An Introduction to the Source Code of the Pyrrho DBMS

Malcolm Crowe, University of the West of Scotland

www.pyrrhodb.com

Version 7.09 (January 2025)

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. This document is for programmers who intend to examine the source code, and includes (for example) details of the data structures and internal locks that Pyrrho uses.

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

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

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

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

Many of the low-level internal details follow the design of StrongDBMS (see strongdbms.com), and all of the upper layers of the Pyrrho engine use shareable data structures similar to StrongDBMS wherever possible. 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 70+ source files in 6 namespaces. The code itself is intended to be quite readable, and uses a recent version (.NET 8.0) of the language.

This document avoids having a section for each class, or for each source file. Instead, the structure of this document reflects the themes of design, with chapters addressing the role in the DBMS of particular groupings of related classes or methods.

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

---

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

1.1 ACID and Serializable Transactions

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

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

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

If we agree that ACID transactions are good, then:

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

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

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

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

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

1.2 The transaction log

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

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

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

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

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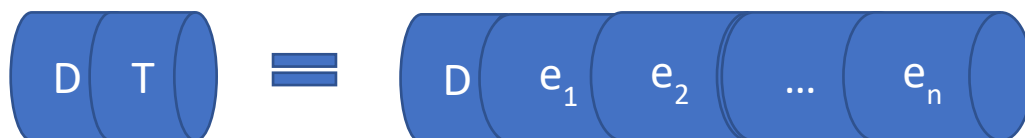
If we think of a transaction commit as comprising a set of elementary operations e, then the transaction log is best implemented as a serialization of these events to the append storage. We can think of these serialized packets as objects in the physical database. In an object-oriented programming language,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

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

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

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

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

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

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

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

1.3 DBMS implementation principles

- If we agree on a globalization strategy, then the DBMS should be neutral, and not specific to a particular machine, platform, or locale.
- A database created on one machine, platform or locale should be usable on another.
- The DBMS should not impose arbitrary size limits on strings, number of columns etc.
- Any that are imposed should be huge (Pyrrho uses 2^{60} as a useful size for such limits).
- If we agree that security is important, then we should use operating system authentication and no other options. Users should not be simply allowed to say who they are.
- The (physical) database should provide support for security, logical, conceptual, application programming, visualisation, internet access and graph models by utilising, and where necessary extending, the advanced features of SQL2023 with metadata and additional primitive types.

Pyrrho DBMS has always had these goals, and from the beginning its feature set included 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. An implementation of typed graph modelling was integrated with the relational system in 2023 (a brief account is in sec 1.18 below and worked examples in sec 6.12).

1.4 Shareable data

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

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

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

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

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

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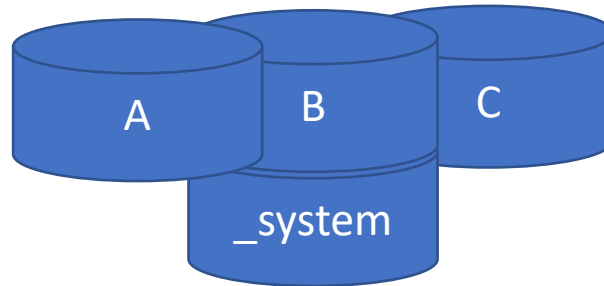
Even more importantly all transactions can start with the current state of the database, without cloning or separately copying any internal structures. When a transaction starts, it starts with the shared database state: as it adds physicals, it transforms. Different transactions will in general start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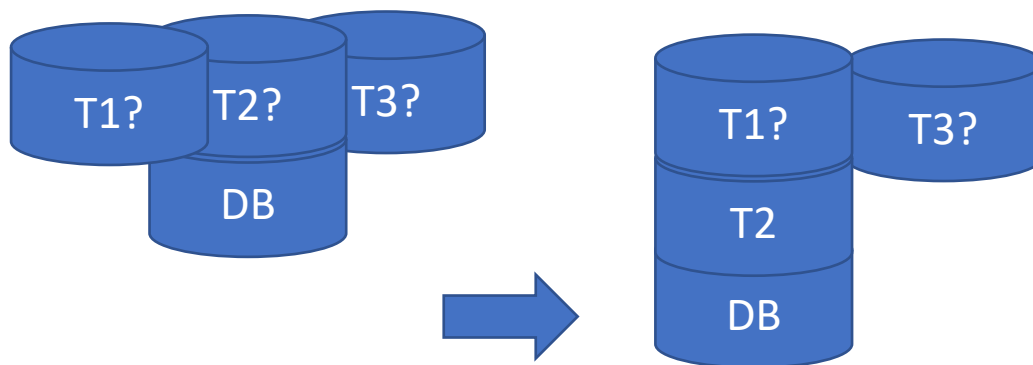
In the above picture, we know what the database DB's state is. Each of concurrent transaction steps T1, T2, and T3 are, if committed, will create a new version of DB (for example (DB)T2.) Because of isolation, from the viewpoint of any of these transactions, they cannot know whether DB has already been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

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

1.5 Transaction conflict

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

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

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

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

1.6 Transaction Validation

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

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

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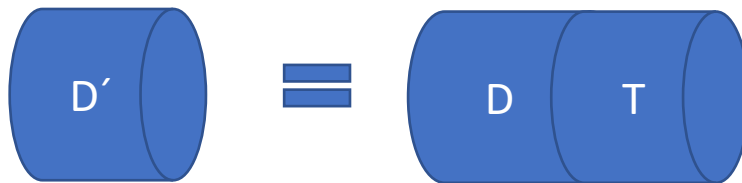
But now we have
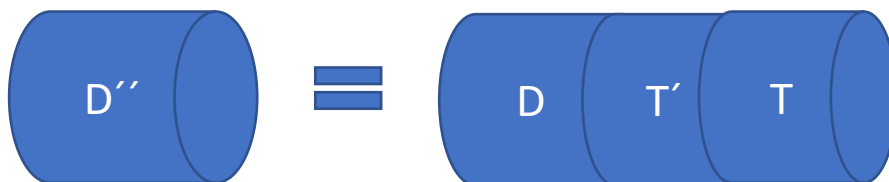
Relocation amounts to swapping the order of the elementary operations e_i that comprise transaction T and T'. Two such cannot be swapped if they conflict. E.g. They change the same object (write/write conflict). The tests for write-write conflicts involve comparing our list of affected physicals with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

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

1.7 Shareable structures

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

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

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

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

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

1.8 Transformation: adding a node

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

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

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

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

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

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

1.9 The choice of programming language

It is helpful if the programming language supports:

- readonly directives (Java has final)
- Generics (Java has these)
- Customizing operators such as $+=$ (not Java)
  - o Because a$+=$b is safer than Add(a,b)
  - o Easy to forget to use the return value a=Add(a,b)
- Implies strong static typing (so not Java)
  - o 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 (see note below).
- TypedValue and all its subclasses

All of these can be made shareable.

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

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

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

1.11 An implementation library: first steps

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

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

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

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

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

1.12 DBObject and Database

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

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

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

1.13 Transaction and B-Tree

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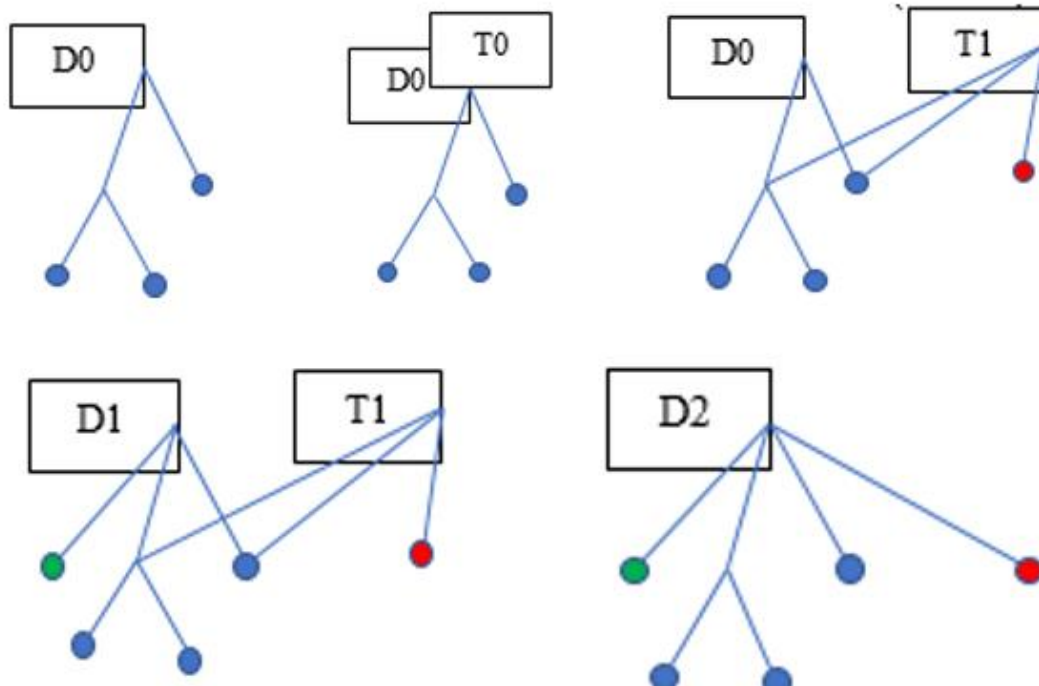
1. Suppose we have a database in any starting state D0.
2. If we start a transaction T0 from this state, initially the T0 is a copy of D0 (i.e. equal pointers).
3. As T0 is modified it becomes T1, which shares many nodes with D0, but not all.
4. D0 also evolves as some other transaction has committed,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

1.14 RowSet review

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

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

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

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

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

predefined constructs, and to take appropriate account of equivalences established by join conditions and similarity of expressions.

All of the above aspects can be subject to delay until all identifiers have been resolved; on the other hand, every time an identifier is resolved there is an opportunity to review the effects on the enclosing rowSet pipeline. The context maintains structures that hold rowsets to be reviewed and properties to be applied, and these are updated along with all other query analysis structures when resolution of identifiers leads to object replacement. See the worked examples in sections 6.8 and 6.9.

1.15 Integrity Constraints

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

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

1.16 Roles, Security, Views and Triggers

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

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

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

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

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

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

---

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

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

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

1.17 Compiled database objects

SQL allows database objects to contain programable code, 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, and deals with a set of specific database objects, with its own set of intermediate results (a heap), and may have handlers for exception conditions.

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

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

The implementation has several groups of classes of objects with multiple subclasses. DBObjects are shareable and can be stored in the database. A DBObject may evaluate in a given Context to give a TypedValue that is also shareable. A DBObject has (or may subclass) a Domain, and a TypedValue has a Domain. Instances of Domains are called QlValues. A RowSet is evaluated a row at a time (using First/Last and Next/Previous): its rows match (or subclass) a given Domain. An Executable in GQL has an incoming Domain (a _Source) and an outgoing _Domain (a Result): it may contain Statements for performing the evaluation.

1.18 The Typed Graph Model

Graph models are typed collections of nodes and edges. This means that node and edge types are defined with particular typed properties including an integer identity, and for edge types, a leaving and an arriving integer property. Each node or edge type has a collection of nodes and edges, and these can be identified with a relational table whose columns of the properties of the node type/edge type, and whose rows are the values of the properties of a particular node.

The leaving and arriving properties of edges can be thought of as connecting the nodes into directed graphs. The leaving and arriving properties behave like foreign keys to a particular node type. Types can have subtypes (using the standard UNDER keyword).

The above description highlights the similarities with the relational model, so that it becomes natural to add the node/edge type behaviour to a relational type by simple metadata added to a standard type declaration with syntax respectively

> NODETYPE ['(' *Identity_*id [CHAR] ')']

Where id is an optional name for the identity column (if not specified, a column of type INT called ID is used, or added if not already specified[5]). The column is automatically the primary key of the node type, but also accesses the database-wide collection of nodes and edges.

EDGETYPE [*Identity_*id] '('[*Leaving_*id '='] *NodeType_*id ',' [*Arriving_*id '='] *NodeType_*id ')'

If not specified, columns of type string called LEAVING and ARRIVING are used or added if not already specified or inherited in the type declaration.

The identifiers ID, LEAVING and ARRIVING are not reserved words, and these columns can be renamed on a per-role basis subject to the usual rules and permissions. The identities of these structural columns are however inherited by subtypes. Columns added to a type in this way are appended to the row type.

---

[5] That is, the optional id in the syntax specifies a name other than the default ID: if such a column is found, it must be of type int; and if no such column is found it will be defined with type int. This footnote also applies to the similar declarations of LEAVING and ARRIVING. In all three cases there will be a new index in the new node or edge type with the simple key column ID, LEAVING or ARRIVING respectively.

The simplest node type (for a new node type called MYTYPE), containing only an identity column, is defined by the SQL statement

CREATE TYPE mytype NODETYPE

Additional columns can be specified in the usual ways, by declaring the new type to be UNDER and existing type and/or adding a clause AS '(' column_list ')' before the metadata part. A subtype of a node or edge type automatically inherits all its properties, so the metadata keywords should not occur in the declaration of a subtype of a node type or edge type. Edge types can be similarly defined in SQL.

GQL's INSERT (or CREATE) statement facilitates rapid creation of graph types, nodes and edges. It uses extra token types for indicating directed arrows for edge connections and a JSON-style notation for providing property lists, so that a single statement can create many node types, edge types, and nodes and edges whose associated tables and columns are set up behind the scenes (in one transaction). Aliases for nodes and edges can be defined following the usual left-to-right conventions. All such database items can be subsequently retrieved using MATCH and modified using SQL DDL statements such as SQL UPDATE, ALTER and DROP[6]. An extra feature allows a graph insert statement to be followed by a THEN clause which allows DDL and DML statements to use the identifiers accumulated during the graph insert. The aliases accumulated during parsing will be discarded once processing of the statement is complete.

Columns for node and edge types can thus be declared in three ways: (a) explicitly in the type clause of CREATE TYPE following the AS keyword, (b) ID, LEAVING and ARRIVING, in metadata in CREATE TYPE, (c) in the graph INSERT statement using previously unknown Node or Edge label(s). In case (c) the values of these properties are also provided for the first node or edge of the new node or edge type.

In all cases, the NodeType.Build method does the actual work of creating the node or edge type and ensuring that the special properties ID (if present[7]), and LEAVING and ARRIVING (for edges) have appropriate columns and indexes in the new type (and its subtypes). Even for a simple type-declaration, the transaction will require several stages of database object construction.

The parsing of these statements results in a set of physical records for the transaction log: (1) PNodeType or PEdgeType giving merely the name of the new type and its immediate supertypes (supertypes keep track of their initially empty list of immediate subtypes); (2) The new columns of the new type, that is, columns not inherited from supertypes; (3) new indexes, for the primary key of the new type if present, and the two special foreign keys[8] of a new edge type (installing these objects will modify the node/edge type to note their uids); and (4) for the graph create statement, Records containing new nodes and edges.

The label part for a node or edge can be a chain of identifiers in supertype to subtype order, but subtypes are first class types in the role and can be referenced on their own. Subtypes inherit the identity column (and leaving, arriving columns for edge types) so that the primary key of the subtype is also the primary key of the supertype. If a chain of labels is used for a new node or edge, any new columns are added to the first type in the chain. A new edge type in a graph insert statement will use the ID columns if any of the nodes it connects[9].

Then supertypes are created before subtypes, node and edge types before edges, and columns before their indexes. The syntax ensures some regularity, and, for the most part, the class structure of the implementation is helpful. But it is useful at this point in the documentation to distinguish the various tasks and how they are supported in the parser. The domain of a type includes columns from supertypes. EdgeType is a subclass of NodeType, so in the discussion of the implementation below we can write node type even if we mean node type or edge type. In this version, nodes (and therefore all records) can specify a list of types (tables) all of which can contribute columns to their tablerow (domain).

---

[6] When a MATCH is obeyed, the dependent statement will be executed for each row of the match. The aliases in a match pattern are rebound for each row traversed for create a current working table. This current working table is also constructed or modified by GQL's INSERT, FOR and RETURN statements.

[7] The ID column if present or inherited must be of type int or string. If present, it will be the primary key for the type and its subtypes.

[8] The leaving and arriving keys will reference the connected nodes by their defining position if these have no primary keys. The leaving and arriving key types may be sets (there is a metadata option to define this if desired).

[9] Thus, if MyEdge is a new edge type created by `MATCH (A:MyLabel{id:3}) CREATE (A)-[:MyEdge]->(:Other)`, the leaving (resp. arriving) types of MyEdge will be MyLabel (resp .Other) even if the matched node A is a subtype of MyLabel. Variants of edge types are automatically created for different connected node types.

The associated metadata (see the next section) needs to be recorded in the transaction log, but the PNodeType and PEdgeType cannot do this, because the columns are created later unless they are inherited from the supertype. Some previously unused fields in the PColumn3 physical record are used to record which special column a new column represents, together with the identity of the relevant index and referenced table if any. This information is picked up by the PColumn.Install routine and updates the NodeType/EdgeType. For an inherited column, the roles are inherited from the supertype, so there is no separate annotation in the log. In all cases corresponding properties are added to the new node or edge type, and the log will show the creation of the new indexes.

By default, open graphs are used: closed graphs can also be specified using GQL's CREATE statements. The above discussion implies that Pyrrho takes a very lenient approach to inferring the intention of any given statement (subject to role privileges) while enforcing any constraints specified in the database, effectively allowing a mixture of GQL and SQL[10]. Labels (introduced by : or IS) in graph insert and match patterns are label expressions: node and edge types are specified by key label sets.

1.18.1 ParseTypeClause and graph type declarations

CREATE TYPE id [UNDER *SuperType*_id] [AS TableContents] [Methods] [*GraphType*_Metadata]

This clearly has five stages of parsing, and we will know that we have a new node type if (a) the UNDER clause specifies an existing node type (in which case metadata cannot be present), or (b) if the metadata part begins with NODETYPE or EDGETYPE. First consider the ordinary create type situation: during the parsing of the AS part if present, a new PType ut with the new columns as specified in the TableContents will have been generated (the new columns collected in ut.rowType) and we will also have the columns inherited from the supertype.

At this point we fix PType/PNodeType/PEdgeType record for the transaction log. As we have not yet seen the metadata section, we will update the PType record to PNodeType/PEdgeType later if need be. The parsing for method type headers comes next.

We parse the metadata if present: the metadata for a Type declaration gives a set of associations CTree<Qlx,TypedValue> as follows (some Qlx keywords look more relevant than others!):

Qlx	TypedValue	Node/Edge
ARROW	name of arriving node type	E
ARROWBASE	present if leaving property is a set of CHAR	E
EDGE	name of ID property if not "ID"	E
LPAREN	name of LEAVING column if not "LEAVING"	E
NODE	name of ID property if not "ID"	N
RARROW	name of leaving node type	E
RARROWBASE	present if arriving propert is a set of CHAR	E
RPAREN	name of ARRIVING property if not "ARRIVING"	E

At this point the ut.rowType does not include any new special columns. Almost all the rest of the work is done by BuildNodeType/BuildEdgeType as described next. In the ParseTypeClause we perform some other tasks around this stage:

Before the BuildNodeType stage, based on ut.domain, we build a set of name/QlValue pairs simplify some of the processing: in the code this list of associations is called ls. We also convert the user-defined type ut we have been preparing to be a NodeType or EdgeType as appropriate (though the properties will be filled in by BuildNodeType/BuildEdgeType. After this stage the PType generated above gets replaced by PNodeType/PEdgeType in FixNodeType/FixEdgeType.

The TableContents part cannot declare columns with the same names as other columns in the existing supertype hierarchy, but the metadata part can adopt existing columns as the special columns (possibly renamed by the metadata; for example, if the supertype had a column AID of type string for its primary key, the metadata could specify NODETYPE(AID)).

---

[10] GQL's reserved words are enforced, while in this version SQL's are not. This means that it is possible for databases to define SQL's reserved words to mean something else, and this may limit the availability of affected SQL syntax.

1.18.2 ParseGraphExp

The following is the syntax used in both CREATE and MATCH:

```
GraphExp =  Node {Edge Node} {',' Node {Edge Node}} .
Node: '(' NodeExp ')'.
Edge: '-[' NodeExp ']->' | '<-[' NodeExp ']-' .
NodeExp = [ Node_id ] [ Label ] [doc] .
Label: ':' Type_id [Label].
```

The new multicharacter tokens in the Edge syntax are respectively called ARROWBASE, ARROW, RARROW and RARROWBASE. The Node preceding Edge in the first syntax is called its leaving node and the Node to the right is called its arriving node, and in the second syntax these are the other way around. Document is as in JSON.

The CreateStatement and MatchStatement are Executable, and both contain a GraphExp. However the result of Match is a rowSet that can be accessed by its dependent clauses.

GraphExp is a list of SqlNode and SqlEdge. SqlEdge is a subclass of SqlNode that identifies the leaving and arriving SqlNodes.

Node_id is an optional Ident for the SqlNode or an QlValue evaluating to a TChar, bearing no relation to the value of ID for the node. The Label part similarly contains Idents or QlValue expressions and is implemented as an SqlTypeExpr. The later parts, as usual, indicate subtypes. The SqlTypeExpr evaluates to a first-class node type name (TTypeSpec/TSubType). All type names must be unique in the role.

The doc part is implemented using a tree structure for the property names and property values, which are themselves QlValues. The property name evaluates to TChar, while the value part is a TypedValue. It can specify ID but should not attempt to define the special properties LEAVING or ARRIVING[11].

If the type name is not found in the database, a new node or edge type will be created whose pathDomain contains the special properties appropriate to a node or edge (automatically generated to match the syntax) and the properties named in the given Document which should either match properties from the supertype if specified, or be new properties specific to the new type. The new type is immediately entered in the transaction's Database.

The main parsing routine for the Graph syntax is ParseGraphExp. It will traverse a single Node or Edge in the syntax. Its parameters consist of the collection of nodes/edges already parsed, and the previous SqlNode in the statement if any. Its return type is a pair consisting of the id for the latest new or referenced node or edge, and a collection of all of the nodes and edges resulting from the CREATE Graph syntax so far, indexed by their lexical position.

In this version the parser simply constructs the above objects without interpreting them. This is done when the CreateStatement/MatchStatement is Obeyed, drawing on the identifiers defined in any preceding QlValues in the current statement.

1.18.3 CreateStatement and MatchStatement

Despite its general-looking name, these Executables are specifically for creating and retrieving typed graphs. They are designed to be called standalone or within executable code where they draw on the Context's ambient values for QlValues. The CREATE statement adds values to be used in later parts of the execution, initially as QlValues but as database records when the create statement is obeyed. During execution, as its name implies, new physical records are added to the transaction.

Each node in the graph expression represents a stage in this process, and the statement is traversed from left to right. However, to enable the SqlType part to be computed, both node endpoints of an edge must be constructed before the edge is constructed. This enables the BuildNodeType method to be called when required. In addition, execution of the Create statement may add properties to a node or edge type, using Alter.

Retrieval/modification of type information and the operation of the autokey mechanism for a missing ID value are the only ways that the database is consulted during execution of the CreateStatement.

As its name implies, traversal of a MatchStatement involves examination of the graph fragments in the database during traversal of its graph expression. Match expressions and matching graph fragments are

---

[11] Note that the SQL syntax allows roles with suitable privileges to manage the names of any type.

traversed from left to right with backtracking. The mechanism uses continuations, so that to each kind of matching (graphs, nodes, patterns, etc.) there corresponds a data structure (GraphStep, NodeStep, PathStep) to control what happens if the match succeeds. Matching a node adds bindings (a stack of values in the context) when a graph expression SqlNode matches a database TNode. In line with the forthcoming GQL standard, bindings are not added if they are duplicates: bindings in repeating patterns are sets rather than lists, and the continuation is not followed if a duplicate is reached, depending on the selected MatchMode.

The Match statement can be used in three ways: as a predicate (returning the single value True, or Null), as a kind of SELECT (returning a rowset showing the sets of bindings that give matching graph fragments in the database), or to contain a DML statement that will be executed for each such row with the bound values of its QlValues. This last mechanism can add physical records to the transaction, but otherwise that MatchStatement itself makes no changes to the transaction.

1.18.4 NodeType.Build

This method is called within ParseTypeClause as mentioned is section 1.18.1, and during execution of CreateStatement. It constructs suitable types in the database, and lays down the physical records for the types, columns and indexes required. That is, we already know the Level3 NodeType structure, but we must create the corresponding structure in the physical records of the database.

The parameters of BuildNodeType are the Level3 node Type being created, an association string to QlValue for the properties of the new node type, and the metadata directives that tell us about any custom versions of special column names. BuildNodeType begins with these.

It is likely that have not the PNodeType/PEdgeType physical records have not yet been constructed (e.g. the we may still have the standard Domain.NodeTypeSpec and Domain.EdgeTypeSpec domains with their negative uids).

We traverse the new row type, ensuring all the special columns have tablecolumns and indexes as appropriate, and then all remaining columns.

For EdgeType we finally fix the leavingType and arrivingType.

Finally, we ensure that the ObInfo and cx.defs information is up to date,

1.18.5 HTMLWebOutput

Pyrrho's HTTPService can be used to output a simple graph showing nodes and edges. For a database *DB* with role *R*, starting with nodetype *NT* and node with id=*ID*, the case-sensitive query syntax is

http://localhost:8180/*DB*/*R*/*NT*/*ID*?NODES

A window size of around 1000x800 px is assumed. The given node will be placed approx 100 px from the left margin. The data returned from the server is generated in Graph.cs by the NodeTable() method: it will be a table with 6 columns, with a row for each node to be displayed, containing the node type name (converted to lower case if it is all caps), and the id in the first two columns, X and Y coordinates for the node in the next two columns (distances in pixels), and for edges the row number of the leaving and arriving nodes in the remaining columns. Nodes whose distance from the specified starting node appears to be more than 1000 px will be excluded from the output.

Script is included that will compute the layout for the graph, given the above semi-computed input. Nodes and edges are displayed as a small circular disk containing the node or edge id, whose background color indicates the node or edge type (a legend for these is displayed), and arrows of length 100 join up the directed edges.

2. Overall structure of the DBMS

2.1 Architecture

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

Namespace	Title in the diagram	Description
Pyrrho		The top level contains only the protocol management files Start.cs, HttpService.cs and Crypt.cs
Pyrrho.Common		Basic data structures: Integer, TypedValue, BTree, and the lexical analyzer for SQL. All classes in Common apart from Exception classes[12], including Bookmarks for traversing them, are immutable and shareable[13].
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[14] are immutable and shareable, and give access to the version of the RowSet that created them.
Pyrrho.Level5	TypedGraph processing	Create and Match for graphs. TGraph, TNode and TEdge are immutable and shareable TypedValues.

2.2 Key Features of the Design

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

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

2. Because data is immutable once recorded, the physical position of a record in the data file (its "defining position") can be used to identify database objects and records for all future time (as names can change, and update and drop details may have a later file position). The transaction log threads

---

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

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

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

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

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

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

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

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

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

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

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

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

11. Executables and QlValues 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 QlValues 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[15].

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

2.3.2 Implementing SQL formats

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

2.3.3 Formats in the in-memory data structures

The DBObject class and its subclasses are described in sections 4 and 5 of this document. During a transaction, the engine is working with a mixture of committed objects and uncommitted DBObjects. Some DBObjects such as Table, or TableColumn correspond to Physical objects such as PTable, PColumn and once committed have defining positions representing their location in the data file. Before commit, they are identified by uids as described in section 2.4. All other properties of DBObjects are managed using a B-Tree of pairs (uid, ob) where ob is a shareable System.Object[16].

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

Compiled objects such as views, routines, constraints etc generally contain DBObjects within them that do not correspond with Physical objects. are parsed on definition and on load from the database file: they are placed in the framing field of their parent DBObject, and their uids are handled quite differently (see

---

[15] 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.

[16] System.Object is not shareable (shareability is not heritable). Most DBObject properties are uid references: the exceptions are Domains (many of which are ad-hoc and lack a uid) and ordering functions.

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

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

Transactions contain uncommitted objects (proposed physical records), whose uids are in the range $2^{62}..5 \times 2^{60}$, 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[18]. 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[19].

---

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

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

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

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 $5\times2^{60}..6\times2^{60}$-1are allocated based on the lexical position of objects in the command text (see worked example in sec 6). Thus, all identifiers that occur in the SQL are replaced during parsing with uids in this range as allocated on the first occurrence in the command text. Columns not referred to will be given temporary uids from the nextHeap range, described below: they cannot use their defining position as this might conflict via a separate reference to the table in the SQL (subqueries, views etc).

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

Several database object types (e.g. Procedure, Check, Trigger, View) define executable code. The physical records in the transaction log record their definition in source form and are called Compiled objects (see sec 3.4.2). During the load phase the executable fragments are parsed, and the resulting executable structures (which have a mixture of lexical[20] and heap uids) are relocated so that their uids are in a contiguous block in the range $6\times2^{60}..7\times2^{60}$-1, denoted `0,`1.. for convenience. Such objects are immutable and shareable, so that any instances (with their view definitions, rowsets etc) will be given new uids in the heap range.

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

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

System objects	$-8\times2^{60}..-1$	Basis._uid (downwards)	Global	
File positions	$0.. 4\times2^{60}$-1	0 (upwards)	Persisted in file	
New Physicals	$4\times2^{60}..5\times2^{60}$-1	Database.nextPos (up)	Local to Transaction	!
Query analysis	$5\times2^{60}..6\times2^{60}$-1	Database.nextIid (up)	Local to Statement	#
Executables	$6\times2^{60}..7\times2^{60}$-1	Database.nextStmt (up)	Local to Database	`
Prepared Stmts	$7\times2^{60}..$ nextPrep	Database.nextPrep (up)	Local to Connection	%
Heap storage	nextPrep.. 8×2^{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:

- 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.

---

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

- 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 stricture. Unusually, it has subfolders for isolating the AssemblyInfo for each of the 3 solutions.

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

- Transaction instances are also created to validate rename, drop and delete operations before these are executed.

3. Basic Data Structures

In this chapter we discuss some of the fundamental data structures used in the DBMS. Data structures selected for discussion in this chapter have been chosen because they are sufficiently complex or unusual to require such discussion. All of the source code described in this section is in the Pyrrho.Common namespace: all of the classes in this namespace are immutable and shareable[21].

3.1 B-Trees and BLists

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

There are several subclasses of BTree used in the database: Some of these implement multilevel indexes. BTree itself is a subclass of an abstract class called ATree. The BTree class provides the main implementation. These basic tree implementations are generic, and require type parameters, e.g. BTree<long,bool> . The supplied type parameters identify the data type used for keys and values. BTree is used when the key type is a IComparable. If the values are also IComparable, CTree is used instead (CTree is then IComparable). In order to report the non-existence of an entry, we use BTree<X_long?> instead of CTree<X,long>.

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

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

3.1.1 B-Tree structure

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

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

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

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

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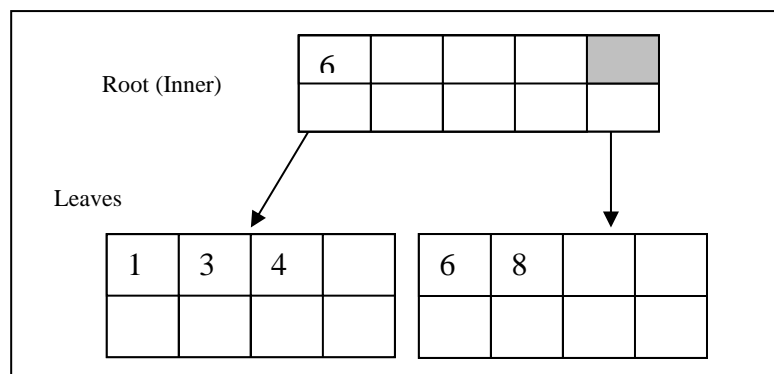
3.1.2 ATree<K,V>

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

---

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

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

The following table shows the most commonly-used operations:

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

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

 t += (key,value);

and to remove a key we write t -= key; . The current version of Visual Studio colours the operator brown to indicate the use of a custom method. Custom operators are static, and so the implementation chosen depends on the declared type of t .

Some B-Trees have values that are also B-Trees, and for these it is convenient to define addition and removal operators for different tuple types (such as triples). BList<V> is a subclass of BTree<int,V> .

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

3.1.3 TreeInfo

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

TreeInfo has the following structure:

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

3.1.4 ABookmark<K,V>

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

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

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

Cursors follow a similar pattern (see section 3.6.10).

3.1.5 ATree<K,V> Subclasses

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

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

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

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

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

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

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

3.2 Other Common Data Structures

3.2.1 Integer

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

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

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

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

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

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

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

3.2.2 Decimal

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

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

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

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

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

3.2.3 Character Data

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

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

3.2.4 Documents

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

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

3.2.5 Domain

Strong types (Domains) are used internally for all processing. Domains are allowed to have columns (such domains can be user defined types or relation types), and parsing and query processing builds ad hoc Domains for results, keys, and signatures. A Domain is committed to the database (with a fixed defining uid) if any of the following is true:

- It is needed to enable an item of committed data to specify its datatype by means of a uid
- It has been declared with CREATE syntax
- It is needed by a stored procedure and is not a simple standard type.

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. Roles can rename most database objects, so that database objects may have several ObInfos for different roles, but, for simplicity, the fields of user-defined data types cannot be changed from those specified by their definer. In the Database layer (level 3), objects have uids but not names. In the Transaction layer (level 4), QlValues and RowSets use the column names for the current role (the current role and user are maintained by the Context).

From version 7.04, perhaps surprisingly, the Table, View, and RowSet classes are *subclasses* of Domain, so that a named table's domain is a subtype of a relation type and contains rows that are of this type. The only difference between the table's type and the relation type is the uid, so that tables with similarly defined columns or even the same named type have disjoint sets of rows, which are assignment compatible. This is useful for the TypedGraph implementation, and otherwise the only noticeable change to SQL behaviour is that such tables inherit and can override domain constraints and (if the type is user-

defined) can override inherited methods and define new ones. Named types are those created by CREATE TYPE, CREATE TABLE, and metadata that adds IRI. All types know their supertype if any and any named subtypes that have been defined. The names of named types (and most other database objects) are role-dependent.

QlValues and RowSet columns are identified by uid (not name), and each uid gives the QlValue used to compute the column. Such QlValues may be simple columns (QlInstance or SqlLiteral) or more complex expressions.

The standard data types have methods of input and output of data, parsing and formatting of value strings, coercing, checking assignability etc., so that a Domain effectively provides similar instructions for data types built from standard data types. The standard types and any unnamed type derived from them are managed in a types tree: from v 7.04 the types tree should not contain named types, which are looked up by name in the role. Standard data types cannot be renamed.

Parsing results in many ad-hoc domain objects being constructed. These need distinct heap uids for the Replace process discussed in sec 6.1[22]. 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

---

[22] The ObTree.ToString() function hides these for brevity, because other objects show their domains

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(...)	*Numerous constructors*
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 QlValues 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 QlValues). The idea is as follows:

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

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

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

3.3 File Storage (level 1)

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

The locking required for transaction management is limited to locking the underlying FileStream during Commit() 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[23]. It uses 8-bit bytes and Unicode UTF8 for strings, but otherwise is independent of machine architecture, operating system, or location.

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

3.3.1 Client-server protocol

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

Request	Intermediate	Final response	
(Connect/Open)		Primary	
(Error)		Exception	
		FatalError	
Authority		*Done*	
BeginTransaction			unacknowledged
CloseConnection			unacknowledged
CloseReader			unacknowledged
Commit	{Warning}	Done	
CommitAndReport CommitAndReport1	{Warning}	TransactionReport	
CommitAndReportTrace CommitAndReportTrace1	{Warning}	TransactionReportTrace	
CommitTrace	{Warning}	DoneTrace	
Execute	{Warning}	Done	
		Schema	
ExecuteTrace	{Warning}	DoneTrace	
		Schema	
ExecuteNonQuery	{Warning}	Done	
ExecuteNonQueryTrace	{Warning}	DoneTrace	
ExecuteReader	{Warning}	Done	
		Schema	
Get	{Warning}	Schema	
Get1		Done	

---

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

		NoData	
Get2	{Warning}	Schema1	
		Done	
		NoData	
GetFileNames		Files	
GetInfo	{Warning}	NoData	
		Columns	
Post	{Warning} Schema	Done	
Put	{Warning} Schema	Done	
Prepare		Done	
ReaderData		NoData	
		ReaderData	
ResetReader		Done	
Rest	{Warning}	Schema	
Rollback		Done	
TypeInfo		(data)	

3.4 Physical (level 2)

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

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

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

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

3.4.1 Physical subclasses (Level 2)

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

Code	Class	Base class	Description
0	EndOfFile	Physical	Checksum record at end of file
1	PTable	Physical	Defines a table name
2	PRole	Physical	A Roleand description
3	PColumn	Physical	Defines a TableColumn or Type field
4	Record	Physical	Records an INSERT to a table
5	Update	Record	Records an UPDATE of a record
6	Change	Physical	Renaming of non-column objects

7	Alter	PColumn2	Modify column definition
8	Drop	Physical	Forget a database object
9	Checkpoint	Physical	A synchronization point
10	Delete	Physical	Forget a record from a table
11	Edit	PDomain	ALTER DOMAIN details
12	PIndex	Physical	Entity, unique, references
13	Modify	Physical	Change proc, method, trigger,check,view
14	PDomain	Physical	Define a Domain
15	PCheck	Physical	Check constraint for something
16	*PProcedure*	*Physical*	*Stored procedure/function (deprecated, see below)*
17	PTrigger	Physical	Define a trigger
18	PView	Physical	Define a niew
19	PUser	Physical	Record a user name
20	PTransaction	Physical	Record a transaction
21	Grant	Physical	Grant privileges to something
22	Revoke	Grant	Revoke privileges
23	PRole1	Physical	Record a role name
24	PColumn2	PColumn	For more column constraints
25	PType	PDomain	A user-defined structured type
26	PMethod	PProcedure	A method for a PType (deprecated, see below)
27	PTransaction2	PTransaction	Distributed transaction support
28	Ordering	Physical	Ordering for a user-defined type
29	(NotUsed)		
30	PDateType	PDomain	For interval types
31	*PTemporalView*	*Physical*	*A View for a Temporal Table (obsolete)*
32	PImportTransaction	PTransaction	A transaction with a source URI
33	Record1	Record	A record with provenance URI
34	PType1	PType	A user-defined type with a reference URI
35	PProcedure2	Physical	(PProcedure2) Specifies return type information
36	PMethod2	PProcedures	(PMethod2) Specifies return type information
37	PIndex1	PIndex	Adapter coercing to a referential constraint
38	Reference	Physical	Adapter coerces to a reference constraint
39	Record2	Record	Used for record subtyping
40	Curated	Physical	Record curation of the database
41	*Partitioned*	*Physical*	*Record a partitioning of the database*
42	PDomain1	PDomain	For OWL data types
43	Namespace	Physical	For OWL data types
44	PTable1	PTable	For OWL row types
45	Alter2	PColumn2	Change column names
46	AlterRowIri	PTable1	Change OWL row types
47	PColumn3	PColumn2	Add new column constraints
48	Alter3	PColumn3	Alter column constraints
49	*PView1*	*Pview*	*Define update rules for a view (obsolete)*
50	Metadata	Physical	Record metadata for a database object
51	PeriodDef	Physical	Define a period (pair of base columns)
52	Versioning	Physical	Specify system or application versioning
53	PCheck2	PCheck	Constraints for more general types of object
54	*Partition*	*Physical*	*Manages schema for a partition*
55	*Reference1*	*Reference*	*For cross-partition references*
56	ColumnPath	Physical	Records path selectors needed for constraints
57	Metadata2	Metadata	Additional fields for column information
58	Index2	Index	Supports deep structure
59	*DeleteReference1*	*Reference*	
60	Authenticate		Credential information for web-based login
61	RestView	View	Views defined over HTTP
62	TriggeredAction		Distinguishes triggered parts of a transaction

63	*RestView1*	*RestView*	*deprecated*
64	Metadata3	Metadata	Additional fields for column information in views
65	RestView2	RestView	Support for GET USING
66	Audit	Physical	Force-written to the log if sensitive data accessed
67	Clearance	Physical	Manage a security property of a user
68	Classify	Physical	Manage a security property of data
69	Enforcement	Physical	Manage a security property of a database object
70	Record3	Record2	Add classification information to a record
71	Update1	Update	For update cascades
72	Delete1	Delete	For delete cascades
73	Drop1	Drop	For drop cascades
74	RefAction	Physical	Manage referential actions on database objects
75	Post	Physical	Manage a posted transaction step
76	*PType2*	*Not used*	
77	PNodeType	PType	A node type in the typed graph model
78	PEdgeType	PNodeType	An edge type in the typed graph model
79	EditType	Edit	Manage the UNDER property for types

3.4.2 Compiled and Framing

Compiled objects include Triggers, Constraints, Tables, Views, UDTypes, Procedures and Methods[24]. Procedures and Methods use the SQL stored persistent modules language as described in the SQL standard, including the handling of conditions (exceptions). When compiled code is invoked, it runs in the definer's role, as specified by the SQL standard.

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

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

There is a subclass of Physical for the Compiled Level2 objects, to provide helper methods for the compilation process. The in-memory data structures resulting from parsing include QlValues, 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 Domain or Table includes the code for constraints, and for UDType includes the 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.

---

[24] Domains could potentially also have constraints, but the syntax does not really allow these to be defined, and so Domain is not currently a subclass of Compiled.

The framing objects, in all cases (apart from Prepared statements, discussed below), form a contiguous block of executable uids. Instancing for these objects (other than views) is simply a matter of adding the framing objects to the current Context. The mapping from View object to instance is illustrated in the following graphic[25]:
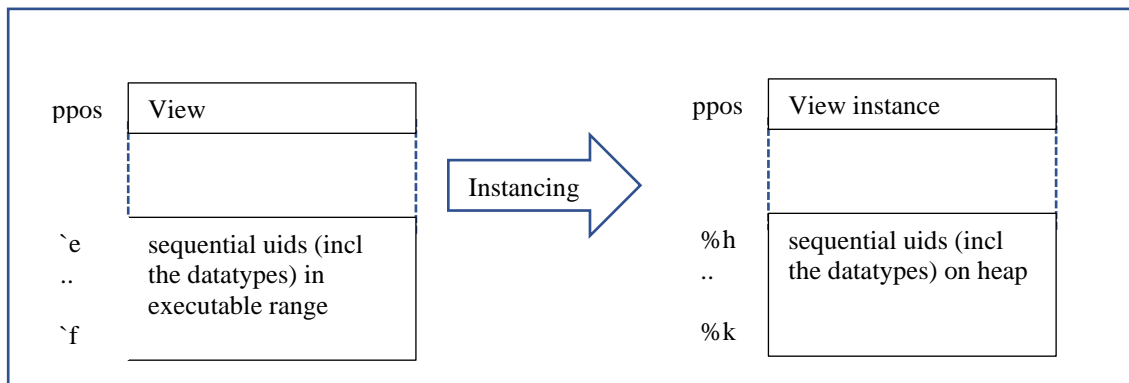
Prepared Statements are simple SQL statements that are immediately parsed into a reserved initial portion of the *heap* uid space. On completion of the parse the object is in the heap range and ready to receive query parameters for immediate execution. The initial prepared statement may contain references 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[26]. 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 ExecuteStatus have been added to help control this process:

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

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

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

---

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

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

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

See the worked examples in sec 6.4-12 below.

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 some subclasses of Basis, including:

Basis subclass	User-defined data
Database, Transaction	DBObjects
SqlRow	Column expressions

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 constant negative uids. It might seem neater if classes had disjoint sets of properties, but this is not practical since classes have subclasses, and it is convenient and more readable when properties with similar semantics are used in other data structures. For example, Databases, SQL Functions and Checks all have names that are strings. Many more DBObjects are renameable, and contain a set of ObInfos giving name information for each role that has been granted access to the DBObject: the definer of the object gives the first entry for this list/

In the code a property uid is defined as follows (using as example ObInfo.Name):

```
internal const long Name = -50; // string
```

The name property is then accessed by

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

This defines name as a readonly property of ObInfo[27].

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)    {
```

---

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

```
        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 += (ObInfo.Name, "xyz");
```

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

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

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

Class	Base Class	Description
Database	Basis	
Transaction	Database	

The level 3 Database structure maintains the following data:

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

- The current position (loadpos) in the associated database file (level 1): at any time this is where the next Commit operation will place a new transaction.

- The id of the user who owns the database

- The list of database objects defined for this database.

All committed DBObjects accessible from the Database class have a defining position given by a fixed position in the transaction log. In v7, the Transaction subclass additionally allows access to its thread-local uncommitted objects, where the defining position is in the range for uncommitted objects, above 2^{62}, and this is derived from the lexical position in all SQL read from the client since the start of the transaction.[29]

In v7.0, many DBObject subclasses are for Query and QlValue 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 QlValue 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 main functionality provided by the Database class is as follows:

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

---

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

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

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 Domain for the result, 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 nextStmt 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.

3.5.4 Role and ObInfo

In Pyrrho, most objects can be renamed on a per-role basis, and accessibility of objects depends on the role. The Role provides a way of looking up objects by name (such as tables, role, types). Prior to September 2022 the Role contained a set of ObInfo containing names, privileges and metadata for each object it can access. Instead, from September 2022, each database object contains an ObInfo for each Role that has been granted access.

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

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

However, 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 -501 $Schema and -502 the engine account[30]. There is also a -55 $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.

---

[30] 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.

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 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.

There is a property called dbobjects that contains the names and defining positions of all top-level named objects accessible by the Role: tables, user-defined types and check constraints. In accordance with the SQL standard, table columns are found in the table's naming information (ObInfo) and method names in a user-defined type's naming information. Procedures have their own list of names, specialised to allow different defining positions for procedures by name and signature (signature is a list of Domains).

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, so that in v7 the Role becomes responsible for all name lookups for level 3 objects. Level 4 objects contains names directly.

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

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

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 QlValues have associated ObInfo. (QlValues always target their definer's Role, and so directly contain name and column information as appropriate.)

3.5.6 Domain and its subclasses

Scalar values are described by a Domain with an empty rowType and representation, as these two properties are reserved for tables and other 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.

A domain with columns models a relation type, and so Domain is the base class for Tables and other structured types. All such types and their columns have role-specific names (see the ObInfo class). Tables are regarded as subtypes of the equivalent unnamed domain.

All Domain classes can have Check constraints. All Domains with columns can have Table properties such as TableCols, TableRows, and Indexes.

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

Class	SuperClass	Notes
Domain	DBObject	
Check	Domain	Check Constraint
EdgeType	NodeType	
ForwardReference	Domain	Could reference a Table or User Defined Type
NodeType	UDType	
RowSet	Domain	See section 3.6.3 for its many subclasses
StandardDataType	Domain	Vanilla versions are predefined
SystemTable	Table	See the Pyrrho manual for the many system tables
Table	Domain	Defines a base table (collects rows of its subtypes)
UDType	Table	User defined types can define methods
View	Domain	Defined columns but does not collect rows
WindowSpecification	Domain	Define a window function

3.5.7 Index

Indexes are constructed when required to implement integrity constraints. There are many types of index, but the most common are PrimaryKey and ForeignKey. Keys can be simple, multi-column or set-valued. The primary key in T gives a single row R(K) in T for each value of the key K, and other indexes with this property implement uniqueness constraints. Foreign keys are more interesting: if a foreign key K of table T references another table U then its value must give a single row R(K) of U, but this process also defines a reverse index from U to T, associating to a row S of U the possibly empty set of rows of T whose primary key K has R(K)=S.

Indexes make it quicker to find rows in a table, and define orderings that can speed up operations such as join.

Additional properties of indexes reflect their uses in ordering and implementing consraints: Descending, RestructUpdate, CascadeUpdate, SetDefaultUpdate, SetNullUpdate, RestruictDelete, CascadeDelete, SetDefaultDelete, SetNullDelete.

SystemTimeIndexes and ApplicationTimeIndexes are also available.

3.5.8 QlValue

QlValues are constructed during parsing, and mostly correspond to SQL syntax nonterminals.

From v7, QlValue is a subclass of DBObject, whose defpos is assigned by the transaction or reader position, or during compilation. QlValues are part of the execution and query processing mechanism. They can be evaluated in a context to yield a TypedValue. Every QlValue has a Domain property.

QlValues may have columns (for example SqlRows), and need not be scalars (for example QlValueSelect).

Class	Subclass of	Comments
BetweenPredicate	QlValue	
ColumnFunction	QlValue	
ExistPredicate	RowSetPredicate	
FormalParameter	QlValue	
InPredicate	QlValue	
LikePredicate	QlValue	
MemberPredicate	QlValue	
NullPredicate	QlValue	
PeriodPredicate	QlValue	
QuentifiedPredicate	QlValue	
RowSetPredicate	QlValue	abstract
SqlCall	QlValue	A procedure call
SqlCaseSearch	QlValue	
SqlCaseSimple	QlValue	
QlInstance	QlValue	Uusally a table column reference
SqlCursor	QlValue	Constructed by DECLARE CURSOR
SqlDefaultConstructor	QlValue	For creating a structured type instance
SqlElement	QlValue	A member of an array

SqlField	QlValue	A member of a structure
SqlFormal	QlValue	A formal parameter
SqlFunction	QlValue	
SqlLiteral	QlValue	
SqlNull	QlValue	NULL: evaluates to TNull.Value
SqlRestValue	QlValue	A value returned by a REST service
SqlRow	QlValue	
SqlRowArray	QlValue	A document array
SqlSecurity	QlValue	A pseudofunction
SqlSelectArray	QlValue	A value of the ARRAY pseduofunction
SqlStar	QlValue	The special selector *
SqlTreatExpr	QlValue	The TREAT pseudofunction
SqlTypeExpr	QlValue	The TYPE pseudofunction
QlValue	DBObject	The base class of the QlValue hierarchy
QlValueArray	QlValue	VALUES
QlValueExpr	QlValue	Deals with unary and binary operators including . [
QlValueMultiset	QlValue	The MULTISET pseudofucntion
QlValueSelect	QlValue	For subquery
QlValueSet	QlValue	The SET pseudofunction (not in the SQL standard)
QlxmlValue	QlValue	The XML pseudofunction
TypePredicate	QlValue	
UniquePredicate	RowSetPredicate	

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

3.5.9 Check, Procedure, Method and Trigger

These are the programming mechanisms provided by SQL. For durability, they are stored in the transaction log in source form, and are compiled at the time of definition or database cold-start. The simplest one is Check, which is an expression provided in a column definition. There is a worked example in section 6.4.

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

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

In this implementation, the following interpretation is made of this section of the standard: Trigger statements may update old and new rows and tables with some obvious restrictions about existence. Changes made by assignment to new row or new table or directly to a column of the table[31] 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.

---

[31] See Note 116 in the SQL standard.

3.5.10 Executable

From v7, Executable is a subclass of DBObject. It has dozens of subclasses as detailed below, some from the SQL specification and others from GQL. Generally, GQL statements support query composition, where a statement following without a separator, or with NEXT, operates on the result of (the working table resulting from) the current statement.

Many Executables can provide a result value (such as a new working table), placed in the Context on execution, return the current or ambient result, or omit any result. Results are unaffected by side effects of evaluation of operands. Current results are from the most recent GqlStatement (including previous or embedded statements).

Class	Result	Comments
AssignmentStatement	Ambient	The right hand side value may modify the current row
BreakStatement	Current	
CallStatement		Without a target procedure, similar to LetStatement. Otherwise, what the target returns with given parameters
CloseStatement	Ambient	For closing a database Connection
CommitStatement	Omitted	Always terminates the transaction
CompoundStatement	Current	
DeleteNode	Current	Gql delete
FetchStatement	Ambient	
FilterStatement	A RowSet	Rows are removed from the current rowSet[32]
ForSelectStatement	Current	Embedded statements are executed for each row of the selection
ForStatement	A RowSet	The unnested values supplied are added as a column to the current rowSet
GetDiagnostics	Ambient	
GraphInsertStatement	Ambient	
Handler	Current	
HandlerStatement	Ambient	
IfThenElse	Current	
IterateStatement	Current	
LetStatement	A RowSet	New columns are added to the current rowSet
LocalVariableDeclaration	Ambient	May update the current bindings
LoopStatement	Current	
MatchStatement	Current	Side effects on the binding table and bindings
MultipleAssignment	Ambient	The right hand side value may modify the current row
OpenStatement	Ambient	For opening a database Connection
OrderAndPageStatement	A RowSet	The current rowSet is ordered or filtered[33] as specified
PreparedStatement	Current	
QuerySearch	Omitted	QuerySearch implements the searched Sql Delete statement
RepeatStatement	Current	
ReturnStatement	A value or Row	A Return statement within a Match statement supplies a row for the current result. In a Procedure, Return gives the result of the procedure.
RollbackStatement	Omitted	Always terminates the transaction
SearchedCaseStatement	Current	
SelectSingle	A Row	As in SQL, but the current rowSet used instead of FROM
SelectStatement	A RowSet	As in SQL, but the current rowSet used instead of FROM
SignalStatement	Current	Result supplied by handler if any
SimpleCaseStatement	Ambient	
SqlInsert	Omitted	
TreatAssignment	Ambient	The right hand side may modify the current row
WhileStatement	Current	
XmlNameSpaces	Ambient	

---

[32] If possible, the server anticipates the filter in an earlier Match calculation.

[33] If possible, the server anticipates limits on the size of the previous Match calculation.

3.5.11 View and RestView

Views differ from Tables in that they do not have a set of tableRows. RestViews are remote views, implemented using HTTP/REST.

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

RestViews

RESTViews, like Views, need to be instanced on reference. Unlike Views, the remote internals of a RESTView are unknown, but we need to be more diligent about optimization of aggregates and selection because of the need to reduce server-server traffic. The Framing for a RESTRView includes details of the remote columns but does not include a RestRowSet. The RestRowSet is constructed during instancing, when the metadata and description for the REST request will be available.

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

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

The basic design is as follows. RestView is a DBObject (subclasses View); PRestView similarly subclasses PView contains the name of the RestView and its file position is used for associating metadata with the RESTView. The column definitions for the RESTView are handled by a "virtual" PTable whose name is the source of the column definitions[34]. The VirtualTable, its columns and domain are contained the framing of the RestView compiled object. Its columns as QlValues. The RestView references the domain and the virtual table, and the domain's structure field references the virtual table.

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

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

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

3.6 Level 4 Data Structures

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

---

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

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 QlValues 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 QlValues, 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 QlValues 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 QlValues during parsing analysis.

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

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

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

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

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

3.6.2 Activation

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

Class	Base Class	Description
Activation	Context	A context for execution of procs, funcs, and methods: local variables, labels, exceptions etc
CalledActivation	Activation	An activation for a procedure or method call, managing a Variables stack
RESTActivation	TargetActivation	Manage REST operation for remote view
TableActivation	TargetActivation	An activation for managing trigger execution
HTTPActivation	TargetActivation	Manage HTTP operation for remote view
TargetActivation	Activation	An activation for controlling insert/update/delete actions
TriggerActivation	Activation	Host for trigger execution

Property	Type	Comments
brk	long	An Activation to break to
cont	long	An Activation to continue to
exceptions	BTree<string,Handler>	
execState	ExecState	saved Transaction and Activation state
ret	TypedValue	
saved	ExecState	
signal	Signal	
locals	BTree<long,object>	

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

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

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

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

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

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

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

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

Note that in other circumstances, fields of structured objects can only be updated by QlValueExpr 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[35]. There is an evaluation pipeline for rowsets, starting with the base tables, applying sorting and joins, aggregation, merging and selection etc, according to a strategy determined during parsing.

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

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[36]. As the RowSet is constructed from its sources, ordering and filtering operations are distributed into a pipeline of RowSets, whose Domain and Sources specify the columns and their origins in other rowsets. At any stage during traversal, the context maintains the current set of cursors.

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

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

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

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

---

[35] They are included in the pre-compiled state of a compiled object that defines local queries (se 3.4.2)..
[36] 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()).

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[37]. The table below shows a number of invariants associated with the rowSet class.

SubClass	Role in pipeline
DistinctRowSet	Remove duplicate rows in the source rowset. The rowType matches its source.
DocArrayRowSet	A rowset whose source is a JSON document. The rowType is a single document column.
EmptyRowSet	A rowset with no source, and no requirements on the rowType.
ExplicitRowSet	A rowset whose source is an array of rows, matching the rowType.
InstanceRowSet	A rowSet with a mapping of its rowType to columns in base tables. (TableRowSet and ViewRowSet are InstanceRowSets)
JoinRowSet	Form the join of two rowsets: the columns are simple and so are the columns of the two sources.
MergeRowSet	Form the union, intersection or EXCEPT of two compatible rowsets The rowType is assignment compatible to the sources.
OrderedRowSet	A rowset formed by reordering the rows in the source. The rowType matches the source.
RoutineCallRowSet	A rowset whose source is a call to a procedure or method
RowSetSection	A rowset forned by selection from the source by row sequence. The rowType matches the source.
SelectRowSet	A rowset formed by selection of rows using SQL expressions. The select list can contain aggregations.
SelectedRowSet	A rowset formed by selection of certain columns from the source. The columns are simple, but may have a different order from the source.
SqlRowSet	A rowset whose source is a list of row-valued SQL expressions The rowType is assignment-compatible with the source.
SystemRowSet	A rowset constructed from data structures in the server
TableRowSet	An instanced rowset whose source is a base table. The rowType matches the source table as seen by the role.
TransitionRowSet	For input/update/delete operations (constructed by TargetActivation)
TransitionTableRowSet	The rowset accessed by OLD TABLE and NEW TABLE during trigger operation. The rowType matches the enclosing transition row set.
TrivialRowSet	A rowset consisting of a single SQL row
ValueRowSet	A list of rows provided elementwise
ViewRowSet	An instance rowset whose source is a local or remote view. The rowType matches the view definition as seen by the role.
WindowRowSet	A rowset from application of a window function to the source rowset

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

Name	Type	Definition	Uid
_tgt	PTrigger.TrigType	TransitionRowSet.TriggerType	-421
_trs	TransitionRowSet	TransitionTableRowSet.Trs	-431
actuals	BList<long?>	RoutineCallRowSet.Actuals	-435
adapters	Adapters	TransitionRowSet._Adapters	-429
built	bool	RowSet.Built	-402
data	BTree<long,TableRow>	TransitionTableRowSet.Data	-432
defaultURL	string	RestRowSet.DefaultURL	-379
defaults	BTree<long,TypedValue>	TransitionRowSet.Defaults	-415
distinct	bool	RowSet.Distinct	-239
docs	BList<QlValue>	DocArrayRowSet.Docs	-440
domain	Domain	DBObject._Domain	-176

---

[37] TransitionRowSet and some system rowsets are unidirectional.

explRows	BList<(long,TRow)>	ExplicitRowSet.ExplRows	-414
filter	PRow	FilterRowSet.IxFilter	-411
first	long	JoinRowSet.Jfirst	-447
from	From	TransitionRowSet,TrsFrom	-416
groupings	BList<long?>	RowSet.Groupings	-406
groupSpec	GroupSpecification	RowSet.Group	-199
having	BTree<long,bool>	RowSet.Having	-200
index	long	IndexRowSet._Index	-410
indexdefpos	long	TransitionRowSet.IxDefPos	-420
join	JoinPart	JoinRowSet._Join	-446
joinCols	BTree<string,int>	RestRowSet.JoinCols	-383
keys	CList<long?>	Index.Keys	-159
lastData	long	Table.LastData	-258
map	BTree<long,long?>	TransitionTableRowSet.Map	-433
matches	BTree<long,TypedValue>	Query._Matches	-182
mtree	MTree	Index.Tree	-164
needed	CTree<long,bool>	RowSet._Needed	-401
offset	int	RowSetSection.Offset	-438
proc	Procedure	RoutineCallRowSet.Proc	-436
ra	TriggerContext	TransitionRowSet.Ra	-424
rb	TriggerContext	TransitionRowSet.Rb	-422
remoteAggregates	bool	RestRowSet.RemoteAggregates	-384
remoteCols	BTree<string,long?>	RestRowSet.RemoteCols	-373
remoteGroups	GroupSpecification	RestRowSet.RemoteGroups	-374
restValue	TArray	RestRowSet.RestValue	-457
restView	long	RestRowSet.RestView	-459
result	RowSet	RoutineCallRowSet.Result	-437
ri	TriggerContext	TransitionRowSet.Ri	-423
rmap	CTree<long,long?>	TransitionTableRowSet.RMap	-434
row	TRow	TrivialRowSet.Singleton	-405
rowOrder	CList<long?>	RowSet.RowOrder	-404
rows	Blist<TRow>	RowSet._Rows	-407
rt	CList<long?>	(domain.rowType)	
second	long	JoinRowSet.Second	-448
size	int	RowSetSection.Size	-439
source	long	From.Source	-151
sqlRows	BList<long?>	SqlRowSet.SqlRows	-413
ta	TriggerContext	TransitionRowSet.Ta	-426
table	long	IndexRowSet.IxTable	-409
tabledefpos	long	SqlInsert._Table	-154
targetAc	Activation	TransitionRowSet.TargetAc	-430
targetTrans	CTree<long,long?>	TransitionRowSet.TargetTrans	-418
tb	TriggerContext	TransitionRowSet.Tb	-425
td	TriggerContext	TransitionRowSet.Td	-428
transTarget	CTree<long,long?>	TransitionRowSet.TransTarget	-419
tree	RTree	OrderedRowSet._RTree	-412
usingCols	BTree<string,long?>	RetRowSet.UsingCols	-259
usingTable	long	RESTRowSet.UsingTable	-260
values	Tmultiset	WindowRowSet.Multi	-441
wf	SqlFunction	WindowRowSet.Window	-442
where	BTree<long,bool>	Query.Where	-190

3.6.4 Cursor

Previously called RowBookmark, this is an abstract and immutable subclass of TRow for traversing rowSets. All RowSets offer a First() that returns a Cursor at position 0, or null, and a Last() that returns a Cursor at the end of the rowset, or null. Cursors are immutable, but their values can be updated (as

usual giving a new cursor, stored in the appropriate context). Note however that an updated cursor continues to traverse the rowset as it was at the start of traversal.

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

The interface offered includes the following:

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

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

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

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

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

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

3.6.5 Rvv

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

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

3.6.6 ETags

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

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

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

4. Locks, Integrity and Transaction Conflicts

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

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

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

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

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

4.2 Transaction conflicts

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

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

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

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

4.2.1 ReadConstraints

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

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

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

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

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

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

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

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

4.2.2 Physical Conflicts

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

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

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

4.2.3 Entity Integrity

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

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

- The set of key columns

- The table (defpos)

- The new key as an array of values

- A pointer to the next IndexConstraint.

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

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

4.2.4 Referential Integrity (Deletion)

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

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

- The set of key columns in the referencing table

- The defining position of the referencing table (refingtable)

- The deleted key as an array of values

- A pointer to the next ReferenceDeletionConstraint.

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

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

4.2.5 Referential Integrity (Insertion)

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

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

- The set of key columns in the referenced table

- The defining position of the referenced table (reftable)

- The new key as an array of values

- A pointer to the next ReferenceInsertionConstraint.

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

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

4.4 System and Application Versioning

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

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

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

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

5. Parsing

Pyrrho uses LL(1) parsing for all of the languages that it processes. The biggest task is the parser for SQL2011. There is a Lexer class, that returns a Qlx 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[38]:

- The transaction has received SQL from the client.

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

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

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

---

[38] 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. Qlx.SELECT, Qlx.ID, Qlx.COLON etc)

- tgs, TGParam values and positions for assisting GQL with graph pattern syntax

- val, the value of the current token, e.g. an integer value,the spelling of the Qlx.ID etc.

The Lexer checks for the occurrence of reserved words in the input, coding the returned token value as the Qlx enumeration. These Qlx 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 Qlx enumeration might result in database files being incompatible with the DBMS, (b) the enumeration contains synonyms such as Qlx.INT and Qlx.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:

Qlx id	Fixed Value	Synonym issues
ARRAY	11	
BOOLEAN	27	
CHAR	(37 recode)	Always recode CHARACTER and STRING to CHAR
CLOB	40	
CURSOR	65	
DATE	67	
EDGETYPE	461	
INT	(128 recode)	Always recode INT to INTEGER
INTEGER	135	
INTERVAL0	137	Former version of INTERVAL
INTERVAL	152	
METADATA	514	
MULTISET	168	
NCHAR	171	
NCLOB	172	
NODETYPE	534	
NULL	177	
NUMERIC	179	
REAL0	199	Previous version of REAL
REAL	203	
PASSWORD	218	
SET	255	
TIME	257	
TIMESTAMP	258	
TYPE	267	
TABLE	297	
XML	356	

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

5.3 Parser

The parser retains the following data:

- The Lexer

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

- The current Context, including the current Database or Transaction

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

5.3.1 Execute status and parsing

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

Execute Status	Purpose of parsing
Parse	Parse or reparse of stored procedure body etc. Execution of procedure body uses the results of the parse (Execute class)
Obey	Immediate execution, e.g. of interactive statement from client
Graph	Match pattern parsing
GraphType	LIKE Graph parsing
Prepare	Prepare implementation
Compile	Compile of source fragments
Commit	Commit has a special state for relocating to transaction file positions

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.
QlValue ParseQlValue(sql,type)	Parse an SQLTyped Value
QlValue ParseQlValueItem(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 uids called 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 #)[39] is allocated a uid on the context's heap (displayed with %), and new heap uids for an instance of its structure. On Commit of a newly defined object, when lexical uids are allocated unused positions in the file position range, these heap uids are relocated to the executable range (displayed with `).

Reference to compiled objects during database load does not require reparsing of their definition, but instancing of their framed objects, using new uids in the executable range.[40]

The SQL parsing process is recursive. During the lexical (left-to-right) phase of analysis the lexer supplies defining positions for unknown QlValue it encounters: these are used as uids during parsing[41]. 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.

---

[39] 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.
[40] For Views defined in terms of other views, see the discussion and example in section 6.6 below.
[41] QlValues use iix (a combination of lexical and defining position) as an enhanced sort of defpos.

As might be expected, this process is not simple. Even in the simplest queries, left-to-right parsing means that identifiers can remain undefined for a long time, and this delays rowset review.

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

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

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

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

6.1.1 Context and Ident management

One of the main purposes of parsing is to replace identifiers by uids. Parsing proceeds from left to right, and initially objects are referenced by an identifier chain. Many database objects have a namespace of associations from role-dependent string to object uid, and during parsing their namespace can be pushed onto or popped from the cx.defs stack. cx.names is a volatile list of the currently defines names, and is used in conjunction with an "ambient position" which is the start of the current scope. Thus on encountering a simple identifier that is in cx.names with a defining position in current scope, it is a reference, and otherwise it will be an unbound name and the current position in the syntax will provide its definition. The rules for identifier chains are given in the next section.

This namespace stack consists of name information for the current level and current names and the list of names at lower levels: BTree<string,long?> and BTree<long,BTree<string,long?>>. Each persistent object, domain and QlValue remembers its components in infos[cx.role.defpos], as object names are role-dependent.

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 SqlReview. Each freshly defined component of the identifier chain has a lexical position, and the ForwardReference objects may have suggested row Domains. Now see 3) below.

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) *Resolve*. When the select depth changes, the corresponding scope in cx.defs is reviewed and values are updated.

4) *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.

6.1.3 Alias and Subquery

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

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

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

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

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

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

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

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

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

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

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

6.1.4 Replacement rules

When we find out more, the DBObject will be replaced: the lexical uid won't change, but the uid may be replaced by that of the resolved object. The rules for replacement of lexically-positioned objects 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 ForwardReference, a Variable, a RowSet, a Cursor, a Table, or an QlValue that has a well-defined source that matches the identifier chain. (If a ForwardReference is being replaced, we check all of its Subs list is well-defined.)

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.

5. For parsing to be complete, the Context should contain no undefined objects or objects with unsatisfied identifier chains. An identifier chain is unsatisfied if it references an object other than itself. Such an object should have been replaced with a well-defined object whose identifier chain is empty or only refers to itself.

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 QlValues by updating the from information; finally

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

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

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

6.1.6 A worked example

An example will help to explain the analysis process. Suppose the database defines (only)

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

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

23	AUTHOR	a Table
34	ID	for 23(0)[INTEGER}
56		an Index on 23(34) PrimaryKey
77	ANAME	for 23(1)[CHAR]
120	BOOK	a Table
129	ID	for 120(0)[INTEGER]
152		an Index on 120(130) PrimaryKey
171	AID	for 120(0)[INTEGER]
195		an Index on 120(173) ForeignKey, RestrictUpdate, RestrictDelete refers to [57]
212	TITLE	for 120(0)[CHAR]

The uid for a database object read from the database file will be its defining position in the file (the position in the transaction log, e.g. 23 for AUTHOR).[42] The Log shows the physical records in the database file, with the definer's names for these objects.[43]

Insert a few records in these tables:

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

---

[42] The parenthesised figure in the column declaration is a column ordinal: if this is -1 the next available position will be used.
[43] There is no need for objects to define standard data types such as INTEGER and CHAR.

```
insert into book(aid,title) values (1,'Dombey & Son'),(2,'Lord Jim'),(1,'David Copperfield')⁴⁴
```

Then, when we parse

```
123456789012345678901234
SELECT aname FROM author
```

aname is initially constructed as a QlValue with uid #8 of subclass SqlReview (i.e. currently undefined) and domain CONTENT. The uid for a new object in a query is its lexical position in the command (e.g. #8 for an identifier starting at character position 8 in the source), or its transaction id, (e.g. !0 for the first persistent object created in the transaction), a defining position in the transaction log (e.g. 23 for table AUTHOR), or a heap uid (e.g. %0 for the first object not in any of these categories).

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

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

where we see that the identifier ANAME will be from an object #1 yet to be built (this is the lexical position of the SelectStatement, built below), and we have a very tentative definition of its domain since the parser so far does not know what table will be used. The names table in the Context contains only this undefined identifier:

```
        {(ANAME=#8)}
```

At the end of ParseTableReferenceItem(), after passing the table name AUTHOR and looking it up in the database, at about line 8400 in Parser.cs, the context has constructed a TableRowSet rf for AUTHOR and QlValues for the column that has been referenced:

```
{(23=Table  TABLE  (34,77)[34,Domain  INTEGER],[77,  CHAR]  rows  2  Indexes:((34)56)  KeyCols:
(34=True),
  34=TableColumn 34 Definer=-502 LastChange=34  INTEGER Table=23,
  77=TableColumn 77 Definer=-502 LastChange=77  CHAR Table=23,
  #8=QlInstance ANAME #8[ANAME] Domain CHAR From:#19 copy from 77,
  #19= TableRowSet #19 AUTHOR TABLE (%1 INTEGER,#8 CHAR) Display=2 Indexes=[(%1)=[56]] key (%1)
targets: 23=#19 From: #19 Target=23 SRow:(34,77) Target:23 AUTHOR,
  %0=Domain TABLE (#8) Display=1[#8,Domain CHAR],
  %1=QlInstance %1 Domain INTEGER ID From:#19 copy from 34)}
```

The blue colour indicates objects that were read from the database file on first load of the database and are shared with any references to these objects. The red colour indicates changes since the previous listing.

When ParseStatement returns (line 203), the SelectStatement has been constructed:

```
{(23=Table  TABLE  (34,77)[34,Domain  INTEGER],[77,  CHAR]  rows  2  Indexes:((34)56)  KeyCols:
(34=True),
  34=TableColumn 34 Definer=-502 LastChange=34  INTEGER Table=23,
  78=TableColumn 78 Definer=-502 LastChange=78  CHAR Table=23,
  #1=SelectRowSet #1 AUTHOR TABLE (#8 CHAR) Display=1 targets: 23=#19 From: #19 Source: #19,
  #8=QlInstance #8  CHAR ANAME From:#19 copy from 77,
  #19= TableRowSet #19 AUTHOR TABLE (%1 INTEGER,#8 CHAR) Display=2 Indexes=[(%1)=[56]] key (%1)
targets: 23=#19 From: #19 Target=23 SRow:(34,77) Target:23 AUTHOR,
  %0=Domain TABLE (#8) Display=1[#8, CHAR],
  %3=SelectStatement %3 Union=#1)}
```

The paler colours shows are used for objects that have not been changed (eventually unchanged lines in these listings are replaced by ellipses ..).

We see that the source of the SelectRowSet is our TableRowSet #19, and its target is table 23. This completes the analysis, as parsing has reached the end of the query: if we allow the execution to proceed, we see the results returned to the client.

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

---

⁴⁴ Note that this insert statement does not provide the book ID (a primary key). Pyrrho will supply suitable values.

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

Before parsing the From clause, at line 8190, we have in cx.obs:

```
{(#8=SqlReview ANAME #8[ANAME]  CONTENT From:#1,
 #14=ForwardReference B #14 Definer=-502⁴⁵[B,TITLE] Subs: (#16=True),
 #16=SqlReview B.TITLE #16[B,TITLE]  CONTENT From:#1 Alias=C,
 %0=Domain TABLE (#8,#16) Display=2[#8, CONTENT],[#16, CONTENT])}
```

We see an SqlReview for each of the currently unknown identifiers ANAME and TITLE, and now also
a ForwardReference object for B. After constructing the table reference item AUTHOR A, at line 8400
in Parser.cs, we have (much as before)

```
{(23=Table TABLE (34,77)[34,Domain INTEGER],[77,Domain CHAR] rows 2 Indexes:((34)56) KeyCols:
(34=True),
 34=TableColumn 34 Definer=-502 LastChange=34 Domain INTEGER Table=23,
 77=TableColumn 77 Definer=-502 LastChange=77 Domain CHAR Table=23,
 #8=QlInstance ANAME #8[ANAME] Domain CHAR From:#32 copy from 77,
 #14=ForwardReference B #14 Definer=-502[B,TITLE] Subs: (#16=True),
 #16=SqlReview B.TITLE #16[B,TITLE]  CONTENT From:#1 Alias=C,
 #32=TableRowSet #32 AUTHOR TABLE (%1 INTEGER,#8 CHAR) Display=2 Indexes=[(%1)=[56]] key (%1)
targets: 23=#32 From: #32 Target=23 SRow:(34,77) Target:23 AUTHOR Alias: A,
 %0=Domain TABLE (#8,#16) Display=2[#8,Domain CHAR],[#16, CONTENT],
 %1=QlInstance %1 Domain INTEGER ID From:#32 copy from 34)}
```

At line 8214, we have constructed the cross join specified, identified B.TITLE, and noted that aliases for
the two columns called ID might come in useful:

```
{(..
 120=Table TABLE (129,171,212)[129,Domain INTEGER],[171,Domain INTEGER],[212,Domain CHAR] rows
3 Indexes:((129)151;(171)195) KeyCols: (129=True,171=True),
 129=TableColumn 129 Definer=-502 LastChange=129 Domain INTEGER Table=120,
 171=TableColumn 171 Definer=-502 LastChange=171 Domain INTEGER Table=120,
 212=TableColumn 212 Definer=-502 LastChange=212 Domain CHAR Table=120,
 #8=QlInstance #8 Domain CHAR ANAME From:#32 Alias=A.ANAME copy from 77,
 #14=ForwardReference B #14 Definer=-502[B,TITLE],
 #16=QlInstance B.TITLE #16[B,TITLE] Domain CHAR From:#42 Alias=C copy from 212,
 #32=TableRowSet #32 AUTHOR TABLE (%1 INTEGER,#8 CHAR) Display=2 Indexes=[(%1)=[56]] key (%1)
targets: 23=#32 From: #32 Target=23 SRow:(34,77) Target:23 AUTHOR Alias: A,
 #40=JoinRowSet #40 (%1 INTEGER,#8 CHAR,%2 INTEGER,%3 INTEGER,#16 CHAR) Display=5 targets:
23=#32,120=#42 CROSS First: #32 Second: #42,
 #42=TableRowSet  #42  BOOK  TABLE  (%2  INTEGER,%3  INTEGER,#16  CHAR)  Display=3
Indexes=[(%2)=[151],(%3)=[195]]  key  (%2)  targets:  120=#42  From:  #40  Target=120
SRow:(129,171,212) Target:120 BOOK Alias: B,
 %0=Domain TABLE (#8,#16) Display=2[#8,Domain CHAR],[#16,Domain CHAR],
 %1=QlInstance %1 Domain INTEGER ID From:#32 Alias=A.ID copy from 34,
 %2=QlInstance %2 Domain INTEGER ID From:#42 Alias=B.ID copy from 129,
 %3=QlInstance %3 Domain INTEGER AID From:#42 copy from 171)}
```

In ParseSelectRowSet, line 8081, before parsing the where clause (at line 8088), the initial state of the
SelectRowSet #1 is

```
{SelectRowSet #1 TABLE (#8 CHAR,#16 CHAR) Display=2 targets: 23=#32,120=#42 Source: #40}
```

After the where condition has been parsed and applied to the SelectRowSet at (line 8078), the details of
the where clause (at #59) have been identified and all the above rowSets have been transformed:

```
{(..
 #1=SelectRowSet #1 TABLE (#8 CHAR,#16 CHAR) Display=2 targets: 23=#32,120=#42 Source: #40,
 #8=QlInstance #8  CHAR ANAME From:#32 Alias=A.ANAME copy from 77,
 #16=QlInstance #16  CHAR TITLE From:#42 copy from 212,
 #32=TableRowSet #32 AUTHOR TABLE (%1 INTEGER,#8 CHAR) Display=2 Indexes=[(%1)=[56]] key (%1)
order (%1) targets: 23=#32 From: #40 Target=23 SRow:(34,77) Target:23 AUTHOR Alias: A,
 #40=JoinRowSet #40 (%1 INTEGER,#8 CHAR,%2 INTEGER,%3 INTEGER,#16 CHAR) Display=5 where (#59)
matching (%1=(%3),%3=(%1)) targets: 23=#32,120=#42 INNER JoinCond: (#59) First: #32 Second: #42
on %1=%3,
 #42=TableRowSet  #42  BOOK  TABLE  (%2  INTEGER,%3  INTEGER,#16  CHAR)  Display=3
Indexes=[(%2)=[151],(%3)=[195]] key (%3) order (%3) targets: 120=#42 From: #40 Target=120
SRow:(129,171,212) Target:120 BOOK Alias: B,
 #59=SqlValueExpr #59  BOOLEAN From:#32 Left:#55 Right:#60 #59(#55=#60),
 %0=Domain TABLE (#8,#16) Display=2[#8, CHAR],[#16, CHAR],
```
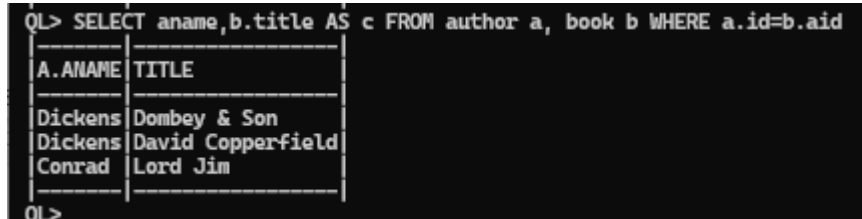
---

⁴⁵ -502 is the built-in alias for the role that started the server, which is the transaction's role if the database has no users defined.

```
%1=QlInstance %1 Domain INTEGER ID From:#32 Alias=A.ID copy from 34,
%2=QlInstance %2 Domain INTEGER ID From:#42 Alias=B.ID copy from 129,
%3=QlInstance %3 Domain INTEGER AID From:#42 copy from 171)}
```

The row orderings on #40 and #42 are required for the *inner* join, but no action is needed because in both cases the order column is the primary key. The final step of ParseCursorSpecification at line 7592 adds the SelectStatement to the context.

```
%5=SelectStatement %5 Union=#1
```

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

The analysis results in an efficient computation of the Join.

6.2 RowSets and Context

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

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

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

The Context keeps track of the structures required for constructing the result rowSet and provides a context for local variables: an execution stack is required in general for evaluation of triggers, constraints and user-defined functions. A new Context must be created for each stack frame, but the new stack frame can start off with all the previous frame's values still visible, because 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 a mutable list of (immutable) 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.

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. When a new grouping key value is found, StartCounter initialises the registers, and then each new row calls AddIn, for the aggregations in the set. After this traversal, the grouped rowset can be built.

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 already ordered by the join columns, then computing the join costs nothing (i.e. O(N)): a join bookmark simply advances the left and right rows returns rows where the join columns have matching values. If the join columns are not in the right order for the join, a OrderedRowSet is interposed during parsing. The cost of ordering is O(NlogN+MlogM) if

both sets need to be ordered. Cross joins cost O(MN) of course. Such ordering steps can be removed later in analysis if it turns out there is a suitable index, or only a single row.

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

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

Some rowsets keep track of available indexes to their rows (importantly, TableRowSet and JoinRowSet) and use available indexes 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 QlValues, 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 QlValue uids and the TableColumn uids involved in these processes.

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

As an example, consider the following (starting with an empty database, but having placed a breakpoint in Executable.cs for SqlInsert._Obey(), line 3846):

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

At the breakpoint, Context.obs contains the following:

```
{(23=Table TABLE (29,50)[29,Domain INTEGER],[50,Domain CHAR] rows 0,
  29=TableColumn 29 Definer=-502 LastChange=29 Domain INTEGER Table=23,
  50=TableColumn 50 Definer=-502 LastChange=50 Domain CHAR Table=23,
  #1=SqlInsert #1 Target: %3 Value: %4 Columns: [%1],
  #13=TableRowSet #13 A TABLE (%0 INTEGER,%1 CHAR) Display=2 targets: 23=#13 From: #13 Target=23
SRow:(29,50) Target:23 A,
  #18=SqlRowArray #18 TableRowSet %3 A TABLE (%1 CHAR) Display=1 targets: 23=#13 From: #13
Target=23 SRow:(29,50) Target:23 A #25,
  #25=SqlRow #25 Domain ROW (#26) Display=1[#26, CHAR]  (#26=Three),
  #26=Three,
  %0=QlInstance %0 Domain INTEGER B From:#13 copy from 29,
  %1=QlInstance %1 Domain CHAR C From:#13 copy from 50,
  %2=Domain TABLE (%1) Display=1[%1,Domain CHAR],
  %3=TableRowSet %3 A TABLE (%1 CHAR) Display=1 targets: 23=#13 From: #13 Target=23 SRow:(29,50)
Target:23 A,
  %4=ExplicitRowSet %4 A TABLE (%1 CHAR) Display=1 targets: 23=#13 From: #13[#25: [%1=Three]],
  %5=Domain ROW (#26) Display=1[#26, CHAR])}
```

Here the entries in blue are the shared objects, read from the database, that were committed by the create statement. Examining the previous entry in the Call Stack, we see that the Execute method began by creating an Activation (a subclass of Context) for executing any triggers or constraints that may be defined on the table, its columns, or their domains. At the breakpoint, therefore, the context cx is actually an Activation, the Executable is SqlInsert, for the target[46] TableRowSet #13. The Values part of an SqlInsert will usually not have simple constant values, and the columns of the data set may be a subset of the table columns, possibly in a different column order (see TableRowSet %3), matching the values clause in SqlRowArray #18, whose values (not necessarily literal constants) will be evaluated to the data ExplicitRowSet %4. Set a breakpoint at line 3860.

SqlInsert.Obey() looks at the targets of the insert (for a modifiable join there will be more than one), and at line 3852 constructs a set `ts` of TargetActivations to control the insert operation. Stepping into line 3852, we see that the Insert() method for TableRowSet constructs a TableActivation for the modification. In general, the construction of each TableActivation requires setting up the machinery for triggers, verifying constraints, and managing cascades, but in this case the only thing it does is to create (at Activation.cs line 279) the TransitionRowSet object for the single target 23. An iteration through the data rowset %4 begins, and at our breakpoint we can see the transition row set as ta._trs:

```
{TransitionRowSet %6 A TABLE (%0 INTEGER,%1 CHAR) Display=2 targets: 23=#13 From: #13 Data: %4
Target: 23}
```

In this case there is a single row and we have the cursor ib is `{[%1=Three] %4}`, and if we now step into ta.EachRow (at line 481 of TableActivation), we have a transition cursor and a target cursor:
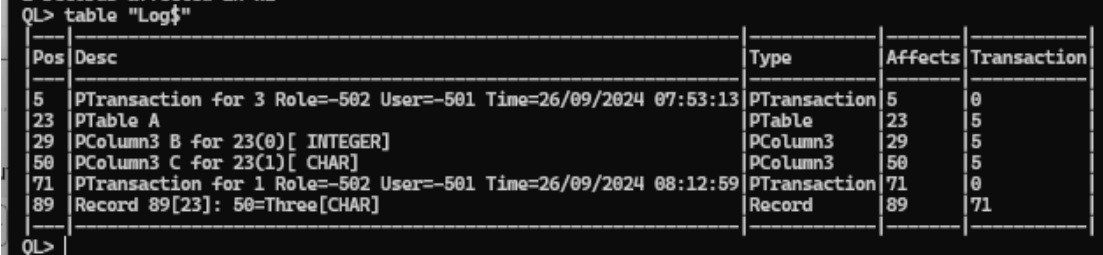
```
trc {[%0= Null,%1=Three] %6}
trc._tgc {[29= Null,50=Three] %6}[47]
```

tgc._rec is a newly constructed TableRow with these values. newRow is set to these values, and as there are no triggers in this example, at line 499 a new Record

```
{Record !0[23]: 29= Null[Null],50=Three[CHAR]}
```

has been made from newRow, which will be committed to the database. Allow execution to continue and examine the table and transaction log, where we see that the record was committed as

```
{Record 89[23]: 50=Three[CHAR]}
```

```
QL> table "Log$"
---------------------------------------------------------------------------------
 Pos Desc                                                          Type        Affects Transaction
---------------------------------------------------------------------------------
 5   PTransaction for 3 Role=-502 User=-501 Time=26/09/2024 07:53:13 PTransaction 5      0
 23  PTable A                                                      PTable      23      5
 29  PColumn3 B for 23(0)[ INTEGER]                                PColumn3    29      5
 50  PColumn3 C for 23(1)[ CHAR]                                   PColumn3    50      5
 71  PTransaction for 1 Role=-502 User=-501 Time=26/09/2024 08:12:59 PTransaction 71     0
 89  Record 89[23]: 50=Three[CHAR]                                 Record      89      71
---------------------------------------------------------------------------------
QL>
```

We will return to this machinery when we discuss Triggers in section 6.5.

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

---

[46] RowSets are generally designed and used for selection, where a TableRowSet will be a source, but for CRUD operations on tables, the TableRowSet is really the target of the operation. Hence at line 3781 we have the confusing phrase "if (cx.obs[source] is RowSet tg". We will soon see that other targets such as views and joins are possible.

[47] There are several contexts in operation here: the TransitionCursor is listed in the SqlInsert's Activation, while the TargetCursor is listed in the TableActivation. These two activations are different Contexts, so it is helpful that the cursors can have the same uid.

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

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

Here is a worked example, based on a test contributed by Fritz Laux. The full data is in the distribution (in Pyrrho\doc\tests.txt, lines 74-114). 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 (note this needs to be supplied on a single command line – if necessary surround it with square brackets []):

```
        1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567890
select city, avg(Salary), count(*) as numPeople from people
        7         8         9        10        11
12345678901234567890123456789012345678901234567890123456789012345
where country = 'GER' group by city having count(*) > 4;
```

At the start of the Build method for the SelectRowSet result (line 2709 in RowSet.cs), the context contains

```
{(23=Table  TABLE  (49,85,125,165,207)[49,Domain  INTEGER  Prec=11  NOT  NULL],[85,Domain  CHAR
Prec=50],[125,Domain NUMERIC Prec=7 Scale=2],[165,Domain CHAR Prec=50],[207,Domain CHAR Prec=50]
rows 30 Indexes:((49)232) KeyCols: (49=True),
  49=TableColumn 49 Definer=-502 LastChange=49 Domain INTEGER Prec=11 NOT NULL Table=23 Not
Null,
  85=TableColumn 85 Definer=-502 LastChange=85 Domain CHAR Prec=50 Table=23,
  125=TableColumn 125 Definer=-502 LastChange=125 Domain NUMERIC Prec=7 Scale=2 Table=23,
  165=TableColumn 165 Definer=-502 LastChange=165 Domain CHAR Prec=50 Table=23,
  207=TableColumn 207 Definer=-502 LastChange=207 Domain CHAR Prec=50 Table=23,
  #1=SelectRowSet #1 PEOPLE TABLE (#8 CHAR,#14 NUMERIC,#27 INTEGER) Aggs (#14,#27) Display=3
key (%1) having (%8) groupSpec: #89 groupings (%5) having (%8) GroupCols(#8) targets: 23=#54
From: #54 Source: #54,
  #8=QlInstance CITY #8[CITY] Domain CHAR Prec=50 From:#54 copy from 207,
  #14=SqlFunction #14  NUMERIC AVG From:#1 AVG(#18),
  #18=QlInstance SALARY #18[SALARY] Domain NUMERIC Prec=10 Scale=2 From:#54 copy from 125,
  #27=SqlFunction #27  INTEGER COUNT From:#1 Alias=NUMPEOPLE COUNT(#33) as NUMPEOPLE,
  #33=1,
  #54=TableRowSet #54 PEOPLE TABLE (%1 INTEGER,%2 CHAR,#18 NUMERIC,%3 CHAR,#8 CHAR) Display=5
Indexes=[(%1)=[232]] key (%1) where (#75) matches (%3=GER) targets: 23=#54 From: #54 Target=23
SRow:(49,85,125,165,207) Target:23 PEOPLE,
```

```
#75=SqlValueExpr #75  BOOLEAN From:#54 Left:#67 Right:#77 #75(#67=#77),
#77=GER,
#89=GroupSpecification #89(%5),
#92=QlInstance #92 Domain CHAR Prec=50 CITY From:#54 copy from 207,
#104=SqlFunction #104  INTEGER COUNT COUNT(#110),
#110=1,
#113=SqlValueExpr #113 Domain BOOLEAN Aggs (#104) From:_ Left:#104 Right:#115 #113(#104>#115),
#115=4,
%0=Domain TABLE (#8,#14,#27) Display=3[#8,Domain CHAR Prec=50],[#14, NUMERIC],[#27, INTEGER]
Aggs (#14,#27),
%1=QlInstance %1 Domain INTEGER Prec=11 ID From:#54 copy from 49,
%2=QlInstance %2 Domain CHAR Prec=50 NAME From:#54 copy from 85,
%3=QlInstance %3 Domain CHAR Prec=50 COUNTRY From:#54 copy from 165,
%5=Grouping ROW (#8)[#8,Domain CHAR Prec=50] (#8=0),
%6=Domain ROW (#8) Display=1[#8,Domain CHAR Prec=50],
%7=Domain ROW (#8)[#8,Domain CHAR Prec=50],
%8=SqlValueExpr %8 Domain BOOLEAN Aggs (#27) From:#1 Left:#27 Right:#115 %8(#27>#115),
%9=Domain ROW (#8)[#8,Domain CHAR Prec=50],
%10=SelectStatement %10 Union=#1)}
```

(Note that the where condition #75 has been pushed down to the TableRowSet source of the SelectRowSet, and in %8, #104 has been replaced by the equivalent #27.)

There are two main loops in the Build method. The first (lines 2739-2757) traverses the source rowSet (#54 in this case) and builds a catalogue of Registers for the aggregated functions and for each value of the grouping keys. For example, the very first cursor from the source rowset is

```
{[%1=61,%2=Tom,#18=50000.00,%3=GER,#8=Berlin] #54}
```

giving a grouping key of {(Berlin)}. The registers are constructed for this rowset in the context, by the AddIn methods called for aggregation functions at line 2756. The structure used by the Context is called funcs and is indexed by the aggregation function uid, the grouping key, and the SelectRowSet uid.

At the end of this first loop (line 2760) we can see what the funcs structure contains for the SelectRowSet #1:

```
{([#8=Berlin]=(#14={6 362000},#27={6 }),
  [#8=Munich]=(#14={6 371000},#27={6 }),
  [#8=Tuebingen]=(#14={3 186000},#27={3 }))}
```

i.e. for each group #8 the registers for the average #14 and count #27 (the average register highlighted holds count and sum).

The second loop (lines 2760-2797) traverses the cx.funcs register collection and for each grouping key that was found, places a row (line 2795) in the SelectRowSet if the having condition is satisfied. This set of rows is added to the SelectRowSet object in the context. In this case the having condition %8 results in the Tübingen group not being displayed.

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])}
```



```
QL> select city, avg(Salary), count(*) as numPeople from people where country = 'GER' group by city having count(*) > 4
```

CITY	AVG	NUMPEOPLE
Berlin	60333.3333333333	6
Munich	61833.3333333333	6

6.2.3 Views

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

6.2.4 RestViews

RestViews are created using a modified syntax for the CREATE VIEW statement. There are three changes: (a) an OF clause that specifies a rowtype for the view, with the same syntax 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 QlValue vs TypedValue

The parser creates QlValues, which in v7 are immutable and only need to be constructed once. QlValues do not have exposed values: the value of an QlValue 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 v7 of Pyrrho is that compiled objects (including triggers, stored procedures etc) are only parsed once per cold start. The uids given to DBObjects generated by the parser during Load are all in the executable range (shown as `0,.. is these notes). These generated objects are not added to the Database objects tree, but stored (in Framing) in the trigger or procedure (Compiled) DBObject. When execution of a procedure body or trigger is required, these objects can simply be added to the activation context (by the line of code cx.obs += ob.framing) to make them available in the context. (References to code modules do not require instancing, unlike Views, whose RowSet instances will be transformed by their context.)

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

As an example, let us look in detail at the processes described in section 3.5.2, for a check definition within an explicit transaction. Starting with an empty database, place a breakpoint in ParseColumnCheckConstraint (line 4367 of Parser.cs) and begin transaction:

```
        1         2         3         4
1234567890123456789012345678901234567890123456789012
begin transaction
create table ca(a char,b int check (b>0))
```

During ParseColumnCheckConstraint, cx.parse is set to Compile (line 4357 of Parser.cs). Then for the construction of the Physical PCheck2 object, (line 4367) cx.obs is:

```
{(!0=Table TABLE (!1,!2)[!1, CHAR],[!2, INTEGER] rows 0,
  !1=TableColumn !1 Definer=-502 LastChange=!1  CHAR Table=!0,
```

```
!2=TableColumn !2 Definer=-502 LastChange=!2  INTEGER Table=!0,
`0=Domain ROW (!2) Display=1[!2, INTEGER],
`3=SqlReview B `3[B]  CONTENT,
`6=SqlValueExpr `6  BOOLEAN From:_ Left:`3 Right:`9 `6(`3>`9),
`9=0)}
```

We see the proposed table and its columns[48], and the parsed version of the check expression with uids `3, `6 and `9 in the executable range. On return from the constructor (at line 4369), because PCheck is a subclass of Compiled, it already has a framing field containing these in addition to a copy of the domain for the new table (pc.framing.obs):

```
{(`0=Domain ROW (!2) Display=1[!2, INTEGER],
`3=SqlReview B `3[B]  CONTENT,
`6=SqlValueExpr `6  BOOLEAN From:_ Left:`3 Right:`9 `6(`3>`9),
`9=0)}
```

When the PCheck is added to the context at line 4329, it constructs the Check object (in PCheck2.Install, line 239 of PCheck.cs) we get a Check object with the above framing, so that once this is added to the context (line 4371) we have in the context:

```
{(!0=Table TABLE (!1,!2)[!1, CHAR],[!2, INTEGER] rows 0,
 !1=TableColumn !1 Definer=-502 LastChange=!1  CHAR Table=!0,
 !2=TableColumn !2 Definer=-502 LastChange=!2  INTEGER Table=!0 Checks:(!3=True),
`0=Domain ROW (!2) Display=1[!2, INTEGER],
`3=SqlReview B `3[B]  CONTENT,
`6=SqlValueExpr `6  BOOLEAN From:_ Left:`3 Right:`9 `6(`3>`9),
`9=0)}
```

The Check constraint is usable in this form during the explicit transaction (after commit, the !0 uids will be replaced by file positions).

After installing the Check object in the transaction (line 248), the context is cleared. (The Transaction's list of physicals will retain the objects !0 to !3 until commit or rollback.)

Curiously, the Check object !3 is used directly from the database cx.db (it is not in cx.obs): you can check that cx.db.objects[0x4000000000000003] is

```
        {Check BOOLEAN From.Target=!2 Source=(b>0) Search=`6}
```

Suppose the next statement is:

```
4       5       6       7
4567890123456789012345678901234 56
insert into ca values('Neg',-99)
```

We will trace through what happens. From example 6.2.1, we know that SqlInsert uses EachRow to construct a new row for a table, and it uses a TargetCursor. So, this time, place a break point at the start of TargetCursor.New (line 5329 of RowSet.cs). At this point we have the following in the TableActivation context (lexical positions currently run on during a transaction as indicated by the ruler above):

```
{(..
  #42=SqlInsert #42 Target: #54 Value: %3 Columns: [%0,%1],
  #54=TableRowSet #54 CA TABLE (%0 CHAR,%1 INTEGER) Display=0 targets: !0=#54 From: #54 Target=!0
SRow:(!1,!2) Target:!0 CA,
  #57=SqlRowArray #57 TableRowSet #54 CA TABLE (%0 CHAR,%1 INTEGER) Display=0 targets: !0=#54
From: #54 Target=!0 SRow:(!1,!2) Target:!0 CA #63,
  #63=SqlRow  #63  Domain  ROW  (#64,#70)  Display=2[#64,  CHAR],[#70,Domain  UNION  ..]
(#64=Neg,#70=QlValueExpr #70 Domain UNION .. From:_ Right:#71 #70(-#71)),
  #64=Neg,
  #70=QlValueExpr #70 Domain UNION .. From:_ Right:#71 #70(-#71),
  #71=99, ..
  %0=QlInstance %0 Domain CHAR A From:#54 copy from !1,
  %1=QlInstance %1 Domain INTEGER B From:#54 copy from !2,
  %2=Domain ROW (%0,%1) Display=2[%0,Domain CHAR],[%1,Domain INTEGER],
  %3=ExplicitRowSet %3 CA TABLE (%0 CHAR,%1 INTEGER) Display=2 targets: !0=#54 From: #54[#63:
[%0=Neg,%1=-99]],
  %5=Domain ROW (#64,#70) Display=2[#64, CHAR],[#70,Domain UNION ..],
  %6=TransitionRowSet %6 CA TABLE (%0 CHAR,%1 INTEGER) Display=2 targets: !0=#54 From: #54 Data:
%3 Target: !0)}
```

At line 5349, the row contents for the insert have computed in vs

---

[48] No physical locations yet. !0, !1 etc are placeholders in the Transaction's proposed physicals list.

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

We find the constraint on the second tablecolumn !2 at line 5377.

`{Check BOOLEAN From.Target=!2 Source=(b>0) Search=`6}`

The next two lines 5379 and 5380 adds the check framing and 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.

```
QL-T>create table ca(a char,b int check (b>0))
QL-T>insert into ca values('Neg',-99)
Column check  fails for column B in table CA
The transaction has been rolled back
QL>
```

6.5 Trigger Implementation

As elsewhere in Pyrrho, parsing of executable SQL takes place on definition[49] of a database object. Definition is within the command processing that creates the object, or the database load on server start-up, and the associated executable uids of compiled objects will differ for these two cases. The compiled objects are placed in the framing property of the associated DBObject, and these are added to the context when the DBObject is instanced.

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

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

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

If these have all been entered in auto-commit mode, the log contains[50]:

Pos	Record Type	Contents
23	PTable XA	
30	PColumn3 B	for 23(0)[INTEGER]
51	PColumn3 C	for 23(1)[INTEGER]
73	PColumn3 D	for 23(2)[CHAR]
112	PTable XB	
119	PColumn3 TOT	for 112(0)[INTEGER]
160	Record 160[112]	119=0[INTEGER]
190	Trigger RUAB	Trigger RUAB Update, Before, EachRow on 23, MR=old row , NR=new row : begin atomic update xb set tot=tot-mr.b+nr.b; set d='changed' end
300	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,

---

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

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

following a server restart[51]) is from the debugger, and contains the items that will be cached in the context when the trigger is used. They have uids in the executable range notated here `0,..:

```
{Framing (obs {
   `55=TableRowSet `55 XA TABLE (`56 INTEGER,`57 INTEGER,`58 CHAR) Display=3 targets: 23=`55
From: `55 Target=23 SRow:(30,51,73) Target:23 XA,
   `56=QlInstance `56  INTEGER B From:`55 copy from 30,
   `57=QlInstance `57  INTEGER C From:`55 copy from 51,
   `58=QlInstance `58  CHAR D From:`55 copy from 73,
   `59=CompoundStatement  `59(`60,`73),
   `60=AssignmentStatement `60 `57=`68,
   `68=SqlValueExpr `68  INTEGER From:`55 Left:`56 Right:`71 `68(`56+`71),
   `71=3,
   `73=UpdateSearch `73 Target: `74,
   `74=TableRowSet  `74  XB  TABLE  (`75  INTEGER) Display=1  targets:  112=`74  From:  `74
Assigs:(UpdateAssignment Vbl: `75 Val: `83=True) Target=112 SRow:(119) Target:112 XB,
   `75=QlInstance `75  INTEGER TOT From:`74 copy from 119,
   `83=SqlValueExpr `83  INTEGER From:`74 Left:`75 Right:`56 `83(`75+`56),
   `89=WhenPart `89_ Stms: (`59))}
```

The CompoundStatement `59 (highlighted) implements the fragment of trigger source "`set c=b+3; update xb set tot=tot+b`".

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

The next step in the test is an insert command:

```
          1         2         3         4
1234567890123456789012345678901234567890 1
insert into xa(b,d) values (7,'inserted')
```

Let us place a break point at the start of Transaction.Execute (line 344 of Transaction.cs), and trace through everything that happens here. Recall that several sorts of insert triggers can be defined on a table, and possibly more than one of any kind, possibly defined by different roles. So, the trigger implementation must take into account the command context (with the session role), the target conext (using the table definer's role) and maybe several different trigger contexts (the trigger definers' roles). After parsing the insert command (at our break in line 344) we have in the main Context cx:

```
{(23=Table  TABLE  (30,51,73)[30,  INTEGER],[51,  INTEGER],[73,  CHAR]  rows  0  Triggers:(Insert,
Before, EachRow=(300=True),Update, Before, EachRow=(190=True)),
  30=TableColumn 30 Definer=-502 LastChange=30  INTEGER Table=23,
  51=TableColumn 51 Definer=-502 LastChange=51  INTEGER Table=23,
  73=TableColumn 73 Definer=-502 LastChange=73  CHAR Table=23,
  #1=SqlInsert #1 Target: %4 Value: %5 Columns: [%0,%2],
  #13=TableRowSet #13 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 targets: 23=#13 From:
#13 Target=23 SRow:(30,51,73) Target:23 XA,
  #21=SqlRowArray #21 TableRowSet %4 XA TABLE (%0 INTEGER,%2 CHAR) Display=2 targets: 23=#13
From: #13 Target=23 SRow:(30,51,73) Target:23 XA #28,
  #28=SqlRow #28 Domain ROW (#29,#31) Display=2[#29, INTEGER],[#31, CHAR]  (#29=7,#31=inserted),
  #29=7,
  #31=inserted,
  %0=QlInstance %0  INTEGER B From:#13 copy from 30,
  %1=QlInstance %1  INTEGER C From:#13 copy from 51,
  %2=QlInstance %2  CHAR D From:#13 copy from 73,
  %3=Domain TABLE (%0,%2) Display=2[%0, INTEGER],[%2, CHAR],
  %4=TableRowSet %4 XA TABLE (%0 INTEGER,%2 CHAR) Display=2 targets: 23=#13 From: #13 Target=23
SRow:(30,51,73) Target:23 XA,
  %5=ExplicitRowSet %5 XA TABLE (%0 INTEGER,%2 CHAR) Display=2 targets: 23=#13 From: #13[#28:
[%0=7,%2=inserted]],
  %6=Domain ROW (#29,#31) Display=2[#29, INTEGER],[#31, CHAR])}
```

The command to be executed is the SqlInsert highlighted above. Let us trace what happens next by single-stepping into SqlInsert #1._Obey() (and stepping over to line 3860 of Executable.cs). It works in an Activation (a sort of Context), whose identifier is 63 here (your mileage may vary). The target and data rowSets have just been identified as

```
tg <- {TableRowSet %4 XA TABLE (%0 INTEGER,%2 CHAR) Display=2 targets: 23=#13 From: #13 Target=23
SRow:(30,51,73) Target:23 XA}
```

---

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

```
data <- {ExplicitRowSet %5 XA TABLE (%0 INTEGER,%2 CHAR) Display=2 targets: 23=#13 From: #13[#28:
[%0=7,%2=inserted]]}
```

respectively. SqlInsert._Obey has constructed a list of TargetActivations `ts`,

```
{(23=TableActivation 64)}
```

and in this example there is just the one TableActivation (with identifier 64). Then ta is TableActivation 64 and in addition to the above objects ta.obs has (a shortcut is in ta._trs)

```
{TransitionRowSet %7 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 targets: 23=#13 From:
#13 Data: %5 Target: 23}
```

The TableActivation also has a list of TriggerActivations called acts,

```
{(300=TriggerActivation 65)}
```

and acts[0] is TriggerActivation 65. The TriggerActivation constructor added the table's triggers to its objects, so that these now include all three sets of objects shown above (the trigger framing, the obs of the main Context, and the TransitionRowSet %7).

The traversal of the *data* TableRowSet %4 has just begun (at line 3853), and the first cursor is the ExplicitCursor:

```
{[%0=7,%2=inserted] %5}
```

Single-step into the EachRow method of TableActivation 64. We create a transition cursor trc, with target cursor tgc and proposed record rc (see lines 477-480 of TableActivation.EachRow() in Activation.cs):

```
trc {[%0=7,%1= Null,%2=inserted] %7}
trc._tgc {[30=7,51= Null,73=inserted] %7}
tgc._rec.vals {(30=7,51= Null,73=inserted)}
```

Recall (from section 6.2.1) that the transitionCursor is in the SqlInsert's activation 63 while the targetCursor is for the TableActivation 64.

At line 488, EachRow() calls Triggers() to obey the trigger RIAB. Step into Triggers(): we start traversal of the list of trigger activations: there is just one, {TriggerActivation 65} so step into the call of Exec() at line 644. The first important thing it does is to create a TriggerCursor for the row for %7, at line 1038, and stores its values in the TriggerActivation:

```
65.values {(23=[30=7,51= Null,73=inserted] %7,30=7,51= Null,73=inserted)}
```

At line 1069, TriggerActivation.Exec() begins to execute the CompoundStatement `59 in TriggerActivation 65, and this (at line 351 of Executable.,cs) creates a local Activation 66. `59._Obey() calls ObeyList(). For convenience. place a break point on the _Obey() call in Executable.ObeyList (line 66).

The first statement executed is the {AssignmentStatement `60 `57=`68}. Recall from above,

```
  `56=QlInstance `56  INTEGER B From:`55 copy from 30,
  `57=QlInstance `57  INTEGER C From:`55 copy from 51,
..
  `60=AssignmentStatement `60 `57=`68,
  `68=SqlValueExpr `68  INTEGER From:`55 Left:`56 Right:`71 `68(`56+`71),
  `71=3,
```

We see that [`68] is [`56]+[`71]. Now [`56] is an QlInstance to copy the current value for table 23 XA,while obs[`71] is the literal 3, so we get 7 and add 3 to get 10 (line 1163). The effect of the AssignmentStatement is thus (at line 1178 of Executable.cs) to store <mark>{10} as the context's value for QlInstance `68, in activation</mark> 66.

At this point, therefore, Activation 66 has the following **values** (including those inherited from 65):

```
66.values {(23=[30=7,51= Null,73=inserted] %7,30=7,51= Null, 73=inserted, `56=7, `57= Null,
`58=inserted)}
```

The next step in the CompoundStatement (at our breakpoint line 69 of Executable.cs) is {UpdateSearch `73 Target: `74}:

```
  `73=UpdateSearch `73 Target: `74,
  `74=TableRowSet  `74  XB  TABLE  (`75  INTEGER)  Display=1  targets:  112=`74  From:  `74
Assigs:(UpdateAssignment Vbl: `75 Val: `83=True) Target=112 SRow:(119) Target:112 XB,
  `75=QlInstance `75  INTEGER TOT From:`74 copy from 119,
  `83=SqlValueExpr `83  INTEGER From:`74 Left:`75 Right:`56 `83(`75+`56),
```

`73._Obey() calls `74.Update(), which (as occurred with SqlInsert above) at line 4086 creates a new TableActivation 67. Again, when using the debugger, it is easiest to step over to line 4095 to look at ta. This has just one further object (ta._trs)

```
{TransitionRowSet %8 XB TABLE (`75 INTEGER) Display=1 targets: 112=`74 From: `74 Data: `74
Assigs:(UpdateAssignment Vbl: `75 Val: `83=True) Target: 112}
```

but no triggers, as XB has none defined. The traversal cursor ib is {[`75=0] `74} and ta.EachRow constructs

```
trc {[`75=0] %8}
tgc {[119=0] %8}
rc.vals {(119=0)}
```

The updates in EachRow (at line 520-527) now construct the new values list with newRow {(119=7)} set at line 523.

At line 562, the transition and target cursors for %8 are (both in the same call of +=) updated for the new values, and the new targetCursor {(119=7) %8} as newRow, and the new value 119=7 are installed in the previous activation (next) 66. At line 567:

```
66.cursors {(112=[`75=0] `74,#13=[%0=7,%2=inserted] %5,`74=[`75=0] `74,%5=[%0=7,%2=inserted]
%5,%7=[`56=7,`57= Null,`58=inserted] %7,%8=[119=7] %8)}
```

```
66.values {(23=[30=7,51= Null,73=inserted] %7,30=7,51= Null,73=inserted,112=[`75=7]
%8,119=7,`56=7,`57=10,`58=inserted,`75=7)}
```

The new Update gets constructed (at line 574) as {Update 160[112]: 119=7[INTEGER] Prev:160}. In case there are after-triggers, the newTables field for 51 is updated with the new TableRow for XB. But there are none, and we return to `73._Obey(), where, at line 4096, *we overwrite 66's version of the transaction with 67's* (it has the new Update !1 in db.physicals). This version is also used to update the list of targetActivations at line 4097 (it is retained for further rows, and there aren't any). The modified activation 66 is returned to.ObeyList() for CompoundStatement `59. As this has no more statements, this also returns: activation 66 was created just for the CompoundStatement execution, so at line 355 we call 66.SlideDown(), which updates 65's values:

```
65.values {(23=[30=7,51= Null,73=inserted] %4,30=7,51= Null,73=inserted, 112=[119=7] %5, 119=7,
`53=7, `54=10,`55=inserted,`72=0)}
```

We are then back in 65.Exec(). At line 1082, we transmit the triggered changes in the row values to update the transitionCursor for %7. In particular, the result of the assignment `57 above gets placed in the newRow for table XA at line 1083:

```
64.newRow {(30=7,51=10,73=inserted)}
```

Each SlideDown in the context stack overwrites the transaction, so that for example in 64.SlideDown, the new Physicals {(!0=TriggeredAction 305,!1=Update 165[116]: 123=7 Prev:165)} accumulate in the parent context.

We have completed execution of each-row-before triggers, and now the insert of the new row can take place, in 64.EachRow(). At line 498 this gives a new Record for the transaction: {Record !2[23]: 30=7[INTEGER],51=10[INTEGER],73=inserted[CHAR]}, which is added to 64.db.physicals.

This means that at line 507 the transaction now has the following Physical objects to be committed:

```
{(!0=TriggeredAction 300⁵²,
  !1=Update 160[112]: 119=7[INTEGER] Prev:160,
  !2=Record !2[23]: 30=7[INTEGER],51=10[INTEGER],73=inserted[CHAR])}
```

We immediately get activation 63 to adopt these, and they will eventually be adopted by the root context #1. The command is finished, and the following records are committed along with the triggeredAction note at position 420 in the transaction log:

390	TriggeredAction	300
397	Update 160[112]	119=7 Prev:160
416	Record 416[23]:	30=7,51=10,73=inserted

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

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

---

[52] The TriggeredAction record is added to the log with the role and user information (auditing).

This adds some more entries in the log, notably:

465	TriggeredAction	300
472	Update 160[112]	119=16 Prev:397
491	Record 491[23]:	30=9,51=12,73=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 RUAB 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. The framing contains the following code (in the framing for Trigger 190):

```
{(`0=SqlOldRow XA `0 Definer=-502 LastChange=73  Null From:`6 (),
 `1=SqlNewRow XA `1 Definer=-502 LastChange=73  Null From:`6 (),
 `6=TableRowSet `6 XA TABLE (`7 INTEGER,`8 INTEGER,`9 CHAR) Display=3 targets: 23=`6 From: `6
Target=23 SRow:(30,51,73) Target:23 XA,
 `7=QlInstance `7  INTEGER B From:`6 copy from 30,
 `8=QlInstance `8  INTEGER C From:`6 copy from 51,
 `9=QlInstance `9  CHAR D From:`6 copy from 73,
 `10=SqlField `10  INTEGER B From:`0 Target=30,
 `11=SqlField `11  INTEGER C From:`0 Target=51,
 `12=SqlField `12  CHAR D From:`0 Target=73,
 `13=SqlField `13  INTEGER B From:`1 Target=30,
 `14=SqlField `14  INTEGER C From:`1 Target=51,
 `15=SqlField `15  CHAR D From:`1 Target=73,
 `16=CompoundStatement  `16(`17,`41),
 `17=UpdateSearch `17 Target: `18,
 `18=TableRowSet `18 XB TABLE (`19 INTEGER) Display=1 targets: 112=`18 From: `18
Assigs:(UpdateAssignment Vbl: `19 Val: `34=True) Target=112 SRow:(119) Target:112 XB,
 `19=QlInstance `19  INTEGER TOT From:`18 copy from 119,
 `27=SqlValueExpr `27  INTEGER From:`18 Left:`19 Right:`10 `27(`19-`10),
 `34=SqlValueExpr `34  INTEGER From:`18 Left:`27 Right:`13 `34(`27+`13),
 `41=AssignmentStatement `41 `9=`46,
 `46=changed,
 `48=WhenPart `48_ Stms: (`16))}
```

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

As far as Activation 64.EachRow(), the execution proceeds similarly to the above example. At this point we construct

```
trc {[%0=7,%1=10,%2=inserted] %3}
tgc {[30=7,51=10,73=inserted] %3}
rc.vals {(30=7,51=10,73=inserted)}
```

and Step A of the update processing (line 529-531 of Activation.cs) sets up the oldrow (-295) and newrow (-293) values in the TableActivation 64. At line 533 we have

```
64.cursors {(23=[%0=7,%1=10,%2=inserted] #8,
            #8=[%0=7,%1=10,%2=inserted] #8,
            %3=[30=7,51=10,73=inserted] %3)}

64.values   {(-295=[30=7,51=10,73=inserted] %3,
             -293=[30=8,51=10,73=updated],
             23=[30=7,51=10,73=inserted] %3,
             30=8,51=10,73=updated,
             %0=7,%1=10,%2=inserted)}
```

At line 535, we call the triggers. When TriggerActivation 65.Exec() is called, this is how things stand:

```
65.obs

{(..
 190=Trigger 190 Definer=-502 LastChange=190 TrigType=Update, Before, EachRow On=23,
 #1=UpdateSearch #1 Target: #8,
```

```
  #8=TableRowSet #8 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 matches (%0=7) targets:
23=#8 From: #8 Assigs:(UpdateAssignment Vbl: %0 Val: #17=True,UpdateAssignment Vbl: %2 Val:
#21=True) Target=23 SRow:(30,51,73) Target:23 XA,
  #17=8,
  #21=updated,
  #38=SqlValueExpr #38  BOOLEAN From:#8 Left:%0 Right:#39 #38(%0=#39),
  #39=7,
..
  %0=QlInstance %0  INTEGER B From:#8 copy from 30,
  %1=QlInstance %1  INTEGER C From:#8 copy from 51,
  %2=QlInstance %2  CHAR D From:#8 copy from 73,
  %3=TransitionRowSet %3 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 matches (%0=7)
targets: 23=#8 From: #8 Data: #8 Assigs:(UpdateAssignment Vbl: %0 Val:
#17=True,UpdateAssignment Vbl: %2 Val: #21=True) Target: 23)}
```

65.Exec() prepares for execution on the current row by copying the values of old row `0 and new row `1 from TableActivation 64 above. At line 1069, its values are

```
65.values {(-295=[30=7,51=10,73=inserted] %3,
            -293=[30=8,51=10,73=updated],
            23=[30=7,51=10,73=inserted] %3,
            30=8,51=10,73=updated,
            `0=[30=7,51=10,73=inserted] %3,
            `1=[30=8,51=10,73=updated],
            `7=8,`8=10,`9=updated,
            %0=7,%1=10,%2=inserted)}
```

At line 1069 of Activation.cs, the TriggerActivation starts to Obey the CompoundStatement `16.

The first step is the UpdateSearch `17, and it calls the Update method on TableRowSet `18 targeting XB. It creates TableActivation 67 with its TransitionRowSet

```
{TransitionRowSet %4 XB TABLE (`19 INTEGER) Display=1 targets: 112=`18 From: `18 Data: `18
Assigs:(UpdateAssignment Vbl: `19 Val: `34=True) Target: 112}
```

The traversal begins, and 67.EachRow creates cursors:

```
trc {[`19=16] %4}
tgc {[119=16] %4}
rc.vals {(119=16)}
```

Evaluating the update `18's UpdateAssignment highlighted above (at line 519) involves using RUAB's old and new rows values `0 and `1 above.

```
  `19=QlInstance `19  INTEGER TOT From:`18 copy from 119,
  `27=SqlValueExpr `27  INTEGER From:`18 Left:`19 Right:`10 `27(`19-`10),
  `34=SqlValueExpr `34  INTEGER From:`18 Left:`27 Right:`13 `34(`27+`13),
```

The evaluation proceeds as follows:

```
`19   :=                `34
      = (    `27      + `13)
      = ((`19  -`10)      +  `13)
      = (([119]-[`0].30)+([`1].30)
      = ((16    - 7)      +  8)
[119] := 17
```

So in 65.values, we get [119]:=17, and this completes the calculation of the values of XB's new row for the update statement. The new TableRow is now constructed, and EachRow generates the Update record for XB [115] in the transaction physicals: {Update 160[112]: 119=17[INTEGER] Prev:472}. The transitionCursor for 67 is also updated (in case there are further cascading changes). and also installed in TableActivation 67. This completes the UpdateSearch `17.

Our next breakpoint is 66.ObeyList, where now x is an Assignment. At this point, we have the following cursors in Activation 66, where the transition cursor is highlighted:

```
66.cursors {(23=[%0=7,%1=10,%2=inserted] #8,
             112=[`19=16] `18,
             #8=[%0=7,%1=10,%2=inserted] #8,
             `18=[`19=16] `18,
             %3=[`7=7,`8=10,`9=inserted] %3,
             %4=[119=17] %4)}
```

The Assignment updates 66's values for `9 to changed.

```
66.values {(-295=[30=7,51=10,73=inserted] %3,
```

```
                    -293=[30=8,51=10,73=updated],
                    23=[30=7,51=10,73=inserted] %3,
                    30=8,51=10,73=updated,112=[119=17] %4,
                    119=17,`0=[30=7,51=10,73=inserted] %3,
                    `1=[30=8,51=10,73=updated],
                    `7=8,`8=10,`9=changed,`19=16,%0=7,%1=10,%2=inserted)}
```

Back in 64.EachRow(), it remains to merge the updates into the new TableRow in step D. At line 573 we have a further item for the Transaction Commit: {Update 416[23]: 30=8[INTEGER],73=changed[CHAR] Prev:416}.

In the transaction log, this shows as

543	Update 160[112]	119=17 Prev:472
562	Update 416[23]:	30=8,73=changed Prev: 416

This completes the discussion of the Update trigger demonstration.

In the test a third table XC and a third trigger for XA are defined, 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]
```

611	PTable XC	
619	PColumn3 TOTB	for 611(0)[INTEGER]
645	PColumn3 TOTC	for 611(1)[INTEGER]
690	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 690 is as follows:
```
{Framing obs(
  `92=TransitionTable `92 OT (`97 ,`98 ,`99 ) old from 23,
  `96=TableRowSet `96 XA TABLE (`97 INTEGER,`98 INTEGER,`99 CHAR) Display=3 targets: 23=`96
From: `96 Target=23 SRow:(30,51,73) Target:23 XA,
  `97=QlInstance `97  INTEGER B From:`96 copy from 30,
  `98=QlInstance `98  INTEGER C From:`96 copy from 51,
  `99=QlInstance `99  CHAR D From:`96 copy from 73,
  `100=CompoundStatement  `100(`101),
  `101=SqlInsert `101 Target: `104 Value: `110 Columns: [`105,`106],
  `104=TableRowSet `104 XC TABLE (`105 INTEGER,`106 INTEGER) Display=2 targets: 611=`104 From:
`104 Target=611 SRow:(619,645) Target:611 XC,
  `105=QlInstance `105  INTEGER TOTB From:`104 copy from 619,
  `106=QlInstance `106  INTEGER TOTC From:`104 copy from 645,
  `107=Domain ROW (`105,`106) Display=2[`105, INTEGER],[`106, INTEGER],
  `110=SelectRowSet `110 OT TABLE (`97 INTEGER,`98 INTEGER,`99 CHAR) targets: 611=`104 Source:
`92,
  `113=QlInstance `97  INTEGER B From:`96 copy from 30,
  `118=QlInstance `98  INTEGER C From:`96 copy from 51,
  `121=Domain TABLE (`97,`98,`99)[`97, INTEGER],[`98, INTEGER],[`99, CHAR],
  `125=SelectStatement `125 Union=`110,
  `126=SqlValueSelect `126 SelectRowSet `110 OT TABLE (`97 INTEGER,`98 INTEGER,`99 CHAR)
Source: `92 (`110),
  `127=WhenPart `127_ Stms: (`100))}
```

The test run is:

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

As far as the start of #1.Obey(), the execution proceeds as before, and the context contains

```
{(23=Table TABLE (30,51,73)[30, INTEGER],[51, INTEGER],[73, CHAR] rows 2 Triggers:(Insert,
Before,      EachRow=(300=True),Update,     Before,     EachRow=(190=True),Delete,     Instead,
EachStatement=(690=True)),
  30=TableColumn 30 Definer=-502 LastChange=30  INTEGER Table=23,
  51=TableColumn 51 Definer=-502 LastChange=51  INTEGER Table=23,
  73=TableColumn 73 Definer=-502 LastChange=73  CHAR Table=23,
 #1=QuerySearch #1 Target: #13,
```

```
  #13=TableRowSet #13 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 matches (%2=changed)
targets: 23=#13 From: #13 Target=23 SRow:(30,51,73) Target:23 XA,
  #23=SqlValueExpr #23  BOOLEAN From:#13 Left:%2 Right:#24 #23(%2=#24),
  #24=changed,
  %0=QlInstance %0  INTEGER B From:#13 copy from 30,
  %1=QlInstance %1  INTEGER C From:#13 copy from 51,
  %2=QlInstance %2  CHAR D From:#13 copy from 73)}
```

This time, we have a statement level trigger, and this is called when #1.Obey() sets up its list ts of TableActivations for the delete operation. When TableActivation 65 is *constructed,* it calls delete Triggers() at line 464 and 466 to execute **before** and **instead** triggers if any. Our trigger 690 is an *instead of* trigger, so is called on the second invocation of Triggers() at line 466. 65's TransitionRowSet is

```
{TransitionRowSet %4 XA TABLE (%0 INTEGER,%1 INTEGER,%2 CHAR) Display=3 matches (%2=changed)
targets: 23=#13 From: #13 Data: #13 Target: 23}
```

and it has set up its TriggerActivation 66 which has all the above objects (the TransitionRowSet, the objects from Activation 64, and the objects from the trigger framing). 66.Exec() sets up transition table rowSets for old and new if requested (lines 1043 and 1048). In this case, TransitionTableRowSet `92 is created at line 1050 for the old table that has been referenced[53], and it replaces(!) the TransitionTable `92 in this activation (also at line 1050):

```
{TransitionTableRowSet `92 XA TABLE (`97 INTEGER,`98 INTEGER,`99 CHAR) Display=3 targets:
23=#13 From: `96 OLD}
```

At line 1069, 66.Exec() begins to obey the trigger body in Activation 67, CompoundStatement `100. This is just SqlInsert `101, and `101.Obey() retrieves the target and data for the operation:

```
tg {TableRowSet `104 XC TABLE (`105 INTEGER,`106 INTEGER) Display=2 targets: 611=`104 From: `104
Target=611 SRow:(619,645) Target:611 XC}
```

```
data {SelectRowSet `110 OT TABLE (`97 INTEGER,`98 INTEGER,`99 CHAR) targets: 611=`104 Source:
`92}
```

It sets up its own target list ts {(611=TableActivation 68)}. As before, we step over a few times to the ta.EachRow statement, noting that the traversal cursor ib is the SelectedRowSetCursor {[`97=8,`98=10,`99=changed] `110}, and that TableActivation 68 has the additional object

```
{TransitionRowSet %5 XC TABLE (`105 INTEGER,`106 INTEGER) Display=2 targets: 611=`104 From: `104
Data: `110 Target: 611}
```

68.EachRow creates the transition cursors:

```
trc {[`105=8,`106=10] %5}
tgc {[619=8,645=10] %5}
rc.vals {(619=8,645=10)}
```

As before, recall that trc is installed in Activation 67, and tgc in TableActivation 68. Table XC has no triggers, so the insert operation simply creates a new Record {Record !1[611]: 619=8[INTEGER],645=10[INTEGER]}.

Because this is an INSTEAD OF trigger, 48.Exec() returns true, and the actual Delete statement is not executed. Following Commit, the log shows

792	Record 792[611]	619=8,645=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.

---

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

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

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

In the transaction log these are:

23	PTable P	
29	PColumn3 Q	for 23(0)[INTEGER]
50	PIndex P	on 23(29) PrimaryKey
66	PColumn3 R	for 23(1)[CHAR]
87	PColumn3 A	for 23(2)[INTEGER]
127	PView V 127	select q,r as s,a from p

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

```
{View V TABLE (`5,`10,`16) Display=3[`5, INTEGER],[`10, CHAR],[`16, INTEGER] ViewDef view v as
select q,r as s,a from p Result `2}
```

has a Framing[54] that contains the result of compiling the view's select statement:

```
{(`2=SelectRowSet `2 P TABLE (`5 INTEGER,`10 CHAR,`16 INTEGER) Display=3 key (`5) targets:
23=`21 From: `21 Source: `21,
  `5=QlInstance Q `5[Q]  INTEGER From:`21 copy from 29,
  `10=QlInstance R `10[R]  CHAR From:`21 Alias=S copy from 66,
  `16=QlInstance A `16[A]  INTEGER From:`21 copy from 87,
  `19=Domain TABLE (`5,`10,`16) Display=3[`5, INTEGER],[`10, CHAR],[`16, INTEGER],
  `21=TableRowSet `21 P TABLE (`5 INTEGER,`10 CHAR,`16 INTEGER) Display=3 Indexes=[(`5)=[50]]
key (`5) targets: 23=`21 From: `21 Target=23 SRow:(29,66,87) Target:23 P,
  `23=SelectStatement `23 Union=`2)}
```

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 inserts entries *into the view*

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

When the transaction is about to execute this statement after parsing (at line 3782 of Executable.cs), the View has been "instanced" giving the following entries in the Context:

```
{(#1=SqlInsert #1 Target: %24 Value: %25 Columns: [%9],
  #18=SqlRowArray #18 SelectRowSet %24 V TABLE (%9 CHAR) Display=1 key (%4)) targets: 23=%20
From: %20 Source: %20 #24,#35,
  #24=SqlRow #24 Domain ROW (#25) Display=1[#25, CHAR]  (#25=Twenty),
  #25=Twenty,
  #35=SqlRow #35 Domain ROW (#36) Display=1[#36, CHAR]  (#36=Thirty),
  #36=Thirty,
  %0=View V TABLE (%4,%9,%15) Display=3[%4,Domain INTEGER],[%9,Domain CHAR],[%15,Domain INTEGER]
ViewDef view v as select q,r as s,a from p Result %1,
  %1=SelectRowSet %1 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 key (%4)) targets:
23=%20 From: %20 Source: %20,
  %4=QlInstance Q %4[Q]  INTEGER From:%20 copy from 29,
  %9=QlInstance R %9[R]  CHAR From:%20 Alias=S copy from 66,
  %15=QlInstance A %15[A]  INTEGER From:%20 copy from 87,
  %18=Domain  TABLE  (%4,%9,%15)  Display=3[%4,Domain  INTEGER],[%9,Domain  CHAR],[%15,Domain
INTEGER],
  %20=TableRowSet %20 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 Indexes=[(%4)=[50]]
key (%4)) targets: 23=%20 From: %20 Target=23 SRow:(29,66,87) Target:23 P,
  %22=SelectStatement %22 Union=%1,
  %23=Domain TABLE (%9) Display=1[%9, CHAR],
  %24=SelectRowSet %24 V TABLE (%9 CHAR) Display=1 key (%4)) targets: 23=%20 From: %20 Source:
%20,
  %25=ExplicitRowSet %25 V TABLE (%9 CHAR) Display=1 key (%4)) targets: 23=%20 From: %20 Source:
%20[#24: [%9=Twenty],#35: [%9=Thirty]],
  %26=Domain ROW (#25) Display=1[#25, CHAR],
```

---

[54] This can be seen with the debugger at this point by pausing execution and inspecting Database.databases["t12"].objects[122], or more easily at a breakpoint by inspecting cx.db.objects[122].

```
%27=Domain ROW (#36) Display=1[#36, CHAR])}
```

Here objects %0-%22 are instances of objects from the view definition and its framing, to use new uids (%4,%9,%15), since query processing will in general modify them: the shared objects `5-`16 are not referenced in any context. The result of the view instance query is the SelectRowSet %1.

Execute() (at line 342 in Transaction.cs) sets up a new Activation. This calls Obey() for the SqlInsert statement #1. #1.Obey() finds its target %24 and adds the data rowset %25 highlighted above.

It examines the targets of the SelectRowSet, and finds just TableRowSet %20, and calls %23.Insert() to create an Activation to do the Insert. After execution of line 3787 of Executable.cs we see the return value

```
ts {(23=TableActivation 63)}
```

The constructor for TableActivation ensures that any dependent compiled objects are instanced (none in this case) and builds a TransitionRowSet for the insert operation on table 23 (the easiest place to see this is at line 3860 where we can inspect the TableActivation ta with the debugger). ta._trs is a transition rowset

```
{TransitionRowSet %28 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 Indexes=[(%4)=[50]]
key (%4)) targets: 23=%20 From: %20 Data: %25 Target: 23}
```

We see that none of the entries in this RowSet mentions V as a target, so that the insert for the View V has become an Insert for the Table P 23. The process is therefore the same as the above discussion of Insert (e.g. section 6.4). Traversal of the data rowset has begun at line 3853 of Executable.cs. The first data cursor is `{[%9=Twenty] %25}`, and the TableActivation.EachRow method deals with it. It begins (at lines 475-480 in Activation.cs) by creating a TransitionCursor `{[%4= Null,%9=Twenty,%15= Null] %28}` whose TargetCursor for the Table P fills in the key column Q using Pyrrho's autokey feature, `{[29=1,66=Twenty,87= Null] %28}`. Similarly for the second row of the insert, so, following the autoCommit, we have simply

187	Record 187[23]	29=1,66=Twenty
214	Record 214[23]	29=2,66=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 UpdateSearch.Obey() is called (line 4013 of Execute.cs) we have

```
{(#1=UpdateSearch #1 Target: %1,
  #16=Forty two,
  #35=SqlValueExpr #35  BOOLEAN From:%20 Left:%4 Right:#36 #35(%4=#36),
  #36=1,
  %0=View V TABLE (%4,%9,%15) Display=3[%4,Domain INTEGER],[%9,Domain CHAR],[%15,Domain INTEGER]
ViewDef view v as select q,r as s,a from p Result %1,
  %1=SelectRowSet %1 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 key (%4)) matches (%4=1)
targets: 23=%20 From: %20 Source: %20 Assigs:(UpdateAssignment Vbl: %9 Val: #16=True),
  %4=QlInstance Q %4[Q]  INTEGER From:%20 copy from 29,
  %9=QlInstance R %9[R]  CHAR From:%20 Alias=S copy from 66,
  %15=QlInstance A %15[A]  INTEGER From:%20 copy from 87,
  %18=Domain  TABLE  (%4,%9,%15)  Display=3[%4,Domain  INTEGER],[%9,Domain  CHAR],[%15,Domain
INTEGER],
  %20=TableRowSet %20 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 Indexes=[(%4)=[50]]
key (%4)) matches (%4=1) targets: 23=%20 From: %20 Assigs:(UpdateAssignment Vbl: %9 Val:
#16=True) Target=23 SRow:(29,66,87) Target:23 P,
  %22=SelectStatement %22 Union=%1)}
```

We see that changes have been made to the instanced objects (highlighted above): to add the update assignments to the rowset, the column identifiers have been merged with the identifiers in the query (lexical positions #16), and and where(#35) has been converted to a matches condition and added to the SelectRowSet %1 and passed down to the instance TableRowSet %20. Again, this reduces the View update to an ordinary table update (the assignments are obeyed around line 514 of Activation.cs). After Commit we just have

259	Update187[23]	66=Forty two Prev:187

Next we have a simple Delete:

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

Once again, (at line 3986 of Executable.cs) we see the RowSets are just for Table P 23:

```
{(#1=QuerySearch #1 Target: %1,
  #22=SqlValueExpr #22  BOOLEAN From:%20 Left:%9 Right:#23 #22(%9=#23),
  #23=Thirty,
  %0=View V TABLE (%4,%9,%15) Display=3[%4,Domain INTEGER],[%9,Domain CHAR],[%15,Domain
INTEGER] ViewDef view v as select q,r as s,a from p Result %1,
  %1=SelectRowSet %1 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 key (%4)) matches
(%9=Thirty) targets: 23=%20 From: %20 Source: %20,
  %4=QlInstance Q %4[Q]  INTEGER From:%20 copy from 29,
  %9=QlInstance R %9[R]  CHAR From:%20 Alias=S copy from 66,
  %15=QlInstance A %15[A]  INTEGER From:%20 copy from 87,
  %18=Domain TABLE (%4,%9,%15) Display=3[%4,Domain INTEGER],[%9,Domain CHAR],[%15,Domain
INTEGER],
  %20=TableRowSet %20 V TABLE (%4 INTEGER,%9 CHAR,%15 INTEGER) Display=3 Indexes=[(%4)=[50]]
key (%4)) matches (%9=Thirty) targets: 23=%20 From: %20 Target=23 SRow:(29,66,87) Target:23 P,
  %22=SelectStatement %22 Union=%1)}
```

and the Commit gives

305	Delete Record 214[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
```

332	Record 332[23]	29=2,66=Fifty
376	PTable T	
383	PColumn3 S	for 376(0)[CHAR]
405	PColumn3 U	for 376(1)[INTEGER]
447	Record 447[376]	383=Forty two,405=42
477	Record 477[376]	383=Fifty,405=48
521	PView W 513	select * from t natural join v

Let us restart the server once again (for reproducibility of compiled objects). The view W is

```
{View W TABLE (`31,`32,`40,`51) Display=4[`31, CHAR],[`32, INTEGER],[`40, INTEGER],[`51,
INTEGER] ViewDef view w as select * from t natural join v Result `26}
```

The framing for W is as follows:

```
{Framing obs(
{(`26=SelectRowSet `26 TABLE (`31 CHAR,`32 INTEGER,`40 INTEGER,`51 INTEGER) Display=4 matching
(`31=(`45),`45=(`31)) targets: 23=`56,376=`30 Source: `33,
  `27=SqlStar * `27  CONTENT,
  `28=Domain TABLE (`31,`32,`40,`51) Display=4[`31, CHAR],[`32, INTEGER],[`40, INTEGER],[`51,
INTEGER],
  `30=TableRowSet `30 T TABLE (`31 CHAR,`32 INTEGER) Display=2 targets: 376=`30 From: `33
Target=376 SRow:(383,405) Target:376 T,
  `31=QlInstance `31  CHAR S From:`30 copy from 383,
  `32=QlInstance `32  INTEGER U From:`30 copy from 405,
  `33=JoinRowSet `33 (`31 CHAR,`32 INTEGER,`40 INTEGER,`51 INTEGER|`45 CHAR) Display=4 matching
(`31=(`45),`45=(`31)) targets: 23=`56,376=`30 INNER First: `59 Second: `60 on `31=`45,
  `36=View V TABLE (`40,`45,`51) Display=3[`40,Domain INTEGER],[`45,Domain CHAR],[`51,Domain
INTEGER] ViewDef view v as select q,r as s,a from p Result `37,
  `37=SelectRowSet `37 V TABLE (`40 INTEGER,`45 CHAR,`51 INTEGER) Display=3 key (`40)) targets:
23=`56 From: `33 Source: `56,
  `40=QlInstance Q `40[Q]  INTEGER From:`56 copy from 29,
  `45=QlInstance R `45[R]  CHAR From:`56 Alias=S copy from 66,
  `51=QlInstance A `51[A]  INTEGER From:`56 copy from 87,
  `54=Domain TABLE (`40,`45,`51) Display=3[`40,Domain INTEGER],[`45,Domain CHAR],[`51,Domain
INTEGER],
  `56=TableRowSet `56 V TABLE (`40 INTEGER,`45 CHAR,`51 INTEGER) Display=3 Indexes=[(`40)=[50]]
key (`40)) targets: 23=`56 From: `56 Target=23 SRow:(29,66,87) Target:23 P,
  `58=SelectStatement `58 Union=`37,
  `59=OrderedRowSet `59 T TABLE (`31 CHAR,`32 INTEGER) Display=2 key (`31) order (`31) targets:
376=`30 From: `33 Source: `30,
  `60=OrderedRowSet `60 V TABLE (`40 INTEGER,`45 CHAR,`51 INTEGER) Display=3 key (`45) order
(`45) targets: 23=`56 From: `33 Source: `37,
  `62=SelectStatement `62 Union=`26)}
```

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

Once again, view W will usually be used in a query, so that the above framing makes this use almost trivial. The framing objects will be instanced to use heap uids, and the resulting instances of TableRowSets such as `37 and `66 will be modified to use actual column uids from the referencing query. The rowset review process may simplify the instance, for example the ordering steps `71, `72 may not be required if a where-condition limits the rows involved.

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

```
         1         2         3
1234567890123456789012345678901 2
update w set u=50,a=21 where q=2
```

This should involve updates to both table P and table T, highlighted below. At UpdateSearch.Obey(), line 4013 of Executable.cs, the context contains

```
{(#1=UpdateSearch #1 Target: %1,
  #16=50,
  #21=21,
  #31=SqlValueExpr #31  BOOLEAN From:%31 Left:%15 Right:#32 #31(%15=#32),
  #32=2,
  %0=View W TABLE (%6,%7,%15,%26) Display=4[%6,Domain CHAR],[%7,Domain INTEGER],[%15,Domain
INTEGER],[%26,Domain INTEGER] ViewDef view w as select * from t natural join v Result %1,
  %1=SelectRowSet %1 W TABLE (%6 CHAR,%7 INTEGER,%15 INTEGER,%26 INTEGER) Display=4)) matches
(%15=2) matching (%6=(%20),%20=(%6)) targets: 23=%31,376=%5 Source: %8 Assigs:(UpdateAssignment
Vbl: %7 Val: #16=True,UpdateAssignment Vbl: %26 Val: #21=True),
  %2=SqlStar * %2  CONTENT,
  %3=Domain TABLE (%6,%7,%15,%26) Display=4[%6,Domain CHAR],[%7,Domain INTEGER],[%15,Domain
INTEGER],[%26,Domain INTEGER],
  %5=TableRowSet %5 W TABLE (%6 CHAR,%7 INTEGER) Display=2)) targets: 376=%5 From: %8
Assigs:(UpdateAssignment Vbl: %7 Val: #16=True) Target=376 SRow:(383,405) Target:376 T,
  %6=QlInstance %6  CHAR S From:%5 copy from 383,
  %7=QlInstance %7  INTEGER U From:%5 copy from 405,
  %8=JoinRowSet %8 W (%6 CHAR,%7 INTEGER,%15 INTEGER,%26 INTEGER|%20 CHAR) Display=4)) matches
(%15=2) matching (%6=(%20),%20=(%6)) targets: 23=%31,376=%5 Assigs:(UpdateAssignment Vbl: %7
Val: #16=True,UpdateAssignment Vbl: %26 Val: #21=True) INNER First: %34 Second: %35 on %6=%20,
  %11=View V TABLE (%15,%20,%26) Display=3[%15,Domain INTEGER],[%20,Domain CHAR],[%26,Domain
INTEGER] ViewDef view v as select q,r as s,a from p Result `37,
  %12=SelectRowSet %12 W TABLE (%15 INTEGER,%20 CHAR,%26 INTEGER) Display=3 key (%15)) where
(#31) matches (%15=2) targets: 23=%31 From: %8 Source: %31 Assigs:( UpdateAssignment Vbl: %26
Val: #21=True),
  %15=QlInstance Q %15[Q]  INTEGER From:%31 copy from 29,
  %20=QlInstance R %20[R]  CHAR From:%31 Alias=S copy from 66,
  %26=QlInstance A %26[A]  INTEGER From:%31 copy from 87,
  %29=Domain TABLE (%15,%20,%26) Display=3[%15,Domain INTEGER],[%20,Domain CHAR],[%26,Domain
INTEGER],
  %31=TableRowSet %31 W TABLE (%15 INTEGER,%20 CHAR,%26 INTEGER) Display=3 Indexes=[(%15)=[50]]
key (%15)) where (#31) matches (%15=2) targets: 23=%31 From: %31 Assigs:(UpdateAssignment Vbl:
%26 Val: #21=True) Target=23 SRow:(29,66,87) Target:23 P,
  %33=SelectStatement %33 Union=%12,
  %34=OrderedRowSet %34 W TABLE (%6 CHAR,%7 INTEGER) Display=2 key (%6) order (%6) targets:
376=%5 From: %8 Source: %5 Assigs:(UpdateAssignment Vbl: %7 Val: #16=True),
  %35=OrderedRowSet %35 W TABLE (%15 INTEGER,%20 CHAR,%26 INTEGER) Display=3 key (%20) order
(%20) where (#31) matches (%15=2) targets: 23=%31 From: %8 Source: %12 Assigs:( UpdateAssignment
Vbl: %26 Val: #21=True),
  %37=SelectStatement %37 Union=%1)}
```

Notice that the where-condition and the assignments for the update have been passed down into the appropriate TableRowSets.

Execute() starts an Activation to handle the update, and calls #1.Obey(). This sets up TableActivations for the two target TableRowSets %31 and %5, and each constructs a TransitionRowSet whose cursors give the desired updates.

After Commit, we see in the log:

588	Update 332[23]	87=21,Prev:332
607	Update 477[376]	405=50,Prev:477

6.7 Prepared Statement implementation

For completeness, here is an example showing the implementation of PreparedStatement, 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 Parser.cs near the end of the ParseSql(PreparedStatement..) method just before the call to QParams() (line 328).

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 %35 Target: UpdateSearch %16 Target: %17 Params: %23,%32}
```

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

```
{(%16=UpdateSearch %16 Target: %17,
  %17=TableRowSet %17 SCE TABLE (%18 INTEGER,%19 CHAR) Display=2 Indexes=[(%18)=[2368]] key
(%18) where (%30) targets: 2336=%17 From: %17 Assigs:(UpdateAssignment Vbl: %18 Val: %23=True)
Target=2336 SRow:(2345,2389) Target:2336 SCE,
  %18=QlInstance %18 Domain INTEGER A From:%17 copy from 2345,
  %19=QlInstance %19 Domain CHAR B From:%17 copy from 2389,
  %23=?%23,
  %30=SqlValueExpr %30  BOOLEAN From:%17 Left:%19 Right:%32 %30(%19=%32),
  %32=?%32,
  %35=PreparedStatement %35 Target: UpdateSearch %16 Target: %17 Params: %23,%32)}
```

and cx.values has become

```
{(%23=6,%32=HalfDozen)}
```

The highlighted query operators here simply collect the value from cx.values. Step to the next line (329), and we see that the parameters have been used to modify the context:

```
{(%16=UpdateSearch %16 Target: %17,
  %17=TableRowSet %17 SCE TABLE (%18 INTEGER,%19 CHAR) Display=2 Indexes=[(%18)=[2368]] key
(%18) where (%30) targets: 2336=%17 From: %17 Assigs:(UpdateAssignment Vbl: %18 Val: %23=True)
Target=2336 SRow:(2345,2389) Target:2336 SCE,
  %18=QlInstance %18 Domain INTEGER A From:%17 copy from 2345,
  %19=QlInstance %19 Domain CHAR B From:%17 copy from 2389,
  %23=6,
  %30=SqlValueExpr %30  BOOLEAN From:%17 Left:%19 Right:%32 %30(%19=%32),
  %32=HalfDozen,
  %35=PreparedStatement %35 Target: UpdateSearch %16 Target: %17 Params: %23,%32)}
```

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

In this section we work through the test in test 18 of the PyrrhoTest program.

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

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

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

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

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

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

Pos	Desc
23	PTable AUTHOR

34	PColumn3 ID for 23(0)[INTEGER]
56	PIndex AUTHOR on 23(34) PrimaryKey
77	PColumn3 ANAME for 23(1)[CHAR]
120	PTable BOOK
129	PColumn3 ID for 120(0)[INTEGER]
151	PIndex BOOK on 120(129) PrimaryKey
171	PColumn3 AUTHID for 120(1)[INTEGER]
198	PIndex1 on 120(171) ForeignKey, RestrictUpdate, RestrictDelete refers to [56]
215	PColumn3 TITLE for 120(2)[CHAR]
258	Record 258[23]: 34=1,77=Dickens
283	Record 283[23]: 34=2,77=Conrad
325	Record 325[120]: 129=1,171=1,215=Dombey & Son
363	Record 363[120]: 129=2,171=2,215=Lord Jim
397	Record 397[120]: 129=3,171=1,215=David Copperfield

Place a breakpoint in ParseProcedureClause at line 4428. 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[55] 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 4455 we have a Procedure pr:

```
{Procedure !0 Definer=-502 LastChange=!0 Table BOOKSBY (`4)[`4, CHAR] rows 0 Arity=1  Params(`0)
Body:_ Clause{System.Char[]}}
```

The Transaction's role (for name/signature lookup) and its list of procedures (by uid, for execution) have structures called **procedures**, which are initially empty. At this point, these are respectively

```
cx.db.role.procedures {(BOOKSBY=((0= CHAR)=!0⁵⁶))}
```
cx.db.role.procedures {(BOOKSBY=((0= CHAR)=!0[56]))}

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

These are sufficient to resolve any recursive references to the new procedure. Next, the body of the procedure is parsed (line 4490), so that by line 4508 the context contains

```
{(23=Table  TABLE  (34,77)[34,Domain  INTEGER],[77,  CHAR]  rows  2  Indexes:((34)56)  KeyCols:
(34=True),
  34=TableColumn 34 Definer=-502 LastChange=34 Domain INTEGER Table=23,
  77=TableColumn 77 Definer=-502 LastChange=77 Domain CHAR Table=23,
  120=Table  TABLE  (129,171,215)[129,Domain  INTEGER],[171,Domain  INTEGER],[215,  CHAR]  rows  3
Indexes:((129)151;(171)198) KeyCols: (129=True,171=True),
  129=TableColumn 129 Definer=-502 LastChange=129 Domain INTEGER Table=120,
  171=TableColumn 171 Definer=-502 LastChange=171 Domain INTEGER Table=120,
  215=TableColumn 215 Definer=-502 LastChange=215 Domain CHAR Table=120,
  !0=Procedure  !0  Definer=-502  LastChange=!0  Table  BOOKSBY  (`4)[`4,  CHAR]  rows  0  Arity=1
Params(`0) Body:`57 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)},
  `0=FormalParameter `0  CHAR AUTH From:!0 IN,
  `2=Domain ROW (`0)[`0, CHAR],
  `3=Table rows 0,
  `4=SqlElement TITLE `4  CHAR From:!0,
  `12=SelectRowSet `12 TABLE (`15 CHAR) Display=1 matching (`21=(`27),`27=(`21)) targets:
23=`20,120=`25 Source: `23,
  `15=QlInstance TITLE `15[TITLE] Domain CHAR From:`25 copy from 215,
  `18=Domain TABLE (`15) Display=1[`15,Domain CHAR],
  `20=TableRowSet `20 AUTHOR TABLE (`21 INTEGER,`22 CHAR) Display=2 Indexes=[(`21)=[56]] key
(`21) order (`21) where (`47) targets: 23=`20 From: `23 Target=23 SRow:(34,77) Target:23 AUTHOR,
  `21=QlInstance `21 Domain INTEGER ID From:`20 Alias=AUTHOR.ID copy from 34,
  `22=QlInstance `22 Domain CHAR ANAME From:`20 copy from 77,
```

---

[55] In fact, we use the signature (sequence of parameter types) to distinguish procedures. The standard limits the amount of overloading that is permitted by requiring similarly named procedures to have different arity.

[56] 4611686018427387904 is 0x4000000000000000 which we write as !0. !0,!1,.. are the positions of "Physical" objects prepared for Transaction.Commit.

```
  `23=JoinRowSet `23 (`21 INTEGER,`22 CHAR,`26 INTEGER,`27 INTEGER,`15 CHAR) Display=5 where
(`47) matching (`21=(`27),`27=(`21)) targets: 23=`20,120=`25 INNER JoinCond: (`34) First: `20
Second: `25 on `21=`27,
  `25=TableRowSet   `25   BOOK   TABLE   (`26   INTEGER,`27   INTEGER,`15   CHAR)   Display=3
Indexes=[(`26)=[151],(`27)=[198]] key (`27) order (`27) targets: 120=`25 From: `23 Target=120
SRow:(129,171,215) Target:120 BOOK Alias: B,
  `26=QlInstance `26 Domain INTEGER ID From:`25 Alias=B.ID copy from 129,
  `27=QlInstance `27 Domain INTEGER AUTHID From:`25 copy from 171,
  `34=SqlValueExpr `34  BOOLEAN From:`20 Left:`21 Right:`27 `34(`21=`27),
  `47=SqlValueExpr `47  BOOLEAN From:`20 Left:`22 Right:`0 `47(`22=`0),
  `54=SelectStatement `54 Union=`12,
  `55=SqlValueSelect   `55   SelectRowSet   `12   TABLE   (`15   CHAR)   Display=1   matching
(`21=(`27),`27=(`21)) targets: 23=`20,120=`25 Source: `23  (`12),
  `57=ReturnStatement `57 -> `55)}
```

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

The framing (containing objects `0 to `57) is stored in the procedure definition !0. With these entries the procedure can be used *within the transaction*. Let's place a breakpoint in Start.cs at line 436, and verify this:

select * from table(booksby('Dickens'))
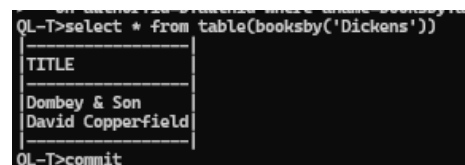
In the cx.obs we see the above items, together with the result of parsing the select statement:

```
{(!0=Procedure !0 Definer=-502 LastChange=!0 Table BOOKSBY (`4)[`4, CHAR] rows 0 Arity=1
Params(`0) Body:`57 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)},
  #166=SelectRowSet #166 BOOKSBY TABLE (`4 CHAR) Display=1 targets: !0=%2 Source: %2,
  #173=SqlStar * #173  CONTENT,
  #185=SqlProcedureCall #185 Table BOOKSBY (`4)[`4, CHAR] rows 0   !0 (#194),
  #194=Dickens,
  `0=FormalParameter `0  CHAR AUTH From:!0 IN,
  `2=Domain ROW (`0)[`0, CHAR],
  `3=Table rows 0,
  `4=SqlElement TITLE `4  CHAR From:!0,
  `12=SelectRowSet  `12 TABLE (`15 CHAR) Display=1 matching (`21=(`27),`27=(`21)) targets:
23=`20,120=`25 Source: `23,
  `15=QlInstance TITLE `15[TITLE] Domain CHAR From:`25 copy from 215,
  `18=Domain TABLE (`15) Display=1[`15,Domain CHAR],
  `20=TableRowSet `20 AUTHOR TABLE (`21 INTEGER,`22 CHAR) Display=2 Indexes=[(`21)=[56]] key
(`21) order (`21) where (`47) targets: 23=`20 From: `23 Target=23 SRow:(34,77) Target:23 AUTHOR,
  `21=QlInstance `21 Domain INTEGER ID From:`20 Alias=AUTHOR.ID copy from 34,
  `22=QlInstance `22 Domain CHAR ANAME From:`20 copy from 77,
  `23=JoinRowSet `23 (`21 INTEGER,`22 CHAR,`26 INTEGER,`27 INTEGER,`15 CHAR) Display=5 where
(`47) matching (`21=(`27),`27=(`21)) targets: 23=`20,120=`25 INNER JoinCond: (`34) First: `20
Second: `25 on `21=`27,
  `25=TableRowSet   `25   BOOK   TABLE   (`26   INTEGER,`27   INTEGER,`15   CHAR)   Display=3
Indexes=[(`26)=[151],(`27)=[198]] key (`27) order (`27) targets: 120=`25 From: `23 Target=120
SRow:(129,171,215) Target:120 BOOK Alias: B,
  `26=QlInstance `26 Domain INTEGER ID From:`25 Alias=B.ID copy from 129,
  `27=QlInstance `27 Domain INTEGER AUTHID From:`25 copy from 171,
  `34=SqlValueExpr `34  BOOLEAN From:`20 Left:`21 Right:`27 `34(`21=`27),
  `47=SqlValueExpr `47  BOOLEAN From:`20 Left:`22 Right:`0 `47(`22=`0),
  `54=SelectStatement `54 Union=`12,
  `55=SqlValueSelect   `55   SelectRowSet   `12   TABLE   (`15   CHAR)   Display=1   matching
(`21=(`27),`27=(`21)) targets: 23=`20,120=`25 Source: `23  (`12),
  `57=ReturnStatement `57 -> `55,
  %0=Domain TABLE (`4) Display=1[`4, CHAR],
  %1=ProcRowSet %1 BOOKSBY (`4 CHAR) Display=1 targets: !0=%1 Call: #185,
  %2=ProcRowSet %2 BOOKSBY (`4 CHAR) Display=1 targets: !0=%2 Call: #185,
  %4=SelectStatement %4 Union=#166)}
```

We see that the body of the procedure has been placed *unchanged* in the context. Proceeding from the breakpoint, we get the expected result. Now commit the transaction, and restart the server.

When the database is reloaded, the compiled object gets reconstructed in Compiled.OnLoad (see file PProcedure.cs, line 173), so that the in-memory database has similar framing objects. The Procedure object itself now has a permanent defining position 446, and following a server restart the framing objects have moved somewhat, so that if we repeat the above select, with a breakpoint at line 432 in Start.cs, we see that the context objects include

```
{Procedure 458 Definer=-502 LastChange=458 Table BOOKSBY (`5)[`5, CHAR] rows 0 Arity=1
Params(`0) Body:`57 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)}}
```

The framing mechanism ensures that the body of the procedure is safely shared by all executions of the procedure, and avoids using run-time interpretation of the procedure source code.

6.9 User-defined Types Implementation

Instead of PTable, CREATE TYPE results in a PType with the type name and any other domain properties of the type, and PMethod records for its methods (initially without bodies). The resulting UDType structure in the database *is a subclass of Table, and can have rows*[57]. The framing objects of the UDType consist of the virtual table and method header information, while the framing objects of the methods follow the same pattern as for Procedures. Instancing a UDType brings the domain information into the Context and instances methods that have bodies.

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

create type point as (x int, y int)

create type rectsize as (w int,h int)

create type line as (strt point,en point)

[create type rect as (tl point,sz rectsize)
  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	PType POINT TYPE
45	PColumn3 X for 23(0)[INTEGER]
66	PColumn3 Y for 23(0)[INTEGER]
105	PType SIZE TYPE
130	PColumn3 W for 105(0)[INTEGER]
151	PColumn3 H for 105(0)[INTEGER]
190	PType LINE TYPE
212	PColumn3 STRT for 190(0)[23]
237	PColumn3 EN for 190(0)[23]
278	PType RECT TYPE
300	PColumn3 TL for 278(0)[23]
323	PColumn3 SZ for 278(0)[105]
346	Method Constructor 346=278.RECT(x1 int,y1 int, x2 int, y2 int)
398	Method Instance 398=278.CENTRE() returns point

We can see from this that the structure of a user-defined type is implemented following the same pattern as table creation, 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 DBObject in the Database[58] for a simple PType such as POINT is

```
{23 UDType POINT TYPE (45,66)[45,Domain INTEGER],[66,Domain INTEGER] rows 0}
```

As we can see, this is an object that can have rows, and retrieving the object from the database gives the current rowset (for the transaction). RECTSIZE and LINE are similar. We will define the method bodies for RECT below: for now we just have the heading and return types (the Domain references are verbose, and shortened here):

```
{Method 346 Definer=-502 LastChange=346 278 UDType RECT TYPE (300,323)[..],[323,105 UDType
RECTSIZE TYPE (130,151)[..] rows 0] rows 0 Arity=4  Params(`5,`7,`9,`11) Body:_ Clause{(x1
int,y1 int, x2 int, y2 int)} UDType=278 UDType RECT TYPE (300,323)[..],[66,Domain INTEGER] rows
0],[..],[151,Domain INTEGER] rows 0] rows 0 MethodType=Constructor}
```

Since it constructs a RECT, its value is an object that can have rows. The executable-range uids `5 etc refer to objects in the 346.framing.obs:

---

[57] If the type is declared UNDER another type, its rows are also stored in its supertype.

[58] To inspect the corresponding objects in the database using the debugger, the simplest way is to place a breakpoint in Start.cs around line 432 (the call to res.First()) and give a harmless command such as select 67. Then examine db.objects`.root.gtr.gtr.. (there are a lot of system definitions with negative uids).

```
{(`5=FormalParameter `5  INTEGER X1 From:346 IN,
 `7=FormalParameter `7  INTEGER Y1 From:346 IN,
 `9=FormalParameter `9  INTEGER X2 From:346 IN,
 `11=FormalParameter `11  INTEGER Y2 From:346 IN,
 `13=Domain   ROW    (`5,`7,`9,`11)[`5,Domain     INTEGER],[`7,Domain    INTEGER],[`9,Domain
INTEGER],[`11,Domain INTEGER])}
```

They are reconstructed slightly differently when the database is reloaded (they will be `0,`2,`4,`6 below). The test now proceeds to create some tables that use these types, in order to illustrate that procedure bodies can be defined or changed later:

```
create table figure(id int primary key,title char)
create table figureline(id int primary key,fig int references figure,what line)
create table figurerect(id int primary key,fig int references figure,what rect)
```

The body of a method is declared in SQL using a CREATE statement, such as the next step in the example:

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

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

| 876 | Modify RECT[346] to begin tl=point(x1,y1); sz=size(x2-x1,y2-y1) end |

Inspecting with the debugger, we see that the Method in db.objects[346] above has been modified to have a body, and the method is now callable as in the previous section. Following a server restart, the framing may be a little different, and the illustration shows the final positions of the parameters and its body, and the framing objects:

```
{[346, {Method 346 Definer=-502 LastChange=346 278 UDType RECT TYPE (300,323)[..] rows 0 Methods:
346 RECT Arity=4  Params(`0,`2,`4,`6) Body:`16 Clause{(x1 int,y1 int, x2 int, y2 int)} UDType=278
UDType RECT TYPE (300,323)[..] rows 0 Methods: 346 RECT MethodType=Constructor}]}

{Framing
{(`0=FormalParameter `0  INTEGER X1 From:346 IN,
 `2=FormalParameter `2  INTEGER Y1 From:346 IN,
 `4=FormalParameter `4  INTEGER X2 From:346 IN,
 `6=FormalParameter `6  INTEGER Y2 From:346 IN,
 `8=Domain ROW (`0,`2,`4,`6)[`0, INTEGER],[`2, INTEGER],[`4, INTEGER],[`6, INTEGER],
 `16=CompoundStatement  `16(`18,`36),
 `18=AssignmentStatement `18 300=`20,
 `20=SqlDefaultConstructor `20 23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER] rows 0
Sce:`33,
 `33=SqlRow `33 Domain ROW (`0,`2) Display=2[`0, INTEGER],[`2, INTEGER]   (`0=FormalParameter
`0  INTEGER X1 From:346 IN,`2=FormalParameter `2  INTEGER Y1 From:346 IN),
 `34=Domain ROW (`0,`2) Display=2[`0, INTEGER],[`2, INTEGER],
 `36=AssignmentStatement `36 323=`38,
 `38=SqlDefaultConstructor `38 105 UDType RECTSIZE TYPE (130,151)[130, INTEGER],[151, INTEGER]
rows 0  Sce:`63,
 `45=SqlValueExpr `45  INTEGER From:346 Left:`4 Right:`0 `45(`4-`0),
 `56=SqlValueExpr `56  INTEGER From:346 Left:`6 Right:`2 `56(`6-`2),
 `63=SqlRow `63 Domain ROW (`45,`56) Display=2[`45, INTEGER],[`56, INTEGER]  (`45=SqlValueExpr
`45  INTEGER From:346 Left:`4 Right:`0 `45(`4-`0),`56=SqlValueExpr `56  INTEGER From:346 Left:`6
Right:`2 `56(`6-`2)),
 `64=Domain ROW (`45,`56) Display=2[`45, INTEGER],[`56, INTEGER])}
```

We see that the body of the procedure is given by the CompoundStatement `16. Note the calls on default constructors for Point at `20 and Rectsize at `38. A constructor does not have a Return statement, as the fields for the new object (Point or Rect) are collected from local variables in the Activation.

Before the body of a method is executed, the values of its top-level fields are placed in the context, which acts as a procedure stack. Expressions of form TL.X are short-circuited using a special QlValue called SqlField, which simply selects the appropriate component of the left-hand value. 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)]

Method 398 with its framing.obs, (showing permanent uids following another server restart) is:

```
{[398, {Method 398 Definer=-502 LastChange=398 23 UDType POINT TYPE (45,66)[45, INTEGER],[66,
INTEGER] rows 0 Arity=0  Params() Body:`112 Clause{() returns point} UDType=278 UDType RECT TYPE
(300,323)[..] rows 0 Methods: 346 RECT,398 CENTRE MethodType=Instance}]}

{Framing (
{(`69=SqlDefaultConstructor `69 23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER] rows 0
Sce:`110,
  `76=QlInstance `76 23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER] rows 0 TL copy from
300,
  `77=QlInstance `77  INTEGER X From:`76 copy from 45,
  `79=SqlValueExpr `79  INTEGER From:`76 Left:`77 Right:`88 `79(`77+`88),
  `85=QlInstance `85 105 UDType RECTSIZE TYPE (130,151)[130, INTEGER],[151, INTEGER] rows 0 SZ
copy from 323,
  `86=QlInstance `86  INTEGER W From:`85 copy from 130,
  `88=SqlValueExpr `88  INTEGER From:`85 Left:`86 Right:`90 `88(`86/`90),
  `90=2,
  `94=SqlValueExpr `94  INTEGER TL.Y From:_ Left:`76 Right:`95 `94(`76.`95),
  `95=QlInstance `95  INTEGER Y From:`76 copy from 66,
  `98=SqlValueExpr `98  INTEGER From:_ Left:`94 Right:`105 `98(`94+`105),
  `101=SqlValueExpr `101  INTEGER SZ.H From:_ Left:`85 Right:`102 `101(`85.`102),
  `102=QlInstance `102  INTEGER H From:`85 copy from 151,
  `105=SqlValueExpr `105  INTEGER From:_ Left:`101 Right:`107 `105(`101/`107),
  `107=2,
  `110=SqlRow   `110   Domain   ROW   (`79,`98)   Display=2[`79,   INTEGER],[`98,   INTEGER]
(`79=SqlValueExpr `79  INTEGER From:`76 Left:`77 Right:`88 `79(`77+`88),`98=SqlValueExpr `98
INTEGER From:_ Left:`94 Right:`105 `98(`94+`105)),
  `111=Domain ROW (`79,`98) Display=2[`79, INTEGER],[`98, INTEGER],
  `112=ReturnStatement `112 -> `69)}
```

Now try the following (with our usual breakpoint, line 432 of Start.cs):

`select rect(3,4,5,6).centre()`

At the breakpoint (on rb = res.First(cx);), res is SelectRowSet #1 and the context contains

```
{(278=278 UDType RECT TYPE (300,323)[300,23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER]
rows 0],[323,105 UDType RECTSIZE TYPE (130,151)[130, INTEGER],[151, INTEGER] rows 0] rows 0
Methods: 346 RECT,398 CENTRE,
  300=TableColumn 300 Definer=-502 LastChange=300 23 UDType POINT TYPE (45,66)[45, INTEGER],[66,
INTEGER] rows 0 Table=278,
  323=TableColumn  323  Definer=-502 LastChange=323  105  UDType  RECTSIZE  TYPE  (130,151)[130,
INTEGER],[151, INTEGER] rows 0 Table=278,
  #1=SelectRowSet #1 TABLE (#21 POINT) Display=1 Source: %7,
```
```
  #12=SqlConstructor #12 278 UDType RECT TYPE (300,323)[300,23 UDType POINT TYPE (45,66)[45,
INTEGER],[66,  INTEGER]  rows  0],[323,105  UDType  RECTSIZE  TYPE  (130,151)[130,  INTEGER],[151,
INTEGER] rows 0] rows 0 Methods: 346 RECT  From:#1  346 (#13,#15,#17,#19),
```
```
  #13=3,
  #15=4,
  #17=5,
  #19=6,
```
```
  #21=SqlMethodCall #21 23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER] rows 0  From:#1
Var=#12  398 (),
```
```
  %0=ProcRowSet %0 RECT (%1 POINT,%2 RECTSIZE) Display=2 targets: 346=%0 Call: #12,
  %1=QlInstance %1 23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER] rows 0 TL From:%0
copy from 300,
  %2=QlInstance %2 105 UDType RECTSIZE TYPE (130,151)[130, INTEGER],[151, INTEGER] rows 0 SZ
From:%0 copy from 323,
  %3=ProcRowSet %3 CENTRE (%4 INTEGER,%5 INTEGER) Display=2 targets: 398=%3 Call: #21,
  %4=QlInstance %4  INTEGER X From:%3 copy from 45,
  %5=QlInstance %5  INTEGER Y From:%3 copy from 66,
  %6=Domain TABLE (#21) Display=1[#21,23 UDType POINT TYPE (45,66)[45, INTEGER],[66, INTEGER]
rows 0],
  %7=TrivialRowSet %7 TABLE (#21 POINT) Display=1[#21= Null],
  %9=SelectStatement %9 Union=#1)}
```

We see calls #12 and #21 on both methods defined for RECT (highlighted), both of which create
ProcRowSets. Restart the server, so that you have same activation numbers as those illustrated here. Set
a breakpoint in Method.Exec() just before the body is Obeyed (line 118 of Method.cs). Give the above
command once more, and proceed from the breakpoint in Start.cs that we had earlier. The first time the
breakpoint for bd._Obey is hit, the activation act 66 has values including our parameters:

```
act.values {(346= Null,`0=3,`2=4,`4=5,`6=6)}
```

We see the parameters 3,4,5,6 that we have provided. It is instructive to step through the execution of the body, but here we step over it to line 139 where we see that the value returned in cx.val is the result of rect(3,4,5,6):

```
cx.val {[300=[45=3,66=4],323=[130=2,151=2]]}
```

The *third* time the breakpoint is hit, activation 54 has values copied from this target targ and no parameters

```
act.values {(45=3,66=4,130=2,151=2,300=[45=3,66=4],323=[130=2,151=2],346=
Null,398=[300=[45=3,66=4],323=[130=2,151=2]],`0=3,`2=4,`4=5,`6=6,`20=[45=3,66=4],`38=[130=2,15
1=2])}
```

and at line 142 we get the returned value of centre() in cx.val

```
cx.val {[45=4,66=5]}
```
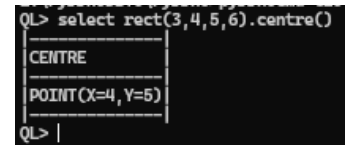

6.10 RESTView implementation

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. We prepare database A for remote access:

```
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"
```

Note that the user name to be supplied is case-sensitive: on Windows it is best to give the domain in upper case and begin the userId with an uppercase letter.

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

As usual, we restart the server to synchronize with the following notes (if we don't, some of the uids below will differ). 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 above committed RestView (in db.objects[23]) will be

```
{RestView WU (`1,`3,`5)[`1, INTEGER],[`3, CHAR],[`5, CHAR] ViewDef (e int, f char, g char) }
```

And the role-based information (infos) is

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

(Recall that the role that we are using is the default -502). Set a breakpoint in Start.cs at line 432. Now try a simple query, such as

```
select * from wu
```

RESTView instancing differs from View instancing in creating QlValues for the columns of the RestView (RestView does not have a Framing), so we get the following objects in the context at the point where traversal begins (Start.cs line 432):

```
{(#1=SelectRowSet #1 WU TABLE (%3 INTEGER,%5 CHAR,%7 CHAR) Display=3 targets: %9=%1 Source: %1,
  #8=SqlStar * #8  CONTENT,
  %0=Domain TABLE (%3,%5,%7) Display=3[%3, INTEGER],[%5, CHAR],[%7, CHAR],
  %1=RestRowSet %1 WU TABLE (%3 INTEGER,%5 CHAR,%7 CHAR) Display=3 targets: %9=%1 Target=%9
SRow:()       http://localhost:8180/A/A/D       RestView      %9      RemoteCols:(%3,%5,%7)
RemoteNames:(%3=E,%5=F,%7=G),
  %3=QlValue E %3  INTEGER,
  %5=QlValue F %5  CHAR,
  %7=QlValue G %7  CHAR,
  %8=Domain TABLE (%3,%5,%7) Display=3[%3, INTEGER],[%5, CHAR],[%7, CHAR],
  %9=RestView TABLE (%3,%5,%7) Display=3[%3, INTEGER],[%5, CHAR],[%7, CHAR] ViewDef (e int, f
char, g char) ,
  %10=Domain TABLE (%3,%5,%7) Display=3[%3, INTEGER],[%5, CHAR],[%7, CHAR],
  %12=SelectStatement %12 Union=#1)}
```

Note that the RestRowSet for the RestView instance has a copy of the metadata. When traversal begins, we soon find we need to build the RestRowSet (see line 6534 in RowSet.cs). This is carried out as a RoundTrip: using a POST request to the remote server for the URL from the metadata. The headers are something like

```
{UserAgent: Pyrrho 7.09alpha
Authorization: Basic ███████████████
If-Unmodified-Since: Wed, 16 Oct 2024 09:53:26 GMT
Accept: application/json
}
```

After placing a debug breakpoint at line 6373, continue execution. In the server window (we have the -H command argument), we see the roundtrip

```
http://localhost:8180/A/A/D
HTTP GET /A/A/D
Received If-Unmodified-Since: Wed, 16 Oct 2024 09:53:26 GMT
Returning ETag: "23,_,155"
---> 2 rows
```

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

At line 6631 the returned information is in the string sr,

```
[{"E": 1, "F": 'Joe', "G": 'Soap', "$pos": 126, "$check": 126},{"E": 2, "F": 'Betty', "G":
'Boop', "$pos": 155, "$check": 155}]
```

and we Parse it into a TArray as the aVal field of the RestRowSet

```
{[[%3=1,%5=Joe,%7=Soap],[%3=2,%5=Betty,%7=Boop]]}.
```

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

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

Remove any breakpoints added in RowSetr.cs above.

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

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

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

```
Pos Desc                                                        Type       Affects Transaction
─── ─────────────────────────────────────────────────────────  ─────────  ─────── ───────────
5   PTransaction for 2 Role=-502 User=-501 Time=15/10/2024 17:18:01  PTransaction 5   0
23  PRestView WU (e int, f char, g char)                        RestView   23      5
58  PMetadata WU{ETAG:ETAG,URL:http://localhost:8180/A/A/D}     Metadata   58      5
116 PTransaction for 5 Role=-502 User=-501 Time=15/10/2024 17:33:02  PTransaction 116 0
134 PTable HU                                                   PTable     134     116
141 PColumn3 E for 134(0)[ INTEGER]                             PColumn3   141     116
163 PIndex HU on 134(141) PrimaryKey                            PIndex     163     116
182 PColumn3 K for 134(1)[ CHAR]                                PColumn3   182     116
204 PColumn3 M for 134(2)[ INTEGER]                             PColumn3   204     116
227 PTransaction for 2 Role=-502 User=-501 Time=15/10/2024 17:33:03  PTransaction 227 0
245 Record 245[134]: 141=1[INTEGER],182=Cleaner[CHAR],204=12500[INTEGER] Record 245 227
280 Record 280[134]: 141=2[INTEGER],182=Manager[CHAR],204=31400[INTEGER] Record 280 227
315 PTransaction for 1 Role=-502 User=-501 Time=15/10/2024 17:33:03  PTransaction 315 0
333 PView VU 333 view VU as select * from WU natural join HU    PView      333     315
─── ─────────────────────────────────────────────────────────  ─────────  ─────── ───────────
DL>
```

We see that VU is committed at file position 333, and if we restart the server we see it is

```
{View VU TABLE (`9,`11,`13,`21,`22) Display=5[`9, INTEGER],[`11, CHAR],[`13, CHAR],[`21,
CHAR],[`22, INTEGER] ViewDef view VU as select * from WU natural join HU Result `2}
```

VU is a View, not a RestView, and so it has a framing:

```
{Framing (
{(`2=SelectRowSet `2 TABLE (`9 INTEGER,`11 CHAR,`13 CHAR,`21 CHAR,`22 INTEGER) Display=5
matching (`9=(`20),`20=(`9)) targets: 134=`19,`15=`7 Source: `17,
 `3=SqlStar * `3 CONTENT,
 `4=Domain TABLE (`9,`11,`13,`21,`22) Display=5[`9, INTEGER],[`11, CHAR],[`13, CHAR],[`21,
CHAR],[`22, INTEGER],
 `7=RestRowSet `7 WU TABLE (`9 INTEGER,`11 CHAR,`13 CHAR) Display=3 targets: `15=`7 From: `17
Target=`15   SRow:()   http://localhost:8180/A/A/D   RestView   `15   RemoteCols:(`9,`11,`13)
RemoteNames:(`9=E,`11=F,`13=G),
 `9=QlValue E `9  INTEGER,
 `11=QlValue F `11  CHAR,
 `13=QlValue G `13  CHAR,
 `14=Domain TABLE (`9,`11,`13) Display=3[`9, INTEGER],[`11, CHAR],[`13, CHAR],
 `15=RestView TABLE (`9,`11,`13) Display=3[`9, INTEGER],[`11, CHAR],[`13, CHAR] ViewDef (e int,
f char, g char) ,
 `16=Domain TABLE (`9,`11,`13) Display=3[`9, INTEGER],[`11, CHAR],[`13, CHAR],
 `17=JoinRowSet `17 (`9 INTEGER,`11 CHAR,`13 CHAR,`21 CHAR,`22 INTEGER|`20 INTEGER) Display=5
matching (`9=(`20),`20=(`9)) targets: 134=`19,`15=`7 INNER First: `23 Second: `19 on `9=`20,
 `19=TableRowSet `19 HU TABLE (`20 INTEGER,`21 CHAR,`22 INTEGER) Display=3
Indexes=[(`20)=[163]] key (`20) order (`20) targets: 134=`19 From: `17 Target=134
SRow:(141,182,204) Target:134 HU,
 `20=QlInstance `20  INTEGER E From:`19 copy from 141,
 `21=QlInstance `21  CHAR K From:`19 copy from 182,
 `22=QlInstance `22  INTEGER M From:`19 copy from 204,
 `23=OrderedRowSet `23 WU TABLE (`9 INTEGER,`11 CHAR,`13 CHAR) Display=3 key (`9) order (`9)
targets: `15=`7 From: `17 Source: `7,
 `25=SelectStatement `25 Union=`2)}
```

The framing for view VU contains an instance of the RestRowSet WU from above, highlighted. Note that there is only one OrderedRowSet needed for the join. 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 436 again), the Context contains:

```
{(134=Table   TABLE   (141,182,204)[141,   INTEGER],[182,   CHAR],[204,   INTEGER]  rows  2
Indexes:((141)163) KeyCols: (141=True),
 #1=SelectRowSet #1 TABLE (#8 INTEGER,#11 CHAR,#14 INTEGER) Display=3)) matches (#8=1) matching
(#8=(%20),%9=(%20),%20=(#8,%9)) targets: 134=%19,%15=%7 Source: %2,
 #8=QlValue E #8  INTEGER From:%17,
 #11=QlValue F #11  CHAR,
 #14=QlInstance #14  INTEGER M From:%19 copy from 204,
 #31=SqlValueExpr #31  BOOLEAN From:%17 Left:#8 Right:#32 #31(#8=#32),
 #32=1,
 %0=Domain TABLE (#8,#11,#14) Display=3[#8, INTEGER],[#11, CHAR],[#14, INTEGER],
 %1=View VU TABLE (#8,#11,%13,%21,#14) Display=5[#8, INTEGER],[#11, CHAR],[%13, CHAR],[%21,
CHAR],[#14, INTEGER] ViewDef view VU as select * from WU natural join HU Result %2,
```

```
   %2=SelectRowSet %2 VU TABLE (#8 INTEGER,#11 CHAR,%13 CHAR,%21 CHAR,#14 INTEGER) Display=5))
where (#31) matches (#8=1) matching (#8=(%20),%20=(#8)) targets: 134=%19,%15=%7 Source: %17,
   %3=SqlStar * %3  CONTENT,
   %4=Domain TABLE (#8,#11,%13,%21,#14) Display=5[#8, INTEGER],[#11, CHAR],[%13, CHAR],[%21,
CHAR],[#14, INTEGER],
   %7=RestRowSet %7 VU TABLE (#8 INTEGER,#11 CHAR,%13 CHAR) Display=3)) where (#31) matches
(#8=1) targets: %15=%7 From: %17 Target=%15 SRow:() http://localhost:8180/A/A/D RestView %15
RemoteCols:(#8,#11,%13) RemoteNames:(#8=E,#11=F,#32=,%13=G),
   %13=QlValue G %13  CHAR,
   %14=Domain TABLE (#8,#11,%13) Display=3[#8, INTEGER],[#11, CHAR],[%13, CHAR],
   %15=RestView TABLE (#8,#11,%13) Display=3[#8, INTEGER],[#11, CHAR],[%13, CHAR] ViewDef (e int,
f char, g char) ,
   %16=Domain TABLE (#8,#11,%13) Display=3[#8, INTEGER],[#11, CHAR],[%13, CHAR],
   %17=JoinRowSet %17 VU (#8 INTEGER,#11 CHAR,%13 CHAR,%21 CHAR,#14 INTEGER|%20 INTEGER)
Display=5)) where (#31) matches (#8=1,%20=1) matching (#8=(%20),%20=(#8)) targets:
134=%19,%15=%7 INNER First: %7 Second: %19 on #8=%20,
   %19=TableRowSet %19 VU TABLE (%20 INTEGER,%21 CHAR,#14 INTEGER) Display=3
Indexes=[(%20)=[163]] key (%20) order (%20) matches (%20=1) targets: 134=%19 From: %17 Target=134
SRow:(141,182,204) Target:134 HU,
   %20=QlInstance %20  INTEGER E From:%19 copy from 141,
   %21=QlInstance %21  CHAR K From:%19 copy from 182,
   %23=OrderedRowSet %23 VU TABLE (#8 INTEGER,#11 CHAR,%13 CHAR) Display=3 key (#8) order (#8)
targets: %15=%7 From: %17 Source: %7,
   %25=SelectStatement %25 Union=%2,
   %27=SelectStatement %27 Union=#1)}
```

We see that VU's framing objects have been instanced at objects %2 to %25, substitutions for e, f, and m have modified all of the objects with red text, the where-condition E=1 has propagated through the objects together with the short-cut matches conditions (extended for the join and highlighted in yellow: note the interesting case of %19).

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, 16 Oct 2024 09:58:53 GMT
Returning ETag: "23,126,126"
--> 1 rows
```

This time, the ETag is for the single tablerow that was returned.

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

The final parts of test 22 perform an insert on RestView WU and an update on View VU, and these steps occur on the remote server. The first command is

insert into wu values(3,'Fred','Bloggs')

At the start of SqlInsert.Obey the context (an Activation) has

```
{(#1=SqlInsert #1 Target: %0 Value: %11 Columns: [%2,%4,%6],
  #16=SqlRowArray #16 RestRowSet %0 WU TABLE (%2 INTEGER,%4 CHAR,%6 CHAR) Display=3 targets:
%8=%0  Target=%8  SRow:()  http://localhost:8180/A/A/D  RestView  %8  RemoteCols:(%2,%4,%6)
RemoteNames:(%2=E,%4=F,%6=G) #22,
  #22=SqlRow #22 Domain ROW (#23,#25,#32) Display=3[#23, INTEGER],[#25, CHAR],[#32, CHAR]
(#23=3,#25=Fred,#32=Bloggs),
  #23=3,
  #25=Fred,
  #32=Bloggs,
  %0=RestRowSet %0 WU TABLE (%2 INTEGER,%4 CHAR,%6 CHAR) Display=3 targets: %8=%0 Target=%8
SRow:()      http://localhost:8180/A/A/D      RestView      %8      RemoteCols:(%2,%4,%6)
RemoteNames:(%2=E,%4=F,%6=G),
  %2=QlValue E %2  INTEGER,
  %4=QlValue F %4  CHAR,
  %6=QlValue G %6  CHAR,
  %7=Domain TABLE (%2,%4,%6) Display=3[%2, INTEGER],[%4, CHAR],[%6, CHAR],
  %8=RestView TABLE (%2,%4,%6) Display=3[%2, INTEGER],[%4, CHAR],[%6, CHAR] ViewDef (e int, f
char, g char) ,
```

```
%9=Domain TABLE (%2,%4,%6) Display=3[%2, INTEGER],[%4, CHAR],[%6, CHAR],
  %10=Domain ROW (%2,%4,%6) Display=3[%2, INTEGER],[%4, CHAR],[%6, CHAR],
  %11=SqlRowSet %11 WU TABLE (%2 INTEGER,%4 CHAR,%6 CHAR) Display=3 targets: %8=%0 SqlRows
[#22],
  %12=Domain ROW (#23,#25,#32) Display=3[#23, INTEGER],[#25, CHAR],[#32, 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, 16 Oct 2024 10:01:13 GMT
Returning ETag: "23,280,280"
Recording ETag "23,280,280"
--> OK
```

The client at this point correctly shows that no changes have been made to database B

```
QL> insert into wu values(3,'Fred','Bloggs')
QL>
```

But we can verify that table D on database A now has all three rows[59]:

```
C:\PyrrhoDB70\Pyrrho>pyrrhocmd A
QL> table D
|-|-----|-----|
|E|F    |G    |
|-|-----|-----|
|1|Joe  |Soap |
|2|Betty|Boop |
|3|Fred |Bloggs|
|-|-----|-----|
QL>
```

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

```
{(#1=UpdateSearch #1 Target: %1,
  #17=Elizabeth,
  #36=SqlValueExpr #36  BOOLEAN From:%16 Left:%8 Right:#37 #36(%8=#37),
  #37=2,
  %0=View    VU    TABLE    (%8,%10,%12,%20,%21)    Display=5[%8,Domain    INTEGER],[%10,Domain
CHAR],[%12,Domain CHAR],[%20,Domain CHAR],[%21,Domain INTEGER] ViewDef view VU as select * from
WU natural join HU Result %1,
  %1=SelectRowSet %1 VU TABLE (%8 INTEGER,%10 CHAR,%12 CHAR,%20 CHAR,%21 INTEGER) Display=5))
matches    (%8=2)    matching    (%8=(%19),%19=(%8))    targets:    134=%18,%14=%6   Source:   %16
Assigs:(UpdateAssignment Vbl: %10 Val: #17=True),
  %2=SqlStar * %2  CONTENT,
  %3=Domain    TABLE    (%8,%10,%12,%20,%21)    Display=5[%8,Domain    INTEGER],[%10,Domain
CHAR],[%12,Domain CHAR],[%20,Domain CHAR],[%21,Domain INTEGER],
  %6=RestRowSet %6 VU TABLE (%8 INTEGER,%10 CHAR,%12 CHAR) Display=3)) where (#36) matches
(%8=2) targets: %14=%6 From: %16 Assigs:(UpdateAssignment Vbl: %10 Val: #17=True) Target=%14
SRow:()      http://localhost:8180/A/A/D      RestView      %14      RemoteCols:(%8,%10,%12)
RemoteNames:(#37=,%8=E,%10=F,%12=G),
  %8=QlValue E %8  INTEGER From:%16,
  %10=QlValue F %10  CHAR,
  %12=QlValue G %12  CHAR,
  %13=Domain    TABLE    (%8,%10,%12)    Display=3[%8,Domain    INTEGER],[%10,Domain   CHAR],[%12,Domain
CHAR],
  %14=RestView  TABLE  (%8,%10,%12)  Display=3[%8,Domain   INTEGER],[%10,Domain  CHAR],[%12,Domain
CHAR] ViewDef (e int, f char, g char),
  %15=Domain    TABLE    (%8,%10,%12)    Display=3[%8,Domain   INTEGER],[%10,Domain   CHAR],[%12,Domain
CHAR],
  %16=JoinRowSet  %16  VU  (%8  INTEGER,%10  CHAR,%12  CHAR,%20  CHAR,%21  INTEGER|%19  INTEGER)
Display=5))  matches  (%8=2,%19=2)  matching  (%8=(%19),%19=(%8))  targets:  134=%18,%14=%6
Assigs:(UpdateAssignment Vbl: %10 Val: #17=True) INNER First: %6 Second: %18 on %8=%19,
```

---

[59] 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.

```
%18=TableRowSet    %18    VU    TABLE    (%19    INTEGER,%20    CHAR,%21    INTEGER)    Display=3
Indexes=[(%19)=[163]] key (%19) order (%19) matches (%19=2) targets: 134=%18 From: %16 Target=134
SRow:(141,182,204) Target:134 HU,
  %19=QlInstance %19  INTEGER E From:%18 copy from 141,
  %20=QlInstance %20  CHAR K From:%18 copy from 182,
  %21=QlInstance %21  INTEGER M From:%18 copy from 204,
  %22=OrderedRowSet %22 VU TABLE (%8 INTEGER,%10 CHAR,%12 CHAR) Display=3 key (%8) order (%8)
targets: %14=%6 From: %16 Source: %6 Assigs:(UpdateAssignment Vbl: %10 Val: #17=True),
  %24=SelectStatement %24 Union=%1)}
```

We see that the OrderedRowSet has been short-circuited, since the JoinRowSet operands are now %6 and %18, the updateSearch statement's source is for %1, which has targets %6 and %18, and the update assignments have been added to the appropriate target rowSet %16. 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[60].

```
http://localhost:8180/A/A/D/E=2
HTTP GET /A/A/D/E=2
Received If-Unmodified-Since: Wed, 16 Oct 2024 10:09:08 GMT
Returning ETag: "23,155,155"
--> 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, 16 Oct 2024 10:09:08 GMT
Returning ETag: "23,155,331"
Recording ETag "23,155,331"
--> OK
```

In Database A we check this:

```
C:\PyrrhoDB70\Pyrrho>pyrrhocmd A
QL> table D
|-|---------|------|
|E|F        |G     |
|-|---------|------|
|1|Joe      |Soap  |
|2|Elizabeth|Boop  |
|3|Fred     |Bloggs|
|-|---------|------|
QL> |
```

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)

In this section we show how Pyrrho supports transactional behavior for sessions that includes a remote client. Restart the server, delete and re-create database A as above, start again with an empty database t23, and define

```
[create view W of (e int, f char, g char) as get etag
'http://localhost:8180/A/A/D']
create table H (e int primary key, k char, m int)
insert into H values (1,'Cleaner',12500), (2,'Manager',31400)
create view V as select * from w natural join H
```

Apart from the URL metadata flag, everything is as in the last section. The behaviour of the select statements is similar. Let us look at the insert statement:

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

When this runs, we get

---

[60] The screenshot is from a test where database A had seen more changes than the previous illustrations.

```
QL> create view v as select * from w natural join H
QL> begin transaction
QL-T>insert into w values(3,'Fred','Bloggs')
QL-T>|
```

```
RoundTrip HEAD http://localhost:8180/A/A/D
HTTP HEAD /A/A/D
Received If-Unmodified-Since: Wed, 16 Oct 2024 10:21:46 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 this, use a dummy select statement (e.g. select 67) for t23, and look at cx.db.physicals.

```
{(!0=Post D http://localhost:8180/A/A/D insert into D values(3,'Fred','Bloggs'); %8)}
```

To commit the transaction,

commit

When Commit is called on the Post record, it executes the round-trip to perform the insert operation, and we get

```
QL-T>commit
QL> |
```

```
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: "379"
Received If-Unmodified-Since: Wed, 16 Oct 2024 09:21:46 GMT
Returning ETag: "23,280,280"
Recording ETag "23,280,280"
--> OK
```

Next, we look at the update operation on the join/view combination, again with an explicit transaction so that we can see where commit occurs.

begin transaction
update v set f='Elizabeth' where e=2

```
QL> begin transaction
QL-T>update v set f='Elizabeth' where e=2
QL-T>|
```

```
RoundTrip HEAD http://localhost:8180/A/A/D
HTTP HEAD /A/A/D/E=2
Received If-Unmodified-Since: Wed, 16 Oct 2024 10:29:26 GMT
Returning ETag:
Recording ETag
--> OK
http://localhost:8180/A/A select E,F,G from D where (E=2) and E=2
HTTP POST /A/A
select E,F,G from D where (E=2) and E=2
Received If-Unmodified-Since: Wed, 16 Oct 2024 10:29:26 GMT
Returning ETag: "23,155,155"
--> 1 rows
Response ETag: 23,155,155
```

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. Before we commit the transaction, as above look at cx.db.physicals. The POST record this time is

```
{(!0=Post D http://localhost:8180/A/A/D update D set F='Elizabeth' where (E=2) and E=2 %14)}
```

and when this record is committed we get

```
QL-T>commit
QL> |
```

```
RoundTrip POST http://localhost:8180/A/A update D set F='Elizabeth' where (E=2) and E=2

HTTP POST /A/A
update D set F='Elizabeth' where (E=2) and E=2

Received If-Match: "397"
Received If-Unmodified-Since: Wed, 16 Oct 2024 09:29:26 GMT
Returning ETag: "23,155,331"
Recording ETag "23,155,331"
--> OK
```

Test23 includes numerous exercises to verify transactional behaviour and conflict detection for this model.

6.10.3 RestView with a Using table

This example is from Test 24 and demonstrates selection, insertion and update for a RestView specified with a using table. As described earlier, this allows a view to select similar rows from a list of source databases, which are accessed using HTTP/1.1 and/or a REST service. Since Pyrrho provides services of this kind, some of the databases may be accessed using Pyrrho itself. The Pyrrho distribution includes a simple web server that can be tailored to provide a suitable HTTP/1.1 service that accesses other DBMS, and this has been demonstrated for MySQL and SQL Server.

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

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

For example, a filter on remote columns can be applied on the remote database: and an aggregation of remote data on a single remote source can be carried out on the remote DBMS. But it soon becomes apparent that in more complex cases some transformation of the original query is required. For example, a COUNT of a remote datum supplied separately by several DBMS will need to be implemented as a SUM of the contributions from these, and D will have just one row per contributor.

In this section we consider a number of examples involving grouping and aggregation. View syntax allows quite general CursorSpecifications, but the RestView syntax is just a simple Domain and URL, so that all consideration of grouping and aggregation is done at the SelectRowSet level. This makes it feasible to consider which groupings and aggregations can be passed down to remote contributors.

The first part of test 24 simply sets up two local databases as sources for the test, DB and DC, accessible by a remote connection. DB:
```
create table T(E int,F char)
insert into T values(3,'Three'),(6,'Six'),(4,'Vier'),(6,'Sechs')
create role DB
grant DB to "domain\user"
```

And DC:
```
create table U(E int,F char)
insert into U values(5,'Five'),(4,'Four'),(8,'Ate')
create role DC
grant DC to "domain\user"
```

Then in another database (such as t24) we can define a table that lists these sources using Pyrrho's HTTP service. Test24 first defines a RESTView that uses just one of these remote databases

```
create view VV of (E int,F char) as get 'http://localhost:8180/DB/DB/t'
```

and this step is included here for completeness.

```
create table VU (d char primary key, k int, u char)
```

```
insert into VU values('B',4,'http://localhost:8180/DB/DB/t')
insert into VU values('C',1,'http://localhost:8180/DC/DC/u')
```

We define a RESTView that collects data from these sources:

```
create view WW of (E int, D char, K int, F char) as get using VU
```

This is committed as a RestView object. Test 24 also defines another table to test working with local joins, see below:

```
create table M (e int primary key, n char, unique(n))
insert into M values (2,'Deux'),(3,'Trois'),(4,'Quatre')
insert into M values (5,'Cinq'),(6,'Six'),(7,'Sept')
```

At this stage the Log for t24 shows a PRestView2 object at location 435 (VV and M are not shown in this illustration):

Pos	Desc
123	PTable VU
130	PColumn3 D for 123(0)[CHAR]
150	PIndex VU on 123(130) PrimaryKey
168	PColumn3 K for 123(1)[INTEGER]
190	PColumn3 U for 123(2)[CHAR]
229	Record 229[123]: 130=B,168=4,190=http://localhost:8180/DB/DB/t
303	Record 303[123]: 130=C,168=1,190=http://localhost:8180/DC/DC/u
377	PRestView2 WW(E int, D char, K int, F char) using 123
..	..

The PRestView2 record is loaded as a RestView object with a UsingRowSet,

We note that the columns of the RestView WW include two local columns D and K coming from the UsingTable 123. The columns E and F are (only) in the remote view 377, 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:

```
{(123=Table TABLE (130,168,190)[130, CHAR],[168, INTEGER],[190, CHAR] rows 2 Indexes:((130)150)
KeyCols: (130=True),
  130=TableColumn 130 Definer=-502 LastChange=130  CHAR Table=123,
  168=TableColumn 168 Definer=-502 LastChange=168  INTEGER Table=123,
  190=TableColumn 190 Definer=-502 LastChange=190  CHAR Table=123,
  #1=SelectRowSet #1 WW TABLE (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) Display=4 matches (%7=1)
targets: %11=%17 Source: %17,
  #8=SqlStar * #8  CONTENT,
  #25=SqlValueExpr #25  BOOLEAN From:%13 Left:%7 Right:#26 #25(%7=#26),
  #26=1,
  %0=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %1=RestRowSet %1 WW TABLE (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) Display=4 where (#25) matches
(%7=1)   targets:   %11=%1   Target=%11   SRow:()      RestView   %11   RemoteCols:(%3,%9)
RemoteNames:(#26=,%3=E,%9=F),
  %3=QlValue E %3  INTEGER,
  %5=QlInstance %5  CHAR D From:%13 copy from 130,
  %7=QlInstance %7  INTEGER K From:%13 copy from 168,
  %9=QlValue F %9  CHAR,
  %10=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %11=RestView TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR]
ViewDef (E int, D char, K int, F char)  UsingTable: 123,
  %12=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %13=TableRowSet %13 VU (%5 CHAR,%7 INTEGER,%16 CHAR) Display=3 Indexes=[(%5)=[150]] key (%5)
where (#25) matches (%7=1) targets: 123=%13 Target=123 SRow:(130,168,190) Target:123 VU,
  %16=QlInstance %16  CHAR U From:%13 copy from 190,
  %17=RestRowSetUsing %17 WW (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) targets: %11=%17 Target=%11
SRow:(130,168,190) Template: %1 UsingTableRowSet:%13 UrlCol:%16,
  %19=SelectStatement %19 Union=#1)}
```

The REST operation is managed by the RestRowSet %1.

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,_,131"
---> 3 rows
Response ETag: 23,_,131
```

```
QL> select * from ww where k=1
|-|-|-|-----|
|E|D|K|F    |
|-|-|-|-----|
|5|C|1|Five |
|4|C|1|Four |
|8|C|1|Ate  |
|-|-|-|-----|
QL> |
```

Now let's have a filter on the remote table:

select * from ww where e=3

```
{(123=Table TABLE (130,168,190)[130, CHAR],[168, INTEGER],[190, CHAR] rows 2
Indexes:((130)150) KeyCols: (130=True),
  130=TableColumn 130 Definer=-502 LastChange=130  CHAR Table=123,
  168=TableColumn 168 Definer=-502 LastChange=168  INTEGER Table=123,
  190=TableColumn 190 Definer=-502 LastChange=190  CHAR Table=123,
  #1=SelectRowSet #1 WW TABLE (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) Display=4 matches (%3=3)
targets: %11=%17 Source: %17,
  #8=SqlStar * #8  CONTENT,
  #25=SqlValueExpr #25  BOOLEAN From:%17 Left:%3 Right:#26 #25(%3=#26),
  #26=3,
  %0=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %1=RestRowSet %1 WW TABLE (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) Display=4 where (#25)
matches (%3=3) targets: %11=%1 Target=%11 SRow:()  RestView %11 RemoteCols:(%3,%9)
RemoteNames:(#26=,%3=E,%9=F),
  %3=QlValue E %3  INTEGER From:%17,
  %5=QlInstance %5  CHAR D From:%13 copy from 130,
  %7=QlInstance %7  INTEGER K From:%13 copy from 168,
  %9=QlValue F %9  CHAR,
  %10=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %11=RestView TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR]
ViewDef (E int, D char, K int, F char)  UsingTable: 123,
  %12=Domain TABLE (%3,%5,%7,%9) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
  %13=TableRowSet %13 VU (%5 CHAR,%7 INTEGER,%16 CHAR) Display=3 Indexes=[(%5)=[150]] key (%5)
targets: 123=%13 Target=123 SRow:(130,168,190) Target:123 VU,
  %16=QlInstance %16  CHAR U From:%13 copy from 190,
  %17=RestRowSetUsing %17 WW (%3 INTEGER,%5 CHAR,%7 INTEGER,%9 CHAR) where (#25) matches
(%3=3) targets: %11=%17 Target=%11 SRow:(130,168,190) Template: %1 UsingTableRowSet:%13
UrlCol:%16,
  %19=SelectStatement %19 Union=#1)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where (E=3) and E=3
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where (E=3) and E=3
Returning ETag: "23,89,89"
---> 1 rows
Response ETag: 23,89,89
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where (E=3) and E=3
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where (E=3) and E=3
Returning ETag:
---> 0 rows
```

```
QL> select * from ww where e=3
|-|-|-|-----|
|E|D|K|F    |
|-|-|-|-----|
|3|B|4|Three|
|-|-|-|-----|
QL> |
```

Here we see that the test E=3 is now in the RestRowSet %1, 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

```
{(..
```
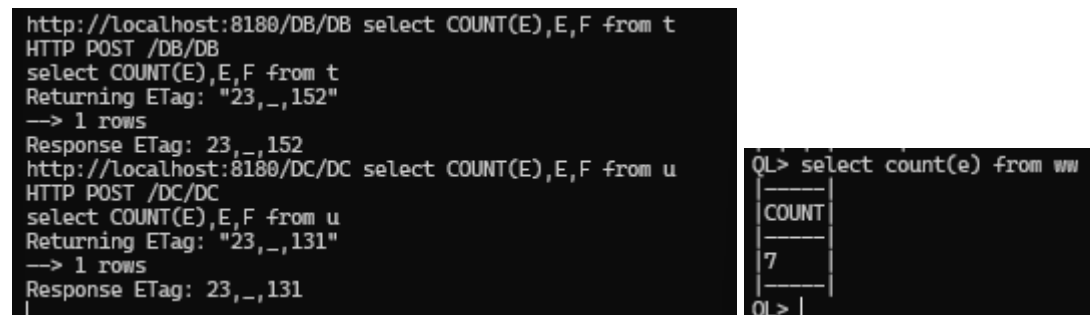
```
   #1=SelectRowSet #1 WW TABLE (#8 INTEGER) Aggs (#8) Display=1 targets: %11=%17 Source: %17,
   #8=SqlFunction #8  INTEGER COUNT From:#1 COUNT(#14),
   #14=QlValue E #14  INTEGER,
   %0=Domain TABLE (#8) Display=1[#8, INTEGER] Aggs (#8),
   %1=RestRowSet %1 WW TABLE (#8 INTEGER,#14 INTEGER|%5 CHAR,%7 INTEGER,%9 CHAR) Aggs (#8)
Display=2  targets:  %11=%1  Target=%11  SRow:()     RestView   %11   RemoteCols:(#14,%9)
RemoteNames:(#14=E,%9=F),
   %5=QlInstance %5  CHAR D From:%13 copy from 130,
   %7=QlInstance %7  INTEGER K From:%13 copy from 168,
   %9=QlValue F %9  CHAR,
   %10=Domain TABLE (#14,%5,%7,%9) Display=4[#14, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
   %11=RestView TABLE (#14,%5,%7,%9) Display=4[#14, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR]
ViewDef (E int, D char, K int, F char)  UsingTable: 123,
   %12=Domain TABLE (#14,%5,%7,%9) Display=4[#14, INTEGER],[%5, CHAR],[%7, INTEGER],[%9, CHAR],
   %13=TableRowSet %13 VU (%5 CHAR,%7 INTEGER,%16 CHAR) Display=3 Indexes=[(%5)=[150]] key (%5)
targets: 123=%13 Target=123 SRow:(130,168,190) Target:123 VU,
   %16=QlInstance %16  CHAR U From:%13 copy from 190,
   %17=RestRowSetUsing %17 WW (#8 INTEGER,#14 INTEGER) Aggs (#8) Display=2 targets: %11=%17
Target=%11 SRow:(130,168,190) Template: %1 UsingTableRowSet:%13 UrlCol:%16,
   %19=SelectStatement %19 Union=#1)}
```

When traversal starts of the SelectRowSet #1, the SelectRowSet.Build() takes change. Place a breakpoint in RestRowSet.RoundTrip() at line 6315 in RowSet.cs.

Then we traverse the UsingTableRowSet %12, and call RestRowSetusing.Build(), we see a JSON document returned, one from each of DB and DC:

```
[{"COUNT": 4, "$#8": {"4": 4}}]
```

```
[{"COUNT": 3, "$#8": {"3": 3}}]
```


We see that the request that was sent to the contributors in RestRowSet.Build(). was select count(E), so that 1 row is returned from each contributor. The REST process supplies the Registers $#8 for the aggregations in addition to the results for contributors, and RestRowSetBuild accumulates them in the global result[61] 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 WW TABLE (#8 UNION) Aggs (#8) Display=1 where (#30) targets: %11=%17
Source: %17,
   #8=SqlFunction #8  CHAR MAX From:#1 MAX(#12),
   #12=QlValue F #12  CHAR,
   #30=SqlValueExpr #30  BOOLEAN From:%17 Left:%3 Right:#31 #30(%3>#31),
   #31=4,
   %0=Domain TABLE (#8) Display=1[#8,Domain UNION of [-67,-135,-152,-179,-203,-257,-258]] Aggs
(#8),
   %1=RestRowSet %1 WW TABLE (#8 CHAR,%3 INTEGER|%5 CHAR,%7 INTEGER,#12 CHAR) Aggs (#8)
Display=2 where (#30) targets: %11=%1 Target=%11 SRow:()  RestView %11 RemoteCols:(%3,#12)
RemoteNames:(#12=F,#31=,%3=E),
   %3=QlValue E %3  INTEGER From:%17,
```

---

[61] The code to do this is in Domain.Coerce (currently at line 3195 of Domain.cs). For full details, see the Pyrrho manual, section 8.8.9.

```
  %5=QlInstance %5  CHAR D From:%13 copy from 130,
  %7=QlInstance %7   INTEGER K From:%13 copy from 168,
  %10=Domain TABLE (%3,%5,%7,#12) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[#12, CHAR],
  %11=RestView TABLE (%3,%5,%7,#12) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[#12,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 123,
  %12=Domain TABLE (%3,%5,%7,#12) Display=4[%3, INTEGER],[%5, CHAR],[%7, INTEGER],[#12, CHAR],
  %13=TableRowSet %13 VU (%5 CHAR,%7 INTEGER,%16 CHAR) Display=3 Indexes=[(%5)=[150]] key (%5)
targets: 123=%13 Target=123 SRow:(130,168,190) Target:123 VU,
  %16=QlInstance %16  CHAR U From:%13 copy from 190,
  %17=RestRowSetUsing %17 WW (#8 CHAR,%3 INTEGER) Aggs (#8) Display=2 where (#30) targets:
%11=%17 Target=%11 SRow:(130,168,190) Template: %1 UsingTableRowSet:%13 UrlCol:%16,
  %19=SelectStatement %19 Union=#1)}
```


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 WW TABLE (#14 UNION,#27 CHAR) Aggs (#8) Display=2 groupSpec: #47
groupings (%19) GroupCols(#27) targets: %11=%17 Source: %17 Ambient(#27),
  #8=SqlFunction #8  INTEGER SUM From:#1 SUM(#12),
  #12=QlValue E #12  INTEGER,
  #14=SqlValueExpr #14 Domain UNION Aggs (#8) of [-135,-179,-203]⁶² From:%17 Left:#8 Right:#15
#14(#8+#15),
  #15=SqlFunction #15  INTEGER CHAR_LENGTH From:#1 CHAR_LENGTH(#27),
  #27=QlValue F #27  CHAR,
  #47=GroupSpecification #47(%19),
  %0=Domain TABLE (#14,#27) Display=2[#14,Domain UNION Aggs (#8) of [-135,-179,-203]],[#27,
CHAR] Aggs (#8),
  %1=RestRowSet %1 WW TABLE (#14 UNION,#27 CHAR|#12 INTEGER,%5 CHAR,%7 INTEGER) Aggs (#8)
Display=2 groupSpec: %24 GroupCols(#27) targets: %11=%1 Target=%11 SRow:()  RestView %11
RemoteCols:(#12,#27) RemoteNames:(#12=E,#27=F),
  %5=QlInstance %5  CHAR D From:%13 copy from 130,
  %7=QlInstance %7   INTEGER K From:%13 copy from 168,
  %10=Domain TABLE (#12,%5,%7,#27) Display=4[#12, INTEGER],[%5, CHAR],[%7, INTEGER],[#27,
CHAR],
  %11=RestView TABLE (#12,%5,%7,#27) Display=4[#12, INTEGER],[%5, CHAR],[%7, INTEGER],[#27,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 123,
  %12=Domain TABLE (#12,%5,%7,#27) Display=4[#12, INTEGER],[%5, CHAR],[%7, INTEGER],[#27,
CHAR],
  %13=TableRowSet %13 VU (%5 CHAR,%7 INTEGER,%16 CHAR) Display=3 Indexes=[(%5)=[150]] key (%5)
targets: 123=%13 Target=123 SRow:(130,168,190) Target:123 VU,
  %16=QlInstance %16  CHAR U From:%13 copy from 190,
  %17=RestRowSetUsing %17 WW (#14 UNION,#27 CHAR,#8 INTEGER) Aggs (#8) Display=3
GroupCols(#27) targets: %11=%17 Target=%11 SRow:(130,168,190) Template: %1
UsingTableRowSet:%13 UrlCol:%16,
  %19=Grouping ROW (#27)[#27, CHAR] (#27=0),
  %20=Domain ROW (#27) Display=1[#27, CHAR],
```

<hr />

⁶² These are internal uids for the standard domains that are allowed in the reply (INT is -135)..

```
    %21=Domain ROW (#27)[#27, CHAR],
    %22=Domain ROW (#27)[#27, CHAR],
    %23=Domain ROW (#27)[#27, CHAR],
    %24=GroupSpecification %24(%19),
    %25=Domain ROW (#27)[#27, CHAR],
    %26=SelectStatement %26 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):

select count(*),k/2 as k2 from ww group by k2

```
{(..
  #1=SelectRowSet #1 WW TABLE (#8 INTEGER,#18 INTEGER) Aggs (#8) Display=2 groupSpec: #41
groupings (%18) GroupCols(#18) targets: %14=%16 Source: %16 Ambient(#18),
  #8=SqlFunction #8  INTEGER COUNT From:#1 COUNT(#14),
  #14=1,
  #17=QlInstance K #17[K]  INTEGER From:%1 copy from 168,
  #18=SqlValueExpr #18  INTEGER K2 From:%16 Left:#17 Right:#19 #18(#17/#19),
  #19=2,
  #41=GroupSpecification #41(%18),
  %0=Domain TABLE (#8,#18) Display=2[#8, INTEGER],[#18, INTEGER] Aggs (#8),
  %1=TableRowSet %1 VU TABLE (%2 CHAR,#17 INTEGER,%3 CHAR) Display=3 Indexes=[(%2)=[150]] key
(%2) targets: 123=%1 From: %1 Target=123 SRow:(130,168,190) Target:123 VU,
  %2=QlInstance %2  CHAR D From:%1 copy from 130,
  %3=QlInstance %3  CHAR U From:%1 copy from 190,
  %4=RestRowSet %4 WW TABLE (#8 INTEGER,#18 INTEGER|%6 INTEGER,%2 CHAR,#17 INTEGER,%12 CHAR)
Aggs (#8) Display=2 groupSpec: %23 groupings (%18) GroupCols(#18) targets: %14=%4 Target=%14
SRow:()  RestView %14 RemoteCols:(%6,%12) RemoteNames:(%6=E,%12=F),
  %6=QlValue E %6  INTEGER,
  %12=QlValue F %12  CHAR,
  %13=Domain TABLE (%6,%2,#17,%12) Display=4[%6, INTEGER],[%2, CHAR],[#17, INTEGER],[%12,
CHAR],
  %14=RestView TABLE (%6,%2,#17,%12) Display=4[%6, INTEGER],[%2, CHAR],[#17, INTEGER],[%12,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 123,
  %15=Domain TABLE (%6,%2,#17,%12) Display=4[%6, INTEGER],[%2, CHAR],[#17, INTEGER],[%12,
CHAR],
  %16=RestRowSetUsing %16 WW (#8 INTEGER,#18 INTEGER) Aggs (#8) Display=2 targets: %14=%16
Target=%14 SRow:(130,168,190) Template: %4 UsingTableRowSet:%1 UrlCol:%3,
  %18=Grouping ROW (#18)[#18, INTEGER] (#18=0),
  %19=Domain ROW (#18) Display=1[#18, INTEGER],
  %20=Domain ROW (#18)[#18, INTEGER],
  %21=Domain ROW (#18)[#18, INTEGER],
  %22=Domain ROW (#18)[#18, INTEGER],
  %23=GroupSpecification %23(%18),
```

```
 %24=Domain ROW (#18)[#18, INTEGER],
 %25=SelectStatement %25 Union=#1)}
```

```
QL> select count(*),k/2 as k2 from ww group by k2
|------|---|
|COUNT |K2 |
|------|---|
|3     |0  |
|4     |2  |
|------|---|
QL> |
```

select avg(e) from ww
```
{(..,
{(23=Table TABLE (30,67,89)[30, CHAR],[67, INTEGER],[89, CHAR] rows 2 Indexes:((30)50)
KeyCols: (30=True),
  30=TableColumn 30 Definer=-502 LastChange=30  CHAR Table=23,
  67=TableColumn 67 Definer=-502 LastChange=67  INTEGER Table=23,
  89=TableColumn 89 Definer=-502 LastChange=89  CHAR Table=23,
  #1=SelectRowSet #1 WW TABLE (#8 NUMERIC) Aggs (#8) Display=1 targets: %15=%17 Source: %17,
  #8=SqlFunction #8  NUMERIC AVG From:#1 AVG(#12),
  #12=QlValue E #12  INTEGER,
  %0=Domain TABLE (#8) Display=1[#8, NUMERIC] Aggs (#8),
  %1=TableRowSet %1 VU TABLE (%2 CHAR,%3 INTEGER,%4 CHAR) Display=3 Indexes=[(%2)=[50]] key
(%2) targets: 23=%1 From: %1 Target=23 SRow:(30,67,89) Target:23 VU,
  %2=QlInstance %2  CHAR D From:%1 copy from 30,
  %3=QlInstance %3  INTEGER K From:%1 copy from 67,
  %4=QlInstance %4  CHAR U From:%1 copy from 89,
  %5=RestRowSet %5 WW TABLE (#8 NUMERIC,#12 INTEGER|%2 CHAR,%3 INTEGER,%13 CHAR) Aggs (#8)
Display=2 targets: %15=%5 Target=%15 SRow:()  RestView %15 RemoteCols:(#12,%13)
RemoteNames:(#12=E,%13=F),
  %13=QlValue F %13  CHAR,
  %14=Domain TABLE (#12,%2,%3,%13) Display=4[#12, INTEGER],[%2, CHAR],[%3, INTEGER],[%13,
CHAR],
  %15=RestView TABLE (#12,%2,%3,%13) Display=4[#12, INTEGER],[%2, CHAR],[%3, INTEGER],[%13,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 23,
  %16=Domain TABLE (#12,%2,%3,%13) Display=4[#12, INTEGER],[%2, CHAR],[%3, INTEGER],[%13,
CHAR],
  %17=RestRowSetUsing %17 WW (#8 NUMERIC,#12 INTEGER) Aggs (#8) Display=2 targets: %15=%17
Target=%15 SRow:(30,67,89) Template: %5 UsingTableRowSet:%1 UrlCol:%4,
  %19=SelectStatement %19 Union=#1)}
```

This time the documents returned from the remote servers are
[{"AVG": 4.75, "$#8": {"4": 19}}]
[{"AVG": 5.66666666666666, "$#8": {"3": 17}}]

```
http://localhost:8180/DB/DB select AVG(E),F from t
HTTP POST /DB/DB
select AVG(E),F from t
Returning ETag: "23,_,148"
--> 1 rows
Response ETag: 23,_,148
http://localhost:8180/DC/DC select AVG(E),F from u
HTTP POST /DC/DC
select AVG(E),F from u
Returning ETag: "23,_,127"
--> 1 rows
Response ETag: 23,_,127
```

```
QL> select avg(e) from ww
|-------------------|
|AVG                |
|-------------------|
|5.142857142857142  |
|-------------------|
QL> |
```

select sum(e)*sum(e),d from ww group by d
```
{(..,
  #1=SelectRowSet #1 WW TABLE (#14 UNION,#22 CHAR) Aggs (#8,#15) Display=2 groupSpec: #38
groupings (%18) GroupCols(#22) targets: %14=%16 Source: %16 Ambient(#22),
  #8=SqlFunction #8  INTEGER SUM From:#1 SUM(#12),
  #12=QlValue E #12  INTEGER,
  #14=SqlValueExpr #14 Domain UNION Aggs (#8,#15) of [-135,-179,-203] From:%16 Left:#8
Right:#15 #14(#8*#15),
  #15=SqlFunction #15  INTEGER SUM From:#1 SUM(#12),
  #22=QlInstance D #22[D]  CHAR From:%1 copy from 30,
  #38=GroupSpecification #38(%18),
  %0=Domain TABLE (#14,#22) Display=2[#14,Domain UNION Aggs (#8,#15) of [-135,-179,-
203]],[#22, CHAR] Aggs (#8,#15),
  %1=TableRowSet %1 VU TABLE (#22 CHAR,%2 INTEGER,%3 CHAR) Display=3 Indexes=[(#22)=[50]] key
(#22) targets: 23=%1 From: %1 Target=23 SRow:(30,67,89) Target:23 VU,
  %2=QlInstance %2  INTEGER K From:%1 copy from 67,
  %3=QlInstance %3  CHAR U From:%1 copy from 89,
```

```
  %4=RestRowSet %4 WW TABLE (#14 UNION,#22 CHAR|#12 INTEGER,%2 INTEGER,%12 CHAR) Aggs (#8,#15)
Display=2 groupSpec: %23 GroupCols(#22) targets: %14=%4 Target=%14 SRow:()  RestView %14
RemoteCols:(#12,%12) RemoteNames:(#12=E,%12=F),
  %12=QlValue F %12  CHAR,
  %13=Domain TABLE (#12,#22,%2,%12) Display=4[#12, INTEGER],[#22, CHAR],[%2, INTEGER],[%12,
CHAR],
  %14=RestView TABLE (#12,#22,%2,%12) Display=4[#12, INTEGER],[#22, CHAR],[%2, INTEGER],[%12,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 23,
  %15=Domain TABLE (#12,#22,%2,%12) Display=4[#12, INTEGER],[#22, CHAR],[%2, INTEGER],[%12,
CHAR],
  %16=RestRowSetUsing %16 WW (#22 CHAR,#14 UNION,#8 INTEGER,#15 INTEGER) Aggs (#8,#15)
Display=4 targets: %14=%16 Target=%14 SRow:(30,67,89) Template: %4 UsingTableRowSet:%1
UrlCol:%3,
  %18=Grouping ROW (#22)[#22, CHAR] (#22=0),
  %19=Domain ROW (#22) Display=1[#22, CHAR],
  %20=Domain ROW (#22)[#22, CHAR],
  %21=Domain ROW (#22)[#22, CHAR],
  %22=Domain ROW (#22)[#22, CHAR],
  %23=GroupSpecification %23(%18),
  %24=Domain ROW (#22)[#22, CHAR],
  %25=SelectStatement %25 Union=#1)}
```

```
http://localhost:8180/DB/DB select (SUM(E)*SUM(E)),'B' as D,F from t group by D
HTTP POST /DB/DB
select (SUM(E)*SUM(E)),'B' as D,F from t group by D
Returning ETag: "23,_,148"
--> 1 rows
Response ETag: 23,_,148
http://localhost:8180/DC/DC select (SUM(E)*SUM(E)),'C' as D,F from u group by D
HTTP POST /DC/DC
select (SUM(E)*SUM(E)),'C' as D,F from u group by D
Returning ETag: "23,_,127"
--> 1 rows
Response ETag: 23,_,127
```

```
QL> select sum(e)*sum(e),d from ww group by d
|————|—|
|Col0|D|
|————|—|
| 361 |B|
| 289 |C|
|————|—|
QL> |
```

Join:
select f,n from ww natural join m

```
{(..,
  332=Table TABLE (339,381)[339, INTEGER],[381, CHAR] rows 6 Indexes:((339)362;(381)404)
KeyCols: (339=True,381=True),
  339=TableColumn 339 Definer=-502 LastChange=339  INTEGER Table=332,
  381=TableColumn 381 Definer=-502 LastChange=381  CHAR Table=332,
  #1=SelectRowSet #1 TABLE (#8 CHAR,#10 CHAR) Display=2 matching (%7=(#18),#18=(%7)) targets:
332=#33,%15=%17 Source: #20,
  #8=QlValue F #8  CHAR,
  #10=QlInstance N #10[N]  CHAR From:#33 copy from 381,
  #20=JoinRowSet #20 (%7 INTEGER,%2 CHAR,%3 INTEGER,#8 CHAR,#10 CHAR|#18 INTEGER) Display=5
matching (%7=(#18),#18=(%7)) targets: 332=#33,%15=%17 INNER First: %19 Second: #33 on %7=%18,
  #33=TableRowSet #33 M TABLE (%18 INTEGER,#10 CHAR) Display=2
Indexes=[(#10)=[404],(%18)=[362]] key (%18) order (%18) targets: 332=#33 From: #20 Target=332
SRow:(339,381) Target:332 M,
  %0=Domain TABLE (#8,#10) Display=2[#8, CHAR],[#10, CHAR],
  %1=TableRowSet %1 VU TABLE (%2 CHAR,%3 INTEGER,%4 CHAR) Display=3 Indexes=[(%2)=[50]] key
(%2) targets: 23=%1 From: #20 Target=23 SRow:(30,67,89) Target:23 VU,
  %2=QlInstance %2  CHAR D From:%1 copy from 30,
  %3=QlInstance %3  INTEGER K From:%1 copy from 67,
  %4=QlInstance %4  CHAR U From:%1 copy from 89,
  %5=RestRowSet %5 WW TABLE (%7 INTEGER,%2 CHAR,%3 INTEGER,#8 CHAR) Display=4 targets: %15=%5
From: #20 Target=%15 SRow:()  RestView %15 RemoteCols:(%7,#8) RemoteNames:(#8=F,%7=E),
  %7=QlValue E %7  INTEGER,
  %14=Domain TABLE (%7,%2,%3,#8) Display=4[%7, INTEGER],[%2, CHAR],[%3, INTEGER],[#8, CHAR],
  %15=RestView TABLE (%7,%2,%3,#8) Display=4[%7, INTEGER],[%2, CHAR],[%3, INTEGER],[#8, CHAR]
ViewDef (E int, D char, K int, F char)  UsingTable: 23,
  %16=Domain TABLE (%7,%2,%3,#8) Display=4[%7, INTEGER],[%2, CHAR],[%3, INTEGER],[#8, CHAR],
  %17=RestRowSetUsing %17 WW (%7 INTEGER,%2 CHAR,%3 INTEGER,#8 CHAR) targets: %15=%17 From:
#20 Target=%15 SRow:(30,67,89) Template: %5 UsingTableRowSet:%1 UrlCol:%4,
  %18=QlInstance %18  INTEGER E From:#33 copy from 339,
```

```
  %19=OrderedRowSet %19 WW (%7 INTEGER,%2 CHAR,%3 INTEGER,#8 CHAR) key (%7) order (%7)
targets: %15=%17 Source: %17,
  %21=SelectStatement %21 Union=#1)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t
Returning ETag: "23,_,148"
--> 4 rows
Response ETag: 23,_,148
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,_,127"
--> 3 rows
Response ETag: 23,_,127
```

```
QL> select f,n from ww natural join m
|------|------|
|F     |N     |
|------|------|
|Tri   |Trois |
|Vier  |Quatre|
|Four  |Quatre|
|Five  |Cinq  |
|Six   |Six   |
|Sechs |Six   |
|------|------|
QL> |
```

Updatable RESTView

update ww set f='Eight' where e=8

Place the breakpoint in UpdateSearch.Obey() (Executable.cs, approx. line 4080) to see the context objects:

```
{(..,
  #1=UpdateSearch #1 Target: %16,
  #17=Eight,
  #32=SqlValueExpr #32  BOOLEAN From:_ Left:%6 Right:#33 #32(%6=#33),
  #33=8,
  %0=TableRowSet %0 VU TABLE (%1 CHAR,%2 INTEGER,%3 CHAR) Display=3 Indexes=[(%1)=[50]] key
(%1) targets: 23=%0 From: %0 Target=23 SRow:(30,67,89) Target:23 VU,
  %1=QlInstance %1  CHAR D From:%0 copy from 30,
  %2=QlInstance %2  INTEGER K From:%0 copy from 67,
  %3=QlInstance %3  CHAR U From:%0 copy from 89,
  %4=RestRowSet %4 WW TABLE (%6 INTEGER,%1 CHAR,%2 INTEGER,%12 CHAR) Display=4 matches (%6=8)
targets: %14=%4 Assigs:(UpdateAssignment Vbl: %12 Val: #17=True) Target=%14 SRow:()  RestView
%14 RemoteCols:(%6,%12) RemoteNames:(%6=E,%12=F),
  %6=QlValue E %6  INTEGER,
  %12=QlValue F %12  CHAR,
  %13=Domain TABLE (%6,%1,%2,%12) Display=4[%6, INTEGER],[%1, CHAR],[%2, INTEGER],[%12, CHAR],
  %14=RestView TABLE (%6,%1,%2,%12) Display=4[%6, INTEGER],[%1, CHAR],[%2, INTEGER],[%12,
CHAR] ViewDef (E int, D char, K int, F char)  UsingTable: 23,
  %15=Domain TABLE (%6,%1,%2,%12) Display=4[%6, INTEGER],[%1, CHAR],[%2, INTEGER],[%12, CHAR],
  %16=RestRowSetUsing %16 WW (%6 INTEGER,%1 CHAR,%2 INTEGER,%12 CHAR) matches (%6=8) targets:
%14=%16 Assigs:(UpdateAssignment Vbl: %12 Val: #17=True) Target=%14 SRow:(30,67,89) Template:
```

```
%4 UsingTableRowSet:%0 UrlCol:%3)}
```

```
http://localhost:8180/DB/DB select E,'B' as D,4 as K,F from t where E=8
HTTP POST /DB/DB
select E,'B' as D,4 as K,F from t where E=8
Returning ETag:
--> 0 rows
http://localhost:8180/DC/DC select E,'C' as D,1 as K,F from u where E=8
HTTP POST /DC/DC
select E,'C' as D,1 as K,F from u where E=8
Returning ETag: "23,127,127"
--> 1 rows
Response ETag: 23,127,127
RoundTrip HEAD http://localhost:8180/DC/DC/u/E=8
HTTP HEAD /DC/DC/u
Returning ETag:
--> OK
RoundTrip POST http://localhost:8180/DC/DC update u set F='Eight' where E=8

HTTP POST /DC/DC
update u set F='Eight' where E=8

Returning ETag: "23,127,244"
--> OK
```

```
QL> update ww set f='Eight' where e=8
QL>
```

No changes have been made to 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 C# library are all subclasses of Versioned (see section 6.4:REST and POCO of the Pyrrho manual). 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 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 t64 (all the tables created with the ENTITY flag). Replace the machine\user string with the user string returned by Pyrrho's "select user" statement.

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

```
create table "Customer" (id int primary key,"NAME" char, unique("NAME")) entity
insert into "Customer" values (10,'John'),(11,'Fred'),(12,'Mary')
[create table "Order" (id int primary key,cust int references "Customer", "OrderDate"
date,"Total" numeric(6,2)) entity]
[insert into "Order" values (1230,10,date'2022-05-10',34.56),(1231,12,date'2022-05-
11',67.89),(1234,11,date'2022-06-04',56.78)]
create table "Item" (id int primary key,"NAME" char, price numeric(6,2)) entity
[insert into "Item" values(71,'Pins',0.78),(72,'Pump',67.0),(73,'Crisps',0.89),
(74,'Rug',56.78),(75,'Bag',33)]
[create table "OrderItem" (it int,oid int references "Order" on delete cascade,
  item int references "Item",qty int,primary key(oid,it)) entity]
[insert into "OrderItem" values
(100,1230,75,1),(101,1230,71,2),(102,1231,73,1),(103,1231,72,1),(103,1234,74,1)]
```

Note that NAME is enclosed in straight double quotes because NAME is a reserved word in SQL
(there are some contexts in which this does not matter). The database contents at this point are:

Then

```
set role "Sales"
table "Role$Class"
```

generates code fragments for each of the tables created above that can be pasted into a C# program. The
fragments can be extracted from the PyrrhoCmd standard output. For example,

```
using Pyrrho;
using Pyrrho.Common;

/// <summary>
/// Class Customer from Database v0, Role Sales
/// PrimaryKey(ID)
/// Unique(NAME)
/// </summary>
[Table(97,194)]
public class Customer : Versioned {
  [Field(PyrrhoDbType.Integer)]
  [AutoKey]
```

```
  public Int64? ID;
  [Field(PyrrhoDbType.String)]
  public String? NAME;
  public Order[]? orders => conn?.FindWith<Order>(("CUST",ID));
}
```

As described in the Pyrrho manual (sec 6.4) this code includes attributes for the fields corresponding to columns in the Customer table. Since NAME is a unique key there is a static method for obtaining a single Customer object given the name; but as is indicated this method simply calls `conn.FindOne<..>` as shown. Such classes also have navigation properties coming from foreign keys, as will be seen below.

The ENTITY metadata flag ensures that all of these classes are Versioned. Identifiers and class/table names are case sensitive (this is C#), and the navigation property names are auto-constructed (so that the property orders has an awkward spelling). FindWith<V> takes a list of (key,value) pairs and returns an ordinary array of V. The Versioned library does not require the use of Pyrrho's list and tree classes: we use only [] in this demo.

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

These classes have all public fields. This is so that, for example, you can modify fields of an object a before calling conn.Put(a), while the library can update the version and any other fields that the database has updated by triggers or other users. Attributes such as [Table(.., ..)] provide internal information for the server, and these values are always easy to identify: for example, in TableAttribute, this information is a pair (tabledefpos, lastschemachange) giving the defining position of the table and the position of the last change made to its schema.

The classes we get can be placed in the main class of a demo program Program.cs as indicated by the highlight below:

```
using Pyrrho;

namespace Poco
{
    public class Program
    {
        // For each class in the Role$Class output,
        // PASTE all the code here apart from the using Pyrrho lines

        static void Main()
        {
            conn = new PyrrhoConnect("Files=t64;Role=Sales");
            conn.Open();
            try
            {
                // Get a list of all orders showing the customer name
                var aa = conn.FindAll<Order>();
                foreach (var a in aa)
                    Console.WriteLine(a.ID + ": " + a.customer?.NAME ?? "");
                if (aa.Length == 0)
                {
                    Console.WriteLine("The Order table is empty");
                    goto skip;
                }
                 // change the customer name of the first (update to a navigation property)
                 if (aa[0].customer is Customer j)
                 {
                     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)
                var m = conn.FindOne<Customer>(("NAME", "Mary"));
                var o = new Order()
                {
                    CUST = m.ID,
                    OrderDate = new Date(DateTime.Now)
                };
                conn.Post(o);
                // lookup a couple of items to add
                var p = conn.FindWith<Item>(("NAME", "Pins"))[0];
```
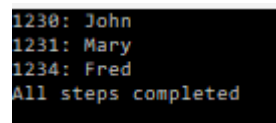
```csharp
                    var i1 = new OrderItem() { OID = o.ID, ITEM = p.ID, QTY = 2 };
                    conn.Post(i1);
                    var b = conn.FindWith<Item>(("NAME", "Bag"))[0];
                    var i2 = new OrderItem() { OID = o.ID, ITEM = b.ID, QTY = 1 };
                    conn.Post(i2);
                    // calculate the total for the new order (M indicates a decimal constant in C#)
                    decimal? t = 0.0m;
                    if (o.orderItems != null)
                        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>(("NAME", "Fred"));
                    var fo = f.orders;
                    if (fo is not null && fo[fo.Length - 1] is Order od)
                        conn.Delete(od);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
                Console.WriteLine("All steps completed");
            skip: Console.ReadLine();
                conn.Close();
            }
        }
}
```

```
1230: John
1231: Mary
1234: Fred
All steps completed
```

After executing this program, the database contains:

```
E:\PyrrhoDB70\Pyrrho>pyrrhocmd t64
QL> set role "Sales"
QL> table "Customer"
|--|------|
|ID|NAME  |
|--|------|
|10|Johnny|
|11|Fred  |
|12|Mary  |
|13|Greta |
|--|------|
QL> table "Order"
|----|----|----------|------|
|ID  |CUST|OrderDate |Total |
|----|----|----------|------|
|1230|10  |10/05/2022|34.56 |
|1231|12  |11/05/2022|67.89 |
|1235|12  |19/10/2024|34.56 |
|----|----|----------|------|
QL> table "Item"
|--|------|------|
|ID|NAME  |PRICE |
|--|------|------|
|71|Pins  |0.78  |
|72|Pump  |67.00 |
|73|Crisps|0.89  |
|74|Rug   |56.78 |
|75|Bag   |33.00 |
|--|------|------|
QL> Table "OrderItem"
|----|----|----|---|
|IT  |OID |ITEM|QTY|
|----|----|----|---|
|100 |1230|75  |1  |
|101 |1230|71  |2  |
|102 |1231|73  |1  |
|103 |1231|72  |1  |
|1   |1235|71  |2  |
|2   |1235|75  |1  |
|----|----|----|---|
QL> |
```

If the database has triggers these are called on Post, Put and Delete. Side effects update **this** (the Versioned object on which the method is called) but not other client objects. There is a method e.Get() that overwrites the fields of the current object with those of the latest version of the entity (if you want to retain both versions in the client, use conn.Get<C>(e)[0] to get a possibly different version of e)

If there is an explicit transaction in progress, these methods prepare the changes in the transaction, waiting for Commit or Rollback. Deferred triggers will be called on Commit.

Experiment with the above demo program by adding extra Console.Writeline statements before trying other changes.

6.12 Typed Graph Implementation

In this section we review some parts of test25. The important point in the implementation of typed graphs in Pyrrho is that graph data can be entered and modified using either relational SQL or the CREATE and MATCH statements, and Pyrrho maintains all of the base tables and indexes to make this work.

This section should be read in conjunction with the notes on SubTypes in section 3.2.5. As we will see, the CRUD view of graph data always delivers, and appears to operate on, rows that match the current table type including data from its supertype. Supertypes are base tables in their own right: the actual columns of a base table are those from the type itself, excluding those from supertypes. For example, an SQLInsert statement modifies the table, its supertypes, and their indexes. Despite this, there is only one Record object for each row inserted in the transaction log, and all these tables share the single TableRow object that is built in the TransitionRowSet (see section 6.2.1).

The implementation has two faces corresponding to the physical and logical layers[63] of the database. The physical layer (Pyrrho's Level 2) can be seen by examining the transaction log, while the logical layer (Pyrrho's Level 3 and above) can be viewed by examining cx.db.objects[].

At Level 3 the names of database objects depend on the current role, and graph nodes are simple rows belonging to a named node or edge type according to the model developed by that role. Node and edge types form a hierarchy, using the UNDER property of SQL user defined types, and keep track of their subtypes and supertypes. While the modeler can choose names for node and edge types and their properties on a per-role basis, there are some obvious restrictions. Node and edge type names must be unique in the role, and property (column) names of a node or edge type must be unique (in the role) in the type hierarchy containing the type. Pyrrho has a base table corresponding to each node or edge type, but the columns of these base tables are just the properties declared for the type (and exclude those of the supertype(s) if any). But a row (TableRow) of any node or edge type includes the fields inherited from supertypes and will also be found among the rows of its supertypes (no copying: the TableRow is immutable and actually shared with all of them).

Every node (TNode) has a unique integer identifier we refer to in these notes as **id** (whatever its column name is in the current role[64]) and this can be modified using an SQL update statement: its value is independent of the role. The TNode also has a uid property that identifies the database TableRow discussed above. TNode is a subclass of TRow and its fields are the fields of this TableRow.

Every edge (TEdge) has a leaving property and an arriving property whose values reference actual nodes: the database maintains indexes to the actual TNodes. These properties are initially called LEAVING and ARRIVING but can be modified for the model. TEdge is a subclass of TNode, so edges have the identifiers and properties discussed above.

A graph (TGraph) is a collection of TNodes. A database TNode, in principle, identifies a graph consisting of all nodes that be reached by following edges. The database maintains a set of such connected TGraphs that cover the database. Any such connected TGraph can be represented by its base node (the oldest, which has the lowest uid). It follows that TGraphs in the database are not modified directly.

TGParam is a class of TypedValues used only in MatchStatements, which has a lexical uid from its first occurrence in the statement, a type indicating its use in the MatchStatement (flags for node, edge, path, group, nullable) and its name (a string). It is initially unbound, and once bound its value is found in the Context's list of bindings. TGParams can occur as the id of a node or edge, as the label part, or in SqlLiterals.

A MatchStatement is a collection of TGParams, a list of SqlNodes, and possibly where clause and a procedural body. An SqlNode keeps track of the TGParams they introduce, and QlValues for id, type, and properties (a list of QlValue key/value pairs: each key is a list of QlValues that evaluate to strings). SqlNode has a subclass SqlEdge, which also has QlValues for its leaving and arriving SqlNode, SqlEdge has a sbclass SqlPath, which has a pattern (a list of SqlNodes) and a pair of integers called the MatchQuantifier in the syntax for MatchStatement. All these evaluate to the binding of their id (a TNode or TEdge).

When a MatchStatement is executed, the Context may already contain bindings from an enclosing Match in progress, but its own bindings will be unbound. The MatchStatement examines the first node in its list of SqlNodes and creates a continuation consisting of the rest of its list. On examination of this first SqlNode, a set of database nodes of its required type and properties is constructed; on examination of an SqlPath, the continuation is the pattern. For each element of this set, the SqlNode's TGParams are bound, and traversal continues to the next node in the continuation. . If there is no next node in the continuation, all SqlNodes in the MatchStatement have been bound to database rows (TNodes): subject to the where clause if any, the body of the match statement is obeyed, or if there is no body, a row of bindings is added to the result of the MatchStatement. When all elements at the current stage have been examined, the TGParams in the SqlNode's list are removed from the list of bindings. When the recursion is complete, the MatchStatement's side-effects have occurred and/or the rowset result has been built.

We trace through this process in examples in section 6.12.3 below.

---

[63] This distinction is for clarity in these notes. Most textbooks on database design view everything in the database server as part of the physical layer.

[64] The name of the id (resp. leaving, arriving) property is initially set to ID (resp. LEAVING, ARRIVING) but can be modified by metadata on creation and subsequently on a per role basis using the SQL alter column machinery.

6.12.1 Using SQL to build and manage node and edge types

In section 6.12.3 we will introduce the CREATE statement, which is the easiest way to build a complex graph. But first we use a simple example to explore the relationship with base tables in the database.

Starting with an empty database, the test declares two node types and a node in SQL. A type can be created as a node or edge type using SQL metadata, or by naming a supertype that is a node or edge type. Node and edge types have special properties whose default names are ID, LEAVING and ARRIVING as we will see.

create type student as (name char, matric char) nodetype

At this point the log shows the following database objects:

23	PNodeType STUDENT NODETYPE
49	PColumn3 NAME for 23(1)[CHAR]
73	PColumn3 MATRIC for 23(1)[CHAR]
99	PMetadata STUDENT{Id:NODETYPE}

We see that the metadata annotation NODETYPE has modified the definition of a user-defined type STUDENT into a NodeType STUDENT with two columns NAME and MATRIC as specified, and a mysterious Id entry (for each instance of STUDENT this will be a reference to the defining position of the corresponding database record). In this and other ways the graph implementation dynamically modifies nodeTypes, so that it is genrally more useful to examine the actual database objects in cx.db rather than the log. A good way to do this is to place a break in Start.cs at the usual place (line 436) to break execution with something harmless like select 67 and then look at this.db.objects.root.gtr.. (there are over 500 system objects with negative uids). At this point, drilling down in the Locals window to db.objects > root > gtr > gtr > gtr > slots, we see

```
    Name Value
▶ [4] {[23, {23 STUDENT NODETYPE (49,73)[49,Domain CHAR],[73,Domain CHAR] rows 0}]}
▶ [5] {[49, {TableColumn 49 Definer=-502 LastChange=49 Domain CHAR Table=23}]}
▶ [6] {[73, {TableColumn 73 Definer=-502 LastChange=73 Domain CHAR Table=23}]}
```

The names and positions of objects (other than types) are role-specific, and are found in the role's entry in ob.infos[]. Here cx.obs[23].infos[-502] is {ObInfo STUDENT Privilege=8388607 NODETYPE} and its names property contains {(MATRIC=73, NAME=49)}.

Nodes of this type can be defined by inserting rows into the underlying table. We can use SQL for this (for the GQL way, see the next section):

insert into student values ('Fred','22/456')

```
QL> insert into student values ('Fred','22/456')
1 records affected in t25
QL> table student
|-----|------|
|NAME|MATRIC|
|-----|------|
|Fred|22/456|
|-----|------|
QL> |
```

Now the log shows the additional entry:

151	Record 151[23]: 49=Fred[CHAR],73=22/456[CHAR]

And examining memoy (with select 67 as above) within cx.db.objects[23] we see

{23 STUDENT NODETYPE (49,73)[49,Domain CHAR],[73,Domain CHAR] rows 1}

23.tableRows[151].vals = {(49=Fred,73=22/456)}

This implementation of GQL supports subtypes, using an SQL-compatible implementation of subtypes. If we decide that we want STUDENT to be a subtype of a new node type PERSON, we could use SQL at this point to create a node type PERSON with the NAME property, and set STUDENT to be a subtype of PERSON. (We could have started with these steps.)

create type person as (name string) nodetype

The new entries in the log are:

197	PNodeType PERSON NODETYPE
223	PColumn3 NAME for 197(-1)[CHAR]

249	PMetadata PERSON{Id:NODETYPE}

And in db.objects we now have additional entries

```
{[197, {197 PERSON NODETYPE (223)[223,Domain CHAR] rows 0}]}
{[223, {TableColumn 223 Definer=-502 LastChange=223 Domain CHAR Table=197}]}
197.tableRows {}
```

If we had started with this step we could simply **create type student under person as (matric string).** But we have already defined the STUDENT nodetype, so instead we should *alter* it to be a subtype of the new PERSON type (using an SQL ALTER TYPE syntax extension):

alter type student set under person

The new log entry is just

302	EditType STUDENT[23] Under: [PERSON]

But several changes have been made to the in-memory structures. The user-defined type STUDENT becomes a subtype of PERSON and the two NAME columns are merged (in a cascade), so that column 49 NAME has already been dropped[65]. Examining cx.db.objects we see changes:

```
{[23, {23 STUDENT NODETYPE (223,73)[223,Domain CHAR][73,Domain CHAR] rows 1 Under=197}]}
{[73, {TableColumn 73 Definer=-502 LastChange=73 Domain CHAR Table=23}]}
{[197, {197 PERSON NODETYPE (223)[223,Domain CHAR] rows 1 Subtypes [23]}]}
{[223, {TableColumn 223 Definer=-502 LastChange=223 Domain CHAR Table=197}]}
23.tableRows[151].vals {(73=22/456,223=Fred)}
197.tableRows[151].vals {(73=22/456,223=Fred)}
```

Fred is now represented by a row in each table (STUDENT, and its supertype PERSON). Although the STUDENT table appears unchanged, we see that the record about Fred shows has NAME in the *supertype's* column 223, not the old STUDENT column 49. Selection from a type contributes columns from supertypes, and insertion in a type also adds a row to each of its supertypes if any. The same row object serves for all of them: here the tableRows shown are the same memory object, accessible via either table.
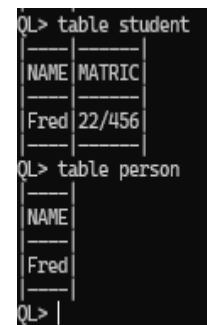
In the next step of the test, we create a new subtype of PERSON.

create type staff under person as (title char)

Since the nominated supertype PERSON is a node type, we know that STAFF must also be a NodeType:

348	PNodeType STAFF NODETYPE Under: 197
373	PColumn3 TITLE for 348(1)[CHAR]

```
{[23, {23 STUDENT NODETYPE (223,73)[223,Domain CHAR][73,Domain CHAR] rows 1 Under=197}]}
{[73, {TableColumn 73 Definer=-502 LastChange=73  CHAR Table=23}]}
{[197, {197 PERSON NODETYPE (223)[223,Domain CHAR] rows 1 Subtypes [23,348]}]}
{[223, {TableColumn 223 Definer=-502 LastChange=223  CHAR Table=197}]}
{[348, {348 STAFF NODETYPE (223,373)[223,Domain CHAR],[373,Domain CHAR] rows 0 Under=197}]}
{[373, {TableColumn 373 Definer=-502 LastChange=373 Domain CHAR Table=348}]}
```

We can see that records for STAFF will be entered in both PERSON and STAFF.

insert into staff values ('Anne','Prof')

We check the Log (there is only one record event, and it has added a tableRow each to tables STAFF and PERSON:

418	Record 418[348]: 223=Anne[CHAR],373=Prof[CHAR]

The SPECIFICTYPE() method described in the SQL standard has been implemented in Pyrrho as can be seen in the illustration. We also see the POSITION pseudocolumn, which Pyrrho uses as a default index.

---

[65] The log records the physical objects committed to the database, and the consequences of cascades are not directly visible. The system tables in the Role$ collection can be used to view the current state database objects.

The next part of test 25 exercises the edge concept, still using SQL-style statements.

In the test, both node types are just PERSON, and the leaving and arriving types are specified in a metadata syntax:

create type married edgetype(bride=person,groom=person)

465	PEdgeType MARRIED EDGETYPE(197,197)
498	PColumn3 BRIDE for 465(0)[POSITION] LeaveCol 197
527	PIndex BRIDE on 465(498) ForeignKey, CascadeUpdate
550	PColumn3 GROOM for 465(1)[POSITION] ArriveCol 197
580	PIndex GROOM on 465(550) ForeignKey, CascadeUpdate

{465 MARRIED EDGETYPE (498,550)[498,Domain POSITION],[550,Domain POSITION]   rows   0   Indexes:((498)527;(550)580)   KeyCols: (498=True,550=True) Leaving 197[527] LeaveCol=498 Arriving 197[580] ArriveCol=550}

All of the usual SQL machinery is available. The test enters two nodes and one edge using SQL:

```
insert into person values('Joe'),('Mary')
[insert into married values((select position from
person where name='Joe'),(select position from
person where name='Mary'))]
```

The edge is just

465.tableRows[749].vals = {(498=694,550=712)}

We introduce the GQL way to add graph data in the next section.

6.12.2 Using CREATE and MATCH to enter graph data

It is much easier to construct graph data incrementally using CREATE statements consisting of graph fragments, similarly to Neo4j. Nodes are indicated using parentheses (), and edges using arrows -[ ]-> or <-[ ]- . The content of a node is a unique identifier (unquoted, or with double quotes) which on first occurrence must be followed by a colon and a type label (and possibly a further colon and type label to indicate a subtype, and optionally further properties using JSON notation. Edges are similar, except that the identifier can be left blank (an all-number identifier will be supplied by the system). Further nodes can be added. The system infers the node and edge types and their columns.

Here is an example in test25:

```
[CREATE
(:Product:WoodScrew {spec:'16/8x4'}),(:Product: WallPlug{spec:'18cm'}),
(Joe:Customer {Name:'Joe Edwards', Address:'10 Station Rd.'}),
(Joe)-[:Ordered {"Date":date'2002-11-22'} ]->(:"Order"{id:201})]
```

We have created node types PRODUCT with some subtypes, CUSTOMER, Order, and Ordered, and the records containing the above information. Note that GQL (and SQL) reserved word rules require double quotes around Order and Date. For more details about the syntax see the Pyrrho manual Pyrrho.doc and section 1.18 of this document.

At this point the log contains:

23	PNodeType PRODUCT NODETYPE
49	PColumn3 SPEC for 23(0)[CHAR]
72	PNodeType WOODSCREW NODETYPE Under: [23]
99	PNodeType WALLPLUG NODETYPE Under: [23]
125	PNodeType CUSTOMER NODETYPE
152	PColumn3 ADDRESS for 125(0)[CHAR]
178	PColumn3 NAME for 125(1)[CHAR]
202	PNodeType Order NODETYPE
226	PColumn3 ID for 202(0)[INTEGER]
249	PEdgeType ORDERED EDGETYPE(125,202)
280	PColumn3 Date for 202(-1)[DATE]
304	PColumn3 LEAVING for 249(1)[POSITION] LeaveCol 125
334	PIndex LEAVING on 249(304) ForeignKey, CascadeUpdate
358	PColumn3 ARRIVING for 249(2)[POSITION] ArriveCol 202
390	PIndex ARRIVING on 249(358) ForeignKey, CascadeUpdate
415	Record 415[72]: 49=16/8x4[CHAR]
433	Record 433[99]: 49=18cm[CHAR]
449	Record 449[125]: 152=10 Station Rd.[CHAR],178=Joe Edwards[CHAR]
493	Record 493[202]: 226=201[INTEGER]
508	Record 508[249]: 280=22/11/2002 00:00:00[DATE], 304=449[INTEGER],358=201[INTEGER]

In this test the information about subTypes is important. The PRODUCT node type has subtypes WOODSCREW and WALLPLUG, as a result of loading the PNodeTypes 72 and 99.

To complete the information about the order, we will add the details of Order 201 in the next step below. We will use the MATCH statement for convenience and as an introduction to this operation:

```
[MATCH (O:"Order"{id:201})
begin MATCH(P:Product{spec:'16/8x4'}) CREATE (O)-[:Item {Qty: 5}]->(P);
    MATCH(P:Product{spec:'18cm'}) CREATE (O)-[:Item {Qty: 3}]->(P) end]
```

The first MATCH looks up the order we want to attach information to, and the others look up products. The second Match has bound P to a WoodScrew, while the second binds P to a WallPlug. Both CREATE statements add an ITEM edge containing the quantity ordered.

Following execution of this Match/Create combination, the transaction log contains numerous schema changes and two new records:

567	PEdgeType ITEM EDGETYPE(202,23)
596	PColumn3 QTY for 567(0)[INTEGER]
621	PColumn3 LEAVING for 567(1)[POSITION] LeaveCol 202
653	PIndex LEAVING on 567(621) ForeignKey, CascadeUpdate
678	PColumn3 ARRIVING for 567(2)[POSITION] ArriveCol 23
710	PIndex ARRIVING on 567(678) ForeignKey, CascadeUpdate
736	Record 736[567]: 596=5[INTEGER],621=201[INTEGER],678=415[INTEGER]
771	Record 771[567]: 596=3[INTEGER],621=201[INTEGER],678=433[INTEGER]

6.12.3 Using MATCH to examine and modify graph data

We have seen a simple use of MATCH in the last section, where the bound identifier evaluated to a node: such a simple match on its own can be used to examine the contents of selected nodes and edges. Obviously if the Match selects several nodes, the result is a cross join, and a sequence of MatchStatements successively binds identifiers as we have seen; the same effect can be achieved using a where clause.

In some ways, unbound identifiers are like the aliases used in ordinary SQL, since they are given a meaning inside the current statement and then forgotten after it is executed. But there is an important difference: by its nature the MatchStatement looks to match unbound identifiers with database objects, so that if you want a new unbound identifier you need to avoid the names of existing database objects (tables, types, procedures, or columns, fields or methods of objects referenced in the current statement), or currently bound identifiers or aliases. These observations also apply to the use of unbound identifiers in the CreateStatement, which also can have a dependent executable statement. In this section we examine more sophisticated opportunities provided by MATCH. At present, these go well beyond the current GQL specification (ISO 39075).

Like the CreateStatement, the Match statement consists of a set of graph fragments, but like a SELECT statement as it builds a rowset whose columns are unbound identifiers in the graph syntax[66]. Such identifiers can occur anywhere in the graph syntax, and as its name implies, the MATCH statement finds all possible values of these such that the graph fragments are found in the database.

```
match ()-[:Item {Qty:A}]->(:T{spec:X}) where A>4
```

```
SQL> match ()-[:Item {Qty:A}]->(:T{spec:X}) where A>4
|---------|-|------|
|T        |A|X     |
|---------|-|------|
|WOODSCREW|5|16/8x4|
|---------|-|------|
SQL> |
```

Taking this example and modifying it a little to reduce the number of steps in the match process, let us trace through the execution of

```
          1         2         3         4         5
12345678901234567890123456789012345678901234567890
match (:"Order")-[:Item where Qty>4]->(:T{spec:X})
```

Place a breakpoint at the start of the MatchStatement.Obey() method, at line 4540 of Executable.cs. At this point the Context contains:

```
{(..
  #8=GqlNode COLON #8 202 Order NODETYPE (226)[226,Domain INTEGER] rows 1 IdCol=226 202 Order,#9
Order,
  #19=GqlEdge COLON #19 567 ITEM EDGETYPE (596,621,678)[596, INTEGER],[621, POSITION],[678,
POSITION] rows 2 Indexes:((621)653;(678)710) KeyCols: (621=True,678=True) Leaving 202[653]
LeaveCol=621 Arriving 23[710] ArriveCol=678 where [#34] 567 ITEM,#20 ITEM leaving #8 ARROWBASE,
  #31=QlInstance #31  INTEGER QTY From:#34 copy from 596,
  #34=SqlValueExpr #34  BOOLEAN From:#34 Left:#31 Right:#35 #34(#31>#35),
  #35=4,
  #40=GqlNode COLON #40 -526  NODETYPE rows 0:GqlLabel T NODETYPE {SPEC=SqlField #48  CHAR X
From:_ Target=#40} #41 T,#48 X,
  #41=GqlLabel T NODETYPE,
  #43=QlValue SPEC #43  CHAR,
  #48=SqlField #48  CHAR X From:_ Target=#40,
  %0=GqlMatchAlt %0  Null  [#8,#19,#40],
  %1=GqlMatch %1  Null  [%0],
  %2=BindingRowSet %2 (#41 CHAR,#48 CHAR),
  %3=MatchStatement %3 GDefs ((202=#9 Order,567=#20 ITEM,#9=#9 Order,#20=#20 ITEM,#41=#41
T,#48=#48 X)) Graphs (%1) Bindings %2)}
```

The MatchStatement is %3: we see it defines bindings %2 (T,X) and has a single graph %1 whose nodes are the GqlNode #8, the GqlEdge #19, and the GqlNode #40.

The two main methods in the implementation of MatchStatement are: ExpNode at line 4919,

```
void ExpNode(Context cx, ExpStep be, Sqlx tok, TNode? pd)
```

which is passed the current position in the Match Expression in a *continuation* be (as we will see below) and some context tok and pd, and DbNode at line 5120,

```
void DbNode(Context cx,NodeStep bn, Sqlx tok, TNode? pd)
```

---

[66] Unlike SELECT, it ignores duplicate bindings.

which is passed a list of relevant nodes in its continuation bn. In both cases the continuation specifies what the algorithm does if the algorithm is able to move to the next GqlMatch node. If all the nodes have matched and there no further nodes, the AddRow method will be called by the EndStep of the continuation to add a row of bindings to the result in an ExplicitRowSet containing the result of the MatchStatement (the working table).

Place breakpoints at lines 4892 (AddRow), 4937 (ExpNode,xn), 5167 (DbNode,dn), 5197 (another), and 5206 (backtrack), add Watch entries in the debugger for step, be, xn, bn, dn, and cx.binding, and collect snapshots from the Prolog-like match-backtrack process:

ExpNode #8: The continuation be specifies future steps. The current graph expression has three nodes and no AltGraph.

	step	0x00000001
▷	be	{Exp[#8,#19,#40],Graph[],End[%3]}
▷	xn	{GqlNode COLON #8 202 Order ...
✕	bn	error CS0103: The name 'bn' d... ↻
✕	dn	error CS0103: The name 'dn' d... ↻
▷	cx.binding	{}

DbNode 493 (see the log entry above). Here the continuation is bn, and shows only one possible node to match.

	step	0x00000002
▷	be	{Exp[#8,#19,#40],Graph[],End[... ↻
▷	xn	{GqlNode COLON #8 202 Ord... ↻
▷	bn	{Node#8:[493],Exp[#19,#40],Grap..
▷	dn	{TNode 493[202]}
▷	cx.binding	{}

ExpNode #19:

	step	0x00000003
▷	be	{Exp[#19,#40],Graph[],End[%3]}
▷	xn	{GqlEdge COLON #19 567 ITEM E...
▷	bn	{Node#8:[493],Exp[#19,#40],G... ↻
▷	dn	{TNode 493[202]} ↻
▷	cx.binding	{(_=0)}

DbNode 736: There are two nodes to test here (the second is tested at step 7).

	step	0x00000004
▷	be	{Exp[#19,#40],Graph[],End[%3]} ↻
▷	xn	{GqlEdge COLON #19 567 ITE... ↻
▷	bn	{Node#19:[736,771],Exp[#40],Gra...
▷	dn	{TEdge 736[567]}
▷	cx.binding	{(_=0)}

ExpNode #40:

	step	0x00000005
▷	be	{Exp[#40],Graph[],End[%3]}
▷	xn	{GqlNode COLON #40 -526 NOD.
▷	bn	{Node#19:[736,771],Exp[#40],... ↻
▷	dn	{TEdge 736[567]} ↻
▷	cx.binding	{(_=(0=TNode 493[202]))}

DbNode 415:

⬥ step	0x00000006	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#40:[415],Exp[],Graph[],En...	
▷ ⬥ dn	{TNode 415[23]}	
▷ ⬥ cx.binding	{(_=(0=TNode 493[202]))}	

AddRow (everything has matched):

⬥ step	0x00000006	↻
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#40:[415],Exp[],Graph[],...	↻
▷ ⬥ dn	{TNode 415[23]}	↻
▷ ⬥ cx.binding	{(_=(0=TNode 493[202],1=TEdge .	

Backtrack (to see if there are other matches):

⬥ step	0x00000006	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#40:[415],Exp[],Graph[],En...	
▷ ⬥ dn	{TNode 415[23]}	
▷ ⬥ cx.binding	{(_=(0=TNode 493[202],1=TEdge ...	

Backtrack:

⬥ step	0x00000004	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#19:[736,771],Exp[#40],Gra..	
▷ ⬥ dn	{TEdge 736[567]}	
▷ ⬥ cx.binding	{(_=(0=TNode 493[202]))}	

DbNode 771:

⬥ step	0x00000007	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#19:[736,771],Exp[#40],Gra...	
▷ ⬥ dn	{TEdge 771[567]}	
▷ ⬥ cx.binding	{(_=())}	

Another (771 does not match, see if there are other nodes to consider):

⬥ step	0x00000007	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#19:[736,771],Exp[#40],Gra...	
▷ ⬥ dn	{TEdge 771[567]}	
▷ ⬥ cx.binding	{(=(0=TNode 493[202]))}	

Backtrack:

⬥ step	0x00000007	
▷ ⬥ be	{Exp[#40],Graph[],End[%3]}	↻
▷ ⬥ xn	{GqlNode COLON #40 -526 N...	↻
▷ ⬥ bn	{Node#19:[736,771],Exp[#40],Gra...	
▷ ⬥ dn	{TEdge 771[567]}	
▷ ⬥ cx.binding	{(_=())}	

Backtrack:

```
        step          0x00000002
▷  be            {Exp[#40],Graph[],End[%3]}      ↻
▷  xn            {GqlNode COLON #40 -526  N... ↻
▷  bn            {Node#8:[493],Exp[#19,#40],Grap...
▷  dn            {TNode 493[202]}
▷  cx.binding    {(_=())}
```

```
QL> match (:"Order")-[:Item where Qty>4]->(:T{spec:X})
|----------|------|
|T         |X     |
|----------|------|
|WOODSCREW |16/8x4|
|----------|------|
QL>
```

This completes this worked example.

The next steps in Test25 show some useful ways of using the dependent statement of the MatchStatement to return information from the selection of nodes and to make updates to existing nodes. For simplicity let us start again with an empty database:

```
[CREATE (a:Person {name:'Fred Smith'})<-[:Child]-(b:Person {name:'Pete Smith'}),
(b)-[:Child]->(:Person {name:'Mary Smith'})]

MATCH (p)-[:Child]->(c) RETURN p.name,c.name AS child

MATCH (p {name:'Pete Smith'}) SET p.name='Peter Smith'
```

```
QL> [CREATE (a:Person {name:'Fred Smith'})<-[:Child]-(b:Person {name:'Pete Smith'}),
> (b)-[:Child]->(:Person {name:'Mary Smith'})]
5 records affected in t25a
QL> MATCH (p)-[:Child]->(c) RETURN p.name,c.name AS child
|-----------|----------|
|NAME       |CHILD     |
|-----------|----------|
|Pete Smith |Fred Smith|
|Pete Smith |Mary Smith|
|-----------|----------|
QL> MATCH (p {name:'Pete Smith'}) SET p.name='Peter Smith'
1 records affected in t25a
QL> table Person
|-----------|
|NAME       |
|-----------|
|Fred Smith |
|Peter Smith|
|Mary Smith |
|-----------|
QL>
```

Finally in this section, let us look at the next step in Test25, which uses a repeating pattern with MATCH, to look for the descendants of Peter Smith.

From the description above, we see that at each stage of the match process we have a GqlMatchExp that that refers to the list of GqlNodes we are traversing, a continuation (an instance of MatchStatement.Step), and other data from the expression (e.g. a token), one or more GqlNodes and a TNode. At each step, the process continues with Next, or backtracks to a previous step. When there are no more nodes to traverse, AddRow is called. There are several subclasses of Step:

ExpNode: Examine the current node in the GqlMatchExp and identify the possible TNodes it can match. Then (Next) call DbNode with this list, and a NodeStep continuation for the list of TNodes found, to be followed by the next ExpStep in the GqlMatchExp. Otherwise (Backtrack) start a GqlMatchAlt if available, or just return.

DbNode: Check that the TNode matches the current GqlNode: if it is an edge, we check for leaving and arriving nodes. If so, update the bindings, and call rhe (Next) of the continuation. Then unbind what we bound. Repeat for other nodes in the list supplied. Then (or on Backtrack) simply return.

PathFirstNode: We should have a previous TNode. Check it also matches the requirement for the first GqlNode in the path pattern, and if so, update the bindings and call PathNode for the next GqlNode (an edge). Then unbind what we bound, and return.

PathFollowingNode: We consider whether the path is completed: if not, the contuation will be a PathFirstNode for the current path. If it is completed we supply our last TNode to be identified with the next node in the calling GqlMatchExp, and call the continuation supplied to us.

```
            1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567890123456
MATCH ({name:'Peter Smith'}) [()-[:Child]->()]+ (x) RETURN x.name
```

The + is shorthand for {1,*} . The pattern must be overall like a complicated edge and therefore must start and end with a node, but these need not be empty: any requirements apply to the repeat and to the preceding and following nodes.

With the same break points as before and adding two more in PathNode (lines5258 and 5266), we get:

```
Step 1: {Exp[#8,%1,#50],Graph[],End[%8]}
  xn<- {GqlNode LBRACE #8 -526  NODETYPE rows 0:GqlLabel GqlLabel {NAME=Peter Smith}}
Step 2: {Node#8:[203,225,273],Exp[%1,#50],Graph[],End[%8]}
  dn<- {TNode 203[23]}
 another, backtrack
Step 3: {Node#8:[203,225,273],Exp[%1,#50],Graph[],End[%8]}
  dn<- {TNode 225[23]}
Step 4: {Exp[%1,#50],Graph[],End[%8]}
  xn<- {GqlPath %1  Null :GqlLabel GqlLabel leaving #8 arriving #50 Null[#32,#35,#45]{1,-1}}
Step 5: {Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xi<- {GqlNode RPAREN #32 23 PERSON ..}
Step 6: {Exp[#35,#45],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xn<- {GqlEdge COLON #35 71 CHILD ..}
Step 7: {Node#35:[247,295],Exp[#45],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<- {TEdge 247[71]}
Step 8: {Exp[#45],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xn<-{GqlNode RPAREN #45 -526  NODETYPE ..}
Step 9: {Node#45:[203],Exp[],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<-{TNode 203[23]}
Step 10: {Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xi<-{GqlNode RPAREN #32 23 PERSON ..}
Step 11: {Exp[#35,#45],Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xn<- {GqlEdge COLON #35 71 CHILD ..}
Step 12: {Node#35:[247],Exp[#45],Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<- {TEdge 247[71]}
 another, backtrack to 10, backtrack
Step 13: {Exp[#50],Graph[],End[%8]}
  xn<- {GqlNode X #50 -526  NODETYPE ..}
Step 14: {Node#50:[203],Exp[],Graph[],End[%8]}
  dn<- {TNode 203[23]}
AddRow67 {(_=(0=TNode 225[23],1=TEdge 247[71],2=TNode 203[23]),#50=TNode 203[23])}
  backtrack to 9, backtrack to 7, backtrack
Step 15: {Node#35:[247,295],Exp[#45],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<- {TEdge 295[71]}
Step 16: {Exp[#45],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xn<- {GqlNode RPAREN #45 -526  NODETYPE ..}
Step 17: {Node#45:[273],Exp[],Path0[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<- {TNode 273[23]}
Step 18: {Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xi<- {GqlNode RPAREN #32 23 PERSON ..}
Step 19: {Exp[#35,#45],Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  xn<- {GqlEdge COLON #35 71 CHILD ..}
Step 20: {Node#35:[295],Exp[#45],Path1[#32,#35,#45],Exp[#50],Graph[],End[%8]}
  dn<- {TEdge 295[71]}
 another, backtrack to 18
Step 21: {Exp[#50],Graph[],End[%8]}
  xn<- {GqlNode X #50 -526  NODETYPE ..}
Step 22: {Node#50:[273],Exp[],Graph[],End[%8]}
  dn<- {TNode 273[23]}
AddRow {(_=(0=TNode 225[23],1=TEdge 295[71],2=TNode 273[23]),#50=TNode 273[23])}
 backtrack to 17, backtrack to 15, backtrack to 5
Step 23: {Node%1:[247,295],Exp[#50],Graph[],End[%8]}
  dn<- {TEdge 247[71]}
 another, backtrack
Step 24: {Node%1:[247,295],Exp[#50],Graph[],End[%8]}
 dn<- {TEdge 295[71]}
 another, backtrack to 3
```

---

[67] AddRow includes the current state of the path, which in this example is not in the binding table.

```
Step 25: {Node#8:[203,225,273],Exp[%1,#50],Graph[],End[%8]}
 dn<- {TNode 273[23]}
 another, backtrack
```

```
QL> MATCH ({name:'Peter Smith'}) [()-[:Child]->()]+ (x) RETURN x.name
|----------|
|NAME      |
|----------|
|Fred Smith|
|Mary Smith|
|----------|
QL>
```

This completes execution of the MatchStatement.

This database contains only one generation of people: as an exercise, add some more person and child edges, and examine how the algorithm behaves for grandchildren and beyond. The trick is that the repeating path is implemented in the continuations.

References

Crowe, M. K.: The Pyrrho DBMS, https://github.com/MalcolmCrowe/ShareableDataStructures subfolder 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

GQL2024: ISO/IEC 39075:2024 Database Languages – GQL (International Standards Organization, 2024)

T. Krijnen, and G. L. T. Meertens, "Making B-Trees work for B". Amsterdam: Stichting Mathematisch Centrum, 1982, Technical Report IW 219

SQL2023: ISO/IEC 9075-2:2023 Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation); ISO/IEC 9075-4:2023: Information Technology – Database Languages – SQL – Part 4: Persistent Stored Modules; ISO/IEX 9075-16:2023 Property Graph Queries (SQL/PGQ) (International Standards Organization, 2016)