

June 2022

# An Introduction to the Source Code of the Pyrrho DBMS

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



Version 7.0 (June 2022)

© 2022 Malcolm Crowe and University of the West of Scotland, UK

<b>An Introduction to the Source Code of the Pyrrho DBMS .....</b>	<b>1</b>
1. Introduction.....	5
1.1 ACID and Serializable Transactions .....	6
1.2 The transaction log .....	6
1.3 DBMS implementation principles.....	7
1.4 Shareable data .....	7
1.5 Transaction conflict.....	8
1.6 Transaction Validation .....	9
1.7 Shareable structures.....	9
1.8 Transformation: adding a node .....	9
1.9 The choice of programming language .....	10
1.10 Shareable database objects.....	11
1.11 An implementation library: first steps .....	11
1.12 DBObject and Database .....	11
1.13 Transaction and B-Tree.....	11
1.14 RowSet review .....	12
1.15 Integrity Constraints.....	13
1.16 Roles, Security, Views and Triggers.....	13
1.17 Compiled database objects.....	13
1.18 Type Tracker .....	14
2. Overall structure of the DBMS .....	15
2.1 Architecture.....	15
2.2 Key Features of the Design.....	16
2.3 Data Formats .....	17
2.3.1 Implementing the data file formats .....	17
2.3.2 Implementing SQL formats .....	17
2.3.3 Formats in the in-memory data structures .....	17
2.3.4 Client-server communications .....	18
2.4 Multi-threading, uids, and dynamic memory layout.....	18
2.5 The folder and project structure for the source code .....	20
3. Basic Data Structures .....	21
3.1 B-Trees and BLists .....	21
3.1.1 B-Tree structure .....	21
3.1.2 ATree<K,V> .....	21
3.1.3 TreeInfo.....	22
3.1.4 ABookmark<K,V> .....	22
3.1.5 ATree<K,V> Subclasses.....	23
3.2 Other Common Data Structures .....	23
3.2.1 Integer .....	23
3.2.2 Decimal .....	24
3.2.3 Character Data .....	24
3.2.4 Documents .....	24
3.2.5 Domain.....	24
3.2.6 TypedValue.....	25
3.2.7 Ident .....	26
3.3 File Storage (level 1).....	26
3.3.1 Client-server protocol .....	27
3.4 Physical (level 2).....	28
3.4.1 Physical subclasses (Level 2).....	28

3.4.2 Compiled and Framing .....	29
3.5 Database Level Data Structures (Level 3) .....	31
3.5.1 Basis .....	31
3.5.2 Database .....	32
3.5.3 Transaction .....	34
3.5.4 Role .....	34
3.5.5 DBObject .....	35
3.5.6 ObInfo .....	37
3.5.7 Domain .....	37
3.5.8 Table .....	38
3.5.9 Index .....	38
3.5.10 SqlValue .....	39
3.5.11 Check .....	41
3.5.12 Procedure .....	41
3.5.13 Method .....	41
3.5.14 Trigger .....	41
3.5.15 Executable .....	42
3.5.16 View .....	43
3.6 Level 4 Data Structures .....	44
3.6.1 Context .....	44
3.6.2 Activation .....	45
3.6.3 RowSet .....	46
3.6.4 Cursor .....	49
3.6.5 Rvv .....	50
3.6.6 ETags .....	50
4. Locks, Integrity and Transaction Conflicts .....	51
4.2 Transaction conflicts .....	51
4.2.1 ReadConstraints .....	51
4.2.2 Physical Conflicts .....	52
4.2.3 Entity Integrity .....	53
4.2.4 Referential Integrity (Deletion) .....	53
4.2.5 Referential Integrity (Insertion) .....	53
4.4 System and Application Versioning .....	54
5. Parsing .....	55
5.1 Connection .....	55
5.2 Lexical analysis .....	55
5.3 Parser .....	56
5.3.1 Execute status and parsing .....	56
5.3.3 Parsing routines .....	57
6. Query Processing and Code Execution .....	58
6.1 Overview of Query Analysis .....	58
6.1.1 Context and Ident management .....	59
6.1.2 Identifier definition .....	60
6.1.3 Alias and Subquery .....	60
6.1.4 Replacement rules .....	60
6.1.5 References and Resolution .....	60
6.1.6 A worked example .....	61
6.2 RowSets and Context .....	65
6.2.1 TransitionRowSet and TableActivation .....	66
6.2.2 Aggregate functions .....	67

6.2.3 Views .....	69
6.2.4 RestViews .....	69
6.3 SqlValue vs TypedValue .....	70
6.4 Persistent Stored Modules.....	70
6.5 Trigger Implementation .....	71
6.6 View Implementation.....	79
6.7 Prepared Statement implementation .....	85
6.8 Stored Procedure implementation.....	86
6.9 User-defined Types Implementation.....	88
6.10 RESTView implementation .....	90
6.10.1 The HTTP1.1 model (test 22) .....	90
6.10.2 The scripted POST model (test 23).....	96
6.10.3 RestView with a Using table.....	98
6.11 Versioned Objects .....	109
7. Permissions and the Security Model.....	113
7.1 Roles .....	113
7.1.1 The schema role for a database .....	113
7.1.2 The guest role (public) .....	113
7.1.3 Other roles.....	113
7.2 Effective permissions.....	114
7.3 Implementation of the Security model.....	114
7.3.1 The Privilege enumeration .....	114
7.3.2 Checking permissions .....	115
7.3.3 Grant and Revoke .....	115
7.3.4 Permissions on newly created objects.....	115
7.3.5 Dropping objects .....	115
7.3.6 Implementation details.....	116
7.4 Mandatory Access Control .....	117
7.4.1 An example .....	118
7.5 The Type system and OWL support .....	123
8. The HTTP service .....	125
8.1 URL format.....	125
8.2 REST implementation.....	126
8.3 RESTViews.....	126
9. References.....	128
APPENDIX.....	129
Demo 1: Introducing Pyrrho DBMS .....	129
Demo 2: Pyrrho DBMS and concurrency .....	138
Part 1: Setting up the demo .....	138
Part 2: A Write-write conflict .....	142

# 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](http://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 explore 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<sup>1</sup> 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](http://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.

---

<sup>1</sup> 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 strict 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

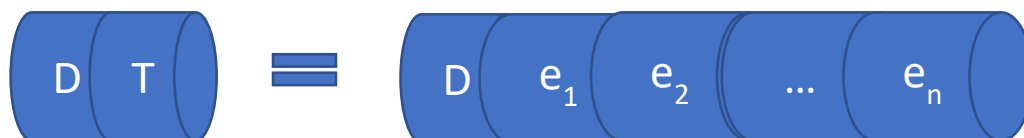
The Appendix contains four demonstrations that were included in the [tutorial](#) in the [DBKDA 2021 Program \(iaria.org\)](#). The first was about the transaction log. In Pyrrho the transaction log defines the content of the database (it is the only thing stored on disk).

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

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 D<sub>0</sub>.



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.

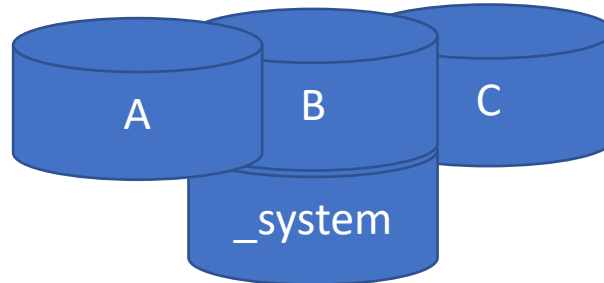
Specifically, strings in Java, Python and C# are immutable: if you make a change, you get a new string, and anyone who had a copy of the previous version sees no change.

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

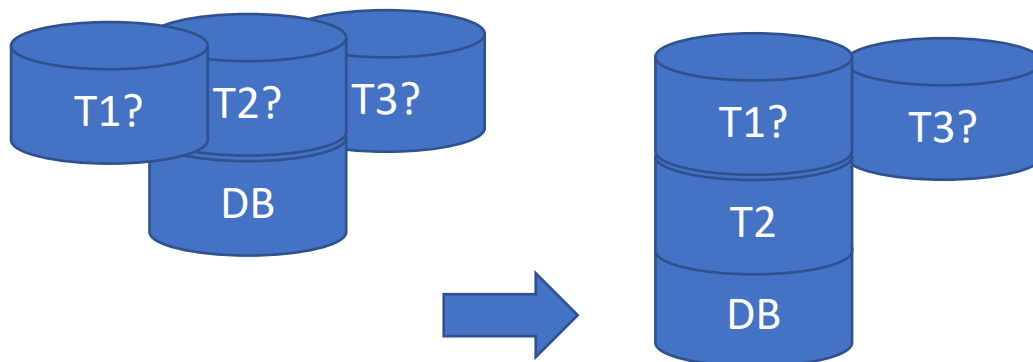
The addition operator can be used in these languages to create a sum of strings. This is basically the model for shareable data structures.

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 updated 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.

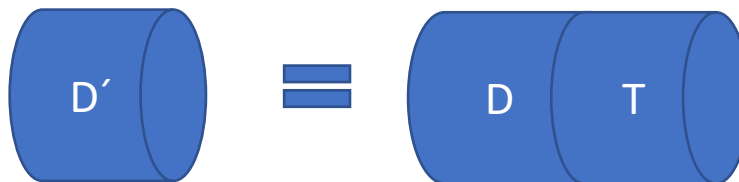


## 1.6 Transaction Validation

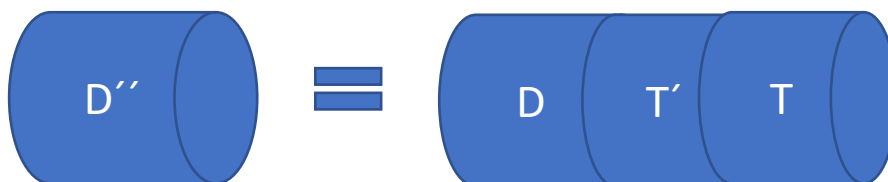
In the second demonstration in the Appendix, we look in detail at the `Commit()` method for a transaction, and the detection of conflicts.

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 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.

## 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).

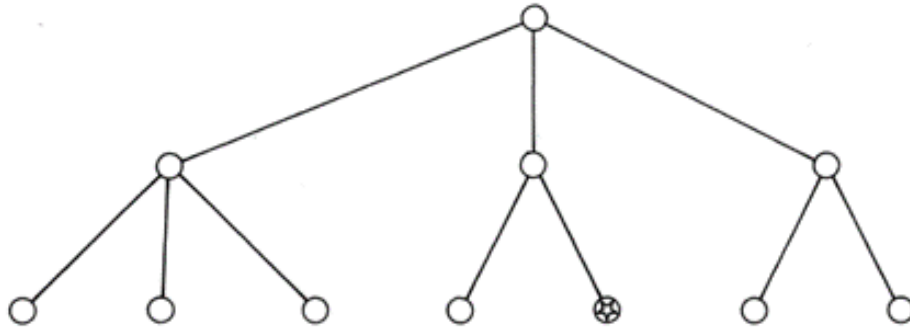
If all of the fields are, in turn, also known to be immutable, then there is no need to clone or copy fields: copying a pointer to the structure itself gives read-only access to the whole thing.

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.

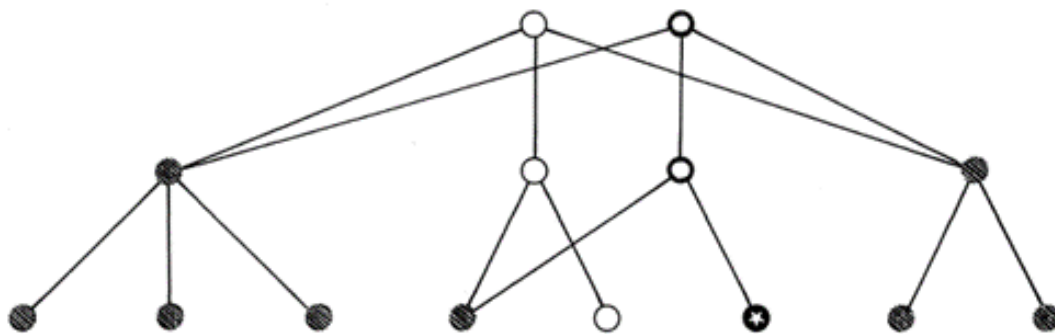
Pointers to shareable structures are never updated but can be replaced when we have a new version to reference. If this new version has some changed fields, it is perfectly fine to continue to use the same pointers for all the unchanged fields

## 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 "deshaired",  
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  $O(\log N)$ , 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  $\text{Add}(a,b)$
  - Easy to forget to use the return value  $a=\text{Add}(a,b)$
- 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

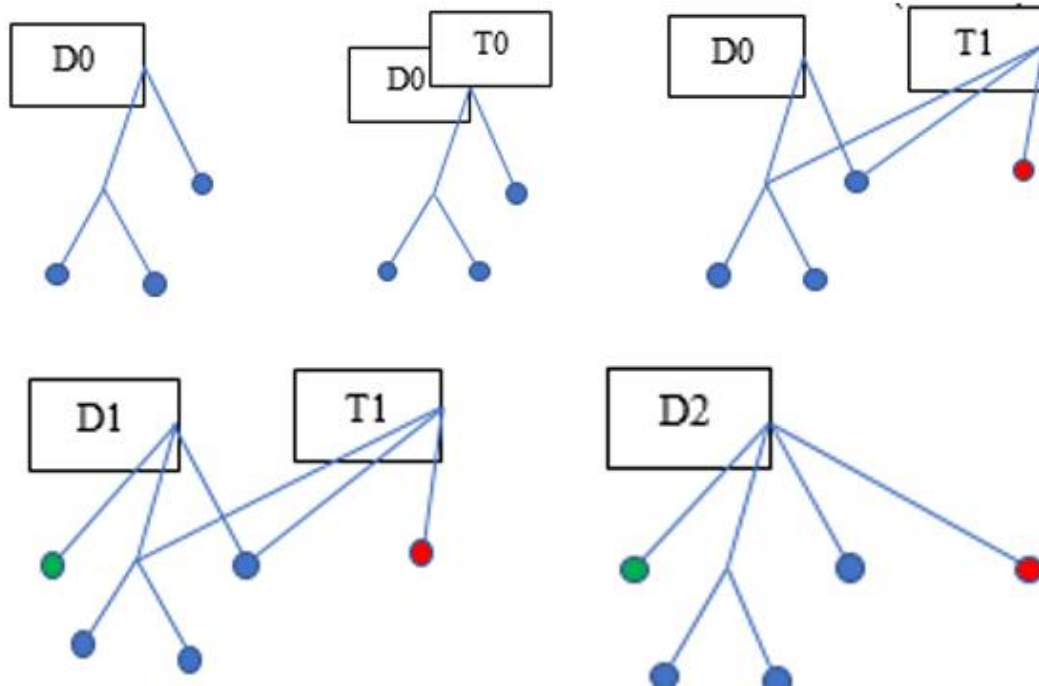
**DBObject** is a subclass of **Basis**, with a public readonly field called defpos, of type long. The defpos (defining position) acts as a uid or unique identifier for every object in the database. As described in section 2.3 below, the range of values of long is divided into ranges for database objects 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 properties 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. Following construction, changes to Rowset objects 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.

The third demonstration in the tutorial illustrated this approach by considering Views, including views of updatable joins. The mechanism used in this version has changed, however, so the corresponding details for the current design are illustrated here in section 6.6 below.

The current implementation focusses on making improvements during left-to-right parsing. 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.<sup>2</sup>

### **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.<sup>3</sup>

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,

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.

Granting permissions to roles and users is easy. Pyrrho diverges from the SQL standard in making the revoking of permissions equally easy. See section 7.12 of the Pyrrho manual.

### **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.

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 the compilation of such programmable elements is done on definition or database load. Thus, many database objects, including views, procedures, check

---

<sup>2</sup> See the last two paragraphs of section 4.23.2 of the SQL standard ISO 9075-02 (2016).

<sup>3</sup> This is a departure from the SQL standard ISO9075, and documented in section 7.12 of the Pyrrho manual.

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.

## **1.18 Type Tracker**

An interesting, and possibly controversial, issue is the TypeTracker. This is maintained by a Role, and allows data for a Record to be interpreted according to the schema as it stood when the record was laid down in the database. The user inserting the record must have permissions to Insert, and the Record will be constructed by that Role in accordance with the table definer's current specification for the Table. The type tracker mechanism ensures that retrieval of this record will use the same column type specifications (which may be a subtype of the current type), subject only to object renaming. Entries in the Log will continue to show the identifiers valid at the time.

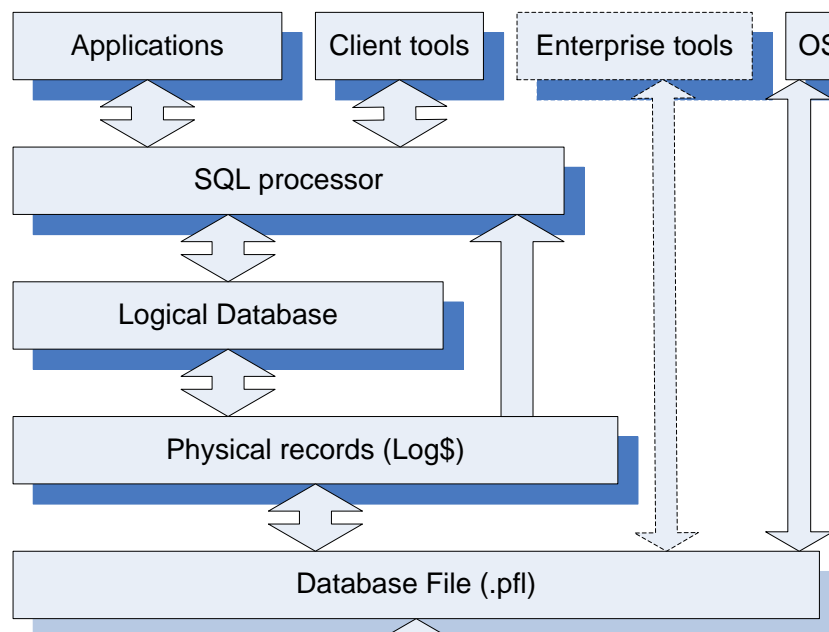
A consequence is that Alter statements must check existing rows for validity, with actions restrict, cascade, delete or set null. See the Pyrrho manual for Pyrrho-specific syntax. Any side effects will be fully recorded in the log.

## 2. Overall structure of the DBMS

### 2.1 Architecture

The following diagram shows the DBMS as a layered design. There is basically a namespace for each of four of the layers, and two other namespaces, 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 <sup>4</sup> , including Bookmarks for traversing them, are immutable and shareable <sup>5</sup> .
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 <sup>6</sup> are immutable and shareable, and give access to the version of the RowSet that created them.



<sup>4</sup> System.Exception is not shareable and so all its subclasses are not Shareable.

<sup>5</sup> Shareable classes can have mutable subclasses (shareability is not heritable). 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.

<sup>6</sup> Cursors used to examine the transaction log directly (LogSystemBookmark and its subclasses, and SysAuditBookmarks) are not shareable.

## 2.2 Key Features of the Design

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

1. Transaction commits correspond one-to-one to disk operations: completion of a transaction is accompanied by a force-write of a database record to the disk. The engine waits for this to complete in every case. Some previous versions of Pyrrho had a 5-byte end-of-file marker which was overwritten by each new transaction, but from version 7, all physical records once written are immutable. Deletion of records or database objects is a matter for the logical database, not the physical database. This makes the database fully auditable: the records for each transaction can always be recovered along with details about the transaction (the user, the timestamp, the role of the transaction).
2. Because data is immutable once recorded, the physical position of a record in the data file (its “defining position”) can be used to identify database objects and records for all future time (as names can change, and update and drop details may have a later file position). The transaction log threads together the physical records that refer to the same defining position but, from version 7, Pyrrho maintains the current state of base table rows in memory (using the TableRow class), and does not follow such non-scalable trails
3. Data structures at the level of the logical database (Level 3) are immutable and shareable. For example, if an entry in a list is to be changed, what happens at the data structure level is that a replacement element for the list is constructed and a new list descriptor which accesses the modified data, while the old list remains accessible from the old list descriptor. In this way creating a local copy or snapshot of the database (which occurs at the start of every transaction) consists merely to making a new header for accessing the lists of database objects etc. As the local transaction progresses, this header will point to new headers for these lists (as they are modified). If the transaction aborts or is rolled back, all of this data can be simply forgotten, leaving the database unchanged. With this design total separation of concurrent transactions is achieved, and local transactions always see consistent states of the database.
4. When a local transaction commits, however, the database cannot simply be replaced by the local transaction object, because other transactions may have been committed in the meantime. If any of these changes conflict with data that this transaction has read (read constraints) or is attempting to modify (transaction conflict), then the transaction cannot be committed. If there is no conflict, the physical records proposed in the local transaction are relocated onto the end of the database.
5. Following a successful commit, the database is updated using these same physical records. Thus all changes are applied twice – once in the local transaction and then after transaction commit – but the first can be usefully seen as a validation step, and involves many operations that do not need to be repeated at the commit stage: evaluation of expressions, check constraints, execution of stored procedures etc.
6. From version 7, database objects such as tables and domains cannot be modified if they hold data. The semantics of such changes in previous versions were not really manageable. There are necessarily several mutable structures: Reader, Writer, Context, and Physical (level 2). Physical objects are used only for marshalling serialisation and associated immutable objects replace Physicals in Level 3.
7. Data recorded in the database is intended to be non-localised (e.g. it uses Unicode with explicit character set and collation sequence information, universal time and date formats), and machine-independent (e.g. no built-in limitations as to machine data precision such as 32-bit). Default value expressions, check constraints, views, stored procedures etc are stored in the physical database in SQL2011 source form, and parsed to a binary form when the database is loaded.
8. The database implementation uses an immutable form of B-Trees throughout (note: B-Trees are *not* binary trees). Lazy traversal of B-Tree structures (using immutable bookmarks) is used throughout the query processing part of the database. This brings dramatic advantages where search conditions can be propagated down to the level of B-Tree traversal.
9. Traversing a rowset recovers rows containing TypeValues, and the bookmark for the current row becomes is accessible from the Context. This matches well with the top-down approach to parsing and query processing that is used throughout Level 4 of the code. In v7, the evaluation stack is somewhat flattened. A new Context is pushed on the context stack for a new procedure block or



activation, or when there is a change of role. The new context receives a copy of the previous context's immutable tree structures, which are re-exposed when the top of the stack is removed.

10. The aim of SQL query processing is to bridge the gap between bottom-up knowledge of traversable data in tables and joins (e.g. columns in the above sense) and top-down analysis of value expressions. Analysis of any kind of query goes through a set of stages: (a) source analysis to establish where the data is coming from, (b) computation of the result data types, (c) conditions analysis which examines which search and join conditions can be handled in table enumeration, (d) ordering analysis which looks not only at the ordering requirements coming from explicit ORDER BY requests from the client but also at the ordering required during join evaluation and aggregation, and finally (e) RowSet construction, which chooses the best enumeration method to meet all the above requirements.
11. Executables and SqlValues are immutable level 3 objects that are constructed by the parser. They are not stored in the database, but reconstructed on creation or loading. Procedure bodies being read from the database can contain SqlValues with positions allocated according to the current Reader position. Objects constructed from the input stream of the transaction can use the position in the input stream provided that this accumulates from the start of the transaction rather than the start of the current input line. This is the responsibility of Server.Execute(sql) and SqlHTTPService parser calls.

## 2.3 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<sup>7</sup>.

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 DBOject 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 DBOjects. Some DBOjects such as Table, or TableColumn correspond to Physical objects such as PTable,

---

<sup>7</sup> 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.

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 DBO object, 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.

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<sup>8</sup> 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<sup>62</sup>-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.

---

<sup>8</sup> Unique within the context. Different contexts may have different versions of the database, but always start with the latest committed version of the database.

Transactions contain uncommitted objects (proposed physical records), whose uids are in the range  $2^{62}..5 \times 2^{60}$ , 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<sup>9</sup>. 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<sup>10</sup>.

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.

During query analysis, transactions allocate space for objects local to the processing of the current Command. Uids in the range  $5 \times 2^{60}..6 \times 2^{60}-1$  are 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 are given uids in the range  $6 \times 2^{60}..7 \times 2^{60}-1$ , 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,  $7 \times 2^{60}..8 \times 2^{60}-1$  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	$-8 \times 2^{60}..-1$	Basis._uid (downwards)	Global	
File positions	$0..4 \times 2^{60}-1$	0 (upwards)	Persisted in file	
New Physicals	$4 \times 2^{60}..5 \times 2^{60}-1$	Database.nextPos (up)	Local to Transaction	!
Query analysis	$5 \times 2^{60}..6 \times 2^{60}-1$	Database.nextId (up)	Local to Statement	#
Executables	$6 \times 2^{60}..7 \times 2^{60}-1$	Database.nextStmt (up)	Local to Database	`
Prepared Stmts	$7 \times 2^{60}.. \text{nextPrep}$	Database.nextPrep (up)	Local to Connection	%
Heap storage	$\text{nextPrep}..8 \times 2^{60}-1$	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:

<sup>9</sup> This stack is useful for recovery in the implementation of SQL exception handling.

<sup>10</sup> During rowset traversal, cursors associate column uids with their values in that row.

- 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.
- 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 structure. Unusually, it has subfolders for isolating the AssemblyInfo for each of the 3 solutions.
- The Common, Level1, Level2, Level3, and Level4 folders contain the real code base for the DBMS.
- 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 `Pyrho.Common` namespace: all of the classes in this namespace are immutable and shareable<sup>11</sup>.

#### 3.1 B-Trees and BLists

Almost all indexing and cataloguing tasks in the database are done by B-Trees. 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 `Comparable`. If the values are also `Comparable`, CTree is used instead (CTree is then `Comparable`).

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

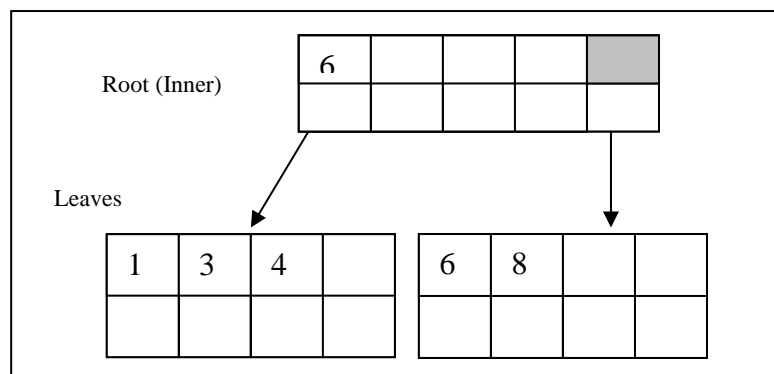
The B-Tree is a widely-used, fully scalable mechanism for maintaining indexes. B-Trees as described in textbooks vary in detail, so the following account is given here to explain the code.

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 KeyValuePair. An Inner Bucket contains key-value pairs whose values are pointers to Buckets, and a further pointer to a Bucket, so that an Inner Bucket contains pointers to N+1 Buckets altogether (“at the next level”). In each Bucket the KeyValuePairs are kept in order of their key values, and the key-value pairs in Inner buckets contain the first key value for the next lower-level Bucket, so that the extra Bucket is for all values bigger than the last key. All of these classes take a type parameter to indicate the key type.

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

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

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



##### 3.1.2 ATree<K,V>

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

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

<sup>11</sup> Remember that neither immutability nor shareability are heritable. Mutability and non-shareability are.

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 IEnumerable 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 (starts 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 rennumbers 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 primitive 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

TInterval
TMTree
TMultiset
TNull
TNumeric
TPartial
TPeriod
TReal
TRow
TRvv
TTimeSpan
TTypeSpec
TUnion
TXml

### 3.2.7 Ident

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

CompareTo(ob)	Support alphanumeric comparison of Ident
string ident	The head portion of the Ident
Ident(...)	<i>Numerous constructors</i>
Iix iix	iix contains a long, usually obtained from the lexical position, a defpos uid, and optionally a query uid. For compiled objects the lexical position and the defpos are generally different, because there are additional rules for executable uids.
int Length	The number of segments in the Ident
Ident sub	The tail of the Ident
string ToString()	A readable version of the Ident

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

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

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

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

## 3.3 File Storage (level 1)

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

The locking required for transaction management is limited to locking the underlying FileStream during Commit() and managing the static list of databases. The FileStream is also locked during seek-read combinations when Readers are created. The binary file transaction log format is almost unchanged since the earliest versions of Pyrrho: every edition of the user manual has documented the file format as a

sequence of physical records<sup>12</sup>. 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	{ Warning }	TransactionReport	
CommitAndReport1			
CommitAndReportTrace	{ Warning }	TransactionReportTrace	
CommitAndReportTrace1			
CommitTrace	{ Warning }	DoneTrace	
Execute	{ Warning }	Done	
		Schema	
ExecuteTrace	{ Warning }	DoneTrace	
		Schema	
ExecuteNonQuery	{ Warning }	Done	
ExecuteNonQueryTrace	{ Warning }	DoneTrace	
ExecuteReader	{ Warning }	Done	
		Schema	
Get	{ Warning }	Schema	
Get1		Done	
		NoData	
Get2	{ Warning }	Schema1	
		Done	
		NoData	
GetFileNames		Files	
GetInfo	{ Warning }	NoData	
		Columns	
Post	{ Warning }	Done	
	Schema		
Put	{ Warning }	Done	
	Schema		
Prepare		Done	
ReaderData		NoData	
		ReaderData	
ResetReader		Done	
Rest	{ Warning }	Schema	
Rollback		Done	
TypeInfo		(data)	

<sup>12</sup> 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  $4 \times 2^{60}$ . 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  $4 \times 2^{60}$  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 Role and 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	<i>PProcedure</i>	<i>Physical</i>	<i>Stored procedure/function (deprecated, see below)</i>
17	PTrigger	Physical	Define a trigger
18	PView	Physical	Define a view
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	<i>PTemporalView</i>	<i>Physical</i>	<i>A View for a Temporal Table (obsolete)</i>
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	<i>Partitioned</i>	<i>Physical</i>	<i>Record a partitioning of the database</i>
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	<i>PView1</i>	<i>Pview</i>	<i>Define update rules for a view (obsolete)</i>
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	<i>Partition</i>	<i>Physical</i>	<i>Manages schema for a partition</i>
55	<i>Reference1</i>	<i>Reference</i>	<i>For cross-partition references</i>
56	ColumnPath	Physical	Records path selectors needed for constraints
57	Metadata2	Metadata	Additional fields for column information
58	Index2	Index	Supports deep structure
59	<i>DeleteReference1</i>	<i>Reference</i>	
60	Authenticate		Credential information for web-based login
61	RestView	View	Views defined over HTTP
62	TriggeredAction		Distinguishes triggered parts of a transaction
63	<i>RestView1</i>	<i>RestView</i>	<i>deprecated</i>
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 in 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.

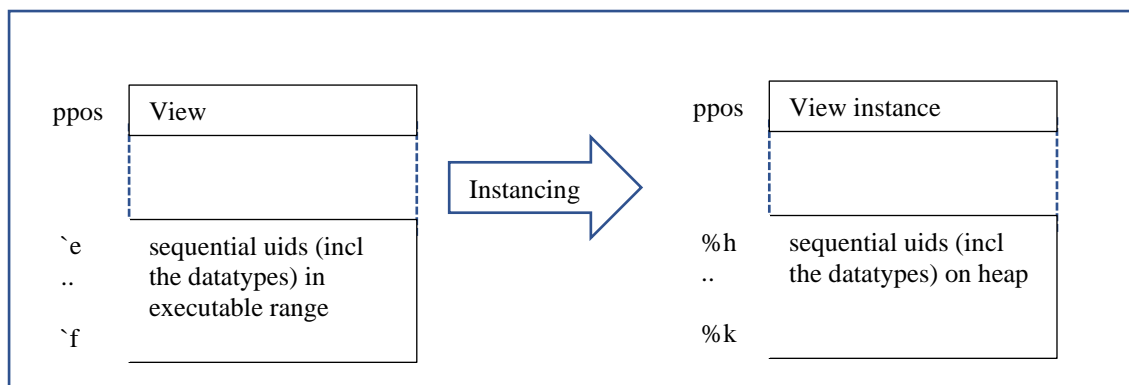
There is a subclass of Physical for the Compiled Level2 objects, to provide helper methods for the compilation process. The in-memory data structures resulting from parsing include SqlValues, RowSets and Executables, and an associated data structure called Framing holds a shareable version of the objects 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 UDTtype 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. Instanting 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<sup>13</sup>:



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<sup>14</sup>. 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. Instanting of view objects may occur during parsing, but to executable range instead of heap.
Compile	For creating compiled objects (including method headers) Use executable range for uids, so that Framing object is ready at the end of parsing.

<sup>13</sup> In the code, `e is called instSFirst, `f is called instSLast, and %h is called instDFirst.

<sup>14</sup> There are also no concerns about “SQL injection”.

	Instanting 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, 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 if cx.parse was Obey reset afterwards with DoneCompileParse
- In PXX(x,wr) constructor, do not call Fix for framing object references.
- In OnLoad for XX we have framing = new Framing(psr.cx);
- In View.Instance(cx) we have cx.instHFirst = cx.nextHeap near start.
- 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
Resultt	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 predefined 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<sup>15</sup>.

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<sup>16</sup>. The current committed state of each database can be obtained from the Database.databases list, and the current state of the transaction logs can be obtained from the Database.databases list. Both of these structures are protected, and accessed using locking in just 2 or 3 places in the code.

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

Class	Base Class	Description
Database	Basis	
Transaction	Database	

The level 3 Database structure maintains the following data:

- Its name, usually just the name of the database file (not including the extension), but see above in this section

<sup>15</sup> For renamable objects, the role’s name for the object is held in an ObInfo structure stored in the Role.

<sup>16</sup> Reader and Writer both contain Contexts, whose db field is a snapshot of the Database.



- 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<sup>62</sup>, and this is derived from the lexical position in all SQL read from the client since the start of the transaction.<sup>17</sup>

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 <sup>18</sup>	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 <code>_system._role</code>	-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

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

Name	Description
------	-------------

<sup>17</sup> Transactions running in different threads cannot see each other's data, so concurrent transactions will use the same range of defining positions.

<sup>18</sup> 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.

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

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<sup>19</sup>. 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<sup>20</sup>, so that in v7 the Role becomes responsible for all name lookups for level 3 objects. Level 4 objects contains names directly.

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

<sup>19</sup> 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.

<sup>20</sup> 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!

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
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
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	orderCategory	OrderCategory		-83
OrderFunc	orderFunc	long	DBObject	-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>	DBObject	-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 UDT type 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<sup>21</sup>). 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.

<sup>21</sup> Accessibility and visibility of columns in a rowset is managed with the Finder mechanism, see sec 3.6.3.

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<sup>22</sup> 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

<sup>22</sup> 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 compiled 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 `RESTRView` 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 `RESTRViews`.

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 `RESTRView`. The column definitions for the `RESTRView` are handled by an anonymous `PTable` whose name is the source of the column definitions, which has an associated `VirtualTable` `DBObject` (subclassing `Table`), and its file position identifies the `RESTRView`'s domain. Associated metadata uses further physical objects.

The framing for the `RestView` contains the Domain as defined in the OF part of the `RESTRView` definition and its columns as `SqlValues`. On instancing, a `RestRowSet` (subclassing `TableRowSet`) is added to the Context, whose columns are copies of the `VirtualTable`'s columns. The columns and their domains do not need to have physical locations in the database as they are remote.

In `RestViews` it is not possible for the rowset generated during `RestRowSet` evaluation to contain more than one reference to the same view. Although each reference to the restview may have different filters and aggregations, the `RestRowSets`' remote queries are passed to separate remote connections (even if they are to the same contributor). It might be possible to form a remote join of contributions that use the same restview, with aliases for the two instances, but such an idea does not generalise comfortably, and it would be better in such cases to agree on a new restview to specify the remote join.

As described in a worked example in section 6 below, there are currently two `RESTRViews` 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 is completed. Activations provide for complex Condition and Signal handling, similar to the operation of exceptions.

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

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

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

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

Contexts 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<sup>23</sup>. 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.

<sup>23</sup> They are included in the pre-compiled state of a compiled object that defines local queries (se 3.4.2)..

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.

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<sup>24</sup>. As the RowSet is constructed from its sources, ordering and filtering operations are distributed into a pipeline of RowSets, whose rowTypes have finders that keep track of which rowSet has a copy of the current value of each for its column uids. Formally the finder is a pair RowSet, Column, and associations (uid, Finder) identify the location of the current value of a base column by uid in the current set of cursors. 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<sup>25</sup>. 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)

<sup>24</sup> 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()).

<sup>25</sup> TransitionRowSet and some system rowsets are unidirectional.

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.
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 formed 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
finder	BTree<long,Finder>	RowSet._Finder	-403
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,Finder>	TransitionTableRowSet.Map	-433
matches	BTree<long,TypedValue>	Query._Matches	-182
mtree	MTree	Index.Tree	-164
needed	BTree<long,Finder>	RowSet._Needed	-401
offset	int	RowSetSection.Offset	-438
proc	Procedure	RoutineCallRowSet.Proc	-436
ra	TriggerContext	TransitionRowSet.Ra	-424
rb	TriggerContext	TransitionRowSet.Rb	-422
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	BTree<long,Finder>	TransitionTableRowSet.RMap	-434
row	TRow	TrivialRowSet.Singleton	-405
rowOrder	CList<long>	RowSet.RowOrder	-404
rows	Blist<TRow>	RowSet._Rows	-407
rt	CList<long>	(domain.rowType)	
second	long	JoinRowSet.Second	-448
size	int	RowSetSection.Size	-439
source	long	From.Source	-151
sqlRows	BList<long>	SqlRowSet.SqlRows	-413
sQMap	BTree<long,Finder>	SelectedRowSet.SQMap	-408
ta	TriggerContext	TransitionRowSet.Ta	-426
table	long	IndexRowSet.IxTable	-409
tabledefpos	long	SqlInsert._Table	-154
targetAc	Activation	TransitionRowSet.TargetAc	-430
targetInfo	ObInfo	TransitionRowSet.TargetInfo	-417
targetTrans	BTree<long,Finder>	TransitionRowSet.TargetTrans	-418
tb	TriggerContext	TransitionRowSet.Tb	-425
td	TriggerContext	TransitionRowSet.Td	-428
transTarget	TriggerContext	TransitionRowSet.TransTarget	-419
tree	RTree	OrderedRowSet._RTree	-412
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 finder listing all of the uids for which it provides values directly or indirectly.

The interface offered includes the following:

long _defpos	The row uid (or 0)
int display	The number of columns
TRow key	<i>Abstract</i> The current key

Cursor Last(Context cx)	<i>Abstract:</i> 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)	<i>Abstract:</i> 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)
Cursor PositionAt(pos)	Returns a bookmark for the given position, or null if there is none.
TableRow Rec()	<i>Abstract</i> 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 can use the current cursor of the appropriate rowset. The Context maintains the current set of Cursors in that context. and both contexts and rowsets manage a set of Finders to get the correct cursor and column to evaluate any currently accessible uid that is not a simple column.

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:  $P=p_0p_1\dots p_n$  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 if 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 altered
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 and 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<sup>26</sup>:

- The transaction has received SQL from the client.
- SQL code is found in a Physical being loaded from the transaction log (when the database 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:

---

<sup>26</sup> 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 as 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 #)<sup>27</sup> 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 instanting of their framed objects, using new uids in the executable range.<sup>28</sup>

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<sup>29</sup>. 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.

<sup>27</sup> 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.

<sup>28</sup> For Views defined in terms of other views, see the discussion and example in section 6.6 below.

<sup>29</sup> 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<sup>60</sup>, 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. The context uses a special structure called *done* that records the new version for each uid that has been scanned. If a part of the query contains more than one reference to the item being replaced, the enclosing structure may be replaced more than once (overwriting its entry in *done*), but when the scan reaches a reference to the changed uid it will not be rescanned.

Thus, during parsing, there are many replacements of one SqlValue (e.g. an alias placeholder) with another (e.g. the expression, column or subquery that it refers to). Such replacements extend to expressions in subqueries, parameter lists (including those of row constructors), grouping and ordering specifications. When parsing reaches the end of the sql statement, the process of query analysis is complete. The set of RowSets defined is then subject to review which takes place in a cascade<sup>30</sup>.

### 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.

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.

---

<sup>30</sup> This account deals with the construction of the rowsets that occurs during parsing. Subsidiary rowsets may need to be rebuilt during traversal of their parent (e.g. lateral joins, aggregations, results of procedure calls).

- 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.

### 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 either a domain other than Content or a well-defined parent (**from**).
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 some) 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.

### 6.1.6 A worked example

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

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

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

23	AUTHOR	a Table
34	INTEGER	
48	ID	for 23
70		an Index (ID) for 23
91	CHAR	
104	ANAME	for 23
148	BOOK	a Table
158	ID	for 148
182		an Index (ID) for 148
204	AID	for 148
230		an Index (AID) referencing 70
250	TITLE	for 148

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). 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')
```

Then, when we parse

```
123456789012345678901234
SELECT aname FROM author
```

aname is initially constructed as an unknown SqlValue with uid #8 and domain CONTENT. When the parser reaches the FROM keyword (at the start of ParseFromClause()), the objects in the Context (cx.obs) so far are:

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

where we see that ANAME already has a From:#1, and the defs table in the Context is just

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

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). The 2 in the cx.defs entry is the “select depth”, which becomes important for more complex queries. When we get into ParseTableReferenceItem, at about line 5944 in Parser.cs, we look up the base table AUTHOR and we are ready to construct a From RowSet for it. The From constructor also updates the ANAME #8, so that by line 5971 the list of objects has become

```
{(23=Table Name=AUTHOR 23 TABLE Definer=-502 Ppos=23:-538 Indexes:((48)70) KeyCols: (48=True),
 34=Domain 34 INTEGER,
 48=TableColumn 48 Domain 34 Definer=-502 Ppos=48 34 Table=23,
 91=Domain 91 CHAR,
 104=TableColumn 104 Domain 91 Definer=-502 Ppos=104 91 Table=23,
 #8=SqlCopy Name=ANAME #8 Domain 91 From:#19 copy from 104,
 #19=From #19:%3 targets: 23=%5 Source: %5 Target=23,
 %0=Domain %0 ROW (#8)[#8,CONTENT],
 %1=Domain %1 TABLE (#8)[#8,Domain 91 CHAR],
 %2=SqlCopy Name=ID %2 Domain 34 From:#19 copy from 48,
 %3=Domain %3 TABLE (#8|%2) Display=1[#8,Domain 91 CHAR],[%2,Domain 34 INTEGER],
```

```
%4=Domain %4 TABLE (%2,#8)[%2,Domain 34 INTEGER],[#8,Domain 91 CHAR],
%5=TableRowSet %5:%4 From: #19 SRow:(48,104) Target:23 Indexes: [(%2)=70]}}
```

The blue colour indicates objects that were read from the database file on first load of the database and not identified by the table reference (-538 is the system uid for a Table). The grey colour indicates objects unchanged from the previous list above (here a light grey is used since Domain %0 is no longer referenced), and red indicates a previous item that has been changed. New items are in black.

Note several subtleties here: we have copied the table information for AUTHOR into the context. We note that the tablecolumns do not actually refer directly to their names in the role<sup>31</sup>. We ensure that all of the columns of the From RowSet (#19, with Domain %2) have uids different from the tablecolumns, since there may be many references to them (this process is called instancing), so that we have replaced the unknown SqlValue #8 with a SqlCopy for the ANAME tablecolumn, and we include the unreferenced column ID (with the heap uid %2) in case it is referenced later in the SQL. We note that the From RowSet displays just one column (indicated here by the vertical bar in the rowType for domain %3). The instance TableRowSet %5 has Domain %4, which matches the order of the columns in the base table 23, but uses the instance uids. The SRow mapping assists with the correspondence to the base table columns for operations such as insert or update.

The From and TableRowSet are separate because it is quite likely that the From clause has more than one TableReferenceItem.

Although we have reached the end of this query, in general there could be much more to do. There might be extra table references to join or merge, and selection, grouping, and ordering. From the point of view of completing the parse, the Context at this stage has noted the following simple and composite definitions (in defs):

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

as these help to identify objects that may be referenced by name later in the SQL.

At the end of parsing (e.g. back in Start.cs at line 426), in this case, we have added the SelectRowSet and SelectStatement:

```
{(-538=TABLE,
 23=Table Name=AUTHOR 23 TABLE Definer=-502 Ppos=23:-538 Indexes:((48)70) KeyCols: (48=True),
 34=Domain 34 INTEGER,
 48=TableColumn 48 Domain 34 Definer=-502 Ppos=48 34 Table=23,
 91=Domain 91 CHAR,
 104=TableColumn 104 Domain 91 Definer=-502 Ppos=104 91 Table=23,
 #1=SelectRowSet #1:%1 targets: 23=%5 Source: #19,
 #8=SqlCopy Name=ANAME #8 Domain 91 From:#19 copy from 104,
 #19=From #19:%3 targets: 23=%5 Source: %5 Target=23,
 %0=Domain %0 ROW (#8)[#8,CONTENT],
 %1=Domain %1 TABLE (#8)[#8,Domain 91 CHAR],
 %2=SqlCopy Name=ID %2 Domain 34 From:#19 copy from 48,
 %3=Domain %3 TABLE (#8|%2) Display=1[#8,Domain 91 CHAR],
 [%2,Domain 34 INTEGER],
 %4=Domain %4 TABLE (%2,#8)[%2,Domain 34 INTEGER],[#8,Domain 91 CHAR],
 %5=TableRowSet %5:%4 From: #19 SRow:(48,104) Target:23 Indexes: [(%2)=70],
 %6=SelectStatement %6 Union=#1)}
```

```
SQL> SELECT aname FROM author
|-----|
|ANAME  |
|-----|
|Dickens|
|Conrad |
|-----|
SQL>
```

The arrangements are only a little different when we have identifier chains: consider

```
1      2      3      4
1234567890123456789012345678901234567890
SELECT book.title,b.id,b.aid FROM book b
```

The parser proceeds from left to right, and constructs ForwardReference objects for identifiers whose meaning is not yet clear, as here for BOOK and B: we discover that BOOK is a table with alias B only after the FROM keyword), and we create lexical uids for the two columns. We use the Context's definitions list to manage the (unknown) prefixes, so that just before identifying the table (line 5939 again) we have objects whose names we know, but nothing else:

```
{(#8=ForwardReference Name=BOOK #8 Domain %0 Definer=-501 Ppos=#8,
 #13=SqlValue Name=TITLE #13 CONTENT From:#1,
```

<sup>31</sup> This database has no users defined so that the role is the schema role, where the table columns have names ID and ANAME, and thus the unknown identifier ANAME has been resolved.

```
#19=ForwardReference Name=B #19 Domain %1 Definer=-501 Ppos=#19,
#21=SqlValue Name=ID #21 CONTENT From:#1,
#26=SqlValue Name=AID #26 CONTENT From:#1,
%0=Domain %0 ROW (#13)[#13,CONTENT],
%1=Domain %1 ROW (#21,#26)[#21,CONTENT],[#26,CONTENT],
%2=Domain %2 TABLE (#13,#21,#26)[#13,CONTENT],[#21,CONTENT],[#26,CONTENT]]}^32
```

and the relationships between identifiers B, BOOK, AID, ID, and TITLE are held in the Context's definitions table defs:

```
{AID=(2:#26,);
B=(2:#24,AID=(2:#26,);ID=(2:#21,)););
BOOK=(2:#8,TITLE=(2:#13,)););
ID=(2:#21,);
TITLE=(2:#13,);,);}
```

By line 5971 we have discovered that BOOK alias B is a table, and we have

```
{(34=Domain 34 INTEGER,
 91=Domain 91 CHAR,
 148=Table Name=BOOK 148 TABLE Definer=-502 Ppos=148:-538 Indexes:((158)182,(204)230)
  KeyCols: (158=True,204=True),
 158=TableColumn 158 Domain 34 Definer=-502 Ppos=158 34 Table=148,
 204=TableColumn 204 Domain 34 Definer=-502 Ppos=204 34 Table=148,
 250=TableColumn 250 Domain 91 Definer=-502 Ppos=250 91 Table=148,
 #8=ForwardReference Name=BOOK #8 Domain %0 Definer=-501 Ppos=#8,
 #13=SqlCopy Name=TITLE #13 Domain 91 From:#35 copy from 250,
 #19=ForwardReference Name=B #19 Domain %1 Definer=-501 Ppos=#19,
 #21=SqlCopy Name=ID #21 Domain 34 From:#35 copy from 158,
 #26=SqlCopy Name=AID #26 Domain 34 From:#35 copy from 204,
 #35=From #35:%3 targets: 148=%5 Source: %5 Alias B Target=148,
 %0=Domain %0 ROW (#13)[#13,CONTENT],
 %1=Domain %1 ROW (#21,#26)[#21,CONTENT],[#26,CONTENT],
 %2=Domain %2 TABLE (#13,#21,#26)[#13,Domain 91 CHAR],
 [%21,Domain 34 INTEGER],[#26,Domain 34 INTEGER],
 %3=Domain %3 TABLE (#13,#21,#26) Display=3[#13,Domain 91 CHAR],
 [%21,Domain 34 INTEGER],[#26,Domain 34 INTEGER],
 %4=Domain %4 TABLE (#21,#26,#13)[#21,Domain 34 INTEGER],
 [%26,Domain 34 INTEGER],[#13,Domain 91 CHAR],
 %5=TableRowSet %5:%4 From: #35 SRow:(158,204,250) Target:148
 Indexes: [(#21)=182,(#26)=230]]}
```

```
SQL> SELECT book.title,b.id,b.aid FROM book b
-----|-----|
|TITLE|ID|AID|
|-----|-----|
|Dombey & Son|1|1|
|Lord Jim|2|2|
|David Copperfield|3|1|
|-----|-----|
SQL>
```

where we see that the undefined objects #8, #19 and domains %0 to %2 are no longer referenced. To save space we will omit unreferenced items from such listings below.

Now 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
```

After parsing the first TableReferenceItem (AUTHOR A) (line 5984 again) we have

```
{(23=Table Name=AUTHOR 23 TABLE Definer=-502 Ppos=23:-538 Indexes:((48)70) KeyCols: (48=True),
 34=Domain 34 INTEGER,
 48=TableColumn 48 Domain 34 Definer=-502 Ppos=48 34 Table=23,
 91=Domain 91 CHAR,
 104=TableColumn 104 Domain 91 Definer=-502 Ppos=104 91 Table=23,
 #8=SqlCopy Name=ANAME #8 Domain 91 From:#32 copy from 104,
 #14=ForwardReference Name=B #14 Domain %1 Definer=-501 Ppos=#14,
 #16=SqlValue Name=TITLE #16 CONTENT From:#1 Alias=C,
 #32=From #32:%4 targets: 23=%6 Source: %6 Alias A Target=23 Indexes=[(%3)70],
 %0=Domain %0 ROW (#8)[#8,CONTENT],
 %1=Domain %1 ROW (#16)[#16,CONTENT],
 %2=Domain %2 TABLE (#8,#16)[#8,Domain 91 CHAR],[#16,CONTENT],
```

<sup>32</sup> Domains %0 and %1 are the provisional domains of unknown objects BOOK and B, %2 is the provisional domain of the query.

```
%3=SqlCopy Name=ID %3 Domain 34 From:#32 copy from 48,
%4=Domain %4 TABLE (#8|%3) Display=1[#8,Domain 91 CHAR],[%3,Domain 34 INTEGER],
%5=Domain %5 TABLE (%3,#8)[%3,Domain 34 INTEGER],[#8,Domain 91 CHAR],
%6=TableRowSet %6:%5 From: #32 SRow:(48,104) Target:23 Indexes: [(%3)=70]}}
```

This is much as above, and B and TITLE are unknown at this stage. After the second table reference item (line 5971 again),

```
{(.
  148=Table Name=BOOK 148 TABLE Definer=-502 Ppos=148:-538 Indexes:((158)182,(204)230)
    KeyCols: (158=True,204=True),
  158=TableColumn 158 Domain 34 Definer=-502 Ppos=158 34 Table=148,
  204=TableColumn 204 Domain 34 Definer=-502 Ppos=204 34 Table=148,
  250=TableColumn 250 Domain 91 Definer=-502 Ppos=250 91 Table=148,
  ..
  #42=From #42:%9 targets: 148=%11 Source: %11 Alias B Target=148 Indexes=[(%7)182,(%8)230],
  ..
  %7=SqlCopy Name=ID %7 Domain 34 From:#42 copy from 158,
  %8=SqlCopy Name=AID %8 Domain 34 From:#42 copy from 204,
  %9=Domain %9 TABLE (#16|%7,%8) Display=1[#16,Domain 91 CHAR],[%7,Domain 34 INTEGER],
    [%8,Domain 34 INTEGER],
  %10=Domain %10 TABLE (%7,%8,#16)[%7,Domain 34 INTEGER],[%8,Domain 34 INTEGER],
    [%16,Domain 91 CHAR],
  %11=TableRowSet %11:%10 From: #42 SRow:(158,204,250) Target:148
    Indexes: [(%7)=182,(%8)=230]}}}
```

Note that (of course) we have different uids for the columns called ID in the different tables. Domains %9 and %10 note uids for two columns that will not be displayed in the results and so have not yet been mentioned in the query. %10's rowType matches the column ordering in the base table. Near the end of ParseFromClause (line 5785), we have replaced %3 and %7 to display them better and added two more entries

```
{(.
  #40=JoinRowSet #40:%12 targets: 23=%6,148=%11 CROSS First: #32 Second: #42,
  ..
  %3=SqlCopy Name=A.ID %3 Domain 34 From:#32 copy from 48,
  ..
  %7=SqlCopy Name=B.ID %7 Domain 34 From:#42 copy from 158,
  %12=Domain %12 TABLE (#8,#16|%3,%7,%8) Display=2[#8,Domain 91 CHAR],[#16,Domain 91 CHAR],
    [%3,Domain 34 INTEGER],[%7,Domain 34 INTEGER],[%8,Domain 34 INTEGER]}}
```

After the where conditions have been parsed (line 5663), we also have

```
#59=SqlValueExpr Name= #59 BOOLEAN Left:%3 Right:%8 #59(%3=%8),
```

and now at line 5664, the rowSet review process takes place (see 1.14). RowSet.New, called for this where-condition, calls RowSet.Apply on JoinRowSet #40 (line 283 of RowSet.cs). JoinRowSet has its own override for the Apply method (line 473 of Join.cs). The where-condition becomes an on-condition (line 552) for an inner join, and at line 574 converts the join to INNER, and applies where conditions and appropriate order specifications to the operands #32 and #42. In this case, these orderSpecifications are (%3) and (%8) and are handled by the indexes on the TableRowSets #32 and #42, so that by the time traversal starts (Start.cs line 426) we have the following new or changed entries:

```
#1=SelectRowSet #1:%2 targets: 23=%6,148=%11 Source: #40,
%6=TableRowSet %6:%5 key (%3) order (%3) From: #32 SRow:(48,104) Target:23 Indexes: [(%3)=70],
..
%11=TableRowSet %11:%10 key (%8) order (%8) From: #42 SRow:(158,204,250) Target:148
  Indexes: [(%7)=182,(%8)=230],
%13=SelectStatement %14 Union=#1}}
```

```
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>
```

These changes result in an efficient computation of the Join. Future refinement of the rowSet review process might remove the two From rowsets #32 and #42 from the pipeline in this case, but the domains of #32 and %6 (that is, domains %4 and %5) do not exactly match.



## 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 immutable 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
<code>SystemRowSet</code>	System Tables as defined in Manual sec 8. Many <code>Cursor</code> types
<code>TableRowSet</code>	Targets a base 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
<code>From</code>	<code>TrivialRowSet</code>	Static results: <code>rowType</code> from command
<code>From</code>	<code>SelectedRowSet</code>	Accesses base table:
<code>JoinPart</code>	<code>JoinRowSet</code>	Cursor classes for different join types
<code>QueryExpression</code>	<code>MergeRowSet</code>	
<code>QuerySearch</code>	<code>TransitionRowSet</code>	For Delete. <code>TransitionRowSet</code> has defpos #0
<code>QuerySpecification</code>	<code>SelectRowSet</code>	
<code>SqlInsert</code>	<code>TransitionRowSet</code>	<code>TransitionRowSet</code> has defpos #0
<code>TableExpression</code>	<code>TableExpRowSet</code>	
<code>UpdateSearch</code>	<code>TransitionRowSet</code>	<code>TransitionRowSet</code> has defpos #0
<code>VALUES</code>	<code>ExplicitRowSet</code>	

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  $O(N\log N + M\log M)$  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 provide 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 Name=A 23 TABLE Definer=-502 Ppos=23:-538,
 29=Domain 29 INTEGER,
 43=TableColumn 43 Domain 29 Definer=-502 Ppos=43 29 Table=23,
 64=Domain 64 CHAR,
 77=TableColumn 77 Domain 64 Definer=-502 Ppos=77 64 Table=23,
 #1=SqlInsert #1 Target: #13 Value: %4 Columns: [77],
 #13=From #13:%1 targets: 23=%3 Source: %3 Target=23,
 #15=SqlCopy Name=C #15 Domain 64 From:#13 copy from 77,
 #18=#25,
 #25=SqlRow #25 Domain %5,
 #26=Three,
 %0=SqlCopy Name=B %0 Domain 29 From:#13 copy from 43,
 %1=Domain %1 TABLE (#15|%0) Display=1[#15,Domain 64 CHAR],[%0,Domain 29 INTEGER],
 %2=Domain %2 TABLE (%0,#15)[%0,Domain 29 INTEGER],[#15,Domain 64 CHAR],
 %3=TableRowSet %3:%2 From: #13 SRow:(43,77) Target:23,
 %4=SqlRowSet %4:%7 targets: 23=%3 SqlRows [#25],
 %5=Domain %5 ROW (#26)[#26,Domain 64 CHAR],
```

```
%6=Domain %6 TABLE (#15|%0) Display=1[#15,Domain 64 CHAR],[%0,Domain 29 INTEGER],
%7=Domain %7 TABLE (#15) Display=1[#15,Domain 64 CHAR],[%0,Domain 29 INTEGER]]}
```

The Execute method for the SqlInsert 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 308) to #1.Obey() (line 3175 of Executable.cs) for the target From #13 and the data SqlRowSet %4. The From #13 has one base table target 23 and a corresponding TableRowSet %3.

SqlInsert.Obey() looks at the targets of the From (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 to control 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 289) the TransitionRowSet object for the single target 23:

```
%8=TransitionRowSet %8:%2 targets: 23=%3 Data: %4 Target: 23
```

Back in SqlInsert.Obey(), (line 3183 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 `{(#15=Three) %4}`, and for each TargetActivation, calls EachRow() at line 3190. In this case, this takes us to Activation.cs, line 517, where `trc` is `{(%0= Null,#15=Three) %8}`, `trc._tgc` is `{(43= Null,77=Three) %8}`, and `tgc._rec` is a Record with these values. `newRow` is set to these values, and as there are no triggers in this example, a new Record `{Record !0[23]: 77=Three}` is made from `newRow` at line 531, which will be committed to the database.

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 %8 for the two cursors above.

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

## 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 file. 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
1234567890123456789012345678901234567890123456789012345
where country = 'GER' group by city having count(*) > 4;
```

At the start of the Build method for the SelectRowSet result (line 3148 in RowSet.cs), the context contains

```
{(-538=TABLE,
  23=Table Name=PEOPLE 23 TABLE Definer=-502 Ppos=23:-538 Indexes:((49)207) KeyCols: (49=True),
  34=Domain 34 INTEGER Prec=11,
  49=TableColumn 49 Domain 34 Definer=-502 Ppos=49 34 Table=23 Not Null,
  72=Domain 72 CHAR Prec=50,
  86=TableColumn 86 Domain 72 Definer=-502 Ppos=86 72 Table=23,
  111=Domain 111 NUMERIC Prec=10 Scale=2,
  127=TableColumn 127 Domain 111 Definer=-502 Ppos=127 111 Table=23,
  154=TableColumn 154 Domain 72 Definer=-502 Ppos=154 72 Table=23,
  182=TableColumn 182 Domain 72 Definer=-502 Ppos=182 72 Table=23,
  #1=SelectRowSet #1:%13 where (#75) groupSpec: #89 groupings (#92) having (%10) GroupCols(#8)
    targets: 23=%8 Source: #54,
    #8=SqlCopy Name=CITY #8 Domain 72 From:#54 copy from 182,
    #14=SqlFunction Name=AVG #14 Domain %2 From:#1 AVG(#18),
    #18=SqlCopy Name=SALARY #18 Domain 111 From:#54 copy from 127,
    #27=SqlFunction Name=COUNT #27 Domain %3 From:#1 Alias=NUMPEOPLE Await (#1) COUNT(#33)
      as NUMPEOPLE,
    #33=1,
    #54=From #54:%6 matches (%5=GER) targets: 23=%8 Source: %8 Target=23 Indexes=[(%3)207],
    #75=SqlValueExpr Name= #75 BOOLEAN Left:%7 Right:#77 #75(%7=#77),
    #77=GER,
    #89=GroupSpecification #89(#92),
    #92=Grouping #92 Domain %15 GROUP (#8=0),
    #104=SqlFunction Name=COUNT #104 Domain %11 COUNT(#110),
    #110=1,
    #113=SqlValueExpr Name= #113 Domain %12 Left:#104 Right:#115 #113(#104>#115),
    #115=4,
    %0=Domain %0 ROW (#8)[#8,CONTENT],
    %1=Domain %1 ROW (#18)[#18,CONTENT],
    %2=Domain %2 TABLE (#8,#14,#27)[#8,Domain 72 CHAR Prec=50],[#14,NUMERIC],[#27,INTEGER]
      Aggs (#14,#27),
    %3=SqlCopy Name=ID %3 Domain 34 From:#54 copy from 49,
    %4=SqlCopy Name=NAME %4 Domain 72 From:#54 copy from 86,
    %5=SqlCopy Name=COUNTRY %5 Domain 72 From:#54 copy from 154,
    %6=Domain %6 TABLE (#8,#18|%3,%4,%5) Display=2
      [#8,Domain 72 CHAR Prec=50],[#18,Domain 111 NUMERIC Prec=7 Scale=2],
      [%3,Domain 34 INTEGER Prec=11],[#4,Domain 72 CHAR Prec=50],[#5,Domain 72 CHAR Prec=50],
    %7=Domain %7 TABLE (%3,%4,#18,%5,#8)
      [%3,Domain 34 INTEGER Prec=11],[#4,Domain 72 CHAR Prec=50],
      [#18,Domain 111 NUMERIC Prec=7 Scale=2],[#5,Domain 72 CHAR Prec=50],
      [#8,Domain 72 CHAR Prec=50],
    %8=TableRowSet %8:%7 key (49) where (#75) matches (%5=GER) From: #54 SRow:(49,86,127,154,182)
      Target:23 Indexes: [(%3)=207],
    %9=Domain %9 BOOLEAN Aggs (#104),
    %10=SqlValueExpr Name= %10 Domain %11 Left:#27 Right:#115 Await (#1) %10(#27>#115),
    %11=Domain %11 BOOLEAN Aggs (#27),
    %12=Domain %12 ROW (#8)[#8,Domain 72 CHAR Prec=50],
    %13=Domain %13 TABLE (#8,#14,#27)
      [#8,Domain 72 CHAR Prec=50],[#14,NUMERIC],[#27,INTEGER] Aggs (#14,#27),
```

```
%14= SelectStatement %16 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 3176-3215) 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

```
{(#8=Berlin,#18=50000,%3=61,%4=Tom,%5=GER) #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 3212. The structure used by the Context is called funcns and is indexed by the aggregation function uid, the grouping key, and the SelectRowSet uid.

The second loop (lines 3219-3249) traverses the cx.funcns register collection and for each grouping key that was found, places a row in the SelectRowSet value 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, together with the result of an expanded query that does the same thing:

```
[ select city, avgSalary, numPeople from
  (select city, avg(Salary) as avgSalary, count(*) as numPeople from
    (select * from people where country = 'GER') GERpeople
   group by city) GERgrouped where numPeople > 4; ]
```

```
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|
|-----|-----|-----|
SQL> [ select city, avgSalary, numPeople from
> (select city, avg(Salary) as avgSalary, count(*) as numPeople from
> (select * from people where country = 'GER') GERpeople
> group by city) GERgrouped
> where numPeople > 4; ]
-----|-----|-----|
|CITY|AVGSALARY|NUMPEOPLE|
|-----|-----|-----|
|Berlin|60333.3333333333|6|
|Munich|61833.3333333333|6|
|-----|-----|-----|
SQL>
```

Parsing this more complicated query results in the same rowset pipeline as was shown above.

## 6.2.3 Views

In this version of Pyrrho, Views are compiled objects in the sense that on definition the given SQL definition of the View 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 code modules (triggers, stored procedures etc) are only parsed once per cold start. The uids given to DBOObjects generated by the parser during Load are all in the executable range. These generated objects are not added to the Database objects tree, but stored (in Framing) in the trigger or procedure DBOObject. 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 to the execution engine. (Views need to be instantiated so the process is a little more complex, see section 6.6.)

A particular 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
123456789012345678901234567890123456789012
create table ca(a char,b int check (b>0))

```

During ParseColumnCheckConstraint, `cx.parse` is set to `Compile`.

Then during the construction of the Physical PCheck record, (PCheck.cs, line 52) the context has:

```

{(!0=Table Name=CA !0 TABLE Definer=-502 Ppos=!0:-538,
 !1=TableColumn !1 CHAR Definer=-502 Ppos=!1 -508 Table=!0,
 !2=TableColumn !2 INTEGER Definer=-502 Ppos=!2 -511 Table=!0,
 `4=SqlCopy Name=B `4 INTEGER copy from !2,
 `6=SqlValueExpr Name= `6 BOOLEAN Left:`4 Right:`9 `6(`4>`9),
 `9=0)}

```

We see the proposed table and its columns, and the parsed version of the check expression with uids `4, `6 and `9 in the executable range. PCheck is a subclass of Compiled, so it already has a framing field containing these:

```

{Framing (`4 SqlCopy Name=B `4 INTEGER copy from !2,
 `6 SqlValueExpr Name= `6 BOOLEAN Left:`4 Right:`9 `6(`4>`9),
 `9 0)}

```

When we construct the Check object (in PCheck2.Install, line 238 of PCheck.cs) we get a Check object with the above framing:

```
{Check Name= !3 Definer=-502 Ppos=!3 From.Target=!2 Source=(b>0) Search=`6}
```

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 241) adds the Check constraint !3 to the transaction, and this changes column !2 in the context to

```
{TableColumn !2 INTEGER Definer=-502 Ppos=!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
456789012345678901234567890123456
insert into ca values('Neg',-99)
```

The Check Constraint exception is raised when the TransitionRowSet constructs the intended record. From the last example, 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 5617 of RowSet.cs). At this point we have the following in the context (lexical positions currently run on during a transaction):

```
{(!0=Table Name=CA !0 TABLE Definer=-502 Ppos=!0:-538,
 !1=TableColumn !1 CHAR Definer=-502 Ppos=!1 -508 Table=!0,
 !2=TableColumn !2 INTEGER Definer=-502 Ppos=!2 -511 Table=!0 Checks:(!3=True),
 #44=SqlInsert #44 Target: #56 Value: %5 Columns: [!1,!2],
 #56=From #56:%2 targets: !0=%4 Source: %4 Target=!0,
 #59=#65,
 #65=SqlRow #65 Domain %6,
 #66=Neg,
 #72=SqlValueExpr Name= #72 UNION Right:#73 #72(-#73),
 #73=99,
 4=SqlCopy Name=B `4 INTEGER copy from !2,
 6=SqlValueExpr Name= `6 BOOLEAN Left: 4 Right: 9 `6(`4>9),
 `9=0,
 %0=SqlCopy Name=A %0 CHAR From:#56 copy from !1,
 %1=SqlCopy Name=B %1 INTEGER From:#56 copy from !2,
 %2=Domain %2 TABLE (%0,%1) Display=2[%0,CHAR],[%1,INTEGER],
 %3=Domain %3 TABLE (%0,%1)[%0,CHAR],[%1,INTEGER],
 %4=TableRowSet %4:%3 From: #56 SRow:(!1,!2) Target:!0,
 %5=SqlRowSet %5:%8 targets: !0=%4 SqlRows [#65],
 %7=Domain %7 ROW (#66,#72)[#66,CHAR],[#72,Domain UNION],
 %8=Domain %8 TABLE (%0,%1) Display=2[%0,CHAR],[%1,INTEGER],
 %9=TransitionRowSet %9:%3 targets: !0=%4 Data: %5 Target: !0)}
```

Here the objects coming from the check constraint have been coloured blue. (All of the others are part of the normal SqlInsert processing as in previous examples.)

In %8.TargetCursor.New() we find the constraint on the second tablecolumn !2 at line 5666. The next line adds the current values vs = {(!1=Neg,!2=-99)} to the context. The next line retrieves the SqlValueExpr `6 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 or grant of a database object. Definition is within the command processing that creates the object, or the database load on server start-up, and the associated 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 instantiated.

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<sup>33</sup>:

Pos	Record Type	Contents
23	PTable XA	
30	PDomain INTEGER	
44	PColumn3 B	
65	PColumn3 C	
87	PDomain CHAR	
100	PColumn3 D	
140	PTable XB	
147	PColumn3 TOT	
189	Record 189[140]	147=0
221	Trigger RUAB	Trigger RUAB Update, Before, EachRow on 23 begin atomic update xb set tot=tot-mr.b+nr.b; set d='changed' end
331	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<sup>34</sup>) 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 with backquotes<sup>35</sup>:

```

{Framing (
  `64 From `64:`68 targets: 23=`70 Source: `70 Target=23,
  `65 SqlCopy Name=B `65 Domain 30 From:`64 copy from 44,
  `66 SqlCopy Name=C `66 Domain 30 From:`64 copy from 65,
  `67 SqlCopy Name=D `67 Domain 87 From:`64 copy from 100,
  `68 Domain `68 TABLE (`65,`66,`67) Display=3
    [`65,Domain 30 135],[`66,Domain 30 135],[`67,Domain 87 CHAR],
  `69 Domain `69 TABLE (`65,`66,`67)
    [`65,Domain 30 135],[`66,Domain 30 135],[`67,Domain 87 CHAR],
  `70 TableRowSet `70:`69 From: `64 SRow:(44,65,100) Target:23,
  `71 CompoundStatement `71(`72,`92),
  `72 AssignmentStatement `72 `66=`82,
  `82 SqlValueExpr Name= `82 Domain 30 Left:`85 Right:`82(`65+`85),
  `85 3,
  `87 From `87:`89 targets: 140=`91 Source: `91 Target=140,
  `88 SqlCopy Name=TOT `88 Domain 30 From:`87 copy from 147,
  `89 Domain `89 TABLE (`88) Display=1[`88,Domain 30 135],
  `90 Domain `90 TABLE (`88)[`88,Domain 30 135],
  `91 TableRowSet `91:`90 From: `87 Assigs:(UpdateAssignment Vb1: `88 Val: `101=True) SRow:(147)
    Target:140,
  `92 UpdateSearch `92 Target: `87,
  `101 SqlValueExpr Name= `101 Domain 30 Left:`88 Right:`65 `101(`88+`65),
  `108 WhenPart `108 Cond: `109 Stms: (`71)) Result `87}}}}

```

The CompoundStatement `71 (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..

<sup>33</sup> In all these worked examples, the actual file positions depend on many non-reproducible details including timestamps. As they say in car-maintenance manuals, “Your mileage may vary.”

<sup>34</sup> The uids and activation identifiers are different for a newly defined trigger.

<sup>35</sup> In these listings Visual Studio has displayed INTEGER as 135 (which is (int)Sqlx.INTEGER). Why?



The next step in the test is an insert command:

```

      1          2          3          4
12345678901234567890123456789012345678901
insert into xa(b,d) values (7,'inserted')
```

Let us place a break point at the start of Transaction.Execute (line 304 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 context (using the table definer's role) and maybe several different trigger contexts (the trigger definers' roles). After parsing the insert command (at line 304) we have in the main Context:

```
{(23=Table Name=XA 23 TABLE Definer=-502 Ppos=23:-538
  Triggers:(Insert, Before, EachRow=(331=True),
    Update, Before, EachRow=(221=True)),
  30=Domain 30 INTEGER,
  44=TableColumn 44 Domain 30 Definer=-502 Ppos=44 30 Table=23,
  65=TableColumn 65 Domain 30 Definer=-502 Ppos=65 30 Table=23,
  87=Domain 87 CHAR,
  100=TableColumn 100 Domain 87 Definer=-502 Ppos=100 87 Table=23,
  #1=SqlInsert #1 Target: #13 Value: %4 Columns: [44,100],
  #13=From #13:%1 targets: 23=%3 Source: %3 Target=23,
  #16=SqlCopy Name=B #16 Domain 30 From:#13 copy from 44,
  #18=SqlCopy Name=D #18 Domain 87 From:#13 copy from 100,
  #21=#28,
  #28=SqlRow #28 Domain %5,
  #29=7,
  #31=inserted,
  %0=SqlCopy Name=C %0 Domain 30 From:#13 copy from 65,
  %1=Domain %1 TABLE (#16,#18|%0) Display=2
    [#16,Domain 30 INTEGER],[#18,Domain 87 CHAR],[%0,Domain 30 INTEGER],
  %2=Domain %2 TABLE (#16,%0,#18)
    [#16,Domain 30 INTEGER],[%0,Domain 30 INTEGER],[#18,Domain 87 CHAR],
  %3=TableRowSet %3:%2 From: #13 SRow:(44,65,100) Target:23,
  %4=SqlRowSet %4:%7 targets: 23=%3 SqlRows [#28],
  %5=Domain %5 ROW (#29,#31)[#29,Domain 30 INTEGER],[#31,Domain 87 CHAR],
  %6=Domain %6 TABLE (#16,#18|%0) Display=2
    [#16,Domain 30 INTEGER],[#18,Domain 87 CHAR],[%0,Domain 30 INTEGER],
  %7=Domain %7 TABLE (#16,#18) Display=2
    [#16,Domain 30 INTEGER],[#18,Domain 87 CHAR],[%0,Domain 30 INTEGER]}}
```

The command to be executed is the SqlInsert highlighted above. Let us trace what happens next in SqlInsert #1.Obey() (Executable.cs at line 3172). 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 <- {From #13:%1 targets: 23=%3 Source: %3 Target=23}
data <- {SqlRowSet %4:%7 targets: 23=%3 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 3187 or so). Then `ta` is TableActivation 9 and in addition to the above objects `ta.obs` has

```
{TransitionRowSet %8:%2 targets: 23=%3 Data: %4 Target: 23}
```

The TableActivation also has a list of TriggerActivations called `acts`,

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

and `act[0]` is TriggerActivation 10. The TriggerActivation constructor added instances of 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).

The traversal of the `data` TableRowSet %3 has just begun (at line 3183), and the first cursor is the SqlCursor:

```
ib {(#16=7,#18=inserted) %4}
```

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 {(#16=7,%0= Null,#18=inserted) %8}
trc._tgc {(44=7,65= Null,100=inserted) %8}
tgc._rec.vals {(44=7,100=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.

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

At line 1119 of Activation.cs, Exec() begins to execute the CompoundStatement `71 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 `72 `66=`82}. Recall from above,

```
`65 SqlCopy Name=B `65 Domain 30 From:`64 copy from 44,
`66 SqlCopy Name=C `66 Domain 30 From:`64 copy from 65,
`67 SqlCopy Name=D `67 Domain 87 From:`64 copy from 100,
..
`70 TableRowSet `70:`69 From: `64 SRow:(44,65,100) Target:23,
`71 CompoundStatement `71(`72,`92),
`72 AssignmentStatement `72 `66=`82,
`82 SqlValueExpr Name= `82 Domain 30 Left:`65 Right:`85 `82(`65+`85),
`85 3,
```

We see that obs[82] is obs[65]+obs[85]. obs[65] is an SqlCopy to copy the current value for table column 44, while obs[85] 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=(44=7,65= Null,100=inserted) %8,44=7,100=inserted,`65=7,`66=10,`67=inserted)}
```

The next step in the CompoundStatement is {UpdateSearch `92 Target: `87}. UpdateSearch.Obey() calls `91.Update(), which creates a new TableActivation 12. Again, when using the debugger, it is easiest to step over to line 3362 to look at ta. This has just one further object

```
{TransitionRowSet `9:`90 targets: 140=`91 Data: `87 Target: 140}
```

but no triggers, as XB has none defined. The traversal cursor ib is {(88=0) `87} and ta.EachRow constructs

```
trc {(88=0) %9}
tgc {(147=0) %9}
rc.vals {(147=0)}
```

The updates in EachRow (at line 557-560) now construct the new values list with newRow {(147=7)}.

At line 587, the transition and target cursors for %9 are updated for the new values, and the new targetCursor {(147=7) %9} is installed in activation 12 at line 589.

At line 590 the cursors for each of the current activations are

```
12: {(140=(88=0) `91,
      `87=(88=0) `87,
      `91=(88=0) `87,
      %3=(#16=7,#18=inserted) %4,
      %4=(#16=7,#18=inserted) %4,
      %8=(59=7,60= Null,61=inserted) %8,
      %9=(147=7) %9)}
11: {(140=(88=0) `91,
      `87=(88=0) `87,
      `91=(88=0) `91,
      %3=(#16=7,#18=inserted) %4,
      %4=(#16=7,#18=inserted) %4,
      %8=(65=7,66= Null,67=inserted) %8,
      %9=(88=0) %9)}
10: {(%3=(#16=7,#18=inserted) %4,
      %4=(#16=7,#18=inserted) %4,
      %8=(65=7,66= Null,67=inserted) %8)}
9:  {(%3=(#16=7,#18=inserted) %4,
      %4=(#16=7,#18=inserted) %4,
      %8=(44=7,65= Null,100=inserted) %8)}
```

The new Update gets constructed (at line 606) as `!1 {Update 189[140]: 147=7 Prev:189}`. 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 ``92.Obey`, where, at line 3363, 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 3371 (it is retained for further rows, and there aren't any). The modified activation 11 is returned to ``71.ObeyList()`. As there are no more statements in the CompoundStatement, this also returns.

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

```
10: {(23=(44=7,65= Null,100=inserted) %8,
    44=7,
    100=inserted,
    140=(147=7) %9,
    147=7,
    `65=7,
    `66=10,
    `67=inserted,`88=0)}
```

We are then back in `10.Exec()`. At line 1083, we transmit the triggered changes in the row values to update the transitionCursor for %6.

The new Physicals `{(!0=TriggeredAction 331,!1=Update 189[140]: 147=7 Prev:189)}` replaces the corresponding entries in the caller's context 3. In particular, the result of the highlighted assignment above gets placed in the newRow for table XA at line 1075, so that the details of these cursors for table XA are `{(44=7,65=10,100=inserted) %1}` and `{(#16=7,#18=inserted,%0=10) %1}` respectively.

This completes execution of each-row-before triggers, and now the insert of the new row can take place, back in `9.EachRow()`. At line 538 this gives a new Record for the transaction: `{Record !2[23]: 44=7,65=10,100=inserted}`.

This means the transaction now has the following Physical objects to commit:

```
{(!0=TriggeredAction 331,
 !1=Update 189[140]: 147=7 Prev:189,
 !2=Record !2[23]: 44=7,65=10,100=inserted)}
```

The command is finished, and the following records are committed along with the triggeredAction note at position 421 in the transaction log:

428	Update 189[140]	147=7 Prev:189
449	Record 449[23]:	44=7,65=10,100=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:

505	Update 189[140]	147=16 Prev:428
526	Record 526[23]:	44=9,65=12,100=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
12345678901234567890123456789012345678901
update xa set b=8,d='updated' where b=7
```

Up as far as at TableActivation `9.EachRow()`, the execution proceeds similarly to the above example. At this point we construct

```
trc {(%0=7,%1=10,%2=inserted) %7}
tgc {(44=7,65=10,100=inserted) %7}
rc.vals {(44=7,65=10,100=inserted)}
```

and Step A of the update processing (line 563 of Activation.cs) sets up the oldrow (-295) and newrow (-293) values in the TableActivation 9. At line 567 we have

```
9.cursors {(23=(%0=7,%1=10,%2=inserted) %5,
           #8=(%0=7,%1=10,%2=inserted) #8,
           %5=(%0=7,%1=10,%2=inserted) #8,
           %7=(44=7,65=10,100=inserted) %7)}

9.values   {(-295=(44=7,65=10,100=inserted) %7,
            -293=(44=8,65=10,100=updated),
            23=(44=7,65=10,100=inserted) %7,
            44=8,
            65=10,
            100=updated)}
```

At line 567, we call the triggers. When TriggerActivation 10.Exec() is called, this is how things stand:

```
10.obs
{(23=Table Name=XA 23 TABLE Definer=-502 Ppos=23:-538
  Triggers:(Insert, Before, EachRow=(331=True),Update, Before, EachRow=(221=True)),
  30=Domain 30 INTEGER,
  44=TableColumn 44 Domain 30 Definer=-502 Ppos=44 30 Table=23,
  65=TableColumn 65 Domain 30 Definer=-502 Ppos=65 30 Table=23,
  87=Domain 87 CHAR,
  100=TableColumn 100 Domain 87 Definer=-502 Ppos=100 87 Table=23,
  221=Trigger Name=RUAB 221 Domain `10 Definer=-502 Ppos=221 TrigType=Update, Before, EachRow
    On=23,
  #8=From #8:%3 matches (%0=7) targets: 23=%5 Source: %5 Target=23,
  #17=8,
  #21=updated,
  #38=SqlValueExpr Name= #38 BOOLEAN Left:%0 Right:#39 #38(%0=#39),
  #39=7,
  `0=SqlOldRow Name=MR `0 Domain `10 From:`6,
  `1=SqlNewRow Name=NR `1 Domain `10 From:`6,
  `6=From `6:`10 targets: 23=`12 Source: `12 Target=23,
  `7=SqlCopy Name=B `7 Domain 30 From:`6 copy from 44,
  `8=SqlCopy Name=C `8 Domain 30 From:`6 copy from 65,
  `9=SqlCopy Name=D `9 Domain 87 From:`6 copy from 100,
  `10=Domain `10 TABLE (`7,`8,`9) Display=3
    [ `7,Domain 30 INTEGER],[ `8,Domain 30 INTEGER],[ `9,Domain 87 CHAR],
  `11=Domain `11 TABLE (`7,`8,`9)
    [ `7,Domain 30 INTEGER],[ `8,Domain 30 INTEGER],[ `9,Domain 87 CHAR],
  `12=TableRowSet `12:`11 From: `6 SRow:(44,65,100) Target:23,
  `13=SqlField Name=B `13 Domain 30 Parent=`0 Field=44,
  `14=SqlField Name=C `14 Domain 30 Parent=`0 Field=65,
  `15=SqlField Name=D `15 Domain 87 Parent=`0 Field=100,
  `16=SqlField Name=B `16 Domain 30 Parent=`1 Field=44,
  `17=SqlField Name=C `17 Domain 30 Parent=`1 Field=65,
  `18=SqlField Name=D `18 Domain 87 Parent=`1 Field=100,
  `19=CompoundStatement `19(`25,`50),
  `20=From `20:`22 targets: 140=`24 Source: `24 Target=140,
  `21=SqlCopy Name=TOT `21 Domain 30 From:`20 copy from 147,
  `22=Domain `22 TABLE (`21) Display=1[ `21,Domain 30 INTEGER],
  `23=Domain `23 TABLE (`21)[ `21,Domain 30 INTEGER],
  `24=TableRowSet `24:`23 From: `20 Assigs:(UpdateAssignment Vb1: `21 Val: `42=True)
    SRow:(147) Target:140,
  `25=UpdateSearch `25 Target: `20,
  `34=SqlValueExpr Name= `34 Domain 30 Left:`21 Right:`13 `34(`21-`13),
  `42=SqlValueExpr Name= `42 Domain 30 Left:`34 Right:`16 `42(`34+`16),
  `50=AssignmentStatement `50 `9=`56,
  `56=changed,
  `58=WhenPart `58 Cond: `59 Stms: (`19),
  %0=SqlCopy Name=B %0 Domain 30 From:#8 copy from 44,
  %1=SqlCopy Name=C %1 Domain 30 From:#8 copy from 65,
  %2=SqlCopy Name=D %2 Domain 87 From:#8 copy from 100,
  %3=Domain %3 TABLE (%0,%1,%2) Display=3
    [%0,Domain 30 INTEGER],[%1,Domain 30 INTEGER],[%2,Domain 87 CHAR],
  %4=Domain %4 TABLE (%0,%1,%2)
    [%0,Domain 30 INTEGER],[%1,Domain 30 INTEGER],[%2,Domain 87 CHAR],
  %5=TableRowSet %5:%4 where (#38) matches (%0=7) From: #8
    Assigs:(UpdateAssignment Vb1: %0 Val: #17=True,
            UpdateAssignment Vb1: %2 Val: #21=True)
    SRow:(44,65,100) Target:23,
  %6=UpdateSearch %6 Target: #8,
  %7=TransitionRowSet %7:%4 where (#38) targets: 23=%6 Data: #8 Target: 23)}
```

```
10.cursors {(23=(%0=7,%1=10,%2=inserted) %5,
            #8=(%0=7,%1=10,%2=inserted) #8,
            %5=(%0=7,%1=10,%2=inserted) #8,
            %7=(44=7,65=10,100=inserted) %7)}
```

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 11111, its values are

```
10.values {(-295=(44=7,65=10,100=inserted) %7,
            -293=(44=8,65=10,100=updated),
            23=(`7=7,`8=10,`9=inserted) %7,
            44=8,65=10,100=updated,
            `0=(44=7,65=10,100=inserted) %7,
            `1=(44=8,65=10,100=updated),
            `7=7,
            `8=10,
            `9=inserted)}
```

At line 1119 of Activation.cs, the TriggerActivation starts to Obey the CompoundStatement `19.

The first step is the UpdateSearch `25, and it calls the Update method on TableRowSet `24 targeting XB. It creates TableActivation 12 with its TransitionRowSet

```
{TransitionRowSet %8:`23 targets: 140=`24 Data: `20 Target: 140}
```

The traversal begins, and 12.EachRow creates cursors:

```
trc {( `22=16) %8}
tgc {(147=16) %8}
rc.vals {(147=16)}
```

Evaluating the update `24's UpdateAssignment highlighted above involves using RUAB's old and new rows values `0 and `1 above. The evaluation proceeds as follows:

```
`21 := `42
      = ( `34      + `16)
      = (( `21 - `13) + `16)
      = (((147)-[ `0].44)+([ `1].44)
      = ((16 - 7) + 8)
[147] := 17
```

So in 12.values, we get [147]:=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 [140] in the transaction physicals: {(1=Update 189[140]: 147=17 Prev:505)}. The transitionCursor for 12 is also updated (in case there are further cascading changes). and also installed in TableActivation 12. This completes the UpdateSearch `25.

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=(147=17) %8)}
```

The second step in the trigger is an Assignment statement in Activation 11. This updates 11's values for `9 to changed.

```
11.values {(-295=(147=16) %8, -293=(147=17), 23=(44=7,65=10,100=inserted)
            %7,44=8,65=10,100=updated,140=(147=17) %8,147=17,`0=(44=7,65=10,100=inserted)
            %7,`1=(44=8,65=10,100=updated),`7=8,`8=10,`9=updated)}
```

Back in 10.EachRow(), it remains to merge the updates into the new TableRow. This generates a further item for the Transaction Commit: {(12=Update 449[23]: 44=8,65=10,100=changed Prev:449)}

In the transaction log, this shows as

578	Update 189[140]	147=17 Prev:505
599	Update 449[23]:	44=8,65=10,100=changed Prev: 449

This completes the discussion of the Update trigger demonstration.

In the test a third table XC and a third trigger for XA are defined<sup>36</sup>, 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)
```

```
[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]
```

648	PTable XC	
656	PColumn3 TOTB	for 648(0)[30]
682	PColumn3 TOTC	for 648(1)[30]
727	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 727 is as follows:

```
{Framing (
`110 TransitionTable `110:`118 Target=23 old from 23,
`114 From `114:`118 targets: 23=`120 Source: `120 Target=23,
`115 SqlCopy Name=B `115 Domain 30 From:`114 copy from 44,
`116 SqlCopy Name=C `116 Domain 30 From:`114 copy from 65,
`117 SqlCopy Name=D `117 Domain 87 From:`114 copy from 100,
`118 Domain TABLE (`115,`116,`117) Display=3
[ `115,Domain 135],[ `116,Domain 135],[ `117,Domain CHAR],
`119 Domain TABLE (`115,`116,`117)[ `115,Domain 135],[ `116,Domain 135],[ `117,Domain CHAR],
`120 TableRowSet `120:`119 From: `114 SRow:(44,65,100) Target:23,
`121 CompoundStatement `121(`122),
`122 SqlInsert `122 Target: `123 Value: `153 Columns: [656,682],
`123 From `123:`126 targets: 648=`128 Source: `128 Target=648,
`124 SqlCopy Name=TOTB `124 Domain 30 From:`123 copy from 656,
`125 SqlCopy Name=TOTC `125 Domain 30 From:`123 copy from 682,
`126 Domain TABLE (`124,`125) Display=2[ `124,Domain 135],[ `125,Domain 135],
`127 Domain TABLE (`124,`125)[ `124,Domain 135],[ `125,Domain 135],
`128 TableRowSet `128:`127 From: `123 SRow:(656,682) Target:648,
`131 SelectRowSet `131:`146 Source: `110,
`137 Domain ROW (`115)[ `115,Domain 135],
`144 Domain ROW (`116)[ `116,Domain 135],
`146 Domain TABLE (`115,`116)[ `115,Domain 135],[ `116,Domain 135],
`150 SelectStatement `150 Union=`131,
`151 SqlValueSelect `151 Domain `146 (`131),
`152 Domain TABLE (`124,`125)[ `115,Domain 135],[ `116,Domain 135],[ `124,Domain 135],
[ `125,Domain 135],
`153 SelectedRowSet `153:`152 targets: 648=`128 Source: `131,
`154 WhenPart `154 Cond: `155 Stms: (`121)
) Result `153}
```

We see the **old** table reference has defined a TransitionTable `110 and a TableRowSet `120, along with SqlCopy for the columns. The test run is:

```
1          2          3
12345678901234567890123456789012
delete from xa where d='changed'
```

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

```
{(23=Table Name=XA 23 TABLE Definer=-502 Ppos=23:-538
  Triggers:(Insert, Before, EachRow=(331=True),
            Update, Before, EachRow=(221=True),
            Delete, Instead, EachStatement=(727=True)),
  30=Domain INTEGER,
  44=TableColumn 44 Domain 30 Definer=-502 Ppos=44 30 Table=23,
  65=TableColumn 65 Domain 30 Definer=-502 Ppos=65 30 Table=23,
  87=Domain CHAR,
  100=TableColumn 100 Domain 87 Definer=-502 Ppos=100 87 Table=23,
  #1=QuerySearch #1 Target: #13,
  #13=From #13:%3 matches (%2=changed) targets: 23=%5 Source: %5 Target=23,
  #23=SqlValueExpr Name= #23 BOOLEAN Left:%2 Right:#24 #23(%2=#24),
  #24=changed,
  %0=SqlCopy Name=B %0 Domain 30 From:#13 copy from 44,
  %1=SqlCopy Name=C %1 Domain 30 From:#13 copy from 65,
  %2=SqlCopy Name=D %2 Domain 87 From:#13 copy from 100,
```

<sup>36</sup> The latest version of the test has two *instead of* triggers. This is simplified here for brevity.

```
%3=Domain TABLE (%0,%1,%2) Display=3[%0,Domain INTEGER],[%1,Domain INTEGER],[%2,Domain
CHAR],
%4=Domain TABLE (%0,%1,%2)[%0,Domain INTEGER],[%1,Domain INTEGER],[%2,Domain CHAR],
%5=TableRowSet %5:%4 where (#23) matches (%2=changed) From: #13 SRow:(44,65,100) Target:23}}
```

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 727 is an *instead of* trigger, so is called on the second invocation of Triggers() at line 495. 10's TransitionRowSet is

```
{TransitionRowSet %7:%4 where (#23) targets: 23=%5 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). 11.Exec(), as before, sets up tables for old and new if requested. In this case, TransitionTableRowSet `110 is for the old table that has been referenced<sup>37</sup>, and replaces(!) the TransitionTable `110 in this activation (line 1101):

```
{TransitionTableRowSet `110:`118 targets: 23=%5 OLD}
```

At line 1119, 11.Exec() begins to obey the trigger body in Activation 12, CompoundStatement `121. This is just SqlInsert `122, and `122.Obey() retrieves the target and data for the operation:

```
tg {From `123:`126 targets: 648=`128 Source: `128 Target=648}
data {SelectedRowSet `153:`152 targets: 648=`128 Source: `131}
```

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 SelectedCursor {(`124=8, `125=10) `153}, and that TableActivation 13 has the additional object

```
{TransitionRowSet %8:`127 targets: 648=`128 Data: `153 Target: 648}
```

13.EachRow creates the transition cursors:

```
trc {(`124=8, `125=10) %8}
tgc {(656=8,682=10) %8}
rc.vals {(656=8,682=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[648]: 656=8,682=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

829	Record 829[648]	656=8,682=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.

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	Domain INTEGER	

<sup>37</sup> For a description of the transition table, see the SQL standard ISO 9075-02(2016) section 4.44.



43	PColumn3 Q	for 23(0)[29]
64	PIndex P	on 23(43) PrimaryKey
80	Domain CHAR	
93	PColumn3 R	for 23(1)[80]
115	PColumn3 A	for 23(2)[29]
155	PView V 155	select q,r as s,a from p

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

```
{Framing (
  `0 Domain ROW (43)[43,Domain 135],
  `3 SelectRowSet `3:`26 targets: 23=`31 Source: `28,
  `6 SqlCopy Name=Q `6 Domain 29 From:`28 copy from 43,
  `9 Domain ROW (`6)[`6,CONTENT],
  `13 SqlCopy Name=R `13 Domain 80 From:`28 Alias=S copy from 93,
  `16 Domain ROW (`13)[`13,CONTENT],
  `21 SqlCopy Name=A `21 Domain 29 From:`28 copy from 115,
  `24 Domain ROW (`21)[`21,CONTENT],
  `26 Domain TABLE (`6,`13,`21)[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
  `28 From `28:`29 targets: 23=`31 Source: `31 Target=23 Indexes=[(`6)64],
  `29 Domain TABLE (`6,`13,`21) Display=3[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
  `30 Domain TABLE (`6,`13,`21)[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
  `31 TableRowSet `31:`30 From: `28 SRow:(43,93,115) Target:23 Indexes: [(`6)=64],
  `33 SelectStatement `33 Union=`3
) Result `3}
```

The greyed-out entries are left over from parsing and no longer referenced. 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 context has the following objects and rowsets and the View has been “instanced” giving the following entries in the Context:

```
{(-539=ROW,
  -538=TABLE,
  -505=CONTENT,
  -503=NULL,
  29=Domain INTEGER,
  80=Domain CHAR,
  155=View Name=V 155 Domain `26 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p
    Ppos: 155 Result `3,
  #1=SqlInsert #1 Target: #13 Value: %37 Columns: [93],
  #13=From #13:%35 targets: 23=%32 Source: %29 Indexes=[(`6)64],
  #15=SqlCopy Name=S #15 Domain 80 From:#13 copy from 93,
  #18=#24,#35,
  #24=SqlRow #24 Domain %38,
  #25=Twenty,
  #35=SqlRow #35 Domain %39,
  #36=Thirty,
  `0=Domain ROW (43)[43,Domain INTEGER],
  `3=SelectRowSet `3:`26 targets: 23=`31 Source: `28,
  `6=SqlCopy Name=Q `6 Domain 29 From:`28 copy from 43,
  `9=Domain ROW (`6)[`6,CONTENT],
  `13=SqlCopy Name=R `13 Domain 80 From:`28 Alias=S copy from 93,
  `16=Domain ROW (`13)[`13,CONTENT],
  `21=SqlCopy Name=A `21 Domain 29 From:`28 copy from 115,
  `24=Domain ROW (`21)[`21,CONTENT],
  `26=Domain TABLE (`6,`13,`21)[`6,Domain INTEGER],[`13,Domain CHAR],[`21,Domain INTEGER],
  `28=From `28:`29 targets: 23=`31 Source: `31 Target=23 Indexes=[(`6)64],
  `29=Domain TABLE (`6,`13,`21) Display=3[`6,Domain INTEGER],[`13,Domain CHAR],[`21,Domain
INTEGER],
  `30=Domain TABLE (`6,`13,`21)[`6,Domain INTEGER],[`13,Domain CHAR],[`21,Domain INTEGER],
  `31=TableRowSet `31:`30 From: `28 SRow:(43,93,115) Target:23 Indexes: [(`6)=64],
  `33=SelectStatement `33 Union=`3,
  %0=View Name=V %0 Domain %27 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p
    Ppos: 155 Result %4,
```



```

%1=Domain ROW (43)[43,Domain INTEGER],
%4=SelectRowSet %4:%27 targets: 23=%32 Source: %29,
%7=SqlCopy Name=Q %7 Domain 29 From:%29 copy from 43,
%10=Domain ROW (%7)[%7,CONTENT],
%17=Domain ROW (#15)[#15,CONTENT],
%22=SqlCopy Name=A %22 Domain 29 From:%29 copy from 115,
%25=Domain ROW (%22)[%22,CONTENT],
%27=Domain TABLE (%7,#15,%22)[%7,Domain INTEGER],[#15,Domain CHAR],[%22,Domain INTEGER],
%29=From %29:%30 targets: 23=%32 Source: %32 Target=23 Indexes=[(`6)64],
%30=Domain TABLE (%7,#15,%22) Display=3
[%7,Domain INTEGER],[#15,Domain CHAR],[%22,Domain INTEGER],
%31=Domain TABLE (%7,#15,%22)[%7,Domain INTEGER],[#15,Domain CHAR],[%22,Domain INTEGER],
%32=TableRowSet %32:%31 From: %29 SRow:(43,93,115) Target:23 Indexes: [(%7)=64],
%34=SelectStatement %34 Union=%4,
%35=Domain TABLE (#15|%7,%22) Display=1
[#15,Domain CHAR],[%7,Domain INTEGER],[%22,Domain INTEGER],
%36=Domain TABLE (#15,%7,%22)[#15,Domain CHAR],[%7,Domain INTEGER],[%22,Domain INTEGER],
%37=SqlRowSet %37:%41 targets: 23=%32 SqlRows [#24,#35],
%38=Domain ROW (#25)[#25,Domain CHAR],
%39=Domain ROW (#36)[#36,Domain CHAR],
%40=Domain TABLE (#15|%7,%22) Display=1
[#15,Domain CHAR],[%7,Domain INTEGER],[%22,Domain INTEGER],
%41=Domain TABLE (#15) Display=1
[#15,Domain CHAR],[%7,Domain INTEGER],[%22,Domain INTEGER]]}

```

Here objects %0-%35 are instances of objects from the view definition and its framing. In this case we see that the instance of the view has different column uids (in the new domain %27), and the original view definition 155 is no longer referenced.

Execute() (at line 304 in Transaction.cs) sets up a new Activation 7. This calls Obey() for the SqlInsert statement #1. #1.Obey(7) finds the target From #13 and adds the data rowset %37:

```

tg {From #13:%34 targets: 23=%31 Source: %28}
data {SqlRowSet %37:%41 targets: 23=%32 SqlRows [#24,#35]}

```

It examines the targets of the From, and finds just TableRowSet %31., and calls %31.Insert() to create an Activation to do the Insert.

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

The constructor for TableActivation 8 ensures that any dependent compiled objects are instanced (none in this case) and builds a TransitionRowSet for the insert operation on table 23.

```
{TransitionRowSet %41:%30 targets: 23=%31 Data: %36 Target: 23}
```

We see that none of the entries in the context 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.

Traversal of the data rowset begins (line 3183 of Executable.cs). The first data cursor is {(#15=Twenty) %37}, and the TableActivation.EachRow method deals with it. It begins by creating a TransitionCursor {(%7= Null,#15=Twenty,%22= Null) %42} whose TargetCursor for the Table P fills in the key column Q using Pyrrho's autokey feature, {(43=1,93=Twenty,115= Null) %42}. Similarly for the second row of the insert, so, following the autoCommit, we have simply

206	Record 206[23]	43=1,93=Twenty
230	Record 230[23]	43=2,93=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

```

{(-539=ROW,
-538=TABLE,
-505=CONTENT,
-503=Null,
29=Domain INTEGER,
80=Domain CHAR,
#8=From #8:%35 matches (%7=1) targets: 23=%32 Source: %29 Indexes=[(`6)64],
#16=Forty two,
#35=SqlValueExpr Name= #35 BOOLEAN Left:%7 Right:#36 #35(%7=#36),

```

```

#36=1,...,
%0=View Name=V %0 Domain %27 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p Ppos:
155 Result %4,
%1=Domain ROW (43)[43,Domain INTEGER],
%4=SelectRowSet %4:%27 targets: 23=%32 Source: %29,
%7=SqlCopy Name=Q %7 Domain 29 From:%29 copy from 43,
%10=Domain ROW (%7)[%7,CONTENT],
%14=SqlCopy Name=R %14 Domain 80 From:%29 Alias=S copy from 93,
%17=Domain ROW (%14)[%14,CONTENT],
%22=SqlCopy Name=A %22 Domain 29 From:%29 copy from 115,
%25=Domain ROW (%22)[%22,CONTENT],
%27=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%29=From %29:%30 matches (%7=1) targets: 23=%32 Source: %32 Target=23 Indexes=[(`6)64],
%30=Domain TABLE (%7,%14,%22) Display=3[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain
INTEGER],
%31=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%32=TableRowSet %32:%31 key (%7) where (%35) matches (%7=1) From: %29
Assigs:(UpdateAssignment Vbl: %14 Val: #16=True) SRow:(43,93,115) Target:23
Indexes: [(%7)=64],
%34=SelectStatement %34 Union=%4,
%35=Domain TABLE (%7,%14,%22) Display=3[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain
INTEGER],
%36=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%37=UpdateSearch %37 Target: #8)}

```

We see that a matches condition has been added to the From object and the where clause has been passed down to the instances From %29 and TableRowSet %32 (this is why we get a fresh instance for every reference). Again, this reduces the View update to an ordinary table update. After Commit we just have

272	Update 206[23]	93=Forty two Prev:206
-----	----------------	-----------------------

Next we have a simple Delete:

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

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

```

{(-539=ROW,
-538=TABLE,
-505=CONTENT,
-503=NULL,
29=Domain INTEGER,
80=Domain CHAR,
#1=QuerySearch #1 Target: #13,
#13=From #13:%35 matches (%14=Thirty) targets: 23=%32 Source: %29 Indexes=[(`6)64],
#22=SqlValueExpr Name= #22 BOOLEAN Left:%14 Right:#23 #22(%14=#23),
#23=Thirty,...,
%0=View Name=V %0 Domain %27 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p
Ppos: 155 Result %4,
%1=Domain ROW (43)[43,Domain INTEGER],
%4=SelectRowSet %4:%27 targets: 23=%32 Source: %29,
%7=SqlCopy Name=Q %7 Domain 29 From:%29 copy from 43,
%10=Domain ROW (%7)[%7,CONTENT],
%14=SqlCopy Name=R %14 Domain 80 From:%29 Alias=S copy from 93,
%17=Domain ROW (%14)[%14,CONTENT],
%22=SqlCopy Name=A %22 Domain 29 From:%29 copy from 115,
%25=Domain ROW (%22)[%22,CONTENT],
%27=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%29=From %29:%30 matches (%14=Thirty) targets: 23=%32 Source: %32 Target=23
Indexes=[(`6)64],
%30=Domain TABLE (%7,%14,%22) Display=3
[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%31=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%32=TableRowSet %32:%31 key (%7) where (%22) matches (%14=Thirty) From: %29 SRow:(43,93,115)
Target:23 Indexes: [(%7)=64],
%34=SelectStatement %34 Union=%4,
%35=Domain TABLE (%7,%14,%22) Display=3[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain
INTEGER],
%36=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER]}}

```

and the Commit gives

318	Delete Record 233[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

```

348	Record 348[23]	43=2,93=Fifty
389	PTable T	
396	PColumn3 S	for 389(0)[80]
419	PColumn3 U	for 389(1)[29]
461	Record 461[389]	396=Forty two,419=42
491	Record 491[389]	396=Fifty,419=48
535	PView W 535	select * from t natural join v

Let us restart the server once again (for reproducibility of Activation numbers). The framing for W is as follows (note the copy of V's framing at `0 to `32):

```

{Framing (
`0 Domain ROW (43)[43,Domain 135],
`3 SelectRowSet `3: 26 targets: 23=`31 Source: `28,
`6 SqlCopy Name=Q `6 Domain 29 From:`28 copy from 43,
`9 Domain ROW (`6)[`6,CONTENT],
`13 SqlCopy Name=R `13 Domain 80 From:`28 Alias=S copy from 93,
`16 Domain ROW (`13)[`13,CONTENT],
`21 SqlCopy Name=A `21 Domain 29 From:`28 copy from 115,
`24 Domain ROW (`21)[`21,CONTENT],
`26 Domain TABLE (`6,`13,`21)[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
`28 From `28:`29 targets: 23=`31 Source: `31 Target=23 Indexes=[(`6)64],
`29 Domain TABLE (`6,`13,`21) Display=3[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
`30 Domain TABLE (`6,`13,`21)[`6,Domain 135],[`13,Domain CHAR],[`21,Domain 135],
`31 TableRowSet `31:`30 From: `28 SRow:(43,93,115) Target:23 Indexes: [(`6)=64],
`33 SelectStatement `33 Union=`3,
`36 SelectRowSet `36:`91 matching (`63=(`63)) targets: 23=`81,389=`45 Source: `46,
`37 SqlStar Name=* `37 CONTENT,
`38 Domain TABLE (`37)[`37,CONTENT],
`40 From `40:`43 order (`63) targets: 389=`45 Source: `89 Target=389,
`42 SqlCopy Name=U `42 Domain 29 From:`40 copy from 419,
`43 Domain TABLE (`63,`42) Display=2[`63,Domain CHAR],[`42,Domain 135],
`44 Domain TABLE (`63,`42)[`63,Domain CHAR],[`42,Domain 135],
`45 TableRowSet `45:`44 From: `40 SRow:(396,419) Target:389,
`46 JoinRowSet `46:`86 matching (`63=(`63)) targets: 23=`81,389=`45 INNER JoinCond: (`88)
    First: `40 Second: `48 on `63=`63,
`48 From `48:`84 order (`63) targets: 23=`81 Source: `78 Indexes=[(`6)64],
`49 View Name=V `49 Domain `76 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p
    Ppos: 155 Result `53,
`50 Domain ROW (43)[43,Domain 135],
`53 SelectRowSet `53:`76 targets: 23=`81 Source: `78,
`56 SqlCopy Name=Q `56 Domain 29 From:`78 copy from 43,
`59 Domain ROW (`56)[`56,CONTENT],
`63 SqlCopy Name=R `63 Domain 80 From:`78 Alias=S copy from 93,
`66 Domain ROW (`63)[`63,CONTENT],
`71 SqlCopy Name=A `71 Domain 29 From:`78 copy from 115,
`74 Domain ROW (`71)[`71,CONTENT],
`76 Domain TABLE (`56,`63,`71)[`56,Domain 135],[`63,Domain CHAR],[`71,Domain 135],
`78 From `78:`79 order (`63) targets: 23=`81 Source: `90 Target=23 Indexes=[(`6)64],
`79 Domain TABLE (`56,`63,`71) Display=3[`56,Domain 135],[`63,Domain CHAR],[`71,Domain 135],
`80 Domain TABLE (`56,`63,`71)[`56,Domain 135],[`63,Domain CHAR],[`71,Domain 135],
`81 TableRowSet `81:`80 From: `78 SRow:(43,93,115) Target:23 Indexes: [(`56)=64],
`83 SelectStatement `83 Union=`53,
`84 Domain TABLE (`56,`63,`71) Display=3[`56,Domain 135],[`63,Domain CHAR],[`71,Domain 135],
`85 Domain TABLE (`56,`63,`71)[`56,Domain 135],[`63,Domain CHAR],[`71,Domain 135],
`86 Domain TABLE (`42,`56,`71|`63) Display=3
    [ `42,Domain 135],[`56,Domain 135],[`71,Domain 135],[`63,Domain CHAR],
`87 SqlValueExpr Name= `87 BOOLEAN From:`46 Left:`63 Right:`63 `87(`63=`63),
`88 SqlValueExpr Name= `88 BOOLEAN Left:`63 Right:`63 `88(`63=`63),
`89 OrderedRowSet `89:`44 key (`63) order (`63) Source: `45,
`90 OrderedRowSet `90:`80 key (`63) order (`63) Source: `81,
`91 Domain TABLE (`42,`56,`71) Display=3[`42,Domain 135],[`56,Domain 135],[`71,Domain 135],
`92 SelectStatement `92 Union=`36) Result `36}

```

Note that the view V has been instantiated as part of view W (so that W's framing contains an instantiated 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 instantiated to use heap uids, and the resulting instances of TableRowSets such as `80 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 `88 and or `89 may not be required if a where-condition limits the rows involved.

The case of inserting into a join is not interesting, but update and delete for joins turn out to work very well. Consider the next step in the test:

```
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

```
{(-539=ROW,
-538=TABLE,
-505=CONTENT,
-503=NULL,
29=Domain INTEGER,
80=Domain CHAR,
#8=From #8:%94 matches (%57=2) matching (%64=(%64)) targets: 23=%82,389=%46 Source: %47
Indexes=[(`56)64],
#16=50,
#21=21,
#31=SqlValueExpr Name= #31 BOOLEAN Left:%57 Right:#32 #31(%57=#32),
#32=2,...,
%0=View Name=W %0 Domain %92 Definer=-502 Ppos=535 ViewDef as select * from t natural join v
Ppos: 535 Result %37,
%1=Domain ROW (43)[43,Domain INTEGER],
%4=SelectRowSet %4:%27 targets: 23=%32 Source: %29,
%7=SqlCopy Name=Q %7 Domain 29 From:%29 copy from 43,
%10=Domain ROW (%7)[%7,CONTENT],
%14=SqlCopy Name=R %14 Domain 80 From:%29 Alias=S copy from 93,
%17=Domain ROW (%14)[%14,CONTENT],
%22=SqlCopy Name=A %22 Domain 29 From:%29 copy from 115,
%25=Domain ROW (%22)[%22,CONTENT],
%27=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%29=From %29:%30 targets: 23=%32 Source: %32 Target=23 Indexes=[(`6)64],
%30=Domain TABLE (%7,%14,%22) Display=3[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain
INTEGER],
%31=Domain TABLE (%7,%14,%22)[%7,Domain INTEGER],[%14,Domain CHAR],[%22,Domain INTEGER],
%32=TableRowSet %32:%31 From: %29 SRow:(43,93,115) Target:23 Indexes: [(%7)=64],
%34=SelectStatement %34 Union=%4,
%37=SelectRowSet %37:%92 matching (%64=(%64)) targets: 23=%82,389=%46 Source: %47,
%38=SqlStar Name=* %38 CONTENT,
%39=Domain TABLE (%38)[%38,CONTENT],
%41=From %41:%44 order (%64) targets: 389=%46 Source: %90 Target=389,
%43=SqlCopy Name=U %43 Domain 29 From:%41 copy from 419,
%44=Domain TABLE (%64,%43) Display=2[%64,Domain CHAR],[%43,Domain INTEGER],
%45=Domain TABLE (%64,%43)[%64,Domain CHAR],[%43,Domain INTEGER],
%46=TableRowSet %46:%45 From: %41 Assigs:(UpdateAssignment Vbl: %43 Val: #16=True)
SRow:(396,419) Target:389,
%47=JoinRowSet %47:%87 matching (%64=(%64)) targets: 23=%82,389=%46 INNER JoinCond: (%89)
First: %41 Second: %49 on %64=%64,
%49=From %49:%85 order (%64) matches (%57=2) matching (%64=(%64)) targets: 23=%82
Source: %79 Indexes=[(`6)64],
%50=View Name=V %50 Domain %77 Definer=-502 Ppos=155 ViewDef as select q,r as s,a from p
Ppos: 155 Result `53,
%51=Domain ROW (43)[43,Domain INTEGER],
%54=SelectRowSet %54:%77 targets: 23=%82 Source: %79,
%57=SqlCopy Name=Q %57 Domain 29 From:%79 copy from 43,
%60=Domain ROW (%57)[%57,CONTENT],
%64=SqlCopy Name=R %64 Domain 80 From:%79 Alias=S copy from 93,
%67=Domain ROW (%64)[%64,CONTENT],
%72=SqlCopy Name=A %72 Domain 29 From:%79 copy from 115,
%75=Domain ROW (%72)[%72,CONTENT],
%77=Domain TABLE (%57,%64,%72)[%57,Domain INTEGER],[%64,Domain CHAR],[%72,Domain INTEGER],
%79=From %79:%80 order (%64) matches (%57=2) targets: 23=%82 Source: %91 Target=23
Indexes=[(`6)64],
%80=Domain TABLE (%57,%64,%72) Display=3
[%57,Domain INTEGER],[%64,Domain CHAR],[%72,Domain INTEGER],
%81=Domain TABLE (%57,%64,%72)[%57,Domain INTEGER],[%64,Domain CHAR],[%72,Domain INTEGER],
%82=TableRowSet %82:%81 key (%57) where (#31) matches (%57=2) From: %79
```

```

Assigs:(UpdateAssignment Vbl: %72 Val: #21=True) SRow:(43,93,115) Target:23
Indexes: [(%57)=64],
%84=SelectStatement %84 Union=%54,
%85=Domain TABLE (%57,%64,%72) Display=3
[%57,Domain INTEGER],[%64,Domain CHAR],[%72,Domain INTEGER],
%86=Domain TABLE (%57,%64,%72)[%57,Domain INTEGER],[%64,Domain CHAR],[%72,Domain INTEGER],
%87=Domain TABLE (%43,%57,%72[%64]) Display=3
[%43,Domain INTEGER],[%57,Domain INTEGER],[%72,Domain INTEGER],[%64,Domain CHAR],
%88=SqlValueExpr Name= %88 BOOLEAN From:%47 Left:%64 Right:%64 %88(%64=%64),
%89=SqlValueExpr Name= %89 BOOLEAN Left:%64 Right:%64 %89(%64=%64),
%90=OrderedRowSet %90:%45 key (%64) order (%64) Source: %46,
%91=OrderedRowSet %91:%81 key (%64) order (%64) matches (%57=2) Source: %82,
%92=Domain TABLE (%43,%57,%72) Display=3
[%43,Domain INTEGER],[%57,Domain INTEGER],[%72,Domain INTEGER],
%93=SelectStatement %93 Union=%37,
%94=Domain TABLE (%43,%57,%72) Display=3[%43,Domain INTEGER],[%57,Domain
INTEGER],[%72,Domain INTEGER],
%95=Domain TABLE (%43,%57,%72)[%43,Domain INTEGER],[%57,Domain INTEGER],[%72,Domain
INTEGER],
%96=UpdateSearch %96 Target: #8)}

```

Notice that the OrderedRowSets are not required in this instance.

Execute() starts an Activation 8 to handle the update, and calls %96.Obey(). This sets up TableActivations for the two targets %46 and %82 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 348[23]	115=21,Prev:348
614	Update 491[389]	419=50,Prev=491

## 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().

If we 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 %40 Target: UpdateSearch %23 Target: %17 Params: %27,%37}
```

and its framing has been simply added to the current context:

```

{(%17=From %17:%20 matches (%19=?%37) targets: 23=%22 Source: %22 Target=23 Indexes=[(%18)66],
%18=SqlCopy Name=A %18 Domain 31 From:%17 copy from 45,
%19=SqlCopy Name=B %19 Domain 84 From:%17 copy from 97,
%20=Domain TABLE (%18,%19) Display=2[%18,Domain INTEGER],[%19,Domain CHAR],
%21=Domain TABLE (%18,%19)[%18,Domain INTEGER],[%19,Domain CHAR],
%22=TableRowSet %22:%21 key (%18) where (%35) matches (%19=?%37) From: %17
Assigs:(UpdateAssignment Vbl: %18 Val: %27=True) SRow:(45,97) Target:23 Indexes: [(%18)=66],
%23=UpdateSearch %23 Target: %17,
%27=?%27,
%35=SqlValueExpr Name= %35 BOOLEAN Left:%19 Right:%37 %35(%19=%37),
%37=?%37,
%40=PreparedStatement %40 Target: UpdateSearch %23 Target: %17 Params: %27,%37)}

```

Step over the call to QParams, and cx.obs has become

```

{(%17=From %17:%20 matches (%19=HalfDozen) targets: 23=%22 Source: %22 Target=23
Indexes=[(%18)66],
%18=SqlCopy Name=A %18 Domain 31 From:%17 copy from 45,
%19=SqlCopy Name=B %19 Domain 84 From:%17 copy from 97,

```

```

%20=Domain TABLE (%18,%19) Display=2[%18,Domain INTEGER],[%19,Domain CHAR],
%21=Domain TABLE (%18,%19)[%18,Domain INTEGER],[%19,Domain CHAR],
%22=TableRowSet %22:%21 key (%18) where (%35) matches (%19=HalfDozen) From: %17
    Assigs:(UpdateAssignment Vb1: %18 Val: %27=True) SRow:(45,97) Target:23
    Indexes: [(%18)=66],
%23=UpdateSearch %23 Target: %17,
%27=6,
%35=SqlValueExpr Name= %35 BOOLEAN Left:%19 Right:%37 %35(%19=%37),
%37=HalfDozen,
%40=PreparedStatement %40 Target: UpdateSearch %23 Target: %17 Params: %27,%37)}

```

We can immediately see that 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 no database file, and in PyrrhoCmd t18 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	Domain INTEGER
48	PColumn3 ID for 23(0)[34]
70	PIndex AUTHOR on 23(48) PrimaryKey
91	Domain CHAR
104	PColumn3 ANAME for 23(1)[91]
148	PTable BOOK
158	PColumn3 ID for 148(0)[34]
182	PIndex BOOK on 148(158) PrimaryKey
204	PColumn3 AUTHID for 148(1)[34]
233	PIndex2 on 148(204) ForeignKey, RestrictUpdate, RestrictDelete refers to [70]
253	PColumn3 TITLE for 148(2)[91]
281	PTransaction for 2 Role=-502 User=-501 Time=06/10/2021 07:40:39
299	Record 299[23]: 48=1,104=Dickens
324	Record 324[23]: 48=2,104=Conrad
366	Record 366[148]: 158=1,204=1,253=Dombey & Son
405	Record 405[148]: 158=2,204=2,253=Lord Jim
440	Record 440[148]: 158=3,204=1,253=David Copperfield

Place a breakpoint in ParseProcedureClause at line 2347. 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 2347 we have:

```

{{Procedure Name=BOOKSBY !0 Domain `4 Definer=-502 BOOKSBY Arity=1 `4 Params(`3) Body:_ Clause{}}

```

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

```

{Framing (
  `3 FormalParameter Name=AUTH `3 CHAR From:!0 IN,

```



June 2022

```
`4 Domain `4 (title char) TABLE (`5)[`5,CHAR] structure=_,  
`5 SqlElement Name=TITLE `5 CHAR From:{0}}}
```

Then `cx.Add(pp)` calls `db.Add()` which calls `PProcedure.Install()`, and this makes changes to the role (for name/arity lookup) and the Database (for execution). Each has a structure called **procedures**, which is initially empty. By line 2347 of `Parser.cs` these are respectively

```
{(BOOKSBY=(1={0}))}  
{({0}=BOOKSBY{1})}
```

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

```
{(23=Table Name=AUTHOR 23 TABLE Definer=-502 Ppos=23:-538 Indexes:((48)70) KeyCols: (48=True),  
48=TableColumn 48 Domain 34 Definer=-502 Ppos=48 34 Table=23,  
91=Domain CHAR,  
104=TableColumn 104 Domain 91 Definer=-502 Ppos=104 91 Table=23,  
148=Table Name=BOOK 148 TABLE Definer=-502 Ppos=148:-538 Indexes:((158)182,(204)233) KeyCols:  
(158=True,204=True),  
158=TableColumn 158 Domain 34 Definer=-502 Ppos=158 34 Table=148,  
204=TableColumn 204 Domain 34 Definer=-502 Ppos=204 34 Table=148,  
253=TableColumn 253 Domain 91 Definer=-502 Ppos=253 91 Table=148,  
!0=Procedure Name=BOOKSBY !0 Domain `4 Definer=-502 BOOKSBY Arity=1 `4 Params(`3) Body:`66  
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)),  
#154=ForwardReference Name=BOOKSBY #154 Domain `62 Definer=-501 Ppos=#154,  
`3=FormalParameter Name=AUTH `3 CHAR From:{0 IN,  
`4=Domain (title char) TABLE (`5)[`5,CHAR] structure=_,  
`5=SqlElement Name=TITLE `5 CHAR From:{0,  
`9=SelectRowSet `9: `17 where (`55) matching (`20=(`29),`29=(`20)) targets: 23=`24,148=`32  
Source: `25,  
`12=SqlCopy Name=TITLE `12 Domain 91 From:`27 copy from 253,  
`15=Domain ROW (`12)[`12,CHAR],  
`17=Domain TABLE (`12)[`12,Domain CHAR],  
`19=From `19:`22 order (`20) targets: 23=`24 Source: `24 Target=23 Indexes=[(`20)70],  
`20=SqlCopy Name=AUTHOR.ID `20 Domain 34 From:`19 copy from 48,  
`21=SqlCopy Name=ANAME `21 Domain 91 From:`19 copy from 104,  
`22=Domain TABLE (`20,`21) Display=2[`20,Domain INTEGER],[`21,Domain CHAR],  
`23=Domain TABLE (`20,`21)[`20,Domain INTEGER],[`21,Domain CHAR],  
`24=TableRowSet `24:`23 key (`20) order (`20) From: `19 SRow:(48,104) Target:23  
Indexes: [(`20)=70],  
`25=JoinRowSet `25:`48 matching (`20=(`29),`29=(`20)) targets: 23=`24,148=`32 INNER  
JoinCond: (`40,`55) First: `19 Second: `27 on `20=`29,  
`27=From `27:`30 order (`29) targets: 148=`32 Source: `32 Alias B Target=148  
Indexes=[(`28)182,(`29)233],  
`28=SqlCopy Name=B.ID `28 Domain 34 From:`27 copy from 158,  
`29=SqlCopy Name=AUTHID `29 Domain 34 From:`27 copy from 204,  
`30=Domain TABLE (`12|`28,`29) Display=1  
[ `12,Domain CHAR],[`28,Domain INTEGER],[`29,Domain INTEGER],  
`31=Domain TABLE (`28,`29,`12)[`28,Domain INTEGER],[`29,Domain INTEGER],[`12,Domain CHAR],  
`32=TableRowSet `32:`31 key (`29) order (`29) From: `27 SRow:(158,204,253) Target:148  
Indexes: [(`28)=182,(`29)=233],  
`40=SqlValueExpr Name= `40 BOOLEAN Left:`20 Right:`29 `40(`20=`29),  
`48=Domain TABLE (`20,`21,`12|`28,`29) Display=3  
[ `20,Domain INTEGER],[`21,Domain CHAR],[`12,Domain CHAR],  
[ `28,Domain INTEGER],[`29,Domain INTEGER],  
`55=SqlValueExpr Name= `55 BOOLEAN Left:`21 Right:`3 `55(`21=`3),  
`62=Domain ROW (`3)[`3,Domain CHAR],  
`64=SelectStatement `64 Union=`9,  
`65=SqlValueSelect `65 Domain `17 (`9),  
`66=ReturnStatement `66 -> `65)}
```

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

```
> on author.id=b.authid where aname=booksby.auth  
SQL-T>select * from table(booksby('Dickens'))  
-----  
TITLE  
-----  
Dombey & Son  
David Copperfield  
-----  
SQL-T>
```

Procedure object itself now has a permanent defining position:

```
{(...
  502=Procedure Name=BOOKSBY 502 Domain `4 Definer=-502 Ppos=502 BOOKSBY Arity=1 `4 Params(`3)
    Body:`66 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. Instanting 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	Domain INTEGER
56	PColumn3 X for 23(0)[42]
77	PColumn3 Y for 23(1)[42]
99	PType POINT[23]
139	PTable (w int,h int)
157	PColumn3 W for 139(0)[42]
179	PColumn3 H for 139(1)[42]
202	PType SIZE[139]
242	PTable (strt point,en point)
269	PColumn3 STRT for 242(0)[99]
295	PColumn3 EN for 242(1)[99]
320	PType LINE[242]
361	PTable (tl point,sz size)
386	PColumn3 TL for 361(0)[99]
410	PColumn3 SZ for 361(1)[202]
436	PType RECT[361]
459	Method Constructor 459=436.RECT\$4(x1 int,y1 int, x2 int, y2 int)
511	Method Instance 511=436.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<sup>38</sup>

```
23 {VirtualTable Name=(x int, y int) 23 TABLE Definer=-502 Ppos=23:-538 RestView=_}
99 {UDType 99 POINT TYPE (56,77)[56,135],[77,135] structure=23}
```

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

```
459 {Method Name=RECT 459 Null Definer=-502 Ppos=459 RECT Arity=4 Params(`0`,`1`,`2`,`3) Body:
  Clause{(x1 int,y1 int, x2 int, y2 int)}
  UDType=UDType RECT TYPE (386,410)
    [386,UDType POINT TYPE (56,77)
      [56,Domain 135],[77,Domain 135] structure=23],
```

<sup>38</sup> In these listings Visual Studio has displayed INTEGER as 135 (which is (int)Sqlx.INTEGER). Why?



```

[410,UDType SIZE TYPE (157,179)
  [157,Domain 135],[179,Domain 135] structure=139]
structure=361 MethodType=Constructor}
511 {Method Name=CENTRE 511 Domain 99 Definer=-502 Ppos=511 CENTRE Arity=0 99 Params) Body:
  Clause{() returns point}
  UDTType=UDType RECT TYPE (386,410)
    [386,UDType POINT TYPE (56,77)
      [56,Domain 135],[77,Domain 135] structure=23],
    [410,UDType SIZE TYPE (157,179)
      [157,Domain 135],[179,Domain 135] structure=139] structure=361
  Methods: 459 RECT$4 MethodType=Instance}

```

(The copy of object 459 in object in 511 shows the previously defined method RECT\$4: up-to-date versions 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 (
  `0 FormalParameter Name=X1 `0 135 From:`0 IN,
  `1 FormalParameter Name=Y1 `1 135 From:`0 IN,
  `2 FormalParameter Name=X2 `2 135 From:`0 IN,
  `3 FormalParameter Name=Y2 `3 135 From:`0 IN,
  `6 CompoundStatement `6(`8,`27),
  `8 AssignmentStatement `8 386=`9,
  `9 SqlDefaultConstructor `9 Domain 99 Sce:`24,
  `24 SqlRow `24 Domain `25,
  `25 Domain ROW (`0,`1)[`0,Domain 135],[`1,Domain 135],
  `27 AssignmentStatement `27 410=`28,
  `28 SqlDefaultConstructor `28 Domain 202 Sce:`57,
  `36 SqlValueExpr Name= `36 Domain 42 Left:`2 Right:`0 `36(`2-`0),
  `49 SqlValueExpr Name= `49 Domain 42 Left:`3 Right:`1 `49(`3-`1),
  `57 SqlRow `57 Domain `58,
  `58 Domain ROW (`36,`49)[`36,Domain 135],[`49,Domain 135])}

```

In particular, we see that the body of the procedure is given by the CompoundStatement `25. Note the calls on default constructors for Point at `9 and Size at `28. 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 (..,
  `60 SqlDefaultConstructor `60 Domain 99 Sce:`105,
  `67 SqlCopy Name=TL `67 Domain 99 copy from 386,
  `68 SqlField Name=X `68 Domain 42 Parent=`67 Field=56,
  `70 SqlValueExpr Name= `70 Domain 42 Left:`68 Right:`79 `70(`68+`79),
  `76 SqlCopy Name=SZ `76 Domain 202 copy from 410,
  `77 SqlField Name=W `77 Domain 42 Parent=`76 Field=157,
  `79 SqlValueExpr Name= `79 Domain 42 Left:`77 Right:`81 `79(`77/`81),
  `81 2,
  `88 SqlCopy Name=TL `88 Domain 99 copy from 386,
  `89 SqlField Name=Y `89 Domain 42 Parent=`88 Field=77,
  `91 SqlValueExpr Name= `91 Domain 42 Left:`89 Right:`100 `91(`89+`100),
  `97 SqlCopy Name=SZ `97 Domain 202 copy from 410,
  `98 SqlField Name=H `98 Domain 42 Parent=`97 Field=179,
  `100 SqlValueExpr Name= `100 Domain 42 Left:`98 Right:`102 `100(`98/`102),
  `102 2,
  `105 SqlRow `105 Domain `106,

```

June 2022

```
`106 Domain ROW (`70,`91)[`70,Domain 135],[`91,Domain 135],  
`107 ReturnStatement `107 -> `60}}
```

and now the method objects have been updated:

```
{Method Name=RECT 459 Null Definer=-502 Ppos=459 RECT Arity=4 Params(`0,`1,`2,`3) Body:`6  
  Clause{(x1 int,y1 int, x2 int, y2 int)}  
  UDTType=UDType RECT TYPE (386,410)  
    [386,UDType POINT TYPE (56,77)  
      [56,Domain 135],[77,Domain 135] structure=23],  
    [410,UDType SIZE TYPE (157,179)  
      [157,Domain 135],[179,Domain 135] structure=139]  
  structure=361 MethodType=Constructor}  
  
{Method Name=CENTRE 511 Domain 99 Definer=-502 Ppos=511 CENTRE Arity=0 99 Params) Body:`107  
  Clause{() returns point}  
  UDTType=UDType RECT TYPE (386,410)  
    [386,UDType POINT TYPE (56,77)  
      [56,Domain 135],[77,Domain 135] structure=23],  
    [410,UDType SIZE TYPE (157,179)  
      [157,Domain 135],[179,Domain 135] structure=139]  
  structure=361 Methods: 459 RECT$4 MethodType=Instance}
```

## 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"
```

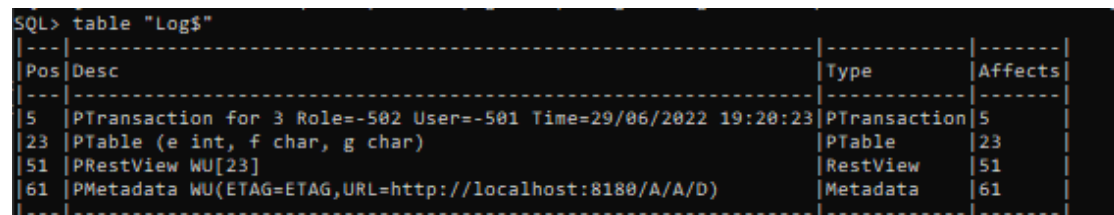
For this section, the server is started with +s and -H flags. The +s flag ensures that Pyrrho’s HTTP service is running, and -H gives us some useful diagnostic information as we will see.

### 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



Pos	Desc	Type	Affects
5	PTransaction for 3 Role=-502 User=-501 Time=29/06/2022 19:20:23	PTransaction	5
23	PTable (e int, f char, g char)	PTable	23
51	PRestView WU[23]	RestView	51
61	PMetadata WU(ETAG=ETAG,URL=http://localhost:8180/A/A/D)	Metadata	61

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 framing for the above committed PRestView will be

```
{Framing (  
  `1=SqlValue Name=E `1 135 From:23,  
  `3=SqlValue Name=F `3 CHAR From:23,  
  `5=SqlValue Name=G `5 CHAR From:23,  
  `7=Domain TABLE (`1,`3,`5)[`1,135],[`3,CHAR],[`5,CHAR] structure=23)}
```

And the current (schema) Role (cx.role.infos[23] etc) contains

```
23 {ObInfo Name=WU 23 TABLE WU Domain TABLE (`1,`3,`5)[`1,135],[`3,CHAR],[`5,CHAR] structure=23  
  Privilege=13729119}  
51 {ObInfo Name=WU 51 Domain `7 WU Domain TABLE (`1,`3,`5)[`1,135],[`3,CHAR],[`5,CHAR]  
  structure=23 Privilege=1077583 ETAG URL http://localhost:8180/A/A/D}
```

Now on a simple query, such as

```
select * from wu
```

Following query processing as in previous sections, and RESTView instancing, we get the following at the point where traversal begins (Start.cs line 426):

```
{(-538=TABLE,
-503=NULL,
23=VirtualTable Name=(e int, f char, g char) 23 Domain `6 Definer=-502 Ppos=23:`6 RestView=51,
#1=SelectRowSet #1:%13 targets: 51=%9 Source: #15,
#8=SqlStar Name=* #8 CONTENT,
#15=From #15:%11 targets: 51=%9 Source: %9 Target=51,
`1=SqlValue Name=E `1 INTEGER From:23,
`3=SqlValue Name=F `3 CHAR From:23,
`5=SqlValue Name=G `5 CHAR From:23,
`7=Domain TABLE (`1,`3,`5)[`1,INTEGER],[`3,CHAR],[`5,CHAR] structure=23,
%0=Domain TABLE (#8)[#8,CONTENT],
%1=RestView Name=WU %1 Domain %8 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
    UsingTableRowSet: _ ViewTable:23,
%2=SqlValue Name=E %2 INTEGER From:23,
%4=SqlValue Name=F %4 CHAR From:23,
%6=SqlValue Name=G %6 CHAR From:23,
%8=Domain TABLE (%2,%4,%6)[%2,INTEGER],[%4,CHAR],[%6,CHAR] structure=23,
%9=RestRowSet %9:%10 targets: 51=%9 SRow:() http://localhost:8180/A/A/D RemoteCols:(%2,%4,%6)
    RemoteNames:(%2=E,%4=F,%6=G),
%10=Domain TABLE (%2,%4,%6)[%2,INTEGER],[%4,CHAR],[%6,CHAR],
%11=Domain TABLE (%2,%4,%6) Display=3[%2,INTEGER],[%4,CHAR],[%6,CHAR],
%12=Domain TABLE (%2,%4,%6)[%2,INTEGER],[%4,CHAR],[%6,CHAR],
%13=Domain TABLE (%2,%4,%6) Display=3[%2,INTEGER],[%4,CHAR],[%6,CHAR],
%14=SelectStatement %14 Union=#1)}
```

Note that the RestRowSet is for the instance and refers to the RestView 51 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. By default, this is carried out using a POST request to the remote server. The headers are something like

```
GET /A/A/D HTTP1.1
Connection: Keep-Alive
Accept: application/json
Authorization: Basic [REDACTED]
Host: localhost:8180
User-Agent: Pyrrho 7.0 alpha
```

In this case (because of the URL flag in the Metadata above, a GET request was sent to the URL "http://localhost:8180/A/A/D". 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, 29 Jun 2022 19:26:41 GMT
Returning ETag: "23,_,185"
--> 2 rows
```

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, the fact that all rows were read (\_ is the uid -1L), and the latest file position for the returned TableRows. The client is using autocommit mode, and so there is no HEAD round trip.

It is a bit confusing that the Pyrrho server handles this request in another thread (the "2 rows" report is from the transaction thread, while the other lines are from the HttpServer thread). The web output from the HttServer thread is in Json format and is received by RestRowSet.GetResponse(), which constructs it into a TArray as the aVal field of the RestRowSet. This completes the Build step for the RestRowSet.

Now 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 145, K 198 and M 221,

132 PTable HU	PTable	132
139 Domain INTEGER	PDomain	139
153 PColumn3 E for 132(0)[139]	PColumn3	153
176 PIndex HU on 132(153) PrimaryKey	PIndex	176
195 Domain CHAR	PDomain	195
208 PColumn3 K for 132(1)[195]	PColumn3	208
232 PColumn3 M for 132(2)[139]	PColumn3	232
256 PTransaction for 2 Roles=-502 User=-501 Time=29/06/2022 19:33:56	PTransaction	256
274 Record 274[132]: 153=1,208=Cleaner,232=12500	Record	274
309 Record 309[132]: 153=2,208=Manager,232=31400	Record	309
344 PTransaction for 1 Roles=-502 User=-501 Time=29/06/2022 19:33:56	PTransaction	344
362 PView VU 362 view VU as select * from WU natural join HU	PView	362
---	-----	-----

VU is committed at file position 362 with framing

```
{Framing (
  `1=SqlValue Name=E `1 135 From:23,
  `3=SqlValue Name=F `3 CHAR From:23,
  `5=SqlValue Name=G `5 CHAR From:23,
  `7=Domain TABLE (`1,`3,`5)[`1,135],[`3,CHAR],[`5,CHAR] structure=23,
  `10=SelectRowSet `10:`40 matching (`16=(`30),`30=(`16)) targets: 51=`23,132=`35 Source: `27,
  `11=SqlStar Name=* `11 CONTENT,
  `12=Domain TABLE (`11)[`11,CONTENT],
  `14=From `14:`25 order (`16) targets: 51=`23 Source: `39 Target=51,
  `15=RestView Name=WU `15 Domain `22 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
    UsingTableRowSet: _ ViewTable:23,
  `16=SqlCopy Name=E `16 Domain 139 From:`29 Alias= copy from 153,
  `18=SqlValue Name=F `18 CHAR From:23,
  `20=SqlValue Name=G `20 CHAR From:23,
  `22=Domain TABLE (`16,`18,`20)[`16,135],[`18,CHAR],[`20,CHAR] structure=23,
  `23=RestRowSet `23:`24 targets: 51=`23 SRow:() http://localhost:8180/A/A/D
    RemoteCols:(`16,`18,`20) RemoteNames:(`16=E,`18=F,`20=G),
  `24=Domain TABLE (`16,`18,`20)[`16,135],[`18,CHAR],[`20,CHAR],
  `25=Domain TABLE (`16,`18,`20) Display=3[`16,135],[`18,CHAR],[`20,CHAR],
  `26=Domain TABLE (`16,`18,`20)[`16,135],[`18,CHAR],[`20,CHAR],
  `27=JoinRowSet `27:`36 matching (`16=(`30),`30=(`16)) targets: 51=`23,132=`35 INNER
    JoinCond: (`38) First: `14 Second: `29 on `16=`30,
  `29=From `29:`33 order (`30) targets: 132=`35 Source: `35 Target=132 Indexes=[(`30)176],
  `30=SqlCopy Name=E `30 Domain 139 From:`29 copy from 153,
  `31=SqlCopy Name=K `31 Domain 195 From:`29 copy from 208,
  `32=SqlCopy Name=M `32 Domain 139 From:`29 copy from 232,
  `33=Domain TABLE (`30,`31,`32) Display=3[`30,Domain 135],[`31,Domain CHAR],[`32,Domain 135],
  `34=Domain TABLE (`30,`31,`32)[`30,Domain 135],[`31,Domain CHAR],[`32,Domain 135],
  `35=TableRowSet `35:`34 key (`30) order (`30) From: `29 SRow:(153,208,232) Target:132
    Indexes: [(`30)=176],
  `36=Domain TABLE (`16,`18,`20,`31,`32)[`30] Display=5
    [`16,Domain 135],[`18,CHAR],[`20,CHAR],[`31,Domain CHAR],[`32,Domain 135],
    [`30,Domain 135],
  `37=SqlValueExpr Name= `37 BOOLEAN From:`27 Left:`16 Right:`30 `37(`16=`30),
  `38=SqlValueExpr Name= `38 BOOLEAN Left:`16 Right:`30 `38(`16=`30),
  `39=OrderedRowSet `39:`24 key (`16) order (`16) targets: 51=`23 Source: `23,
  `40=Domain TABLE (`16,`18,`20,`31,`32) Display=5
    [`16,Domain 135],[`18,CHAR],[`20,CHAR],[`31,Domain CHAR],[`32,Domain 135],
  `41=SelectStatement `41 Union=`10)}
```

Notice that the join needs to order the rows of one of its components (because there is no index for the remote rowset). The framing for view VU contains an instance of the RestRowSet WU from above, shown in blue. 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 426 again), the Context contains (in addition to copies of VU's framing objects<sup>39</sup>):

```
{(-538=TABLE,
-505=CONTENT,
-503=NULL,
139=Domain INTEGER,
195=Domain CHAR,
#1=SelectRowSet #1:%3 where (#31) matching (%20=(%34),%34=(%20)) targets: 51=%27,132=%39
Source: #21,
#21=From #21:%46 matches (%20=1,%34=1) matching (%20=(%34),%34=(%20)) targets: 51=%27,132=%39
Source: %31 Indexes=[('30)176],
#31=SqlValueExpr Name= #31 BOOLEAN Left:%20 Right:#32 #31(%20=#32),
#32=1,...,
%0=Domain ROW (%20)[%20,Domain INTEGER],
%1=Domain ROW (%22)[%22,CHAR],
%2=Domain ROW (%36)[%36,Domain INTEGER],
%3=Domain TABLE (%20,%22,%36)[%20,Domain INTEGER],[%22,CHAR],[%36,Domain INTEGER],
%4=View Name=VU %4 Domain %44 Definer=-502 Ppos=362
ViewDef view VU as select * from WU natural join HU Ppos: 362 Result %14,
%5=SqlValue Name=E %5 INTEGER From:23,
%7=SqlValue Name=F %7 CHAR From:23,
%9=SqlValue Name=G %9 CHAR From:23,
%11=Domain TABLE (%5,%7,%9)[%5,INTEGER],[%7,CHAR],[%9,CHAR] structure=23,
%14=SelectRowSet %14:%44 matching (%20=(%34),%34=(%20)) targets: 51=%27,132=%39 Source: %31,
%15=SqlStar Name=* %15 CONTENT,
%16=Domain TABLE (%15)[%15,CONTENT],
%18=From %18:%29 order (%20) matches (%20=1) matching (%20=(%34),%34=(%20)) targets: 51=%27
Source: %27 Target=51,
%19=RestView Name=WU %19 Domain %26 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
UsingTableRowSet: _ ViewTable:23,
%20=SqlCopy Name=E %20 Domain 139 From:%33 Alias= copy from 153,
%22=SqlValue Name=F %22 CHAR From:23,
%24=SqlValue Name=G %24 CHAR From:23,
%26=Domain TABLE (%20,%22,%24)[%20,INTEGER],[%22,CHAR],[%24,CHAR] structure=23,
%27=RestRowSet %27:%28 matches (%20=1) targets: 51=%27 SRow:() http://localhost:8180/A/A/D
RemoteCols: (%20,%22,%24) RemoteNames: (%20=E,%22=F,%24=G),
%28=Domain TABLE (%20,%22,%24)[%20,INTEGER],[%22,CHAR],[%24,CHAR],
%29=Domain TABLE (%20,%22,%24) Display=3[%20,INTEGER],[%22,CHAR],[%24,CHAR],
%30=Domain TABLE (%20,%22,%24)[%20,INTEGER],[%22,CHAR],[%24,CHAR],
%31=JoinRowSet %31:%40 matching (%20=(%34),%34=(%20)) targets: 51=%27,132=%39 INNER
JoinCond: (%42) First: %18 Second: %33 on %20=%34,
%33=From %33:%37 order (%34) matches (%34=1) matching (%20=(%34),%34=(%20)) targets: 132=%39
Source: %39 Target=132 Indexes=[('30)176],
%34=SqlCopy Name=E %34 Domain 139 From:%33 copy from 153,
%35=SqlCopy Name=K %35 Domain 195 From:%33 copy from 208,
%36=SqlCopy Name=M %36 Domain 139 From:%33 copy from 232,
%37=Domain TABLE (%34,%35,%36) Display=3
[%34,Domain INTEGER],[%35,Domain CHAR],[%36,Domain INTEGER],
%38=Domain TABLE (%34,%35,%36)[%34,Domain INTEGER],[%35,Domain CHAR],[%36,Domain INTEGER],
%39=TableRowSet %39:%38 key (%34) order (%34) matches (%34=1) From: %33 SRow:(153,208,232)
Target:132 Indexes: [(%34)=176],
%40=Domain TABLE (%20,%22,%24,%35,%36[%34) Display=5
[%20,Domain INTEGER],[%22,CHAR],[%24,CHAR],[%35,Domain CHAR],[%36,Domain INTEGER],
[%34,Domain INTEGER],
%41=SqlValueExpr Name= %41 BOOLEAN From:%31 Left:%20 Right:%34 %41(%20=%34),
%42=SqlValueExpr Name= %42 BOOLEAN Left:%20 Right:%34 %42(%20=%34),
%43=OrderedRowSet %43:%28 key (%20) order (%20) targets: 51=%27 Source: %27,
%44=Domain TABLE (%20,%22,%24,%35,%36) Display=5
[%20,Domain INTEGER],[%22,CHAR],[%24,CHAR],[%35,Domain CHAR],[%36,Domain INTEGER],
%45=SelectStatement %45 Union=%14,
%46=Domain TABLE (%20,%22,%24,%35,%36) Display=5
[%20,Domain INTEGER],[%22,CHAR],[%24,CHAR],[%35,Domain CHAR],[%36,Domain INTEGER],
%47=Domain TABLE (%20,%22,%24,%35,%36)[%20,Domain INTEGER],[%22,CHAR],[%24,CHAR],
[%35,Domain CHAR],[%36,Domain INTEGER],
%48=SelectStatement %48 Union=#1)}
```

After instancing, the 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

<sup>39</sup> The illustration omits the framing objects `0... themselves..

June 2022

this has resulted in the OrderedRowSet from the framing being short-circuited at %43. The match %20=1 was applied as %34=1 in some rowsets because of the matching columns %20=%34.

```
SQL> select e, f, m from VU where e=1
|-|---|-----|
|E|F|  M|
|-|---|-----|
|1|Joe|12500|
|-|---|-----|
SQL>
```

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, 29 Jun 2022 19:40:13 GMT
Returning ETag: "23,155,155"
--> 1 rows
```

This time, the ETag is for the tablerow that was returned. The final parts of test 22 performs 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

```
{(-538=TABLE,
-503=NULL,
23=VirtualTable Name=(e int, f char, g char) 23 Domain `6 Definer=-502 Ppos=23:`6 RestView=51,
51=RestView Name=WU 51 Domain `7 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
UsingTableRowSet: _ ViewTable:23,
#1=SqlInsert #1 Target: #13 Value: %12 Columns: [],
#13=From #13:%10 targets: 51=%8 Source: %8 Target=51,
#16=#22,
#22=SqlRow #22 Domain %13,
#23=3,
#25=Fred,
#32=Bloggs,...,
%0=RestView Name=WU %0 Domain %7 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
UsingTableRowSet: _ ViewTable:23,
%1=SqlValue Name=E %1 INTEGER From:23,
%3=SqlValue Name=F %3 CHAR From:23,
%5=SqlValue Name=G %5 CHAR From:23,
%7=Domain TABLE (%1,%3,%5)[%1,INTEGER],[%3,CHAR],[%5,CHAR] structure=23,
%8=RestRowSet %8:%9 targets: 51=%8 SRow:() http://localhost:8180/A/A/D RemoteCols:(%1,%3,%5)
RemoteNames:(%1=E,%3=F,%5=G),
%9=Domain TABLE (%1,%3,%5) Display=3[%1,INTEGER],[%3,CHAR],[%5,CHAR],
%10=Domain TABLE (%1,%3,%5) Display=3[%1,INTEGER],[%3,CHAR],[%5,CHAR],
%11=Domain TABLE (%1,%3,%5)[%1,INTEGER],[%3,CHAR],[%5,CHAR],
%12=SqlRowSet %12:%14 targets: 51=%8 SqlRows [#22],
%13=Domain ROW (#23,#25,#32)[#23,INTEGER],[#25,CHAR],[#32,CHAR],
%14=Domain TABLE (%1,%3,%5) Display=3[%1,INTEGER],[%3,CHAR],[%5,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: Wed, 29 Jun 2022 19:51:34 GMT
Returning ETag: "23,320,320"
Recording ETag "23,320,320"
--> 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>
```



June 2022

But we can verify that table D on database A now has all three rows<sup>40</sup>:

```
http://localhost:8180/A/A/D
HTTP GET /A/A/D
Received If-Unmodified-Since: Wed, 29 Jun 2022 19:56:44 GMT
Returning ETag: "23,_,320"
--> 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,
-505=CONTENT,
-503=NULL,
139=Domain INTEGER,
195=Domain CHAR,
#8=From #8:%42 matches (%16=2,%30=2) matching (%16=(%30),%30=(%16)) targets: 51=%23,132=%35
Source: %27 Indexes=[(`30)176],
#17=Elizabeth,
#36=SqlValueExpr Name= #36 BOOLEAN Left:%16 Right:#37 #36(%16=#37),
#37=2,
%0=View Name=VU %0 Domain %40 Definer=-502 Ppos=362 ViewDef view VU as select * from WU natural
join HU Ppos: 362 Result %10,...,
%1=SqlValue Name=E %1 INTEGER From:23,
%3=SqlValue Name=F %3 CHAR From:23,
%5=SqlValue Name=G %5 CHAR From:23,
%7=Domain TABLE (%1,%3,%5)[%1,INTEGER],[%3,CHAR],[%5,CHAR] structure=23,
%10=SelectRowSet %10:%40 matching (%16=(%30),%30=(%16)) targets: 51=%23,132=%35 Source: %27,
%11=SqlStar Name=* %11 CONTENT,
%12=Domain TABLE (%11)[%11,CONTENT],
%14=From %14:%25 order (%16) matches (%16=2) matching (%16=(%30),%30=(%16)) targets: 51=%23
Source: %23 Target=51,
%15=RestView Name=WU %15 Domain %22 Definer=-502 Ppos=51 ViewDef Ppos: 51 Result _
UsingTableRowSet: _ ViewTable:23,
%16=SqlCopy Name=E %16 Domain 139 From:%29 Alias= copy from 153,
%18=SqlValue Name=F %18 CHAR From:23,
%20=SqlValue Name=G %20 CHAR From:23,
%22=Domain TABLE (%16,%18,%20)[%16,INTEGER],[%18,CHAR],[%20,CHAR] structure=23,
%23=RestRowSet %23:%24 matches (%16=2) targets: 51=%23
Assigns:(UpdateAssignment Vbl: %18 Val: #17=True) SRow:() http://localhost:8180/A/A/D
RemoteCols: (%16,%18,%20) RemoteNames: (%16=E,%18=F,%20=G),
%24=Domain TABLE (%16,%18,%20)[%16,INTEGER],[%18,CHAR],[%20,CHAR],
%25=Domain TABLE (%16,%18,%20) Display=3[%16,INTEGER],[%18,CHAR],[%20,CHAR],
%26=Domain TABLE (%16,%18,%20)[%16,INTEGER],[%18,CHAR],[%20,CHAR],
%27=JoinRowSet %27:%36 matching (%16=(%30),%30=(%16)) targets: 51=%23,132=%35 INNER
JoinCond: (%38) First: %14 Second: %29 on %16=%30,
%29=From %29:%33 order (%30) matches (%30=2) matching (%16=(%30),%30=(%16)) targets: 132=%35
Source: %35 Target=132 Indexes=[(`30)176],
%30=SqlCopy Name=E %30 Domain 139 From:%29 copy from 153,
%31=SqlCopy Name=K %31 Domain 195 From:%29 copy from 208,
%32=SqlCopy Name=M %32 Domain 139 From:%29 copy from 232,
%33=Domain TABLE (%30,%31,%32) Display=3
[%30,Domain INTEGER],[%31,Domain CHAR],[%32,Domain INTEGER],
%34=Domain TABLE (%30,%31,%32)[%30,Domain INTEGER],[%31,Domain CHAR],[%32,Domain INTEGER],
%35=TableRowSet %35:%34 key (%30) order (%30) matches (%30=2) From: %29 SRow:(153,208,232)
Target:132 Indexes: [(%30)=176],
%36=Domain TABLE (%16,%18,%20,%31,%32|%30) Display=5
[%16,Domain INTEGER],[%18,CHAR],[%20,CHAR],[%31,Domain CHAR],
```

<sup>40</sup> 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.

```

[%32,Domain INTEGER],[%30,Domain INTEGER],
%37=SqlValueExpr Name= %37 BOOLEAN From:%27 Left:%16 Right:%30 %37(%16=%30),
%38=SqlValueExpr Name= %38 BOOLEAN Left:%16 Right:%30 %38(%16=%30),
%39=OrderedRowSet %39:%24 key (%16) order (%16) targets: 51=%23 Source: %23,
%40=Domain TABLE (%16,%18,%20,%31,%32) Display=5
[%16,Domain INTEGER],[%18,CHAR],[%20,CHAR],[%31,Domain CHAR],[%32,Domain INTEGER],
%41=SelectStatement %41 Union=%10,
%42=Domain TABLE (%16,%18,%20,%31,%32) Display=5
[%16,Domain INTEGER],[%18,CHAR],[%20,CHAR],[%31,Domain CHAR],[%32,Domain INTEGER],
%43=Domain TABLE (%16,%18,%20,%31,%32)
[%16,Domain INTEGER],[%18,CHAR],[%20,CHAR],[%31,Domain CHAR],[%32,Domain INTEGER],
%44=UpdateSearch %44 Target: #8)}

```

We see that again the OrderedRowSet has been short-circuited, the updateSearch statement's source is #8 which has targets %23 and %35, and the update assignments have been added to the appropriate target rowSet %23. 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<sup>41</sup>.

```

http://localhost:8180/A/A/D/E=2
HTTP GET /A/A/D/E=2
Received If-Unmodified-Since: Wed, 29 Jun 2022 19:57:44 GMT
Returning ETag: "23,185,185"
--> 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: Wed, 29 Jun 2022 19:57:44 GMT
Returning ETag: "23,185,371"
Recording ETag "23,185,371"
--> OK

```

In Database A we check this:

```

SQL> table D
|-----|-----|
|E|F      |G      |
|-----|-----|
|1|Joe     |Soap   |
|2|Elizabeth|Boop   |
|3|Fred    |Bloggs |
|-----|-----|
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 statement is similar, so really the first difference occurs in the insert statement, which we run in an explicit transaction so that we see the difference in this model:

```

begin transaction
insert into w values(3,'Fred','Bloggs')

```

When this runs, we get

---

<sup>41</sup> See the previous footnote.



June 2022

```
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: Wed, 29 Jun 2022 20:11:52 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

```
{(!=Post D http://localhost:8180/A/A/D insert into D values(3,'Fred','Bloggs'); 51)}
```

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: 408
Received If-Unmodified-Since: Wed, 29 Jun 2022 19:11:52 GMT
Returning ETag: "23,320,320"
Recording ETag "23,320,320"
--> 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.

```
SQL> create table H (e int primary key, k char, m int)
SQL> insert into H values (1,'Cleaner',12500), (2,'Manager',31400)
2 records affected in B
SQL> create view V as select * from w natural join H
SQL> begin transaction
SQL-T>update v set f='Elizabeth' where e=2
0 records in transaction B
SQL-T>
RoundTrip HEAD http://localhost:8180/A/A/D
HTTP HEAD /A/A/D/E=2
Received If-Unmodified-Since: Wed, 29 Jun 2022 20:20:32 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: Wed, 29 Jun 2022 20:20:32 GMT
Returning ETag: "23,185,185"
--> 1 rows
Response ETag: 23,185,185
```

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. (We can verify this by checking the contents of table D on database A in another window.) On Commit once again there is a Post record

```
{(!=Post D http://localhost:8180/A/A/D update D set F='Elizabeth' where E=2 51)}
```

and when this record is committed we get

```
SQL-T>commit
SQL>
```

```

HTTP HEAD /A/A/D
Received If-Match: 23,185,185
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: 408
Received If-Unmodified-Since: Wed, 29 Jun 2022 19:20:32 GMT
Returning ETag: "23,185,371"
Recording ETag "23,185,371"
--> OK

```

(Because of the debugging delay the server has sent a Request Timeout status, asking for the client to close the connection.)

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 WV 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	Type	Affects
122	PTable VU	PTable	122
129	Domain CHAR	PDomain	129
142	PColumn3 D for 122(0)[129]	PColumn3	142
164	PIndex VU on 122(142) PrimaryKey	PIndex	164
182	Domain INTEGER	PDomain	182
196	PColumn3 K for 122(1)[182]	PColumn3	196
219	PColumn3 U for 122(2)[129]	PColumn3	219
260	Record 260[122]: 142=B,196=4,219=http://localhost:8180/DB/DB/t	Record	260
334	Record 334[122]: 142=C,196=1,219=http://localhost:8180/DC/DC/u	Record	334
408	PTable (E int, D char, K int, F char)	PTable	408
445	PRestView2 WW(408) using 12	RestView2	445
..	..		

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 Name=WW 445 Domain `16 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
    UsingTableRowSet: `23 ViewTable:408}
```

with a framing for its compiled objects:

```
{Framing
`6 Domain ROW (142)[142,Domain CHAR],
`9 SqlValue Name=E `9 135 From:408,
`15 SqlValue Name=F `15 CHAR From:408,
`16 Domain (E int, D char, K int, F char)
    VIEW (`9,`18,`19,`15)[`9,135],[`18,CHAR],[`19,135],[`15,CHAR] structure=408,
`17 From `17:`21 targets: 122=`23 Source: `23 Target=122 Indexes=[(`18)164],
`18 SqlCopy Name=D `18 Domain 129 From:`17 copy from 142,
`19 SqlCopy Name=K `19 Domain 182 From:`17 copy from 196,
`20 SqlCopy Name=U `20 Domain 129 From:`17 copy from 219,
`21 Domain TABLE (`18,`19,`20) Display=3[`18,Domain CHAR],[`19,Domain 135],[`20,Domain CHAR],
`22 Domain TABLE (`18,`19,`20) [`18,Domain CHAR],[`19,Domain 135],[`20,Domain CHAR],
`23 TableRowSet `23:`22 From: `17 SRow:(142,196,219) Target:122 Indexes: [(`18)=164]}}
```

We note that the columns of the RestView WW include two local columns D and K coming from the UsingTableRowset `23. The remaining columns are (only) in the remote view 408, 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, modified<sup>42</sup> as highlighted):

```
{(-539=ROW,
-538=TABLE,
-505=CONTENT,
-503=NULL,
129=Domain CHAR,
182=Domain INTEGER,
```

<sup>42</sup> Instances are not shared, so that modified versions can have same defining position.

```

408=VirtualTable Name=(E int, D char, K int, F char) 408 TABLE Definer=-502 Ppos=408:-538
    RestView=445,
#1=SelectRowSet #1:%25 where (#25) targets: %22=%22 Source: #15,
#8=SqlStar Name=* #8 CONTENT,
#15=From #15:%23 matches (%15=1) targets: %22=%22 Source: %22 Target=445,
#25=SqlValueExpr Name= #25 BOOLEAN Left:%15 Right:#26 #25(%15=#26),
#26=1,
%0=Domain TABLE (#8)[#8,CONTENT],
%1=RestView Name=WW %1 Domain %12 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
    UsingTableRowSet: %19 ViewTable:408,
%2=Domain ROW (142)[142,Domain CHAR],
%5=SqlValue Name=E %5 INTEGER From:408,
%11=SqlValue Name=F %11 CHAR From:408,
%12=Domain (E int, D char, K int, F char)
    VIEW (%5,%14,%15,%11)[%5,INTEGER],[%14,CHAR],[%15,INTEGER],[%11,CHAR] structure=408,
%13=From %13:%17 targets: 122=%19 Source: %19 Target=122 Indexes=[('18)164],
%14=SqlCopy Name=D %14 Domain 129 From:%13 copy from 142,
%15=SqlCopy Name=K %15 Domain 182 From:%13 copy from 196,
%16=SqlCopy Name=U %16 Domain 129 From:%13 copy from 219,
%17=Domain TABLE (%14,%15,%16) Display=3
    [%14,Domain CHAR],[%15,Domain INTEGER],[%16,Domain CHAR],
%18=Domain TABLE (%14,%15,%16)[%14,Domain CHAR],[%15,Domain INTEGER],[%16,Domain CHAR],
%19=TableRowSet %19:%18 key (%14) where (#25) matches (%15=1) From: %13 SRow:(142,196,219)
    Target:122 Indexes: [(%14)=164],
%20=RestRowSet %20:%21 targets: 445=%20 SRow:() RemoteCols:(%5,%11)
    RemoteNames:(%5=E,%11=F) UsingTableRowSet %19,
%21=Domain TABLE (%5,%14,%15,%11)
    [%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%22=RestRowSetUsing %22:%21 matches (%15=1) targets: 445=%22 SRow:(142,196,219)
    ViewDomain: %21 Template: %20 UsingTableRowSet:%19 UriCol:%16,
%23=Domain TABLE (%5,%14,%15,%11) Display=4
    [%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%24=Domain TABLE (%5,%11,%14,%15)
    [%5,INTEGER],[%11,CHAR],[%14,Domain CHAR],[%15,Domain INTEGER],
%25=Domain TABLE (%5,%14,%15,%11) Display=4
    [%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%26=SelectStatement %26 Union=#1)}

```

The REST operation is managed by the RestRowSet %20. Notice that it does not have the where-condition in this test.

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,-1,159"
--> 3 rows
Response ETag: 23,-1,159

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
```

```

{(...,
408=VirtualTable Name=(E int, D char, K int, F char) 408 TABLE Definer=-502 Ppos=408:-538
RestView=445,
#1=SelectRowSet #1:%25 where (#25) targets: %22=%22 Source: #15,
#8=SqlStar Name=* #8 CONTENT,
#15=From #15:%23 matches (%5=3) targets: %22=%22 Source: %22 Target=445,
#25=SqlValueExpr Name= #25 BOOLEAN Left:%5 Right:#26 #25(%5=#26),
#26=3,...,
%0=Domain TABLE (#8)[#8,CONTENT],
%1=RestView Name=WW %1 Domain %12 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
    UsingTableRowSet: %19 ViewTable:408,
%2=Domain ROW (142)[142,Domain CHAR],

```

```

%5=SqlValue Name=E %5 INTEGER From:408,
%11=SqlValue Name=F %11 CHAR From:408,
%12=Domain (E int, D char, K int, F char)
VIEW (%5,%14,%15,%11)[%5,INTEGER],[%14,CHAR],[%15,INTEGER],[%11,CHAR] structure=408,
%13=From %13:%17 targets: 122=%19 Source: %19 Target=122 Indexes=[(`18)164],
%14=SqlCopy Name=D %14 Domain 129 From:%13 copy from 142,
%15=SqlCopy Name=K %15 Domain 182 From:%13 copy from 196,
%16=SqlCopy Name=U %16 Domain 129 From:%13 copy from 219,
%17=Domain TABLE (%14,%15,%16) Display=3
[%14,Domain CHAR],[%15,Domain INTEGER],[%16,Domain CHAR],
%18=Domain TABLE (%14,%15,%16)[%14,Domain CHAR],[%15,Domain INTEGER],[%16,Domain CHAR],
%19=TableRowSet %19:%18 From: %13 SRow:(142,196,219) Target:122 Indexes: [(%14)=164],
%20=RestRowSet %20:%21 matches (%5=3) targets: 445=%20 SRow:() RemoteCols:(%5,%11)
RemoteNames:(%5=E,%11=F) UsingTableRowSet %19,
%21=Domain TABLE (%5,%14,%15,%11)
[%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%22=RestRowSetUsing %22:%21 matches (%5=3) targets: 445=%22 SRow:(142,196,219)
ViewDomain: %21 Template: %20 UsingTableRowSet:%19 UriCol:%16,
%23=Domain TABLE (%5,%14,%15,%11) Display=4
[%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%24=Domain TABLE (%5,%11,%14,%15)
[%5,INTEGER],[%11,CHAR],[%14,Domain CHAR],[%15,Domain INTEGER],
%25=Domain TABLE (%5,%14,%15,%11) Display=4
[%5,INTEGER],[%14,Domain CHAR],[%15,Domain INTEGER],[%11,CHAR],
%26=SelectStatement %26 Union=#1)}
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where E=3
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where E=3
Returning ETag: "23,117,117"
--> 1 rows
Response ETag: 23,117,117
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where E=3
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where E=3
--> 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 %20, 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:%26 targets: %23=%23 Source: #22,
#8=SqlFunction Name=COUNT #8 INTEGER From:#1 Await (#1) COUNT(%6),
#22=From #22:%28 targets: %23=%23 Source: %23 Target=445,...,
%0=Domain ROW (%6)[%6,CONTENT],
%1=Domain TABLE (#8)[#8,INTEGER] Aggs (#8),
%2=RestView Name=WW %2 Domain %13 Definere=-502 Ppos=445 ViewDef Ppos: 445 Result _
UsingTableRowSet: %20 ViewTable:408,
%3=Domain ROW (142)[142,Domain CHAR],
%6=SqlValue Name=E %6 INTEGER From:408,
%12=SqlValue Name=F %12 CHAR From:408,
%13=Domain (E int, D char, K int, F char)
VIEW (%6,%15,%16,%12)[%6,INTEGER],[%15,CHAR],[%16,INTEGER],[%12,CHAR] structure=408,
%14=From %14:%18 targets: 122=%20 Source: %20 Target=122 Indexes=[(`18)164],
%15=SqlCopy Name=D %15 Domain 129 From:%14 copy from 142,
%16=SqlCopy Name=K %16 Domain 182 From:%14 copy from 196,
%17=SqlCopy Name=U %17 Domain 129 From:%14 copy from 219,
%18=Domain TABLE (%15,%16,%17) Display=3
[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%19=Domain TABLE (%15,%16,%17)[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%20=TableRowSet %20:%19 From: %14 SRow:(142,196,219) Target:122 Indexes: [(%15)=164],
%21=RestRowSet %21:%27 targets: 445=%21 SRow:() RemoteCols:(%6,%12)
RemoteNames:(%6=E,%12=F) UsingTableRowSet %20,
%22=Domain TABLE (%6|%15,%16,%12) Display=1
[%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],[%12,CHAR],
%23=RestRowSetUsing %23:%29 targets: 445=%23 SRow:(142,196,219) ViewDomain: %29

```

```

Template: %21 UsingTableRowSet:%20 UriCol:%17,
%24=Domain TABLE (%6,%15,%16,%12) Display=4
[%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],[%12,CHAR],
%25=Domain TABLE (%6,%12,%15,%16)
[%6,INTEGER],[%12,CHAR],[%15,Domain CHAR],[%16,Domain INTEGER],
%26=Domain TABLE (#8)[#8,INTEGER] Aggs (#8),
%27=Domain TABLE (#8)[#8,INTEGER] Aggs (#8),
%28=Domain TABLE (#8)[#8,INTEGER],
%29=Domain TABLE (#8)[#8,INTEGER],
%30=SelectStatement %30 Union=#1}

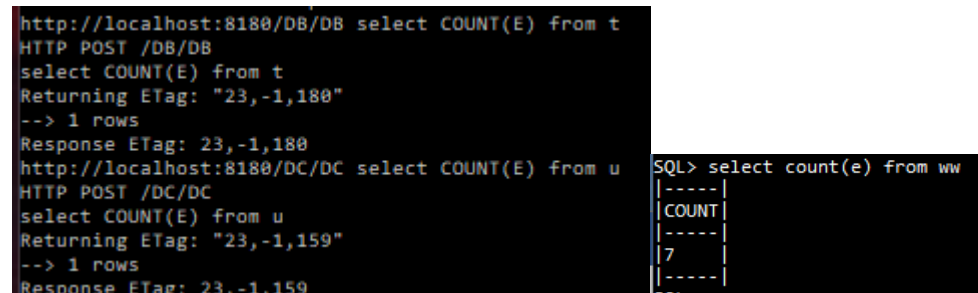
```

When traversal starts of the SelectRowSet #1, the SelectRowSet.Build() takes change. Place a breakpoint in RestRowSet.Build() at kline 6924 in RowSet.cs.

Then we traverse the UsingTableRowSet %21, 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) from t
HTTP POST /DB/DB
select COUNT(E) from t
Returning ETag: "23,-1,180"
--> 1 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select COUNT(E) from u
HTTP POST /DC/DC
select COUNT(E) from u
Returning ETag: "23,-1,159"
--> 1 rows
Response ETag: 23,-1,159
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 for the aggregations in addition to the results for contributors, and RestRowSetBuild accumulates them in the global result<sup>43</sup> 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:%26 where (#30) targets: %23=%23 Source: #20,
#8=SqlFunction Name=MAX #8 CHAR From:#1 Await (#1) MAX(%12),
#20=From #20:%28 where (#30) targets: %23=%23 Source: %23 Target=445,
#30=SqlValueExpr Name= #30 BOOLEAN Left:%6 Right:#31 #30(%6>#31),
#31=4,...,
%0=Domain ROW (%12)[%12,Domain UNION],
%1=Domain TABLE (#8)[#8,CHAR] Aggs (#8),
%2=RestView Name=WW %2 Domain %13 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
UsingTableRowSet: %20 ViewTable:408,
%3=Domain ROW (142)[142,Domain CHAR],
%6=SqlValue Name=E %6 INTEGER From:408,
%12=SqlValue Name=F %12 CHAR From:408,
%13=Domain (E int, D char, K int, F char)
VIEW (%6,%15,%16,%12)[%6,INTEGER],[%15,CHAR],[%16,INTEGER],[%12,CHAR] structure=408,
%14=From %14:%18 targets: 122=%20 Source: %20 Target=122 Indexes=[(`18)164],
%15=SqlCopy Name=D %15 Domain 129 From:%14 copy from 142,
%16=SqlCopy Name=K %16 Domain 182 From:%14 copy from 196,
%17=SqlCopy Name=U %17 Domain 129 From:%14 copy from 219,
%18=Domain TABLE (%15,%16,%17) Display=3
[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%19=Domain TABLE (%15,%16,%17)[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%20=TableRowSet %20:%19 From: %14 SRow:(142,196,219) Target:122 Indexes: [(%15)=164],

```

<sup>43</sup> The code to do this is in Domain.Coerce (currently at line 2880 of Domain.cs). For full details, see the Pyrrho manual, section 8.8.9.



```

%21=RestRowSet %21:%27 where (#30) targets: 445=%21 SRow:() RemoteCols:(%6,%12)
RemoteNames:(%6=E,%12=F) UsingTableRowSet %20,
%22=Domain TABLE (%12,%6,%15,%16) Display=1
[%12,CHAR],[%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],
%23=RestRowSetUsing %23:%29 where (#30) targets: 445=%23 SRow:(142,196,219) ViewDomain: %29
Template: %21 UsingTableRowSet:%20 UrlCol:%17,
%24=Domain TABLE (%12,%6,%15,%16) Display=4
[%12,CHAR],[%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],
%25=Domain TABLE (%6,%12,%15,%16)
[%6,INTEGER],[%12,CHAR],[%15,Domain CHAR],[%16,Domain INTEGER],
%26=Domain TABLE (#8)[#8,CHAR] Aggs (#8),
%27=Domain TABLE (#8)[#8,CHAR] Aggs (#8),
%28=Domain TABLE (#8)[#8,CHAR],
%29=Domain TABLE (#8)[#8,CHAR],
%30=SelectStatement %30 Union=#1)}

http://localhost:8180/DB/DB select MAX(F) from t where (E>4)
HTTP POST /DB/DB
select MAX(F) from t where (E>4)
--> 1 rows
http://localhost:8180/DC/DC select MAX(F) from u where (E>4)
HTTP POST /DC/DC
select MAX(F) from u where (E>4)
--> 1 rows

```

```

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:%30 groupSpec: #47 groupings (#50) GroupCols(%14) targets: %26=%26
Source: #38,
#8=SqlFunction Name=SUM #8 INTEGER From:#1 Await (#1) SUM(%8),
#14=SqlValueExpr Name= #14 Domain %23 From:#1 Left:#8 Right:#15 #14(#8+#15),
#15=SqlFunction Name=CHAR_LENGTH #15 INTEGER From:#1 CHAR_LENGTH(%14),
#38=From #38:%33 targets: %26=%26 Source: %26 Target=445,
#47=GroupSpecification #47(#50),
#50=Grouping #50 Domain %29 GROUP (%14=0),
%0=Domain ROW (%8)[%8,CONTENT],
%1=Domain ROW (%14)[%14,CHAR],
%2=Domain UNION Aggs (#8),
%3=Domain TABLE (#14,%14)[%14,Domain INTEGER Aggs (#8)],[%14,CHAR] Aggs (#8),
%4=RestView Name=WW %4 Domain %15 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
UsingTableRowSet: %22 ViewTable:408,
%5=Domain ROW (142)[142,Domain CHAR],
%8=SqlValue Name=E %8 INTEGER From:408,
%14=SqlValue Name=F %14 CHAR From:408,
%15=Domain (E int, D char, K int, F char)
VIEW (%8,%17,%18,%14)[%8,INTEGER],[%17,CHAR],[%18,INTEGER],[%14,CHAR] structure=408,
%16=From %16:%20 targets: 122=%22 Source: %22 Target=122 Indexes=[(%18)=164],
%17=SqlCopy Name=D %17 Domain 129 From:%16 copy from 142,
%18=SqlCopy Name=K %18 Domain 182 From:%16 copy from 196,
%19=SqlCopy Name=U %19 Domain 129 From:%16 copy from 219,
%20=Domain TABLE (%17,%18,%19) Display=3
[%17,Domain CHAR],[%18,Domain INTEGER],[%19,Domain CHAR],
%21=Domain TABLE (%17,%18,%19)[%17,Domain CHAR],[%18,Domain INTEGER],[%19,Domain CHAR],
%22=TableRowSet %22:%21 From: %16 SRow:(142,196,219) Target:122 Indexes: [(%17)=164],
%23=Domain INTEGER Aggs (#8),
%24=RestRowSet %24:%32 GroupCols(%14) targets: 445=%24 SRow:() RemoteCols:(%8,%14)
RemoteNames:(%8=E,%14=F) UsingTableRowSet %22,
%25=Domain TABLE (%8,%14,%17,%18) Display=2
[%8,INTEGER],[%14,CHAR],[%17,Domain CHAR],[%18,Domain INTEGER],
%26=RestRowSetUsing %26:%34 targets: 445=%26 SRow:(142,196,219) ViewDomain: %34
Template: %24 UsingTableRowSet:%22 UrlCol:%19,
%27=Domain TABLE (%8,%14,%17,%18) Display=4
[%8,INTEGER],[%14,CHAR],[%17,Domain CHAR],[%18,Domain INTEGER],
%28=Domain TABLE (%8,%14,%17,%18)
[%8,INTEGER],[%14,CHAR],[%17,Domain CHAR],[%18,Domain INTEGER],
%29=Domain ROW (%14)[%14,CHAR],
%30=Domain TABLE (#14,%14)[%14,Domain INTEGER Aggs (#8)],[%14,CHAR] Aggs (#8),
%31=Domain ROW (%14)[%14,CHAR],
%32=Domain TABLE (#14,%14)[%14,Domain INTEGER Aggs (#8)],[%14,CHAR] Aggs (#8),
%33=Domain TABLE (#14,%14)[%14,Domain INTEGER Aggs (#8)],[%14,CHAR],

```

June 2022

```
%34=Domain TABLE (#14,%14,#8)[#14,Domain INTEGER Aggs (#8)],[%14,CHAR],[#8,INTEGER],
%35=SelectStatement %35 Union=#1)}
```

The JSON documents returned are

```
"[{\"Col0\": 11, \"F\": 'Sechs', \"$#9\": {\"0\": 6}},
  {\"Col0\": 9, \"F\": 'Six', \"$#9\": {\"0\": 6}},
  {\"Col0\": 8, \"F\": 'Three', \"$#9\": {\"0\": 3}},
  {\"Col0\": 8, \"F\": 'Vier', \"$#9\": {\"0\": 4}}]"
"([{\"Col0\": 11, \"F\": 'Ate', \"$#9\": {\"0\": 8}},
  {\"Col0\": 9, \"F\": 'Five', \"$#9\": {\"0\": 5}},
  {\"Col0\": 8, \"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 from t group by F
HTTP POST /DB/DB
select (SUM(E)+CHAR_LENGTH(F)),F from t group by F
Returning ETag: "23,-1,180"
--> 4 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select (SUM(E)+CHAR_LENGTH(F)),F from u group by F
HTTP POST /DC/DC
select (SUM(E)+CHAR_LENGTH(F)),F from u group by F
Returning ETag: "23,-1,159"
--> 3 rows

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:%27 groupSpec: #41 groupings (#44) GroupCols(#18) targets: %23=%23
Source: #32,
#8=SqlFunction Name=COUNT #8 INTEGER From:#1 Await (#1) COUNT(#14),
#14=1,
#18=SqlValueExpr Name=K2 #18 Domain 182 From:#1 Left:%16 Right:#19 #18(%16/#19),
#19=2,
#32=From #32:%30 targets: %23=%23 Source: %23 Target=445,
#41=GroupSpecification #41(#44),
#44=Grouping #44 Domain %26 GROUP (#18=0),...,
%0=Domain ROW (%16)[%16,CONTENT],
%1=Domain TABLE (#8,#18)[#8,INTEGER],[#18,Domain INTEGER] Aggs (#8),
%2=RestView Name=WW %2 Domain %13 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
  UsingTableRowSet: %20 ViewTable:408,
%3=Domain ROW (142)[142,Domain CHAR],
%6=SqlValue Name=E %6 INTEGER From:408,
%12=SqlValue Name=F %12 CHAR From:408,
%13=Domain (E int, D char, K int, F char)
  VIEW (%6,%15,%16,%12)[%6,INTEGER],[%15,CHAR],[%16,INTEGER],[%12,CHAR] structure=408,
%14=From %14:%18 targets: 122=%20 Source: %20 Target=122 Indexes=[(`18)164],
%15=SqlCopy Name=D %15 Domain 129 From:%14 copy from 142,
%16=SqlCopy Name=K %16 Domain 182 From:%14 copy from 196,
%17=SqlCopy Name=U %17 Domain 129 From:%14 copy from 219,
%18=Domain TABLE (%15,%16,%17) Display=3
  [%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%19=Domain TABLE (%15,%16,%17)[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
%20=TableRowSet %20:%19 From: %14 SRow:(142,196,219) Target:122 Indexes: [(%15)=164],
%21=RestRowSet %21:%29 GroupCols(#18) targets: 445=%21 SRow:() RemoteCols:(%6,%12)
  RemoteNames:(%6=E,%12=F) UsingTableRowSet %20,
%22=Domain TABLE (%16,%6,%15,%12) Display=1
  [%16,Domain INTEGER],[%6,INTEGER],[%15,Domain CHAR],[%12,CHAR],
%23=RestRowSetUsing %23:%31 targets: 445=%23 SRow:(142,196,219) ViewDomain: %31
  Template: %21 UsingTableRowSet:%20 UrlCol:%17,
%24=Domain TABLE (%16,%6,%15,%12) Display=4
  [%16,Domain INTEGER],[%6,INTEGER],[%15,Domain CHAR],[%12,CHAR],
```



```

%25=Domain TABLE (%6,%12,%15,%16)
    [%6,INTEGER],[%12,CHAR],[%15,Domain CHAR],[%16,Domain INTEGER],
%26=Domain ROW (#18)[#18,Domain INTEGER],
%27=Domain TABLE (#8,#18)[#8,INTEGER],[#18,Domain INTEGER] Aggs (#8),
%28=Domain ROW (#18)[#18,Domain INTEGER],
%29=Domain TABLE (#8,#18)[#8,INTEGER],[#18,Domain INTEGER] Aggs (#8),
%30=Domain TABLE (#8,#18)[#8,INTEGER],[#18,Domain INTEGER],
%31=Domain TABLE (#8,#18)[#8,INTEGER],[#18,Domain INTEGER],
%32=SelectStatement %32 Union=#1)}

http://localhost:8180/DB/DB select COUNT(*),(4/2) as K2 from t group by K2
HTTP POST /DB/DB
select COUNT(*),(4/2) as K2 from t group by K2
Returning ETag: "23,-1,180"
--> 1 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select COUNT(*),(1/2) as K2 from u group by K2
HTTP POST /DC/DC
select COUNT(*),(1/2) as K2 from u group by K2
Returning ETag: "23,-1,159"
--> 1 rows
Response ETag: 23,-1,159
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:%26 targets: %23=%23 Source: #20,
  #8=SqlFunction Name=AVG #8 NUMERIC From:#1 Await (#1) AVG(%6),
  #20=From #20:%28 targets: %23=%23 Source: %23 Target=445,
  %0=Domain ROW (%6)[%6,CONTENT],
  %1=Domain TABLE (#8)[#8,NUMERIC] Aggs (#8),
  %2=RestView Name=WW %2 Domain %13 Definier=-502 Ppos=445 ViewDef Ppos: 445 Result _
UsingTableRowSet: %20 ViewTable:408,
  %3=Domain ROW (142)[142,Domain CHAR],
  %6=SqlValue Name=E %6 INTEGER From:408,
  %12=SqlValue Name=F %12 CHAR From:408,
  %13=Domain (E int, D char, K int, F char) VIEW
(%6,%15,%16,%12)[%6,INTEGER],[%15,CHAR],[%16,INTEGER],[%12,CHAR] structure=408,
  %14=From %14:%18 targets: 122=%20 Source: %20 Target=122 Indexes=[(`18)164],
  %15=SqlCopy Name=D %15 Domain 129 From:%14 copy from 142,
  %16=SqlCopy Name=K %16 Domain 182 From:%14 copy from 196,
  %17=SqlCopy Name=U %17 Domain 129 From:%14 copy from 219,
  %18=Domain TABLE (%15,%16,%17) Display=3
    [%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
  %19=Domain TABLE (%15,%16,%17)[%15,Domain CHAR],[%16,Domain INTEGER],[%17,Domain CHAR],
  %20=TableRowSet %20:%19 From: %14 SRow:(142,196,219) Target:122 Indexes: [(%15)=164],
  %21=RestRowSet %21:%27 targets: 445=%21 SRow:() RemoteCols:(%6,%12)
    RemoteNames:(%6=E,%12=F) UsingTableRowSet %20,
  %22=Domain TABLE (%6|%15,%16,%12) Display=1
    [%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],[%12,CHAR],
  %23=RestRowSetUsing %23:%29 targets: 445=%23 SRow:(142,196,219) ViewDomain: %29
    Template: %21 UsingTableRowSet:%20 UrlCol:%17,
  %24=Domain TABLE (%6,%15,%16,%12) Display=4
    [%6,INTEGER],[%15,Domain CHAR],[%16,Domain INTEGER],[%12,CHAR],
  %25=Domain TABLE (%6,%12,%15,%16)
    [%6,INTEGER],[%12,CHAR],[%15,Domain CHAR],[%16,Domain INTEGER],
  %26=Domain TABLE (#8)[#8,NUMERIC] Aggs (#8),
  %27=Domain TABLE (#8)[#8,NUMERIC] Aggs (#8),
  %28=Domain TABLE (#8)[#8,NUMERIC],
  %29=Domain TABLE (#8)[#8,NUMERIC],
  %30=SelectStatement %30 Union=#1)}

```

This time the documents returned from the remote servers are

```

"[{"AVG": 4.75, "$8": {"4": 19}}]"
"[{"AVG": 5.666666666666667, "$8": {"3": 17}}]"

```

```

Response ETag: 23,-1,180
http://localhost:8180/DB/DB select AVG(E) from t
HTTP POST /DB/DB
select AVG(E) from t
Returning ETag: "23,-1,180"
--> 1 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select AVG(E) from u
HTTP POST /DC/DC
select AVG(E) from u
Returning ETag: "23,-1,159"
--> 1 rows
Response ETag: 23,-1,159

```

```

SQL> Select avg(e) from w
-----
|AVG|
-----
|5.14285714285714|
-----

```

**select sum(e)\*sum(e),d from ww group by d**

```

{(...,
#1=SelectRowSet #1:%31 groupSpec: #38 groupings (#41) GroupCols(%17) targets: %27=%27
Source: #29,
#8=SqlFunction Name=SUM #8 INTEGER From:#1 Await (#1) SUM(%8),
#14=SqlValueExpr Name= #14 Domain %24 From:#1 Left:#8 Right:#15 #14(#8*#15),
#15=SqlFunction Name=SUM #15 INTEGER From:#1 Await (#1) SUM(%8),
#29=From #29:%34 targets: %27=%27 Source: %27 Target=445,
#38=GroupSpecification #38(#41),
#41=Grouping #41 Domain %30 GROUP (%17=0),...,
%0=Domain ROW (%8)[%8,CONTENT],
%1=Domain UNION Aggs (#8,#15),
%2=Domain ROW (%17)[%17,CONTENT],
%3=Domain TABLE (#14,%17)[#14,Domain INTEGER Aggs (#8,#15)],[%17,Domain CHAR] Aggs (#8,#15),
%4=RestView Name=WW %4 Domain %15 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
UsingTableRowSet: %22 ViewTable:408,
%5=Domain ROW (142)[142,Domain CHAR],
%8=SqlValue Name=E %8 INTEGER From:408,
%14=SqlValue Name=F %14 CHAR From:408,
%15=Domain (E int, D char, K int, F char) VIEW
(%8,%17,%18,%14)[%8,INTEGER],[%17,CHAR],[%18,INTEGER],[%14,CHAR] structure=408,
%16=From %16:%20 targets: 122=%22 Source: %22 Target=122 Indexes=[(`18)164],
%17=SqlCopy Name=D %17 Domain 129 From:%16 copy from 142,
%18=SqlCopy Name=K %18 Domain 182 From:%16 copy from 196,
%19=SqlCopy Name=U %19 Domain 129 From:%16 copy from 219,
%20=Domain TABLE (%17,%18,%19) Display=3[%17,Domain CHAR],[%18,Domain INTEGER],[%19,Domain
CHAR],
%21=Domain TABLE (%17,%18,%19)[%17,Domain CHAR],[%18,Domain INTEGER],[%19,Domain CHAR],
%22=TableRowSet %22:%21 From: %16 SRow:(142,196,219) Target:122 Indexes: [(%17)=164],
%23=Domain UNION Aggs (#8,#15),
%24=Domain INTEGER Aggs (#8,#15),
%25=RestRowSet %25:%33 GroupCols(%17) targets: 445=%25 SRow:() RemoteCols:(%8,%14)
RemoteNames:(%8=E,%14=F) UsingTableRowSet %22,
%26=Domain TABLE (%8,%17,%18,%14) Display=2
[%8,INTEGER],[%17,Domain CHAR],[%18,Domain INTEGER],[%14,CHAR],
%27=RestRowSetUsing %27:%35 targets: 445=%27 SRow:(142,196,219) ViewDomain: %35
Template: %25 UsingTableRowSet:%22 UrlCol:%19,
%28=Domain TABLE (%8,%17,%18,%14) Display=4
[%8,INTEGER],[%17,Domain CHAR],[%18,Domain INTEGER],[%14,CHAR],
%29=Domain TABLE (%8,%14,%17,%18)
[%8,INTEGER],[%14,CHAR],[%17,Domain CHAR],[%18,Domain INTEGER],
%30=Domain ROW (%17)[%17,Domain CHAR],
%31=Domain TABLE (#14,%17)
[#14,Domain INTEGER Aggs (#8,#15)],[%17,Domain CHAR] Aggs (#8,#15),
%32=Domain ROW (%17)[%17,Domain CHAR],
%33=Domain TABLE (#14,%17)[#14,Domain INTEGER Aggs (#8,#15)],[%17,Domain CHAR]
Aggs (#8,#15),
%34=Domain TABLE (#14,%17)[#14,Domain INTEGER Aggs (#8,#15)],[%17,Domain CHAR],
%35=Domain TABLE (#14,%17,#8,#15)
[#14,Domain INTEGER Aggs (#8,#15)],[%17,Domain CHAR],[#8,INTEGER],[#15,INTEGER],
%36=SelectStatement %36 Union=#1)}

```

```

http://localhost:8180/DB/DB select (SUM(E)*SUM(E)), 'B' as D from t group by D
HTTP POST /DB/DB
select (SUM(E)*SUM(E)), 'B' as D from t group by D
Returning ETag: "23,-1,180"
--> 1 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select (SUM(E)*SUM(E)), 'C' as D from u group by D
HTTP POST /DC/DC
select (SUM(E)*SUM(E)), 'C' as D from u group by D
Returning ETag: "23,-1,159"
--> 1 rows
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
```

```

{(...,
  477=Table Name=M 477 TABLE Definer=-502 Ppos=477:-538 Indexes:((484)508,(527)552) KeyCols:
(484=True,527=True),
  484=TableColumn 484 Domain 182 Definer=-502 Ppos=484 182 Table=477,
  527=TableColumn 527 Domain 129 Definer=-502 Ppos=527 129 Table=477,
  #1=SelectRowSet #1:%2 matching (%7=(%27),%27=(%7)) targets: 477=%30,%24=%24 Source: #20,
  #10=SqlCopy Name=N #10 Domain 129 From:#33 copy from 527,
  #17=From #17:%25 order (%7) targets: %24=%24 Source: %34 Target=445,
  #20=JoinRowSet #20:%31 matching (%7=(%27),%27=(%7)) targets: 477=%30,%24=%24 INNER
    JoinCond: (%33) First: #17 Second: #33 on %7=%27,
  #33=From #33:%28 order (%27) targets: 477=%30 Source: %30 Target=477
    Indexes=[(#10)552,(%27)508], ..
  %0=Domain ROW (%13)[%13,CONTENT],
  %1=Domain ROW (%10)[%10,CONTENT],
  %2=Domain TABLE (%13,%10)[%13,CHAR],[%10,Domain CHAR],
  %3=RestView Name=WW %3 Domain %14 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
    UsingTableRowSet: %21 ViewTable:408,
  %4=Domain ROW (142)[142,Domain CHAR],
  %7=SqlCopy Name=E %7 Domain 182 From:#33 Alias= copy from 484,
  %13=SqlValue Name=F %13 CHAR From:408,
  %14=Domain (E int, D char, K int, F char)
    VIEW (%7,%16,%17,%13)[%7,INTEGER],[%16,CHAR],[%17,INTEGER],[%13,CHAR] structure=408,
  %15=From %15:%19 targets: 122=%21 Source: %21 Target=122 Indexes=[(`18)164],
  %16=SqlCopy Name=D %16 Domain 129 From:%15 copy from 142,
  %17=SqlCopy Name=K %17 Domain 182 From:%15 copy from 196,
  %18=SqlCopy Name=U %18 Domain 129 From:%15 copy from 219,
  %19=Domain TABLE (%16,%17,%18) Display=3
    [%16,Domain CHAR],[%17,Domain INTEGER],[%18,Domain CHAR],
  %20=Domain TABLE (%16,%17,%18)[%16,Domain CHAR],[%17,Domain INTEGER],[%18,Domain CHAR],
  %21=TableRowSet %21:%20 From: %15 SRow:(142,196,219) Target:122 Indexes: [(%16)=164],
  %22=RestRowSet %22:%23 targets: 445=%22 SRow:() RemoteCols:(%7,%13)
    RemoteNames:(%7=E,%13=F) UsingTableRowSet %21,
  %23=Domain TABLE (%13|%7,%16,%17) Display=1
    [%13,CHAR],[%7,INTEGER],[%16,Domain CHAR],[%17,Domain INTEGER],
  %24=RestRowSetUsing %24:%23 targets: 445=%24 SRow:(142,196,219) ViewDomain: %23
    Template: %22 UsingTableRowSet:%21 UrlCol:%18,
  %25=Domain TABLE (%13,%7,%16,%17) Display=4
    [%13,CHAR],[%7,INTEGER],[%16,Domain CHAR],[%17,Domain INTEGER],
  %26=Domain TABLE (%7,%13,%16,%17)
    [%7,INTEGER],[%13,CHAR],[%16,Domain CHAR],[%17,Domain INTEGER],
  %27=SqlCopy Name=E %27 Domain 182 From:#33 copy from 484,
  %28=Domain TABLE (#10|%27) Display=1[#10,Domain CHAR],[%27,Domain INTEGER],
  %29=Domain TABLE (%27,%10)[%27,Domain INTEGER],[%10,Domain CHAR],
  %30=TableRowSet %30:%29 key (%27) order (%27) From: #33 SRow:(484,527) Target:477
    Indexes: [(#10)=552,(%27)=508],
  %31=Domain TABLE (%13,%7,%16,%17,%10|%27) Display=5
    [%13,CHAR],[%7,Domain INTEGER],[%16,Domain CHAR],[%17,Domain INTEGER],
    [%10,Domain CHAR],[%27,Domain INTEGER],
  %32=SqlValueExpr Name= %32 BOOLEAN From:#20 Left:%7 Right:%27 %32(%7=%27),
  %33=SqlValueExpr Name= %33 BOOLEAN Left:%7 Right:%27 %33(%7=%27),
  %34=OrderedRowSet %34:%23 key (%7) order (%7) targets: 445=%24 Source: %24,
  %35=SelectStatement %35 Union=#1)}

```

```

http://localhost:8180/DB/DB select F,E,'B' as D,4 as K from t
HTTP POST /DB/DB
select F,E,'B' as D,4 as K from t
Returning ETag: "23,-1,180"
--> 4 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select F,E,'C' as D,1 as K from u
HTTP POST /DC/DC
select F,E,'C' as D,1 as K from u
Returning ETag: "23,-1,159"
--> 3 rows
Response ETag: 23,-1,159
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

```

{(...,
  #8=From #8:%22 matches (%4=8) targets: %21=%21 Source: %21 Target=445,
  #17=Eight,
  #32=SqlValueExpr Name= #32 BOOLEAN Left:%4 Right:#33 #32(%4=#33),
  #33=8,
  %0=RestView Name=WW %0 Domain %11 Definer=-502 Ppos=445 ViewDef Ppos: 445 Result _
    UsingTableRowSet: %18 ViewTable:408,
  %1=Domain ROW (142)[142,Domain CHAR],
  %4=SqlValue Name=E %4 INTEGER From:408,
  %10=SqlValue Name=F %10 CHAR From:408,
  %11=Domain (E int, D char, K int, F char)
    VIEW (%4,%13,%14,%10)[%4,INTEGER],[%13,CHAR],[%14,INTEGER],[%10,CHAR] structure=408,
  %12=From %12:%16 targets: 122=%18 Source: %18 Target=122 Indexes=[(`18)164],
  %13=SqlCopy Name=D %13 Domain 129 From:%12 copy from 142,
  %14=SqlCopy Name=K %14 Domain 182 From:%12 copy from 196,
  %15=SqlCopy Name=U %15 Domain 129 From:%12 copy from 219,
  %16=Domain TABLE (%13,%14,%15) Display=3
    [%13,Domain CHAR],[%14,Domain INTEGER],[%15,Domain CHAR],
  %17=Domain TABLE (%13,%14,%15)[%13,Domain CHAR],[%14,Domain INTEGER],[%15,Domain CHAR],
  %18=TableRowSet %18:%17 From: %12 SRow:(142,196,219) Target:122 Indexes: [(%13)=164],
  %19=RestRowSet %19:%20 matches (%4=8) targets: 445=%19 SRow:() RemoteCols:(%4,%10)
    RemoteNames:(%4=E,%10=F) UsingTableRowSet %18,
  %20=Domain TABLE (%4,%13,%14,%10) Display=4
    [%4,INTEGER],[%13,Domain CHAR],[%14,Domain INTEGER],[%10,CHAR],
  %21=RestRowSetUsing %21:%20 matches (%4=8) targets: 445=%21
    Assigs:(UpdateAssignment Vbl: %10 Val: #17=True) SRow:(142,196,219) ViewDomain: %20
    Template: %19 UsingTableRowSet:%18 UriCol:%15,
  %22=Domain TABLE (%4,%13,%14,%10) Display=4
    [%4,INTEGER],[%13,Domain CHAR],[%14,Domain INTEGER],[%10,CHAR],
  %23=Domain TABLE (%4,%10,%13,%14)
    [%4,INTEGER],[%10,CHAR],[%13,Domain CHAR],[%14,Domain INTEGER],
  %24=UpdateSearch %24 Target: #8)}

```

```

http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where (E=8) and E=8
8
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where (E=8) and E=8
--> 0 rows
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where (E=8) and E=8
8
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where (E=8) and E=8
Returning ETag: "23,159,159"
--> 1 rows
Response ETag: 23,159,159
RoundTrip HEAD http://localhost:8180/DC/DC/u/E=8
HTTP HEAD /DC/DC/u
--> OK
RoundTrip POST http://localhost:8180/DC/DC update u set F='Eight' where (E=8) and E=8
d E=8
HTTP POST /DC/DC
update u set F='Eight' where (E=8) and E=8
--> 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 current\_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 "Orders" (id int primary key,cust int references "Customer", "OrderDate"
date,"Total" numeric(6,2)) entity]
[insert into "Orders" 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",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

```
table "Role$Class"
```

generates fragments that can be pasted into a C# program as follows:

```
using System;
using Pyrrho;
```

```

/// <summary>
/// Class Customer from Database D, Role Sales
/// </summary>
[Table(104,273)]
public class Customer : Versioned
{
    [Key(0)]
    // autoKey enabled
    public Int64? ID;
    [Unique(216, 0)]
    public String NAME;
    public Orders[] orderss => conn.FindWith<Orders>(("CUST",ID));
}

/// <summary>
/// Class Orders from Database D, Role Sales
/// </summary>
[Table(377,613)]
public class Orders : Versioned {
    [Key(0)]
    // autoKey enabled
    public Int64? ID;
    public Int64 CUST;
    [Field(PyrrhoDbType.Date)]
    public Date OrderDate;
    [Field(PyrrhoDbType.Decimal,6,2)]
    public Decimal Total;
    public Customer customer => conn.FindOne<Customer>(CUST);
    public OrderItem[] orderItems => conn.FindWith<OrderItem>(("OID", ID));
}

/// <summary>
/// Class Item from Database D, Role Sales
/// </summary>
[Table(777,923)]
public class Item : Versioned {
    [Key(0)]
    // autoKey enabled
    public Int64? ID;
    public String NAME;
    [Field(PyrrhoDbType.Decimal,6,2)]
    public Decimal PRICE;
    public OrderItem[] orderItems => conn.FindWith<OrderItem>(("ITEM", ID));
}

/// <summary>
/// Class OrderItem from Database D, Role Sales
/// </summary>
[Table(1123,1349)]
public class OrderItem : Versioned {
    [Key(1)]
    // autoKey enabled
    public Int64? IT;
    [Key(0)]
    public Int64 OID;
    public Int64 ITEM;
    public Int64 QTY;
    public Orders orders => conn.FindOne<Orders>(OID);
    public Item item => conn.FindOne<Item>(ITEM);
}

```

The ENTITY metadata flag ensures that all of these classes are Versioned. Note the extra “navigation properties” defined by the integrity constraints. Identifiers and class/table names are case sensitive (this is C#), and the navigation property names are auto-constructed (so that the property orderss has an awkward spelling). FindWith<> takes a list of (key,value) pairs.

The autokey feature of Pyrrho will supply a suitable integer key value for a new entity, but this requires a (long) cast if the key value is used later.

These classes have all public fields. This is so that conn.Put(a) can update the Versioned.version property of a (and maybe other fields if there are triggers). The Versioned library does not require the use of Pyrrho’s list and tree classes: we use only [] in this demo. Attributes such as [Table(104,104)] provide

internal data for the server. For the 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.

If the above classes are at the start of file Program.cs, let the rest be a demo program:

```
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 void Main()
        {
            var conn = new PyrrhoConnect("Files=D;Role=Sales");
            conn.Open();
            try
            {
                // Get a list of all orders showing the customer name
                var aa = conn.FindAll<Orders>();
                foreach (var a in aa)
                    Console.WriteLine(a.ID + " : " + a.customer.NAME);
                if (aa.Length == 0)
                {
                    Console.WriteLine("The Customer 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 Orders() { CUST = (long)m.ID };
                conn.Post(o);
                // we need double quotes around NAME again as it is a reserved word in SQL
                var p = conn.FindWith<Item>(("NAME", "Pins"))[0];
                var i1 = new OrderItem() { OID=(long)o.ID, ITEM=(long)p.ID, QTY = 2 };
                conn.Post(i1);
                // the single quotes are optional if no confusion!
                var b = conn.FindWith<Item>(("NAME", "Bag"))[0];
                var i2 = new OrderItem() { OID=(long)o.ID, ITEM = (long)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 += 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]);
                // try to delete the Rug item (fails)
                var r = conn.FindWith<Item>(("NAME", "Rug"))[0];
                r.Delete();
            } catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            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()

June 2022

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.



## 7. Permissions and the Security Model

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

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

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

### 7.1 Roles

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

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

The system database is set up during Pyrrho initialisation and is called Database.\_system .It contains all the domains, system tables and their columns. All databases inherit these objects on creation. But any domain reference committed to a database file must have a uid matching a defining position in the transaction log, and so a suitable local version of the domain is created for that database to use. The Database maintains an index (with key Domain) to keep track of its local domains.

When a database connects it gains a connection string that must specify the current user and may also specify a role. If the user is known and can only access one role, this role is automatically selected. If the current user is unknown, an ad-hoc user is created. Such an ad-hoc user will only be added to the database if auditing requires it<sup>44</sup> (and in that case becomes a known user, initially with no privileges).

#### 7.1.1 The schema role for a database

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

The schema role maintains a list of the known role and user names.

#### 7.1.2 The guest role (public)

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

Objects granted to PUBLIC are added to the namespace of all roles.

#### 7.1.3 Other roles

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

---

<sup>44</sup> It is unusual for an auditable object to be accessible to the public. But why not?

New database objects are added to the definer's role only. They are added to other roles on GRANT.

## 7.2 Effective permissions

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

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

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

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

## 7.3 Implementation of the Security model

The system database has a Role (with uid -502), and this uid is used for the schema role in any database. The system database User has uid -501. An empty database inherits these, and so does every database in the Database.databases list.

The other predefined role is "guest" and has a uid of -55. This role has access to all public data, that is, it contains all grants to PUBLIC (a dummy user with uid -1).

As is almost implied by the table in section 7.2,

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

### 7.3.1 The Privilege enumeration

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

Flag	Meaning	Flag	Meaning
0x1	Select	0x400	Grant Option for Select
0x2	Insert	0x800	Grant Option for Insert
0x4	Delete	0x1000	Grant Option for Delete
0x8	Update	0x2000	Grant Option for Update
0x10	References	0x4000	Grant Option for References
0x20	Execute	0x8000	Grant Option for Execute
0x40	Owner	0x10000	Grant Option for Owner
0x80	Role	0x20000	Admin Option for Role
0x100	Usage	0x40000	Grant Option for Usage
0x200	Handler	0x80000	Grant Option for Handler

### 7.3.2 Checking permissions

The main methods and properties for this are as follows:

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

### 7.3.3 Grant and Revoke

The main methods and properties for this are as follows:

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

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

REVOKE does not change the list of users or roles. There is nothing to stop a user name being re-used (since there is nothing to stop rights being restored to an existing user or role). It is obviously business process question whether users' real names are used or not (or whether a proposed new user is checked for a match against an existing one).

### 7.3.4 Permissions on newly created objects

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

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

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

### 7.3.5 Dropping objects

The main methods and properties for this are as follows:

- When a database object is dropped it must be removed from any Roles that have privileges on it. This is done by Role.RemoveObject.

- Similarly when a grantee is dropped there is a `DBObject.RemoveGrantee` method to remove all references to it in grantees lists help by database objects.
- All late-parsed data in the schema (view definitions, procedure bodies, triggers, default values etc. are parsed in a `DropTransaction`: objects holding references to the dropped object either prevent the drop occurring (`RESTRICT`), or are dropped in a `CASCADE`, depending on the action specified.

### 7.3.6 Implementation details

During `Load()` initially `_Role` -502

Set by `PTransaction` for its physicals to `ptrole`

then back to -502

In `Transact()` for TCP session (the connection string always specifies a user):

1. initially role is guest. -502 is always the schema role.
2. if the user is unknown in the database
  - a. if the schema role has users, make an uncommitted user id for them with no privileges
  - b. (schema role has no users)
    - i. if the user's name matches the server account, use the schema role
    - ii. otherwise deny access
3. if the connection string specifies a role (add it to the new Tx).
  - a. If the role is not known, report no such role.
  - b. if the role can be used by the public, we are done
  - c. otherwise:
    - i. if the user is known
      1. if the user can use the role, we are done.
      2. if the user is the database owner and the role is schema role, we are done (owner can always manage security model if nothing else)
      3. otherwise deny access
    - ii. if the user is unknown report access denied (even if it is the server a/c)
4. if the connection string does not specify a role
  - a. if the database has roles
    - i. if the user owns the database, use the schema role
    - ii. if the user is known,
      1. if there is just one role the user can use, we are done.
      2. otherwise use guest role (user may be intending to select one)
    - iii. use the guest role: we are done
  - b. (the database has no roles). role is guest

`Protocol.Authority` changes the connection string and `db.mem[_Role]` and updates the session role.

In `Transaction.Commit()` we commit nothing if there are no Physicals, or just an ad-hoc `PUser`. At end of `Transaction.Commit()` remember to add the Connection back in to the new database object `PRole`.

In `Transaction.Audit()` we commit the user if ad-hoc and not defined by a concurrent transaction.

`Install()` `CREATE ROLE` gets a new uid unless it's the first one (-502 is overloaded in that case)

`DROP ROLE` removes the role's name from the list (but does not change the list of users)

`PUser Install()` `GRANT` role doesn't care and might grant -502 (`GRANT TO PUBLIC` is to uid -55

but the first user grantee for the schema role becomes the database owner

During query processing and execution, we do not use `cx.db.role` but `cx.role`. This is initialised to `cx.db.role` but is changed to the definer's role during code execution (including constraint evaluation). `cx.user` always directly addresses `cx.db.conn.user`.

## 7.4 Mandatory Access Control

See section 3.4.2 of the Pyrrho Manual for an introduction to this section.

As usual, implementation of rules throws up unexpected complications. Tables have classifications as do the records they contain, and the interplay between them and user clearances is far from simple. The main purpose of the classification information for a table is to specify the set of groups and references that will apply to records classified above D. It can also specify a minimum clearance level for access to the table. The SA can completely specify or modify the classification of any record in the table (but for best results should use subsets of the groups and references that they have specified for the table).

I have the following for users other than the SA in my first implementation. (As usual in Pyrrho, any exception will roll back the transaction.)

### **Read**

1. If the user does not have select privilege on any of the columns selected or select \* has been specified and the user does not have select privilege for any columns, throw an informative exception (such as “User cannot select column x”, or “user cannot access any columns”).
2. If Select is enforced and the user’s clearance level does not exceed the table’s classification level, report that the table does not exist.

Even if the table contains rows to which the user’s clearance would give them access.

3. If Select is enforced by the table and the user’s clearance does not allow access to a given record, skip the record.
4. If Select is enforced and any records with classification above D are accessed, an audit record is added to the database immediately, whether or not the user’s transaction commits.

This cannot be handled within the ReadConstraint mechanism since ReadConstraints only apply in explicit transaction. The context should record what audit records have already been written to avoid repetition within the same context.

### **Insert**

Apart from actions by the SA:

1. If Insert is enforced by the table and the user does not have insert privilege or the user’s clearance does not exceed the table’s classification, throw an Access Denied exception.
2. If Insert is enforced by the table and the user has insert privilege, construct a record whose classification is equal to the user’s clearance, and insert it.

The new record’s classification label will have the user’s minimum clearance level: if this is above D, the groups will be the subset of the user’s groups that are in the table classification, and the references will be the same as the table (a subset of the user’s references).

3. If Insert is not enforced and the user has insert privilege, the record inserted will have level D classification.

### **Update**

Apart from actions by the SA:

1. If the user does not have update privilege for the table, throw an Access Denied exception.
2. If Update is not enforced the record’s classification will be unaffected (presumably it will be level D).
3. If Update is enforced by the table and the user’s clearance does not allow access to the table, throw an Access Denied exception.

Even if the update would access records that would match the user’s clearance.

4. If Update is enforced by the table, and a record selected for update is not one to which the user has clearance or does not match the user’s clearance level, throw an Access Denied exception.

Even if the user has a higher clearance than the record’s classification.

5. The updated record must have the same classification as the old.

### **Delete**

1. If the user does not have delete permission for the table, throw an Access Denied exception.

Even if the user has a high security clearance.

2. If Delete is enforced by the table for the table or the user’s clearance does not exceed the table’s classification, throw an Access Denied exception.

Even if the delete would actually only remove records that match the user’s clearance.

3. If Delete is enforced by the table and the user has delete privilege for the table, but the record to be deleted has a classification level different from the user or the clearance does not allow access to the record, throw an Access Denied exception.

Even if the delete is attempting to remove an unclassified record.

### 7.4.1 An example

In this example, the server is running on MALCOLM2, and the client accounts are all on the MALCOLM1 machine. Apart from Malcolm himself, there are accounts Fred and Student. We start without the database mac: on creation the server automatically adds a role mac and grants it to Malcolm, who becomes the database owner (and therefore the security administrator).

Note that backslash must not be doubled inside double-quoted strings.

#### A. Logged in with MALCOLM1\Malcolm (not the server account)

1. Starting with empty database mac

```
SQL> create table A(B int,C char)
```

```
SQL> create table D(E char primary key) security level D groups Army Navy references Defence scope read
```

```
SQL> create table F(G char primary key,H char security level C)
```

```
SQL> grant "mac" to "MALCOLM1\Student"
```

```
SQL> grant "mac" to "MALCOLM1\Fred"
```

```
SQL> grant security level B groups Army references Defence Cyber to "MALCOLM1\Student"
```

```
SQL> table "Sys$User"
```

Pos	Name	SetPassword	InitialRole	Clearance
26	MALCOLM1\Malcolm		mac	
366	MALCOLM1\Student		mac	B{ARMY}[CYBER,DEFENCE]
416	MALCOLM1\Fred		mac	

3. Add some rows with and without classification

```
SQL> insert into A values(2,'Two')
```

1 records affected in mac

```
SQL> insert into A values(3,'Three') security level C
```

1 records affected in mac

```
SQL> insert into D values('Test')
```

1 records affected in mac

```
SQL> insert into F values('MI6','sis.gov.uk')
```

1 records affected in mac

```
SQL> table "Sys$Classification"
```

Pos	Type	Classification	LastTransaction
553	Record	C	537
154	Table	D{ARMY,NAVY}[DEFENCE]	138
313	TableColumn	C	248

4. Check we can see two rows in A, one row in D and two columns in F

```
SQL> table A
```

Pos	Name
2	Two
3	Three

```
SQL> table D
```

Pos	Name
1	Test

```
SQL> table F
```

Pos	Name
6	MI6
7	sis.gov.uk

```

Command Prompt - pyrhocmd -h:192.168.1.197 mac
Microsoft Windows [Version 10.0.19042.541]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Malcolm>
E:\>cd pyrhocmd\pyrhocmd -h:192.168.1.197 mac
SQL> create table A(B int,C char)
SQL> create table D(E char primary key) security level D groups Army Navy references Defence scope read
SQL> create table F(G char primary key,H char security level C)
SQL> grant "mac" to "MALCOLM1\Student"
SQL> grant "mac" to "MALCOLM1\Fred"
SQL> grant security level B groups Army references Defence Cyber to "MALCOLM1\Student"
SQL> table "Sys$User"
+-----+-----+-----+-----+
|Pos|Name|SetPassword|InitialRole|Clearance|
+-----+-----+-----+-----+
|26|MALCOLM1\Malcolm|mac|mac| |
|366|MALCOLM1\Student|mac|mac|B{ARMY}[CYBER,DEFENCE]|
|416|MALCOLM1\Fred|mac|mac|
+-----+-----+-----+-----+
SQL> insert into A values(2,'Two')
1 records affected in mac
SQL> insert into A values(3,'Three') security level C
1 records affected in mac
SQL> insert into D values('Test')
1 records affected in mac
SQL> insert into F values('MI6','sis.gov.uk')
1 records affected in mac
SQL> table "Sys$Classification"
+-----+-----+-----+-----+
|Pos|Type|Classification|LastTransaction|
+-----+-----+-----+-----+
|553|Record|C|537|
|154|Table|D{ARMY,NAVY}[DEFENCE]|138|
|313|TableColumn|C|248|
+-----+-----+-----+-----+
SQL> table A
+-----+
|B|C|
+-----+
|2|Two|
|3|Three|
+-----+
SQL> table D
+-----+
|E|
+-----+
|Test|
+-----+
SQL> table F
+-----+
|G|H|
+-----+
|MI6|sis.gov.uk|
+-----+
SQL>

```

June 2022

|---|-----|

### B. Logged in as Fred

5. Check we can only see one row in A, one column in F, and nothing in D

```
SQL> table A
```

```
|---|
|B|C|
|---|
|2|Two|
|---|
```

```
SQL> table D
```

Access denied

```
SQL> table F
```

```
|---|
|G|
|---|
|MI6|
|---|
```

6. Check we can add a row in A, D and F

```
SQL> insert into A values(4, 'Four')
```

1 records affected in mac

```
SQL> insert into D values('Fred wrote this')
```

1 records affected in mac

```
SQL> insert into F values('UWS')
```

1 records affected in mac

```
SQL> table a
```

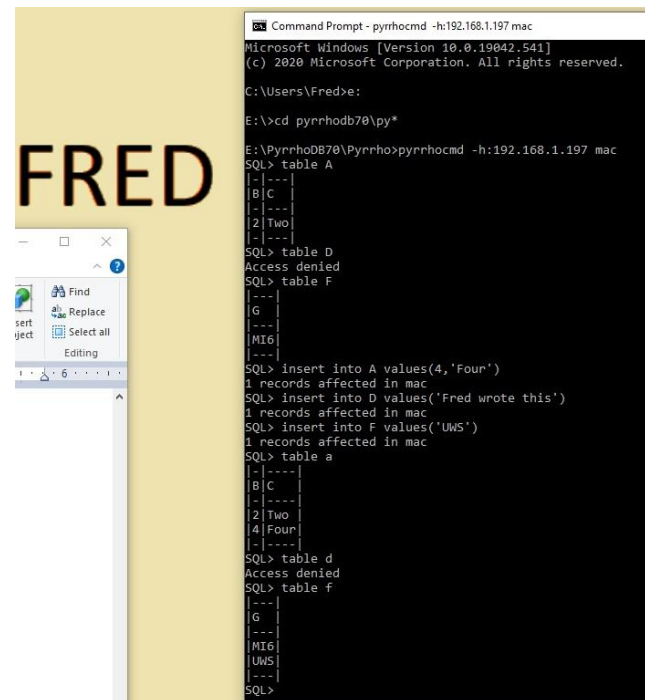
```
|---|
|B|C|
|---|
|2|Two|
|4|Four|
|---|
```

```
SQL> table d
```

Access denied

```
SQL> table f
```

```
|---|
|G|
|---|
|MI6|
|UWS|
|---|
```



### C. Logged in as Student

7. Check we can see three rows in A, two rows in D and two columns in F

```
SQL> table A
```

```
|---|
|B|C|
|---|
|2|Two|
|3|Three|
|4|Four|
|---|
```

```
SQL> table D
```

```
|-----|
|E|
|-----|
|Fred wrote this|
|Test|
|-----|
```

```
SQL> table F
```

```
|---|
|G|H|
|---|
|MI6|sis.gov.uk|
|UWS|
|---|
```

8. Check we can only make changes in table D (enforcement in D is only on read)

```
SQL> update A set c = 'No' where b=2
```

Access denied

```
SQL> update A set c = 'No' where b=3
```

Access denied

June 2022

```
SQL> update A set c = 'No' where b=4
Access denied
SQL> update D set E='Fred?' where
E<>'Test'
1 records affected in mac
SQL> update F set H='www.sis.gov.uk' where
G='MI6'
Access denied
SQL> update F set H='www.uws.ac.uk' where
G='UWS'
Access denied
9. Check we can add and update our rows in all three
tables
SQL> insert into A values(5,'Fiv')
1 records affected in mac
SQL> update A set c='Five' where b=5
1 records affected in mac
SQL> insert into D values('Another')
1 records affected in mac
SQL> insert into F
values('BBC','bbc.co.uk')
1 records affected in mac
SQL> update F set H='www.bbc.co.uk' where
G='BBC'
1 records affected in mac
```

10. Check we can see our rows and changes

SQL> table A

B	C
2	Two
3	Three
4	Four
5	Five

SQL> table D

E
Another
Fred?
Test

SQL> table F

G	H
BBC	www.bbc.co.uk
MI6	sis.gov.uk
UWS	

SQL>

#### D. Logged in as Fred

11. Check Fred can't see the new rows

SQL> table a

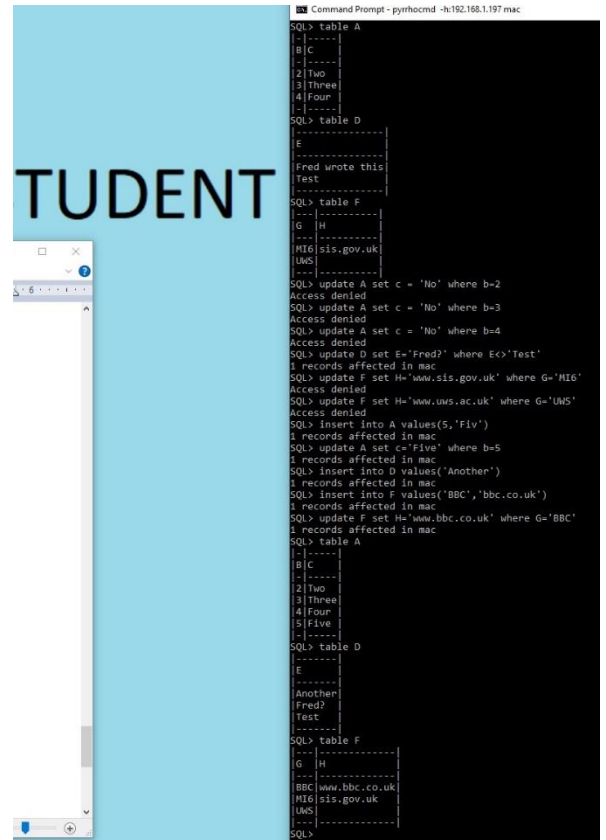
B	C
2	Two
4	Four

SQL> table d

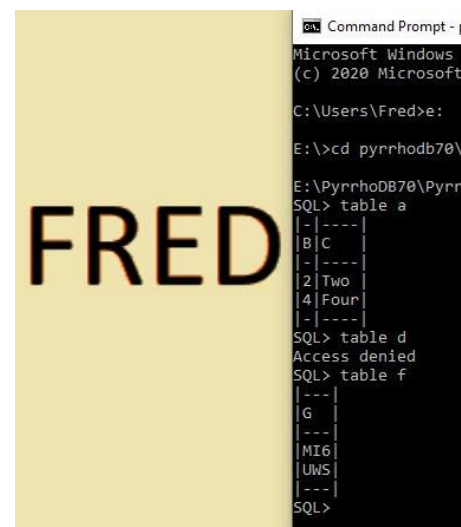
Access denied

SQL> table f

G	H
MI6	
UWS	



```
Command Prompt - pyrrhodb70
SQL> table A
+-----+
| B | C |
+-----+
| 2 | Two |
| 3 | Three |
| 4 | Four |
+-----+
SQL> table D
+-----+
| E |
+-----+
| Fred wrote this |
| Test |
+-----+
SQL> table F
+-----+
| G | H |
+-----+
| MI6 | sis.gov.uk |
| UWS |
+-----+
SQL> update A set c = 'No' where b=2
Access denied
SQL> update A set c = 'No' where b=3
Access denied
SQL> update A set c = 'No' where b=4
Access denied
SQL> update D set E='Fred?' where E<>'Test'
1 records affected in mac
SQL> update F set H='www.sis.gov.uk' where G='MI6'
Access denied
SQL> update F set H='www.uws.ac.uk' where G='UWS'
Access denied
SQL> insert into A values(5,'Fiv')
1 records affected in mac
SQL> update A set c='Five' where b=5
1 records affected in mac
SQL> insert into D values('Another')
1 records affected in mac
SQL> insert into F values('BBC','bbc.co.uk')
1 records affected in mac
SQL> update F set H='www.bbc.co.uk' where G='BBC'
1 records affected in mac
SQL> table A
+-----+
| B | C |
+-----+
| 2 | Two |
| 3 | Three |
| 4 | Four |
| 5 | Five |
+-----+
SQL> table D
+-----+
| E |
+-----+
| Another |
| Fred? |
| Test |
+-----+
SQL> table F
+-----+
| G | H |
+-----+
| BBC | www.bbc.co.uk |
| MI6 | sis.gov.uk |
| UWS |
+-----+
SQL>
```



```
Command Prompt - pyrrhodb70
Microsoft Windows
(c) 2020 Microsoft
C:\Users\Fred>e:
E:\>cd pyrrhodb70\
E:\Pyrrhodb70\Pyrrhodb70>
SQL> table a
+-----+
| B | C |
+-----+
| 2 | Two |
| 4 | Four |
+-----+
SQL> table d
Access denied
SQL> table f
+-----+
| G |
+-----+
| MI6 |
| UWS |
+-----+
SQL>
```



SQL&gt;

**E. Logged in as database owner**

12. Check all tables including the security information

SQL&gt; select B,C,security from A

B	C	SECURITY
2	Two	
3	Three	
4	Four	C
5	Five	

SQL&gt; select E,security from D

E	SECURITY
Another	
Fred?	
Test	

SQL&gt; select G,H,security from F

G	H	SECURITY
BBC	www.bbc.co.uk	B
MI6	sis.gov.uk	B
UWS		

SQL&gt; select \* from A where security=level c

B	C
3	Three

SQL&gt; update A set security=level B where security=level C

1 records affected in mac

SQL&gt; update F set security=level D where G='BBC'

1 records affected in mac

SQL&gt; table "Sys\$Classification"

Pos	Type	Classification	LastTransaction
553	Record	B	537
1022	Record	B	1005
154	Table	D{ARMY,NAVY}[DEFENCE]	138
313	TableColumn	C	248

```
SQL> select B,C,security from A
B C SECURITY
2 Two
3 Three
4 Four C
5 Five

SQL> select E,security from D
E SECURITY
Another
Fred?
Test

SQL> select G,H,security from F
G H SECURITY
BBC www.bbc.co.uk B
MI6 sis.gov.uk B
UWS

SQL> select * from A where security=level c
B C
3 Three

SQL> update A set security=level B where security=level C
1 records affected in mac
SQL> update F set security=level D where G='BBC'
1 records affected in mac
SQL> table "Sys$Classification"
Pos Type Classification LastTransaction
553 Record B 537
1022 Record B 1005
154 Table D{ARMY,NAVY}[DEFENCE] 138
313 TableColumn C 248
```

**F. Logged in as Student**

13. Check we can still see our row in A

SQL&gt; select \* from a where b=5

B	C
5	Five

14. Check we can no longer update our rows in A or F

SQL&gt; delete from A where b=5

Access denied

SQL&gt; update F set H='bbc.com' where G='BBC'

Access denied

DE

```
SQL> select * from a where b=5
B C
5 Five

SQL> delete from A where b=5
Access denied
SQL> update F set H='bbc.com' where G='BBC'
Access denied
SQL>
```

**G. Logged in as Fred**

15. Check we can see the row about the BBC

SQL&gt; table F

G
BBC
MI6
UWS

FRED

```
SQL> table F
G
BBC
MI6
UWS
```

**H. Logged in as database owner**

16. Check that auditing has been happening

SQL&gt; table "Sys\$Audit"

Pos	User	Table	Timestamp
665	MALCOLM1\Fred	62	03/10/2020 10:58:52
684	MALCOLM1\Fred	62	03/10/2020 10:59:08
824	MALCOLM1\Fred	62	03/10/2020 10:59:21
849	MALCOLM1\Student	62	03/10/2020 11:00:28
868	MALCOLM1\Student	62	03/10/2020 11:00:40
893	MALCOLM1\Student	62	03/10/2020 11:00:40
918	MALCOLM1\Student	62	03/10/2020 11:00:40
986	MALCOLM1\Student	62	03/10/2020 11:00:52
1050	MALCOLM1\Student	62	03/10/2020 11:00:52
1273	MALCOLM1\Student	62	03/10/2020 11:01:02
1292	MALCOLM1\Fred	62	03/10/2020 11:01:32
1424	MALCOLM1\Student	62	03/10/2020 11:02:42
1449	MALCOLM1\Student	62	03/10/2020 11:02:52

SQL&gt; table "Sys\$AuditKey"

Pos	Seq	Col	Key
824	0	82	4
868	0	82	2
893	0	82	3
918	0	82	4
1050	0	82	5
1424	0	82	5
1449	0	82	5

17. Finally, here is the complete database log:

SQL&gt; table "Log\$"

Pos	Desc	Type	Affects
26	PUser MALCOLM1\Malcolm	PUser	26
46	PTransaction for 3 Role=5 User=26 Time=10/03/2020 10:56:59	PTransaction	46
62	PTable A	PTable	62
68	Domain INTEGER	PDomain	68
82	PColumn3 B for 62(0)[68]	PColumn3	82
103	Domain CHAR	PDomain	103
116	PColumn3 C for 62(1)[103]	PColumn3	116
138	PTransaction for 5 Role=5 User=26 Time=10/03/2020 10:56:59	PTransaction	138
154	PTable D	PTable	154
161	PColumn3 E for 154(0)[103]	PColumn3	161
184	PIndex D on 154(161) PrimaryKey	PIndex	184
203	Classify 154 D{ARMY,NAVY}{DEFENCE}	Classify	203
239	Enforcement [154] SCOPE read	Enforcement	239
248	PTransaction for 5 Role=5 User=26 Time=10/03/2020 10:57:00	PTransaction	248
264	PTable F	PTable	264
271	PColumn3 G for 264(0)[103]	PColumn3	271
294	PIndex F on 264(271) PrimaryKey	PIndex	294
313	PColumn3 H for 264(1)[103]	PColumn3	313
337	Classify 313 C	Classify	337
350	PTransaction for 2 Role=5 User=26 Time=10/03/2020 10:57:08	PTransaction	350
366	PUser MALCOLM1\Student	PUser	366
388	Grant UseRole on 5 to 366	Grant	388
400	PTransaction for 2 Role=5 User=26 Time=10/03/2020 10:57:08	PTransaction	400
416	PUser MALCOLM1\Fred	PUser	416
435	Grant UseRole on 5 to 416	Grant	435
447	PTransaction for 1 Role=5 User=26 Time=10/03/2020 10:57:08	PTransaction	447
463	Clearance 366 B{ARMY}{CYBER,DEFENCE}	Clearance	463
500	PTransaction for 1 Role=5 User=26 Time=10/03/2020 10:57:18	PTransaction	500
516	Record 516[62]: 82=2,116=Two	Record	516
537	PTransaction for 1 Role=5 User=26 Time=10/03/2020 10:57:18	PTransaction	537
553	Record3 553[62]: 82=3,116=Three Classification: C	Record3	553
580	PTransaction for 1 Role=5 User=26 Time=10/03/2020 10:57:18	PTransaction	580
596	Record 596[154]: 161=Test	Record	596
615	PTransaction for 1 Role=5 User=26 Time=10/03/2020 10:57:18	PTransaction	615
631	Record 631[264]: 271=MI6,313=sis.gov.uk	Record	631
665	Audit: MALCOLM1\Fred [62] 10/03/2020 10:58:52	Audit	665
684	Audit: MALCOLM1\Fred [62] 10/03/2020 10:59:08	Audit	684
703	PTransaction for 1 Role=5 User=416 Time=10/03/2020 10:59:08	PTransaction	703
720	Record 720[62]: 82=4,116=Four	Record	720
742	PTransaction for 1 Role=5 User=416 Time=10/03/2020 10:59:08	PTransaction	742
759	Record 759[154]: 161=Fred wrote this	Record	759
789	PTransaction for 1 Role=5 User=416 Time=10/03/2020 10:59:10	PTransaction	789
806	Record 806[264]: 271=UWS	Record	806
824	Audit: MALCOLM1\Fred [62] 10/03/2020 10:59:21 {82='4'}	Audit	824
849	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:28	Audit	849
868	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:40 {82='2'}	Audit	868
893	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:40 {82='3'}	Audit	893
918	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:40 {82='4'}	Audit	918
943	PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:40	PTransaction	943
960	Update 759[154]: 161=Fred? Prev:759	Update	759
986	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:52	Audit	986
1005	PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:52	PTransaction	1005
1022	Record3 1022[62]: 82=5,116=Fiv Classification: B	Record3	1022
1050	Audit: MALCOLM1\Student [62] 10/03/2020 11:00:52 {82='5'}	Audit	1050
1075	PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:52	PTransaction	1075
1092	Update 1022[62]: 82=5,116=Five Prev:1022	Update	1022

1120 PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:52	PTransaction 1120	
1137 Record 1137[154]: 161=Another	Record	1137
1159 PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:52	PTransaction 1159	
1176 Record3 1176[264]: 271=BBC,313=bbc.co.uk Classification: B	Record3	1176
1213 PTransaction for 1 Role=5 User=366 Time=10/03/2020 11:00:54	PTransaction 1213	
1230 Update 1176[264]: 271=BBC,313=www.bbc.co.uk Prev:1176	Update	1176
1273 Audit: MALCOLM1\Student [62] 10/03/2020 11:01:02	Audit	1273
1292 Audit: MALCOLM1\Fred [62] 10/03/2020 11:01:32	Audit	1292
1311 PTransaction for 1 Role=5 User=26 Time=10/03/2020 11:02:10	PTransaction 1311	
1327 Update1 553[62]: 82=3,116=Three Classification: B Prev:553	Update1	553
1359 PTransaction for 1 Role=5 User=26 Time=10/03/2020 11:02:10	PTransaction 1359	
1375 Update1 1176[264]: 271=BBC,313=www.bbc.co.uk Classification: Prev:1176	Update1	1176
1424 Audit: MALCOLM1\Student [62] 10/03/2020 11:02:42 {82='5'}	Audit	1424
1449 Audit: MALCOLM1\Student [62] 10/03/2020 11:02:52 {82='5'}	Audit	1449
---- -----		

## 7.5 The Type system and OWL support

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

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

In a user-defined type, defpos gives the Type definition, which can be accessed for method definitions etc, while the structType field gives the structure type. From v7 the fields of Types are SqlValues, not TableColumns.

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

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

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

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

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

During query analysis of the select lists, selectors have unique SqlName identifiers, and the identity of the corresponding data and types is propagated up and down the query parse tree, using the SqlValue Setup method to supply constraining type information. Selectors are either explicitly in the column lists in select statements, or implicit as in select \* or aggregation operations.

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

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

## 8. The HTTP service

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

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

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

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

### 8.1 URL format

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

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

Selector matches

```
[$table]Table_id
[$procedure]Procedure_id
[$where]Column_id=string
[$select]Column_id{,Column_id}
[$key]string
```

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

Processing matches:

```
orderasc Column_id{, Column_id}
orderdesc Column_id{, Column_id}
skip Int_string
count Int_string
```

Suffix matches

```
$edmx
$xml
$sql
```

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

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

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

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

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

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

## 8.2 REST implementation

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

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

A single SELECT statement can be sent in a GET request to the Role URL, and the returned data will be the entire RowSet that results (after where-conditions if supplied), together with an ETag as a cache verification token.

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

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

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

## 8.3 RESTViews

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

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

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

The rules are quite complicated:

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

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

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

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

QS (AVG(E+K), F) group by F ->

CS (SUM(E+[K]) as C\_2, F COUNT(E+[K]) as D\_2),

[K] is the current value of K from UF

GF (SUM(C\_2) as C2, F, COUNT(D\_2) as D2)

-> QS(C2/D2 as "AVG(E+K)", F)

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

We will add a set of query-rewriting rules to the database engine aiming to reduce D by recursive analysis of QS and the views and tables it references. As the later sections of this document explain, some of these rules can be very simple, such as filtering by rows or columns of V, while others involve performing some aggregations remotely (extreme cases such as select count(\*) from V needs only one row to be returned). In particular, we will study the interactions between grouped aggregations and joins. The

analysis will in general be recursive, since views may be defined using aggregations and joins of other views and local tables.

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

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

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

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

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

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

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

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

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

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

## 9. References

Crowe, M. K.: The Pyrrho Database Management System, University of the West of Scotland, (2005-18) [www.pyrrhodb.com](http://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)



## APPENDIX

Tutorial demonstrations from [DBKDA 2021 Program \(iaria.org\)](https://iaria.org), updated for this version.

### **Demo 1: Introducing Pyrrho DBMS**

Malcolm Crowe, 14 May 2021 (this document revised 20 May 2022)

[Slide 7 @ 03:47 in the [Tutorial video](#)]



Our first demonstration is about the transaction log. The transaction log defines the contents of the database.

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) 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.

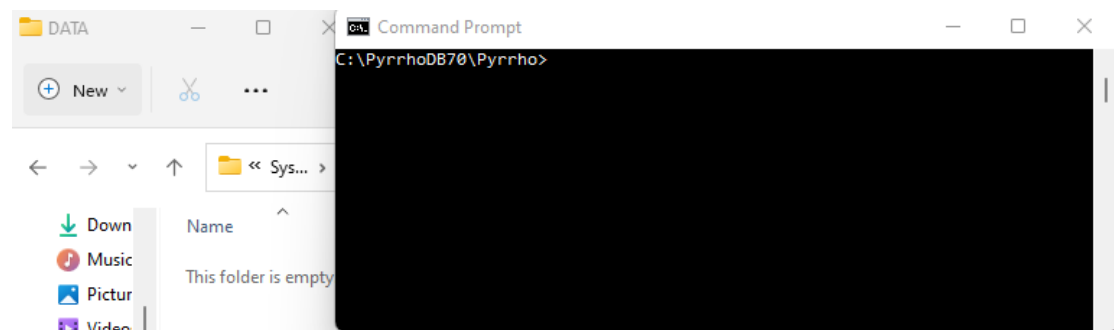
Any state of the database is thus the result of the entire sequence of committed transactions, starting from a known initial database state corresponding to an empty database.

[8 @ 05:06 ]

Let's start with a command window, set to whatever folder the distribution is in. Don't worry if yours is different.

The Pyrrho distribution folder contains the server and command line processor executable files, and the PyrrhoLink dll. (You can copy these files to anywhere convenient.)

This command window will be for the server.



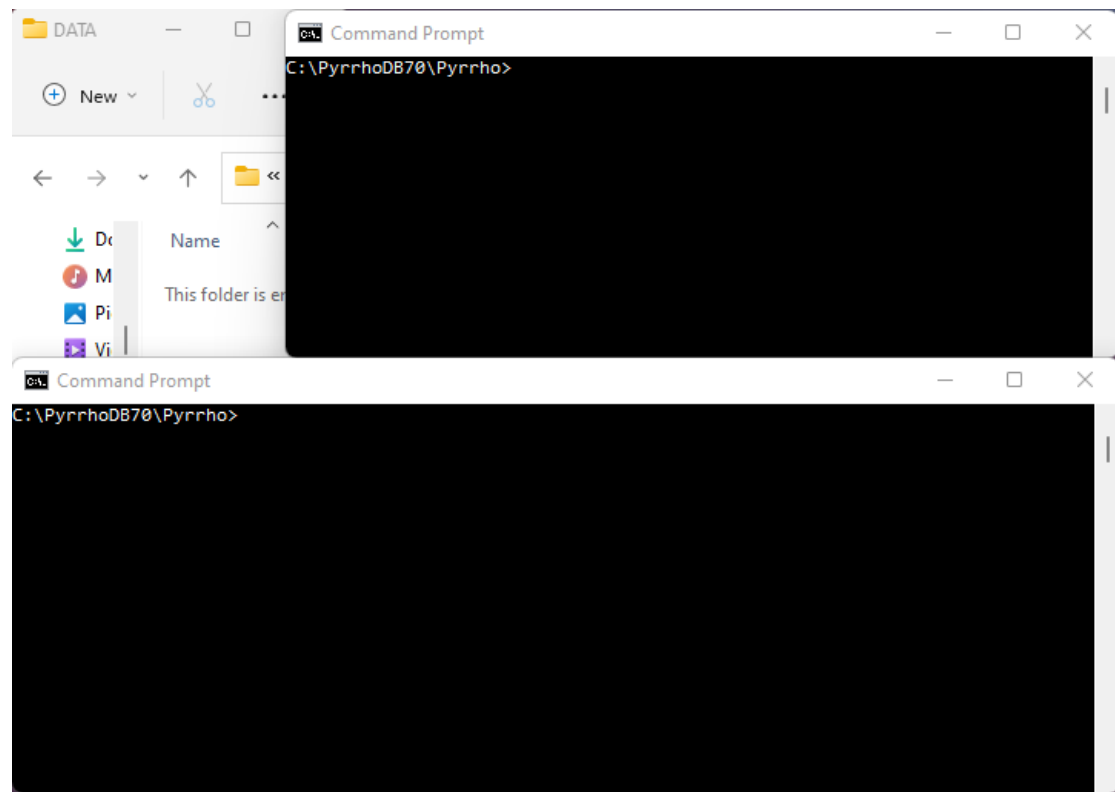
[9 @ 05:29]

Let us agree on a folder for the database files. Create a new folder \DATA. To save space, as here, you can overlap the windows a bit.

[10 @ 05:42]

Before we start the server, let's add a command window for the client, also with the same folder. We will see it is better to make it wider than normal.

June 2022

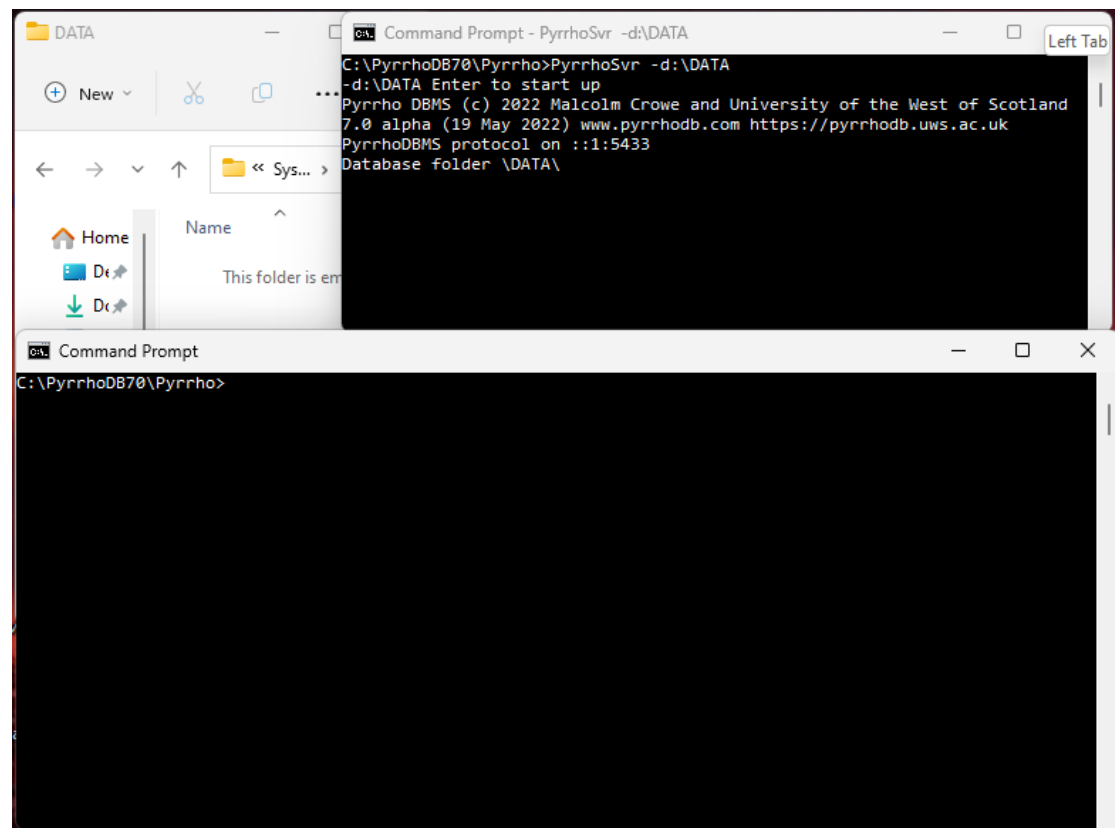


[11 @ 05:54]

The Pyrrho server is called PyrrhoSvr, and it runs in an ordinary account with no special privileges. You can copy it to anywhere convenient.

It is easiest just to use it in a command window. When we specify the server, we can provide a folder for it to store database files in.

**PyrrhoSvr -d:\DATA**



[12 @ 06:16]

When prompted, we verify that the options specified are the ones we want, and click Enter.

[13 @ 06:26]

You then need to leave the command window open (but of course you can minimise it). It can be useful for diagnostic information if something goes wrong.

On startup, Pyrrho checks that it can access the specified folder, but places nothing in it. Databases are created by users, and the transaction log will be stored by the server on disk, usually in this default folder. It also announces its TCP/IP request port.

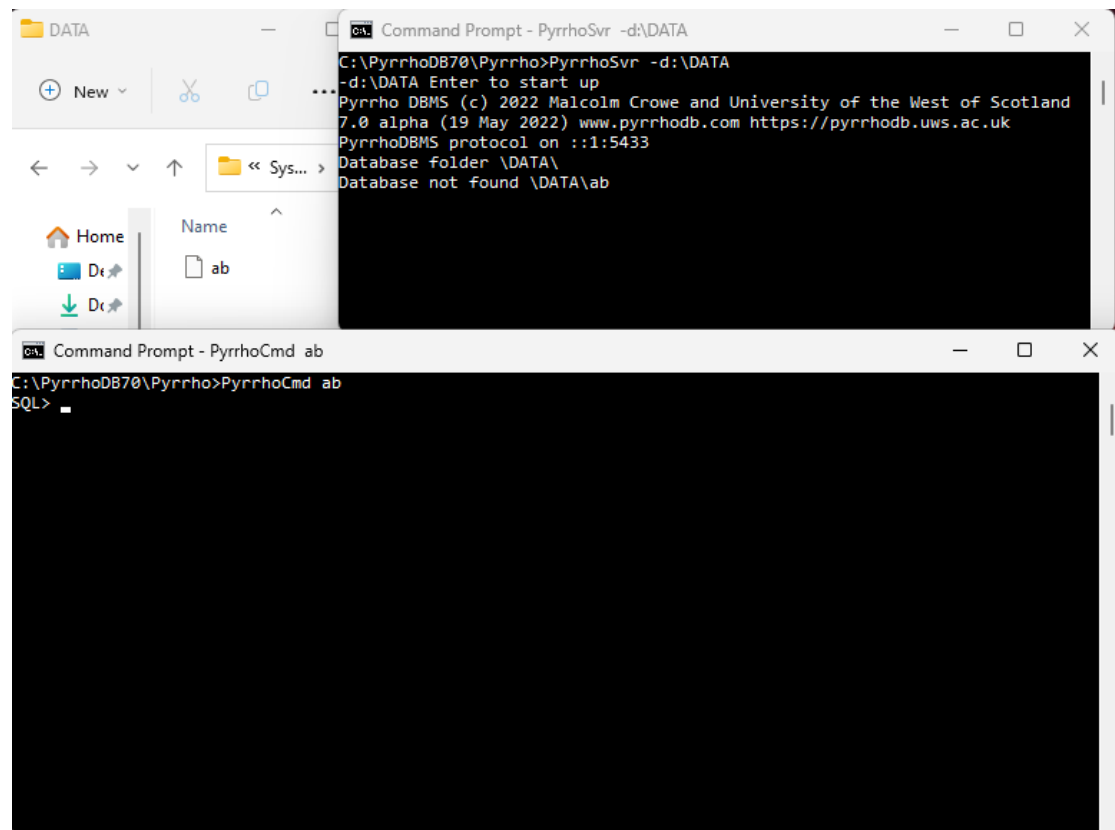
[14 @ 06:58]

The simplest way to get Pyrrho to do work for you is to connect to it using the command line processor PyrrhoCmd, and when you do that you can specify the database you are connecting to. It can be a new database, as here.

**PyrrhoCmd ab**

[15 @ 07:17]

And when you give the command, the server immediately creates an empty database in the database folder.



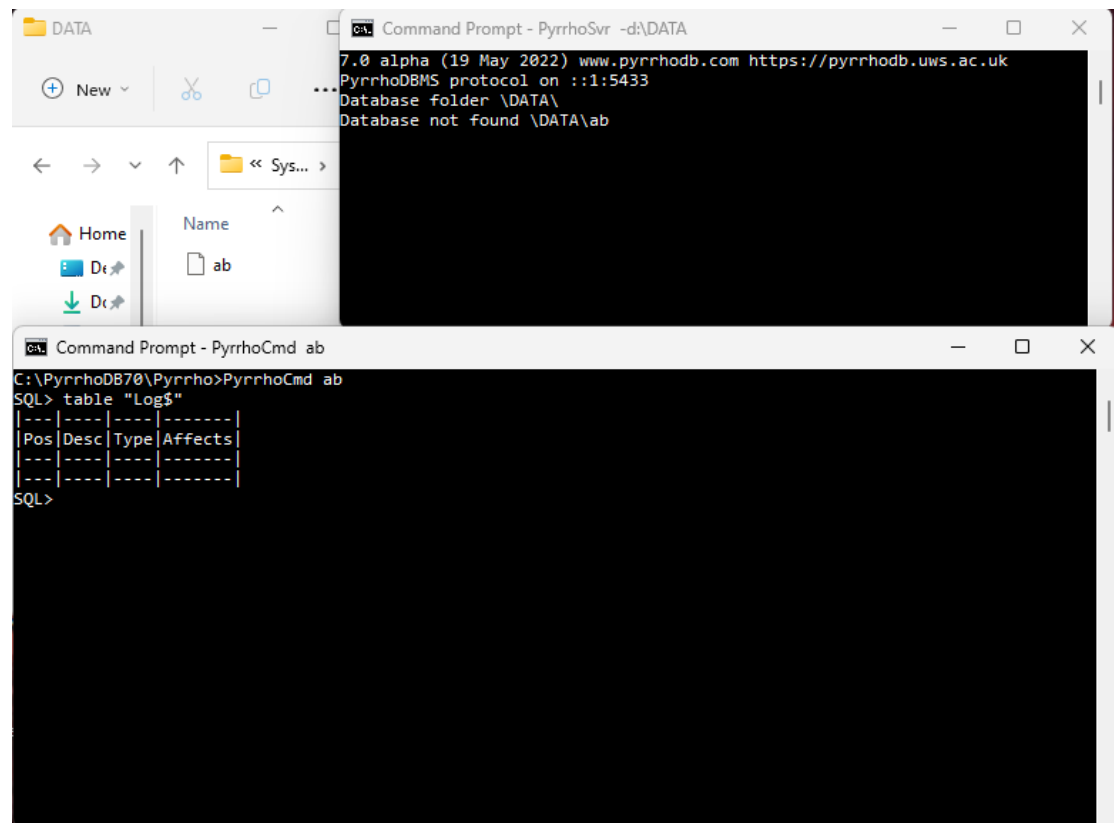
This is a transaction log, and at the moment it contains just 5 bytes, and it is opened exclusively by the server, so that we can't look at it unless the server is stopped.

[16 @ 07:38]

We can examine the contents of the transaction log with the table "Log\$" statement. Log\$ is one of many system tables for inspecting server internals from SQL. It is empty at the moment.

SQL> **table "Log\$"**

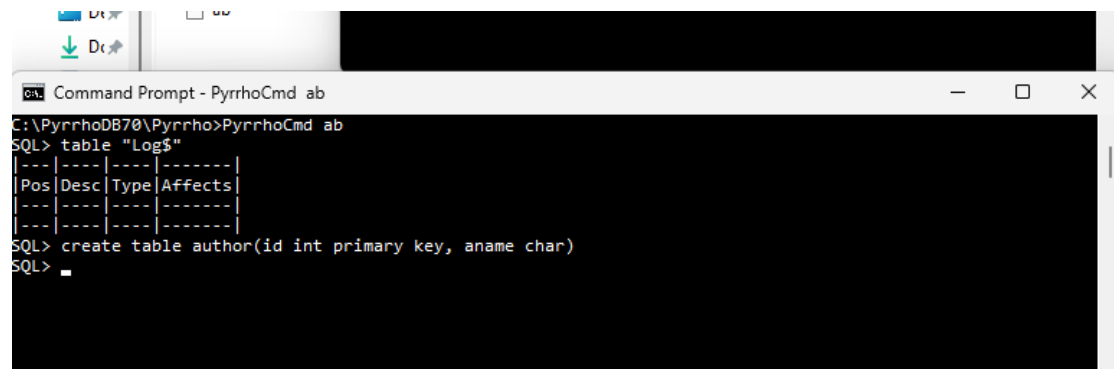
(table "Log\$" is another way of writing `select * from "Log$"`.) Be careful to use a straight double-quote character here.



[17 @ 07:54]

Let's make a base table in the database ab, by giving a CREATE TABLE command, and this is the normal syntax in the SQL standard, but Pyrrho has its own ideas about types. So CHAR, for example, is an unbounded character string, and INT is actually a "bigint" – it's up to more than 2000 bits

**SQL> create table author(id int primary key, aname char)**



[18 @ 08:20]

So, we have created the table, but let's look at it: of course it will be empty.

**SQL> table author**

```

C:\PyrrhoDB70\Pyrrho>PyrrhoCmd ab
SQL> table "Log$"
|---|---|---|---|
|Pos|Desc|Type|Affects|
|---|---|---|---|
|---|---|---|---|
SQL> create table author(id int primary key, aname char)
SQL> table author
|---|---|
|ID|ANAME|
|---|---|
|---|---|
SQL>

```

[19 @ 08:29]

Let's examine how this has been represented in the transaction log.

```

---|---|---|---|
Pos|Desc|Type|Affects|
---|---|---|---|
QL> create table author(id int primary key, aname char)
QL> table author
--|---|
ID|ANAME|
--|---|
--|---|
QL> table "Log$"
---|---|---|---|
Pos|Desc|Type|Affects|
---|---|---|---|
5 |PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:42:25|PTransaction|5|
23 |PTable AUTHOR|PTable|23|
34 |Domain INTEGER|PDomain|34|
48 |PColumn3 ID for 23(0)[34]|PColumn3|48|
70 |PIndex AUTHOR on 23(48) PrimaryKey|PIndex|70|
91 |Domain CHAR|PDomain|91|
104 |PColumn3 ANAME for 23(1)[91]|PColumn3|104|
---|---|---|---|
QL>

```

We can see that there are 7 objects in the transaction log, all wrapped in a single transaction. The AUTHOR table is mentioned, and the domains of the two columns, and the two column names, and the primary key, as defined in the create table request, is defined there along with its key.

You will notice later on that the file positions that are here (actual byte positions in the file) are used as unique identifiers for objects in the database. Pyrrho uses them throughout: names can be changed but the file position of the definition can't. So here the table AUTHOR is referred to as 23, the domain INTEGER is 34, ID is 48, and that becomes the primary key, and so on. The numbers in brackets give the ordering of the columns in the table.

The log shows the identity of the user who made the changes, and the role they were using, but in this database, no users have been defined yet, so these are system defaults.

[20 @ 09:38]

Let us add some rows to the AUTHOR table (be careful to use the straight single-quote character here):

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

June 2022

```
Command Prompt - PyrrhoCmd ab

SQL> create table author(id int primary key, aname char)
SQL> table author
ID|ANAME
SQL> table "Log$"
Pos Desc Type Affects
5 PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:42:25 PTransaction 5
23 PTable AUTHOR PTable 23
34 Domain INTEGER PDomain 34
48 PColumn3 ID for 23(0)[34] PColumn3 48
70 PIndex AUTHOR on 23(48) PrimaryKey PIndex 70
91 Domain CHAR PDomain 91
104 PColumn3 ANAME for 23(1)[91] PColumn3 104
SQL> insert into author values(1,'Dickens'),(2,'Conrad')
2 records affected in ab
SQL>
```

Let is look at our AUTHOR table

SQL> table author

```
Command Prompt - PyrrhoCmd ab

SQL> table "Log$"
Pos Desc Type Affects
5 PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:42:25 PTransaction 5
23 PTable AUTHOR PTable 23
34 Domain INTEGER PDomain 34
48 PColumn3 ID for 23(0)[34] PColumn3 48
70 PIndex AUTHOR on 23(48) PrimaryKey PIndex 70
91 Domain CHAR PDomain 91
104 PColumn3 ANAME for 23(1)[91] PColumn3 104
SQL> insert into author values(1,'Dickens'),(2,'Conrad')
2 records affected in ab
SQL> table author
ID|ANAME
1|Dickens
2|Conrad
SQL>
```

[21 @ 09:53]

In the transaction log we see that the auto-committed transaction has both records.

```
Command Prompt - PyrrhoCmd ab

SQL> table author
ID|ANAME
1|Dickens
2|Conrad
SQL> table "Log$"
Pos Desc Type Affects
5 PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:42:25 PTransaction 5
23 PTable AUTHOR PTable 23
34 Domain INTEGER PDomain 34
48 PColumn3 ID for 23(0)[34] PColumn3 48
70 PIndex AUTHOR on 23(48) PrimaryKey PIndex 70
91 Domain CHAR PDomain 91
104 PColumn3 ANAME for 23(1)[91] PColumn3 104
130 PTransaction for 2 Role=-502 User=-501 Time=20/05/2022 15:45:48 PTransaction 130
148 Record 148[23]: 48=1,104=Dickens Record 148
173 Record 173[23]: 48=2,104=Conrad Record 173
SQL>
```

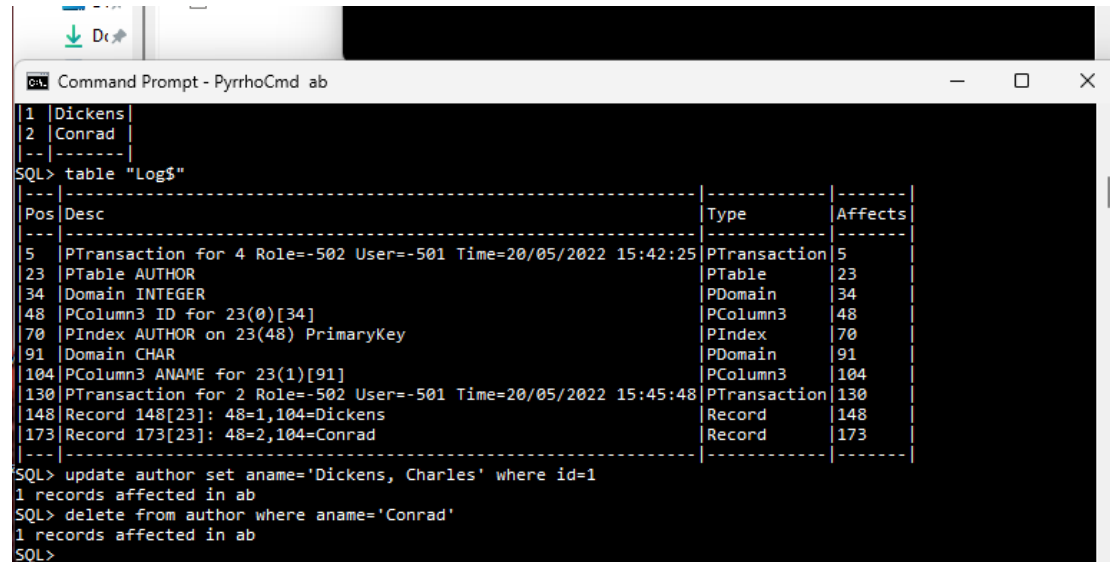
June 2022

[22 @ 10:04]

We can update, and delete.

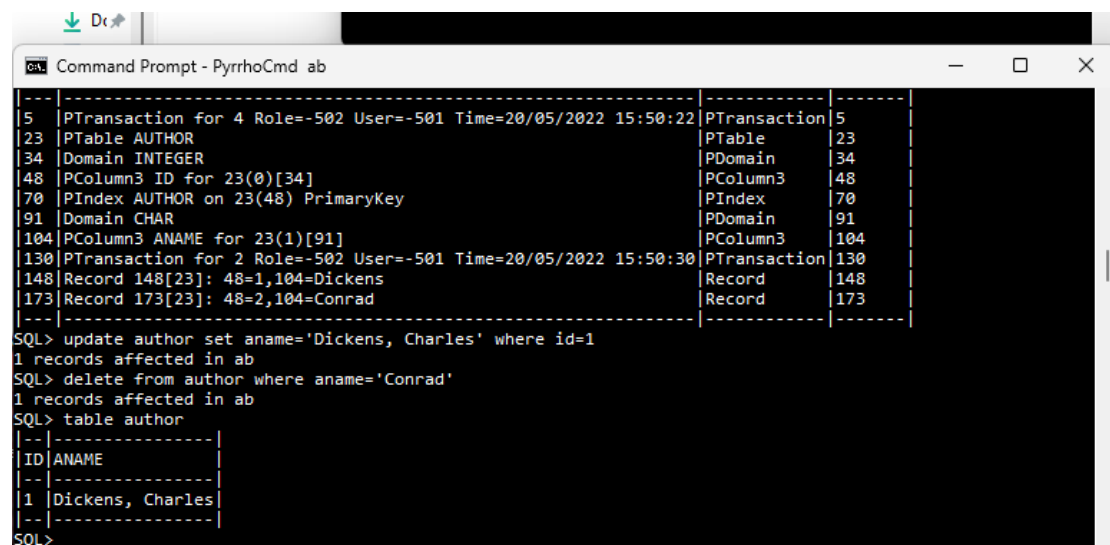
SQL> update author set aname='Dickens, Charles' where id=1

SQL> delete from author where aname='Conrad'



```
1 | Dickens |
2 | Conrad |
--|-----|
SQL> table "Log$"
-----|-----|-----|
Pos|Desc|Type|Affects|
-----|-----|-----|
5 | PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:42:25 | PTransaction | 5 |
23 | PTable AUTHOR | PTable | 23 |
34 | Domain INTEGER | PDomain | 34 |
48 | PColumn3 ID for 23(0)[34] | PColumn3 | 48 |
70 | PIndex AUTHOR on 23(48) PrimaryKey | PIndex | 70 |
91 | Domain CHAR | PDomain | 91 |
104 | PColumn3 ANAME for 23(1)[91] | PColumn3 | 104 |
130 | PTransaction for 2 Role=-502 User=-501 Time=20/05/2022 15:45:48 | PTransaction | 130 |
148 | Record 148[23]: 48=1,104=Dickens | Record | 148 |
173 | Record 173[23]: 48=2,104=Conrad | Record | 173 |
-----|-----|-----|
SQL> update author set aname='Dickens, Charles' where id=1
1 records affected in ab
SQL> delete from author where aname='Conrad'
1 records affected in ab
SQL>
```

SQL> table author



```
5 | PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:50:22 | PTransaction | 5 |
23 | PTable AUTHOR | PTable | 23 |
34 | Domain INTEGER | PDomain | 34 |
48 | PColumn3 ID for 23(0)[34] | PColumn3 | 48 |
70 | PIndex AUTHOR on 23(48) PrimaryKey | PIndex | 70 |
91 | Domain CHAR | PDomain | 91 |
104 | PColumn3 ANAME for 23(1)[91] | PColumn3 | 104 |
130 | PTransaction for 2 Role=-502 User=-501 Time=20/05/2022 15:50:30 | PTransaction | 130 |
148 | Record 148[23]: 48=1,104=Dickens | Record | 148 |
173 | Record 173[23]: 48=2,104=Conrad | Record | 173 |
-----|-----|-----|
SQL> update author set aname='Dickens, Charles' where id=1
1 records affected in ab
SQL> delete from author where aname='Conrad'
1 records affected in ab
SQL> table author
-----|-----|
ID|ANAME|
-----|-----|
1 | Dickens, Charles |
-----|-----|
SQL>
```

[23 @ 10:18]

Finally let's see the update and delete in the Log.



```

C:\PyrrhoDB70\Pyrrho>PyrrhoSvr -d:\DATA
-d:\DATA Enter to start up
Pyrrho DBMS (c) 2022 Malcolm Crowe and University of the West of Scotland
7.0 alpha (19 May 2022) www.pyrrhodb.com https://pyrrhodb.uws.ac.uk
PyrrhoDBMS protocol on ::1:5433
Database folder \DATA\
Database not found \DATA\ab

SQL> table "Log$"

```

Pos	Desc	Type	Affects
5	PTransaction for 4 Role=-502 User=-501 Time=20/05/2022 15:50:22	PTransaction	5
23	PTable AUTHOR	PTable	23
34	Domain INTEGER	PDomain	34
48	PColumn3 ID for 23(0)[34]	PColumn3	48
70	PIndex AUTHOR on 23(48) PrimaryKey	PIndex	70
91	Domain CHAR	PDomain	91
104	PColumn3 ANAME for 23(1)[91]	PColumn3	104
130	PTransaction for 2 Role=-502 User=-501 Time=20/05/2022 15:50:30	PTransaction	130
148	Record 148[23]: 48=1,104=Dickens	Record	148
173	Record 173[23]: 48=2,104=Conrad	Record	173
197	PTransaction for 1 Role=-502 User=-501 Time=20/05/2022 15:50:39	PTransaction	197
215	Update 148[23]: 104=Dickens, Charles Prev:148	Update	148
250	PTransaction for 1 Role=-502 User=-501 Time=20/05/2022 15:50:40	PTransaction	250
268	Delete Record 173[23]	Delete1	173

```

SQL>

```

This completes the demonstration showing the use of the transaction log and the transaction markers.

## Demo 2: Pyrrho DBMS and concurrency

Malcolm Crowe, 2 June 2021 (revised 20 May 2022)

This demo is updated from a [tutorial](#) at DBKDA 2021.

We look in detail at Pyrrho's 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.

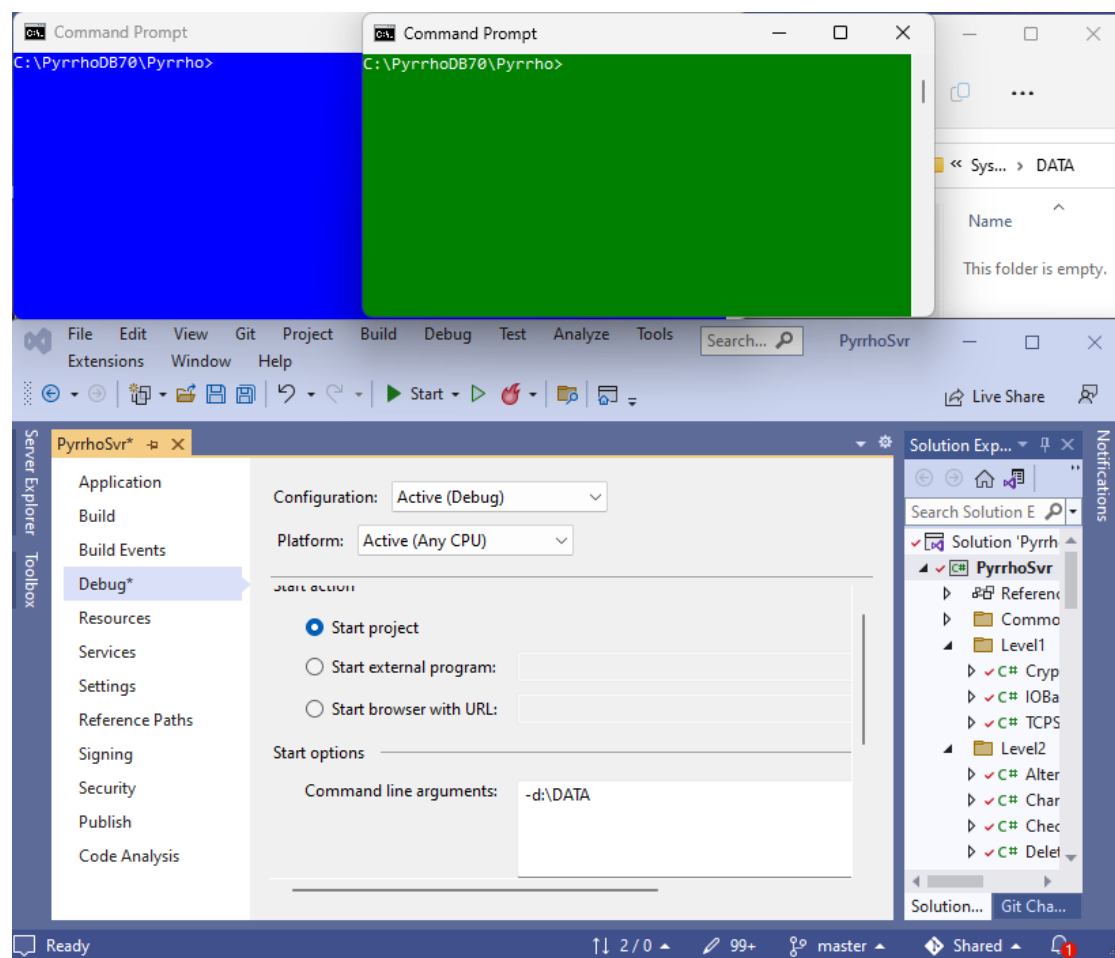
If that is the case, we can relocate all our proposed changes to come after the committed transactions.

The tests for write-write conflicts involve comparing our list of physicals with those of the other transactions.

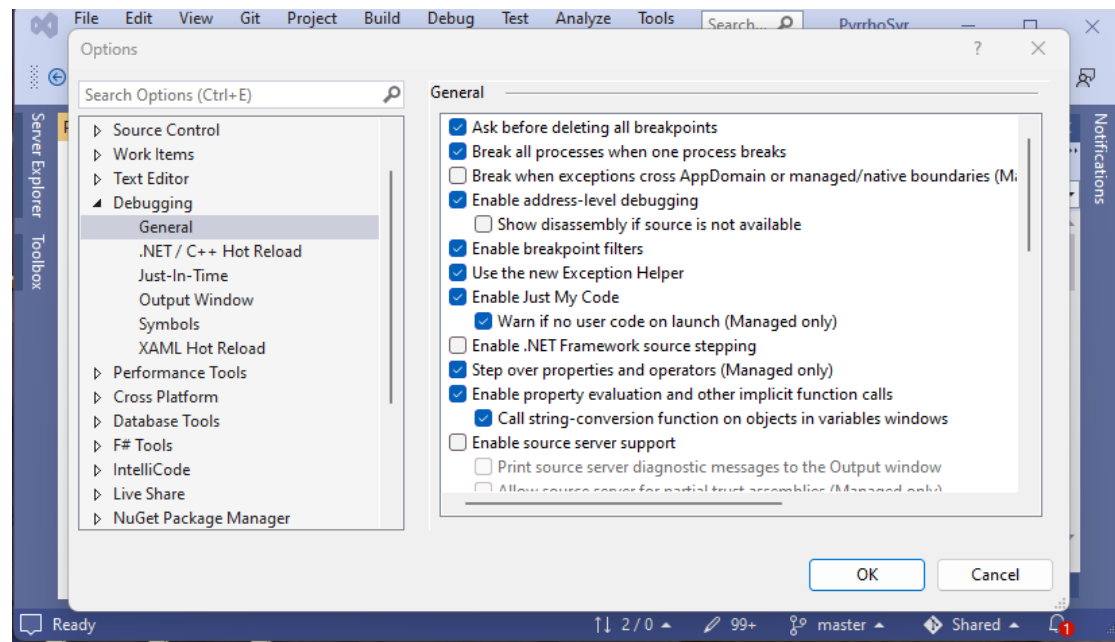
For checking read-write conflicts, we collect “read constraints” when we are making Cursors.

### Part 1: Setting up the demo

Use Visual Studio to open the solution in Pyrrho's src\Shared folder. In the Solution properties, under Debug>Command Line Arguments, enter -d: followed by a suitable database folder such as \DATA. The folder should not contain a file t10.

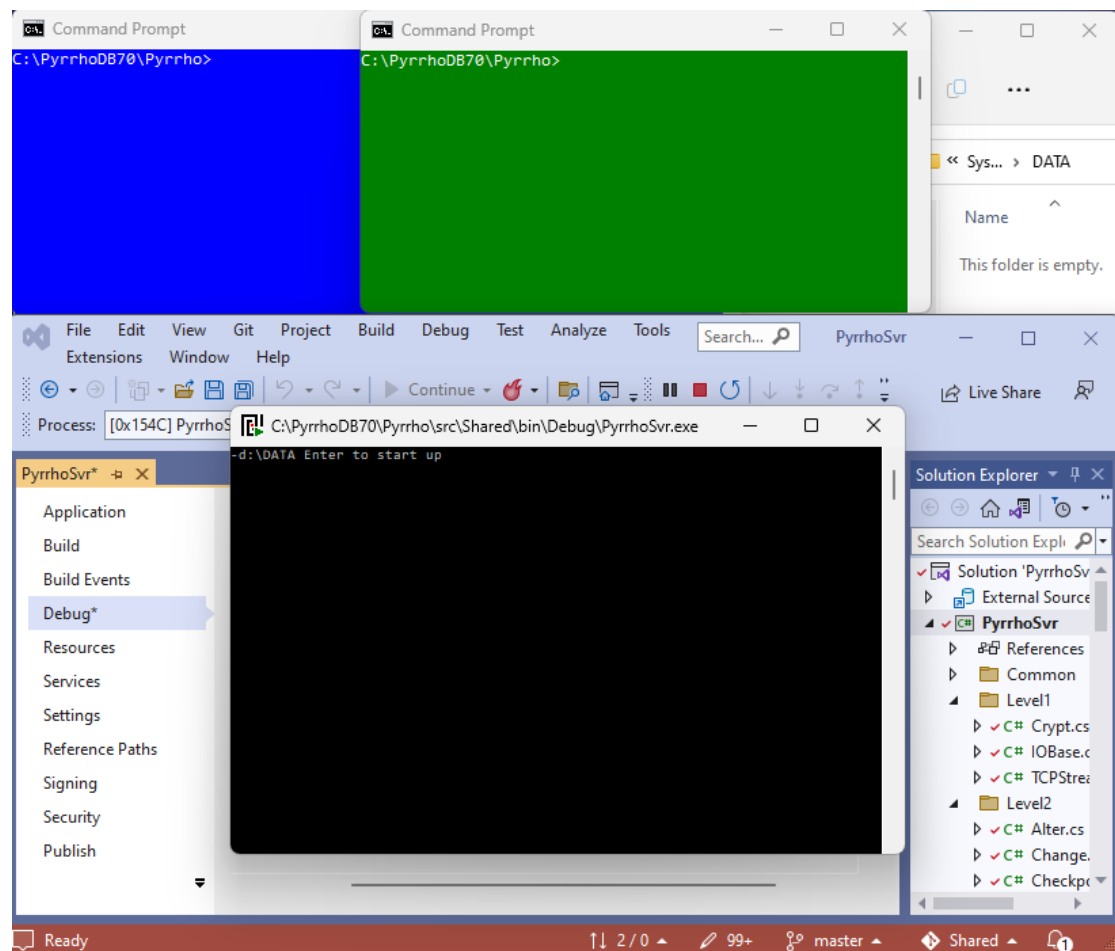


Also ensure that in the Debug>Options.. window,

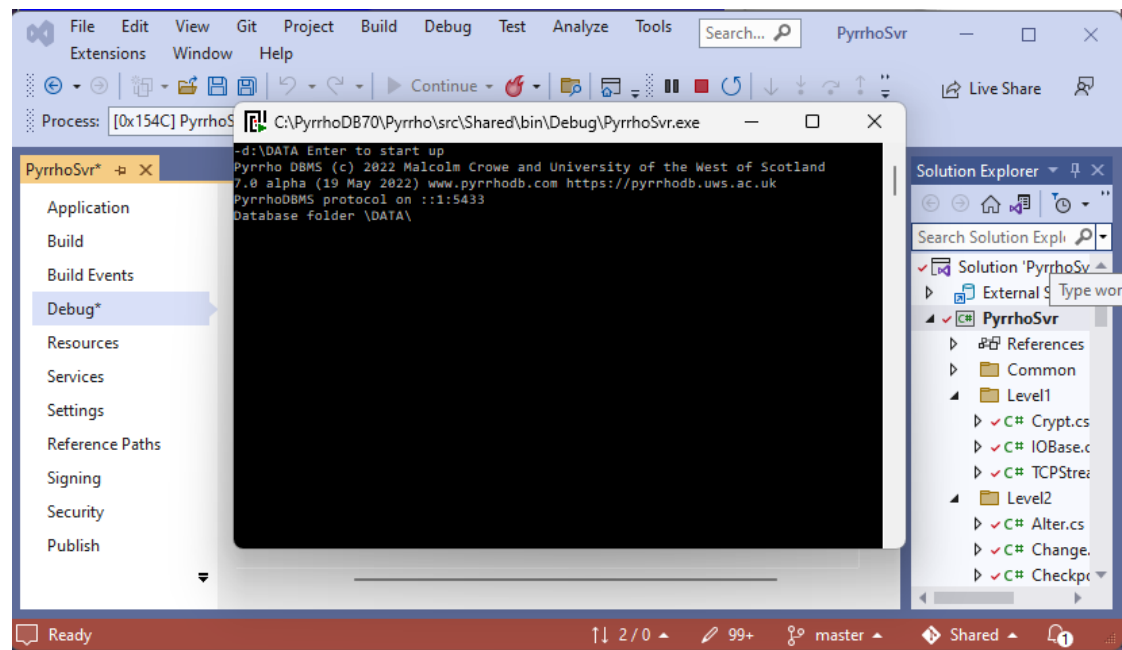


the checkbox Step over properties and operators (managed only) is checked. Click OK.

Click Start.



In the pop-up command window, hit the enter key to start up the server:

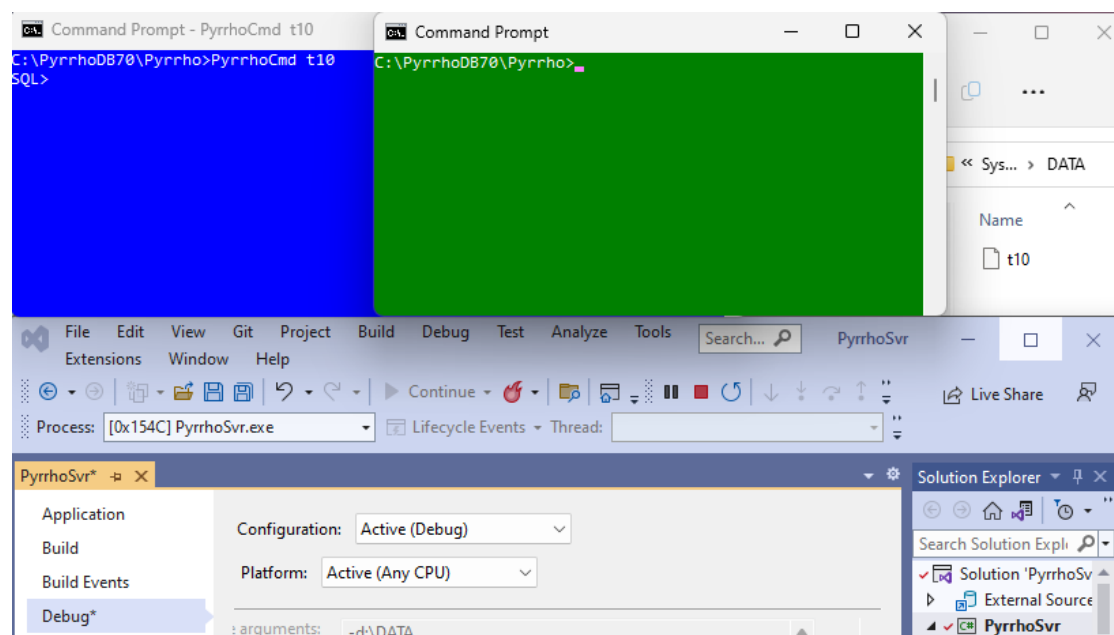


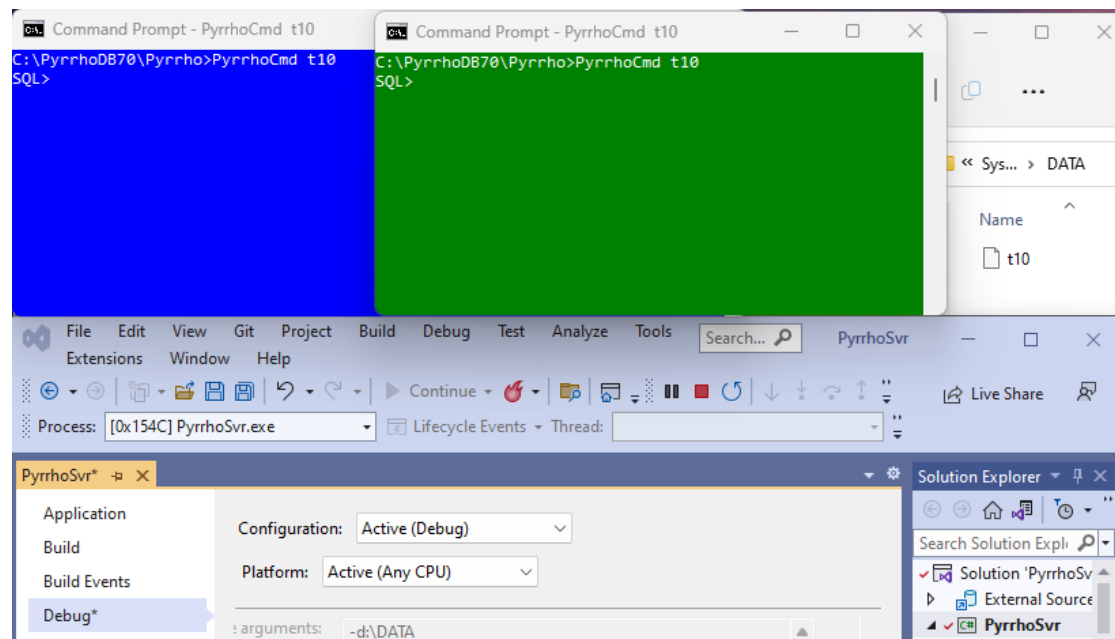
Minimise the pop-up command window.

At this point ensure there is no database t10 in the nominated folder.

The command we want to run in both windows is the same: `PyrrhoCmd t10`. Once we do one of them, the DBMS immediately creates the database as we have seen in demo 1.

### PyrrhoCmd t10

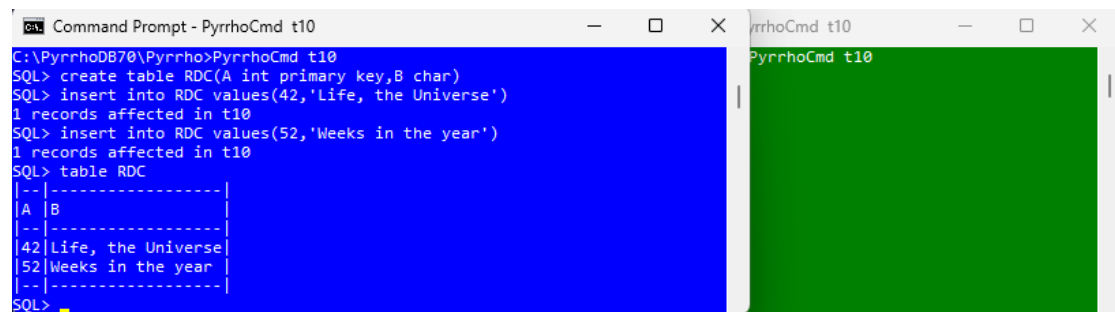




From now on, the folder window can be hidden (it does not change).

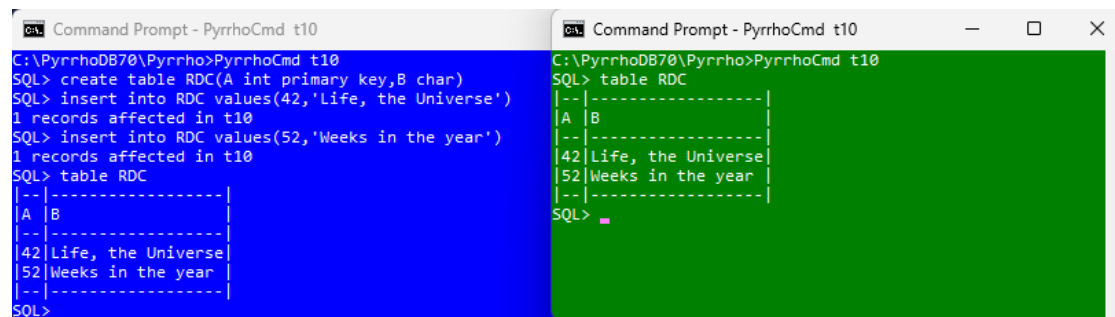
In the blue window, let us create a table (it is called RDC because we will demonstrate read-write conflicts), by giving the following commands:

```
create table RDC(A int primary key,B char)
insert into RDC values(42,'Life, the Universe')
insert into RDC values(52,'Weeks in the year')
table RDC
```



That has all been committed, because we are running in auto-commit mode, so the green window will pick it up. It is also running in auto-commit mode, so a new command starts a new transaction, and it will check the current state of the database.

```
table RDC
```



This completes the set up for this demo. We will repeat this part of the demo later.

## Part 2: A Write-write conflict

Now we start an explicit transaction in the blue window.

### begin transaction

```

C:\> Command Prompt - PyrrhoCmd t10
SQL> create table RDC(A int primary key,B char)
SQL> insert into RDC values(42,'Life, the Universe')
1 records affected in t10
SQL> insert into RDC values(52,'Weeks in the year')
1 records affected in t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|
SQL> begin transaction
SQL-T>

C:\PyrrhoDB70\Pyrrho>PyrrhoCmd t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|

```

If we just relied on auto-commit mode it is very difficult to synchronise an overlap of transactions. For a demo, it works very well to use explicit transactions.

Notice that in an explicit transaction the prompt changes to SQL-T>. As long as the transaction is running, we will get these prompts. When the transaction is over, either because of commit, or rollback, or because we have done something wrong, it will go back to SQL>.

We will request conflicting changes to table RDC in these two clients. In the blue window, let's "delete from RDC where A=42", to delete the first row.

### delete from RDC where A=42

```

C:\> Command Prompt - PyrrhoCmd t10
1 records affected in t10
SQL> insert into RDC values(52,'Weeks in the year')
1 records affected in t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|
SQL> begin transaction
SQL-T>delete from RDC where A=42
1 records in transaction t10
SQL-T>

C:\PyrrhoDB70\Pyrrho>PyrrhoCmd t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|

```

That's in a transaction, so nothing has been written to disk yet.

In the green window, we are still in auto-commit mode and we are going to make an update to the same row, which will be committed straightaway to disk.

### update RDC set B='The answer to the ultimate question' where A=42

This should make the blue window unable to commit its transaction because of the conflict.

```

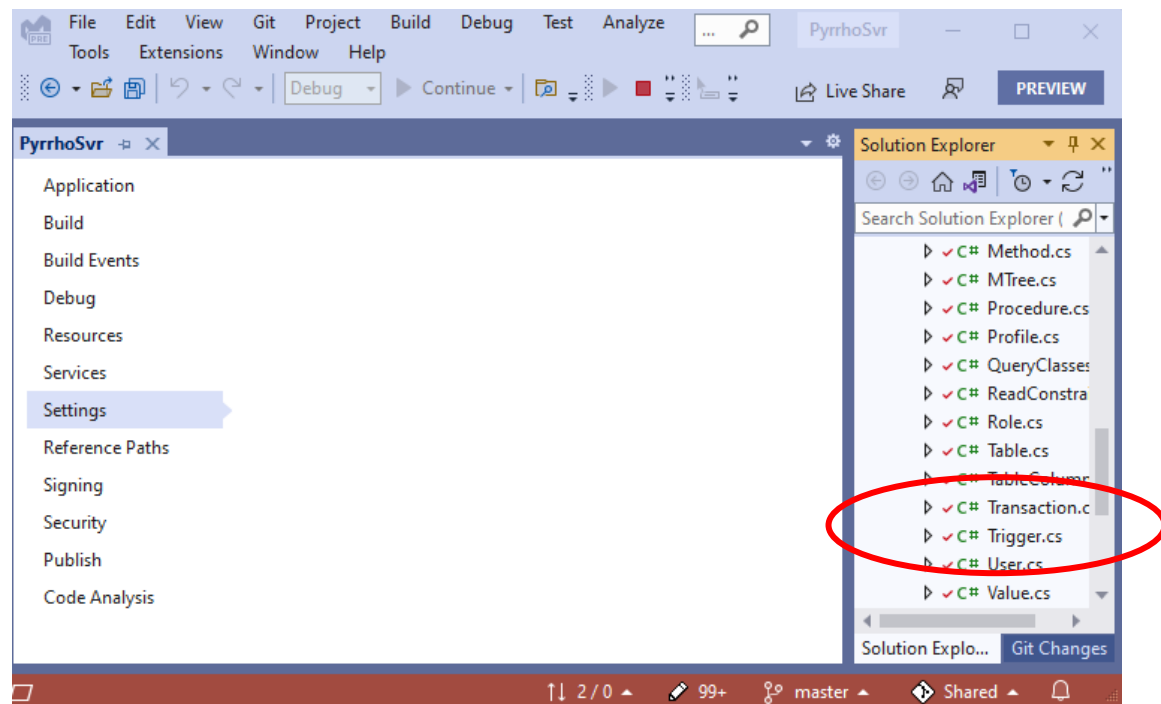
C:\> Command Prompt - PyrrhoCmd t10
1 records affected in t10
SQL> insert into RDC values(52,'Weeks in the year')
1 records affected in t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|
SQL> begin transaction
SQL-T>delete from RDC where A=42
1 records in transaction t10
SQL-T>

C:\PyrrhoDB70\Pyrrho>PyrrhoCmd t10
SQL> table RDC
--|-----|
|A|B|
--|-----|
|42|Life, the Universe|
|52|Weeks in the year|
--|-----|
SQL> update RDC set B='The answer to the ultimate question' where A=42
1 records affected in t10
SQL>

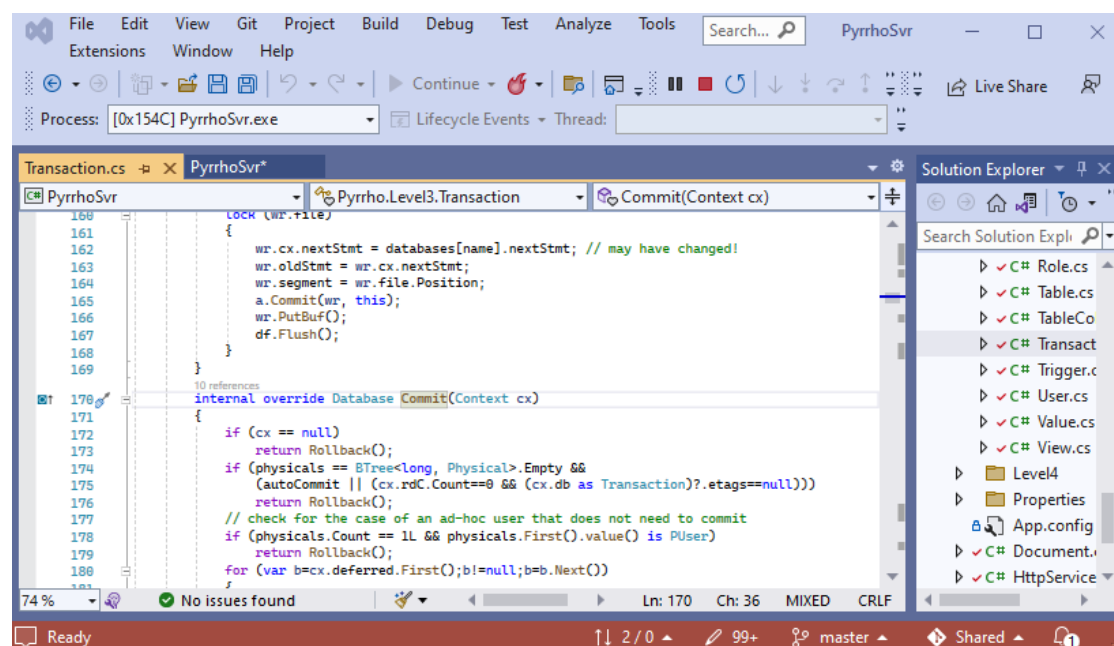
```

In order to see what happens, let us set a breakpoint in the debugger. In Solution Explorer, in the Level3 folder, locate file Transaction.cs and double-click.

June 2022

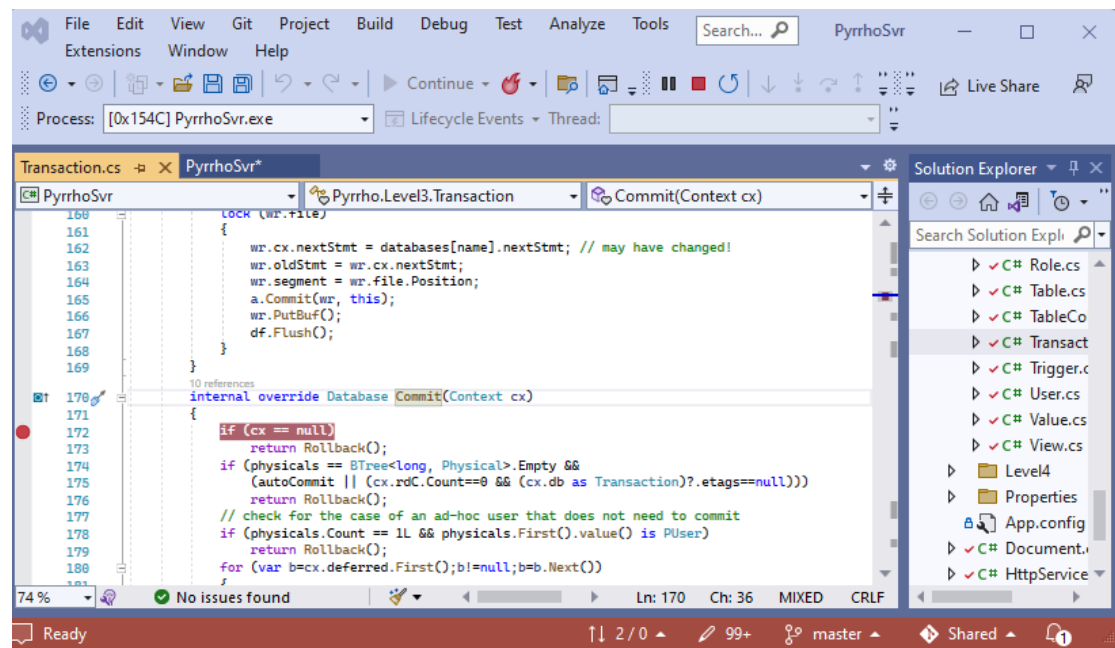


Scroll down to line 170 at the start of the Transaction.Commit() method.



Click the mouse in the left margin to set a break point at line 172 of the Transaction.cs file.

June 2022

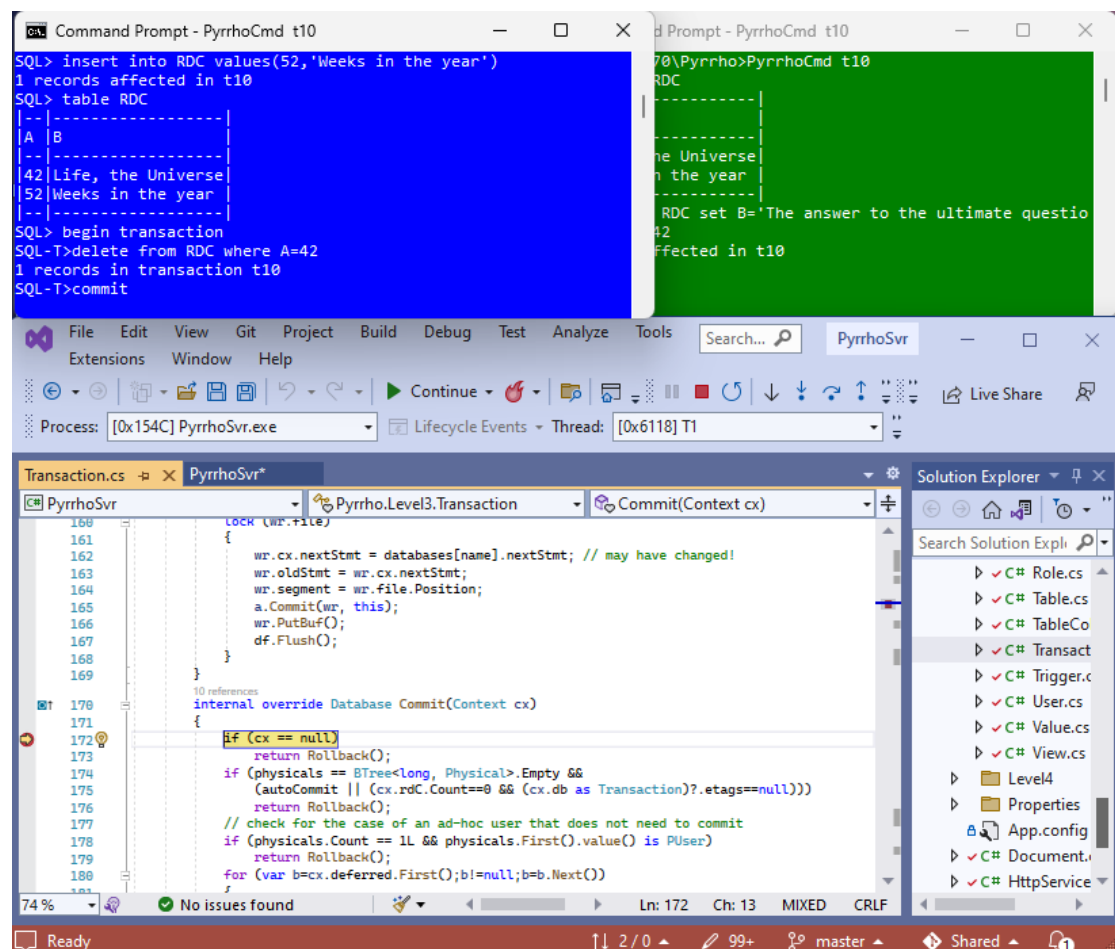


[39 @ 19:17]

Then when we issue the commit command in the blue window,

**commit**

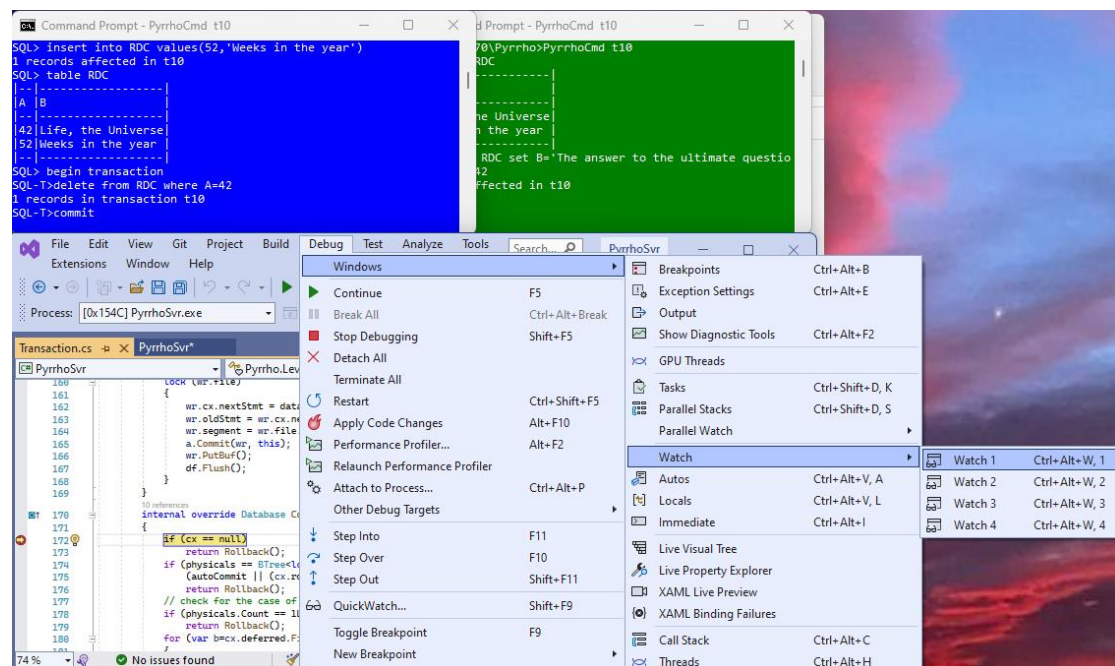
Visual Studio stops at the breakpoint.



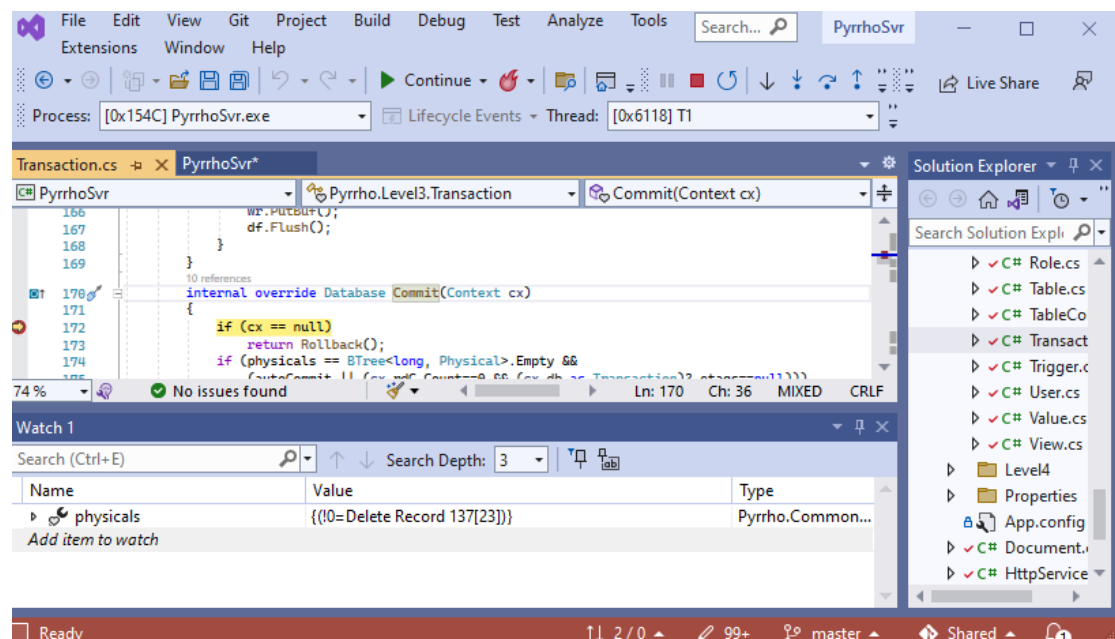


June 2022


Make the Debug>Windows>Watch>Watch 1 window visible,

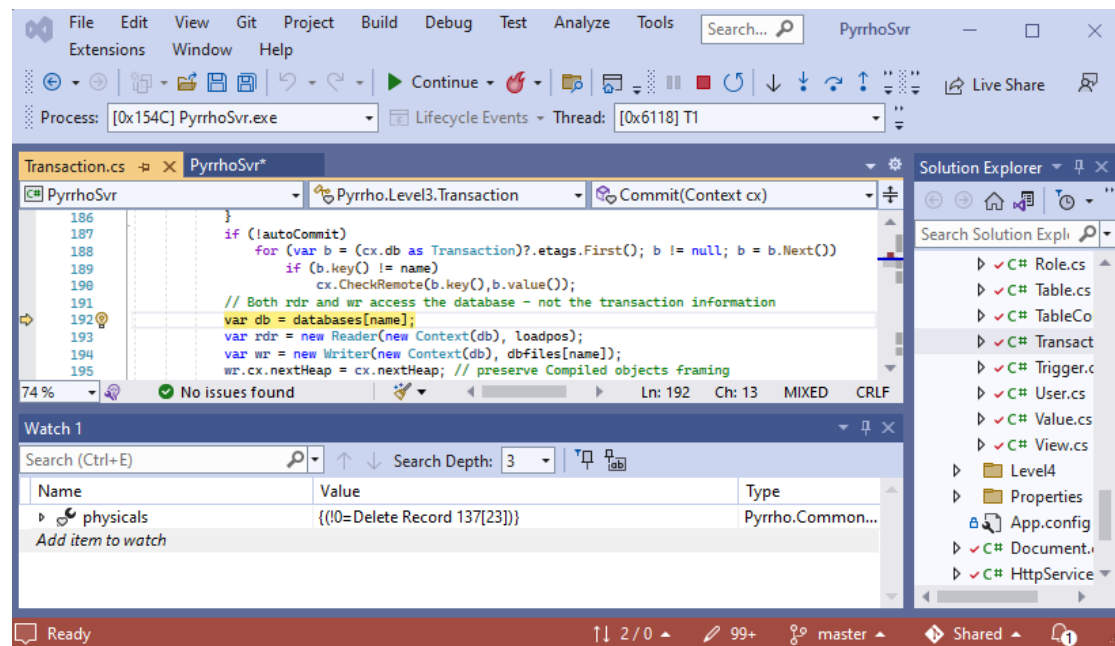


and Dock it below the text window. Use it to examine **physicals**:



**physicals** is the list of Physical records that the Transaction wishes to commit, and it's just the single Physical record to delete row 137 of table 23 in the database, which is "Life, the Universe".

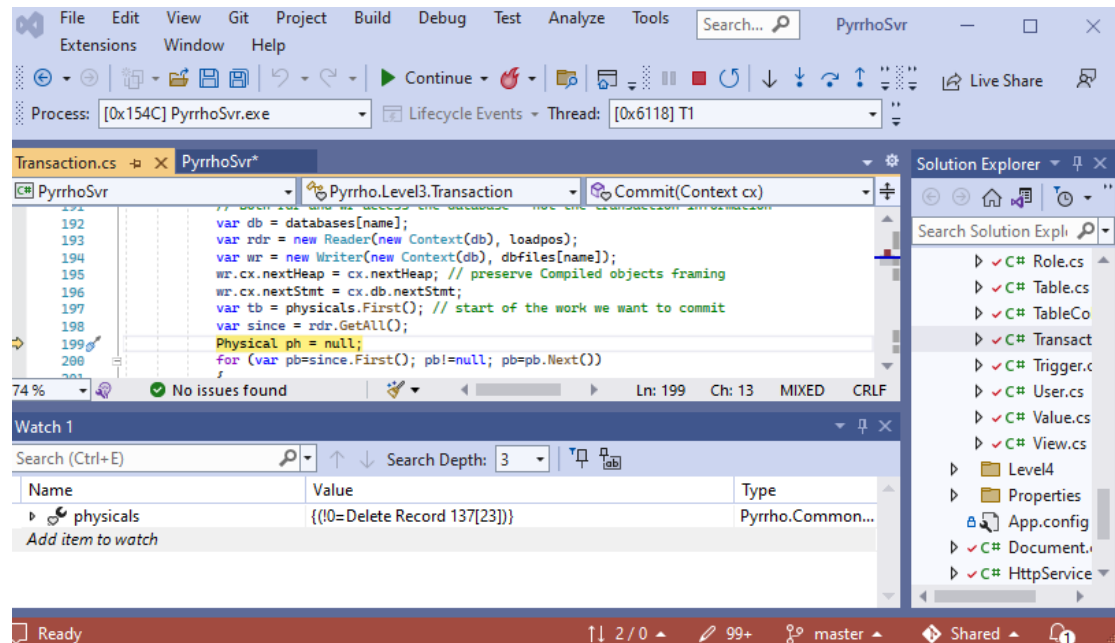
 Step Over a few times to get to line 192.



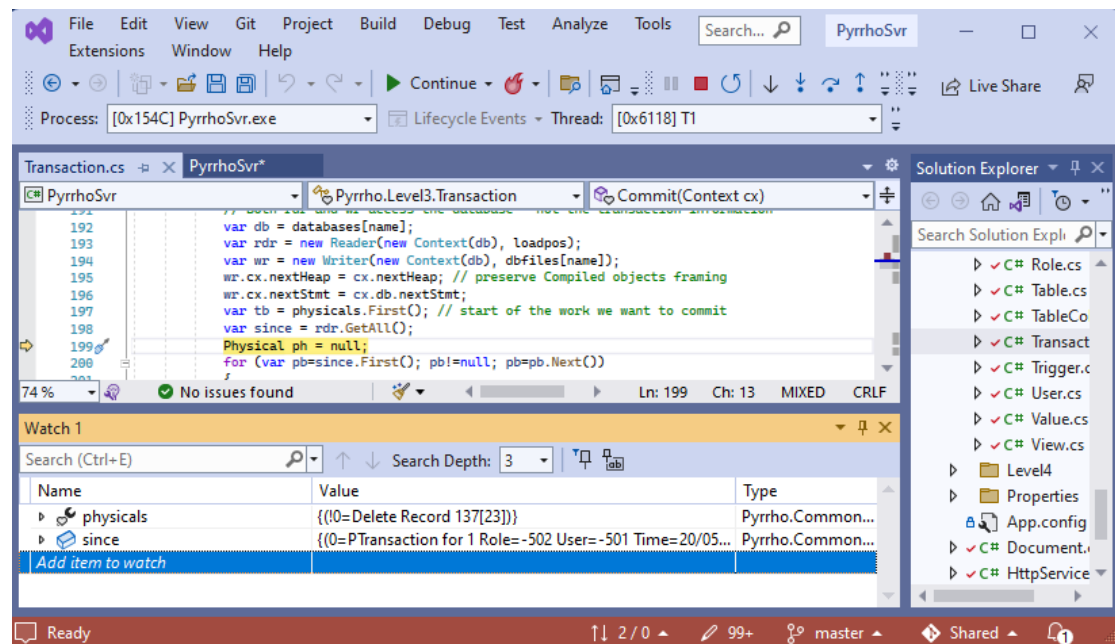
This line of code and the next two do the following. db will be the up-to-date copy of the database, as it was left by the green window. We also open a Reader rdr and a Writer wr: a Reader to look at this database, specifically at the records that have been committed since the start of our Transaction; and a Writer, where we will prepare the records that we are going to add to the database if our validation succeeds.

These are not shareable and are subject to locking protocols. They also work on the same FileStream. (Transaction Commit() repeats the validation step after the locking the FileStream.)

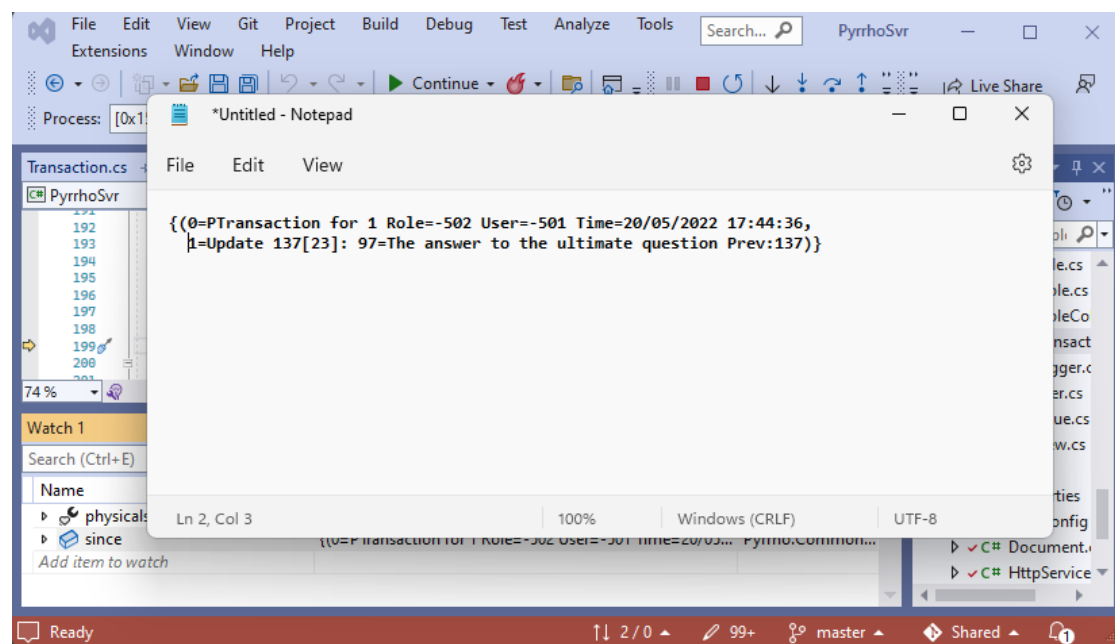
Step Over to line 199, just after the line saying “since=rdr.GetAll();” This gets the records that have been committed to the database since the start of our transaction.



Notice on line 197 that tb starts at the first of our physicals. Use the Watch window to examine **since**.



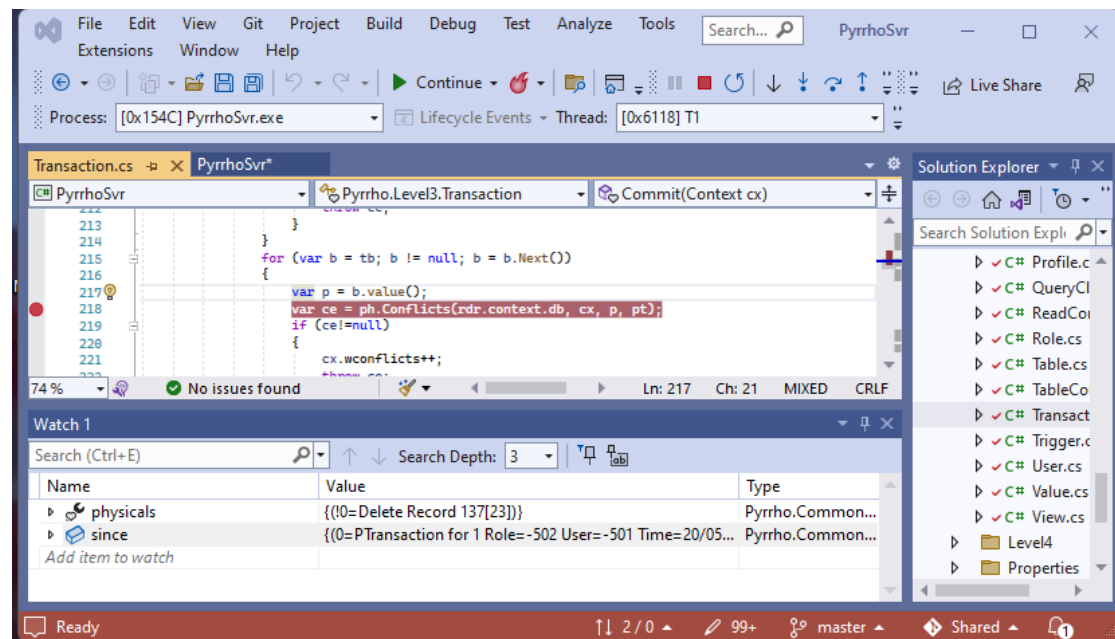
Right-click and copy the value of since into a Notepad. Add a little extra white space to make it more readable.



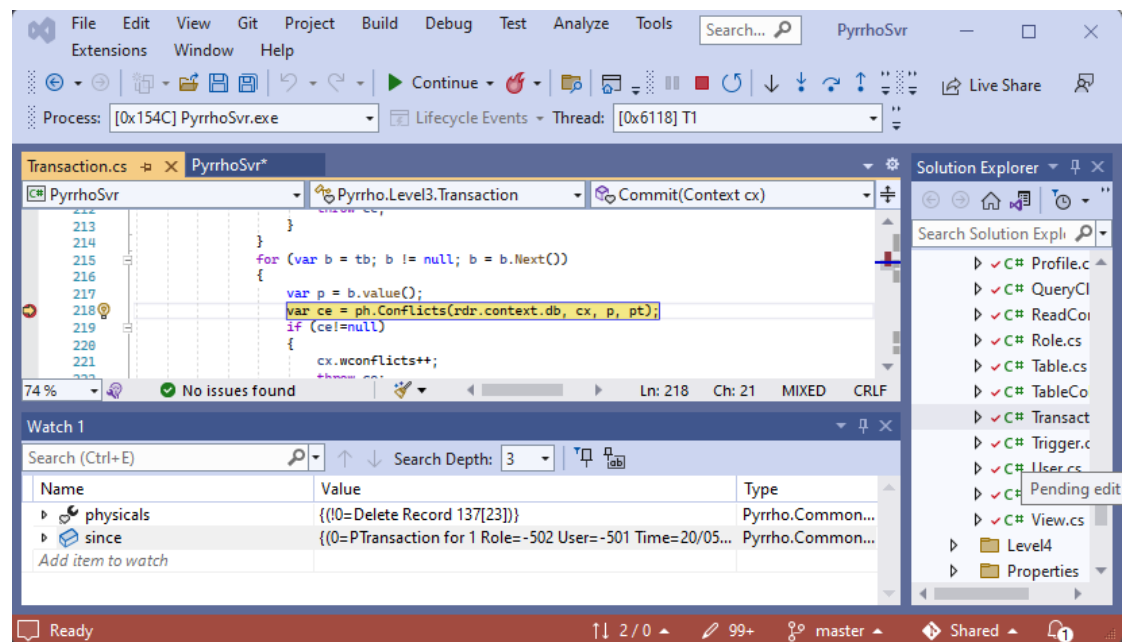
We can see we have got the transaction marker and the update that the green window has made.

The *for* statement we can see in the previous screenshot will look at the records in **since**. Each time round the loop we will examine a record **ph** in **since**.

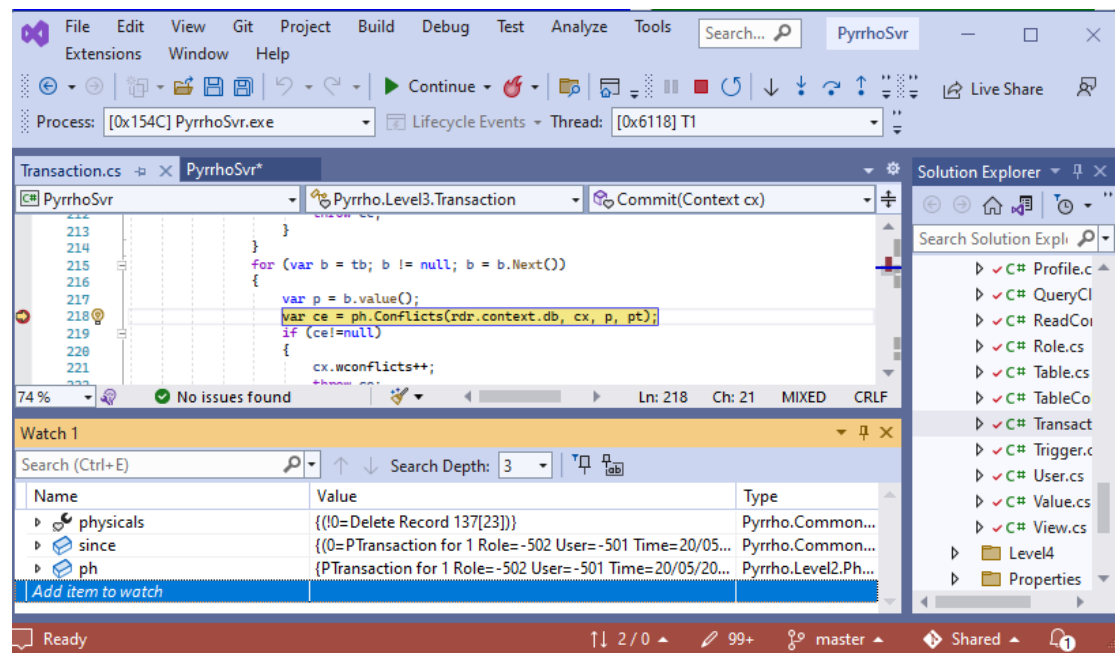
Set a breakpoint at line 218.



Click Continue



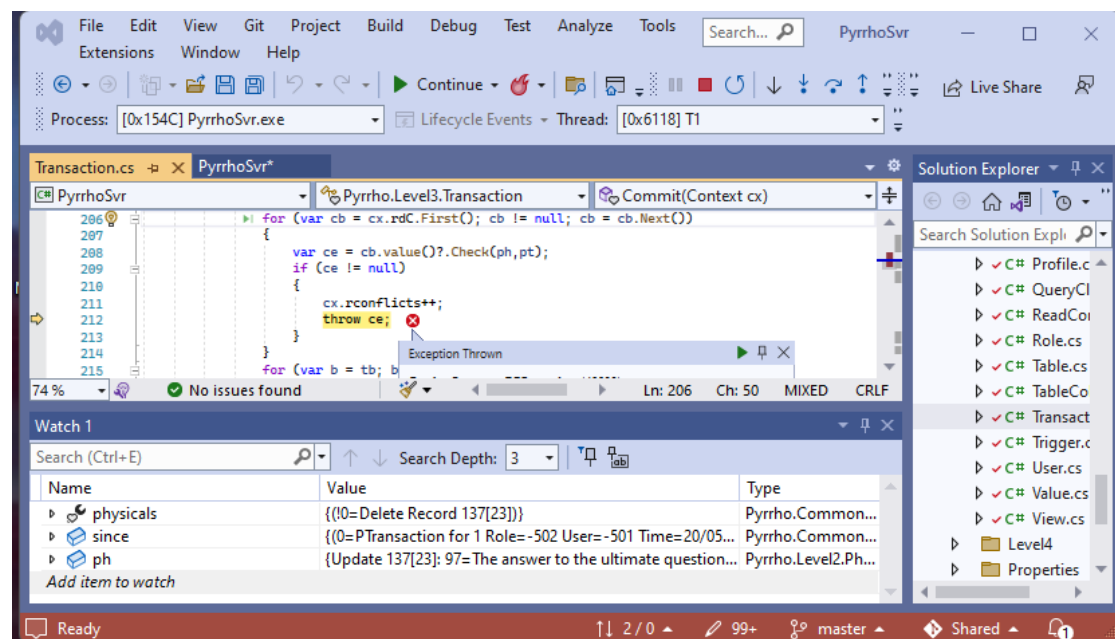
The first time we hit this particular line, ph is the transaction record from since, as we can see in the Watch window,



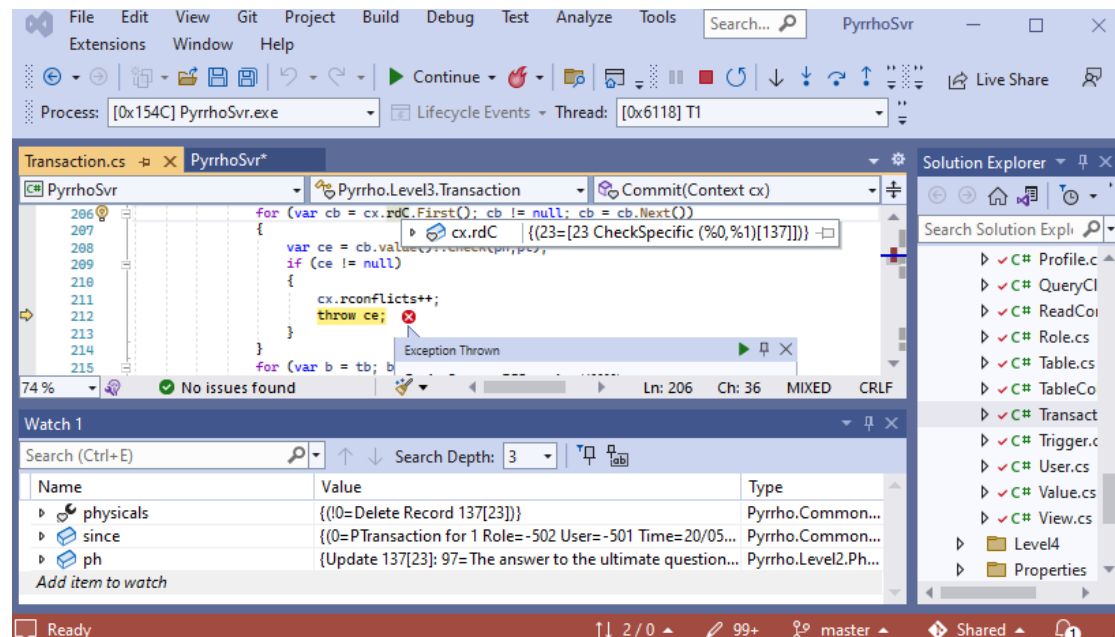
which is not very interesting.

Click Continue

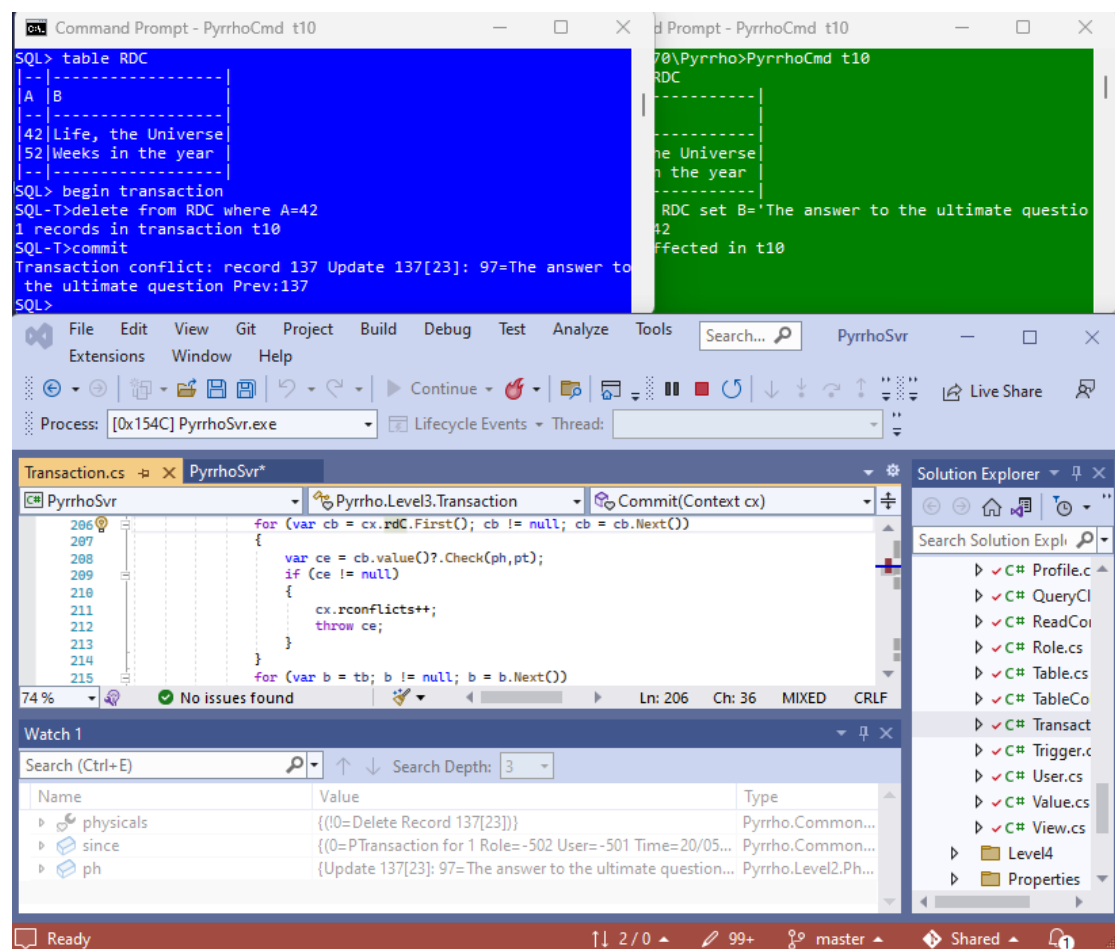
Actually (2022) we don't get to the second time, because the server discovers a read-write conflict first, at line 212!



Hover the mouse over cx.rdC or use the watch window:



We see that the server considers we have read record 137 and this conflicts with the update at 137 done that was done by the other user. (If this hadn't happened the mechanism described in the video would have caught the write-write conflict.) Click Continue.



We see that we get a complaint back in the blue window that record 137 has just been updated. The transaction has been rolled back, as we see from the prompt.

That completes the first experiment that we want to do in this demonstration. As we have already seen a read-write conflict, there is no need for part 3 of this Demo.