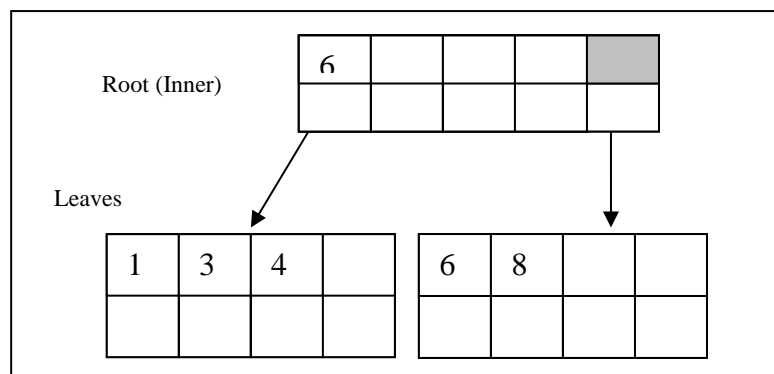
3.1.2 ATree<K,V>

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 -.

---

[13] Remember that neither immutability nor shareability are heritable. Mutability and non-shareability are.

For a multilevel index, Key can be an array or row (this is implemented in MTree and RTree, see section 3.2). ATree itself is immutable and shareable (but its subclasses might not be, see footnote).

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
ABookmark<K,V>Last()	Provides a bookmark for the last 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 a BTree 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. Custom operators are static, and so the implementation chosen depends on the declared type of t .

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). BList<V> is a subclass of BTree<int,V> .

If x is BTree<K,V> and V is a class with a default value d, we can write safe code such as x[k]??d, and this avoids having to check x.Contains(k) . If V is object we currently need to write extra brackets in expressions such as (long)(x[k]??-1L).

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 or previous item if any. Every B-Tree provides method First() and Last() that returns a bookmark for the first (resp. last) element of the tree (or null if the tree is empty).

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

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

Cursors follow a similar pattern (see section 3.6.10).

3.1.5 ATree<K,V> Subclasses

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

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

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

Name	BaseClass	Description
BList<V>	BTree<K,V>	Same as BList<V> with a shortcut for adding to the end
CList<V>	BList<V>	Same as BList<V> where V is IComparable, allows comparison of lists
BTree<K,V>	ATree<K,V>	The main implementation of B-Trees, where K is IComparable
CTree<K,V>	BTree<K,V>	V is also IComparable, and so is the tree
ObTree	BTree<long,DBObject>	Adds a ToString() useful for debugging
SqlTree	CTree<TypedValue, TypedValue>	For one-level indexes where the keys and values have readonly strong types
Idents	BTree<string, (Iix,Idents)>	This behaves like a lookup tree for Ident->DBObject. Iix contains a lexical position, defining position, and select depth

If V is a value such as int or long, it is often convenient to use nullable versions: for example a queue of longs can be conveniently implemented as BList<long?>, and then q -= 0 means "remove the head of the queue" since 0 is an int, the head of the list is x[0], which may be null, and q += k means "add k to the end of the queue" (k is converted to long). BList therefore always renumbers nodes to be a sequence starting at 0, and so is a slower implementation than BTree.

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

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

3.2 Other Common Data Structures

3.2.1 Integer

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

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

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

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

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

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

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

3.2.2 Decimal

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

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

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

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

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

3.2.3 Character Data

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

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

3.2.4 Documents

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

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

3.2.5 Domain

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

However, in v7, SqlValue and RowSet columns are identified by uid (not name), and each uid gives the SqlValue used to compute the column. Such SqlValues may be simple columns (SqlCopy or SqlLiteral) or more complex expressions. The Domain's rowType is a list of these SqlValue uids and so the Domain of the SqlValue or RowSet (despite being called a datatype) also contains the evaluation instructions in addition to determining the underlying primtive types. The Domain.CompareTo method ignores differences in these instructions, while rowType comparison checks that the evaluation expressions match.

The Domain provides methods of input and output of data, parsing and formatting of value strings, coercing, checking assignability etc.

Parsing results in many ad-hoc domain objects being constructed. When committing objects to a database the Domain.Create function looks to see if the database already defines a domain with the same structure (Domain.CompareTo), and if so, the relevant domain definition is referenced as the structure of the Domain.

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

Name	Description
Null	The data type of the null value
Wild	The data type of a wildcard for traversing compound indexes
Bool	The Boolean data type (see BooleanType)
RdfBool	The iri-defined version of this
Blob	The data type for byte[]
MTree	Multi-level index (used in implementation of MTree indexes)
Partial	Partially-ordered set (ditto)
Char	The unbounded Unicode character string
RdfString	The iri-defined version of this
XML	The SQL XML type
Int	A high-precision integer (up to 2048 bits)
RdfInteger	The iri-defined version of this (in principle unbounded)
RdfInt	`value>=-2147483648 and value<=2147483647`
RdfLong	`value>=-9223372036854775808 and value<=9223372036854775807`
RdfShort	`value>=-32768 and value<=32768`
RdfByte	`value>=-128 and value<=127`
RdfUnsignedInt	`value>=0 and value<=4294967295`
RdfUnsignedLong	`value>=0 and value<=18446744073709551615`
RdfUnsignedShort	`value>=0 and value<=65535`
RdfUnsignedByte	`value>=0 and value<=255`
RdfNonPositiveInteger	`value<=0`
RdfNegativeInteger	`value<0`
RdfPositiveInteger	`value>0`
RdfNonNegativeInteger	`value>=0`
Numeric	The SQL fixed point datatype
RdfDecimal	The iri-defined version of this
Real	The SQL approximate-precision datatype
RdfDouble	The iri-defined version of this
RdfFloat	Defined as Real with 6 digits of precision
Date	The SQL date type
RdfDate	The iri-defined version of this
Timespan	The SQL time type
Timestamp	The SQL timestamp data type
RdfDateTime	The iri-defined version of this
Interval	The SQL Interval type
Collection	The SQL array type
Multiset	The SQL multiset type
UnionNumeric	A union data type for constants that can be coerced to numeric or real
UnionDate	A union of Date, Timespan, Timestamp, Interval for constants

See also sec 3.5.3.

3.2.6 TypedValue

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

Cursor
TArray
TBlob
TBool
TChar
TContext
TDateTime
TDocArray
TDocument
TInt

sequence of physical records[14]. It uses 8-bit bytes and Unicode UTF8 for strings, but otherwise is independent of machine architecture, operating system, or location.

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

3.3.1 Client-server protocol

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

Request	Intermediate	Final response	
(Connect/Open)		Primary	
(Error)		Exception	
		FatalError	
Authority		*Done*	
BeginTransaction			unacknowledged
CloseConnection			unacknowledged
CloseReader			unacknowledged
Commit	{Warning}	Done	
CommitAndReport CommitAndReport1	{Warning}	TransactionReport	
CommitAndReportTrace CommitAndReportTrace1	{Warning}	TransactionReportTrace	
CommitTrace	{Warning}	DoneTrace	
Execute	{Warning}	Done	
		Schema	
ExecuteTrace	{Warning}	DoneTrace	
		Schema	
ExecuteNonQuery	{Warning}	Done	
ExecuteNonQueryTrace	{Warning}	DoneTrace	
ExecuteReader	{Warning}	Done	
		Schema	
Get Get1	{Warning}	Schema	
		Done	
		NoData	
Get2	{Warning}	Schema1	
		Done	
		NoData	
GetFileNames		Files	
GetInfo	{Warning}	NoData	
		Columns	
Post	{Warning} Schema	Done	
Put	{Warning} Schema	Done	
Prepare		Done	
ReaderData		NoData	
		ReaderData	
ResetReader		Done	
Rest	{Warning}	Schema	
Rollback		Done	
TypeInfo		(data)	

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

3.4 Physical (level 2)

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

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

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

In version 7 many of the so-called Physical classes in memory are subclasses of Compiled, and these have a framing field structures belonging to level 4: for example, expressions and executable statements. These objects are not serialised to or from disk: as in previous versions of Pyrrho stored procedures, queries, triggers etc are recoded in the log in source (string) form. During database Load these source strings are compiled (see sec 3.4.2 below) into an immutable form that is simply cached in the Context when required.

3.4.1 Physical subclasses (Level 2)

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

Code	Class	Base class	Description
0	EndOfFile	Physical	Checksum record at end of file
1	PTable	Physical	Defines a table name
2	PRole	Physical	A Roleand description
3	PColumn	Physical	Defines a TableColumn or Type field
4	Record	Physical	Records an INSERT to a table
5	Update	Record	Records an UPDATE of a record
6	Change	Physical	Renaming of non-column objects
7	Alter	PColumn2	Modify column definition
8	Drop	Physical	Forget a database object
9	Checkpoint	Physical	A synchronization point
10	Delete	Physical	Forget a record from a table
11	Edit	PDomain	ALTER DOMAIN details
12	PIndex	Physical	Entity, unique, references
13	Modify	Physical	Change proc, method, trigger,check,view
14	PDomain	Physical	Define a Domain
15	PCheck	Physical	Check constraint for something
16	*PProcedure*	*Physical*	*Stored procedure/function (deprecated, see below)*
17	PTrigger	Physical	Define a trigger
18	PView	Physical	Define a niew
19	PUser	Physical	Record a user name
20	PTransaction	Physical	Record a transaction
21	Grant	Physical	Grant privileges to something
22	Revoke	Grant	Revoke privileges
23	PRole1	Physical	Record a role name
24	PColumn2	PColumn	For more column constraints
25	PType	PDomain	A user-defined structured type

26	PMethod	PProcedure	A method for a PType (deprecated, see below)
27	PTransaction2	PTransaction	Distributed transaction support
28	Ordering	Physical	Ordering for a user-defined type
29	(NotUsed)		
30	PDateType	PDomain	For interval types
31	*PTemporalView*	*Physical*	*A View for a Temporal Table (obsolete)*
32	PImportTransaction	PTransaction	A transaction with a source URI
33	Record1	Record	A record with provenance URI
34	PType1	PType	A user-defined type with a reference URI
35	PProcedure2	Physical	(PProcedure2) Specifies return type information
36	PMethod2	PProcedures	(PMethod2) Specifies return type information
37	PIndex1	PIndex	Adapter coercing to a referential constraint
38	Reference	Physical	Adapter coerces to a reference constraint
39	Record2	Record	Used for record subtyping
40	Curated	Physical	Record curation of the database
41	*Partitioned*	*Physical*	*Record a partitioning of the database*
42	PDomain1	PDomain	For OWL data types
43	Namespace	Physical	For OWL data types
44	PTable1	PTable	For OWL row types
45	Alter2	PColumn2	Change column names
46	AlterRowIri	PTable1	Change OWL row types
47	PColumn3	PColumn2	Add new column constraints
48	Alter3	PColumn3	Alter column constraints
49	*PView1*	*Pview*	*Define update rules for a view (obsolete)*
50	Metadata	Physical	Record metadata for a database object
51	PeriodDef	Physical	Define a period (pair of base columns)
52	Versioning	Physical	Specify system or application versioning
53	PCheck2	PCheck	Constraints for more general types of object
54	*Partition*	*Physical*	*Manages schema for a partition*
55	*Reference1*	*Reference*	*For cross-partition references*
56	ColumnPath	Physical	Records path selectors needed for constraints
57	Metadata2	Metadata	Additional fields for column information
58	Index2	Index	Supports deep structure
59	*DeleteReference1*	*Reference*	
60	Authenticate		Credential information for web-based login
61	RestView	View	Views defined over HTTP
62	TriggeredAction		Distinguishes triggered parts of a transaction
63	*RestView1*	*RestView*	*deprecated*
64	Metadata3	Metadata	Additional fields for column information in views
65	RestView2	RestView	Support for GET USING

3.4.2 Compiled and Framing

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

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

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

by the definer. This is generally supported by previous versions of the DBMS, but objects will display differently in the Log$ system tables.

There is a subclass of Physical for the Compiled Level2 objects, to provide helper methods for the compilation process. The in-memory data structures resulting from parsing include SqlValues, RowSets and Executables, and an associated data structure called Framing holds a shareable version of the objects thus created for the compiled object.

OnLoad() recreates the Framing (see sec 3.5.18) for the compiled object from deserialised source code. A flag is set so that the parsing routines use uids in the range starting at `0 instead of lexical uids and heap uids, so that each compiled object in the database has its own private range of executable uids.

Views are a bit more complicated as they are embedded into ambient query processing, and a single query may contain multiple references to the view, so that each reference to a View creates a new instance in the Context with its own uids.

The framing of a UDType includes the Domain for the structure table and method declarations. These have different semantics. The type and its fields will have physical uids, because they are used to serialise fields in Records, while the methods will have executable uids for formal parameters.

The framing objects, in all cases (apart from Prepared statements, discussed below), form a contiguous block of executable uids. Instancing for these objects (other than views) is simply a matter of adding the framing objects to the current Context. The mapping from View object to instance is illustrated in the following graphic[15]:
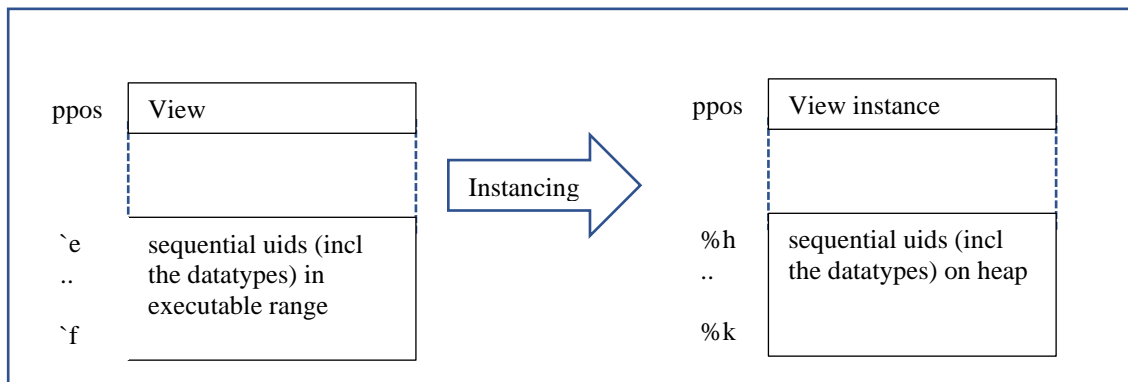
There is an interaction with the Transaction Commit process, as the range of executable uids for a particular compiled object may no longer be available: the relocation process for committing a compiled object also relocates the framing objects to the next executable statement position in the target database.

Prepared Statements are simple SQL statements that are immediately parsed into a reserved initial portion of the *heap* uid space. On completion of the parse the object is in the heap range and ready to receive query parameters for immediate execution. The initial prepared statement may contain references to compiled objects that are instanced to the heap as above. In PyrrhoDBMS, prepared statements can only be executed in the current connection, so that they are local to the application that defines them, and there is no need for mechanisms to describe their parameters or deallocate them[16]. In standard SQL, execution parameters for prepared statements must be simple constant value expressions. Prepared statements are never written to the database file and do not need to be instanced.

Some extra cases for ExecutionStatus have been added to help control this process:

ExecutionStatus	Comment
Parse	For database load. Use executable range for uids, so that Framing object is ready at the end of parsing. Instancing of view objects may occur during parsing, but to executable range instead of heap.
Compile	For creating compiled objects (including method headers)

---

[15] In the code, `e is called instSFirst, `f is called instSLast, and %h is called instDFirst. These are computed in View.Instance().

[16] There are also no concerns about "SQL injection".

	Use executable range for uids, so that Framing object is ready at the end of parsing. Instancing of view objects may occur during parsing, but to executable range instead of heap.
Obey	For command input and execution (including data definition) Lexical source code positions are used for uids. Use heap for uids that do not correspond to source code. Copy View instance to use heap uids as shown above. Change to Compile state when source for a compiled object is to be parsed At RdrClose() reinitialise nextStmt and nextHeap.
Prepare	For creating PreparedStatements, Use heap range for uids starting at nextPrep, which is then updated to follow Prepared statements. Use instancing for referenced compiled objects to heap range as in Obey.
SubView	Ensures view instances are added to the executable range as specified above

It is important that the same stages occur for committed and uncommitted objects. Note that there are formulae for uid to newuid values in every case, so no need for scanning, done or uids.

To implement the above mechanism, for XX = (roughly) Check, Column, View, Procedure, Modify, Trigger, we want the following.

- Before parsing source for XX if cx.parse is Obey, set cx.parse to Compile (ForConstraintParse does this for Column for generation rule) and restore it at the end of the parsing routine.
- The Compiled class has a public long called nst which is set to the first framing DBObject we want in the framing: sometimes this is from an earlier object in the database, e.g. a previously defined table.
- In OnLoad for XX we have framing = new Framing(psr.cx,nst);
- Instancing for Views makes copies of all of the framing objects as shown above. In View.Instance(cx) we set up cx.instDFirst etc to control the relocation in the above diagram. At the end of instancing a View, set the nextHeap pointer to after the new instanced framing objects.

See the worked examples in sec 6.4-7 below.

Framing is a subclass of Basis (see below).

Key	Property	Type	Comment	Uid
Depths	depths	BTree<int, BTree<long, DBObject>>		
Obs	obs	BTree<long,DBObject>		-449
Resullt	result	long	RowSet	-452

3.5 Database Level Data Structures (Level 3)

3.5.1 Basis

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

```
// negative keys are for system data, positive for user-defined data

public readonly BTree<long, object> mem;
```

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

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

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

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

Key	Property	Type	Uid
Name	name	string	-50

In the code a key is defined as follows:

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

The name property is then accessed by

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

This defines name as a readonly property of Basis[17].

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

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

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

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

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

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

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

The following sections list other subclasses of Basis (other than Framing, see the previous section).

Key	Property	Type	Subclass	Uid
Val	val	long	UpdateAssignment	-237
Vbl	vbl	long	UpdateAssignment	-238
Items	items	CList<long?>	OrderSpec	-217
_Generation	generation	Generation	GenerationRule	-273
GenExp	exp	SqlValue	GenerationRule	-274
GenString	gfs	string	GenerationRule	-275

3.5.2 Database

Database is a subclass of Basis. The database class manages all of the database objects and all transactional proposals for updating them. The base class (Database) is used to share databases with a number of connections. Like other level 3 objects, Database is immutable and shareable, so that Reader, Writer and Transaction all have their own version of the database they are working on[18]. 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:

---

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

[18] Reader and Writer both contain Contexts, whose db field is a snapshot of the Database.

- 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.[19]

In v7.0, many DBObject subclasses are for Query and SqlValue objects that do not correspond to physical records but have been constructed on an ad-hoc basis 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 (for committed objects) is still a fixed position in the transaction log.

The system database _system contains the predefined types and system tables, and two roles; $Schema and _public. Every database inherits the objects from _system including the guest role (this is just _public).

The properties defined by the Database class are as follows:

Key	Property	Type	Comments	Uid
Cascade	cascade	bool	only for Transaction	-227
ColTracker	colTracker	BTree<long,BTree<long,long?>>	for Log$ rowsets	-318
_Connection	conn	BTree<string,string>	the connection string	-261
Curated	curated	long	Point at which the database was archived	-53
Format	format	int	50 for Pyrrho v5-6, 51 for v7	-54
Guest	guest	long	Role	-55
LastModified	lastModified	DateTime	UTC last modification	-279
Levels	levels	BTree<Level,long?>	classification	-56
LevelUids	cache	BTree<long,Level>		-57
Log	log	BTree<long,Physical.Type>	Index to the log	-188
NextPos	nextPos	long[20]	Proposed new Physicals	-395
NextPrep	nextPrep	long	Connection uids, dynamic	-394
NextStmt	nextStmt	long	Uncommitted compiled code	-393
NextId	nextId	long	Offset of next transaction step	-58
Owner	owner	long		-59
Public		long	-1L	-311
Role	role	long	Role	-285
_Role	_role	long	role	-302
Roles	roles	BTree<string,long?>		-60
_Schema		long	Same as _system._role	-291
SchemaKey	schemakey	long	for POCO	-286
Types	types	BTree<Domain,Domain>	nameless types	-61
TypeTracker	typeTracker	BTree<long,BTree<long,Domain>>	for Alter Type	-315
User	user	long	User	-277
_User	_user	long	user	-301

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

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

The 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 and 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.
- 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		-278
Diagnostics	diagnostics	BTree<Sqlx.TypedValue>	For GET DIAGNOSTICS	-280
ReadConstraint	readConstraint	BTree<long,ReadConstraint>		-283
Step	step	long		-276
TriggeredAction	triggeredAction	long	role boundary	-288

3.5.4 Role

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). Prior to September 2022 the Role contained a set of ObInfo containing names, privileges and metadata for each object it can access. Instead, from September 2022, each database object contains an ObInfo for each Role that has been granted access.

A default role (initially with the same name as the database) is created with definer $Schema.

The first role to be defined in the database sets the name of the default role and immediately becomes the database's defining role. Nothing else is updated.

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

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

User is a subclass of Role:

Key	Property	Type	Comments	Uid
Password	pwd	string	hidden	-303
Clearance	clearance	Level		-305

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[21]. 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. When a database object (or role) is defined, it records the defining role. The defining user can be determined from the transaction log but is not generally needed.

User and role ids are indexed by the default role, together with all domain and type definitions. Roles inherit naming information for objects granted to them and can modify object names as seen from their role.

3.5.5 DBObject

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

Roles can be granted access to many DBObject types, including roles, tables (excluding system tables), views, columns, fields, procedures, methods, domains and user-defined types, so that in v7 the Role object maintains a list (infos) that gives access privileges. The effective row-type of a Table depends on which columns have been granted to the role. In Pyrrho this facility was extended to allow role-based metadata and names[22], so that in v7 the Role becomes responsible for all name lookups for level 3 objects. Level 4 objects contains names directly.

---

[21] 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.

[22] 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!

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

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

Basis properties for DBObject are as follows

Key	Property	Type	Comments	
_Alias	alias	string		-62
Classification	classification	Level		-63
Definer	definer	long	Role	-64
Defpos		long	for Rest service $pos	-257
Dependents	dependents	BTree<long,bool>	Objects that must be serialised before the current one	-65
Depth	depth	int	Max depth of Dependents (>=1)	-66
Description	desc	string		-67
_Domain	domain	Domain	Not for Domains	-176
Framing	framing	BTree<long,DBObject>	For compiled objects	-167
LastChange	lastChange	long		-68
Sensitive	sensitive	bool		-69

DBObject has the following subclasses documented in later sections below:

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

The following subclasses define further key uids:

Key	Property	Type	Subclass	Uid
GroupKind	kind	Sqlx	Grouping	-232
Groups	groups	BList<Grouping>	Grouping	-233
Members	members	BList<long?>	Grouping	-234
DistinctGp	distinct	bool	GroupSpecification	-235
Sets	sets	BList<long?>	GroupSpecification	-236
Current	current	bool	WindowBound	-218
Distance	distance	TypedValue	WindowBound	-219
Preceding	preceding	bool	WindowBound	-220
Unbounded	unbounded	bool	WindowBound	-221
Exclude	exclude	Sqlx	WindowSpecification	-222
High	high	WindowBound	WindowSpecification	-223
Low	low	WindowBound	WindowSpecification	-224
Order	order	OrderSpec	WindowSpecification	-225
OrderWindow	orderWindow	string	WindowSpecification	-226
OrdType	ordType	Domain	WindowSpecification	-227
Partition	partition	int	WindowSpecification	-228
PartitionType	partitionType	Domain	WindowSpecification	-229
Units	units	Splx	WindowSpecification	-230

March 2023

WQuery	query	Query	WindowSpecification	-231
Checks	checks	BTree<long,Check>	TableColumn	-268
Generated	generated	GenerationRule (see below)	TableColumn	-269
Table	tabledefpos	long	TableColumn	-270
UpdateAssignments	update	BList<UpdateAssignment>	TableColumn	-271
UpdateString	updateString	string	TableColumn	-272
Prev	prev	TableColumn	ColumnPath	-321
StartCol	startCol	long	PeriodDef	-387
EndCol	endCol	long	PeriodDef	-388
ParamMode	paramMode	Sqlx	ParamInfo	-98
Result	result	Sqlx	ParamInfo	-99

3.5.6 ObInfo

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

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

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

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

3.5.7 Domain

In v7, the Common level SqlDataType has been removed, and instead there is much greater use internally of the Domain and ObInfo classes.

Scalar values are described by a Domain with an empty rowType and representation, as these two properties are reserved for structured types. For more details see section 3.2.5.

Domains are intrinsic to committed objects, while their ObInfo depends on the current Role. The definer's role is used for procedure and constraint execution, while the transaction's current role is used for query processing.

The properties of Domain are as follows:

Key	Property	Type	Comments	Uid

37

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

3.5.8 Table

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

Key	Property	Type	Comments	Uid
ApplicationPS	appPS	long	PeriodSpecification	-262
Enforcement	enforcement	Grant.Privilege	For mandatory access control	-263
Indexes	indexes	BTree<CList<long?>, long?>	Key cols, Index	-264
LastData	lastData	long	Ppos of most recent insert, update or delete	-258
SystemPS	sysPS	long	PeriodSpecification	-265
TableChecks	tableChecks	BTree<long,bool>	Checks	-266
TableCols	tblCols	BTree<long,bool>	TableColumns or type fields	-332
TableRows	tableRows	BTree<long,TableRow>	see note below	-181
Triggers	ptriggers	BTree<PTrigger.TrigType, BTree<long,bool>>	Triggers	-267

SystemTable has an additional property:

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

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

3.5.9 Index

Indexes are constructed when required to implement integrity constraints.

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

3.5.10 SqlValue

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

Key	Property	Type	Comments	Uid
_Columns	columns	CList<long?>	SqlValue	-299
_From	from	long	Directs to a Query	-306
IIx	iix	Iix	Holds lexical position and defpos	-261
Left	left	long	DBObject or SqlValue	-308
_Meta	meta	MetaData		-307
Right	right	long	SqlValue	-309
Sub	sub	long	SqlValue	-310

SqlValue has the following virtual Methods:

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

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

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

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

SqlValue subclasses have the following additional properties:

Key	Property	Type	Subclass	Uid
Bits	bits	BList<long?>	ColumnFunction	-333
Escape	escape	long	LikePredicate	-358
_Like	like	bool	LikePredicate	-359
Found	found	bool	MemberPredicate	-360
Lhs	lhs	long	MemberPredicate	-361
Rhs	rhs	long	MemberPredicate	-362
NIsNull	isnull	bool	NullPredicate	-364
NVal	val	long	NullPredicate	-365
All	all	bool	QuantifiedPredicate etc	-348
Between	between	bool	QuantifiedPredicate etc	-349
Found	found	bool	QuantifiedPredicate etc	-350
High	high	long	QuantifiedPredicate etc	-351
Low	low	long	QuantifiedPredicate etc	-352
Op	op	Sqlx	QuantifiedPredicate etc	-353
Select	select	long	QuantifiedPredicate etc	-354
Vals	vals	BList<long?>	QuantifiedPredicate etc	-355
What	what	long	QuantifiedPredicate etc	-356
Where	where	long	QuantifiedPredicate etc	-357
QExpr	expr	long	QueryPredicate	-363
Call	call	long	SqlCall	-335
TableCol	tableCol	long	SqlTableCol	-322
CopyFrom	copyFrom	long	SqlCopy	-284
Sce	sce	SqlRow	SqlConstructor DqlDefaultConstructor	-336
Udt	ut	Domain	SqlConstructor SqlDefaultConstructor	-337
Spec	spec	long	SqlCursor	-334
Field	field	string	SqlField	-314
Filter	filter	long	SqlFunction	-338
Mod	mod	Sqlx	SqlFunction	-340
Monotonic	monotonic	bool	SqlFunction	-341
Op1	op1	long	SqlFunction	-342
Op2	op2	long	SqlFunction	-343
Query	query	long	SqlFunction	-344
_Val	val	long	SqlFunction	-345
Window	window	long	SqlFunction	-346
WindowId	windowId	long	SqlFunction	-347
_Val	val	TypedValue	SqlLiteral	-317
TransitionRowSet	trs	long	SqlOldTable, SqlOldRowCol	-318
Rows	rows	BList<long?>	SqlRowArray	-319
ArrayValuedQE	aqe	long	SqlSelectArray	-327
TableRow	rec	TableRow	SqlTableRowStart	-311
TreatExpr	val	long	SqlTreatExpr	-313
TreatType	type	long	SqlTypeExpr	-312
Array	array	BList<long?>	SqlValueArray	-328

Svs	svs	long	SqlValueArray	-329
Modifier	mod	Sqlx	SqlValueExpr	-316
Expr	expr	long	SqlValueSelect	-330
Source	source	string	SqlValueSelect	-331
Attrs	attrs	BTree<int, (XmlName,SqlValue)>	SqlXmlValue	-323
Children	children	BList<long?>	SqlXmlValue	-324
Content	content	long	SqlXmlValue	-325
Element	element	XmlName	SqlXmlValue	-326

3.5.11 Check

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

3.5.12 Procedure

In SQL, procedures are looked up by name and arity (not by signature: if the syntax includes a signature, e.g. DROP or GRANT, it is only to get the arity). The Role has a two-stage lookup table called procedures to implement this aspect. Functions are procedures with a return type. The _Domain for the Procedure gives the return type.

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

3.5.13 Method

Method is a subclass of Procedure. The ObInfo for a UDType has a two-stage finding methods by name and arity called methodInfos. _Domain gives the return type. The owning UDT is TypeDef.

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

3.5.14 Trigger

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

Section 4.44 of the SQL standard ISO9075 is quite complex. Any given table can have any number of triggers defined, and more than one of any type. For example, if a table has more than one Update trigger each can refer to the old row or the new row using different identifiers (and may have different definers and hence may access different columns). As the transition row set is traversed, each row must be acted on by each of these triggers in turn (but in an implementation-defined order). Within the trigger definition a column of the row may be referenced directly or via the old row or new row, For each statement affecting a table, an old table is constructed if required for any trigger, consisting of the rows that will be accessed by the statement. For each row, an old row and a new row are constructed if required for an update trigger. These belong to the TableActivation (sec 3.6.2 below) and are used to install the relevant structures in each TriggerActivation that needs them.

In this implementation, the following interpretation is made of this section of the standard: Trigger statements may update old and new rows and tables with some obvious restrictions about existence. Changes made by assignment to new row or new table or directly to a column of the table[23] are seen by other triggers in the implementation-defined order mentioned above. But changes made within a trigger definition to an old table or old row are not seen by other triggers.

Worked examples to illustrate aspects of trigger operation are to be found in section 6.5 below.

3.5.15 Executable

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

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

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

---

[23] See Note 116 in the SQL standard.

LhsType	lhsType	Domain	MultipleAssignment	-107
List	list	BList<long?>	MultipleAssignment	-108
Rhs	rhs	long	MultipleAssignment	-109
Ret	ret	long	ReturnStatement	-110
CS	cs	long	SelectStatement	-95
Outs	outs	BList<long?>	SelectSingle	-142
Else	els	BList<long?>	SimpleCaseStatement, SearchedCaseStatement	-111
Operand	operand	long	SimpleCaseStatement	-112
Whens	whens	BList<long?>	SimpleCaseStatement, SearchedCaseStatement	-113
Exception	exception	Execption	Signal	-136
Objects	objects	BList<object>	Signal	-137
_Signal	signal	string	Signal	-138
SetList	setlist	BTree<Sqlx,long?>	Signal	-139
SType	stype	Sqlx	Signal	-140
Cond	cond	long	WhenPart	-114
Stms	stms	BList<long?>	WhenPart, LoopStatement	-115
Loop	loop	long	WhileStatement, RepeatStatement, LoopStatement	-121
Search	search	long	WhileStatement, RepeatStatement	-122
What	what	BList<long?>	WhileStatement	-123
Nsps	nsps	BTree<string,string>	XmlNameSpaces	-120
QMarks	qMarks	CList<long?>	PreparedStatement	-396
Target	target	Executable	PreparedStatement	-397

3.5.16 View

Key	Property	Type	Comments	Uid
Targets	targets	CList<long?>		-154
ViewCols	viewCols	CTree<string,long?>		-378
ViewDef	viewdef	string		-379
ViewQuery	viewQry	QueryExpression		-380

RestView is a subclass of View

Key	Property	Type	Comments	Uid
ClientName	nm	string	Deprecated	-381
ClientPassword	pw	string	Deprecated	-382
Mime	mime	string		-255
SqlAgent	sqlAgent	string		-256
UsingTablePos	usingTable	long	Table	-385
ViewStruct	viewStruct	Domain		-386
ViewTable	viewTable	long	Table	-371

Where views are constructed by a simple query (a filter, or a selection of columns) there might seem to be a direct match between view column uids and table column uids, so that the usual SqlCopy mechanism would suffice. However, a single command might reference a view more than once and these instances need to be distinguished from each other. As with table references, instancing creates new objects to ensure the uids are different. With Views, there is an added complication since we would like to parse the view definition (a select statement) just once: when the resulting complied information is instanced, we want a new instance of all of the framing objects so that can add modifications coming from the command such as where conditions (we do not want to alter the compiled object itself). A view may have multiple targets for insert/update/delete.

RestViews

RESTViews, like Views, need to be instanced on reference. Unlike Views, the remote internals of a RESTView are unknown, but we need to be more diligent about optimization of aggregates and selection

because of the need to reduce server-server traffic. The Framing for a RESTRView includes details of the remote columns but does not include a RestRowSet. The RestRowSet is constructed during instancing, when the metadata and description for the REST request will be available.

The metadata for a RESTView includes details of the remote access: the URL, the SqlAgent, and authentication details. Since all of these are details about a remote database, they may be subject to change, and will depend on the session role. (Access to the remote system is a matter for its own security and authentication procedures.)

In testing, we need to ensure that the in-memory database contains the right compiled objects (a) on definition, before commit; (b) after commit, without server restart, (c) after restart on database load. Then we need to check the instancing process and consider optimization for various forms of aggregation. Finally, consider updatability of RESTViews.

The basic design is as follows. RestView is a DBObject (subclasses View); PRestView similarly subclasses PView contains the name of the RestView and its file position is used for associating metadata with the RESTView. The column definitions for the RESTView are handled by a "virtual" PTable whose name is the source of the column definitions[24]. The VirtualTable, its columns and domain are contained the framing of the RestView compiled object. Its columns as SqlValues. The RestView references the domain and the virtual table, and the domain's structure field references the virtual table.

Instancing of RestViews proceeds in the same way as for Views. A RestRowSet (subclassing TableRowSet) is added to the Context, whose columns are copies of the VirtualTable's columns.

As described in a worked example in section 6 below, there are currently two RESTViews implementations in Pyrrho v7. The first uses simple HTTP1.1 requests HEAD, GET, PUT, POST (for insert), and DELETE. It relies on a time-based ETag mechanism to detect transaction conflict, but even in an explicit local transaction, the remote operations are performed immediately, not waiting for commit. This mechanism is selected by including the metadata flag "URL" in the restview definition.

The second implementation provides a better match to the normal transaction model. All of the remote operations in the current transaction are gathered into a script that is sent to the remote server with appropriate ETags when the local transaction commits. This mechanism is selected by default.

3.6 Level 4 Data Structures

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

3.6.1 Context

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

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

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

---

[24] In the syntax definitions for RestView, this is currentlygiven as Representation. The next version of Pyrrho will allow the full TableDefinition syntax at this point, to allow the declaration of remote constraints.

is completed. Activations provide for complex Condition and Signal handling, similar to the operation of exceptions.

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

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

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

- The current transaction snapshot **tr**.

- A set of DBObjects called **obs** consisting of the SqlValues and Queries in the current evaluation. During parsing, there are also (a) a set of definitions called **defs**, which helps with looking up identifier chains, and (b) a structure called **depths**, which organises the set of objects by nesting depth to help with the evolution of queries and sqlValues during parsing analysis.

- A set of RowSets called **data**, one of which is the current **result**, and an association from Queries to RowSets called **results**. The construction of these items completes the compilation process (see sec 3.4.2).

- Volatile lists of TypedValues by uids: Cursors (**cursors**), variables (**values**), a return value (**val**) a top-level Cursor (**rb**). These things enable SqlValues to be evaluated given the context and a RowSet (the **finder**). See sec 3.6.4.

- An Rvv structure called **affected** for the current explicit transaction, containing details of records read or updated by the transaction. The Pyrrho protocol allows the client to access this data (e.g. CommitAndReport). It is used in Transaction.Commit to check for read-write conflicts in read-only transactions. See sec 3.6.5.

- An ETags structure, containing collected ETag validation data for the current context. See sec 3.6.6.

Comtexts form a stack and they may have different roles and therefore permissions. Generally on exit from a context, the values and result are slid down the stack to the parent. More interesting cases arise for Activations (see below) as these are special sorts of Context for procedural code, including triggers and constraints.

3.6.2 Activation

Activation is a subclass of Context. It has the following subclasses:

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
RESTActivation	TargetActivation	Manage REST operation for remote view
TableActivation	TargetActivation	An activation for managing trigger execution
HTTPActivation	TargetActivation	Manage HTTP operation for remote view
TargetActivation	Activation	An activation for controlling insert/update/delete actions
TriggerActivation	Activation	Host for trigger execution

Property	Type	Comments
brk	long	An Activation to break to
cont	long	An Activation to continue to
exceptions	BTree<string,Handler>	

execState	ExecState	saved Transaction and Activation state
ret	TypedValue	
saved	ExecState	
signal	Signal	
locals	BTree<long,object>	

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

Activations provide an exception handling mechanism. Signals cause a change of Context, and the behaviour depends on the kind of Handler defined for that condition. Thus the loop in a CompoundStatement will check the Context that results from Obeying and Executable. If the context has not changed, the next statement in the CompoundStatement is Obeyed. Otherwise we break out of the loop.

When an Activation initializes, it starts with values and other information copied from the parent context. A CalledActivation will set up local variables corresponding to parameters (and, for methods, target fields).A TriggerActivation installs a cursor for the current row from the TransitionRowSet, adding cursors for oldRow and oldTable if these are defined. TargetActivations assist with insert/update/delete operations. The subclass TableActivation also manages Triggers, which operate with the help of TriggerActivations. see the worked example ins section 6.5. A table can define multiple triggers, so modifications to a row may involve the operation of a number of TriggerActivations. During such activity, the transaction state is passed between the different activations, as required by the semantics defined in the SQL standard.

At the end of the activation (e.g. a return statement), the SlideDown method deals with how changes to non-local values should affect the parent context. A number of cases can be distinguished, depending on the type of the parent Context:

Activation: The base SlideDown behaviour is just to copy the changed *non-local* values into the values list.

CalledActivation: A called activation may be for a structured type, in which case updates may be for fields of the target object; while other local variables and parameters will be handled by the ProcedureCall semantics. There is no need for an override of SlideDown.

TriggerActivation: Values assigned to columns of the target table are passed down to the TableActivation (as TargetActivation) and target cursor. Then the base Activation.SlideDown version is called.

TargetActivation: Values assigned to columns of the target table are installed in the target, but this is dealt with by the target's class (in Insert/Update/Delete). There is no need for an override of SlideDown.

Note that in other circumstances, fields of structured objects can only be updated by SqlValueExpr where the operator is dot (e.g. an assignment to x.y), and in that case the whole object value is considered altered as above.

3.6.3 RowSet

RowSets are DBObjects that deliver the result of query processing. RowSets are immutable and shareable, and constructed in a Context during parsing[25]. There is an evaluation pipeline for rowsets, starting with the base tables, applying sorting and joins, aggregation, merging and selection etc, according to a strategy determined during parsing.

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

---

[25] They are included in the pre-compiled state of a compiled object that defines local queries (se 3.4.2)..

Other rowset types (derived tables) have one or more source rowsets, traversed before or during traversal of the result. As far as possible, traversal of the resulting rowset proceeds recursively: a request for a row of a rowset recursively requests a row of the source from which it can be computed. This approach is worthwhile because it is very likely that not all rows will be traversed. JoinRowSets and MergeRowSets use possibly sorted rowset operands, which are built before traversal, but the columns are simple to compute. Aggregation and ordering sometimes require the evaluation pipeline to be broken up with Trivial or ExplicitRowSets constructed for intermediate results. Subqueries require the construction of auxiliary source rowsets during parsing, and window functions and lateral joins require rebuilding of the source rowset when needed during traversal.

All of these are constructed on completion of parsing of the SQL statement that contains them[26]. As the RowSet is constructed from its sources, ordering and filtering operations are distributed into a pipeline of RowSets, whose Domain and Sources specify the columns and their origins in other rowsets. At any stage during traversal, the context maintains the current set of cursors.

As far as possible this pipeline is built during the initial construction of the RowSet, using a number of static methods (often called _Mem) that prepare the set of properties of the RowSet. Some properties must be added later (such as where-conditions), and for this purpose there is a RowSet.New method that adds further properties and is able to modify source rowsets as required. Since the Context contains the current set of rowsets during execution, this New method has access to the Context to retrieve and/or update the source rowsets.

Several compiled objects, such as views and procedures, contain rowsets that are constructed during compilation and are referred to in all references to the compiled object. These can be referenced in different future contexts, possibly with several separate references to a single view. The Instance method creates a fresh instance of a shared DBObject whose properties are modifiable and provides fresh column uids for each instance. Finally, the compiled rowset pipeline can be improved, in the Context.Review method, by propagating filters, groupings and aggregations from the referencing query to deeper levels of the pipeline and removing unnecessary steps.

It is possible during both this process and the Build method to discover that a different index can be used for traversal, and for this reason the notion of a separate IndexRowSet has been dropped in the current version.

The very last thing that is computed during construction or review of a rowset is an assertion, whose values have the almost self-explanatory names SimpleCols (no expressions), MatchesTarget (domain and source rowtypes match), ProvidesTarget (the source contains all the column uids), and AssignTarget (rows of the source are assignment compatible to the domain). For example, a modifiable rowset must have simple columns, rowsets for insertion or merging must have matching rowtypes.

There are numerous subclasses of RowSet. Each RowSet subclass has one or more associated Cursor subclasses with a similar name. Each Cursor subclass has its own implementations of Next and Previous[27]. The table below shows a number of invariants associated with the rowSet class.

SubClass	Role in pipeline
DistinctRowSet	Remove duplicate rows in the source rowset. The rowType matches its source.
DocArrayRowSet	A rowset whose source is a JSON document. The rowType is a single document column.
EmptyRowSet	A rowset with no source, and no requirements on the rowType.
ExplicitRowSet	A rowset whose source is an array of rows, matching the rowType.
InstanceRowSet	A rowset with a mapping of its rowType to columns in base tables. (TableRowSet and ViewRowSet are InstanceRowSets)
JoinRowSet	Form the join of two rowsets: the columns are simple and so are the columns of the two sources.
MergeRowSet	Form the union, intersection or EXCEPT of two compatible rowsets The rowType is assignment compatible to the sources.

---

[26] A separate step builds the rows of the rowset. Building is delayed until traversal, and some rowsets need to be rebuilt if ambient values change. A Cursor always continues to traverse the RowSet (by Next() or Previous()).as it stood at the time of cursor creation (by First(), Last(), or PositionAt()).

[27] TransitionRowSet and some system rowsets are unidirectional.

OrderedRowSet	A rowset formed by reordering the rows in the source. The rowType matches the source.
RoutineCallRowSet	A rowset whose source is a call to a procedure or method
RowSetSection	A rowset forned by selection from the source by row sequence. The rowType matches the source.
SelectRowSet	A rowset formed by selection of rows using SQL expressions. The select list can contain aggregations.
SelectedRowSet	A rowset formed by selection of certain columns from the source. The columns are simple, but may have a different order from the source.
SqlRowSet	A rowset whose source is a list of row-valued SQL expressions The rowType is assignment-compatible with the source.
SystemRowSet	A rowset constructed from data structures in the server
TableRowSet	An instanced rowset whose source is a base table. The rowType matches the source table as seen by the role.
TransitionRowSet	For input/update/delete operations (constructed by TargetActivation)
TransitionTableRowSet	The rowset accessed by OLD TABLE and NEW TABLE during trigger operation. The rowType matches the enclosing transition row set.
TrivialRowSet	A rowset consisting of a single SQL row
ValueRowSet	A list of rows provided elementwise
ViewRowSet	An instance rowset whose source is a local or remote view. The rowType matches the view definition as seen by the role.
WindowRowSet	A rowset from application of a window function to the source rowset

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

Name	Type	Definition	Uid
_tgt	PTrigger.TrigType	TransitionRowSet.TriggerType	-421
_trs	TransitionRowSet	TransitionTableRowSet.Trs	-431
actuals	BList<long?>	RoutineCallRowSet.Actuals	-435
adapters	Adapters	TransitionRowSet._Adapters	-429
built	bool	RowSet.Built	-402
data	BTree<long,TableRow>	TransitionTableRowSet.Data	-432
defaultURL	string	RestRowSet.DefaultURL	-379
defaults	BTree<long,TypedValue>	TransitionRowSet.Defaults	-415
distinct	bool	RowSet.Distinct	-239
docs	BList<SqlValue>	DocArrayRowSet.Docs	-440
domain	Domain	DBObject._Domain	-176
explRows	BList<(long,TRow)>	ExplicitRowSet.ExplRows	-414
filter	PRow	FilterRowSet.IxFilter	-411
first	long	JoinRowSet.Jfirst	-447
from	From	TransitionRowSet,TrsFrom	-416
groupings	BList<long?>	RowSet.Groupings	-406
groupSpec	GroupSpecification	RowSet.Group	-199
having	BTree<long,bool>	RowSet.Having	-200
index	long	IndexRowSet._Index	-410
indexdefpos	long	TransitionRowSet.IxDefPos	-420
join	JoinPart	JoinRowSet._Join	-446
joinCols	BTree<string,int>	RestRowSet.JoinCols	-383
keys	CList<long?>	Index.Keys	-159
lastData	long	Table.LastData	-258
map	BTree<long,long?>	TransitionTableRowSet.Map	-433
matches	BTree<long,TypedValue>	Query._Matches	-182
mtree	MTree	Index.Tree	-164
needed	CTree<long,bool>	RowSet._Needed	-401
offset	int	RowSetSection.Offset	-438

proc	Procedure	RoutineCallRowSet.Proc	-436
ra	TriggerContext	TransitionRowSet.Ra	-424
rb	TriggerContext	TransitionRowSet.Rb	-422
remoteAggregates	bool	RestRowSet.RemoteAggregates	-384
remoteCols	BTree<string,long?>	RestRowSet.RemoteCols	-373
remoteGroups	GroupSpecification	RestRowSet.RemoteGroups	-374
restValue	TArray	RestRowSet.RestValue	-457
restView	long	RestRowSet.RestView	-459
result	RowSet	RoutineCallRowSet.Result	-437
ri	TriggerContext	TransitionRowSet.Ri	-423
rmap	CTree<long,long?>	TransitionTableRowSet.RMap	-434
row	TRow	TrivialRowSet.Singleton	-405
rowOrder	CList<long?>	RowSet.RowOrder	-404
rows	Blist<TRow>	RowSet._Rows	-407
rt	CList<long?>	(domain.rowType)	
second	long	JoinRowSet.Second	-448
size	int	RowSetSection.Size	-439
source	long	From.Source	-151
sqlRows	BList<long?>	SqlRowSet.SqlRows	-413
ta	TriggerContext	TransitionRowSet.Ta	-426
table	long	IndexRowSet.IxTable	-409
tabledefpos	long	SqlInsert._Table	-154
targetAc	Activation	TransitionRowSet.TargetAc	-430
targetTrans	CTree<long,long?>	TransitionRowSet.TargetTrans	-418
tb	TriggerContext	TransitionRowSet.Tb	-425
td	TriggerContext	TransitionRowSet.Td	-428
transTarget	CTree<long,long?>	TransitionRowSet.TransTarget	-419
tree	RTree	OrderedRowSet._RTree	-412
usingCols	BTree<string,long?>	RetRowSet.UsingCols	-259
usingTable	long	RESTRowSet.UsingTable	-260
values	Tmultiset	WindowRowSet.Multi	-441
wf	SqlFunction	WindowRowSet.Window	-442
where	BTree<long,bool>	Query.Where	-190

3.6.4 Cursor

Previously called RowBookmark, this is an abstract and immutable subclass of TRow for traversing rowSets. All RowSets offer a First() that returns a Cursor at position 0, or null, and a Last() that returns a Cursor at the end of the rowset, or null. Cursors are immutable, but their values can be updated (as usual giving a new cursor, stored in the appropriate context). Note however that an updated cursor continues to traverse the rowset as it was at the start of traversal.

The Context remembers the current Cursor for each RowSet it defines: it contains the values for the current row as defined in the row's representation. The construction of some rowsets (e.g. grouped and windowed) uses a temporary context. Each RowSet has a field ids for its columns (this is the representation tree for its Domain).

The interface offered includes the following:

long _defpos	The row uid (or 0)
int display	The number of columns
TRow key	*Abstract* The current key
Cursor Last(Context cx)	*Abstract:* Returns a bookmark for the last row, or returns null if there is none
BTree<long,TypedValue> _needed	Ambient data required for evaluation
Cursor Next(Context cx)	*Abstract:* Returns a bookmark for the next row, or returns null if there is none
int _pos	The current position: starts at 0 for First() cursor in a traversal
long _ppos	The log position for the current row (or 0)

49

Cursor PositionAt(pos)	Returns a bookmark for the given position, or null if there is none.
TableRow Rec()	*Abstract* The current table row if defined for this rowset
long _rowsetpos	The rowset uid

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

From the above interface, it is clear that the most important property of a cursor is its role in traversing a RowSet (Next() and Previous()). But cursors also play a useful role in SqlValue evaluation. Recall that an SqlValue uid refers to a cell in a row. The cursor is the row: so evaluations of simple columns cam use the current cursor of the appropriate rowset. The Context maintains the current set of Cursors in that context, and the RowSet whose cursor is currently supplying values (the finder).

This is important because most rowsets are built from their source rowsets. Many rowsets require building at traversal time (DistinctRowSet, SelectRowSet, OrderedRowSet, even lateral joins) and the same evaluation mechanism needs to be used consistently at every stage. To make this work, the context contains a finder field (Contexts are not immutable), which is fixed for each cursor evaluation step.

All rowsets have cursors that can be built from their source rowsets, using a static New method (in case the source is exhausted, New can return null): with this normal method of constructing cursors, the cursor constructor is protected or private. There are just a few cursors that can be built from their targets, in order to perform Insert operations: these are the cursors for TransitionRowSet, SelectedRowSet, RestRowSet, OrderedRowSet, and JoinRowSet (!), and so these have constructors that are internal. TrivialRowSet also has a Cursor with internal constructors.

Two important Cursor subclasses: TargetCursor and TriggerCursor are not used for traversal but are used as part of this evaluation machinery, as explained in section 6.5. The TriggerCursor is constructed from a targetCursor for each row trigger and ensures that the TriggerActivation has the right finder for trigger execution.

3.6.5 Rvv

Rvv is a CTree<long, CTree<long, long?>> structure, mapping from a table defining position to a list of row defining positions (or -1 for the whole table) and the length of the database when that row was last updated (a proxy for time).

Rvv information is collected by Pyrrho during explicit transactions, and records all information read or updated in a transaction. It is accessed during Transaction.Commit and compared with the changes that have been made to the database since the start of the transaction. Rvv is a shareable object but is not currently placed in any shareable object. It can be seen in operation in test 10 and Demo 2 (see Appendix).

3.6.6 ETags

As described in RFC 7232, an ETag is a string value returned from an HTTP/1.1 server that enables conditional requests to be made. Pyrrho's HTTPService supports RFC7232. This section describes how the service is used in the main Pyrrho protocol service to support transaction-based RestViews.

ETag strings are described in RFC7232 as a cookie containing information meaningful to the server. For Pyrrho this cookie is the string representation of an Rvv.

The ETags object is mutable. In any given context it gives the the following RFC 7232 information: the date to be used for the next Unmodified-Since and optionally an ETag to assert in the next HTTP request. It also contains information for the local database and each URL used in a class called HttpParams. The HttpParams class provides the information needed to access an HTTP1.1 server to check an ETag, It contains the URL for the server, a set of credentials and authorization strings, and an ETag string.

4. Locks, Integrity and Transaction Conflicts

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.

The database file is locked during the validation step of commit (the transaction proposals are checked, the file is locked, and then checked again). The validation step includes any modifications made by triggers and cascades and is discussed in section 4.2 below.

Outside of this validation step, and initial load of the database, reading of the database file is only required for access to data for some system tables, and the operating system file object is locked during Seek operations. The transaction commit results in a new shared version of the database which is available for the start of any other transaction.

During a transaction, mandatory access control may require the generation of audit records, and the database file is also locked while these are added to the log.

The subsections below provide an overview of the validation requirements from serialisability (4.2.1 and 4.3.2) and integrity constraints (4.2.3 to 4.3.5).

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: $P = p_0 p_1 \ldots p_n$ etc.

4.2.1 ReadConstraints

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[28]:

- 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 Connection

The server is multi-threaded, and a new thread is created for each Connection. The connection string gives the user identity and can request an initial role for the connection.

1.   If the database has no users, the system account has all privileges on the database. There is no need to record the user id of the system account, and there is no auditing or mandatory access control. The system account has the identity of the account that starts the server.

2.   Otherwise the database has users, and:

a.   The log can be read only by the database owner and the system account and is not subject to audit.

b.   All other use of the database must have a valid user id and session role and the need for audit is determined by object properties. Guest/Public is a role not a user id.

i.   The user's identity must be defined (a User database object) for the current transaction, so that we can audit their activities if necessary. An ad-hoc user object must be installed if a matching user id cannot be found, initially with a transaction-local uid..

1.   If this object is committed (including by an audit record), it is then a defined user, and will be re-used next time this account connects to the database.

ii.   If the user is only allowed to use one role, this is supplied by default on connection.

iii.   If the role is not set or the current user cannot use any other role, the session role will be the Guest/Public role.

iv.   There is no way to set the session role to $Schema.

These rules are sufficient in all cases to ensure that every connection is immediately equipped with a session user and a session role.

5.2 Lexical analysis

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

---

[28] 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.

- 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	
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.

5.3 Parser

The parser retains the following data:

- The Lexer

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

- The current Context, including the current Database or Transaction

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

5.3.1 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)

5.3.3 Parsing routines

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

The routines accessible from other classes are as follows:

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

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

new Parser(cx).ParseSql(sql)

and similar calls work.

6. Query Processing and Code Execution

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

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

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

6.1 Overview of Query Analysis

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

In addition, Pyrrho operates a very lazy approach to rowSet building and 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. Many RowSets require a build step before traversal (where rows are ordered, grouped or joined), and in some cases, the build step is repeated during traversal (e.g. for so-called lateral joins and to optimise REST operations).

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. Intermediate results were all some kind of Query. Version 7 and later handles this differently, and executables work with rowsets instead of queries. All SQL query text, whether coming from the database file or from an interactive user, is immediately parsed into structures in which identifiers of all the above types have been replaced by long defpos. Each database object reference is (or is changed during object relocation to) its defining position in the transaction log. Parsing of new SQL (new queries or object definitions) takes place in a particular transaction Context, and each reference to a shared object (that does not have a lexical position in the SQL as displayed with #)[29] is allocated a uid on the context's heap (displayed with %), and new heap uids for an instance of its structure. On Commit of a newly defined object, when lexical uids are allocated unused positions in the file position range, these heap uids are relocated to the executable range (displayed with `).

Reference to compiled objects during database load does not require reparsing of their definition, but instancing of their framed objects, using new uids in the executable range.[30]

The SQL parsing process is recursive. During the lexical (left-to-right) phase of analysis the lexer supplies defining positions for unknown SqlValue it encounters: these are used as uids during parsing[31]. 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: the query is progressively rewritten during the parse, so that at the end of parsing all object uids still in use identify specific (instance) objects.

---

[29] For uid ranges and their representation in these notes, se section 2.3. The lifetime of uids in the executable range is until the next server restart, whereas the heap is initialised on each transaction step.
[30] For Views defined in terms of other views, see the discussion and example in section 6.6 below.
[31] SqlValues use iix (a combination of lexical and defining position) as an enhanced sort of defpos.

The simplest sort of query has the form SELECT *items* FROM *something*. Here *something* will be a RowSet (e.g. a base table), and *items* defines the Domain of the result, which in general is not the same as the Domain of the FROM's RowSet. Many of the parsing routines for queries return a pair (Domain,RowSet) as information about both parts of the query (*items*, and *something*) is progressively gathered. So, ParseSelectList always creates a new Domain with a lexical position given by the position of the E of the SELECT keyword. When we reach the end of the from clause (EOF, or a matching right parenthesis), we will create the resulting rowset, whose lexical position is given by the position of the S of SELECT..

For example, in a query such as "Select b+c as d from a" , the meaning of b and c does not become clear until we reach a, so that b and c (and the expression b+c) will initially be given lexical uids (numbers greater than 2^{60}, rendered by the debugger as #n where n is the character position of the start of the identifier or the top operator of the expression. When we reach a, and discover it is a base table, a RowSet will be constructed whose defpos is also a lexical uid, which refers to an instance rowset for the contents of table a. This instance rowset will have a domain mapped from the rowType of the shared table a, which will be used to work out the meaning of the expression b+c.

Parsing proceeds from left to right, and objects are replaced one at a time when further information about them is known. For example, replacing a single select item by uid, affects all objects that refer to it, including the containing queries (rowset and domain). Queries can also be replaced (with the same uid) when conditions and filters are moved within the resulting structure. When a uid is to be replaced with another in this process, there is a context method called Replace, deals with a queue of such requests: to process a single request, the private DoReplace method deals with one request at a time, and examining all objects in the context in order of depth, modifying each according to the replacement and creating a list of objects that have been processed. Modifications to objects are carried out by adding or modifying a property in the mem field (see the Basis class).

Some RowSet modifications have side effect. For example, adding a Order or Distinct property adds an intermediate rowset to the evaluation pipeline. On adding a where clause, it may be discovered that this implies a mathes property, and maybe makes an ordering operation unnecessary. Where clauses on joins and matches conditions can often be passed down to one of the join operands. Properties that cannot be passed down to sources include domain, aggregations, and groups. It is helpful to remember that terms in the where clause may apply to the FROM table-expression: this means that the where clause in a select statement is typically applied on a source tablerowset, not on the selectrowset itself: we will see examples of this below. At the end of parsing, the only where-conditions at any level should be those whose expressions are known at that level and not lower, so at this stage the where property of a selectrowset must contain only expressions combining operands from more than one source.

6.1.1 Context and Ident management

One of the main purposes of parsing is to replace identifiers by uids. Parsing proceeds from left to right, and initially objects are referenced by an identifier chain. The cx.defs structure is designed to assist in this process, by containing at least the first component of all identifier chains that have been mentioned in the command, arranged as an ordered tree. If the start of an identifier chain identifies a well-defined object, there is no need to enter remaining components in the cx.defs tree. If not, unknown SqlValue Content objects are constructed for the components of the chain and entered into the Context, while the reference is returned as a dotted SqlValueExpr.

Forward references in SQL only occur in select lists, so some dotted expressions will be replaced later as appropriate by rowset column references (e.g. in joins), while structured objects will still have dotted expressions.

Every SqlValue has a lexical id and a uid (combined in the class Iix): the lexical is the start of the first component of the name string, and the uid will initially be the same as the lexical id. The uid will always reference a DBObject in cx.obs, but it may be undefined (an SqlValue with Content domain and from<0). Defined objects include subclasses of SqlValue, RowSet and Executable. Iix also contains a select depth.

cx.defs has type Idents: CTree<string,(Iix,Idents)>: An Ident is decomposed into its constituent string and Iix elements when entered into defs. When the uid becomes defined (possibly by replacement of the Iix part), the Idents tree is no longer consulted.

At any stage:
- We have a list of DBObjects in cx.obs, identified by uid. An
- We have a list of identifiers in cx.defs indicating a uid and list of children and their aliases.

- Several identifiers in cx.defs may lead to the same DBObject.
- Not all of the identifiers below a given one need be from that object: some might be aliases of columns of an object with the same name that is still in scope. [is this really possible?]

6.1.2 Identifier definition

When we encounter an identifier chain such as a.b.c.d, the strategy is to start with the first component and treat all of the prefixes as follows

1) *Undefined.* All parts may be undefined. In that case we want to create a chain of ForwardReference objects ending with an undefined SqlValue (i.e. with **from**<0). Each freshly defined component of the identifier chain has a lexical position, and the ForwardReference objects have suggested row Domains.
2) *Definition.* Some of the chain, e.g. a.b may have occurred before, and this part of the chain now references an object ob: what happens depends on ob:
   a) If ob is a rowset, then probably the new item c is a column or column alias
   b) if ob is a procedure, function, or loop identifier, then c is a local variable
   c) If ob is ForwardReference, adjust its suggested Domain by adding a new leaf.
3) *Found.* If the entire chain refers to a well-defined DBObject the result is the uid of this immutable DBObject: there is no need to modify the lexical position or uid it already has. Nor is any action needed in respect of the prefixes used to reference it: they no longer matter. There are no changes to obs or defs.

In both cases 1) and 2) the newly defined SqlValue is entered into obs and defs, and in all cases the identifier chain is discarded.

6.1.3 Alias and Subquery

When we find a column alias, we copy the new column reference with the alias id.

Subqueries create new rowsets and are treated as rowsets: columns are placed in obs and defs as usual. The columns of a rowset comprise the columns indicated in its select list (sometimes called the display of the domain).

The scoping rules for references attempt to follow the SQL standard. An expression in rowset rs can have column reference u as an operand if one of the following apply:

1) u is an alias for an earlier column of rs
2) u is a column of a source of rs
3) rs is an operand in a possibly iterated join and u is a column of an earlier join operand (so-called lateral join case)
4) rs is a subquery in the select list of a rowset es and u is a column of a source of es

The above also applies for where-condition in non-aggregating rowsets. For an aggregating rowset, a where-condition rs can have a column reference u as an operand if one of the following applies:

1) u is an alias of a grouped column of rs
2) case 2), 3) or 4) above applies

A having condition for rs can have column reference u as an operand if one of the following applies

1) u identifies, or is contained in an expression that matches, a grouping column of rs
2) case 3) or 4) above applies

An ordering expression for rs can have a column reference u as an operand if u is a column of rs.

A parenthesized comma-separated list of columns is a row.

A subquery with a single non-parenthesised column can be treated as a simple expression in a select list (i.e. not a table or row). In a join, a subquery with one column can be a table. Elsewhere such a subquery can be a list of values. Subqueries with more than one column give a table in all contexts. A subquery sq can be a source ss of a rowset rs, in which case the select list of sq becomes the list of columns of ss.

6.1.4 Replacement rules

When we find out more, the DBObject will be replaced: lexical uid won't change, but the uid may be replaced by that of the resolved object. The rules for replacement are as follows:

1. We replace the object identified by a.b.c.d with the first well-defined DBObject that it can represent. The DBObject is well-defined if it is a Variable, a RowSet, a Cursor, a Table, an SqlStar, or an SqlValue that has a well-defined source.

2. Aliases are replaced with the referenced DBObject as soon as this is known.

3. When we replace an object, with another whose uid is different, the depth information in the Context ensures that changes cascade to all occurrences of the old (lexical) uid. The parent information is also updated, and the depth information is fixed if the depth has also changed.

4. We try to avoid having redundant well-defined DBObjects. Uids related by join conditions are not redundant. If two DBObjects with different uids are guaranteed to have the same value or structure in a given context, the **matches** resp **matching** properties for that context should make this clear.

6.1.5 References and Resolution

When we reach a table or view reference item, it is instanced. Then we traverse the cx.defs information and replace any undefined items by the new instance information. For every single replacement (cx.Replace())

(a) the non-well-defined identifier is replaced by the well-defined one,

(b) the changes cascade to all referencing objects (using the depth mechanism) and

(c) to parent SqlValues by updating the from information; finally

(d) the context's depth information is also updated.

Some columns added during instancing may be referenced later in where-conditions, groupings, ordering etc. These may in turn contain subqueries etc. The lexical id of any referenced identifiers (unless they already have sone) will be the first reference in the command string. The effect of this is that the lexical id of any DBObject is its first occurrence in the SQL input if any.

Column values sv in select lists are resolved once the FROM table expression has been computed. The containing table is noted in the **from** property. Rowsets later in the query (r.defpos>sv.from) will regard such identifiers as having known values during evaluation.

6.1.6 A worked example

An example will help to explain the analysis 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:

23	AUTHOR	a Table
34	ID	for 23[INTEGER}
57		an Index (ID) for 23
78	ANAME	for 23[CHAR]
123	BOOK	a Table
132	ID	for 123[INTEGER]
155		an Index (ID) for 123
175	AID	for 123[INTEGER]
200		an Index (AID) referencing 57
217	TITLE	for 123[CHAR]

The uid for a database object read from the database file will be its defining position in the file (the position in the transaction log e.g. 23 for AUTHOR). The Log shows the physical records in the database file, with the definer's names for these objects.[32]

Insert a few records in these tables:

insert into author values (1,'Dickens'),(2,'Conrad')

insert into book(aid,title) values (1,'Dombey & Son'),(2,'Lord Jim'),(1,'David Copperfield')[33]

Then, when we parse

```
123456789012345678901234
SELECT aname FROM author
```

aname is initially constructed as an unknown SqlValue with uid #8 and domain CONTENT. The uid for a new object in a query is its lexical position in the command (e.g. #8 for an identifier starting at character position 8 in the source), or its transaction id, (e.g. !0 for the first persistent object created in the transaction), or a heap uid (e.g. %0 for the first object not in any of these categories).

When the parser reaches the FROM keyword (at the start of ParseFromClause()), the objects in the Context (cx.obs) so far are:

```
{(#8=SqlValue #8 CONTENT ANAME From:#1,
  %0=Domain TABLE (#8) Display=1[#8,CONTENT])}
```

where we see that ANAME already has a From:#1, yet to be built, and we have a very tentative definition of the domain defined by the query. The defs table in the Context is just

```
        {ANAME=(2:#8,);}
```

The 2 in the cx.defs entry is the "select depth", which becomes important for more complex queries. When we get well into ParseTableReferenceItem, at about line 6181in Parser.cs, we look up the base table AUTHOR

```
        {Table 23 Domain `2 Definer=-502 LastChange=78:`2 Indexes:((34)57) KeyCols: (34=True)}
```

The Table in this version is a Compiled object. The database contains no users yet, so the definer is just the default -502. Such objects contain sub-objects in a field called Framing, which here contains just the table's Domain object `2

```
        {Framing (`2 Domain TABLE (34,78)[34,INTEGER],[78,CHAR])}
```

The uid `2 is in a special range for Compiled subobjects. (Such uids depend on the order of compilation by the server. If the table definition had been the first one read from the file, this uid would be `0.) At line 6206 of the Parser, a RowSet for the table reference #19 has been constructed and we have in cx.obs

```
{(23=Table 23 Domain `0 Definer=-502 LastChange=78:`2 Indexes:((34)57) KeyCols: (34=True),
  34=TableColumn 34 INTEGER Definer=-502 LastChange=34 -511 Table:23,
  78=TableColumn 78 CHAR Definer=-502 LastChange=78 -508 Table:23,
  #8=SqlValue ANAME #8 CONTENT From:#1,
  #19=TableRowSet #19:%3 Indexes=[(%1)57] key (%1) targets: 23=#19 From: #19 Target=23
       SRow:(34,78) Target:23 AUTHOR,
  `2=Domain TABLE (34,78)[34,INTEGER],[78,CHAR],
  %0=Domain TABLE (#8) Display=1[#8,CONTENT],
  %1=SqlCopy %1 INTEGER ID From:#19 copy from 34,
  %2=SqlCopy %2 CHAR ANAME From:#19 copy from 78,
  %3=Domain TABLE (%1,%2) Display=2[%1,INTEGER],[%2,CHAR])}
```

The blue colour indicates objects that were read from the database file on first load of the database and are shared with any references to these objects. In black we have a TableRowSet for the reference we have just found in the From clause #19 with columns for this instance. We use paler colours to indicate objects unchanged from a previous list (we will use red to indicate any changes during analysis).

cx.defs at this point contains

```
{ANAME=(2:#8,);AUTHOR=(2:#19,ANAME=(2:#19|%2,);ID=(2:#19|%1,););ID=(2:#19|%1,);}
```

---

[32] There is no need to have objects to define standard data types such as INTEGER and CHAR.

[33] Note that this insert statement does not provide the book ID (a primary key). Pyrrho will supply suitable values.

We can see an entry for the AUTHOR table, and that columns ANAME and ID have been been instanced as %1 and %2.

The next major stage in parsing is the analysis of the table expression following the FROM keyword, identifying the column names in the given query. By line 5956 of Parser.cs we have:

```
{(23=Table 23 Domain `0 Definer=-502 LastChange=78:`0 Indexes:((34)57) KeyCols: (34=True),
  34=TableColumn 34 INTEGER Definer=-502 LastChange=34 -511 Table=23,
  78=TableColumn 78 CHAR Definer=-502 LastChange=78 -508 Table=23,
  #1=SelectRowSet #1:%0 key (%1) targets: 23=#19 From: #19 Source: #19,
  #8=SqlCopy #8 CHAR ANAME From:#19 copy from 78,
  #19=TableRowSet #19:%3 Indexes=[(%1)57] key (%1) targets: 23=#19 From: #19 Target=23
       SRow:(34,78) Target:23 AUTHOR,
  `0=Domain TABLE (34,78)[34,INTEGER],[78,CHAR],
  %0=Domain TABLE (#8) Display=1[#8,CHAR],
  %1=SqlCopy %1 INTEGER ID From:#19 copy from 34,
  %3=Domain TABLE (%1,#8) Display=2[%1,INTEGER],[#8,CHAR])}
```

In this case, we have already completed the analysis, as parsing has reached the end of the query:

For the next example, we look at the identifier chains that often occur in SQL. Identifier chains in the select list tend to be forward references, resolved when the table expression is complete. Consider the following query where we have a join of AUTHOR and BOOK.

```
        1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234
SELECT aname,b.title AS c FROM author a, book b WHERE a.id=b.aid
```

Restart the server (for reproducible uids in these notes). Before parsing the From clause, at line 6017, we have

```
{(#8=SqlValue ANAME #8 CONTENT From:#1,
  #14=ForwardReference B #14 CONTENT Definer=-501 LastChange=#14,
  #16=SqlValue TITLE #16 CONTENT From:#1 Alias=C,
  %0=Domain TABLE (#8,#16) Display=2[#8,CONTENT],[#16,CONTENT])}
```

At line 5956, just before parsing the where clause, we have identified the select list and created the RowSets:

```
{(23=Table 23 Domain `0 Definer=-502 LastChange=78:`0 Indexes:((34)57) KeyCols: (34=True),
  34=TableColumn 34 INTEGER Definer=-502 LastChange=34 -511 Table=23,
  78=TableColumn 78 CHAR Definer=-502 LastChange=78 -508 Table=23,
  123=Table 123 Domain `1 Definer=-502 LastChange=217:`1 Indexes:((132)155;(175)200)
       KeyCols: (132=True,175=True),
  132=TableColumn 132 INTEGER Definer=-502 LastChange=132 -511 Table=123,
  175=TableColumn 175 INTEGER Definer=-502 LastChange=175 -511 Table=123,
  217=TableColumn 217 CHAR Definer=-502 LastChange=217 -508 Table=123,
  #1=SelectRowSet #1:%0 targets: 23=#32,123=#42 Source: #40,
  #8=SqlCopy #8 CHAR ANAME From:#32 copy from 78,
  #16=SqlCopy #16 CHAR TITLE From:#42 Alias=C copy from 217,
  #32=TableRowSet #32:%3 Indexes=[(%1)57] key (%1) targets: 23=#32 From: #40 Target=23
       SRow:(34,78) Target:23 AUTHOR Alias: A,
  #40=JoinRowSet #40:%8 targets: 23=#32,123=#42 CROSS First: #32 Second: #42,
  #42=TableRowSet #42:%7 Indexes=[(%4)155,(%5)200] key (%4) targets: 123=#42 From: #42
       Target=123 SRow:(132,175,217) Target:123 BOOK Alias: B,
  `0=Domain TABLE (34,78)[34,INTEGER],[78,CHAR],
  `1=Domain TABLE (132,175,217)[132,INTEGER],[175,INTEGER],[217,CHAR],
  %0=Domain TABLE (#8,#16) Display=2[#8,CHAR],[#16,CHAR],
  %1=SqlCopy %1 INTEGER A.ID From:#32 copy from 34,
  %3=Domain TABLE (%1,#8) Display=2[%1,INTEGER],[#8,CHAR],
  %4=SqlCopy %4 INTEGER B.ID From:#42 copy from 132,
  %5=SqlCopy %5 INTEGER AID From:#42 copy from 175,
  %7=Domain TABLE (%4,%5,#16) Display=3[%4,INTEGER],[%5,INTEGER],[#16,CHAR],
  %8=Domain TABLE (%1,#8,%4,%5,#16) Display=5[%1,INTEGER],[#8,CHAR],[%4,INTEGER],[%5,INTEGER],
       [#16,CHAR])}
```

Note that (of course) we have different uids for the columns called ID in the different tables, and new identifier chains have been introduced to disambiguate them in the results. After the where conditions have been parsed (line 6005), the rowSets have been transformed and uids #55 to #60 identified:

```
{(..
  #1=SelectRowSet #1:%0 targets: 23=#32,123=#42 Source: #40,
```

```
#8=SqlCopy #8 CHAR ANAME From:#32 copy from 78,
#16=SqlCopy #16 CHAR TITLE From:#42 Alias=C copy from 217,
#32=TableRowSet #32:%3 Indexes=[(#55)57] key (#55) order (#55) targets: 23=#32 From: #40
    Target=23 SRow:(34,78) Target:23 AUTHOR Alias: A,
#40=JoinRowSet #40:%8 where (#59) matching (#55=(#60),#60=(#55)) targets: 23=#32,123=#42
    INNER JoinCond: (#59) First: #32 Second: #42 on #55=#60,
#42=TableRowSet #42:%7 Indexes=[(#60)200,(%4)155] key (#60) order (#60) targets: 123=#42
    From: #42 Target=123 SRow:(132,175,217) Target:123 BOOK Alias: B,
#55=SqlCopy #55 INTEGER A.ID From:#32 copy from 34,
#59=SqlValueExpr #59 BOOLEAN Left:#55 Right:#60 #59(#55=#60),
#60=SqlCopy #60 INTEGER AID From:#42 copy from 175,
`0=Domain TABLE (34,78)[34,INTEGER],[78,CHAR],
`1=Domain TABLE (132,175,217)[132,INTEGER],[175,INTEGER],[217,CHAR],
%0=Domain TABLE (#8,#16) Display=2[#8,CHAR],[#16,CHAR],
%3=Domain TABLE (#55,#8) Display=2[#55,INTEGER],[#8,CHAR],
%4=SqlCopy %4 INTEGER B.ID From:#42 copy from 132,
%7=Domain TABLE (%4,#60,#16) Display=3[%4,INTEGER],[#60,INTEGER],[#16,CHAR],
%8=Domain TABLE (#55,#8,%4,#60,#16) Display=5[#55,INTEGER],[#8,CHAR],[%4,INTEGER],
    [#60,INTEGER],[#16,CHAR])}
```

The row orderings, noted as required for the inner join, have not resulted in the creation of OrderedRowSets because the join condition operands are table keys.

```
SQL> SELECT aname,b.title AS c FROM author a, book b WHERE a.id=b.aid
|-------|-----------------|
|ANAME  |C                |
|-------|-----------------|
|Dickens|Dombey & Son     |
|Dickens|David Copperfield|
|Conrad |Lord Jim         |
|-------|-----------------|
SQL>
```

The analysis results in an efficient computation of the Join.

6.2 RowSets and Context

In v7 executables that operate on tables and views (such as SelectStatement, or InsertStatement) are generally associated with rowsets rather than queries. In previous versions, the rowset associated with a query often had the same defining position (uid) and the Context maintained separate lists for object uids and rowset uids. This departure from uniqueness of uids persists in v7. The top-level query in an SQL statement (usually a CursorSpecification) thus typically has the same defpos or uid as the SelectRowSet that contains its result, and a From that targets a base table or view has the same defpos as the TransitionRowSet that manages an associated CRUD operation (this common uid is now called Nuid, while the table or view is called Target).

RowSets give access to a Cursor for traversing a set of rows. Unless an operation such as sorting requires all its data, the rowset rows are not computed until traversal begins. Recall that queries, rowsets and cursors are all immutable data structures, rowsets and queries are nested structures, with SQL commands and results at the top level, and base table references at the bottom: each level contains a reference to an immuable structure at the next level down. A cursor contains a reference to its own rowset. This arrangement requires rowsets to have uids, and the Context keeps track of them. SystemRowSets are the same for all databases, and have negative uids.

RowSets may have assertions: SimpleColumns, ProvidesTarget, AssignTarget, MatchesTarget. When the RowSet is constructed, its rowType is compared with that of its source, and the resulting assertion speeds up the per-row computation of Cursor values.

RowSet class	Comments
SystemRowSet	System Tables as defined in Manual sec 8. Many Cursor types
TableRowSet	Targets a bae table in the database

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

Syntax	RowSet class	Comments
From	TrivialRowSet	Static results: rowType from command
From	SelectedRowSet	Accesses base table:
JoinPart	JoinRowSet	Cursor classes for different join types
QueryExpression	MergeRowSet	

QuerySearch	TransitionRowSet	For Delete. TransitionRowSet has defpos #0
QuerySpecification	SelectRowSet	
SqlInsert	TransitionRowSet	TransitionRowSet has defpos #0
TableExpression	TableExpRowSet	
UpdateSearch	TransitionRowSet	TransitionRowSet has defpos #0
VALUES	ExplicitRowSet	

Other RowSets are given uids as required from the different volatile uid ranges (see section 2.3).

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

The context contains mutable objects required during parsing, rowset building and execution. These include an ObTree cache of current objects. During parsing this collection has an index called depths that arranges the DBObjects according to depth (logical dependents). During rowset traversal the context contains a set of cursors. During creation of a cursor at one level, the source finder overwrites the rowsets finder: this is restored once the cursor has been constructed.

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

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

For such transformations, there is an Apply method, that seeks to apply a set of properties to a rowset, to see if applying one or more of these changes is valid and whether it can be based down to the rowsets source or whether the pipeline can be simplified.

Some RowSet classes have a Build method: notably OrderedRowSet, SelectRowSet, RoutineCallRowSet RestRowSet, and RestRowUsing. As the method's name suggests, these rowsets perform an initial traversal of their source to yield a set of rows that can be more simply traversed later. SelectRowSet only needs the initial build if it has aggregations. The OrderedRowSet remembers the immutable cursors constructed during this initial traversal, as these can help with updatable joins and views.

Some rowsets keep track of available indexes to their rows (importantly, TableRowSet and JoinRowSet) and use an available index to improve traversal performance.

6.2.1 TransitionRowSet and TableActivation

TransitionRowSets are used for Insert, Update and Delete operations, usually in association with a subclass of Context called TableActivation. In this section we provide a worked example to illustrate their operation, while a later section covers trigger operation. The transition row set concept is specified in the SQL standard, and the following notes explain Pyrrho's implementation.

While queries generally fetch values from base tables, the TransitionRowSet prepares new physical records to modify TableRowsets, called targets. A TransitionCursor has a rowType similar to those of the last section, whose uids correspond to SqlValues, and a cursor is to all intents and purposes a row in the result of the query. We recall that the result of a query is a derived table called a rowset, and a Cursor is a subclass of TRow.

The TransitionCursor has a field called TargetCursor which is also a subclass of TRow but whose column uids identify TableColumns, and it is used to create a new Record for the target table. In the code, if trc is a TransitionCursor, then trc._tgc is the associated TargetCursor, and trc._tgc._rec is the Record that will be written to the database on commit (for insert and update) or marked deleted (for delete).

The columns of TableRowSets correspond directly to the columns of the base tables (as lists). The TransitionRowSet has maps TargetTrans and TransTarget which give the correspondence between the SqlValue uids and the TableColumn uids involved in these processes.

An SqlInsert operation also provides a set of rows to be inserted in the table. Standard SQL provides syntax for specifying columns so that the columns of these rows need to be mapped to the columns of the base table. This is handled in the From constructor.

As a first example, consider the following (starting with an empty database):

```
        1         2         3
12345678901234567890123456789012
create table a (b int, c char)
insert into a(c) values ('Three')
```

By the conclusion of parsing the insert statement, but before executing it, (in Transaction.Execute()) the Context contains the following objects:

```
{(23=Table 23 Domain `2 Definer=-502 LastChange=51:`2,
 29=TableColumn 29 INTEGER Definer=-502 LastChange=29 -511 Table=23,
 51=TableColumn 51 CHAR Definer=-502 LastChange=51 -508 Table=23,
 #1=SqlInsert #1 Target: #13 Value: %4 Columns: [%1],
 #13=TableRowSet #13:%2 targets: 23=#13 From: #13 Target=23 SRow:(29,51) Target:23 A,
 #18=#25,
 #25=SqlRow #25 Domain %5 ,
 #26=Three,
 `2=Domain TABLE (29,51)[29,INTEGER],[51,CHAR],
 %0=SqlCopy %0 INTEGER B From:#13 copy from 29,
 %1=SqlCopy %1 CHAR C From:#13 copy from 51,
 %2=Domain TABLE (%0,%1) Display=2[%0,INTEGER],[%1,CHAR],
 %3=Domain TABLE (%1) Display=1[%0,INTEGER],[%1,CHAR],
 %4=SqlRowSet %4:%6 targets: 23=#13 SqlRows [#25],
 %5=Domain ROW (#26)[#26,CHAR],
 %6=Domain TABLE (%1) Display=1[%0,INTEGER],[%1,CHAR])}
```

Here the entries in blue are shared objects read from the database (the `2 entry was constructed for Table 23 on commit: if the database is read from disk it will be set up as `0).

The Execute method begins by creating a Context (in fact a subclass of Context called Activation) for executing any constraints that may be defined on the table, its columns, or their domains. It then leads (line 342) to #1.Obey() (line 3177 of Executable.cs) for the target TableRowSet #13 and the data SqlRowSet %4. The TableRowSet is associated with the reference to table 23 in at character position #13 in the query (if there are multiple references to a table in a query, they get different instances).

SqlInsert.Obey() looks at the targets of the insert (for a modifiable join there will be more than one), and constructs a set `ts` of TargetActivations to control the insert operation. For a table target, the Insert() method constructs a TableActivation for the modification. In general, the construction of each TableActivation requires setting up the machinery for verifying constraints and managing cascades, but in this case the only thing it does is to create (at Activation.cs line 269) the TransitionRowSet object for the single target 23:

```
  {TransitionRowSet %7:%2 targets: 23=#13 Data: %4 Target: 23}
```

Back in SqlInsert.Obey(), (line 3184 of Executable.cs) the next stage traverses the supplied list of rows with an SqlCursor `ib`. In this case the single row ('Three') has the SqlCursor `{(%1=Three) %4}}`, and for each TargetActivation, we call EachRow() at line 3192. In this case, this takes us to Activation.cs, line 488, where cursors trc `{(%0= Null,%1=Three) %7}`, and trc._tgc is `{(29= Null,51=Three) %7}`, and tgc._rec is a TableRow with these values. newRow is set to these values, and as there are no triggers in this example, a new Record `{Record !0[23]: 29= 51=Three}` is made from newRow at line 516, which will be committed to the database.

```
SQL> insert into a(c) values ('Three')
1 records affected in tt
SQL> table a
|-|-----|
|B|C    |
|-|-----|
| |Three|
|-|-----|
SQL>
```

There are several contexts in operation here: the TransitionCursor is listed in the SqlInsert's Activation, while the TargetCursor is listed in the TableActivation. This is why there is no difficulty in having the same uid %7 for the two cursors above.

6.2.2 Aggregate functions

Aggregate functions in Pyrrho use a Register structure to accumulate partial results during building of TableRowSet, JoinRowSet, and RestRowSet. The Register structure allows for the issue that for some aggregate functions (e.g. SUM, MAX) the data type of the result depends on the values added. For WindowRowSet the Register also contains a copy of the previous cursor.

Building is required if there are aggregations and involves a preliminary traversal of the source to calculate the register values, indexed by group key or window key. The having clause if present will contain expressions containing aggregation functions depending on the non-key columns, and this may require additional registers for the aggregate functions they contain. Following Building, the resulting rows are traversed with selection but without computation. Where clauses apply before Building and may not contain aggregations.

An SqlValue is an aggregation if

A1 it is an aggregate SqlFunction instance, or

A2 an expression or predicate that contains at least one operand that is an aggregation.

If any column of a domain has aggregations, then all its columns are aggregations or groupings.

If a rowset has groups, it must have aggregations. The arguments of an aggregate function cannot be aggregations from the same rowset.

There is a method for enumerating the aggregate SqlFunction uids contained in an SqlValue. Every SqlValue identifies the rowset that defines it. Aggregations are not normally evaluated until all the rowsets containing their operands have been built: the list of such rowsets is called await, and the context has a property called awaiting that lists the aggregations in that state, which is initialised at the start of traversal of a rowset. If there are any awaiting, there is a preliminary building traversal. If the aggregation is listed there, a call on Eval will recurse to operands, will call StartCounter/AddIn on any aggregating SqlFunctions found in the recursion, and will return a value based on the partial sums. If the await collection is limited to a single source, the aggregation can be pushed down to that source. The from property of the aggregation indicates the rowset where the expression was created: its domain will contain aggs (or for RestRowSets, remoteAggregations) which together will list all of the aggregating SqlFunction uids that are required, and their registers and group keys will also be listed in the funcs property of the Context during building: the funcs structure provides the actual values of the SqlFunctions after building.

Once this preliminary traversal is complete, the resulting rowset is traversed, filling in the remaining columns using the Registers of the aggregated functions. At this point the having clause is checked to see whether this row should be included in the results.

Here is a worked example, based on a test contributed by Fritz Laux. The full data is in the distribution (in Pyrrho\doc\tests.txt, lines 73-113). The table is created as follows:

```
[CREATE TABLE people (Id INT(11) NOT NULL, Name VARCHAR(50) , Salary NUMERIC(10,2) ,
country VARCHAR(50), city VARCHAR(50) , PRIMARY KEY (Id) ); ]
```

and following the insertion of some data, we have a query

```
        1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
select city, avg(Salary), count(*) as numPeople from people
        7         8         9        10        11
12345678901234567890123456789012345678901234567890012345
where country = 'GER' group by city having count(*) > 4;
```

At the start of the Build method for the SelectRowSet result (line 2771 in RowSet.cs), the context contains

```
{(23=Table 23 Domain `2 Definer=-502 LastChange=182:`2 Indexes:((49)207) KeyCols: (49=True),
  34=Domain INTEGER Prec=11,
  49=TableColumn 49 Domain 34 Definer=-502 LastChange=49 34 Table=23 Not Null,
  72=Domain CHAR Prec=50,
  86=TableColumn 86 Domain 72 Definer=-502 LastChange=86 72 Table=23,
  111=Domain NUMERIC Prec=7 Scale=2,
  127=TableColumn 127 Domain 111 Definer=-502 LastChange=127 111 Table=23,
  154=TableColumn 154 Domain 72 Definer=-502 LastChange=154 72 Table=23,
  182=TableColumn 182 Domain 72 Definer=-502 LastChange=182 72 Table=23,
  #1=SelectRowSet #1:%11 key (%1) where (#75) having (%8) matches (#67=GER) groupSpec: #89
```

```
         groupings (#92) having (%8) GroupCols(#8) targets: 23=#54 From: #54 Source: #54,
#8=SqlCopy #8 Domain 72 CITY From:#54 copy from 182,
#14=SqlFunction #14 NUMERIC AVG From:#1 AVG(#18),
#18=SqlCopy #18 Domain 111 SALARY From:#54 copy from 127,
#27=SqlFunction #27 INTEGER COUNT From:#1 Alias=NUMPEOPLE COUNT(#33) as NUMPEOPLE,
#33=1,
#54=TableRowSet #54:%6 Indexes=[(%1)207] key (%1) where (#75) matches (#67=GER)
      targets: 23=#54 From: #54 Target=23 SRow:(49,86,127,154,182) Target:23 PEOPLE,
#67=SqlCopy #67 Domain 72 COUNTRY From:#54 copy from 154,
#75=SqlValueExpr #75 BOOLEAN Left:#67 Right:#77 #75(#67=#77),
#77=GER,
#89=GroupSpecification #89(#92),
#92=Grouping #92 Domain %10 GROUP (#8=0),
#104=SqlFunction #104 INTEGER COUNT COUNT(#110),
#110=1,
#113=SqlValueExpr #113 Domain %7 Left:#104 Right:#115 #113(#104>#115),
#115=4,
`2=Domain TABLE (49,86,127,154,182)[49,Domain INTEGER Prec=11],[86,Domain CHAR Prec=50],
      [127,Domain NUMERIC Prec=7 Scale=2],[154,Domain CHAR Prec=50],[182,Domain CHAR Prec=50],
%0=Domain TABLE (#8,#14,#27) Display=3[#8,Domain CHAR Prec=50],[#14,NUMERIC],[#27,INTEGER]
      Aggs (#14,#27),
%1=SqlCopy %1 Domain 34 ID From:#54 copy from 49,
%2=SqlCopy %2 Domain 72 NAME From:#54 copy from 86,
%6=Domain TABLE (%1,%2,#18,#67,#8) Display=5[%1,Domain INTEGER Prec=11],
      [%2,Domain CHAR Prec=50],[#18,Domain NUMERIC Prec=7 Scale=2],
      [#67,Domain CHAR Prec=50],[#8,Domain CHAR Prec=50],
%7=Domain BOOLEAN Aggs (#104),
%8=SqlValueExpr %8 Domain %9 Left:#27 Right:#115 %8(#27>#115),
%9=Domain BOOLEAN Aggs (#27),
%10=Domain ROW (#8)[#8,Domain CHAR Prec=50],
%11=Domain TABLE (#8,#14,#27|%1,%2,#18,#67) Display=3[#8,Domain CHAR Prec=50],[#14,NUMERIC],
      [#27,INTEGER],[%1,Domain INTEGER Prec=11],[%2,Domain CHAR Prec=50],
      [#18,Domain NUMERIC Prec=7 Scale=2],[#67,Domain CHAR Prec=50] Aggs (#14,#27),
%12=SelectStatement %12 Union=#1)}
```

(Note that the where condition #75 has been pushed down to the TableRowSet.)

There are two main loops in the Build method. The first (lines 2795-2823) traverses the source rowSet (#54 in this case) and builds a catalogue of Registers for the aggregated functions and for each value of the grouping keys. For example, the very first cursor from the source rowset is

```
{(%1=61,%2=Tom,#18=50000.00,%4=GER,#8=Berlin) #54}
```

giving a grouping key of `{(Berlin)}`. The registers are constructed for this rowset in the context, by the AddIn methods called for aggregation functions at line 2820. The structure used by the Context is called funcs and is indexed by the aggregation function uid, the grouping key, and the SelectRowSet uid.

The second loop (lines 2827-2861) traverses the cx.funcs register collection and for each grouping key that was found, places a row in the SelectRowSet if the having condition is satisfied. This set of rows is added to the SelectRowSet object in the context.

At the end of the Build method we have constructed the following rows:

```
{(0=(#8=Berlin,#14=60333.3333333333,#27=6),
  1=(#8=Munich,#14=61833.3333333333,#27=6))}
```

During traversal of the results, GroupingBookmarks are constructed for these values, and the result is shown below:

```
E:\PyrrhoDB70\Pyrrho>pyrrhocmd f1
SQL> select city, avg(Salary), count(*) as numPeople from people where country = 'GER' group by city having count(*) > 4;
|------|----------------|---------|
|CITY  |AVG             |NUMPEOPLE|
|------|----------------|---------|
|Berlin|60333.3333333333|6        |
|Munich|61833.3333333333|6        |
|------|----------------|---------|
```

6.2.3 Views

Views are compiled objects whose SQL definition is translated into RowSet terms. As with Table references, each reference to the View in a query results in a separate instance of the view with different column uids, but for views the compiled object can be a complex system of rowsets. The resulting instance is then reviewed for possible simplification based on the surrounding details (selection of rows and columns, new column matches from joins etc). There is a worked example in section 6.5.

6.2.4 RestViews

RestViews are created using a modified syntax for the CREATE VIEW statement. There are three changes: (a) an OF clause that specifies a rowtype for the view, in the same format as for defining the columns of a table; (b) the keyword GET; and (c) a USING clause that nominates a table. For details see the Pyrrho manual. Metadata fields are used on the RestView object to supply further information such as the URL to be used, and which remote DBMS is expected.

There are two RowSet classes used in the RESTView implementation, RestRowSet and RestRowSetUsing. Building a RestRowSetUsing constructs a union of RestRowSets based on the contents of the using table, which supplies a defaultUrl for each. Otherwise the defaultUrl comes from the RestView metadata.

There are currently two implementations of RESTView in the Pyrrho server with slight differences in operation, described in section 6.10 below, based respectively on transaction time overlap and ETag matching. The mechanism is selected by the metadata. For compatibility with previous Pyrrho versions, the URL of a simple RestView may be given by vi.description.

A query may target more than one RestView. During analysis, the uids in the query are associated with RestRowSets that supply them.

When we build the RestRowSetUsing, for each row of the using table we call RestRowSet.Build to create the remote SQL and perform the round trip (generated for each contributor, as they might have different sqlAgents). The RestRowSet value will be a set of rows that is added to the RestRowSetUsing's value. Traversal of the RestRowSetUsing is then a simple traversal of the resulting array of rows.

In computing aggregations for RestRowSet, Pyrrho returns some extra fields in the REST results so that e.g., selecting an average from a restview can be computed by a single row from each contributor. The mechanism supports grouping. Otherwise these extra fields are not used. See section 8.3 for details.

For worked examples of RestView processing, see section 6.10.

6.3 SqlValue vs TypedValue

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. The result of evaluation is a TypedValue, and this has many subclasses. During rowset traversal, each Cursor computes the TypedValue for its current row, and the context maintains a list of the currently open cursors.

Activations are subclasses of Context constructed for executing compiled statements (Executable has many subclasses), and form stacks during execution. In general, activations in the current stack may have different roles, (definer) users, and permissions.

6.4 Persistent Stored Modules

A big change in this version of Pyrrho is that compiled objects (including triggers, stored procedures etc) are only parsed once per cold start. The uids given to DBObjects generated by the parser during Load are all in the executable range (shown as `0,.. is these notes). These generated objects are not added to the Database objects tree, but stored (in Framing) in the trigger or procedure DBObject. When execution of a procedure body or trigger is required, these objects can simply be added to the activation context (by the line of code cx.obs += ob.framing) to make them available in the context. (References to code modules do not require instancing, unlike Views, whose RowSet instances will be transformed by their context similarly to the very first example 6.1.6 above.)

A particularly simple case is afforded by check constraints. Like procedures and views, the definition is compiled on its first occurrence and the resulting compiled objects are retained in the parent object's framing field (the constraint may have been defined for a table, a table column, or a domain).

As an example, let us look in detail at the processes described in section 3.5.2, for a check definition within an explicit transaction. Starting with an empty database and begin transaction:

```
        1         2         3         4
12345678901234567890123456789012345678901234567890
begin transaction
create table ca(a char,b int check (b>0))
```

During ParseColumnCheckConstraint, cx.parse is set to Compile (line 2484 of Parser.cs). Then during the construction of the Physical PCheck2 object, (line 2508) the context has:

```
{(-511=Domain INTEGER,
 -508=Domain CHAR,
 !0=Table !0 Domain `2 Definer=-502 LastChange=!2:`2,
 !1=TableColumn !1 CHAR Definer=-502 LastChange=!1 -508 Table=!0,
 !2=TableColumn !2 INTEGER Definer=-502 LastChange=!2 -511 Table=!0,
 `2=Domain TABLE (!1,!2)[!1,Domain CHAR],[!2,Domain INTEGER],
 `11=SqlCopy `11 INTEGER B From:_ copy from !2,
 `13=SqlValueExpr `13 BOOLEAN Left:`11 Right:`16 `13(`11>`16),
 `16=0)}
```

We see the proposed table and its columns, and the parsed version of the check expression with uids `11, `13 and `16 in the executable range. PCheck is a subclass of Compiled, so it already has a framing field containing these in addition to the domain for the new table:

```
{Framing (`2 Domain TABLE (!1,!2)[!1,Domain CHAR],[!2,Domain INTEGER],
 `11 SqlCopy `11 INTEGER B From:_ copy from !2,
 `13 SqlValueExpr `13 BOOLEAN Left:`11 Right:`16 `13(`11>`16),
 `16 0)}
```

When we construct the Check object (in PCheck2.Install, line 237 of PCheck.cs) we get a Check object with the above framing:

```
{Check !3 Definer=-502 LastChange=!3 From.Target=!2 Source=(b>0) Search=`13}
```

The Check constraint is usable in this form within the explicit transaction (after commit the !0 uids will be replaced by file positions). The next step (line 239) adds the Check constraint !3 to the transaction, and cx.Install(nc,p) changes column !2 in the context to

```
{TableColumn !2 INTEGER Definer=-502 LastChange=!2 -511 Table=!0 Checks:(!3=True)}
```

and adds it to the transaction. After installing the Check object in the transaction, the context is cleared.

Suppose the next statement is:

```
4      5        6        7
45678901234567890123456789012345 6
insert into ca values('Neg',-99)
```

Let us trace through what happens. From example 6.2.1, we know that SqlInsert uses EachRow to construct a new row for a table, and it uses a TargetCursor. So, this time, place a break point at the start of TargetCursor.New (line 5197 of RowSet.cs). At this point we have the following in the TableActivation context (lexical positions currently run on during a transaction as indicated by the ruler above):

```
{(!0=Table !0 Domain `2 Definer=-502 LastChange=!2:`2,
 !1=TableColumn !1 CHAR Definer=-502 LastChange=!1 -508 Table=!0,
 !2=TableColumn !2 INTEGER Definer=-502 LastChange=!2 -511 Table=!0 Checks:(!3=True),
 #44=SqlInsert #44 Target: #56 Value: %3,
 #56=TableRowSet #56:%2 targets: !0=#56 From: #56 Target=!0 SRow:(!1,!2) Target:!0 CA,
 #59=#65,
 #65=SqlRow #65 Domain %5 ,
 #66=Neg,
 #72=SqlValueExpr #72 UNION Right:#73 #72(-#73),
 #73=99,
 `2=Domain TABLE (!1,!2)[!1,Domain CHAR],[!2,Domain INTEGER],
 %0=SqlCopy %0 CHAR A From:#56 copy from !1,
 %1=SqlCopy %1 INTEGER B From:#56 copy from !2,
 %2=Domain TABLE (%0,%1) Display=2[%0,Domain CHAR],[%1,Domain INTEGER],
 %3=SqlRowSet %3:%6 targets: !0=#56 SqlRows [#65],
 %5=Domain ROW (#66,#72)[#66,Domain CHAR],[#72,Domain UNION],
 %6=Domain TABLE (%0,%1) Display=2[%0,Domain CHAR],[%1,Domain INTEGER],
 %7=TransitionRowSet %7:%2 targets: !0=#56 Data: %3 Target: !0)}
```

The objects from the check constraint !3 have not yet been added to the context. At line 5212, the row contents for the insert have computed in vs

```
{(!1=Neg,!2=-99)}
```

In %7.TargetCursor.New() we find the constraint on the second tablecolumn !2 at line 5241.

```
{Check !3 Definer=-502 LastChange=!3 From.Target=!2 Source=(b>0) Search=`13}
```

The next two lines 5255 and 5256 adds the check framing and current values `vs = {(!1=Neg,!2=-99)}` to the context. The next line retrieves the SqlValueExpr `13 from the context, and this evaluates to false, raising the exception.

6.5 Trigger Implementation

As elsewhere in Pyrrho, parsing of executable SQL takes place on definition[34] of a database object. Definition is within the command processing that creates the object, or the database load on server start-up, and the associated executable uids of compiled objects will differ for these two cases. The compiled objects are placed in the framing property of the associated DBObject, and these are added to the context when the DBObject is instanced.

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

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

```
create table xa(b int,c int,d char)
create table xb(tot int)
insert into xb values (0)
[create trigger ruab before update on xa referencing old as mr new as nr
for each row begin atomic update xb set tot=tot-mr.b+nr.b; set d='changed'
end]
[create trigger riab before insert on xa
for each row begin atomic set c=b+3; update xb set tot=tot+b end]
```

If these have all been entered in auto-commit mode, the log contains[35]:

Pos	Record Type	Contents
23	PTable XA	
30	PColumn3 B	
52	PColumn3 C	
75	PColumn3 D	
116	PTable XB	
123	PColumn3 TOT	
165	Record 165[116]	123=0
195	Trigger RUAB	Trigger RUAB Update, Before, EachRow on 23 begin atomic update xb set tot=tot-mr.b+nr.b; set d='changed' end
305	Trigger RIAB	Trigger RIAB Insert, Before, EachRow on 23: begin atomic set c=b+3; update xb set tot=tot+b end

The Trigger definitions are stored in the log in source form, but the compiled contents are now in memory in the `framing` field of the Trigger object. The following "readable" version of this field (for RIAB, following a server restart[36]) is from the debugger, and contains the items that will be cached in the context when the trigger is used. They have uids in the executable range notated here `0,..[37]:

```
{Framing (
{(`64=TableRowSet `64:`68 targets: 23=`64 From: `64 Target=23 SRow:(30,52,75) Target:23 XA,
  `65=SqlCopy `65 135 B From:`64 copy from 30,
  `66=SqlCopy `66 135 C From:`64 copy from 52,
  `67=SqlCopy `67 CHAR D From:`64 copy from 75,
  `68=Domain TABLE (`65,`66,`67) Display=3[`65,135],[`66,135],[`67,CHAR],
  `69=CompoundStatement  `69(`70,`85),
```

---

[34] In previous versions, parsing would also take place for each role granted the compiled object: this is no longer required because execution will always use the definer's role.

[35] In all these worked examples, the actual file positions depend on many non-reproducible details including timestamps. As they used to say in car-maintenance manuals, "Your mileage may vary."

[36] The uids and activation identifiers are different for a newly defined objects, as previously noted.

[37] In these listings Visual Studio has displayed INTEGER as 135 (which is (int)Sqlx.INTEGER). Why?

```
  `70=AssignmentStatement `70 `66=`80,
   `80=SqlValueExpr `80 135 Left:`65 Right:`83 `80(`65+`83),
    `83=3,
  `85=UpdateSearch `85 Target: `86,
   `86=TableRowSet `86:`88 targets: 116=`86 From: `86 Assigs:(UpdateAssignment Vbl: `87 Val:
`99=True) Target=116 SRow:(123) Target:116 XB,
    `87=SqlCopy `87 135 TOT From:`86 copy from 123,
    `88=Domain TABLE (`87) Display=1[`87,135],
    `99=SqlValueExpr `99 135 Left:`87 Right:`65 `99(`87+`65),
    `106=WhenPart `106 Cond: `107 Stms: (`69))}
Result `86}}}}
```

The CompoundStatement `69 (highlighted) implements the fragment of trigger source "set c=b+3; update xb set tot=tot+b".

Executables are placed in an Activation prior to execution, and Activations form a stack: each routine instance has its own set of objects, while inheriting other sorts of object from the Context..

The next step in the test is an insert command:
```
            1         2         3         4
1234567890123456789012345678901234567890 1
insert into xa(b,d) values (7,'inserted')
```

Let us place a break point at the start of Transaction.Execute (line 339 of Transaction.cs), and trace through everything that happens here. Recall that several sorts of insert triggers can be defined on a table, and possibly more than one of any kind, possibly defined by different roles. So the trigger implementation must take into account the command context (with the session role), the target conext (using the table definer's role) and maybe several different trigger contexts (the trigger definers' roles). After parsing the insert command (at line 339) we have in the main Context cx:

```
{(23=Table 23 Domain `0 Definer=-502 LastChange=75:`0
        Triggers:(Insert, Before, EachRow=(305=True),Update, Before, EachRow=(195=True)),
  30=TableColumn 30 INTEGER Definer=-502 LastChange=30 -511 Table=23,
  52=TableColumn 52 INTEGER Definer=-502 LastChange=52 -511 Table=23,
  75=TableColumn 75 CHAR Definer=-502 LastChange=75 -508 Table=23,
  #1=SqlInsert #1 Target: #13 Value: %5 Columns: [%0,%2],
  #13=TableRowSet #13:%3 targets: 23=#13 From: #13 Target=23 SRow:(30,52,75) Target:23 XA,
  #21=#28,
  #28=SqlRow #28 Domain %6 ,
  #29=7,
  #31=inserted,
  `0=Domain TABLE (30,52,75)[30,INTEGER],[52,INTEGER],[75,CHAR],
  %0=SqlCopy %0 INTEGER B From:#13 copy from 30,
  %1=SqlCopy %1 INTEGER C From:#13 copy from 52,
  %2=SqlCopy %2 CHAR D From:#13 copy from 75,
  %3=Domain TABLE (%0,%1,%2) Display=3[%0,INTEGER],[%1,INTEGER],[%2,CHAR],
  %4=Domain TABLE (%0,%2) Display=2[%0,INTEGER],[%1,INTEGER],[%2,CHAR],
  %5=SqlRowSet %5:%7 targets: 23=#13 SqlRows [#28],
  %6=Domain ROW (#29,#31)[#29,INTEGER],[#31,CHAR],
  %7=Domain TABLE (%0,%2) Display=2[%0,INTEGER],[%1,INTEGER],[%2,CHAR])}
```

The command to be executed is the SqlInsert highlighted above. Let us trace what happens next in SqlInsert #1.Obey() (Executable.cs at line 3174). It works in an Activation (a sort of Context), whose identifier is 8 here. The next couple of lines identify the target and data for the insert as

```
tg <- {TableRowSet #13:%3 targets: 23=#13 From: #13 Target=23 SRow:(30,52,75) Target:23 XA}
data <- {SqlRowSet %5:%7 targets: 23=#13 SqlRows [#28]}
```

respectively. SqlInsert.Obey constructs a list of TargetActivations ts,

```
ts <- {(23=TableActivation 9)}
```

and this time there is just the one TableActivation (with identifier 9). To see this in the debugger it is easiest to step over to the line where ta has been identified (line 3190 or so). Then ta is TableActivation 9 and in addition to the above objects ta.obs has

```
{TransitionRowSet %8:%3 targets: 23=#13 Data: %5 Target: 23}
```

The TableActivation also has a list of TriggerActivations called acts,

```
{(305=TriggerActivation 10)}
```

and act[0] is TriggerActivation 10. The TriggerActivation constructor added the table's triggers to its objects, so that it now has all three sets of objects above (from the trigger framing, the main Context, and the TransitionRowSet %8). .

The traversal of the *data* TableRowSet %5 has just begun (at line 3185), and the first cursor is the SqlCursor:

```
ib {(%0=7,%2=inserted) %5}
```

In the EachRow method of TableActivation 9, we create a transition cursor, with target cursor and proposed record (see line 512-8 of TableActivation in Activation.cs):

```
trc {(%0=7,%1= Null,%2=inserted) %8}
trc._tgc {(30=7,52= Null,75=inserted) %8}
tgc._rec.vals {(30=7,52= Null,75=inserted)}
```

Recall (from section 6.2.1) that the transitionCursor is in the SqlInsert's activation 8 while the targetCursor is for the TableActivation 9.

At line 503, EachRow() calls Triggers() to obey the trigger RIAB. In Triggers() we start traversal of the list of trigger activations: there is just one, {TriggerActivation 10} and it calls Exec().

At line 1077 of Activation.cs, Exec() begins to execute the CompoundStatement `69 in TriggerActivation 10, and this (at line 259 of Executable.,cs) creates a local Activation 11. CompoundStatement.Obey() calls ObeyList(). For convenience. place a break point on the Obey() call in ObeyList (line 62).

The first statement executed is the {AssignmentStatement `70 `66=`80}. Recall from above,

```
 `65=SqlCopy  `65 INTEGER B From:`64 copy from 30,
 `66=SqlCopy  `66 INTEGER C From:`64 copy from 52,
 `67=SqlCopy  `67 CHAR D From:`64 copy from 75,
 `68=Domain TABLE (`65,`66,`67) Display=3[`65,INTEGER],[`66,INTEGER],[`67,CHAR],
 `69=CompoundStatement  `69(`70,`85),
 `70=AssignmentStatement `70 `66=`80,
 `80=SqlValueExpr `80 INTEGER Left:`65 Right:`83 `80(`65+`83),
 `83=3,
```

We see that obs[`80] is obs[`65]+obs[`83]. obs[`65] is an SqlCopy to copy the current value for table column 30,while obs[`83] is the literal 3, so we get 7 and add 3 to get 10. The effect of the AssignmentStatement is thus to store {10} as the context's value for SqlCopy `66, in activation 11.

At this point, therefore, Activation 11 has the following **values**:

```
{(23=(30=7,52= Null,75=inserted) %8,30=7,52= Null,75=inserted,`65=7,`66=10,`67=inserted)}
```

The next step in the CompoundStatement is {UpdateSearch `85 Target: `86}. UpdateSearch.Obey() calls `86.Update(), which creates a new TableActivation 12. Again, when using the debugger, it is easiest to step over to line 3361 to look at ta. This has just one further object

```
{TransitionRowSet %9:`88 targets: 116=`86 Data: `86 Target: 116}
```

but no triggers, as XB has none defined. The traversal cursor ib is {(`87=0) `86} and ta.EachRow constructs

```
trc {(`87=0) %9}
tgc {(123=0) %9}
rc.vals {(123=0)}
```

The updates in EachRow (at line 541-543) now construct the new values list with newRow {(123=7)}.

At line 574, the transition and target cursors for %9 are updated for the new values, and the new targetCursor {(123=7) %9} is installed in activation 12 at line 576.

The new Update gets constructed (at line 576) as {Update 165[116]: 123=7 Prev:165}. In case there are after-triggers, the newTables field for 12 is updated with the new TableRow for XB. But there are none, and we return to `85.Obey(), where, at line 3365, *we overwrite 11's version of the transaction with 12's* (it has the new Update !1). This version is also used to update the list of targetActivations at line 3366 (it is retained for further rows, and there aren't any). The modified activation 11 is returned to `69.ObeyList(). As there are no more statements in the CompoundStatement, this also returns.

Activation 11 was created just for the CompoundStatement execution, so (at line 267) we call 11.SlideDown(), which updates 10's values:

```
{(23=(30=7,52= Null,75=inserted) %8,
 30=7,
 52= Null,
 75=inserted,116=(123=7) %9,
 123=7,
 `65=7,
 `66=10,
 `67=inserted,
 `87=7)}
```

We are then back in 10.Exec(). At line 1088, we transmit the triggered changes in the row values to update the transitionCursor for %6. In particular, the result of the highlighted assignment above gets placed in the newRow for table XA at line 1075. Each SlideDown in the context stack overwrites the transaction, so that for example in 9.SlideDown, the new Physicals {(!0=TriggeredAction 305,!1=Update 165[116]: 123=7 Prev:165)} accumulate in the parent context.

We have completed execution of each-row-before triggers, and now the insert of the new row can take place, in 9.EachRow(). At line 516 this gives a new Record for the transaction: {Record !2[23]: 30=7,52=10,75=inserted}.

This means that at line 517 the transaction now has the following Physical objects to be committed:

```
{(!0=TriggeredAction 305,
  !1=Update 165[116]: 123=7 Prev:165,
  !2=Record !2[23]: 30=7,52=10,75=inserted)}
```

We immediately get activation 8 to adopt these, and they will eventually be adopted by the root context #1. The command is finished, and the following records are committed along with the triggeredAction note at position 421 in the transaction log:

| 402 | Update 165[116] | 123=7 Prev:165 |
| 421 | Record 421[23]: | 30=7,52=10,75=inserted |

In the test, there is now another insert statement, which we apply to the database

```
insert into xa(b, d) values(9, 'Nine')
```

This adds some more entries in the log, notably:

| 477 | Update 165[116] | 123=16 Prev:402 |
| 496 | Record 496[23]: | 30=9,52=12,75=Nine |

So that your uids and activation numbers match the numbers in the following notes, restart the server again.

In the next part of the test, we have an update statement. From the definition of the update trigger above

```
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 d='changed' end
```

we see that this will demonstrate the operation of OLD ROW and NEW ROW.

```
          1         2         3         4
1234567890123456789012345678901234567890 1
update xa set b=8,d='updated' where b=7
```

As far as at Activation 8.EachRow(), the execution proceeds similarly to the above example. At this point we construct

```
trc {(#15=7,%1=10,#19=inserted) %4}
tgc {(30=7,52=10,75=inserted) %4}
rc.vals {(30=7,52=10,75=inserted)}
```

and Step A of the update processing (line 542 of Activation.cs) sets up the oldrow (-295) and newrow (-293) values in the TableActivation 9. At line 545 we have

```
9.cursors {(23=(#15=7,%1=10,#19=inserted) #8,
           #8=(#15=7,%1=10,#19=inserted) #8,
           %4=(30=7,52=10,75=inserted) %4)}

9.values   {(-295=(30=7,52=10,75=inserted) %4,
             -293=(30=8,52=10,75=updated),
              23=(30=7,52=10,75=inserted) %4,
              30=8,
              52=10,
```

```
                       75=updated)}
```

At line 545, we call the triggers. When TriggerActivation 10.Exec() is called, this is how things stand:

```
10.obs
{(23=Table 23 Domain `0 Definer=-502 LastChange=75:`0 Triggers:(Insert, Before,
EachRow=(305=True),Update, Before, EachRow=(195=True)),
  30=TableColumn 30 INTEGER Definer=-502 LastChange=30 -511 Table=23,
  52=TableColumn 52 INTEGER Definer=-502 LastChange=52 -511 Table=23,
  75=TableColumn 75 CHAR Definer=-502 LastChange=75 -508 Table=23,
  195=Trigger 195 Domain `12 Definer=-502 LastChange=195 TrigType=Update, Before, EachRow
On=23,
  #1=UpdateSearch #1 Target: #8,
  #8=TableRowSet #8:%3 where (#38) matches (#15=7) targets: 23=#8 From: #8
       Assigs:(UpdateAssignment Vbl: #15 Val: #17=True,
               UpdateAssignment Vbl: #19 Val: #21=True) Target=23 SRow:(30,52,75)
       Target:23 XA,
  #15=SqlCopy #15 INTEGER B From:#8 copy from 30,
  #17=8,
  #19=SqlCopy #19 CHAR D From:#8 copy from 75,
  #21=updated,
  #38=SqlValueExpr #38 BOOLEAN Left:#15 Right:#39 #38(#15=#39),
  #39=7,
  `0=Domain TABLE (30,52,75)[30,INTEGER],[52,INTEGER],[75,CHAR],
  `2=SqlOldRow `2 Domain `12 MR From:`8,
  `3=SqlNewRow `3 Domain `12 NR From:`8,
  `8=TableRowSet `8:`12 targets: 23=`8 From: `8 Target=23 SRow:(30,52,75) Target:23 XA,
  `9=SqlCopy `9 INTEGER B From:`8 copy from 30,
  `10=SqlCopy `10 INTEGER C From:`8 copy from 52,
  `11=SqlCopy `11 CHAR D From:`8 copy from 75,
  `12=Domain TABLE (`9,`10,`11) Display=3[`9,INTEGER],[`10,INTEGER],[`11,CHAR],
  `13=SqlField `13 INTEGER B Parent=`2 Field=30,
  `14=SqlField `14 INTEGER C Parent=`2 Field=52,
  `15=SqlField `15 CHAR D Parent=`2 Field=75,
  `16=SqlField `16 INTEGER B Parent=`3 Field=30,
  `17=SqlField `17 INTEGER C Parent=`3 Field=52,
  `18=SqlField `18 CHAR D Parent=`3 Field=75,
  `19=CompoundStatement  `19(`20,`50),
  `20=UpdateSearch `20 Target: `21,
  `21=TableRowSet `21:`23 targets: 116=`21 From: `21
       Assigs:(UpdateAssignment Vbl: `22 Val: `42=True)
       Target=116 SRow:(123) Target:116 XB,
  `22=SqlCopy `22 INTEGER TOT From:`21 copy from 123,
  `23=Domain TABLE (`22) Display=1[`22,INTEGER],
  `34=SqlValueExpr `34 INTEGER Left:`22 Right:`13 `34(`22-`13),
  `42=SqlValueExpr `42 INTEGER Left:`34 Right:`16 `42(`34+`16),
  `50=AssignmentStatement `50 `11=`56,
  `56=changed,
  `58=WhenPart `58 Cond: `59 Stms: (`19),
  %1=SqlCopy %1 INTEGER C From:#8 copy from 52,
  %3=Domain TABLE (#15,%1,#19) Display=3[#15,INTEGER],[%1,INTEGER],[#19,CHAR],
  %4=TransitionRowSet %4:%3 where (#38) targets: 23=#8 Data: #8 Target: 23)}

10.cursors {(23=(#15=7,%1=10,#19=inserted) #8,
             #8=(#15=7,%1=10,#19=inserted) #8,
             %4=(30=7,52=10,75=inserted) %4)}
```

10.Exec() prepares for execution on the current row by copying the values of old row and new row from TableActivation 9 above. At line 1069, its values are

```
10.values {(-295=(30=7,52=10,75=inserted) %4,
            -293=(30=8,52=10,75=updated),
            23=(`9=7,`10=10,`11=inserted) %4,
            30=8,
            52=10,
            75=updated,
            `2=(30=7,52=10,75=inserted) %4,
            `3=(30=8,52=10,75=updated),
            `9=7,
            `10=10,
            `11=inserted)}
```

At line 1077 of Activation.cs, the TriggerActivation starts to Obey the CompoundStatement `19.

The first step is the UpdateSearch `20, and it calls the Update method on TableRowSet `21 targeting XB. It creates TableActivation 12 with its TransitionRowSet

```
{TransitionRowSet %5:`23 targets: 116=`21 Data: `21 Target: 116}
```

The traversal begins, and 12.EachRow creates cursors:

```
trc {(`22=16) %5}

tgc {(123=16) %5}
rc.vals {(123=16)}
```

Evaluating the update `21's UpdateAssignment highlighted above involves using RUAB's old and new rows values `2 and `3 above. The evaluation proceeds as follows:

```
`21   :=              `42
      = (    `34         +`16)
      = ((`22  -`13)    +`16)
      = (([123]-[`2].30)+([`3].30)
      = ((16   - 7)     + 8)
[123] := 17
```

So in 12.values, we get [123]:=17, and this completes the calculation of the values of XB's new row. The new TableRow is now constructed, and _Update() generates the Update record for XB [116] in the transaction physicals: {(!1= {Update 421[23]: 30=8,75=changed Prev:421})}. The transitionCursor for 12 is also updated (in case there are further cascading changes). and also installed in TableActivation 12. This completes the UpdateSearch `20.

At this point, we have the following cursors in Activation 11, where the transition cursor is highlighted:

```
11.cursors {(23=(%0=7,%1=10,%2=inserted) %5,
            140=(`21=16) `24,
            #8=(%0=7,%1=10,%2=inserted) #8,
            `20=(`21=16) `20,
            `24=(`21=16) `24,
            %5=(%0=7,%1=10,%2=inserted) #8,
            %7=(`7=7,`8=10,`9=inserted) %7,
            %8=(123=17) %8)}
```

The second step in the trigger is an Assignment statement in Activation 11. This updates 11's values for `11 to changed.

```
11.values {(-295=(30=7,52=10,75=inserted) %4,
           -293=(30=8,52=10,75=updated),
           23=(30=7,52=10,75=inserted) %4,
           30=8,
           52=10,
           75=updated,116=(123=17) %5,
           123=17,
           `2=(30=7,52=10,75=inserted) %4,
           `3=(30=8,52=10,75=updated),
           `9=8,
           `10=10,
           `11=changed,
           `22=17)}
```

Back in 10.EachRow(), it remains to merge the updates into the new TableRow in step D. At line 587 we have a further item for the Transaction Commit: {Update 421[23]: 30=8,75=changed Prev:421}

In the transaction log, this shows as

| 548 | Update 165[116] | 123=17 Prev:477 |
| 567 | Update 421[23]: | 30=8,75=changed Prev: 421 |

This completes the discussion of the Update trigger demonstration.

In the test a third table XC and a third trigger for XA are defined[38], and the next step is a Delete operation, demonstrating an INSTEAD OF statement-level trigger and the use of the OLD TABLE feature of SQL. (See section 3.4.2.)

```
create table xc(totb int,totc int)
```

---

[38] The latest version of the test has two *instead of* triggers. This is simplified here for brevity.

```
[create trigger sdai instead of delete on xa referencing old table as ot
for each statement begin atomic insert into xc (select b,c from ot) end]
```

616	PTable XC	
624	PColumn3 TOTB	for 616(0)[INTEGER]
661	PColumn3 TOTC	for 616(1)[INTEGER]
697	Trigger SDAI	Delete, Instead, EachStatement on 23,OT=old table : begin atomic insert into xc (select b,c from ot) end

After server restart, the framing for 697 is as follows:
```
{Framing (
   `109=TransitionTable `109:`117 old from 23,
   `113=TableRowSet `113:`117 targets: 23=`113 From: `113 Target=23 SRow:(30,52,75) Target:23
         XA,
   `114=SqlCopy `114 135 B From:`113 copy from 30,
   `115=SqlCopy `115 135 C From:`113 copy from 52,
   `116=SqlCopy `116 CHAR D From:`113 copy from 75,
   `117=Domain TABLE (`114,`115,`116) Display=3[`114,135],[`115,135],[`116,CHAR],
   `118=CompoundStatement  `118(`119),
   `119=SqlInsert `119 Target: `122 Value: `150,
   `122=TableRowSet `122:`125 targets: 616=`122 From: `122 Target=616 SRow:(624,651) Target:616
         XC,
   `123=SqlCopy `123 135 TOTB From:`122 copy from 624,
   `124=SqlCopy `124 135 TOTC From:`122 copy from 651,
   `125=Domain TABLE (`123,`124) Display=2[`123,135],[`124,135],
   `130=SelectRowSet `130:`143 Source: `109,
   `143=Domain TABLE (`114,`115) Display=2[`114,135],[`115,135],
   `147=SelectStatement `147 Union=`130,
   `148=SqlValueSelect `148 Domain `143  (`130),
   `149=Domain TABLE (`123,`124) Display=2[`114,135],[`115,135],[`123,135],[`124,135],
   `150=SelectedRowSet `150:`149 targets: 616=`122 Source: `130,
   `151=WhenPart `151 Cond: `152 Stms: (`118)) Result `150}
```

We see the **old** table reference has defined a TransitionTable `109 and a TableRowSet `113, along with SqlCopy for the columns. The test run is:
```
            1         2         3
1234567890123456789012345678901
delete from xa where d='changed'
```

Much as before, at QuerySearch #1.Obey() Activation 9 has

```
{(23=Table 23 Domain `0 Definer=-502 LastChange=75:`0 Triggers:(Insert, Before,
       EachRow=(305=True),Update, Before, EachRow=(195=True),Delete, Instead,
       EachStatement=(697=True)),
  30=TableColumn 30 INTEGER Definer=-502 LastChange=30 -511 Table=23,
  52=TableColumn 52 INTEGER Definer=-502 LastChange=52 -511 Table=23,
  75=TableColumn 75 CHAR Definer=-502 LastChange=75 -508 Table=23,
 #1=QuerySearch #1 Target: #13,
 #13=TableRowSet #13:%3 where (#23) matches (#22=changed) targets: 23=#13 From: #13 Target=23
       SRow:(30,52,75) Target:23 XA,
 #22=SqlCopy #22 CHAR D From:#13 copy from 75,
 #23=SqlValueExpr #23 BOOLEAN Left:#22 Right:#24 #23(#22=#24),
 #24=changed,
  `0=Domain TABLE (30,52,75)[30,INTEGER],[52,INTEGER],[75,CHAR],
 %0=SqlCopy %0 INTEGER B From:#13 copy from 30,
 %1=SqlCopy %1 INTEGER C From:#13 copy from 52,
 %3=Domain TABLE (%0,%1,#22) Display=3[%0,INTEGER],[%1,INTEGER],[#22,CHAR])}
```

This time, we have a statement level trigger, and this is called when a TableActivation for the delete operation is constructed. So, when #1.Obey() is setting up a list ts of TableActivations for the delete operation, %5.Delete() calls the constructor for TableActivation 10. This calls Triggers() at line 493 to execute the statement-level **before** and **instead** triggers if any. Our trigger 697 is an *instead of* trigger, so is called on the second invocation of Triggers() at line 473. 10's TransitionRowSet is

```
  {TransitionRowSet %5:%3 where (#23) targets: 23=#13 Data: #13 Target: 23}
```

and it has set up its TriggerActivation 11 which has all the above objects (the TransitionRowSet, the objects from Activation 9, and the objects from the trigger framing):

```
{(..,
  `0=Domain TABLE (30,52,75)[30,INTEGER],[52,INTEGER],[75,CHAR],
  `109=TransitionTable `109:`117 old from 23,
  `113=TableRowSet `113:`117 targets: 23=`113 From: `113 Target=23 SRow:(30,52,75) Target:23
```

```
         XA,
 `114=SqlCopy `114 INTEGER B From:`113 copy from 30,
 `115=SqlCopy `115 INTEGER C From:`113 copy from 52,
 `116=SqlCopy `116 CHAR D From:`113 copy from 75,
 `117=Domain TABLE (`114,`115,`116) Display=3[`114,INTEGER],[`115,INTEGER],[`116,CHAR],
 `118=CompoundStatement  `118(`119),
 `119=SqlInsert `119 Target: `122 Value: `150,
 `122=TableRowSet `122:`125 targets: 616=`122 From: `122 Target=616 SRow:(624,651) Target:616
         XC,
 `123=SqlCopy `123 INTEGER TOTB From:`122 copy from 624,
 `124=SqlCopy `124 INTEGER TOTC From:`122 copy from 651,
 `125=Domain TABLE (`123,`124) Display=2[`123,INTEGER],[`124,INTEGER],
 `130=SelectRowSet `130:`143 Source: `109,
 `143=Domain TABLE (`114,`115) Display=2[`114,INTEGER],[`115,INTEGER],
 `147=SelectStatement `147 Union=`130,
 `148=SqlValueSelect `148 Domain `143  (`130),
 `149=Domain TABLE (`123,`124) Display=2[`114,INTEGER],[`115,INTEGER],[`123,INTEGER],
         [`124,INTEGER],
 `150=SelectedRowSet `150:`149 targets: 616=`122 Source: `130,
 `151=WhenPart `151 Cond: `152 Stms: (`118),
 %0=SqlCopy %0 INTEGER B From:#13 copy from 30,
 %1=SqlCopy %1 INTEGER C From:#13 copy from 52,
 %3=Domain TABLE (%0,%1,#22) Display=3[%0,INTEGER],[%1,INTEGER],[#22,CHAR],
 %5=TransitionRowSet %5:%3 where (#23) targets: 23=#13 Data: #13 Target: 23)}
```

11.Exec(), as before, sets up tables for old and new if requested. In this case, TransitionTableRowSet `109 is for the old table that has been referenced[39], and replaces(!) the TransitionTable `109 in this activation (line 1101):

<code style="color:red">{TransitionTableRowSet `109:`117 targets: 23=#13 OLD}</code>

At line 1077, 11.Exec() begins to obey the trigger body in Activation 12, CompoundStatement `118. This is just SqlInsert `119, and `119.Obey() retrieves the target and data for the operation:

```
tg {TableRowSet `122:`125 targets: 616=`122 From: `122 Target=616 SRow:(624,651) Target:616 XC}
data {SelectedRowSet `150:`149 targets: 616=`122 Source: `130}
```

It sets up its own target list ts {(648=TableActivation 13)}. As before, we step over a few times to the ta.EachRow statement, noting that the traversal cursor ib is the SelectCursor {(`123=8,`124=10) `150}, and that TableActivation 13 has the additional object

```
{TransitionRowSet %6:`125 targets: 616=`122 Data: `150 Target: 616}
```

13.EachRow creates the transition cursors:

```
trc {(`123=8,`124=10) %6}
tgc {(624=8,651=10) %6}
rc.vals {(624=8,651=10)}
```

As before, recall that trc is installed in Activation 12, and tgc in TableActivation 13. Table XC has no triggers, so the insert operation simply creates a new Record `{Record !1[616]: 624=8,651=10}`.

Because this is an INSTEAD OF trigger, 35.Exec() returns true, and the actual Delete statement is not executed. Following Commit, the log shows

799	Record 799[616]	624=8,681=10

This completes this demonstration.

6.6 View Implementation

Because of the use case of virtual data warehousing, where (possibly behind the scenes) tables are virtual and mediated by views, it is interesting to enable views to be modifiable, that is, capable of supporting insert, update and delete. We cover modifiable views in this section. Not all views are modifiable, but a great many should be.

A View is a compiled object (class PView is a subclass of Compiled) and a view definition is compiled as soon as the server sees it, i.e. on creation by a CREATE VIEW SQL statement, or on initial Load of the database. The compiled parts of the View are held in a Framing object. We recall that the framing part is never written to disk but reconstructed in this way for each server instance.

---

[39] For a description of the transition table, see the SQL standard ISO 9075-02(2016) section 4.44.

In this section we follow the execution of the parts of test 12 that deal with views.. For simplicity we omit the earlier steps, and start with an empty database t12, and the definitions

```
create table p(q int primary key,r char,a int)
create view v as select q,r as s,a from p
```

In the transaction log these are:

23	PTable P	
29	PColumn3 Q	for 23(0)[INTEGER]
51	PIndex P	on 23(29) PrimaryKey
67	PColumn3 R	for 23(1)[CHAR]
90	PColumn3 A	for 23(2)[INTEGER]
131	PView V 131	select q,r as s,a from p

As usual, we restart the server so that complied objects and activation numbers are reproducible in these notes. Then the Framing for the View is

```
{Framing (
{(`4=SelectRowSet `4:`24 key (`7) targets: 23=`26 From: `26 Source: `26,
 `7=SqlCopy `7 INTEGER Q From:`26 copy from 29,
 `13=SqlCopy `13 CHAR R From:`26 Alias=S copy from 67,
 `20=SqlCopy `20 INTEGER A From:`26 copy from 90,
 `24=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `26=TableRowSet `26:`30 Indexes=[(`7)51] key (`7) targets: 23=`26 From: `26 Target=23
SRow:(29,67,90) Target:23 P,
 `30=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `32=SelectStatement `32 Union=`4)}
Result `4}
```

This framing is designed for selection, as in most cases a reference to the view will be within a query. In this section, we want to show that the view can also be a target for insert, update and delete.

The next statement in the test is

```
          1         2         3         4
12345678901234567890123456789012345678901234
insert into v(s) values('Twenty'),('Thirty')
```

When the transaction is about to execute this statement after parsing, the View has been "instanced" giving the following entries in the Context:

```
{(-538=TABLE,
 -505=CONTENT,
 -503=Null,
 #1=SqlInsert #1 Target: %1 Value: %31 Columns: [%10],
 #18=#24,#35,
 #24=SqlRow #24 Domain %32 ,
 #25=Twenty,
 #35=SqlRow #35 Domain %33 ,
 #36=Thirty,
 `4=SelectRowSet `4:`24 key (`7) targets: 23=`26 From: `26 Source: `26,
 `7=SqlCopy `7 INTEGER Q From:`26 copy from 29,
 `13=SqlCopy `13 CHAR R From:`26 Alias=S copy from 67,
 `20=SqlCopy `20 INTEGER A From:`26 copy from 90,
 `24=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `26=TableRowSet  `26:`30  Indexes=[(`7)51]  key  (`7)  targets:  23=`26  From:  `26  Target=23
SRow:(29,67,90) Target:23 P,
 `30=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `32=SelectStatement `32 Union=`4,
 %0=View %0 Domain %21 Definer=-502 LastChange=173 ViewDef view v as select q,r as s,a from p
Ppos: 131 Result %1,
 %1=SelectRowSet %1:%21 key (%4) targets: 23=%23 From: %23 Source: %23,
 %4=SqlCopy %4 INTEGER Q From:%23 copy from 29,
 %10=SqlCopy %10 CHAR R From:%23 Alias=S copy from 67,
 %17=SqlCopy %17 INTEGER A From:%23 copy from 90,
 %21=Domain TABLE (%4,%10,%17) Display=3[%4,INTEGER],[%10,CHAR],[%17,INTEGER],
 %23=TableRowSet  %23:%27  Indexes=[(%4)51]  key  (%4)  targets:  23=%23  From:  %23  Target=23
SRow:(29,67,90) Target:23 P,
 %27=Domain TABLE (%4,%10,%17) Display=3[%4,INTEGER],[%10,CHAR],[%17,INTEGER],
 %29=SelectStatement %29 Union=%1,
 %30=Domain TABLE (%10) Display=1[%4,INTEGER],[%10,CHAR],[%17,INTEGER],
 %31=SqlRowSet %31:%34 targets: 23=%23 SqlRows [#24,#35],
 %32=Domain ROW (#25)[#25,CHAR],
```

```
%33=Domain ROW (#36)[#36,CHAR],
%34=Domain TABLE (%10) Display=1[%4,INTEGER],[%10,CHAR],[%17,INTEGER)}
```

Here objects %1-%29 are instances of objects from the view definition and its framing. The instance of the view has unique ids (%4,%10,%17) and the shared objects `3-`31 are no longer referenced in this context.

Execute() (at line 339 in Transaction.cs) sets up a new Activation 7. This calls Obey() for the SqlInsert statement #1. #1.Obey(7) finds the target %1 and adds the data rowset %31:

```
tg {SelectRowSet %1:%21 key (%4) targets: 23=%23 From: %23 Source: %23}
data {SqlRowSet %31:%34 targets: 23=%23 SqlRows [#24,#35]}
```

It examines the targets of the From, and finds just TableRowSet %23, and calls %23.Insert() to create an Activation to do the Insert. At line 3184 of Executable.cs we see

```
ts {(23=TableActivation 8)}
```

The constructor for TableActivation 8 ensures that any dependent compiled objects are instanced (none in this case, but Table 23's Domain `0 is added)

```
`0 {Domain TABLE (29,67,90)[29,INTEGER],[67,CHAR],[90,INTEGER]}
```

and builds a TransitionRowSet for the insert operation on table 23 (the easiest place to see this is at line 3188 where we can inspect ta with the debugger). ta is a TableActivation with transition rowset _trs

```
{TransitionRowSet %35:%27 targets: 23=%23 Data: %31 Target: 23}
```

We see that none of the entries in the TableActivation mentions V as a target, so that the insert for the View V has become an Insert for the Table P 23. The procedure is therefore the same as the above discussion of Insert (e.g. section 6.4).

Traversal of the data rowset begins (line 3185 of Executable.cs). The first data cursor is `{(%10=Twenty) %31}`, and the TableActivation.EachRow method deals with it. It begins (at line 491 in Activation.cs) by creating a TransitionCursor `{(%4= Null,%11=Twenty,%19= Null) %35}}` whose TargetCursor for the Table P fills in the key column Q using Pyrrho's autokey feature, `{(`==29=1==`,67=Twenty,90= Null) %35}`. Similarly for the second row of the insert, so, following the autoCommit, we have simply

| 191 | Record 191[23] | 29=1,67=Twenty |
| 218 | Record 218[23] | 29=2,67=Thirty |

The next step in the test is

```
update v set s='Forty two' where q=1
```

This will start again with a clean Context. Similarly to the above, this time when Execute is called we have

```
{(-538=TABLE,
  -505=CONTENT,
  -503=Null,
  #1=UpdateSearch #1 Target: %1,
  #14=SqlCopy #14 CHAR R From:%23 Alias=S copy from 67,
  #16=Forty two,
  #34=SqlCopy #34 INTEGER Q From:%23 copy from 29,
  #35=SqlValueExpr #35 BOOLEAN Left:#34 Right:#36 #35(#34=#36),
  #36=1,
  `4=SelectRowSet `4:`24 key (`7) targets: 23=`26 From: `26 Source: `26,
  `7=SqlCopy `7 INTEGER Q From:`26 copy from 29,
  `13=SqlCopy `13 CHAR R From:`26 Alias=S copy from 67,
  `20=SqlCopy `20 INTEGER A From:`26 copy from 90,
  `24=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
  `26=TableRowSet `26:`30 Indexes=[(`7)51] key (`7) targets: 23=`26 From: `26 Target=23
      SRow:(29,67,90) Target:23 P,
  `30=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
  `32=SelectStatement `32 Union=`4,
  %0=View %0 Domain %21 Definer=-502 LastChange=173 ViewDef view v as select q,r as s,a from p
Ppos: 131 Result %1,
  %1=SelectRowSet %1:%21 key (#34) where (#35) matches (#34=1) targets: 23=%23 From: %23
      Source: %23 Assigs:(UpdateAssignment Vbl: #14 Val: #16=True),
  %17=SqlCopy %17 INTEGER A From:%23 copy from 90,
  %21=Domain TABLE (#34,#14,%17) Display=3[#34,INTEGER],[#14,CHAR],[%17,INTEGER],
  %23=TableRowSet %23:%27 Indexes=[(#34)51] key (#34) where (#35) matches (#34=1)
      targets: 23=%23 From: %23 Assigs:(UpdateAssignment Vbl: #14 Val: #16=True)
      Target=23 SRow:(29,67,90) Target:23 P,
```

```
%27=Domain TABLE (#34,#14,%17) Display=3[#34,INTEGER],[#14,CHAR],[%17,INTEGER],
%29=SelectStatement %29 Union=%1)}
```

We see that a matches condition and where clause have been added to the Rowsets and the where clause has been passed down to the instance TableRowSet %23 (these changes is why we need a fresh instance for every reference). Again, this reduces the View update to an ordinary table update (the assignments are obeyed around line 531 of Activation.cs). After Commit we just have

263	Update 191[23]	67=Forty two Prev:191

Next we have a simple Delete:

```
delete from v where s='Thirty'
```

Once again, we see the RowSets are just for Table P 23:

```
{(-538=TABLE,
 -505=CONTENT,
 -503=Null,
 #1=QuerySearch #1 Target: %1,
 #21=SqlCopy #21 CHAR R From:%23 Alias=S copy from 67,
 #22=SqlValueExpr #22 BOOLEAN Left:#21 Right:#23 #22(#21=#23),
 #23=Thirty,
 `4=SelectRowSet `4:`24 key (`7) targets: 23=`26 From: `26 Source: `26,
 `7=SqlCopy `7 INTEGER Q From:`26 copy from 29,
 `13=SqlCopy `13 CHAR R From:`26 Alias=S copy from 67,
 `20=SqlCopy `20 INTEGER A From:`26 copy from 90,
 `24=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `26=TableRowSet `26:`30 Indexes=[(`7)51] key (`7) targets: 23=`26 From: `26 Target=23
       SRow:(29,67,90) Target:23 P,
 `30=Domain TABLE (`7,`13,`20) Display=3[`7,INTEGER],[`13,CHAR],[`20,INTEGER],
 `32=SelectStatement `32 Union=`4,
 %0=View %0 Domain %21 Definer=-502 LastChange=173 ViewDef view v as select q,r as s,a from p
       Ppos: 131 Result %1,
 %1=SelectRowSet %1:%21 key (%4) where (#22) matches (#21=Thirty) targets: 23=%23 From: %23
       Source: %23,
 %4=SqlCopy %4 INTEGER Q From:%23 copy from 29,
 %17=SqlCopy %17 INTEGER A From:%23 copy from 90,
 %21=Domain TABLE (%4,#21,%17) Display=3[%4,INTEGER],[#21,CHAR],[%17,INTEGER],
 %23=TableRowSet %23:%27 Indexes=[(%4)51] key (%4) where (#22) matches (#21=Thirty)
       targets: 23=%23 From: %23 Target=23 SRow:(29,67,90) Target:23 P,
 %27=Domain TABLE (%4,#21,%17) Display=3[%4,INTEGER],[#21,CHAR],[%17,INTEGER],
 %29=SelectStatement %29 Union=%1)}
```

and the Commit gives

309	Delete Record 218[23]	

For the tests on Views that are joins or joins of views, we prepare some further items:
```
insert into p(r) values('Fifty')
create table t(s char,u int)
insert into t values('Forty two',42),('Fifty',48)
create view w as select * from t natural join v
```

336	Record 336[23]	29=2,67=Fifty
380	PTable T	
387	PColumn3 S	for 380(0)[CHAR]
411	PColumn3 U	for 380(1)[INTEGER]
454	Record 454[380]	387=Forty two,411=42
484	Record 484[380]	387=Fifty,411=48
528	PView W 528	select * from t natural join v

Let us restart the server once again (for reproducibility of compiled objects and activation numbers). The framing for W is as follows:

```
{Framing (
 `36=SelectRowSet `36:`38 Indexes=[(`51)51] matching (`41=(`41)) targets: 23=`70,380=`40
       Source: `44,
 `37=SqlStar * `37 CONTENT,
 `38=Domain TABLE (`42,`51,`64) Display=3[`42,135],[`51,135],[`64,135],
 `40=TableRowSet `40:`43 targets: 380=`40 From: `44 Target=380 SRow:(387,411) Target:380 T,
 `41=SqlCopy `41 CHAR R From:`70 Alias=S copy from 67,
 `42=SqlCopy `42 135 U From:`40 copy from 411,
 `43=Domain TABLE (`41,`42) Display=2[`41,CHAR],[`42,135],
 `44=JoinRowSet `44:`77 matching (`41=(`41)) targets: 23=`70,380=`40 INNER JoinCond: (`79)
```

```
        First: `80 Second: `81 on `41=`41,
 `47=View `47 Domain `68 Definer=-502 LastChange=173
        ViewDef view v as select q,r as s,a from p Ppos: 131 Result `48,
 `48=SelectRowSet `48:`68 key (`51) targets: 23=`70 From: `70 Source: `70,
 `51=SqlCopy `51 135 Q From:`70 copy from 29,
 `64=SqlCopy `64 135 A From:`70 copy from 90,
 `68=Domain TABLE (`51,`41,`64) Display=3[`51,135],[`41,CHAR],[`64,135],
 `70=TableRowSet `70:`74 Indexes=[(`51)51] key (`51) targets: 23=`70 From: `70 Target=23
        SRow:(29,67,90) Target:23 P,
 `74=Domain TABLE (`51,`41,`64) Display=3[`51,135],[`41,CHAR],[`64,135],
 `76=SelectStatement `76 Union=`48,
 `77=Domain TABLE (`42,`51,`64|`41) Display=3[`42,135],[`51,135],[`64,135],[`41,CHAR],
 `78=SqlValueExpr `78 BOOLEAN From:`44 Left:`41 Right:`41 `78(`41=`41),
 `79=SqlValueExpr `79 BOOLEAN Left:`41 Right:`41 `79(`41=`41),
 `80=OrderedRowSet `80:`43 key (`41) order (`41) targets: 380=`40 Source: `40,
 `81=OrderedRowSet `81:`68 key (`41) order (`41) targets: 23=`70 Source: `48,
 `82=SelectStatement `82 Union=`36)
Result `36}
```

Note that the view V has been instanced as part of view W (so that W's framing contains an instanced version of V shown in pale blue).

Once again, view W will usually be used in a query, so that the above framing makes this use almost trivial. The above framing objects will be instanced to use heap uids, and the resulting instances of TableRowSets such as `40 and `70 will be updated to use actual column uids from the referencing query. The rowset review process may simplify the instance, for example the ordering steps `80 and or `81 may not be required if a where-condition limits the rows involved.

The case of inserting into a join is not interesting. Consider the next step in the test:

```
          1         2         3
1234567890123456789012345678901234567890
update w set u=50,a=21 where q=2
```

This should involve updates to both table P and table T, highlighted below. At Execute(), the context (excluding the framing items) contains

```
{(-538=TABLE,
 -505=CONTENT,
 -503=Null,
 #1=UpdateSearch #1 Target: %1,
 #14=SqlCopy #14 INTEGER U From:%5 copy from 411,
 #16=50,
 #19=SqlCopy #19 INTEGER A From:%35 copy from 90,
 #21=21,
 #30=SqlCopy #30 INTEGER Q From:%35 copy from 29,
 #31=SqlValueExpr #31 BOOLEAN Left:#30 Right:#32 #31(#30=#32),
 #32=2,..
 %0=View %0 Domain %3 Definer=-502 LastChange=577
        ViewDef view w as select * from t natural join v Ppos: 528 Result %1,
 %1=SelectRowSet %1:%3 Indexes=[(#30)51] where (#31) matches (#30=2) matching (%6=(%6))
        targets: 23=%35,380=%5 Source: %9
        Assigs:(UpdateAssignment Vbl: #14 Val: #16=True,
               UpdateAssignment Vbl: #19 Val: #21=True),
 %2=SqlStar * %2 CONTENT,
 %3=Domain TABLE (#14,#30,#19) Display=3[#14,INTEGER],[#30,INTEGER],[#19,INTEGER],
 %5=TableRowSet %5:%8 targets: 380=%5 From: %9
        Assigs:(UpdateAssignment Vbl: #14 Val: #16=True)
        Target=380 SRow:(387,411) Target:380 T,
 %6=SqlCopy %6 CHAR R From:%35 Alias=S copy from 67,
 %8=Domain TABLE (%6,#14) Display=2[%6,CHAR],[#14,INTEGER],
 %9=JoinRowSet %9:%42 where (#31) matches (#30=2) matching (%6=(%6)) targets: 23=%35,380=%5
        Assigs:(UpdateAssignment Vbl: #14 Val: #16=True,
               UpdateAssignment Vbl: #19 Val: #21=True)
        INNER JoinCond: (%44) First: %45 Second: %46 on %6=%6,
 %12=View %12 Domain %33 Definer=-502 LastChange=173
        ViewDef view v as select q,r as s,a from p Ppos: 131 Result `48,
 %13=SelectRowSet %13:%33 key (#30) matches (#30=2) targets: 23=%35 From: %35 Source: %35
        Assigs:(UpdateAssignment Vbl: #19 Val: #21=True),
 %33=Domain TABLE (#30,%6,#19) Display=3[#30,INTEGER],[%6,CHAR],[#19,INTEGER],
 %35=TableRowSet %35:%39 Indexes=[(#30)51] key (#30) matches (#30=2) targets: 23=%35
        From: %35
        Assigs:(UpdateAssignment Vbl: #19 Val: #21=True)
```

```
        Target=23 SRow:(29,67,90) Target:23 P,
 %39=Domain TABLE (#30,%6,#19) Display=3[#30,INTEGER],[%6,CHAR],[#19,INTEGER],
 %41=SelectStatement %41 Union=%13,
 %42=Domain TABLE (#14,#30,#19|%6) Display=3[#14,INTEGER],[#30,INTEGER],[#19,INTEGER],
      [%6,CHAR],
 %43=SqlValueExpr %43 BOOLEAN From:%9 Left:%6 Right:%6 %43(%6=%6),
 %44=SqlValueExpr %44 BOOLEAN Left:%6 Right:%6 %44(%6=%6),
 %45=OrderedRowSet %45:%8 key (%6) order (%6) targets: 380=%5 Source: %5
      Assigs:(UpdateAssignment Vbl: #14 Val: #16=True),
 %46=OrderedRowSet %46:%33 key (%6) order (%6) matches (#30=2) targets: 23=%35 Source: %13
      Assigs:(UpdateAssignment Vbl: #19 Val: #21=True),
 %47=SelectStatement %47 Union=%1)}
```

Notice that the OrderedRowSets are no longer referenced in this instance.

Execute() starts an Activation 8 to handle the update, and calls #1.Obey(). This sets up TableActivations for the two targets %35 and %5 of the Join (both targets are simple tables), and each constructs a TransitionRowSet whose cursors give the desired updates.

After Commit, we see in the log:

595	Update 336[23]	90=21,Prev:336
614	Update 484[380]	411=50,Prev:484

6.7 Prepared Statement implementation

For completeness, here is an example showing the implementation of PreparedStatement, again based on one of the steps in test12 with commit.

This test sets up a prepared statement Upd1 as `"update sce set a=? where b=?",` and a bit later Executes the prepared statement Upd1 with parameters `"" + 6, "'HalfDozen'".`

Start PyrrhoSvr with a breakpoint in Parse.cs near the end of the ParseSql(PreparedStatement..) method just before the call to QParams() (line 288).

Run the PyrrhoTest application with command arguments

`PyrrhoTest 12 0 commit`

From sec 3.4.2 we know that the parsing of prepared statements uses only heap uids, in a semi-persistent range of the heap space.

The second time the breakpoint in PyrrhoSvr is hit, we see that the PreparedStatement pre is

```
{PreparedStatement %41 Target: UpdateSearch %18 Target: %19 Params: %28,%38}
```

and its objects have been simply added to the current context:

```
{(`2=Domain TABLE (31,71)[31,INTEGER],[71,CHAR],
  %18=UpdateSearch %18 Target: %19,
  %19=TableRowSet %19:%22 Indexes=[(%20)53] key (%20) where (%36) matches (%21=?%38) targets:
23=%19 From: %19 Assigs:(UpdateAssignment Vbl: %20 Val: %28=True) Target=23 SRow:(31,71)
Target:23 SCE,
  %20=SqlCopy %20 INTEGER A From:%19 copy from 31,
  %21=SqlCopy %21 CHAR B From:%19 copy from 71,
  %22=Domain TABLE (%20,%21) Display=2[%20,INTEGER],[%21,CHAR],
  %28=?%28,
  %36=SqlValueExpr %36 BOOLEAN Left:%21 Right:%38 %36(%21=%38),
  %38=?%38,
  %41=PreparedStatement %41 Target: UpdateSearch %18 Target: %19 Params: %28,%38)}
```

and cx.values has become

```
{(%28=6,%38=HalfDozen)}
```

The query operator here simply collects the value from cx.values. We have a simple UpdateSearch whose target rowSet selects the row containing the value HalfDozen in column b and updates column a to the value 6.

6.8 Stored Procedure implementation

The mechanism for stored procedures is somewhat different from the previous sections, so it is worthwhile to provide a worked example. In this section we work through the test in test 18 of the PyrrhoTest program.

Again start with an empty database t18, and give the following commands to set up the test.

```
create table author(id int primary key,aname char)
```

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

```
insert into author values (1,'Dickens'),(2,'Conrad')
```

```
insert into book(authid,title) values (1,'Dombey & Son'),(2,'Lord Jim'),(1,'David Copperfield')
```

This gives the following objects in database t18 as we see from the log (we omit the PTransaction markers)::

Pos	Desc
23	PTable AUTHOR
34	PColumn3 ID for 23(0)[INTEGER]
57	PIndex AUTHOR on 23(34) PrimaryKey
78	PColumn3 ANAME for 23(1)[CHAR]
123	PTable BOOK
132	PColumn3 ID for 123(0)[INTEGER]
182	PIndex BOOK on 123(132) PrimaryKey
175	PColumn3 AUTHID for 123(1)[INTEGER]
203	PIndex2 on 123(175) ForeignKey, RestrictUpdate, RestrictDelete refers to [57]
220	PColumn3 TITLE for 148(2)[91]
265	Record 265[23]: 34=1,78=Dickens
290	Record 290[23]: 34=2,78=Conrad
332	Record 332[123]: 132=1,175=1,220=Dombey & Son
370	Record 370[123]: 132=2,175=2,220=Lord Jim
404	Record 404[123]: 132=3,175=1,220=David Copperfield

Place a breakpoint in ParseProcedureClause at line 2567. Begin a transaction, and enter the function definition:

```
begin transaction
```

```
[create function booksby(auth char) returns table(title char)
  return table (select title from author inner join book b
  on author.id=b.authid where aname=booksby.auth)]
```

The purpose of parsing the procedure clause is to create the physical procedure definition (PProcedure) to be entered in the database. This is a fairly complex process because the SQL standard requires the arity to be taken into account in looking up procedure names, so we show the details here. When parsing the parameter list and returns clause, we construct a skeleton procedure that can resolve recursive references, so at line 2595 we have:

```
{Procedure !0 Domain `8 Definer=-502 BOOKSBY Arity=1 `8 Params(`5) Body:_ Clause{}}
```

The PProcedure pp has begun to prepare the procedure framing. At this stage it contains:

```
{Framing (
  `5=FormalParameter `5 CHAR AUTH From:!0 IN,
  `7=SqlElement TITLE `7 CHAR From:!0,
  `8=Domain (title char)  TABLE (`7)[`7,CHAR] structure=_)}
```

Then cx.Add(pp) calls db.Add() which calls PProcedure.Install(), and this makes changes to the Transaction's role (for name/arity lookup) and its list of proedures (for execution). Each has a structure called **procedures**, which is initially empty. By line 2597 of Parser.cs these are respectively

```
cx.db.role.procedures {(BOOKSBY=(1=!0))}
```

```
cx.db.procedures {(!0=BOOKSBY$1⁴⁰)}
```

Next, the body of the procedure is parsed, so that by line 2398 the context contains

```
{(23=Table 23 Domain `2 Definer=-502 LastChange=78:`2 Indexes:((34)57) KeyCols: (34=True),
```

---

[40] BOOKSBY$1 is the definer's name for the procedure. Roles can have their own names for database objects.

```
 34=TableColumn 34 INTEGER Definer=-502 LastChange=34 -511 Table=23,
 78=TableColumn 78 CHAR Definer=-502 LastChange=78 -508 Table=23,
 123=Table 123 Domain `4 Definer=-502 LastChange=220:`4 Indexes:((132)155;(175)203) KeyCols:
(132=True,175=True),
 132=TableColumn 132 INTEGER Definer=-502 LastChange=132 -511 Table=123,
 175=TableColumn 175 INTEGER Definer=-502 LastChange=175 -511 Table=123,
 220=TableColumn 220 CHAR Definer=-502 LastChange=220 -508 Table=123,
 !0=Procedure !0 Domain `8 Definer=-502 BOOKSBY Arity=1 `8 Params(`5) Body:`65
      Clause{(auth char) returns table(title char) return table (select title from author
              inner join book b on author.id=b.authid where aname=booksby.auth)},
 `2=Domain TABLE (34,78)[34,INTEGER],[78,CHAR],
 `4=Domain TABLE (132,175,220)[132,INTEGER],[175,INTEGER],[220,CHAR],
 `5=FormalParameter `5 CHAR AUTH From:!0 IN,
 `7=SqlElement TITLE `7 CHAR From:!0,
 `8=Domain (title char)  TABLE (`7)[`7,CHAR] structure=_,
 `13=SelectRowSet `13:`20 matching (`23=(`30),`30=(`23)) targets: 23=`22,123=`28 Source: `26,
 `16=SqlCopy `16 CHAR TITLE From:`28 copy from 220,
 `20=Domain TABLE (`16) Display=1[`16,CHAR],
 `22=TableRowSet `22:`25 Indexes=[(`23)57] key (`23) order (`23) targets: 23=`22 From: `26
      Target=23 SRow:(34,78) Target:23 AUTHOR,
 `23=SqlCopy `23 INTEGER AUTHOR.ID From:`22 copy from 34,
 `24=SqlCopy `24 CHAR ANAME From:`22 copy from 78,
 `25=Domain TABLE (`23,`24) Display=2[`23,INTEGER],[`24,CHAR],
 `26=JoinRowSet `26:`48 where (`55) matching (`23=(`30),`30=(`23)) targets: 23=`22,123=`28
      INNER JoinCond: (`40) First: `22 Second: `28 on `23=`30,
 `28=TableRowSet `28:`32 Indexes=[(`29)155,(`30)203] key (`30) order (`30) targets: 123=`28
      From: `28 Target=123 SRow:(132,175,220) Target:123 BOOK Alias: B,
 `29=SqlCopy `29 INTEGER B.ID From:`28 copy from 132,
 `30=SqlCopy `30 INTEGER AUTHID From:`28 copy from 175,
 `32=Domain TABLE (`29,`30,`16) Display=3[`29,INTEGER],[`30,INTEGER],[`16,CHAR],
 `40=SqlValueExpr `40 BOOLEAN Left:`23 Right:`30 `40(`23=`30),
 `48=Domain TABLE (`23,`24,`29,`30,`16) Display=5[`23,INTEGER],[`24,CHAR],[`29,INTEGER],
      [`30,INTEGER],[`16,CHAR],
 `55=SqlValueExpr `55 BOOLEAN Left:`24 Right:`5 `55(`24=`5),
 `63=SelectStatement `63 Union=`13,
 `64=SqlValueSelect `64 Domain `20  (`13),
 `65=ReturnStatement `65 -> `64)}
```

The available indexes have already been used to avoid having to order the join operands.

The framing result rowset is `65. With these entries the procedure can be used within the transaction.

`select * from table(booksby('Dickens'))`

Now commit the transaction, and restart the server.

When the database is reloaded, the compiled object gets reconstructed in Compiled.OnLoad, so that the in-memory database has the same framing objects. The Procedure object itself now has a permanent defining position 465, and following a server restart the framing objects have moved somewhat so that if we repeat the above select, with a breakpoint at line 432 in Start.cs, we see that the context objects include

```
{(…
  465=Procedure 465 Domain `7 Definer=-502 LastChange=465 BOOKSBY Arity=1 `7 Params(`3) Body:`63
Clause{(auth char) returns table(title char) return table (select title from author inner join
book b on author.id=b.authid where aname=booksby.auth)},..}}
```

6.9 User-defined Types Implementation

CREATE TYPE results in two or more records in the log, a PTable with a name consisting of the column information, a PType with the type name and any other domain properties of the type, and PMethod records for its methods (initially without bodies). The structure field of the UDType refers to the PTable, and the type field of the Methods refers to the UDType. The framing objects of the UDType consist of the virtual table and method header information, while the framing objects of the methods follow the same pattern as for Procedures. Instancing a UDType brings the domain information into the Context and instances methods that have bodies.

This worked example is based on test20 of the test suite.

`create type point as (x int, y int)`

```
create type size as (w int,h int)

create type line as (strt point,en point)

[create type rect as (tl point,sz size)
  constructor method rect(x1 int,y1 int, x2 int, y2 int),
  method centre() returns point]
```

After these declarations, we have the following in the log (omitting PTransaction markers).

Pos	Desc
23	PTable (x int, y int)
42	PColumn3 X for 23(0)[INTEGER]
64	PColumn3 Y for 23(1)[INTEGER]
87	PType POINT[23]
127	PTable (w int,h int)
145	PColumn3 W for 127(0)[INTEGER]
167	PColumn3 H for 127(1)[INTEGER]
190	PType SIZE[127]
229	PTable (strt point,en point)
256	PColumn3 STRT for 229(0)[87]
282	PColumn3 EN for 229(1)[87]
307	PType LINE[242]
348	PTable (tl point,sz size)
373	PColumn3 TL for 348(0)[87]
397	PColumn3 SZ for 348(1)[190]
423	PType RECT[348]
446	Method Constructor 446=423.RECT$4(x1 int,y1 int, x2 int, y2 int)
498	Method Instance 498=423.CENTRE$0() returns point

We can see from this that the structure of a user-defined type is implemented using a base table whose name begins with a left parenthesis, and the method declarations have their own physical record type, so that the methods already have defining positions even though they have no bodies yet.

The corresponding DBObjects in the Database for a simple PType such as POINT are[41]

```
23 {VirtualTable 23 Domain `3 Definer=-502 LastChange=64:`3 RestView=_}
87 {UDType POINT TYPE (42,64)[42,135],[64,135] structure=23}
```

We will define the method bodies below: for now we just have the heading and return types:

```
446 {Method 446 Domain 423 Definer=-502 LastChange=446 RECT$4 Arity=4 423
        Params(`21,`23,`25,`27) Body:_ Clause{(x1 int,y1 int, x2 int, y2 int)}
        UDType=UDType RECT TYPE (373,397)
                [373,UDType POINT TYPE (42,64)[42,135],[64,135] structure=23],
                [397,UDType SIZE TYPE (145,167)[145,135],[167,135] structure=127]
        structure=348 MethodType=Constructor}

498 {Method 498 Domain 87 Definer=-502 LastChange=498 CENTRE$0 Arity=0 87 Params()
        Body:_ Clause{() returns point}
        UDType=UDType RECT TYPE (373,397)
                [373,UDType POINT TYPE (42,64)[42,135],[64,135] structure=23],
                [397,UDType SIZE TYPE (145,167)[145,135],[167,135] structure=127]
        structure=348 MethodType=Instance Methods: 459 RECT$4 }
```

(The copy of object 446 in object in 498 shows the previously defined method RECT$4: up-to-date lists will be used when the method is referenced.)

The body of a method is declared in SQL using a CREATE statement, such as

```
[create constructor method rect(x1 int,y1 int,x2 int,y2 int)
  begin tl=point(x1,y1); sz=size(x2-x1,y2-y1) end]
```

and the procedure body is added to the physical database in source form using a Modify record type.

When the Modify is Installed in the Database, it updates the framing of the in-memory object 459 to become like the following (shown after a server restart):

```
{Framing (
  `5=FormalParameter `5 135 X1 From:446 IN,
  `7=FormalParameter `7 135 Y1 From:446 IN,
  `9=FormalParameter `9 135 X2 From:446 IN,
  `11=FormalParameter `11 135 Y2 From:446 IN,
```

---

[41] In these listings Visual Studio has displayed INTEGER as 135 (which is (int)Sqlx.INTEGER). Why?

```
`19=CompoundStatement  `19(`21,`40),
`21=AssignmentStatement `21 373=`22,
`22=SqlDefaultConstructor `22 Domain 87  Sce:`37,
`37=SqlRow `37 Domain `38 ,
`38=Domain ROW (`5,`7)[`5,135],[`7,135],
`40=AssignmentStatement `40 397=`41,
`41=SqlDefaultConstructor `41 Domain 190  Sce:`70,
`49=SqlValueExpr `49 135 Left:`9 Right:`5 `49(`9-`5),
`62=SqlValueExpr `62 135 Left:`11 Right:`7 `62(`11-`7),
`70=SqlRow `70 Domain `71 ,
`71=Domain ROW (`49,`62)[`49,135],[`62,135])}
```

In particular, we see that the body of the procedure is given by the CompoundStatement `19. Note the calls on default constructors for Point at `22 and Size at `41. A constructor does not have a Return statement, as the fields for the new Rect are collected from local variables in the Activation.

Instantiation and execution of a method follows much the same pattern as in the previous section. Before the body of a method is executed, the values of its top-level fields are placed in the context. Expressions of form TL.X are short-circuited using a special SqlValue called SqlField, which simply selects the component of the left-hand value whose uid matches the field column. To see this in action, let us declare CENTRE:

```
[create method centre() returns point for rect
  return point(tl.x+sz.w/2,tl.y+sz.h/2)]
```

The framing for 511 then becomes:

```
{Framing (`79=SqlDefaultConstructor `79 Domain 87  Sce:`124,
  `86=SqlCopy `86 Domain 87 TL From:_ copy from 373,
  `87=SqlField `87 135 X Parent=`86 Field=42,
  `89=SqlValueExpr `89 135 Left:`87 Right:`98 `89(`87+`98),
  `95=SqlCopy `95 Domain 190 SZ From:_ copy from 397,
  `96=SqlField `96 135 W Parent=`95 Field=145,
  `98=SqlValueExpr `98 135 Left:`96 Right:`100 `98(`96/`100),
  `100=2,
  `107=SqlCopy `107 Domain 87 TL From:_ copy from 373,
  `108=SqlField `108 135 Y Parent=`107 Field=64,
  `110=SqlValueExpr `110 135 Left:`108 Right:`119 `110(`108+`119),
  `116=SqlCopy `116 Domain 190 SZ From:_ copy from 397,
  `117=SqlField `117 135 H Parent=`116 Field=167,
  `119=SqlValueExpr `119 135 Left:`117 Right:`121 `119(`117/`121),
  `121=2,
  `124=SqlRow `124 Domain `125 ,
  `125=Domain ROW (`89,`110)[`89,135],[`110,135],
  `126=ReturnStatement `126 -> `79)}
```

and now the method objects have been updated:

```
{{Method 446 Null Definer=-502 LastChange=446 RECT$4 Arity=4  Params(`5,`7,`9,`11) Body:`19
Clause{(x1 int,y1 int, x2 int, y2 int)} .. }
```

```
{Method 498 Domain 87 Definer=-502 LastChange=498 CENTRE$0 Arity=0 87 Params) Body:`126
      Clause{() returns point} .. }
```

6.10 RESTView implementation

In this example, based on tests 22 and 23 of the test suite, the "remote" database will be another database on the same server but will be accessed over TCP/IP using REST. For example, suppose database A contains
```
create table D (e int primary key, f char, g char)
insert into D values (1,'Joe','Soap'), (2,'Betty','Boop')
create role A
grant A to "domain\userId"
```

For this section, the server is started with +s and -H flags. The +s flag ensures that Pyrrho's HTTP service is running, and -H gives us some useful diagnostic information as we will see.
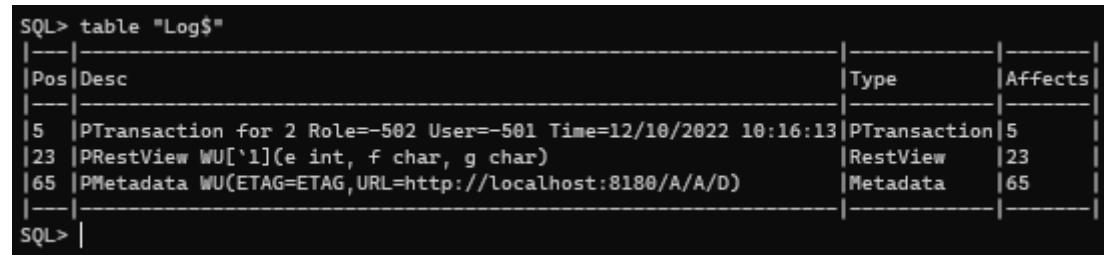
6.10.1 The HTTP1.1 model (test 22)

Suppose that in an empty database t22 we define

```
[create view WU of (e int, f char, g char) as get etag url
'http://localhost:8180/A/A/D']
```

The etag metadata flag specifies the use of the etag mechanism. This follows RFC 7232 to enable a sort of transaction control on the remote database, and briefly noted below. The url metadata flag indicates the use of this URL-based model. On Commit, database B will contain something like

```
SQL> table "Log$"
|---|------------------------------------------------------------------|------------|-------|
|Pos|Desc                                                              |Type        |Affects|
|---|------------------------------------------------------------------|------------|-------|
|5  |PTransaction for 2 Role=-502 User=-501 Time=12/10/2022 10:16:13   |PTransaction|5      |
|23 |PRestView WU['1](e int, f char, g char)                           |RestView    |23     |
|65 |PMetadata WU(ETAG=ETAG,URL=http://localhost:8180/A/A/D)           |Metadata    |65     |
|---|------------------------------------------------------------------|------------|-------|
SQL>
```

As usual, we restart the server to synchronize with the following notes (if we don't, the framing objects below will have different positions). Consider now what the Context will contain at various stages starting with the view definition. Since there are no previous objects in the database, the object and framing for the above committed RestView (in db.objects[23]) will be

```
{RestView 23 Domain `4 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)  Ppos: 23
Result _ UsingTableRowSet: _ ViewTable:`0}
{(`0=VirtualTable `0 Domain `4 Definer=-502 LastChange=`0:`4 RestView=_,
  `1=SqlValue E `1 INTEGER From:`0,
  `2=SqlValue F `2 CHAR From:`0,
  `3=SqlValue G `3 CHAR From:`0,
  `4=Domain (e int, f char, g char)  VIEW (`1,`2,`3)[`1, INTEGER],[`2, CHAR],[`3, CHAR]
structure=`0)}
```

And the role-based information (infos) for the RestView is

```
{(-502=ObInfo WU Privilege=8388607 ETAG URL http://localhost:8180/A/A/D)}
```

(Recall that the role that we are using is the default -502). Set a breakpoint in Start.cs at line 430. Now on a simple query, such as

```
select * from wu
```

following query processing as in previous sections, and RESTView instancing, we get the following objects in the context at the point where traversal begins (Start.cs line 430):

```
{(-505= CONTENT,
  -503= Null,
  #1=SelectRowSet #1:%0 targets: 23=%7 From: #15 Source: %7,
  #8=SqlStar * #8 CONTENT,
  `0=VirtualTable `0 Domain `4 Definer=-502 LastChange=65:`4 RestView=23,
  `1=SqlValue E `1 INTEGER From:`0,
  `2=SqlValue F `2 CHAR From:`0,
  `3=SqlValue G `3 CHAR From:`0,
  `4=Domain (e int, f char, g char)  VIEW (`1,`2,`3)[`1, INTEGER],[`2, CHAR],[`3, CHAR]
structure=`0,
  %0=Domain TABLE (%3,%4,%5) Display=3[%3, INTEGER],[%4, CHAR],[%5, CHAR],
  %1=RestView %1 Domain %6 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)  Ppos: 23
Result _ UsingTableRowSet: _ ViewTable:`0,
  %2=VirtualTable %2 Domain %6 Definer=-502 LastChange=`0:%6 RestView=_,
  %3=SqlValue E %3 INTEGER From:%2,
  %4=SqlValue F %4 CHAR From:%2,
  %5=SqlValue G %5 CHAR From:%2,
  %6=Domain (e int, f char, g char)  VIEW (%3,%4,%5)[%3, INTEGER],[%4, CHAR],[%5, CHAR]
structure=%2,
  %7=RestRowSet %7:%8 targets: 23=%7 From: #15 Target=23 SRow:() http://localhost:8180/A/A/D
RemoteCols:(%3,%4,%5) RemoteNames:(%3=E,%4=F,%5=G),
  %8=Domain TABLE (%3,%4,%5) Display=3[%3, INTEGER],[%4, CHAR],[%5, CHAR],
  %9=SelectStatement %9 Union=#1)}
```

Note that the RestRowSet is for the instance and refers to the RestView 23 that defines the metadata. Note also the differences between the framing objects in light blue and the instanced versions in blue. When traversal begins, we soon find we need to build the RestRowSet (see line 5736 in RowSet.cs). By default, this is carried out using a POST request to the remote server, using the URL from the metadata. The headers are something like

```
UserAgent: Pyrrho 7.03alpha
Authorization: Basic ▓▓▓▓▓▓▓▓▓▓▓▓▓
If-Unmodified-Since: Wed, 11 Jan 2023 06:53:22 GMT
Accept: application/json
```

After placing a debug breakpoint at line 5839, continue execution. In the server window (we have the -H command argument), we see the roundtrip

```
http://localhost:8180/A/A/D
HTTP GET /A/A/D
Received If-Unmodified-Since: Wed, 11 Jan 2023 06:53:22 GMT
Returning ETag: "23,_,160"
```

The returned information (shown in the server window as requested by the -H command line argument) gives an ETag for the returned information. See the Pyrrho manual section 3.8.1: here the numbers specify the base table uid 23 (in database A), the fact that all rows were read, and the latest file position for the returned TableRows. The client is using autocommit mode, and so there is no HEAD round trip.

At line 5836 we collect the returned information in a string sr, and Parse it into a TArray as the aVal field of the RestRowSet {[(%3=1,%4=Joe,%5=Soap),(%3=2,%4=Betty,%5=Boop)]} .

This completes the Build step for the RestRowSet. The server constructs a reply to the client based on this value. The client shows this as follows:

```
SQL> select * from wu
|-|-----|----|
|E|F    |G   |
|-|-----|----|
|1|Joe  |Soap|
|2|Betty|Boop|
|-|-----|----|
SQL>
```

We next consider an example where a RESTView forms part of a join. In order also to show the effect of instancing during compilation, let us make a view for the join. In database t22, define

```
create table HU (e int primary key, k char, m int)
insert into HU values (1,'Cleaner',12500), (2,'Manager',31400)
create view VU as select * from WU natural join HU
```

HU is committed at file position 124: its columns are E 139, K 181 and M 205,

Pos	Desc	Type	Affects
5	PTransaction for 2 Role=-502 User=-501 Time=11/01/2023 06:43:22	PTransaction	5
23	PRestView WU['1](e int, f char, g char)	RestView	23
65	PMetadata WU(ETAG=ETAG,URL=http://localhost:8180/A/A/D)	Metadata	65
117	PTransaction for 5 Role=-502 User=-501 Time=11/01/2023 07:03:13	PTransaction	117
135	PTable HU	PTable	135
142	PColumn3 E for 135(0)[INTEGER]	PColumn3	142
165	PIndex HU on 135(142) PrimaryKey	PIndex	165
184	PColumn3 K for 135(1)[CHAR]	PColumn3	184
208	PColumn3 M for 135(2)[INTEGER]	PColumn3	208
232	PTransaction for 2 Role=-502 User=-501 Time=11/01/2023 07:03:13	PTransaction	232
250	Record 250[135]: 142=1[INTEGER],184=Cleaner[CHAR],208=12500[INTEGER]	Record	250
285	Record 285[135]: 142=2[INTEGER],184=Manager[CHAR],208=31400[INTEGER]	Record	285
320	PTransaction for 1 Role=-502 User=-501 Time=11/01/2023 07:03:14	PTransaction	320
338	PView VU 338 view VU as select * from WU natural join HU	PView	338

We see that VU is committed at file position 339, and if we restart the server we see it has framing

```
{Framing (
  `11=SelectRowSet `11:`13 matching (`18=(`27),`27=(`18)) targets: 23=`22,135=`26 Source: `24,
  `12=SqlStar * `12 CONTENT,
  `13=Domain TABLE (`18,`19,`20,`28,`29) Display=5[`18, INTEGER],[`19, CHAR],[`20, CHAR],[`28,
CHAR],[`29, INTEGER],
  `16=RestView `16 Domain `21 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)  Ppos:
23 Result _ UsingTableRowSet: _ ViewTable:`0,
  `17=VirtualTable `17 Domain `21 Definer=-502 LastChange=`0:`21 RestView=_,
  `18=SqlValue E `18 INTEGER From:`17,
```

```
  `19=SqlValue F `19 CHAR From:`17,
  `20=SqlValue G `20 CHAR From:`17,
  `21=Domain (e int, f char, g char)  VIEW (`18,`19,`20)[`18, INTEGER],[`19, CHAR],[`20, CHAR]
structure=`17,
  `22=RestRowSet `22:`23 targets: 23=`22 From: `24 Target=23 SRow:() http://localhost:8180/A/A/D
RemoteCols:(`18,`19,`20) RemoteNames:(`18=E,`19=F,`20=G),
  `23=Domain TABLE (`18,`19,`20) Display=3[`18, INTEGER],[`19, CHAR],[`20, CHAR],
  `24=JoinRowSet `24:`33 matching (`18=(`27),`27=(`18)) targets: 23=`22,135=`26 INNER First:
`36 Second: `26 on `18=`27,
  `26=TableRowSet `26:`31 Indexes=[(`27)165] key (`27) order (`27) targets: 135=`26 From: `26
Target=135 SRow:(142,184,208) Target:135 HU,
  `27=SqlCopy `27 INTEGER E From:`26 copy from 142,
  `28=SqlCopy `28 CHAR K From:`26 copy from 184,
  `29=SqlCopy `29 INTEGER M From:`26 copy from 208,
  `30=Domain ROW (`27)[`27, INTEGER],
  `31=Domain TABLE (`27,`28,`29) Display=3[`27, INTEGER],[`28, CHAR],[`29, INTEGER],
  `32=Domain ROW (`27)[`27, INTEGER],
  `33=Domain  TABLE  (`18,`19,`20,`28,`29|`27)  Display=5[`18,  INTEGER],[`19,  CHAR],[`20,
CHAR],[`28, CHAR],[`29, INTEGER],[`27, INTEGER],
  `35=Domain ROW (`27)[`27, INTEGER],
  `36=OrderedRowSet `36:`23 key (`18) order (`18) targets: 23=`22 Source: `22,
  `37=SelectStatement `37 Union=`11)}
```

The framing for view VU contains an instance of the RestRowSet WU from above, highlighted. When VU is referenced in a query, it will in turn be instanced and optimised. To check this, we will use the query

select e, f, m from VU where e=1

Following view instancing for VU, and just before traversal starts (Start.cs line 432 again), the Context contains (in addition to copies of VU's framing objects[42]):

```
{(-539= ROW,
 -505= CONTENT,
 -503= Null,
 135=Table 135 Domain `5 Definer=-502 LastChange=208:`5 Indexes:((142)165) KeyCols: (142=True),
 142=TableColumn 142 INTEGER Definer=-502 LastChange=142 -511 Table=135,
 184=TableColumn 184 CHAR Definer=-502 LastChange=184 -508 Table=135,
 208=TableColumn 208 INTEGER Definer=-502 LastChange=208 -511 Table=135,
 #1=SelectRowSet    #1:%0    Indexes=[(%18)165]    where    (#31)    matches    (#8=1)    matching
(#8=(%18),%18=(#8)) targets: 23=%13,135=%17 Source: %2,
 #8=SqlValue E #8 INTEGER From:%8,
 #11=SqlValue F #11 CHAR From:%8,
 #14=SqlCopy #14 INTEGER M From:%17 copy from 208,
 #31=SqlValueExpr #31 BOOLEAN Left:#8 Right:#32 #31(#8=#32),
 #32=1,..
 %0=Domain TABLE (#8,#11,#14) Display=3[#8, INTEGER],[#11, CHAR],[#14, INTEGER],
 %1=View %1 Domain %4 Definer=-502 LastChange=391 ViewDef view VU as select * from WU natural
join HU Ppos: 338 Result %2,
 %2=SelectRowSet  %2:%4  where  (#31)  matches  (#8=1)  matching  (#8=(%18),%18=(#8))  targets:
23=%13,135=%17 Source: %15,
 %3=SqlStar * %3 CONTENT,
 %4=Domain  TABLE  (#8,#11,%11,%19,#14)  Display=5[#8,  INTEGER],[#11,  CHAR],[%11,  CHAR],[%19,
CHAR],[#14, INTEGER],
 %7=RestView %7 Domain %12 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)  Ppos:
23 Result _ UsingTableRowSet: _ ViewTable:`0,
 %8=VirtualTable %8 Domain %12 Definer=-502 LastChange=`0:%12 RestView=_,
 %11=SqlValue G %11 CHAR From:%8,
 %12=Domain (e int, f char, g char)  VIEW (#8,#11,%11)[#8, INTEGER],[#11, CHAR],[%11, CHAR]
structure=%8,
 %13=RestRowSet  %13:%14  matches  (#8=1)  targets:  23=%13  From:  %15  Target=23  SRow:()
http://localhost:8180/A/A/D RemoteCols:(#8,#11,%11) RemoteNames:(#8=E,#11=F,%11=G),
 %14=Domain TABLE (#8,#11,%11) Display=3[#8, INTEGER],[#11, CHAR],[%11, CHAR],
 %15=JoinRowSet %15:%24 where (#31) matches (#8=1,%18=1) matching (#8=(%18),%18=(#8)) targets:
23=%13,135=%17 INNER First: %27 Second: %17 on #8=%18,
 %17=TableRowSet %17:%22 Indexes=[(%18)165] key (%18) order (%18) targets: 135=%17 From: %17
Target=135 SRow:(142,184,208) Target:135 HU,
 %18=SqlCopy %18 INTEGER E From:%17 copy from 142,
 %19=SqlCopy %19 CHAR K From:%17 copy from 184,
 %21=Domain ROW (%18)[%18, INTEGER],
 %22=Domain TABLE (%18,%19,#14) Display=3[%18, INTEGER],[%19, CHAR],[#14, INTEGER],
```

---

[42] The illustration omits the framing objects `10,.. themselves..

```
 %23=Domain ROW (%18)[%18, INTEGER],
 %24=Domain   TABLE   (#8,#11,%11,%19,#14|%18)   Display=5[#8,   INTEGER],[#11,   CHAR],[%11,
CHAR],[%19, CHAR],[#14, INTEGER],[%18, INTEGER],
 %25=Domain ROW (%9)[%9, INTEGER],
 %26=Domain ROW (%18)[%18, INTEGER],
 %27=OrderedRowSet %27:%14 key (#8) order (#8) targets: 23=%13 Source: %13,
 %28=SelectStatement %28 Union=%2,
 %29=SelectStatement %30 Union=#1)}
```

We see that VU's framing objects have been instanced at objects %1 to %28, and then substitutions for e, f, and m have modified all of the objects coloured red (the unmodified framing objects are shown in blue). During rowset review, the where-condition E=1 propagated through the objects together with short-cut matches conditions (highlighted in yellow).

And the -H diagnostic trace on the server shows that the condition E=1 has been passed successfully to the remote server so that only one row is accessed::

```
http://localhost:8180/A/A/D/E=1
HTTP GET /A/A/D/E=1
Received If-Unmodified-Since: Wed, 11 Jan 2023 07:06:20 GMT
Returning ETag: "23,131,131"
--> 1 rows
```

This time, the ETag is for the single tablerow that was returned.

```
SQL> select e, f, m from VU where e=1
|-|---|-----|
|E|F  |M    |
|-|---|-----|
|1|Joe|12500|
|-|---|-----|
SQL>
```

The final parts of test 22 perform an insert on RestView WU and an update on View VU, and these steps occur on the remote server. The first command is

`insert into wu values(3,'Fred','Bloggs')`

At the start of SqlInsert.Obey the context (Activation 77 if you have done all the above steps) has

```
{(-505=CONTENT,
 -503=Null,
 #1=SqlInsert #1 Target: %9 Value: %11,
 #16=#22,
 #22=SqlRow #22 Domain %12 ,
 #23=3,
 #25=Fred,
 #32=Bloggs,
 `0=VirtualTable `0 Domain `7 Definer=-502 LastChange=65:`7 RestView=23,
 `2=SqlValue E `2 INTEGER From:`0,
 `4=SqlValue F `4 CHAR From:`0,
 `6=SqlValue G `6 CHAR From:`0,
 `7=Domain (e int, f char, g char)  VIEW (`2,`4,`6)[`2,INTEGER],[`4,CHAR],[`6,CHAR]
     structure=`0,
 %0=RestView %0 Domain %8 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)  Ppos: 23
     Result _ UsingTableRowSet: _ ViewTable:`0,
 %1=VirtualTable %1 Domain %8 Definer=-502 LastChange=`0:%8 RestView=23,
 %3=SqlValue E %3 INTEGER From:%1,
 %5=SqlValue F %5 CHAR From:%1,
 %7=SqlValue G %7 CHAR From:%1,
 %8=Domain (e int, f char, g char)  VIEW (%3,%5,%7)[%3,INTEGER],[%5,CHAR],[%7,CHAR]
     structure=%1,
 %9=RestRowSet %9:%10 targets: 23=%9 From: #13 Target=23 SRow:() http://localhost:8180/A/A/D
     RemoteCols:(%3,%5,%7) RemoteNames:(%3=E,%5=F,%7=G),
 %10=Domain TABLE (%3,%5,%7) Display=3[%3,INTEGER],[%5,CHAR],[%7,CHAR],
 %11=SqlRowSet %11:%13 targets: 23=%9 SqlRows [#22],
 %12=Domain ROW (#23,#25,#32)[#23,INTEGER],[#25,CHAR],[#32,CHAR],
 %13=Domain TABLE (%3,%5,%7) Display=3[%3,INTEGER],[%5,CHAR],[%7,CHAR])}
```

The Insert method for the RestRowSet uses an HTTPActivation, which constructs a POST request for a round trip to the remote server. Allow this to continue. With the -H flag the server diagnostics show the successful HTTP round trip

```
RoundTrip POST http://localhost:8180/A/A/D [{"E":3,"F":'Fred',"G":'Bloggs'}]
HTTP POST /A/A/D
[{"E":3,"F":'Fred',"G":'Bloggs'}]
Received If-Unmodified-Since: Fri, 07 Oct 2022 11:10:29 GMT
Returning ETag: "23,294,294"
Recording ETag "23,294,294"
--> OK
```

The client at this point correctly shows that no changes have been made to database B

```
SQL> insert into wu values(3,'Fred','Bloggs')
0 records affected in t22
SQL> |
```

But we can verify that table D on database A now has all three rows[43]:

```
http://localhost:8180/A/A/D
HTTP GET /A/A/D
Received If-Unmodified-Since: Fri, 07 Oct 2022 11:13:55 GMT
Returning ETag: "23,_,294"
--> 3 rows
```

```
SQL> select * from wu
|-|-----|------|
|E|F    |G     |
|-|-----|------|
|1|Joe  |Soap  |
|2|Betty|Boop  |
|3|Fred |Bloggs|
|-|-----|------|
SQL>
```

The second remote update is *to the join*

update vu set f='Elizabeth' where e=2

At the start of UpdateSearch.Obey the context has (again omitting the copies of framing objects)
```
{(-538=TABLE,
{(-538=TABLE,
  -505=CONTENT,
  -503=Null,
  #1=UpdateSearch #1 Target: %1,
  #15=SqlValue F #15 CHAR From:%7,
  #17=Elizabeth,
  #35=SqlValue E #35 INTEGER From:%7,
  #36=SqlValueExpr #36 BOOLEAN Left:#35 Right:#37 #36(#35=#37),
  #37=2, ..,
  %0=View %0 Domain %3 Definer=-502 LastChange=392
      ViewDef view VU as select * from WU natural join HU Ppos: 339 Result %1,
  %1=SelectRowSet %1:%3 where (#36,%26) matches (#35=2) matching (#35=(%20),%20=(#35))
      targets: 23=%15,136=%19 Source: %17 Assigs:(UpdateAssignment Vbl: #15 Val: #17=True),
  %2=SqlStar * %2 CONTENT,
  %3=Domain TABLE (#35,#15,%13,%21,%22) Display=5[#35,INTEGER],[#15,CHAR],[%13,CHAR],
      [%21,CHAR],[%22,INTEGER],
  %6=RestView %6 Domain %14 Definer=-502 LastChange=23 ViewDef (e int, f char, g char)
      Ppos: 23 Result _ UsingTableRowSet: _ ViewTable:`0,
  %7=VirtualTable %7 Domain %14 Definer=-502 LastChange=`0:%14 RestView=23,
  %13=SqlValue G %13 CHAR From:%7,
  %14=Domain (e int, f char, g char)  VIEW (#35,#15,%13)[#35,INTEGER],[#15,CHAR],[%13,CHAR]
      structure=%7,
  %15=RestRowSet %15:%16 matches (#35=2) targets: 23=%15 From: %17
      Assigs:(UpdateAssignment Vbl: #15 Val: #17=True) Target=23 SRow:()
      RemoteCols:(#35,#15,%13) RemoteNames:(#15=F,#35=E,%13=G),
  %16=Domain TABLE (#35,#15,%13) Display=3[#35,INTEGER],[#15,CHAR],[%13,CHAR],
  %17=JoinRowSet %17:%24 where (#36) matches (#35=2,%20=2) matching (#35=(%20),%20=(#35))
      targets: 23=%15,136=%19 Assigs:(UpdateAssignment Vbl: #15 Val: #17=True)
```

---

[43] Even if the above step in database B was in an explicit transaction which is rolled back, the remote insert is committed immediately with this URL-based implementation. For better transactional behaviour, the scripted REST model described in the next section should be used.

```
         INNER First: %15 Second: %19 on #35=%20,
  %19=TableRowSet %19:%23 Indexes=[(%20)166] key (%20) order (%20) targets: 136=%19 From: %19
         Assigs:(UpdateAssignment Vbl: #15 Val: #17=True)
         Target=136 SRow:(143,185,209) Target:136 HU,
  %20=SqlCopy %20 INTEGER E From:%19 copy from 143,
  %21=SqlCopy %21 CHAR K From:%19 copy from 185,
  %22=SqlCopy %22 INTEGER M From:%19 copy from 209,
  %23=Domain TABLE (%20,%21,%22) Display=3[%20,INTEGER],[%21,CHAR],[%22,INTEGER],
  %24=Domain TABLE (#35,#15,#13,%21,%22|%20) Display=5[#35,INTEGER],[#15,CHAR],[#13,CHAR],
         [%21,CHAR],[%22,INTEGER],[%20,INTEGER],
  %25=SqlValueExpr %25 BOOLEAN From:%17 Left:#35 Right:%20 %25(#35=%20),
  %26=SqlValueExpr %26 BOOLEAN Left:#35 Right:%20 %26(#35=%20),
  %27=OrderedRowSet %27:%16 key (#35) order (#35) targets: 23=%15 Source: %15
         Assigs:(UpdateAssignment Vbl: #15 Val: #17=True),
  %28=SelectStatement %28 Union=%1)}
```

We see that again the OrderedRowSet has been short-circuited, the updateSearch statement's source is for %1, which has targets %14 and %18, and the update assignments have been added to the appropriate target rowSet %14. Letting execution continue, we get

```
SQL> update vu set f='Elizabeth' where e=2
0 records affected in t22
SQL>
```

A GET was used to discover which records satisfy the where condition (just one), and then a PUT round trip performs the update[44].

```
http://localhost:8180/A/A/D/E=2
HTTP GET /A/A/D/E=2
Received If-Unmodified-Since: Fri, 07 Oct 2022 11:15:57 GMT
Returning ETag: "23,160,160"
--> 1 rows
RoundTrip PUT http://localhost:8180/A/A/D/E=2 [{"E":2,"F":'Elizabeth',"G":'Boop'}]
HTTP PUT /A/A/D/E=2
[{"E":2,"F":'Elizabeth',"G":'Boop'}]
Received If-Unmodified-Since: Fri, 07 Oct 2022 11:15:57 GMT
Returning ETag: "23,294,345"
Recording ETag "23,294,345"
--> OK
```

In Database A we check this:

```
E:\PyrrhoDB70\Pyrrho>pyrrhocmd A
SQL> table D
|-|---------|----|
|E|F        |G   |
|-|---------|----|
|1|Joe      |Soap|
|2|Betty    |Boop|
|3|Elizabeth|Boop|
|-|---------|----|
SQL> |
```

The important point is that the process of rowSet review has decomposed the update of the join to an update of the appropriate (remote) table.

6.10.2 The scripted POST model (test 23)

Reset database A to the starting state above, start again with an empty database t23, and define
```
[create view W of (e int, f char, g char) as get etag
'http://localhost:8180/A/A/D']
create table H (e int primary key, k char, m int)
insert into H values (1,'Cleaner',12500), (2,'Manager',31400)
create view V as select * from w natural join H
```

Apart from the URL metadata flag, everything is as in the last section. The behaviour of the select statements is similar. Let us look at the insert statement:

---

[44] See the previous footnote.

```
begin transaction
insert into w values(3,'Fred','Bloggs')
```

 When this runs, we get

```
SQL> begin transaction
SQL-T>insert into w values(3,'Fred','Bloggs')
0 records in transaction B
SQL-T>_
```

```
RoundTrip HEAD http://localhost:8180/A/A/D
HTTP HEAD /A/A/D
Received If-Unmodified-Since: Sat, 15 Oct 2022 21:26:27 GMT
Returning ETag:
--> OK
```

We see that no change has been made in the remote server (the transaction has not been committed). However, the RESTActivation that is controlling the insert operation has generated a Post record in the transaction. To see what happens, place a break point at the start of Transaction.Commit, and give the command

commit

Then we can see in the debugger that physicals contains

`{(!0=Post D http://localhost:8180/A/A/D insert into D values(3,'Fred','Bloggs'); 23)}`

When Commit is called on the Post record, it executes the round-trip to perform the insert operation, and we get

```
SQL-T>commit
SQL>
```

```
RoundTrip POST http://localhost:8180/A/A insert into D values(3,'Fred','Bloggs');

HTTP POST /A/A
insert into D values(3,'Fred','Bloggs');

Received If-Match: 385
Received If-Unmodified-Since: Sat, 15 Oct 2022 20:26:27 GMT
Returning ETag: "23,294,294"
Recording ETag "23,294,294"
--> OK
```

Next, we look at the update operation on the join/view combination, again with an explicit transaction so that we can see where commit occurs.

```
begin transaction
update v set f='Elizabeth' where e=2
```

```
SQL-T>update v set f='Elizabeth' where e=2
0 records in transaction t23
SQL-T>|
```

```
HTTP HEAD /A/A/D/E=2
Received If-Unmodified-Since: Sat, 15 Oct 2022 21:43:53 GMT
Returning ETag:
--> OK
http://localhost:8180/A/A select E,F,G from D where E=2
HTTP POST /A/A
select E,F,G from D where E=2
Received If-Unmodified-Since: Sat, 15 Oct 2022 21:43:53 GMT
Returning ETag: "23,160,160"
--> 1 rows
Response ETag: 23,160,160
```

Here we can see that the server has discovered that only one row will need to be changed, and has obtained an ETag for it. No changes have been made yet, however. On Commit once again there is a Post record

`{(!0=Post D http://localhost:8180/A/A/D update D set F='Elizabeth' where E=2 23)}`

and when this record is committed we get

```
SQL-T>commit
SQL>
```
```
HTTP HEAD /A/A
Received If-Match: 23,160,160
Returning ETag:
RoundTrip POST http://localhost:8180/A/A update D set F='Elizabeth' where E=2

HTTP POST /A/A
update D set F='Elizabeth' where E=2

Received If-Match: 385
```

Test23 includes numerous exercises to verify transactional behaviour for this model.

6.10.3 RestView with a Using table

This example is from Test 24 and demonstrates selection, insertion and update for a RestView specified with a using table. As described earlier, this allows a view to select similar rows from a list of source databases, which are accessed using HTTP/1.1 and/or a REST service. Since Pyrrho provides services of this kind, some of the databases may be accessed using Pyrrho itself. The Pyrrho distribution includes a simple web server that can be tailored to provide a suitable HTTP/1.1 service that accesses other DBMS, and this has been demonstrated for MySQL and SQL Server.

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

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

For example, a filter on remote columns can be applied on the remote database: and an aggregation of remote data on a single remote source can be carried out on the remote DBMS. But it soon becomes apparent that in more complex cases some transformation of the original query is required. For example, a COUNT of a remote datum supplied separately by several DBMS will need to be implemented as a SUM of the contributions from these, and D will have just one row per contributor.

In this section we consider a number of examples involving grouping and aggregation. View syntax allows quite general CursorSpecifications, but the RestView syntax is just a simple Domain and URL, so that all consideration of grouping and aggregation is done at the SelectRowSet level. This makes it feasible to consider which groupings and aggregations can be passed down to remote contributors.

The first part of test 24 simply sets up two local databases as sources for the test, DB and DC, accessible by a remote connection. DB:
```
create table T(E int,F char)
insert into T values(3,'Three'),(6,'Six'),(4,'Vier'),(6,'Sechs')
create role DB
grant DB to "domain\user"
```

And DC:
```
create table U(E int,F char)
insert into U values(5,'Five'),(4,'Four'),(8,'Ate')
create role DC
grant DC to "domain\user"
```

Then in another database (such as t24) we can define a table that lists these sources using Pyrrho's HTTP service. Test24 first defines a RESTView that uses just one of these remote databases

```
create view VV of (E int,F char) as get 'http://localhost:8180/DB/DB/t'
```

and this step is included here for completeness.

```
create table VU (d char primary key, k int, u char)
insert into VU values('B',4,'http://localhost:8180/DB/DB/t')
insert into VU values('C',1,'http://localhost:8180/DC/DC/u')
```

We define a RESTView that collects data from these sources:

```
create view WW of (E int, D char, K int, F char) as get using VU
```

This is committed as a RestView object. Test 24 also defines another table to test working with local joins, see below:

```
create table M (e int primary key, n char, unique(n))
insert into M values (2,'Deux'),(3,'Trois'),(4,'Quatre')
insert into M values (5,'Cinq'),(6,'Six'),(7,'Sept')
```

At this stage the Log for t24 shows a PRestView2 object at location 435 (VV and M are not shown in this illustration):

Pos	Desc
125	PTable VU
132	PColumn3 D for 125(0)[CHAR]
154	PIndex VU on 125(132) PrimaryKey
172	PColumn3 K for 125(1)[INTEGER]
195	PColumn3 U for 125(2)[CHAR]
236	Record 236[125]: 132=B,172=4,195=http://localhost:8180/DB/DB/t
310	Record 310[125]: 132=C,172=1,195=http://localhost:8180/DC/DC/u
384	PRestView2 WW(`9) using 125
..	..

The PRestView2 record is loaded as a RestView object with a UsingRowSet, With the debugger we can see the in-memory version, which following a server restart is

```
{RestView 384 Domain `16 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
Ppos: 384 Result _ UsingTableRowSet: `17 ViewTable:`7}
```

with a framing for its compiled objects:

```
{Framing (
  `7=VirtualTable `7 Domain `16 Definer=-502 LastChange=`7:`16 RestView=384,
  `9=SqlValue E `9 135⁴⁵ From:`7,
  `11=SqlCopy `11 CHAR D From:`17 copy from 132,
  `13=SqlCopy `13 135 K From:`17 copy from 172,
  `15=SqlValue F `15 CHAR From:`7,
  `16=Domain (E int, D char, K int, F char)  VIEW (`9,`11,`13,`15)[`9,135],[`11,CHAR],
       [`13,135],[`15,CHAR] structure=`7,
  `17=TableRowSet `17:`22 Indexes=[(`18)154] key (`18) targets: 125=`17 From: `17 Target=125
       SRow:(132,172,195) Target:125 VU,
  `18=SqlCopy `18 CHAR D From:`17 copy from 132,
  `19=SqlCopy `19 135 K From:`17 copy from 172,
  `20=SqlCopy `20 CHAR U From:`17 copy from 195,
  `21=Domain TABLE (`18,`19,`20) Display=3[`18,CHAR],[`19,135],[`20,CHAR],
  `22=Domain TABLE (`11,`13|`20) Display=2[`11,CHAR],[`13,135],[`20,CHAR])}
```

We note that the columns of the RestView WW include two local columns D and K coming from the UsingTableRowset `17. Objects `18-`21 are not referenced. The columns E and F are (only) in the remote view 384, and will be identified as remote columns in the rowsets that will be constructed.

First consider the case where there is a filter on the usingTable:

```
 select * from ww where k=1
```

Just before traversal starts we have (we omit the copy of the shared framing objects from above: their instances are objects %2 to %16 below, modified⁴⁶ as highlighted):

```
{(-538=TABLE,
  -505=CONTENT,
  -503=Null,
  #1=SelectRowSet #1:%0 where (#25) matches (#24=1) targets: 384=%20 From: #15 Source: %20,
```

---

[45] See footnote on p.86

[46] Instances are not shared, so that modified versions can have same defining position.

```
  #8=SqlStar * #8 CONTENT,
  #24=SqlCopy #24 INTEGER K From:%12 copy from 172,
  #25=SqlValueExpr #25 BOOLEAN Left:#24 Right:#26 #25(#24=#26),
  #26=1, ..,
  %0=Domain TABLE (%4,%6,#24,%10) Display=4[%4,INTEGER],[%6,CHAR],[#24,INTEGER],[%10,CHAR],
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
        Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %4=SqlValue E %4 INTEGER From:%2,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %10=SqlValue F %10 CHAR From:%2,
  %11=Domain (E int, D char, K int, F char)  VIEW (%4,%6,#24,%10)[%4,INTEGER],[%6,CHAR],
        [#24,INTEGER],[%10,CHAR] structure=%2,
  %12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) where (#25) matches (#24=1)
        targets: 125=%12 From: %12 Target=125 SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,#24|%15) Display=2[%6,CHAR],[#24,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%19 targets: 384=%18 Target=384 SRow:()  RemoteCols:(%4,%10)
RemoteNames:(%4=E,%10=F) UsingTableRowSet %12,
  %19=Domain TABLE (%4,%6,#24,%10) Display=4[%4,INTEGER],[%6,CHAR],[#24,INTEGER],[%10,CHAR],
  %20=RestRowSetUsing %20:%19 where (#25) matches (#24=1) targets: 384=%20 From: #15
Target=384 SRow:(132,172,195) ViewDomain: %19 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
  %21=SelectStatement %21 Union=#1)}
```

The REST operation is managed by the RestRowSet %18.

When we allow the server to continue, we see that no request is made to the first contributor. But note that the columns D and K are included in the remote request in case they are needed to evaluate a select expression:

```
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u
Returning ETag: "23,_,134"
--> 3 rows
Response ETag: 23,_,134
```

```
SQL> select * from ww where k=1
|-|-|-|----|
|E|D|K|F   |
|-|-|-|----|
|5|C|1|Five|
|4|C|1|Four|
|8|C|1|Ate |
|-|-|-|----|
SQL>
```

Now let's have a filter on the remote table:

select * from ww where e=3

```
{(..,
  #1=SelectRowSet #1:%0 where (#25) matches (#24=3) targets: 384=%20 From: #15 Source: %20,
  #8=SqlStar * #8 CONTENT,
  #24=SqlValue E #24 INTEGER From:%2,
  #25=SqlValueExpr #25 BOOLEAN Left:#24 Right:#26 #25(#24=#26),
  #26=3,..,
  %0=Domain TABLE (#24,%6,%8,%10) Display=4[#24,INTEGER],[%6,CHAR],[%8,INTEGER],[%10,CHAR],
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
        Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %8=SqlCopy %8 INTEGER K From:%12 copy from 172,
  %10=SqlValue F %10 CHAR From:%2,
  %11=Domain (E int, D char, K int, F char)  VIEW (#24,%6,%8,%10)[#24,INTEGER],[%6,CHAR],
        [%8,INTEGER],[%10,CHAR] structure=%2,
  %12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: %12 Target=125
        SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,%8|%15) Display=2[%6,CHAR],[%8,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%19 where (#25) matches (#24=3) targets: 384=%18 Target=384 SRow:()
```

```
        RemoteCols:(#24,%10) RemoteNames:(#24=E,%10=F) UsingTableRowSet %12,
  %19=Domain TABLE (#24,%6,%8,%10) Display=4[#24,INTEGER],[%6,CHAR],[%8,INTEGER],[%10,CHAR],
  %20=RestRowSetUsing %20:%19 where (#25) matches (#24=3) targets: 384=%20 From: #15
        Target=384 SRow:(132,172,195) ViewDomain: %19 Template: %18 UsingTableRowSet:%12
        UrlCol:%15,
  %21=SelectStatement %21 Union=#1)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where (E=3) and E=3
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where (E=3) and E=3
Returning ETag: "23,92,92"
--> 1 rows
Response ETag: 23,92,92
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where (E=3) and E=3
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where (E=3) and E=3
Returning ETag:
--> 0 rows
```

```
SQL> Select * from ww where e=3
|-|-|-|-----|
|E|D|K|F    |
|-|-|-|-----|
|3|B|4|Three|
|-|-|-|-----|
SQL>
```

Here we see that the test E=3 is now in the RestRowSet template %18, and is passed to each contributor, and this has limited the amount of data transferred from the remote system.

As mentioned above, the contributing rowsets provide registers for aggregations:

select count(e) from ww
```
{(..,
  #1=SelectRowSet #1:%21 targets: 384=%20 From: #22 Source: %20,
  #8=SqlFunction #8 INTEGER COUNT From:#1 COUNT(#14),
  #14=SqlValue E #14 INTEGER From:%2, ...,
  %0=Domain TABLE (#8) Display=1[#8,INTEGER] Aggs (#8),
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
        Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %8=SqlCopy %8 INTEGER K From:%12 copy from 172,
  %10=SqlValue F %10 CHAR From:%2,
  %11=Domain (E int, D char, K int, F char)  VIEW (#14,%6,%8,%10)[#14,INTEGER],[%6,CHAR],
        [%8,INTEGER],[%10,CHAR] structure=%2,
  %12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: %12 Target=125
        SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,%8|%15) Display=2[%6,CHAR],[%8,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%22 targets: 384=%18 Target=384 SRow:()  RemoteCols:(#14,%10)
        RemoteNames:(#14=E,%10=F) UsingTableRowSet %12,
  %19=Domain TABLE (#14,%6,%8,%10) Display=4[#14,INTEGER],[%6,CHAR],[%8,INTEGER],[%10,CHAR],
  %20=RestRowSetUsing %20:%23 targets: 384=%20 From: #22 Target=384 SRow:(132,172,195)
        ViewDomain: %23 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
  %21=Domain TABLE (#8|#14,%6,%8,%10) Display=1[#8,INTEGER],[#14,INTEGER],[%6,CHAR],
        [%8,INTEGER],[%10,CHAR] Aggs (#8),
  %22=Domain TABLE (#8|#14,%6,%8,%10) Display=1[#8,INTEGER],[#14,INTEGER],[%6,CHAR],
        [%8,INTEGER],[%10,CHAR] Aggs (#8),
  %23=Domain TABLE (#8,#14,%6,%8,%10)[#8,INTEGER],[#14,INTEGER],[%6,CHAR],
        [%8,INTEGER],[%10,CHAR],
  %24=SelectStatement %24 Union=#1)}
```

When traversal starts of the SelectRowSet #1, the SelectRowSet.Build() takes change. Place a breakpoint in RestRowSet.Build() at line 6528 in RowSet.cs.

Then we traverse the UsingTableRowSet %12, and call RestRowSetusing.Build(), we see a JSON document returned, one from each of DB and DC:

```
"[{\"COUNT\": 4, \"$#8\": {\"4\": 4}}]"
```

```
"[{\"COUNT\": 3, \"$#8\": {\"3\": 3}}]"
```

```
http://localhost:8180/DB/DB select COUNT(E),E from t
HTTP POST /DB/DB
select COUNT(E),E from t
Returning ETag: "23,_,155"
--> 1 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select COUNT(E),E from u
HTTP POST /DC/DC
select COUNT(E),E from u
Returning ETag: "23,_,134"
--> 1 rows
Response ETag: 23,_,134
```

```
SQL> select count(e) from ww
|-----|
|COUNT|
|-----|
|7    |
|-----|
```

We see that the request that was sent to the contributors in RestRowSet.Build(). was select count(E), so that 1 row is returned from each contributor. The REST process supplies the Registers $#8 for the aggregations in addition to the results for contributors, and RestRowSetBuild accumulates them in the global result[47] without the need to rewrite the query to form the sum of the counts (in the past we would have rewritten this query changing COUNT to SUM). Importantly, this trick works not only for other aggregation functions but for any expression containing aggregations.

Once again the network traffic has been minimised.

The count(*) case is similarly handled (note there is a difference between count(E) and count(*)). (Currently count(*) without a where condition is treated with a special shortcut.)

select max(f) from ww where e>4

```
{(.. ,
  #1=SelectRowSet #1:%21 where (#30) targets: 384=%20 From: #20 Source: %20,
  #8=SqlFunction #8 CHAR MAX From:#1 MAX(#12),
  #12=SqlValue F #12 CHAR From:%2,
  #29=SqlValue E #29 INTEGER From:%2,
  #30=SqlValueExpr #30 BOOLEAN Left:#29 Right:#31 #30(#29>#31),
  #31=4,
  %0=Domain TABLE (#8) Display=1[#8,CHAR] Aggs (#8),
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
       Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %8=SqlCopy %8 INTEGER K From:%12 copy from 172,
  %11=Domain (E int, D char, K int, F char)  VIEW (#29,%6,%8,#12)[#29,INTEGER],[%6,CHAR],
       [%8,INTEGER],[#12,CHAR] structure=%2,
  %12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: %12 Target=125
       SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,%8|%15) Display=2[%6,CHAR],[%8,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%22 where (#30) targets: 384=%18 Target=384 SRow:()  RemoteCols:(#29,#12)
       RemoteNames:(#12=F,#29=E) UsingTableRowSet %12,
  %19=Domain TABLE (#29,%6,%8,#12) Display=4[#29,INTEGER],[%6,CHAR],[%8,INTEGER],[#12,CHAR],
  %20=RestRowSetUsing %20:%23 where (#30) targets: 384=%20 From: #20 Target=384
       SRow:(132,172,195) ViewDomain: %23 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
  %21=Domain TABLE (#8|#29,%6,%8,#12) Display=1[#8,CHAR],[#29,INTEGER],[%6,CHAR],
       [%8,INTEGER],[#12,CHAR] Aggs (#8),
  %22=Domain TABLE (#8|#29,%6,%8,#12) Display=1[#8,CHAR],[#29,INTEGER],[%6,CHAR],
       [%8,INTEGER],[#12,CHAR] Aggs (#8),
  %23=Domain TABLE (#8,#12,#29,%6,%8)[#8,CHAR],[#12,CHAR],[#29,INTEGER]
       [%6,CHAR],[%8,INTEGER],
  %24=SelectStatement %24 Union=#1)}
```

---

[47] The code to do this is in Domain.Coerce (currently at line 3000 of Domain.cs). For full details, see the Pyrrho manual, section 8.8.9.

```
http://localhost:8180/DB/DB select MAX(F),F from t where (E>4)
HTTP POST /DB/DB
select MAX(F),F from t where (E>4)
Returning ETag: "23,155,155"
--> 1 rows
Response ETag: 23,155,155
http://localhost:8180/DC/DC select MAX(F),F from u where (E>4)
HTTP POST /DC/DC
select MAX(F),F from u where (E>4)
Returning ETag: "23,134,134"
--> 1 rows
Response ETag: 23,134,134
SQL> select max(f) from ww having e>4
|---|
|MAX|
|---|
|Six|
|---|
```

We see that the where condition has been passed to the RestRowSet, so that each remote rowset returns its maximum, and the registers so that the RestRowSetUsing can form the overall maximum without rewriting the query.

Grouping:

```
select sum(e)+char_length(f),f  from ww group by f
{(..,
  #1=SelectRowSet #1:%24 groupSpec: #47 groupings (#50) GroupCols(#27) targets: 384=%22
      From: #38 Source: %22,
  #8=SqlFunction #8 INTEGER SUM From:#1 SUM(#12),
  #12=SqlValue E #12 INTEGER From:%3,
  #14=SqlValueExpr #14 Domain %19 From:#1 Left:#8 Right:#15 #14(#8+#15),
  #15=SqlFunction #15 INTEGER CHAR_LENGTH From:#1 CHAR_LENGTH(#27),
  #27=SqlValue F #27 CHAR From:%3,
  #47=GroupSpecification #47(#50),
  #50=Grouping #50 Domain %26 GROUP (#27=0),..,
  %0=Domain UNION Aggs (#8),
  %1=Domain TABLE (#14,#27) Display=2[#14,Domain INTEGER Aggs (#8)],[#27,CHAR] Aggs (#8),
  %2=RestView %2 Domain %12 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
      Ppos: 384 Result _ UsingTableRowSet: %13 ViewTable:`7,
  %3=VirtualTable %3 Domain %12 Definer=-502 LastChange=`7:%12 RestView=384,
  %7=SqlCopy %7 CHAR D From:%13 copy from 132,
  %9=SqlCopy %9 INTEGER K From:%13 copy from 172,
  %12=Domain (E int, D char, K int, F char)  VIEW (#12,%7,%9,#27)[#12,INTEGER],[%7,CHAR],
      [%9,INTEGER],[#27,CHAR] structure=%3,
  %13=TableRowSet %13:%18 Indexes=[(%14)154] key (%14) targets: 125=%13 From: %13 Target=125
      SRow:(132,172,195) Target:125 VU,
  %14=SqlCopy %14 CHAR D From:%13 copy from 132,
  %15=SqlCopy %15 INTEGER K From:%13 copy from 172,
  %16=SqlCopy %16 CHAR U From:%13 copy from 195,
  %17=Domain TABLE (%14,%15,%16) Display=3[%14,CHAR],[%15,INTEGER],[%16,CHAR],
  %18=Domain TABLE (%7,%9|%16) Display=2[%7,CHAR],[%9,INTEGER],[%16,CHAR],
  %19=Domain INTEGER Aggs (#8),
  %20=RestRowSet %20:%27 groupSpec: #47 groupings (#50) GroupCols(#27) targets: 384=%20
      Target=384 SRow:()  RemoteCols:(#12,#27) RemoteNames:(#12=E,#27=F)
      UsingTableRowSet %13,
  %21=Domain TABLE (#12,%7,%9,#27) Display=4[#12,INTEGER],[%7,CHAR],[%9,INTEGER],[#27,CHAR],
  %22=RestRowSetUsing %22:%28 targets: 384=%22 From: #38 Target=384 SRow:(132,172,195)
      ViewDomain: %28 Template: %20 UsingTableRowSet:%13 UrlCol:%16,
  %23=Domain ROW (#27)[#27,CHAR],
  %24=Domain TABLE (#14,#27|#12,%7,%9) Display=2[#14,Domain INTEGER Aggs (#8)],
      [#27,CHAR],[#12,INTEGER],[%7,CHAR],[%9,INTEGER] Aggs (#8),
  %25=Domain ROW (#27)[#27,CHAR],
  %26=Domain ROW (#27)[#27,CHAR],
  %27=Domain TABLE (#14,#27|#12,%7,%9) Display=2[#14,Domain INTEGER Aggs (#8)],
      [#27,CHAR],[#12,INTEGER],[%7,CHAR],[%9,INTEGER] Aggs (#8),
  %28=Domain TABLE (#12,#14,#27,%7,%9,#8)[#12,INTEGER],
      [#14,Domain INTEGER Aggs (#8)],[#27,CHAR],[%7,CHAR],[%9,INTEGER],[#8,INTEGER],
  %29=SelectStatement %29 Union=#1)}
```

The JSON documents returned are

"[{\"Col0\": Null, \"F\": 'Sechs', \"$#9\": {\"0\": 6}},{\"Col0\": Null, \"F\": 'Six',
\"$#9\": {\"0\": 6}},{\"Col0\": Null, \"F\": 'Three', \"$#9\": {\"0\": 3}},{\"Col0\": Null,
\"F\": 'Vier', \"$#9\": {\"0\": 4}}]"
"[{\"Col0\": Null, \"F\": 'Ate', \"$#9\": {\"0\": 8}},{\"Col0\": Null, \"F\": 'Five',
\"$#9\": {\"0\": 5}},{\"Col0\": Null, \"F\": 'Four', \"$#9\": {\"0\": 4}}]"

All of the F's are different so combining the results is almost trivial (as an exercise try another example):

```
http://localhost:8180/DB/DB select (SUM(E)+CHAR_LENGTH(F)),F,E from t group by F
HTTP POST /DB/DB
select (SUM(E)+CHAR_LENGTH(F)),F,E from t group by F
Returning ETag: "23,_,155"
--> 4 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select (SUM(E)+CHAR_LENGTH(F)),F,E from u group by F
HTTP POST /DC/DC
select (SUM(E)+CHAR_LENGTH(F)),F,E from u group by F
Returning ETag: "23,_,134"
--> 3 rows
Response ETag: 23,_,134
SQL> select sum(e)+char_length(f),f  from ww group by f
|----|-----|
|Col0|F    |
|----|-----|
|11  |Ate  |
|9   |Five |
|8   |Four |
|11  |Sechs|
|9   |Six  |
|8   |Three|
|8   |Vier |
|----|-----|
SQL>
```

select count(*),k/2 as k2 from ww group by k2

```
{(..,
  #1=SelectRowSet #1:%22 groupSpec: #41 groupings (#44) GroupCols(#18) targets: 384=%20
        From: #32 Source: %20,
  #8=SqlFunction #8 INTEGER COUNT From:#1 COUNT(#14),
  #14=1,
  #17=SqlCopy #17 INTEGER K From:%12 copy from 172,
  #18=SqlValueExpr #18 INTEGER K2 From:#1 Left:#17 Right:#19 #18(#17/#19),
  #19=2,
  #41=GroupSpecification #41(#44),
  #44=Grouping #44 Domain %24 GROUP (#18=0),..,
  %0=Domain TABLE (#8,#18) Display=2[#8,INTEGER],[#18,INTEGER] Aggs (#8),
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
        Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %4=SqlValue E %4 INTEGER From:%2,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %10=SqlValue F %10 CHAR From:%2,
  %11=Domain (E int, D char, K int, F char)  VIEW (%4,%6,#17,%10)[%4,INTEGER],[%6,CHAR],
        [#17,INTEGER],[%10,CHAR] structure=%2,
  %12=TableRowSet %12:#17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: %12 Target=125
        SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,#17|%15) Display=2[%6,CHAR],[#17,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%25 groupSpec: #41 groupings (#44) GroupCols(#18) targets: 384=%18
        Target=384 SRow:()  RemoteCols:(%4,%10) RemoteNames:(%4=E,%10=F) UsingTableRowSet %12,
  %19=Domain TABLE (%4,%6,#17,%10) Display=4[%4,INTEGER],[%6,CHAR],[#17,INTEGER],[%10,CHAR],
  %20=RestRowSetUsing %20:%26 targets: 384=%20 From: #32 Target=384 SRow:(132,172,195)
        ViewDomain: %26 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
  %21=Domain ROW (#18)[#18,INTEGER],
  %22=Domain TABLE (#8,#18|%4,%6,#17,%10) Display=2[#8,INTEGER],[#18,INTEGER],[%4,INTEGER],
        [%6,CHAR],[#17,INTEGER],[%10,CHAR] Aggs (#8),
  %23=Domain ROW (#18)[#18,INTEGER],
  %24=Domain ROW (#18)[#18,INTEGER],
  %25=Domain TABLE (#8,#18|%4,%6,#17,%10) Display=2[#8,INTEGER],[#18,INTEGER],[%4,INTEGER],
        [%6,CHAR],[#17,INTEGER],[%10,CHAR] Aggs (#8),
  %26=Domain TABLE (#8,#17,#18,%4,%6,%10)[#8,INTEGER],[#17,INTEGER],[#18,INTEGER],
        [%4,INTEGER],[%6,CHAR],[%10,CHAR],
  %27=SelectStatement %27 Union=#1)}
```

101

```
http://localhost:8180/DB/DB select COUNT(*),(4/2) as K2,E,'B' as D,4 as K,F from t group by K2
HTTP POST /DB/DB
select COUNT(*),(4/2) as K2,E,'B' as D,4 as K,F from t group by K2
Returning ETag: "23,_,155"
--> 1 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select COUNT(*),(1/2) as K2,E,'C' as D,1 as K,F from u group by K2
HTTP POST /DC/DC
select COUNT(*),(1/2) as K2,E,'C' as D,1 as K,F from u group by K2
Returning ETag: "23,_,134"
SQL> select count(*),k/2 as k2 from ww group by k2
|-----|--|
|COUNT|K2|
|-----|--|
|3    |0 |
|4    |2 |
|-----|--|
```

select avg(e) from ww

```
{(..,
  #1=SelectRowSet #1:%21 targets: 384=%20 From: #20 Source: %20,
  #8=SqlFunction #8 NUMERIC AVG From:#1 AVG(#12),
  #12=SqlValue E #12 INTEGER From:%2,..
  %0=Domain TABLE (#8) Display=1[#8,NUMERIC] Aggs (#8),
  %1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
      Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
  %2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
  %6=SqlCopy %6 CHAR D From:%12 copy from 132,
  %8=SqlCopy %8 INTEGER K From:%12 copy from 172,
  %10=SqlValue F %10 CHAR From:%2,
  %11=Domain (E int, D char, K int, F char)  VIEW (#12,%6,%8,%10)[#12,INTEGER],[%6,CHAR],
      [%8,INTEGER],[%10,CHAR] structure=%2,
  %12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: %12 Target=125
      SRow:(132,172,195) Target:125 VU,
  %13=SqlCopy %13 CHAR D From:%12 copy from 132,
  %14=SqlCopy %14 INTEGER K From:%12 copy from 172,
  %15=SqlCopy %15 CHAR U From:%12 copy from 195,
  %16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
  %17=Domain TABLE (%6,%8|%15) Display=2[%6,CHAR],[%8,INTEGER],[%15,CHAR],
  %18=RestRowSet %18:%22 targets: 384=%18 Target=384 SRow:()  RemoteCols:(#12,%10)
      RemoteNames:(#12=E,%10=F) UsingTableRowSet %12,
  %19=Domain TABLE (#12,%6,%8,%10) Display=4[#12,INTEGER],[%6,CHAR],[%8,INTEGER],[%10,CHAR],
  %20=RestRowSetUsing %20:%23 targets: 384=%20 From: #20 Target=384 SRow:(132,172,195)
      ViewDomain: %23 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
  %21=Domain TABLE (#8|#12,%6,%8,%10) Display=1[#8,NUMERIC],[#12,INTEGER],[%6,CHAR],
      [%8,INTEGER],[%10,CHAR] Aggs (#8),
  %22=Domain TABLE (#8|#12,%6,%8,%10) Display=1[#8,NUMERIC],[#12,INTEGER],[%6,CHAR],
      [%8,INTEGER],[%10,CHAR] Aggs (#8),
  %23=Domain TABLE (#8,#12,%6,%8,%10)[#8,NUMERIC],[#12,INTEGER],[%6,CHAR],
      [%8,INTEGER],[%10,CHAR],
  %24=SelectStatement %24 Union=#1)}
```

This time the documents returned from the remote servers are
"[{\"AVG\": 4.75, \"$#8\": {\"4\": 19}}]"
"[{\"AVG\": 5.66666666666667, \"$#8\": {\"3\": 17}}]"

```
http://localhost:8180/DB/DB select AVG(E),E from t
HTTP POST /DB/DB
select AVG(E),E from t
Returning ETag: "23,_,155"
--> 1 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select AVG(E),E from u
HTTP POST /DC/DC
select AVG(E),E from u
Returning ETag: "23,_,134"
--> 1 rows
Response ETag: 23,_,134
```

```
SQL> Select avg(e) from w
|----------------|
|AVG             |
|----------------|
|5.14285714285714|
|----------------|
```

select sum(e)*sum(e),d from ww group by d

```
{(..,
  #1=SelectRowSet #1:%25 groupSpec: #38 groupings (#41) GroupCols(#22) targets: 384=%23
      From: #29 Source: %23,
  #8=SqlFunction #8 INTEGER SUM From:#1 SUM(#12),
  #12=SqlValue E #12 INTEGER From:%3,
  #14=SqlValueExpr #14 Domain %20 From:#1 Left:#8 Right:#15 #14(#8*#15),
```

```
#15=SqlFunction #15 INTEGER SUM From:#1 SUM(#12),
#22=SqlCopy #22 CHAR D From:%13 copy from 132,
#38=GroupSpecification #38(#41),
#41=Grouping #41 Domain %27 GROUP (#22=0),..,
%0=Domain UNION Aggs (#8,#15),
%1=Domain TABLE (#14,#22) Display=2[#14,Domain INTEGER Aggs (#8,#15)],[#22,CHAR]
     Aggs (#8,#15),
%2=RestView %2 Domain %12 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
     Ppos: 384 Result _ UsingTableRowSet: %13 ViewTable:`7,
%3=VirtualTable %3 Domain %12 Definer=-502 LastChange=`7:%12 RestView=384,
%9=SqlCopy %9 INTEGER K From:%13 copy from 172,
%11=SqlValue F %11 CHAR From:%3,
%12=Domain (E int, D char, K int, F char)  VIEW (#12,#22,%9,%11)[#12,INTEGER],[#22,CHAR],
     [%9,INTEGER],[%11,CHAR] structure=%3,
%13=TableRowSet %13:%18 Indexes=[(%14)154] key (%14) targets: 125=%13 From: %13 Target=125
     SRow:(132,172,195) Target:125 VU,
%14=SqlCopy %14 CHAR D From:%13 copy from 132,
%15=SqlCopy %15 INTEGER K From:%13 copy from 172,
%16=SqlCopy %16 CHAR U From:%13 copy from 195,
%17=Domain TABLE (%14,%15,%16) Display=3[%14,CHAR],[%15,INTEGER],[%16,CHAR],
%18=Domain TABLE (#22,%9|%16) Display=2[#22,CHAR],[%9,INTEGER],[%16,CHAR],
%19=Domain UNION Aggs (#8,#15),
%20=Domain INTEGER Aggs (#8,#15),
%21=RestRowSet %21:%28 groupSpec: #38 groupings (#41) GroupCols(#22) targets: 384=%21
     Target=384 SRow:()  RemoteCols:(#12,%11) RemoteNames:(#12=E,%11=F)
     UsingTableRowSet %13,
%22=Domain TABLE (#12,#22,%9,%11) Display=4[#12,INTEGER],[#22,CHAR],[%9,INTEGER],[%11,CHAR],
%23=RestRowSetUsing %23:%29 targets: 384=%23 From: #29 Target=384 SRow:(132,172,195)
     ViewDomain: %29 Template: %21 UsingTableRowSet:%13 UrlCol:%16,
%24=Domain ROW (#22)[#22,CHAR],
%25=Domain TABLE (#14,#22|#12,%9,%11) Display=2[#14,Domain INTEGER Aggs (#8,#15)],
     [#22,CHAR],[#12,INTEGER],[%9,INTEGER],[%11,CHAR] Aggs (#8,#15),
%26=Domain ROW (#22)[#22,CHAR],
%27=Domain ROW (#22)[#22,CHAR],
%28=Domain TABLE (#14,#22|#12,%9,%11) Display=2[#14,Domain INTEGER Aggs (#8,#15)],
     [#22,CHAR],[#12,INTEGER],[%9,INTEGER],[%11,CHAR] Aggs (#8,#15),
%29=Domain TABLE (#12,#14,#22,%9,%11,#8,#15)[#12,INTEGER],
     [#14,Domain INTEGER Aggs (#8,#15)],[#22,CHAR],[%9,INTEGER],[%11,CHAR],[#8,INTEGER],
     [#15,INTEGER],
%30=SelectStatement %30 Union=#1)}
```

```
http://localhost:8180/DB/DB select (SUM(E)*SUM(E)),'B' as D,E from t group by D
HTTP POST /DB/DB
select (SUM(E)*SUM(E)),'B' as D,E from t group by D
Returning ETag: "23,_,155"
--> 1 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select (SUM(E)*SUM(E)),'C' as D,E from u group by D
HTTP POST /DC/DC
select (SUM(E)*SUM(E)),'C' as D,E from u group by D
Returning ETag: "23,_,134"
--> 1 rows
Response ETag: 23,_,134
```

```
SQL> select sum(e)*sum(e),d from ww group by d
----|-|
Col0|D|
----|-|
361 |B|
289 |C|
----|-|
```

Join:

select f,n from ww natural join m
```
{(..,
 454=Table 454 Domain `23 Definer=-502 LastChange=504:`23 Indexes:((461)485;(504)529)
      KeyCols: (461=True,504=True),
 461=TableColumn 461 INTEGER Definer=-502 LastChange=461 -511 Table=454,
 504=TableColumn 504 CHAR Definer=-502 LastChange=504 -508 Table=454,
 #1=SelectRowSet #1:%0 where (%26) matching (%4=(%21),%21=(%4)) targets: 384=%20,454=#33
      Source: #20,
 #8=SqlValue F #8 CHAR From:%2,
 #10=SqlCopy #10 CHAR N From:#33 copy from 504,
 #20=JoinRowSet #20:%24 where (%26) matching (%4=(%21),%21=(%4)) targets: 384=%20,454=#33
      INNER First: %27 Second: #33 on %4=%21,
```

```
#33=TableRowSet #33:%23 Indexes=[(#10)529,(%21)485] key (%21) order (%21) targets: 454=#33
      From: #33 Target=454 SRow:(461,504) Target:454 M, //
%0=Domain TABLE (#8,#10) Display=2[#8,CHAR],[#10,CHAR],
%1=RestView %1 Domain %11 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
      Ppos: 384 Result _ UsingTableRowSet: %12 ViewTable:`7,
%2=VirtualTable %2 Domain %11 Definer=-502 LastChange=`7:%11 RestView=384,
%4=SqlValue E %4 INTEGER From:%2,
%6=SqlCopy %6 CHAR D From:%12 copy from 132,
%8=SqlCopy %8 INTEGER K From:%12 copy from 172,
%11=Domain (E int, D char, K int, F char)  VIEW (%4,%6,%8,#8)[%4,INTEGER],[%6,CHAR],
      [%8,INTEGER],[#8,CHAR] structure=%2,
%12=TableRowSet %12:%17 Indexes=[(%13)154] key (%13) targets: 125=%12 From: #20 Target=125
      SRow:(132,172,195) Target:125 VU,
%13=SqlCopy %13 CHAR D From:%12 copy from 132,
%14=SqlCopy %14 INTEGER K From:%12 copy from 172,
%15=SqlCopy %15 CHAR U From:%12 copy from 195,
%16=Domain TABLE (%13,%14,%15) Display=3[%13,CHAR],[%14,INTEGER],[%15,CHAR],
%17=Domain TABLE (%6,%8|%15) Display=2[%6,CHAR],[%8,INTEGER],[%15,CHAR],
%18=RestRowSet %18:%19 targets: 384=%18 From: #20 Target=384 SRow:()  RemoteCols:(%4,#8)
      RemoteNames:(#8=F,%4=E) UsingTableRowSet %12,
%19=Domain TABLE (%4,%6,%8,#8) Display=4[%4,INTEGER],[%6,CHAR],[%8,INTEGER],[#8,CHAR],
%20=RestRowSetUsing %20:%19 targets: 384=%20 From: #20 Target=384 SRow:(132,172,195)
      ViewDomain: %19 Template: %18 UsingTableRowSet:%12 UrlCol:%15,
%21=SqlCopy %21 INTEGER E From:#33 copy from 461,
%23=Domain TABLE (%21,#10) Display=2[%21,INTEGER],[#10,CHAR],
%24=Domain TABLE (%4,%6,%8,#8,#10|%21) Display=5[%4,INTEGER],[%6,CHAR],[%8,INTEGER],
      [#8,CHAR],[#10,CHAR],[%21,INTEGER],
%25=SqlValueExpr %25 BOOLEAN From:#20 Left:%4 Right:%21 %25(%4=%21),
%26=SqlValueExpr %26 BOOLEAN Left:%4 Right:%21 %26(%4=%21),
%27=OrderedRowSet %27:%19 key (%4) order (%4) targets: 384=%20 Source: %20,
%28=SelectStatement %28 Union=#1)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t
Returning ETag: "23,_,155"
--> 4 rows
Response ETag: 23,_,155
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u
Returning ETag: "23,_,134"
--> 3 rows
Response ETag: 23,_,134
```

```
SQL> select f,n from ww natural join m
|-----|------|
|F    |N     |
|-----|------|
|Three|Trois |
|Vier |Quatre|
|Four |Quatre|
|Five |Cinq  |
|Six  |Six   |
|Sechs|Six   |
|-----|------|
```

Updatable RESTView

update ww set f='Eight' where e=8

Place the breakpoint in UpdateSearch.Obey() (Executable.cs, line 3348) to see the context objects:

```
{(..,
{(-538=TABLE,
 #1=UpdateSearch #1 Target: %19,
 #15=SqlValue F #15 CHAR From:%1,
 #17=Eight,
 #31=SqlValue E #31 INTEGER From:%1,
 #32=SqlValueExpr #32 BOOLEAN Left:#31 Right:#33 #32(#31=#33),
 #33=8,..,
%0=RestView %0 Domain %10 Definer=-502 LastChange=384 ViewDef (E int, D char, K int, F char)
      Ppos: 384 Result _ UsingTableRowSet: %11 ViewTable:`7,
%1=VirtualTable %1 Domain %10 Definer=-502 LastChange=`7:%10 RestView=384,
%5=SqlCopy %5 CHAR D From:%11 copy from 132,
%7=SqlCopy %7 INTEGER K From:%11 copy from 172,
%10=Domain (E int, D char, K int, F char)  VIEW (#31,%5,%7,#15)[#31,INTEGER],[%5,CHAR],
      [%7,INTEGER],[#15,CHAR] structure=%1,
```

```
%11=TableRowSet %11:%16 Indexes=[(%12)154] key (%12) targets: 125=%11 From: %11 Target=125
      SRow:(132,172,195) Target:125 VU,
%12=SqlCopy %12 CHAR D From:%11 copy from 132,
%13=SqlCopy %13 INTEGER K From:%11 copy from 172,
%14=SqlCopy %14 CHAR U From:%11 copy from 195,
%15=Domain TABLE (%12,%13,%14) Display=3[%12,CHAR],[%13,INTEGER],[%14,CHAR],
%16=Domain TABLE (%5,%7|%14) Display=2[%5,CHAR],[%7,INTEGER],[%14,CHAR],
%17=RestRowSet %17:%18 where (#32) matches (#31=8) targets: 384=%17
      Assigs:(UpdateAssignment Vbl: #15 Val: #17=True) Target=384 SRow:()
      RemoteCols:(#31,#15) RemoteNames:(#15=F,#31=E) UsingTableRowSet %11,
%18=Domain TABLE (#31,%5,%7,#15) Display=4[#31,INTEGER],[%5,CHAR],[%7,INTEGER],[#15,CHAR],
%19=RestRowSetUsing %19:%18 where (#32) matches (#31=8) targets: 384=%19 From: #8
      Assigs:(UpdateAssignment Vbl: #15 Val: #17=True) Target=384 SRow:(132,172,195)
      ViewDomain: %18 Template: %17 UsingTableRowSet:%11 UrlCol:%14)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where (E=8) and E=8
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where (E=8) and E=8
Returning ETag:
--> 0 rows
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where (E=8) and E=8
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where (E=8) and E=8
Returning ETag: "23,134,134"
--> 1 rows
Response ETag: 23,134,134
RoundTrip HEAD http://localhost:8180/DC/DC/u/E=8
HTTP HEAD /DC/DC/u
Returning ETag:
--> OK
RoundTrip POST http://localhost:8180/DC/DC update u set F='Eight' where (E=8) and E=8

HTTP POST /DC/DC
update u set F='Eight' where (E=8) and E=8

Returning ETag: "23,134,258"
--> OK
```

```
SQL> update ww set f='Eight' where e=8
0 records affected in t24
SQL>
```

Recall that the "affected" score is only for the local database t24. The trace from the server shows that the post to the remote database was successful.

6.11 Versioned Objects

Entity classes in the Pyrrho library are all subclasses of Versioned. A base table can be designated an Entity using the ENTITY metadata flag as illustrated in the example below. Not all base tables should be entities (for example, in the usual 3NF sort of database, there are auxiliary tables for many-to-many properties, and their rows are not entities).

As in the Java persistence API and Microsoft's LINQ, integrity constraints are supported with dynamic navigation properties, as illustrated in the example below.

The Versioned class currently looks like this:

```
class Versioned {
      public PyrrhoConnect conn;
      public string entity; // url or /tabledefpos/defpos
      public string version; // etag or ppos
};
```

It may seem surprising that these fields are not protected or read-only. But for example to create a new entity e (a new row in a base table) we would like to use its constructor, and at this time the entity and version information will not be known. Once we have constructed it, and we have an PyrrhoConnect conn to the correct database and role, we can call conn.Post(e), or if e.conn is valid, e.Post().

To retrieve entities use FindAll<C>(), FindOne<C>(), FindWith<C>(w) or Get<C>(w). These fill in the Versioned fields for all entities retrieved.

To modify an entity, simply change its fields as required, and call e.Put() or conn.Put(e). This will update the version field on commit (and possibly other fields because of triggers etc).

The Post and Put methods will fill in the entity and version information when the entity is committed (and overwrite other fields of this entity if they have been changed by triggers etc).

If the entity or version information is no longer correct, the Put and Delete methods will fail.

The client can have several versions of the same entity. However, there are few good reasons for doing this, and deep copying is required to make a copy of the client-side entity: it is simpler just to fetch another copy from the database.

In this demo, let us use the following simple database D (all the tables created with the ENTITY flag). Replace the machine\user string with the user string returned by Pyrrho's "select user" statement.

```
create role "Sales"
grant "Sales" to "machine\user"
set role "Sales"
create table "Customer" (id int primary key,"NAME" char, unique("NAME")) entity
insert into "Customer" values (10,'John'),(11,'Fred'),(12,'Mary')
[create table "Order" (id int primary key,cust int references "Customer", "OrderDate"
date,"Total" numeric(6,2)) entity]
[insert into "Order" values (1230,10,date'2022-05-10',34.56),(1231,12,date'2022-05-
11',67.89),(1234,11,date'2022-06-04',56.78)]
create table "Item" (id int primary key,"NAME" char, price numeric(6,2)) entity
insert into "Item"
values(71,'Pins',0.78),(72,'Pump',67.0),(73,'Crisps',0.89),(74,'Rug',56.78),(75,'Bag',33)
[create table "OrderItem" (it int,oid int references "Orders" on delete cascade,
  item int references "Item",qty int,primary key(oid,it)) entity]
[insert into "OrderItem" values
(100,1230,75,1),(101,1230,71,2),(102,1231,73,1),(103,1231,72,1),(103,1234,74,1)]
```
Note that NAME is enclosed in straight double quotes because NAME is a reserved word in SQL (there are some contexts in which this does not matter). Then
```
set role "Sales"
table "Role$Class"
```

generates code fragments for each of the tables created above that can be pasted into a C# program. The fragments can be extracted from the PyrrhoCmd standard output. Here is the first – the hidden numbers will probably be different for you:

```
using System;
using Pyrrho;

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

As described in the Pyrrho manual (sec 6.4) this code includes attributes for the fields corresponding to columns in the Customer table. Since NAME is a unique key there is a static method for obtaining a single Customer object given the name; but as is indicated this method simply calls conn.FindOne<..> as shown. Such classes also have navigation properties coming from foreign keys, as will be seen below.

The ENTITY metadata flag ensures that all of these classes are Versioned. Identifiers and class/table names are case sensitive (this is C#), and the navigation property names are auto-constructed (so that the property orders has an awkward spelling). FindWith<V> takes a list of (key,value) pairs and returns an ordinary array of V. The Versioned library does not require the use of Pyrrho's list and tree classes: we use only [] in this demo.

The autokey feature of Pyrrho can supply a suitable integer key value for a new entity, but this requires a (long) cast if the key value is used later.

March 2023

These classes have all public fields. This is so that, for example, you can modify fields of an object a before calling conn.Put(a), while the library can update the version and any other fields that the database has updated by triggers or other users. Attributes such as [Table(.., ..)] provide internal information for the server, and these values are always easy to identify: for example, in TableAttribute, this information is a pair (tabledefpos, lastschemachange) giving the defining position of the table and the position of the last change made to its schema.

The classes we get are placed in the main class of a demo program Program.cs, as indicated by the highlight below:

```
using Pyrrho;

namespace Demo
{
    /// <summary>
    /// The demo could be made more elegant with some app-specific helper methods
    /// and extra indexes (e.g. here the lookup for Customer by Name is useful)
    /// </summary>
    internal class Program
    {
        static PyrrhoConnect conn;
// For each class in the Role$Class output,
// PASTE all the code here apart from the using Pyrrho line
        static void Main()
        {
            conn = new PyrrhoConnect("Files=Demo;Role=Sales");
            conn.Open();
            try
            {
// Get a list of all orders showing the customer name
                var aa = conn.FindAll<Order>();
                foreach (var a in aa)
                    Console.WriteLine(a.ID + ": " + a.customer.NAME);
                if (aa.Length == 0)
                {
                    Console.WriteLine("The Order table is empty");
                    goto skip;
                }
// change the customer name of the first (update to a navigation property)
                var j = aa[0].customer;
                j.NAME = "Johnny";
                j.Put();
// add a new customer (autokey is used here)
                var g = new Customer() { NAME = "Greta" };
                conn.Post(g);
// place a new order for Mary (secondary index, single quotes optional here!)
                var m = conn.FindOne<Customer>("Mary");
                var o = new Order() { CUST = (long)m.ID,
                                OrderDate = new Date(DateTime.Now) };
                conn.Post(o);
// lookup a couple of items to add
                var p = conn.FindWith<Item>(("NAME","Pins"))[0];
                var i1 = new OrderItem() { OID=o.ID, ITEM=p.ID, QTY = 2 };
                conn.Post(i1);
                var b = conn.FindWith<Item>(("NAME","Bag"))[0];
                var i2 = new OrderItem() { OID=o.ID, ITEM = b.ID, QTY = 1 };
                conn.Post(i2);
// calculate the total for the new order (M indicates a decimal constant in C#)
                var t = 0.0m;
                foreach (var i in o.orderItems)
                    t += (decimal)(i.item.PRICE * i.QTY);
                o.Total = t;
                o.Put();
                // delete the last order for Fred
                var f = conn.FindOne<Customer>("Fred");
                var fo = f.orderss;
                conn.Delete(f.orderss[fo.Length - 1]);
            } catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.WriteLine("All steps completed");
        skip:    Console.ReadLine();
            conn.Close();
```

```
            }
        }
}
```

If the database has triggers these are called on Post, Put and Delete. Side effects update this (the Versioned object on which the method is called) but not other client objects. There is a method e.Get() that overwrites the fields of the current object with those of the latest version of the entity (if you want to retain both versions in the client, use conn.Get<C>(e)[0] to get a possibly different version of e)

If there is an explicit transaction in progress, these methods prepare the changes in the transaction, waiting for Commit or Rollback. Deferred triggers will be called on Commit.

Experiment with the above demo program by adding extra Console.Writeline statements before trying other changes.

March 2023

References

Crowe, M. K.: The Pyrrho Database Management System, University of the West of Scotland, (2005-18) www.pyrrhodb.com

Crowe, M. K.: The Pyrrho DBMS,
https://github.com/MalcolmCrowe/ShareableDataStructures/PyrrhoV7alpha

Crowe, M. K.: The Pyrrho DBMS, https://pyrrhodb/blogspot.com

Crowe, M.K., Matalonga, S., Laiho, M.: StrongDBMS: built from immutable components, DBKDA 2019

T. Krijnen, and G. L. T. Meertens, "Making B-Trees work for B". Amsterdam: Stichting Mathematisch Centrum, 1982, Technical Report IW 219

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)