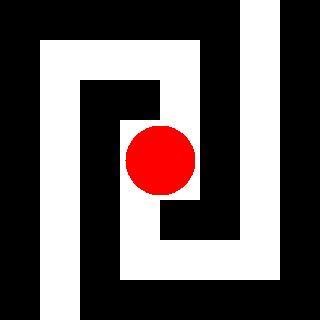
The Pyrrho Book

Malcolm Crowe, University of the West of Scotland

www.pyrrhodb.com



2nd edition

Paisley: University of the West of Scotland, 2021

# Chapter 1: Introduction to the Book

This book focuses on a number of aspects of database management systems that are important in real life application, but in which current products fall short of what is required. The book will make a contribution to DBMS design by suggesting a number of fundamental improvements to DBMS architecture, and validating them by means of a working proof-of-concept DBMS, Pyrrho, that includes not just the basic relational model, but most of the features of ISO-standard SQL and some other suggestions from the database community. We will discuss the trade-offs in speed and complexity involved in including these features: over the years the set of additional features offered by Pyrrho has changed, with some advanced features (RDF, SPARQL, JPA, client-side data models etc) being removed from Pyrrho where the added complexities have not justified themselves.

In this book, supporting data consistency is a prime concern. In particular, when working with a single database, not matter how many other users are accessing the same database, it should not be the application’s responsibility to maintain consistency or to clean up after errors. While databases should be fast and scalable, 100% accuracy is more important than speed. Discussion of alternatives to Pyrrho and to the relational model are included in this book where relevant: it is a misconception that the failings of database products are somehow the fault of the relational model or SQL.

This edition of the book has been updated to Pyrrho version 7, a major re-implementation of the DBMS based on shareable data structures. These immutable objects provide a radical way of supporting concurrent operations.

This book will take for granted the concerns of introductory texts, such as relational design, SQL, entity-relationship modelling, joins and triggers. One of these concerns, “normal form” data, is notable here in that it contributes to database correctness, since any duplication of data has a potential for inconsistency. It represents an efficiency trade-off since with normal form, the information for the domain needs to be reconstructed by recombining data from these extra tables or selecting columns from larger tables; and where normalisation is reversed for efficiency reasons, it is at the cost of repairing duplicates, for example with the help of triggers. These considerations are of interest, but they come before the starting point for this book: we assume good database design.

The most serious criticism of existing commercial products relates to ACID properties and implementation of transactions, where in many cases they don’t follow their own documentation. When this book is used in a university course I would strongly recommend including the Transactions tutorial from DBTechNET.org which provides a useful critique of a range of commercial DBMS from this viewpoint, and supplies a set of interesting exercises to try out on any other supposedly ACID compliant DBMS. But rather than merely criticise these products, this book offers a set of design proposals for solving the problems of consistency, embodied in the open-source Pyrrho DBMS. In order to show the practicality of these design proposals for advanced database scenarios, this book goes beyond simplified examples to showcase Pyrrho’s implementation of strongly typed features of standard SQL. The Pyrrho DBMS enforces transaction isolation (to the level of conflict serializability) and all defined constraints in all circumstances, and supports role-based security and mandatory access control. Triggers and object-oriented stored procedures are supported and fully integrated with the transaction model. There is a novel approach to distributed databases based on REST and versioning.

There are some practical sessions in appendices. Most of these use Pyrrho as the database engine, but the first appendix gives a more general introduction to building database applications, using a number of database products. Shorter practical exercises are included in the chapters as the corresponding topic is presented. Unfortunately no product sticks rigidly to the standard SQL syntax is described in ISO-9075. Pyrrho is closer than most but the SQL syntax appendix is really required reference during practical sessions. A final group of appendices cover some technical details on Pyrrho: the error codes, and system tables.

I expect all readers will have attended at least one course on Databases, but the subject area is quite complex. Actual building of database applications has been a neglected area in most university programmes. The first set of practical exercises in Appendix 1 helps to motivate the discussion by showing some simple approaches to database application development.

## 1.1 Download Pyrrho

All editions of Pyrrho are available from [www.pyrrhodb.org](http://www.pyrrhodb.org/) for immediate download, for Wiodnows or Linux/Mono. Provided the .NET framework has been installed, it is possible to extract all of the files in the distribution to a single folder, and start to use Pyrrho in this folder without making any system or registry changes.[[1]](#footnote-1) The distribution contains a user manual called Pyrrho.docx: there is some overlap of material between Pyrrho.docx and this book, but the focus in this book is on general principles of databases, while the manual contains more technical details about Pyrrho.

Pyrrho is free and open-source. The client OSPLink.dll supports object versioning, SWI-Prolog and PHP. There is also a client library for Java: OSPJ provides the package org.pyrrhodb.\*. There are also some technical explanations of Pyrrho’s inner workings in a document called SourceIntro.docx, and there is a catalogue of the C# classes used in the server in Classes.xlsx.

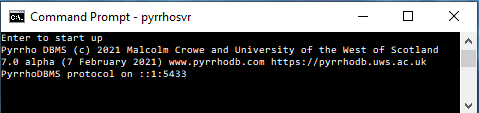
Database files are generally smaller than those of other database products. Files do not contain any indexes or empty space, so an empty database file is less than 1KB. Database files do grow larger if there are many updates because Pyrrho maintains a full historical record.

Currently PyrrhoSvr.exe is approximately 1MB, and when running the server process starts out with about 12MB of memory, but requires approximately ten times as much additional memory as the size of the database file. Memory (RAM) is required only for current data, so if many records in the database have been deleted, or much of the database file consists of updates, the working memory required may well be less than the size of the database file.

Pyrrho is intellectual property of the University of the West of Scotland, United Kingdom. The associated documentation and source code, where available, are copyright of the University of the West of Scotland. You are allowed to view and test the code, and incorporate it in other software, provided you do not create a competing product. You can redistribute any of the files available on the Pyrrho website in their entirety or embed the dlls or any of the source code in an application. Your use of this intellectual property is governed by a standard end-user license agreement, which permits the uses described above without charges. All other use requires a license from the University of the West of Scotland.

## 1.2 The Pyrrho DBMS server

The server PyrrhoSvr.exe is normally placed in the folder that will also contain the database files. The PyrrhoSvr can be started from the command line, by the user who owns this folder. It is a good idea to run the server in a command window, because occasionally this window is used for diagnostic output. (If you are using Embedded Pyrrho only, the database engine is included in the application and so the server does not need to be running.)



Pyrrho provides its client service by default on port 5433, but will find another port if 5433 is already in use. There are flags +s and +S to set up a REST service on ports http 8180 and https 8133, using Windows authentication. (You can supply your own server certificate for transport layer security and/or specify different ports.)

If you try to start the HTTP services and you get Access denied, you can either run the server as administrator, or you can fix the http url reservations. To do this open a command prompt as administrator and issue the following commands (with your full user name where shown):

netsh http add urlacl http://127.0.0.1:8180/ user=*DOMAIN\user* netsh http add urlacl https://127.0.0.1:8133/ user=*DOMAIN\user*

If you get other error messages try using different ports using the command line options described below.

## 1.3 Starting the server

The server is normally started from the command line, in the same folder as the server binary: in the distribution this is in the Pyrrho folder closest to the root of the distribution. The command line syntax is as follows:

**PyrrhoSvr** [-h:*host*] [-H:*hostname*] [–p:*port*] [+s[:*port*]] [+S:[*port*]] [-d:*path*]

On Linux systems, you will need the Mono runtime installed, and the command line begins **mono** **PyrrhoSvr.exe** .

The –h and –p arguments are used to set the TCP host name and port number to something other than 127.0.0.1 and 5433 respectively. This can be a useful and simple security precaution. Note that the host IP address used must match the host name given in connection strings. See Appendix 10.

The +s and +S flags can modify the ports for the REST service from the defaults of 8180 and 8133.

The –d flag can be used to specify the server’s database folder: we will explore the use of this option below.

## 1.4 Database files

PyrrhoSvr.exe, the folder that contains it, and all the database files in this folder are normally owned by the same user, called the server account in the following notes. Note that the “database owner” is different – as described in this section.

If the server creates a database (file) on behalf of a client, the database initially contains no security information and can be accessed only by the server account. On the creation of the first role, or the first grant of permissions to a user, the client user’s identity will be recorded in the database file: this user is then established as the database owner, and by default has full administrative control over the database. Access to the file is then as defined by the permissions granted.

Under Linux and even Windows, database file names are case-sensitive. By default, following the SQL standard, database object names are not case-sensitive unless enclosed in double quotes.

You can inspect the database folder from time to time to check everything is in order. It is possible to rename a database file. For embedded applications, the database file(s) should be installed alongside the application (e.g. as an asset).

## 1.5 The client programs

There are two client utilities at present: a traditional command-line interpreter PyrrhoCmd, and a Windows client called PyrrhoSQL. As with all Pyrrho clients, the PyrrhoLink.dll assembly is also required. We discuss these first. The distribution also contains a REST client and a transaction profiling utility. The simplest possible approach is simply to place PyrrhoLink.dll in the same folder as the application that is using it.

PyrrhoCmd is a console application for simple interaction with the Pyrrho server. Basically it allows SQL statements to be issued at the command prompt and displays the results of SELECT statements in a simple form. For most purposes it is best to place the command line utilities such as PyrrhoCmd and the client dll in a different location from the server. They occupy very little disk space: and the databases will be created in the server’s folder.

With embedded databases things are different: Databases will be in the same folder as applications using an embedded edition of Pyrrho.

## 1.6 Checking it works

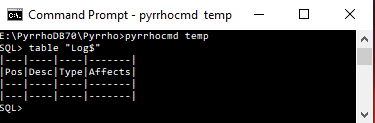
For simplicity, on the same machine as the server, open a command window and use cd to navigate to the same folder as the client executable. Be sure to use the correct version of PyrrhoCmd: you cannot use the professional version of PyrrhoCmd with the open-source server or vice versa. (It is not usually a good idea to start up PyrrhoCmd or the server by double-clicking, because the command line parameters can be useful.)

Ensure the server is running, and there is no database file temp in the server folder, and try the command line

PyrrhoCmd temp

SQL> table "Log$"

In SQL2016 **table** *id* is the same as **select \* from** *id* for base tables and system tables.



This will create a small (5-byte) database file called temp. While the server is running it will now have exclusive access to this file, so it you want to delete it or inpect it with other software you will need to stop the server. You can use control-C, or close the window, to exit from PyrrhoCmd. (If you want to delete the database file that you have just created, you will need to stop the server.)

When starting up PyrrhoCmd, the following command line arguments are supported:

*database* One or more database names on the server.

|  |  |
| --- | --- |
| -h:hostname | Contact a server on another machine. The default is localhost |
| -p:*nnnn* | Contact the server listening on this port number. The default is 5433 |
| -s | Silent: suppress warnings about uploads and downloads |
| -e:command | Use the given command instead of taking console input. (Then the  SQL> prompt is not used.) |
| -f:*file* | Take SQL statements from the given file instead of from the console. |
| -b | No downloads of Blobs |
| -? | Show this information and exit. |

Pyrrho can support locales other than English, but such localisations are not currently included in the distribution. Whether the command prompt (console) window is able to display the localised output will depend on system installation details that are outside Pyrrho’s control. Localisation is more effective with Windows Forms or Web Forms applications.

### The SQL> prompt

PyrrhoCmd is normally used interactively. At the SQL> prompt you can give a single SQL statement. There is no need to add a semicolon at the end. There is no maximum line length either, so if the command wraps around in PyrrhoCmd’s window this is okay.

SQL> set role ranking

Unless you use multiline command as described below, be careful not to use the return key in the middle of an SQL statement as the end of line is interpreted by PyrrhoCmd as EOF for the SQL statement.

At the SQL command prompt, instead of giving an SQL statement, you can specify a command file using @*filename.* Command files are ordinary text files containing an SQL statement on each line.

If wraparound annoys you, then you can enclose multi-line SQL statements in [ ] . [ and ] must then enclose the input, i.e. be the first and last non-blank characters in the input.

SQL> [ create table directors ( id int primary key,

> surname char,

> firstname char, pic blob ) ]

Note that continuation lines are prompted for with > . It is okay to enclose a one-line statement in [ ] .

Note that Pyrrho creates variable length data fields if the length information is missing, as here. This seems strange at first: a field defined as CHAR is actually a string.

Binary data is actually stored inside the database table, and in SQL such data is inserted using hex encoding. But PyrrhoCmd supports a special syntax that uses a filename as a value:

SQL> [ insert into directors (id, surname, firstname) values (1, 'Spielberg', 'Steven', ~spielberg.gif) ]

The above example shows how PyrrhoCmd allows the syntax ~*source* as an alternative to the SQL2011 binary large object syntax X'474946…' . PyrrhoCmd searches for the file in the current folder, and embeds the data into the SQL statement before the statement is sent to the server.

As this behaviour may not be what users expect, the first time Pyrrho uploads or downloads a blob, a message is written to the console, e.g.:

Note: the contents of *source* is being copied as a blob to the server

*source* can be enclosed in single or double quotes, and may be a URL, i.e. ~source can be

~"http://*something*". Another use of ~*file*, for importing datafrom spreadsheets, is described in appendix 3 (section A3.3).

Data is retrieved from the database using TABLE or SELECT statements, as indicated above.

If data returned from the server includes blobs, by default PyrrhoCmd puts these into client-side files with new names of form Blob*nn* .

Blobs retrieved to the client side by this method end up in PyrrhoCmd’s working directory (which is usually different from the database folder). To view them it is usually necessary to change the file extension, e.g. to Blob1.gif. However, on the server side, such data is actually stored permanently inside database files.

Transactions in Pyrrho are mandatory, and are always serializable. By default, each command is committed immediately unless an error occurs. Alternatively, you can start an explicit transaction at the SQL> prompt:

SQL> begin transaction

Then the command line prompt changes to SQL-T> to remind you that a transaction is in progress. This will continue until you issue a **rollback** or **commit** command at the SQL-T> prompt. If an error is reported by the database engine during an explicit transaction, you will get an additional message saying that the transaction has been rolled back.

Note that this reminder and warning behaviour is generated in the command line client on the basis of naïve text matching (of “begin transaction” etc). The use of comments and other noise may affect these feedback mechanisms. The database engine however is not confused. Serious transactions should use the API instead of the command line.

# Chapter 2 Data Consistency, Transparency and Durability

When people speak of databases, consistency, transparency and durability are three of the main properties they ought to expect. But in database software it has very strangely become normal to support inconsistency, and undermine the efforts of software engineers to provide reliable systems, in a mistaken pursuit of speed at all costs. It is not clever to base any decisions on incorrect data, so getting a wrong answer before your competitors is no real advantage. In this book we assume that inconsistency is always bad: since every database starts out in a consistent state it is the job of the DBMS to prevent inconsistencies from creeping in.

Companies almost always store the data for their business processes (customers, employees, accounts etc) in databases. Interestingly, the data in company databases often is retained for decades, long after the computer systems used to create then have ceased to exist. Legacy data still needs to be accessed.

Although computer books often refer to “the corporate database” it is rare now for a company to have just one. But any of these corporate databases can be accessed by numerous other computers and programs in the company or its various branches or customers; and database management systems need to be able to cope with many concurrent connections to the database. Such database systems are said to be server-based (client-server database systems): a program that needs to access the data in the database opens a connection to the server, and uses a standard protocol to send and receive the data it wants. One of the main topics in this course will be the ways that the DBMS ensures consistency and integrity in a busy database system, and is as available as possible.

Some computer programs use local databases: that is, there are no other programs that need to access the data while it is in use. Such database systems are often called embedded. For example, many devices today (phones, cars, fridges etc) have computer systems that use local databases. With HTML5 we have ways for web pages to have local data.

Many important pieces of information in a company are not stored in databases, however. It is fairly rare for spreadsheets, web pages or office documents to be stored in the database. Instead these are stored in the file system as ordinary files. Strangely, although such individual files are commonly security-protected in company computer systems and shared explicitly among user groups, most companies still apply security in an all-or-nothing way the entire database, and not to the separate types of data inside it, despite the availability of security facilities in the database management system. This is yet another aspect we will look at in this module, as it would seem that the security facilities are not currently used because they don’t quite match the company’s needs.

The database data files are accessed by the database server, so they normally belong to the user account that starts up the database server (often this is a special anonymous account). The database server controls who can connect to the database.

The Pyrrho database management system is named after an ancient Greek philosopher, Pyrrho of Elis (360-272BC), who founded the school of Scepticism. We know of this school from writers such as Diogenes Laertius and Sextus Empiricus, and several books about Pyrrhonism (e.g. by Floridi) have recently appeared.

And their philosophy was called investigatory, from their investigating or seeking the truth on all sides.

(Diogenes Laertius p 405)

Pyrrho’s approach was to support investigation rather than mere acceptance of dogmatic or oracular utterance.

Accordingly in this database management system, care is taken to preserve any supporting evidence for data that can be gathered automatically, such as the record of who entered the data, when and (if possible) why; and to maintain a complete record of subsequent alterations to the data on the same basis. The fact and circumstances of such data entry and maintenance provide some evidence for the truthfulness of the data, and, conversely, makes any unusual activity or data easier to investigate. This additional information is available, normally only to the database owner, via SQL queries through the use of system tables, as described in Chapter 8.2 of this manual. It is of course possible to use such automatically-recorded data in databases and applications.

In other ways Pyrrho supports investigation. For example, in SQL2011 renaming of objects requires copying of its data to a new object, In Pyrrho, by contrast, tables and other database objects can be renamed, so that the history of their data can be preserved. From version 4.5, object naming is rolebased (see section 3.6).

The logo on the front cover of this book combines the ancient “Greek key” design, used traditionally in architecture, with the initial letters of Pyrrho, and suggests security in its interlocking elements.

## 2.1 Durability

Of the trio of topics mentioned in the chapter heading, **Durability** looks the easiest: we assume that we want to keep data in a good form, and our business operations need to be recorded properly. Of course the resulting values of data (account balances etc) will be modified later on as a result of our business processes, so that durability means that we should be able to show later that the data had this value today.

Strangely, very few database systems really have the durability property. When values are deleted or modified there is usually no way to recover the values that were there before. Some commercial systems support a transaction log, but there are no mechanisms to require it to be kept, and in practice such documents are seen as redundant/duplication and deleted as a matter of routine.

As a result, for real durability, database designers need to create special tables (journals, histories etc) when such records are a legal requirement.

In the Pyrrho DBMS the transaction log is precisely the durable record of the database, and so it cannot be deleted without deleting the entire database.[[2]](#footnote-2) The current state of the data (with its indexes etc) is in memory. A similar approach has been reported for in-memory column-oriented DBMS by Wust et al (2012). This architecture brings significant advantages: not only do we have durability of the transaction record, but two other advantages: (a) committing a transaction involves appending the transaction record to the end of the file, and (b) the amount of writing to the disk during operation is reduced (according to benchmark measurements for Pyrrho (Crowe 2005)) by a *factor* of 70.

This large difference in performance arises because in the traditional database architecture it is not only the current state of the data that is held on disk, but also all of the indexes and other data structures, and so any change to the database results in changes to many parts of the disk files. In 2004, when Pyrrho’s design was first published, objections from the database community centred on the large amount of memory that would be needed for real commercial databases. In practice databases of 20GB are regarded as reasonable. However, standard DBMS generally use fixed size data fields, while Pyrrho does not, and in 2013, 20GB of memory no longer seems such a large amount.

Pyrrho’s approach works best where durability is important, such as customer records, or financial transactions, where data might need to be retrieved years later. There are circumstances where durability may be less important, for example in an enterprise service bus implementation, where the horizon for durability is measured in minutes rather than years. In ESB systems, such messages would normally only be captured for permanent storage as part of a special troubleshooting activity.

## 2.2 Transparency

**Transparency** (or accountability) means we should be able to discover why and when data changes: who or what made the change, was it routine or unusual? In this course we will see this is closely tied to the concept of business roles. What becomes important is not just who made the change, but also what role they were playing (e.g. Iyer 2009, Oh and Park 2003): were they carrying out part of their day-to-day role of sales clerk or were they doing something else? Some managers may be authorised to play more than one role, but if they are carrying out a standard procedure it is reasonable for them to say what it is (and not just arbitrary, unaccountable, caprice).

A good database design will build in role-based support for the standard business procedures it supports, but very few do. Most DBMS’s simply allow anyone with the power to do something to do it, and don’t provide a mechanism to track the roles being played.

In Pyrrho the transaction record includes not only the user identity for a transaction but the role. A user can only use one role at a time for any given database. Roles should capture and restrict to normal business operations. If it becomes necessary for intervention to correct some unusual condition, some administrative role with greater permissions can be used. Auditing will highlight these and they might usefully indicate a need for process improvement (Moorthy et al, 2011).

## 2.3 Consistency

**Consistency** means that the data does not contain any contradictions. In good database design a first step in this direction is to minimise copied data: if information is repeated in different places it become hard to ensure consistency when that information changes. Dependent information (e.g. total or count) should be correct when it is accessed. In line with the “no copies” rule, it is actually best if a sum or count is recomputed when required, rather than if an old value is stored somewhere.

As mentioned above, in many DBMS, a single transaction results in many changes to data files, many of which are effectively copies, e.g. a new row in a table would typically have the new primary key value stored in several places, bringing a risk of inconsistency. In the next section we explore the link between transactions and consistency, and examine what this means for constraints.

Another difficult area for consistency is where some data is stored in one database (or one computer) and some elsewhere (in a file, in another database, or on another computer). In such situations it is best if responsibility for ensuring consistency resides somewhere, for example, with (one of?) the DBMS involved, but often there are real difficulties, for example, a transfer from one bank to another. Recent research has revisited this problem, addressing the impact of service oriented architectures (e.g. Lars Frank 2011). We return to this sort of problem below.

## 2.4 Transactions

The practical way of ensuring consistency is to use transactions. A transaction consists of a set of changes to the database that is logically ATOMIC. That is, although there might be more than one step, the process comprising these steps is indivisible. The classic example is that of a bank transfer. Although there are two steps (taking a sum of money from one account and placing it in another) the process of transfer is logically indivisible. During the process the total amount of money in the bank will be wrong. So while the separate steps are proceeding, nobody else should be able to see any of the changes until the process is complete and the data is consistent. That is, the transaction needs to be ISOLATED until it is either completed (COMMIT) or abandoned (ROLLBACK).

All practical DBMS allow concurrent access so that several clients can be operating on the database at the same time. Not all of them will be making changes to the database, and in any case changes are likely to affect different parts of the database, and so won’t affect each other. The theory of database transactions considers each transaction as a sequence of read and write operations each occurring at a particular time. Concurrent transactions are transactions whose operations overlap in time: they are valid as long as they are serialisable, that is, if the reads and writes could have achieved the same results if all of the operations of the transactions had been moved in time so that the transactions do not overlap. Many DBMS products write changes to disk storage before the transaction commits: in times past this was needed since some transactions might involve too much data to be held in memory. Researchers have examined higher-level disk operations to improve this mechanism, e.g. Ouyang et al (2011). Pyrrho’s approach is to assume memory is large enough to avoid doing any writing to nonvolatile storage until the transaction commits.

In the database literature transactions are called ACID (atomic, consistent, isolated and durable). All DBMSs provide for transactions, but most allow exceptions to the ACID principles. Allowing exceptions means sacrificing consistency, and in practice many systems then need to introduce notions of compensation activities, to undo changes that may have depended on a transaction that has now been cancelled.[[3]](#footnote-3) For such compensation activities to be automated, they need to be specified in advance for each transaction. In the vast majority of cases, the transaction is not cancelled, and the compensation action is just discarded. It has been estimated that up to 40% of DBMS activity relates to preparation of compensation actions that are never needed, and complex cases have been described that require whole hierarchies of compensation actions.

Dependent systems should not take any consequential action until the database transaction is committed: if this rule is followed there should be no need for compensation. There are several reasons commonly given for not following the rules. One, hinted at above, is that the transaction may involve third parties and the delays involved in using distributed commit protocols seem excessive. This amounts to parties proceeding in the absence of agreement, and must be seen as a risky step. Another reason is that most DBMS using locking for transaction control, and so parts of the database are locked during the distributed transaction protocol, which can be costly. This last point is really quite hard to understand, since robust “optimistic” transaction protocols that minimise locking have been well-documented for decades. Pyrrho uses optimistic concurrency control, and minimises this sort of delay: it also rigorously enforces transaction isolation so that it is impossible to know anything about any ongoing transaction.

The difficulties caused by the bad behaviour of pessimistic (locking) transaction management are farreaching, and have even led to many businesses deciding that they cannot afford transaction management. Other aspects of RDBMS technology have also been blamed for poor performance, and there are many vendors offering no-SQL databases, or columnar databases. Many commercial DBMS prohibit benchmark testing of their products in their licensing arrangements. Pyrrho positively invites benchmarking (Crowe 2005), and despite its rigour its performance is comparable with commercial products.

Pyrrho’s exceptionally good transaction isolation was demonstrated at DBKDA 2020 (Crowe and Laux, 2020) based on the TPC-C benchmark, modified to create greater concurrency.

## 2.5 Application programming interfaces: ADO.NET

There are numerous APIs for contacting database servers: the oldest in common use are ODBC and JDBC. Java Persistence had a brief vogue, as did Microsoft’s LINQ. The PHP API is like a cut-down version of ADO.NET, and versions of ADO.NET are also used for MySQL and Pyrrho.

We will use some ADO.NET sample code in the lab. The following sequence is typical of standard ADO.NET. The first step uses the database connection string. Every database has its own style of connection string - for lots of examples see www.connectionstrings.com var conn = new XXXConnection(“*connectionstring”*); var cmd = conn.CreateCommand(); cmd.CommandText = “*some* SQL SELECT string”;

var rdr = cmd.ExecuteReader();

while (rdr.Read())

{

…. // access the returned data using rdr[0], rdr[1],..

}

rdr.Close(); conn.Close();

This coding pattern is used for SQL strings that do SELECT. You can use cmd.ExecuteNonQuery() for other sorts of SQL commands (update, delete etc).

You can only have one active data reader per connection (you can close one and start another of course). You can have more than one Connection but remember that the DBMS will treat the two connections as completely separate, so that the transaction mechanisms may mean that the two connections see the same or different data.

For this reason, if we need to traverse data from several tables together, we should use SQL joins (this should save a lot of work anyway). We generally keep connections open for as little time as possible, as they can consume resources on the server.

PHP starts the same way:

$conn = new COM(“*connectionObject*”);

$conn->ConnectionString = *connectionString*;

$rdr = $conn->Execute(*SQLstring*);

$row = $rdr->Read();

// Read returns -1 at the end of data, so we continue while $row is not an int: while(!is\_int($row))

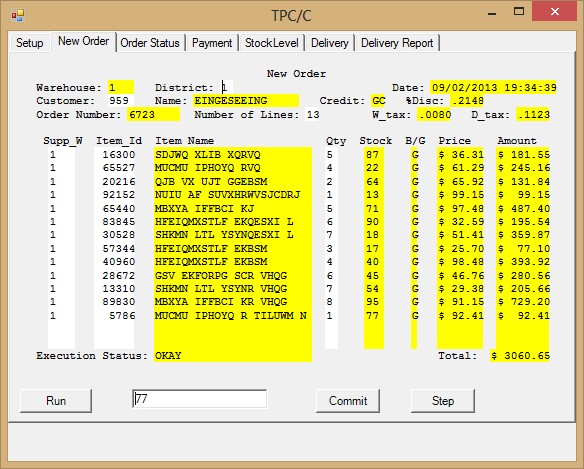
{…// $row[0] etc for access to data returned

}

$rdr->Close();

## 2.6 A first benchmark

The C benchmark from the Transaction Processing Council (Raab et al, 2001) is a legendary test of database performance, and models a clerical order-entry OLTP system. In this benchmark (TPCC) each new order transaction involves over 20 round-trips to the database as the information is built up and submitted, and transactional processing is required. On supercomputing clusters transaction rates of 30 million per minute are reported. Thomson et al (2012) report on the different approaches to achieving such high transaction rates: other than using expensive hardware these all sacrifice something important from the above principles.

Results for ACID RDMBS in PCs are more modest, with 1500-2000 per minute being more normal. In the past I have benchmarked Pyrrho at 2000 per minute (on a Dell laptop with Windows 7). On my current 8-core laptop and Windows 8, I get just over 1000 per minute, but the CPU usage is only 14% as the server runs on just one core.

The TPCC benchmark is designed to behave badly with concurrency, since the next-order-number is a bottleneck. Total throughput of the benchmark is lower with 2 terminals because of transaction conflicts, but then increases slowly as more terminals are added. I can use the CPU more by running multiple concurrent terminals (with 5 the CPU usage reaches 57%).

The TPCC application in the open-source distribution has tabs for the various functional tests of Tpcc, but for the purposes of database tuning the two most interesting are Setup and New Order (pictured). The Setup page allows you to decide how many warehouses, and contains buttons for creating the database, its list of products, the districts with their lists of customers, and the warehouses with their stock. The product descriptions and customer names and addresses are all generated according to randomising rules in the TPCC specification: parameters such as tax are also randomised according to the specification. All of this takes around ten minutes on a PC for a single warehouse, and the database is initially 110MB. You can observe the progress of building the database by using a command window: PyrrhoCmd Tpcc, and then at the SQL> prompt, table “Role$Table” shows the number of rows that have been set up in each table. If the server is shut down and restarted, it takes about a minute for the database to be read in from disk, so if you start up the TPCC application you may have to wait this long before the window appears.

The NewOrder page has two useful buttons: Run will run 2000 new orders and this will take one or two minutes. The Step button allows you to see how a single order is built up. Each step models an action of the clerk to select a district, a customer, an item, a quantity, entering them in the white parts of the screen, and shows the responses from the database in the yellow parts of the screen.

It is not wise to use a single benchmark for performance tuning. But since it includes quite large data sets this particular benchmark can be used with a database engine to investigate what happens to network traffic (it is a huge advantage to use fixed size blocks), what is the best size for BTree buckets (this hardly matters), whether minimising disk reads is worthwhile (5%), what enumeration optimisation can be done (3% for a constant-key shortcut), will a specific index class for integer keys work better than a generic one (no), what is the cost of database features such as multiple database connections (1%), whether only using one kind of integer internally would help (no) etc.

## 2.7 Pyrrho’s internal structure

The database file contains (is) the transaction log, and this gets read in its entirety when the database is loaded following a restart of the database server (cold start). Each database is in a separate file. The database server operates on many database files on behalf of many clients. A single client application can operate many databases at once, by opening connections to one or more databases at a time.

The database file begins with a five-byte “magic cookie” that specifies the database log format.

All subsequent records in the transaction log record the role and user for the transaction[[4]](#footnote-4) together with a universal-time timestamp. All transaction records are immutable, and can be referred to using their position in the data file, which cannot be changed. The maximum size of the database file is 0x40000000, but big data files are broken into 8GB sections for ease of management.

The data formats used for these transaction records are fully described in the Pyrrho manual: there are about 40 different types of record for specifying domains, columns, tables, procedures, and for setting up and modifying roles and security permissions. Any database object or record can be referred to by its defining position. The name of an object thus becomes metadata and can be changed or made role-dependent, so that the same object can be named differently by different roles.

Most of the records will contain data for the base tables. All of these are binary records that do not depend on the machine data formats for the platform used apart from the basic concept of octet. Character data uses Unicode UTF-8 encoding. Integers are represented in the data file as sequences of octets, corresponding roughly to base-256 arithmetic. The maximum integer allowed has 2048 bits. Numeric data is defined by two such integers, for mantissa and power-of-10 scale. Blobs (binary large objects) are stored in the data file like any other data as an integer (possibly a very large one) followed by the blob data as a list of octets. All dates and times use universal time.

This design brings many benefits as briefly mentioned in the above account: platform and locale independence, the ability to refer to a database object by its defining position. Most importantly this simple transaction log design of the database gives a natural automatic serialisation of transactions.

The data file represents level 1 of the Pyrrho engine. At level 2 (Physical) the design consists of the transaction log records, and the concept of data type is defined at level 2. The logical database level is level 3, all of whose structures are in-memory. When a transaction (new or modified level 3 objects) is committed, the level 2 version of the changes is serialised to the data file(s), the in-memory level-3 structures are updated, and the new level 3 head nodes for the database affected will be installed in the list of databases currently loaded in the server.

Each ongoing transaction (level 4) uses a separate space of proposed database objects, so that the objects being defined within them have temporary “positions” above 0x40000000 and these numbers are only unique within that ongoing transaction. At the start of transaction copies are made of the head node of each database in the transaction: effectively this takes a snapshot of the database at the start of the transaction, and the transaction proceeds on the basis of this starting database state.

Committing a transaction requires serialising the transaction to the data file/transaction log, and the defining positions of objects are not known until this is done. During serialisation these are relocated to their actual positions in the data file/transaction log.

The actual process of committing a transaction takes place in 3 stages. In stage 1, the connected datafiles are checked for records that conflict with the current transaction. If conflicts are found, the transaction rolls back. In stage 2, the databases are locked, and this check is repeated for even more recent records, and if all is well, the transaction is serialised to the data file. Finally (stage 3), the data just serialised is installed in the (level 3) data structures, and the locks are released.

Transactions that merely read data cannot conflict with any other transaction: it is as if the entire transaction takes place at the begin time.[[5]](#footnote-5) For transactions that make changes to the database, everything read by the transaction must still be valid at the time the transaction is committed, so it is as if all of the steps of the transaction take place at the commit time.

From this account we see that even with optimistic concurrency, there is always a short time when locks are applied, but locking only occurs at the point of committing the transaction. All of the transaction processing including constraint checking, triggers etc takes place beforehand and all of the data required to commit the transaction has been assembled and is ready for serialisation.

Finally, all of the above discussion needs to be understood in connection with Pyrrho’s multi-threading model. Pyrrho DBMS uses multithreading at the level of connections (level 4): each connection runs in a different thread. The transaction mechanism described in the last chapter applies within the connection’s thread, collecting the database from the server’s (level 3) base thread at the start of each transaction, and synchronising with it when the transaction commits.

In the next chapter we will consider the effects of this approach to transactions and compare with the practice in other DBMS.

The disadvantage of the design is that a cold restart of the database server requires re-reading the entire transaction log: some countervailing measures are (1) using a multi-database design since Pyrrho supports multi-database connections, (2) the technique of partition sequencing discussed in chapter 10.

### Key features of Pyrrho’s design

The above considerations lead to the following feature set for Pyrrho.

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. There is a 5-byte end-of-file marker[[6]](#footnote-6) which is overwritten by each new transaction, but otherwise the 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). Needless to say, the current (in-memory) structure of the database object, or the current values of a record, may well depend on subsequent data. Both the structures in the log and the in-memory database state can be examined using SQL with the help of an extensive collection of system tables (see the Pyrrho manual for details).
3. From version 7 of Pyrrho, Database, Transaction, Table and all other database objects, Query, Domain, Value, Procedure, RowSet and TypedValue and their subclasses are all implemented as shared immutable objects. This helps greatly with transaction safety, and carries little penaly is speed, provided there is plenty of physical memory (RAM). The data structures that enable this are outlined in the next section. There is no way for a Transaction to discover the uncommitted state of any other transaction.
4. When a local transaction commits, there is a validation step, 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 plans 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. Thus the defining positions of any new data will be different from those created in memory for the local transaction: the entire local transaction structure is therefore forgotten even in the case of a successful commit. Instead, the database is updated by reading the new records back from the disk (or disk cache). 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.
5. Because of transaction separation, checking for transaction conflicts cannot be done at the level of the logical database (Level 3). It is done at the physical level (Level 2), with the help of a set of rules for what constitutes a conflict. Data relating to read constraints needs to be passed down to level 2 in a special data structure since these do not correspond to proposed changes to the database.
6. 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 machineindependent (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 database in SQL2011 source form and reparsed when required. This has the advantage that changes consequential on renaming of objects can be supported at the logical database level, where the edits can be applied to the source forms in memory.
7. Database values are strongly typed (TypedValues), but during query processing the server works with SqlValues, which are expressions obtained from the query language. An SqlValue can be evaluated in a Context, to get a TypedValue. Thus a Query is a context whose RowSet is a row of SqlValues (an SqlRow) with a RowEnumerator which modifies the values of the row columns as it moves. This matches well with the top-down approach to parsing and query processing that is used throughout Level 4 of the code.
8. All database objects that contain SQL code (stored procedures, views, check constraints, triggers) are parsed once only, when the definition is given or loaded from the disk. The resulting parsed structures are immutable and shareable: when referred to in queries they can be instanced by occurrence; and when executed they can run in a fresh context (like a stack frame). All such code runs with the privileges of the definer. Rowsets created during such a parse can be transformed by the context of traversal, by adding filters, optimising aggregations, and removing unnecessary steps.
9. The Pyrrho server is multi-threaded, with one thread per connection. Threads are kept completely separate except for a link to the database list. It is shared between all the ongoing transactions that have connected to it, but of course it is immutable. Each transaction gets an immutable copy of the commited objects in the database when it starts up. When one of the transactions wishes to commit, it validates its proposed changes against the shared database, which it then locks until it is finished writing its data (at most one transaction per database can be in this commit phase). It then locks the updates the database list. For full details of the transaction c[propcess, see Appendix.

### Pyrrho’s immutable and shareable data structures

All of the data structures used in Pyrrho are based on a special kind of unbalanced immutable B-Tree in C# whose operations of search, insert and delete are all O(LogN) in the worst case. Each structure is all accessed through single root node reference; and any modification generates a new root node. Taking a copy of any of these structures is very quick, since access is through this one pointer.

All of the structures are generic, and can be instantiated for different key and value types K and V. For BTree<K,V> K must be IComparable, and for CTree<K,V> both K and V are IComparable. There are also BList<V> and Clist<V>: implemented as trees whose key type is int; finding the k-th entry in the list is O(LogN) again, however, apart from adding to the tail of the list, insertion and deletion in lists are O(N) since the list needs to be rebuilt. There is an introduction to Pyrrho’s B-Tree structures in section 3.8 below, and more extensive details are in the SourceIntro document.

Pyrrho’s data structures are then derived from these B-Trees and tuples formed from them, for example a one-to many association of longs will be implemented as CTree<long,CTree<long,bool>>.

Since these objects are immutable, any change creates a new copy, leaving the old copy still accessible. The old and new copies differ by at most O(LogN) entries: the rest are unchanged and are shared between the old new copies. But to ensure correct use the API for these structures is designed as follows:

1. Each subclass has a public constant called Empty, implemented as a single 64-bit null.
2. The addition and removal operations are defined as addition operators.

For example, to see if a BList<V> x is empty, test for x==BList<V>.Empty; to remove the head of the list x, write x -= 0. To add v at the end of the list, write x += v; to insert v at position k, write x += (k,v). It is convenient to define additional operators for dervied classes: for example if MyClass is a subclass of CTree<long,(string,DateTime)>, with a constructor

MyClass(CTree<long,(string,DateTime)> t):base(t.root){}

we could define a new + operator as follows:

public static MyClass operator+ (MyClass m,(long,string,DateTime)p)

{ var (n,s,d)=x; return new MyClass(m+(n,(s,d)); }

Visual Studio provides comprehensive help with ensuring type safety; it colours these + and += operators brown (the syntax colouring for methods) so it is easy to verify which operator implementation has been selected. The above operator will be selected for a MyClass m provided the declared tpe of m is MyClass. It would not be selected if m’s declared type was simply CTree<long,(string,DateTime)>.

There is a simple demonstration in the Appendix to show the logarithmic behaviour of the tree structure.

# Chapter 3: Database Design

The physical layer of a relational database consists of a set of named base tables, whose columns contain values drawn from prescribed sets of values (domains). Standard domains include integers, fixed- and floating point numbers, strings of various kinds, dates, times etc. At the logical layer above this the data in some of these base tables are seen as specifying entities and relations: so that the rows of base tables are seen to give details of unique individual objects identified by primary keys (first normal form), and with relationships to other entities defined in other tables.

At the conceptual level above this we have the business model where these entities and their relationships serve a business purpose.

As mentioned in the introduction, the rules of Normal Form are intended to make it easier to maintain the consistency of data in the database as modifications are made to it. For example, if the same information is contained in several rows of a table, it becomes difficult to change any of these rows consistently, and if we are allowed to update some of the repeated data without updating it all at once, there is an obvious danger that some data will be left unchanged (update anomaly). Second and third normal forms ensure at least that such repeated information is no more than coincidence.

If a row of a table aims to provide too much information, it can happen that at the point of inserting a new entry we are unable to provide information in all of the cells (insertion anomaly). And if some information is removed from the database, but is referred to elsewhere, we have a deletion anomaly. Foreign key relationships can help avoid deletion anomalies.

However, it is not really the job of the database engine to be prescriptive about such matters, merely to provide the tools that the database designers want. There are certain expectations about the standard data types that are supported and their interpretation in various cultures: dates can be represented in different timezones, strings can use international character sets and specific collation sequences, and there are national standards for dates, currencies etc. It should be possible for a column to contain values of a user-defined type (e.g. with subfields) or an array. We will return to some of these aspects in later sections.

## 3.1 Constraints

It should be possible to apply a domain constraint, e.g. to specify that a number should be in a certain range, have a limited set of discrete values, or have a default value. It should be possible in addition to specify a constraint for a column, or an automatic rule to generate the value of a column based on other attributes. Several popular ways of getting the database to generate a primary key for a new row are available. It should be possible to specify a constraint for a table, for example that all values of a particular column are found in the table.

The SQL standard imposes many restrictions on the expressions used to define such constraints, and these are enforced to a greater or lesser degree by different databases. Pyrrho allows any search condition to be used as a column or table constraint, and allows such constraints to be modified later. However, it prevents adding a constraint that is not currently satisfied by data in the table, and does not allow any operation (not even a step in a transaction) that violates any constraint.

### Example 1 database

The PyrhoSvr should already be running.



Start up a command window using the command

PyrrhoCmd Bank

Paste the following text into the PyrrhoCmd window at the SQL> prompt (on Windows, right-click the title bar and select Edit>Paste):

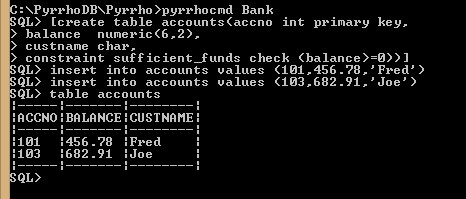
**[create table accounts(accno int primary key, balance numeric(6,2), custname char,**

**constraint sufficient\_funds check (balance>=0))]**

**insert into accounts values (101,456.78,'Fred')**

**insert into accounts values (103,682.91,'Joe') table accounts**

The output should look similar to the following:



## 3.2 Pyrrho’s logs and system tables

Examine the log.[[7]](#footnote-7) Apart from the DOS window wraparound, it looks like this:

SQL> table "Log$"

|---|--------------------------------------------------------------------------------|------------|-------|-----------|

|Pos|Desc |Type |Affects|Transaction|

|---|--------------------------------------------------------------------------------|------------|-------|-----------|

|4 |PRole Bank |PRole |0 |-1 |

|32 |PUser MALCOLM-NB\Malcolm |PUser |32 |-1 |

|55 |PTransaction for 9 Role=4 User=32 Time=10/02/2013 16:27:15 |PTransaction|0 |0 |

|71 |PTable ACCOUNTS |PTable |71 |55 |

|84 |PDomain INTEGER: INTEGER |PDomain |84 |55 |

|106|PDomain NUMERIC%6\_2: NUMERIC,P=6,S=2 |PDomain |106 |55 |

|134|PDomain CHAR: CHAR |PDomain |134 |55 |

|152|PColumn ACCNO for 71(0)[84] |PColumn3 |152 |55 |

|174|PIndex U(56) on 71(152) PrimaryKey |PIndex |174 |55 |

|195|PColumn BALANCE for 71(1)[106] |PColumn3 |195 |55 |

|220|PColumn CUSTNAME for 71(2)[134] |PColumn3 |220 |55 |

|247|Check SUFFICIENT\_FUNDS [71]: (balance>=0) |PCheck |71 |55 |

|284|PTransaction for 1 Role=4 User=32 Time=10/02/2013 16:27:15 |PTransaction|0 |0 |

|300|Record for 71 ([152] 101:INTEGER)([195] 456.78:NUMERIC,P=6,S=2)([220] Fred:CHAR)|Record |300 |284 |

|334|PTransaction for 1 Role=4 User=32 Time=10/02/2013 16:27:15 |PTransaction|0 |0 |

|350|Record for 71 ([152] 103:INTEGER)([195] 682.91:NUMERIC,P=6,S=2)([220] Joe:CHAR) |Record |350 |334 |

|---|--------------------------------------------------------------------------------|------------|-------|-----------|

SQL>

Pyrrho identifies everything in the database by its defining position Pos. When the database is first created a default “Schema” role and the owner are recorded (here at Pos 4 and 32). These are the only records that do not have transaction information. Then we see 3 transactions corresponding to the three SQL commands issued so far. Each transaction records the use, the role and the timestamp.

The last two transactions are the Insert statements (of type Record). The first transaction defines the accounts table and you can see the steps involved in setting up the three domains and the three columns.

The “Log$” table is one of a great many system tables, that allow the SQL engine to examine the database history. There are tables whose names begin with Log$ that are a historical record, while the system tables, beginning with Sys$ or Role$ show the current database objects.

A particularly useful one is Role$Table:

SQL> table "Role$Table"

|---|--------|-------|----|--------|----------------|----------|------|-----|

|Pos|Name |Columns|Rows|Triggers|CheckConstraints|References|RowIri|Owner|

|---|--------|-------|----|--------|----------------|----------|------|-----|

|71 |ACCOUNTS|3 |2 |0 |1 |0 | |Bank |

|---|--------|-------|----|--------|----------------|----------|------|-----|

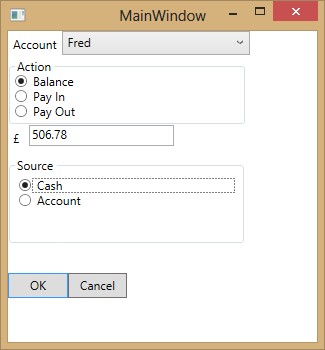
SQL>

This records the current name of table 71 as ACCOUNTS, and shows it currently has 2 rows.

Note that the script we used for creating the table was all in lower case. The SQL standard says that unquoted identifiers are not case-sensitive. If you want case-sensitivity or special characters you need to double-quote the identifier, as we did with “Log$”.

There are some notes on this aspect of the design of Pyrrho DBMS at the end of this chapter.

### Example 2 C# application

Let us write a simple application that uses this database. This one uses Windows Presentation Foundation for simplicity: many other platforms are available. Our purpose is to show the ADO.NET-style interaction with the database. The Pyrrho-specific parts are highlighted in yellow, while the ADO.NET-style parts are highlighted in green.

To create it from scratch, start up Visual Studio and select New Project>Visual C#>Windows>WPF Application. Add a Reference to PyrrhoLink.dll.

Add user interface elements as shown (I used a StackPanel instead of a Grid for the main window and used horizontal and vertical StackPanels for the detail of the groups). You can see my solution in the resources for this text.

The beginning of the code for the application shows how it works. Note the declaration of the PyrrhoConnection and the code for creating and opening the connection (highlighted in yellow).

When we need data from the database, we use ADO.NET incantations, one example is highlighted in green. The transaction to do the funds transfer is highlighted in grey, and notice how it is surrounded with an exception handler.

using Pyrrho;

namespace WpfApplication2

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window {

|  |  |
| --- | --- |
| PyrrhoConnect db = null; | |
| public MainWindow() |  |

{

InitializeComponent();

db = new PyrrhoConnect("Files=Bank");

db.Open();

Account2.Visibility = Visibility.Hidden;

}

class AcInfo

{

public int id;

public string name;

public AcInfo(int i, string n) { id = i; name = n; }

public override string ToString()

{

return name;

}

}

AcInfo selected = null,other = null;

private void Account1\_DropDownOpened(object sender, EventArgs e)

{

var cmd = db.CreateCommand();

cmd.CommandText = "select accno, custname from accounts";

Account1.Items.Clear();

var rdr = cmd.ExecuteReader();

while (rdr.Read())

Account1.Items.Add(new AcInfo(rdr.GetInt32(0), rdr.GetString(1)));

rdr.Close();

}

private void Account1\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

selected = Account1.SelectedItem as AcInfo;

if (selected != null)

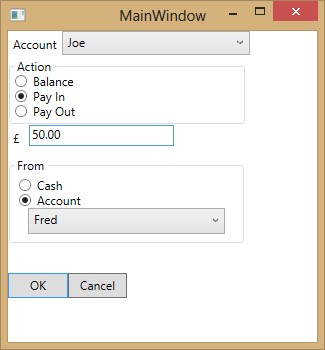
{

var cmd = db.CreateCommand();

cmd.CommandText = "select balance from accounts where accno=" + selected.id;

var rdr = cmd.ExecuteReader();

if (rdr.Read())

 Amount.Text = "" + rdr[0];

rdr.Close();

}

else

Amount.Text = "0.00";

CancelButton.IsEnabled = true;

}

private void Account2\_DropDownOpened(object sender, EventArgs e)

{

if (selected != null)

{

var cmd = db.CreateCommand();

cmd.CommandText = "select accno, custname from accounts";

Account2.Items.Clear();

var rdr = cmd.ExecuteReader();

while (rdr.Read())

if (rdr.GetInt32(0)!=selected.id)

Account2.Items.Add(new AcInfo(rdr.GetInt32(0), rdr.GetString(1)));

rdr.Close();

}

}

private void OKButton\_Click(object sender, RoutedEventArgs e)

{

AcInfo sub = ((bool)PayOut.IsChecked) ? selected : ((bool)PayIn.IsChecked) ? other : null;

AcInfo add = ((bool)PayIn.IsChecked) ? selected : ((bool)PayOut.IsChecked) ? other : null;

try

{

var tr = db.BeginTransaction();

var cmd = db.CreateCommand();

if (sub != null)

{

cmd.CommandText = "update accounts set balance = balance-" + Amount.Text

+ " where accno=" + sub.id;

cmd.ExecuteNonQuery();

}

if (add != null)

{

cmd.CommandText = "update accounts set balance = balance+" + Amount.Text

+ " where accno=" + sub.id;

cmd.ExecuteNonQuery();

}

tr.Commit();

Balance.IsChecked = true;

Cash.IsChecked = true;

Accoun2.SelectedIndex = -1;

Account\_SelectionChanged(Sender, null);

}

catch (Exception ex)

{

Status.Content = ex.Message;

}

}

For reasons of space the UI code is omitted here.

Examination of the code above is for most purposes a sufficient introduction to the ADO.NET API. The only aspect that is not obvious is that you can only have one DataReader open per connection: this is an ADO.NET restriction. If you have a single connection, and you open a DataReader with rdr=cmd.ExecuteReader(), then you must call rdr.Close() before you open another reader. (You can have several connections open but in that case you are not guaranteed that they will see exactly the same data as they have started at different times.)

## 3.4 The Java Library

The Pyrrho Java Connector OSPJC and the org.pyrrhodb.\* package have been significantly modified as of April 2015. In earlier versions of Pyrrho there was an attempt to allow client applications to define the data model unilaterally using Java annotations, in the manner specified for javax.org. From around version 4.5 this has really been untenable, and annotations that differ from the database’s implied data model will in future be reported as errors.

The library is contained in OSPJC\bin in the Open Source Distribution of Pyrrho. It is best to copy this folder to where your Java project is and compile and execute with

javac –cp . xxxx.java java –cp . xxxx

Some features of JDBC 4.1 are completely incompatible with the architecture of PyrrhoDB (and the SQL2011 standard) and thus are unlikely to be incorporated at any stage. These include the DriverManager class, the SQLPermission class, PreparedStaments and their parameters, DataSources and Savepoints. The assumption is that clients open a Connection to a database, and use Statements and ResultSets to manipulate the database.

On the other hand, the intention is that entities specified as such in the database metadata should be retrievable using strongly-typed (generic) client-side methods with the help of reflection. Specifically, single-entity short cuts from Connection will lead to generic versions of first() and next() that automatically populate the public fields of specified entity classes.

import org.pyrrhodb.Connection;

import java.sql.Statement;

import java.sql.ResultSet;

public class JCTest

{

static Connection conn;

public static void main(String args[]) throws Exception

{

conn = Connection.getConnection ("localhost","def","guest","def");

CreateTable();

ShowTable();

}

static void CreateTable() throws Exception

{ try {

conn.act("drop table a"); } catch (Exception e) {}

conn.act("create table a(b int,c char)");

conn.act("insert into a values(1,'One'),(2,'Two')");

}

static void ShowTable()

{ try {

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("select \* from a"); for (boolean b = rs.first();b;b=rs.next())

{

System.out.println(""+rs.getInt("B")+"; "+rs.getString("C"));

} }

catch(Exception e)

{

System.out.println(e.getMessage());

}

}

}

## 3.5 SWI-Prolog

The Open Source Edition of Pyrrho also comes with some support for SWI-Prolog. This is contained in a module pyrrho.pl which is part of the distribution. The code is at an early stage, so comments are welcome. The following documentation uses the conventions of the SWI-Prolog project.

The interface with SWI-Prolog is implemented by providing SWI-Prolog support for the Pyrrho protocol (see technical details in the Pyrrho manual). The following publicly-visible functions are currently supported:

|  |  |
| --- | --- |
| **connect**( -*Conn*,  +ConnectionString ) | Establish a connection to the Open Source Pyrrho server. Conn has the form  **conn**(*InStream*,*InBuffer*,*OutStream*,*OutBuffer*). Codes in OutBuffer are held in reverse order. |
| **sql\_reader**(+Conn0, -Conn1, +SQLString, -Columns) | Like ExecuteReader on the connection. Conn0. Conn1 is the updated connection. Columns is a list of entries of form **column**(*Name*,*Type*) . |
| **read\_row(**+Conn0,Conn1,+Columns,  -Row) | Reads the next row (fails if there is no next row) from the connection Conn0. Conn1 is the updated connection. Columns is the column list as returned from sql\_reader. Row is a list of corresponding values for the current row. |
| close\_reader(+*Conn*) | Closes the reader on connection Conn. |
| **field**(+Columns,+Row,+Name,Value) | Extracts a named value from a row. The atom null is used for null values. |

## 3.6 LINQ

Language-Integrated Query was an innovation in C# 3.0. Linq allows queries of the sort

var query1 = from t in things where t.Cost > 300 select new { t.Owner.Name, t.Cost };

to be written directly in C#.

The Pyrrho support for Linq is therefore inspired by the idea of supporting queries to simple small databases, and avoiding declarations and annotations wherever possible. The client-side objects can be modified using the methods in sec A9.6 but queries should always be to a new connection. The Linq support is only for single-component primary keys (they can be any scalar type and do not have be called “Id”).

To illustrate a complete program using LINQ, first create a database called home, with two tables with the following structure:

create table "Person" ("Id" int primary key, "Name" char, "City" char,"Age" int)

create table "Thing" ("Id" int primary key,"Owner" int references “Person”, "Cost" int, "Descr" char)

Then the Role$Class system table (see Appendix 8) provides text for the corresponding class definitions for your to paste in your code. The part that you writ starts after the “class Program” heading below. The PyrrhoConnect connects to the database as usual, and the database is opened. Two PyrrhoTable<> declarations form a link between client side data and data in the home database. Then the LINQ machinery is available. (For the program to produce output, there needs to be some data in the tables.)

using System;

using System.Collections.Generic; using System.Linq; using System.Text; using Pyrrho;

namespace ConsoleApplication1

{

/// <summary>

/// Class Person from Database home, Role home

/// </summary>

public class Person {

public System.Int64 Id; // primary key

public System.String Name;

public System.String City;

public System.Int64 Age;

}

/// <summary>

/// Class Thing from Database home, Role home

/// </summary>

public class Thing {

public System.Int64 Id; // primary key

public Person Owner;

public System.Int64 Cost;

public System.String Descr;

}

class Program

{

static void Main(string[] args)

{

// Data source.

PyrrhoConnect db = new PyrrhoConnect("Files=home");

db.Open();

// constructs an index for looking up t.Owner as a side effect

var people = new PyrrhoTable<Person>(db);

var things = new PyrrhoTable<Thing>(db);

// Query creation.

var query1 = from t in things where t.Cost > 300 select new { t.Owner.Name, t.Cost };

// Query execution.

foreach (var t in query1)

Console.WriteLine(t.ToString());

var query2 = from p in people

select new { p.Name, Things=from t in things

where t.Owner.Id == p.Id select t};

foreach (var t in query2)

{

Console.WriteLine(t.Name + ":");

foreach (var u in t.Things)

Console.WriteLine(" " + u.Descr);

}

db.Close();

}

}

}

## 3.7 Documents

Databases outlast hardware and software platforms. Many “legacy” database systems still are in use today after 50 years. Databases need to be designed for interoperability, so that the data formats will still make sense years from now. In laboratories we have been looking at how databases can be accessed from a variety of languages and systems. In the labs, we have been using Windows systems, but as many of you will know, the computing industry has been very careful to ensure that systems can work together over the Internet.

TCP/IP is a good start and helps overcome many difficulties by enabling clients and servers to communicate. Java and PHP are available on all platforms, and even C# is available on Linux if the Mono runtimes are installed.

In particular, we have looked at how the very different worlds of .NET and PHP can communicate. There are many examples today of non-relational and “NoSQL” database systems. All of these for various reasons depart from the standard SQL and relational database design. Some of these issues were discussed earlier in this course.

The point of view strongly expressed in this book is that relational DBMS offer a valuable generalpurpose data management infrastructure, and that their most valuable benefits are the ACID properties, especially consistency and durability.

One of the more interesting NoSQL databases today is MongoDB. MongoDB provides schema-less data management for JSON-like “documents” identified by a special \_id key, and there are some notes in the following based on the documentation available on <https://www.mongodb.com/>.

To select a document from a MongoDB collection, a query consists of a Json object with a set of keyvalue pairs: the result set is the set of documents in the collection that match these properties. The properties are not limited to equality conditions. This is a very powerful and attractive way of selecting data from large distributed collections, and much to be preferred over standard SQL in many cases.

Mongo does not use SQL, does not support transactions (it does claim strict consistency) and does not have mechanisms for creating joins. I have begun to develop a MongoDB service offered by the Pyrrho server that when complete will offer full ACID guarantees on top of the full Mongo service. It will do this by running on the Pyrrho database engine, enhanced as necessary.

The enhancements turn out to be very exciting as extensions to SQL. SQL already allows columns in relational tables to contain structured types and XML data. It is exciting to allow integrity and referential constraints to apply to fields within such structured objects. For example, in any document collection the \_id key is the primary key: it is a field within the JSON object.

With Codd’s principles in mind, Pyrrho allows extensions to SQL to support embedded JSON, so that anything that can be done using the Mongo service can also be done using the resulting extended SQL, including the construction of document sets.[[8]](#footnote-8)

For simplicity we will allow PyrrhoCmd to be used by extending SQL a bit. Let’s allow SQL syntax that supports documents, beginning with DOCUMENT as a standard type:

**create table people(doc document)**

We won’t specify primary key (doc.\_id) as this is the default.

**insert into people values ({NAME: "sue", AGE: 26, STATUS: "A", GROUPS: ["news", "sports"]})**

That is, we allow Json literals in value lists[[9]](#footnote-9). Any document can be added to this people table. (Alas, Mongo is case sensitive so we need capitals here or else we need double quotes below). For the query example that follows on page 7, let’s allow almost standard SQL

**select doc from people where doc.age>18 order by doc.age**

where we allow the standard dot notation in SQL to take us into the fields of documents. However, Pyrrho no longer supplies a MonoDB service, and does not support Mongo’s $ operators. It is of course possible in Pyrrho to use normal SQL syntax to make updates to documents, as the document fields appear to the parser like columns, and where conditions for documents allow searching for documents with particular field values. A document array is a sequence of documents neclosed in square breackets and this can be seached like a table.

However, a document is not a table, but instead a rather special column (usually the only one) in a base table.

## 3.8 Pyrrho DBMS low level design

In this section we discuss some of the fundamental data structures used in the DBMS. The data structures in this section have been chosen because they are sufficiently complex or unusual to require such discussion. Obviously this section can be skipped at a first or even later reading.

### B-Trees

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

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

The B-Tree is a standard, 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 pairs (called Slots in the code). An Inner Bucket contains Slots 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 Slots are kept in order of their key values, and the Slots 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.

6

6

Root (Inner)

The BTree dynamically reorganizes its structure so that (apart from the root) all Buckets have at least N/2 Slots, 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.

6

8

1

3

4

Leaves

The basic operations on B-Trees are custom + and – operators. To add a K,V pair to a B-Tree x, we write x += (k,v); and to remove the key k, we write x -= k; . there were some more examples in section 2.7.

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

The following table shows some other commonly-used operations for a BTree<K,V>:

|  |  |
| --- | --- |
| **Name** | **Description** |
| long Count | The number of items in the tree |
| V this[K key] | Get the value for a given key |
| bool Contains(K key) | Whether the tree contains the given key |
| Bookmark First() | Get a bookmark for the first entry in the tree, or null of the tree is empty |
| Bookmark Last() | Get a bookmark for the last entry in the tree, or null of the tree is empty |

From the last entry it seems B-Trees are traversed using bookmarks (or cursors) instead of enumerators.The operations on Bookmarks are very simple:

|  |  |
| --- | --- |
| **Name** | **Description** |
| K key() | Get the key at the current position |
| V value() | Get the value at the current position |
| Bookmark Next() | Returns a bookmark for the next entry in the tree or null if the current entry is the last |
| Bookmark Previous() | Get a bookmark for the previous entry in the tree, or null of the currenrt entry is the first |

We see that Bookmarks are immutable: when we move to the next position in the tree we get a new Bookmark. Also note that it is permitted to modify the tree that the First or Last method provided: but the bookmark continues to traverse the tree as it was at that time.

Unfortunately, C#’s foreach syntax is not really suitable for immutable bookmarks. Instead the standard way of traversing a tree t uses an ordinary for statement::

for (var b=t.First(); b!=null; b=b.Next())

{ // do something with b

}

### 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** |
| Domain kType | Defines the type of a compound key. |
| Domain vType | The type of values indexed using the tree. |
| 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. |
| int depth | The number of components in the key. |

### Integer

All Integer data (the reserved word INT defines a column to be an Integer field[[10]](#footnote-10)) 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 22039-1 .

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.

### 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. Division of Decimals keeps the result to 12 significant figures by default, but of course SQL allows columns to have a specificed precision and/or scale.

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 2103-1 .

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.

### 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.

### Documents

From v.5.1 Pyrrho includes an implementation of Documents as in MongoDB. Assignment of documents follows the MongoDB prescriptions, where $-operators determine how new data is combined into the existing document. The same mechanism is implemented for Update records in the database, so Document fields in Update records normally contain these operators, and Pyrrho computes and caches the updated document when the Update is installed in the database.

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

Document matching recurses down to matching of fields, and then may involve comparisons of (say) 4 with {‘$gt’:3} , so care is taken in the coding that this is implemented as a comparison of {‘$gt’:3} with 4 rather than the other way around, so that Document comparison occurs.

Documents are always retained in memory as in MongoDB, and during updates the modifying Document is stored in the database, while the modified document is only in memory. The PhysBase keeps track of the documents by indexes for (colpos,recpos)->Document and (ObjectId)->Document.

Documents in memory contain no reserved $ keys apart from $id. A document containing $id is a DBRef, and when this is referenced the second index above is used to retrieve the referenced document.

### Domain

Strong types are used internally for all transaction-level processing. The main mechanism for this is the Domain. It provides methods of input and output of data, parsing, coercing, checking assignability etc. Predefined domains include a full set of standard types.

Domain constraints are applied at Level 2, i.e. at the point where a a Record or Update is being prepared for serialisation to the physical database, but they can use level 3 information (e.g. lookup tables implemented using table references). Such dependencies are tracked using the referers list, so that updates to the referenced tables may be restricted: for example where a column uses a domain, a table uses a column, a type uses a record structure (defined by a table). Although tables can be altered, changes and not permitted to the domain of a non-empty column.

The Domain structure also controls the fields (columns) of a structured type, the supertypes (under) a data type has, and the order function that is used. This enables types with custom orderings to be used for primary keys, since the indexes are constructed during database loading. However, the actual names of columns and types are not considered part of the Domain (or Tables) (they are role-dependent, so are generated at the session level). For this reason, types with the same name are not necessarily compatible.

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

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

# Chapter 4: Database Servers

In this chapter, we look at the architecture of a database service from the viewpoint of the communication between client and server. We will consider some alternative architectures along the way.

## 4.1 Servers and services

The usual model of computing is that a great many processes (programs) are running on any computer at any time. Under Windows or Linux around 50 processes have started up by the time you log in. These are almost all services, some are part of the operating system and some are separate executables called servers (in Windows the boundary is often blurred since DLLs are called operating system extensions!).

Separate services are set up when something needs to be shared between processes (between users on a multi-user system, or between tasks where several are running at the same time). What is shared might be

* a resource, such as a printer
* a data file
* a communication gateway so that messages to or from different tasks on the computer don’t get jumbled up

Some resources (such as the processor, memory, devices) are shared directly inside the operating system and tasks requiring them are kept in numerous queues for “system locks”. For example, in C# a program needing a resource whose lock is called MyLock (say) will wrap a section of code with a declaration such as lock(MyLock) { … }, and then execution halts at the lock request until the resource is available, and the lock is released at the end of the critical section.

## 4.2 TCP/IP services

For other things that aren’t at quite such a low level, a server process manages a request queue. A very popular way of doing this is to use TCP/IP. Each server is assigned a port on which it listens for requests for its service. When a client sends a message to this port, the TCP mechanism creates a twoway communication channel (using a new server port) for subsequent communication in that session. Messages on this 2-way channel will generally follow an application-defined protocol of request and response, maybe including callbacks and exception handling.

All operating systems have limits on the number of ports that can be open at any time, so it is important for the TCP channel to get closed as soon as possible. This will happen if either the client or server process terminates, but obviously it is important not to wait until then.

Many application protocols are designed to be very short-term. A Web service lasts only as long as it takes to respond to a single request from the client. An email service remains connected just for long enough to send a message. In these cases, the messages between client and server are simple and text-based. For email, there are headers such as To:, From: and Subject:, followed by the message body. For web servers, the first line of the request contains a verb and a URL, and the last line is blank; while the response begins with a status code, then headers, a blank line, and then the body of the response. (PUT and POST requests also have a body following the blank line.)

Services almost always listen on well-known port numbers, although there is always the option that servers and clients agree on the use of some other port. These port numbers are assigned by the Internet Assigned Numbers Authority ([www.iana.org)](http://www.iana.org/), so you can look up the one you want. Even Pyrrho has its own port number of 5433.

On a PC you can see what listeners are in operation by using netstat –a –b . If you see an IP address in square brackets with colons in it this is an IPv6 address, e.g. [::].

## 4.3 The application protocol

Once the application’s 2-way channel is established, ordinary I/O operations for Streams can be used. The size of the first packet received is used in the end-to-end TCP packet negotiation. For many TCP/IP services (e.g. HTTP) there is only one message in each direction, so this is fine. But if there will be many messages, it is important to remember that the packet size negotiation is done again if the packet size changes. For best results always ensure that all packets used on a channel have the same size. This simple trick can affect communication speeds by a factor of 1000. Pyrrho always uses a packet size of 2048 octets.

For client-server communications it is often important to use asynchronous I/O calls. This is extremely important for interactive applications: you don’t want the user’s display to freeze while the process wits for a reply from a remote server. Many programming systems (such as Windows Forms) already ensure that the interactive events are done by a separate thread, but this brings its own complications, and the run-time system will insist on cross-thread method invocation (Invoke()) where required.

The messages on this two-way communication channel are called (application) protocol data units (PDU). For example, if your sporting club was not using SQL, you might have messages for enrolling a new member, for recording a match, for a score etc. The start of the message might have an identifier to say which message it is (using a small integer 1,2,3 for the different actions) and a set of strings for the different data involved. If you were using SQL, each request might just be a string consisting of an SQL statement (e.g. “insert into members ..”). This assumes that authentication of the user was already done during the connection step.

## 4.4 The client library

Since a service is designed to be used by a number of different applications, there is usually a client-side library whose methods manage the low-level communication with the server. Since low-level details will usually be platform-dependent, it is considered bad design for a client application to write directly to the communications stream. Instead, programming conventions have grown up for different sorts of service so that (for example) client libraries for database services all look very similar, as we have seen in Labs 1 and 2 in this course.

In the early days of programming distributed systems, a monolithic application would be simply broken into client and server by placing some of its components (functions etc) on the server side. Application code on the client side would be unaffected, because proxy functions were provided on the client side with the same names and parameters as before. The proxy would “marshal” its parameters for transmission to the server. This data would be disassembled on the server and used to cal the real server-side function, and so on. Nowadays we try to design in client-server design from the outset.

## 4.5 Sessions

As we have seen, the lifetime of the connection is an important consideration: the length of time connected to the server using PCT is called a TCP/IP session. For database services it often happens that very large amounts of data need to be transmitted between the client and the server, involving many steps and client-side decisions. As we will see, the whole notion of database transactions is very important. Assuming that the processing envisaged by the application involves zero or more transactions, we can imagine that a transaction involves one or more TCP/IP sessions. We will discuss this further later in this course.

At this stage, though, we will note that if the server needs to preserve information about the whole series of transactions (“session state”) from one TCP/IP session to the next, this will require management on multi-server installations where subsequent sessions might be on another server. Either steps will be taken to ensure that subsequent requests from the same client are dealt with by the same server (“server affinity”), or servers will need to share session state somehow.

Another issue is that the TCP/IP setup time referred to earlier can become a significant cost. Some client server systems create pools of connections and threads to try to make this process as efficient as possible.

For both of these reasons, there is an argument for holding a connection open for the duration of a transaction, even though this might be quite a long time.

## 4.6 Database concurrency

In the meantime, the database server needs to be able to deal with other clients. With this design, it becomes inevitable that the database server will manage multiple clients concurrently (dealing with requests in different threads). Even requests from different clients for the same database may be handled at the same time.

How database servers manage this is crucial. In order to preserve consistency we cannot see a jumble of half-done transactions when we look at a database. Transactions (as we will see) need to be atomic. To achieve consistency and atomicity we could lock parts of the database in advance, delaying conflicting transactions until we are done (pessimistic), or we could simply allow all transactions to proceed in isolation, only identifying conflict when transactions commit (optimistic).

Either way, the database server programming will be complicated.

As mentioned above, on big installations there will be many servers. The scenario that a request might be handled by any of them is quite common, but there is an option to place different databases on different (clusters of) servers. Large databases might be partitioned so that data is spread over many servers. We will consider later in the course the different ways that servers can cooperate on managing a database.

Viewing this problem from the viewpoint of the application, we note that on the Internet, the traditional “pessimistic” solution of transactions locking the resources they are considering updating is widely considered unsuitable, and most vendors have settled for a concept of “eventual consistency” (e.g. see Elbushra and Lindström, 2014), relying on compensation methods for resolving conflicts (e.g Lessner et al 2012). Most application frameworks designed for the Web use a combination of optimistic concurrency (on the network) and pessimistic concurrency (in the DBMS). All such hybrid solutions really place the responsibility for verifying consistency on the application, where it does not really belong. We return to this problem in Chapter 5.

## 4.7 Databases and the file system

Changes to the database need to be made durable, so they need to get written to durable media, such as hard disks. The simplest possible sort of database file organisation would have a single (text?) file per database. These days computer memories are so large that many databases would fit in memory – we can almost imagine the whole thing simply being written to disk periodically. But in the interests of speed, we also have indexes to locate information very quickly, and these typically occupy similar amounts of memory to the data itself.

In the old days, computer memories were small, so that the data and indexes were kept on disk. Pages being modified would be in memory, and clever page replacement algorithms would ensure that the disk contents kept pace with changes to the data.

Pyrrho has a different approach: all the indexes and database objects are kept (only) in memory, and the disk file consists exactly of the transaction log. This makes the committing of a transaction very simple, typically a single disk write operation, appending the transaction data to the end of the database.

It is obviously good practice for the DBMS to have exclusive access to the database file while it is operating.

## 4.8 Alternative architectures

Needless to say, various alternatives to the above architectural description are possible. On a personal computer, sharing of data is less of an issue, so for example MS Access is a database product that supports no sharing at all. If such a database is on a shared folder, whole file-locking mechanisms (exclusive access above) are the best that can be managed.

Another no-sharing possibility is where an application has an embedded database. This is a persistent store in the application that is not accessible by any other application. If the database actually uses database technology such as SQL then the application will include a database engine in its executable code.

In other cases, the database technology is less important than the sharing, and people share “No-SQL” databases such a twitter feeds. Such arrangements involve rapid non-transacted publication of data, and nobody minds too much if a tweet gets overwritten. Since the data is not structured, there are no indexes to keep in synchronisation with data, and no database constraints.

Yet another sort of database is provided in “big data” warehousing, where huge read-only aggregations from data sources have been made accessible. Typically these systems avoid trouble by disallowing updates to the warehoused data.

Finally, at the ultra-fast end of the spectrum we have in-memory databases where writing to durable media does not matter.

In chapter 6 we will spend some time looking at information security issues. In client-server systems it must be assumed that authentication is dealt with at the time that connection to the server is established. Secure protocols such as HTTPS can help with this. Many databases are designed for use by people playing different roles: bank managers, bank tellers etc. These roles could be separated out completely by the client applications, but in this course we will study how good database design can help with maintaining a more secure system. Database objects can have permissions associated with them for access by people in different roles; and users can be authorised to exercise roles at different times.

# Chapter 5 Locking vs Optimistic Transactions

In this chapter we return to the question of transaction control as a way of ensuring data consistency in multi-user databases. Although all commercial relational RDMS products use only pessimistic concurrency control, there is a long tradition of research to suggest that optimistic concurrency provides a better solution (e.g. Menascé and Nakamishi 1980, Haritsa et al 1990, Kaspi and Venkataraman 2014). The biggest conceptual hurdle in developing applications for Pyrrho is the use of optimistic transactions. It is very important for programmers to accept this approach as a fact of life, explained in the following paragraphs, and not try to imitate a locking model.

All good database architectures today support the ACID properties of transactions (atomicity, consistency, isolation and durability). Database products that use pessimistic locking (such as SQL Server or Oracle) acquire these locks on behalf of transactions by default, and it is not usually necessary for an application to deal with these issues directly. In a pessimistic locking product, transactions can be delayed (blocked) while waiting for the required locks to become available.

A transaction can fail because it conflicts with another transaction. For example, with pessimistic locking, the server may detect that two (or more) transactions have become deadlocked, that is, all of the transactions in the group is waiting for a lock that is held by another transaction in the group. In these circumstances, the server will abort one of the transactions, and reclaim its locks, so that other transactions in the group can proceed.

With pessimistic locking, if a transaction reaches its commit point, the commit will generally succeed. If it does not complete, it retains locks on database resources until it is rolled back. With SQL Server, for example, once a transaction T begins, it acquires locks on data that it accesses. If it updates any data, it acquires an exclusive lock on the data. Until T commits or is rolled back, no other transaction can access any data written by T or make any change to data read by T.

With optimistic locking, the first sign of failure may well be when the transaction tries to commit. A transaction will fail if it tries to make a change that conflicts with a change made by another transaction. Optimistic transaction handling avoids chaos by inserting an important into the commit process: what a transaction proposes to commit is checked against what updates have been made to the database since the start of the transaction. For this to work well, the transaction needs to work in isolation from others, unaware of what else is happening in the database.

In both cases, it is important for database applications to be prepared to restart transactions. In the case of pessimistic transactions this would normally follow deadlock detection or timeout. With pessimistic locking an attempt could simply be made to re-acquire the same locks: this step could be performed automatically by the server. However, it is not good practice for the transaction’s sequence of SQL statements to be simply replayed, since generally the state of the database will have changed (this is why the transaction failed), and the application should start again to see what to do in this new situation..

In the classic transaction example of withdrawing money from a bank account, a transaction for making a transfer might include an SQL statement of the form “update myaccount set balance=balance-100” or “update myaccount set balance=3456”. Writing SQL statements in the first form makes them apparently easier to restart, but the point being made here is that it should be the client application’s responsibility to decide if the statements should simply be replayed on restart. The server should not simply make assumptions about the business logic of the transaction. Pyrrho transaction checking includes checking that data read by the transaction has not been changed by another transaction.

## 5.1 A scenario of transaction conflict

To simplify the discussion, let us consider operations on a table Products.

Products: (id int primary key, description char, quantity int check(quantity>=0),price int)

In this system, some transactions will involve a purchase (failing in the absence of sufficient stock) A(x): SELECT price FROM Products WHERE id=A.x; UPDATE Products SET quantity=quantity-1 WHERE id=A.x

Some transactions will request several items (failing if any item is unavailable)

B(x1,x2..,xn): BEGIN TRANSACTION A(x1); A(x2);..;A(xn); COMMIT

Now for normal levels of activity, these transactions execute very rapidly, and concurrency is not likely to be a problem. In order to have problems with transaction failures we need to assume huge transaction volumes, a huge size to the Products table, or that the Products table is geographically distributed over many servers, etc. So, assuming that we can imagine transaction conflict occurring at all, we note that transactions of type B(x1,..xn) will conflict with any concurrent A(x) where x=xi some i.

Other transactions will involve adjustments to the stock levels or prices. For example, we might implement a policy of discounting plentiful items whose description matches a given pattern (e.g.

‘%BOLT%’)

C(y): UPDATE Products SET price=price\*0.9 WHERE quantity>40 AND description like C.y

This transaction will conflict with any A(x) such that description(x) is like y , even if quantity(x)≤40. Not all database experts consider that conflict arises here. With PCC we can imagine that all rows of Products would be locked for the duration of C(y) transaction. With OCC, some implementations might not report any conflict on the grounds that C(y) does not modify quantities. However, as transactions A and B also compute a required payment for the items purchased, the cost of the purchase might be debatable if transaction C was concurrently applying a discount. On these grounds we would say that C(y) will conflict with any A(x) such that quantity(x)>100.

Suppose Products contains an item (456,’500 3x5 BOLT’,101,3.00), and A(456) followed by C(‘%BOLT’), the cost of A is 3.00 and the Products table reads (456,’500 3x5 BOLT’,100,3.00) . If C is followed by A, the cost of A is 2.70 and the Products table contains (456,’500 3x5 BOLT’,100,2.70) . The principle of serialisability of transactions should ensure that no other outcomes are possible. In the absence of proper concurrency control, both A and C might proceed on the basis of a snapshot of the Products table at the start of their transactions, so that A would cost 3.00 and the Products table would have (456,’500 3x5 BOLT’,100,2.70). In this business scenario perhaps this uncertainty is unimportant, but in scenarios such as control of dangerous industrial processes, such uncertainties could be a matter of life and death.

To ensure serialisability, we must conclude that A(x) and C(y) will conflict if A(x) affects the value of a Boolean expression used by C, i.e. if quantity(x)=101 and description(x) like y. In this scenario a conflict will also be detected if C actually updates the price used by A.

As a result of these considerations, the conflict implementation rules require that changes made since the start of a transaction are checked for conflict with anything read by the transaction. Suppose that C starts before A starts, but A starts and commits before C attempts to commit. Then C will fail, because one of the quantities read by C has changed. If C commits first, then A will fail if x’s price has changed.

## 5.2 Transactions and Locking Protocols

In chapter 2 we saw an introduction to the detailed operation of Pyrrho’s optimistic transaction mechanism. The main commercial database systems use pessimistic (locking) protocols. The normal argument in favour of locking is that once locks are acquired, a correctly formed transaction can be guaranteed to complete, whereas an optimistic system might have to start over having done some work. However, in pessimistic system the process of acquiring locks can be lengthy and may require the application to release its locks and start again. Years ago there was a lot of debate in both directions, with many research papers claiming that optimistic methods result in higher throughput (e.g. Kung and Robinson 1981). So in both cases, transaction failure can result from conflict: for pessimistic control, it is a conflict of intention, while for optimistic control it is a conflict of action.

Optimistic Concurrency Control (OCC) is not widely used in commercial DBMS products. Although it always produces serialisable transaction behaviour, there is no way of guaranteeing in advance that any given transaction will be able to commit. With pessimistic concurrency control (PCC), a transaction is allowed to delay starting until it has succeeded in locking the data it wishes to read or write, and these locks are maintained by the DBMS until the end of the transaction that owns them.

PCC is subject to denial-of-service attacks, since an attacker can repeatedly request locks on a large set of data items, thus delaying legitimate transactions. Such an attack might leave few traces, since the delay will occur even if the attacker’s transaction makes no changes and simply times out.

In widely distributed information systems made possible by the Internet, the impossibility of maintaining database locking while a user organises their payment methods has led many to abandon the use of ACID databases altogether (Lessner et al 2012). Where locking processes have been combined with such distributed transactions it has been found necessary to introduce the idea of compensation processes, effectively to automate the cancelling of supposedly durable transaction commits.

Recent work from Microsoft, Google, and IBM supports optimistic transaction management by using versioning (e.g. Garus 2012, Guenther 2012, IBM 2011), and this represents a big change from always using locking protocols. In fact, optimistic transaction management is the default in Microsoft’s Entity Framework, Google’s Datastore, and IBMs EJB implementations.

At first sight the requirement at the Internet level for optimistic transactions seems fundamentally at odds with the requirement in the large commercial database systems to use locking protocols. Numerous papers (e.g. those cited above) provide complex workarounds but for many purposes the difficulties are not as severe as might be expected.

The first reason is that internet applications deal directly with databases on behalf of their multiple users: as far as the database is concerned there is only one client. Similarly, the databases at the heart of messaging systems have only one client, namely the enterprise server, and different enterprise servers (Exchange, BizTalk, WebSphere etc) use different databases.

The second is that concurrent access to the database is often limited by tuning arrangements. In many DBMS there may be parallel transactions, but the steps in these transactions are serialised by the TCP request socket mechanism: few DBMS use multi-threading at the step level. As we have seen Pyrrho’s multithreading is at the connection level, and assumes that transactions for that connection are serialised by the client application so that any thread-specific data is disposed of at transaction boundaries.

Finally the granularity of locking can be adjusted. For normal row-based CRUD operations Pyrrho detects conflicts at the row level and detects conflicts at the level of database objects only when schema changes are being made. This level of granularity is slightly harder to achieve in pessimistic systems since a great deal of work may be required in advance to identify exactly which rows will be updated.

## 5.3 Snapshot isolation

Snapshot isolation is when transactions cannot see the activities of concurrent transactions, but proceeds with a view of how the database stood at the time the transaction started. Many DBMSs support snapshot isolation (SI) as an optional mode: Pyrrho enforces it. Importantly, snashot isolation does not ensure transaction serialisation: for this, the validation phase mentioned earlier is needed too.

In databases that provide so-called “serialisable snapshot isolation” (SSI) there is a check made before a transaction commits that no updates since the snapshot conflict with any updates the transaction is about to commit. It used to be claimed that this results in serialisable transactions, but it does not. Transactions are only serialisable if all of their activities (reads and writes) have effect as if there is no concurrency, i.e. there is an ordering of the effective times of transactions such that the same results are achieved.

The commit time of a transaction is when its changes are written to durable media (not the time of the commit request): for example as in Pyrrho by flush-writes to the transaction log. To ensure this serialisability is valid we need to ensure that all of the records we have accessed, some of which we may be about to change, still have the values found in the snapshot taken at the start of the transaction.

Thus a first requirement at commit time of a transaction T is to ensure that none of this data has been modified by other commits since the start of T. The set of read records and proposed physical changes is checked against physical changes that other transactions have committed since the start of T. The first part (the read records) is handled by a list of read constraints maintained by the local transaction. For changes, the proposed physical records about to be written to disk are checked against those that have been committed to the database since the start of T. For example if we are updating a value we need to be sure that the table and record are still there, and the column still exists and has the right type. These checks guarantee consistency of the database.

There are three further checks that are needed to guarantee data integrity. The DBMS will already have checked for entity and referential integrity against the values it has in the snapshot, but at commit time checks are needed against any relevant changes committed by other transactions. These are (a) for primary and unique keys to check that a duplicate key has not been added since the start of T, (b) to check that a key referenced by an inserted record has not been deleted since the start of T and (c) to check that a key we are deleting is not referenced by a record that has been inserted since the start of T.

Needless to say, as soon as we start to check the effects of other transactions, isolation is over, so such checks can only be made during T’s commit, and all the steps described in the last two paragraphs must be done before any other transaction can start to commit. Pyrrho goes to a lot of trouble to reduce the amount of processing needed to carry out these checks, but inevitably the cost of performing the checks is linearly dependent on the number C of changes by T and also on the number D of physical changes by other transactions since the start of T. The cost R of read constraints in T is a bit better than linear in the number of records read: R increases by 1 for each table all of whose records are read, and for each specific record read. The total cost of guaranteeing serialisability is *O*(CD+D*log*R).

In some ways, given the commitment to transaction safety, Pyrrho’s design is near-optimal, and even works well for distributed databases and transactions. It is not excessive, as the benchmarks in Lecture 1 show, but people are not used to paying for this level of serialisability.

There are alternative approaches. If the transaction contains only updates, SSI does guarantee serialisability. If the correctness of reads is not an issue, they should be taken out of the transaction, e.g. handled in a parallel connection. If a transaction dos require correct reads, and SSI is the best available in a DBMS, true serialisability can be gained by upgrading reads to updates of the form “set x=x”. If the DBMS supports row versioning, the application can check these at commit time. But it seems wrong to leave transaction safety to the client.

## 5.4 Transaction masters

The enforcement of full ACID requires that for each resource there should be a single transaction master that enforces serialisability of transactions that access its data, and any process that needs to know an up-to-date value in the resource, or commit a change to this value, needs to be able to communicate with that transaction master.

Many designers of large systems dislike the notion of a single server playing such an important role. There are two objections (a) it acts as a bottleneck at times of heavy traffic, (b) it represents a single point of failure.

If the network is partitioned so that the transaction master is unreachable, then no updates should occur. If the transaction master is permanently out of action, then the remaining network can elect a new one, but it is not reasonable for two disconnected parts of the network to continue separately.

Partitioning a database “horizontally” (for example on a geographical basis) can spread the load between transaction masters. This can enable higher throughput, provided transactions that use data from more than one partition are very rare, as even a small number of distributed transactions can cause serious delays. Transaction profiling can help to identify useful choices of partitions (e.g. see Curino et al 2011). We return to consider distributed databases in Chapter 8.

Despite all the theory, and despite some very well-publicised disasters, few business people accept the need for a single transaction master for each database fragment, or the need to stop all transactions if the network becomes partitioned.

## 5.5 ADO.NET and transactions

It is rather important to understand that ADO.NET 2.0 introduced two parallel mechanisms of operation, with different concurrency mechanisms.

* The DataSet and DataAdapter types used a disconnected data model based on optimistic concurrency
* The DbCommand and DataReader types used pessimistic concurrency.

Because only one DataReader can be open for a given database connection, this all works in a fairly natural way: the duration of locking is the length of time that the data reader is open. We can make local changes to a DataSet, and when we try to commit them to the database, ADO.NET makes a new transaction for us, and checks that the affected records are still the same as when we read them.

## 5.6 Versioning

From April 2015, Pyrrho has a type of row versioning which this section compares with corresponding features in IBM’s SolidDB and Microsoft’s SQL Server.

Pyrrho always supplies a pseudocolumn in all base tables called CHECK. The value is a string that includes the transaction log name, defining position and current offset of the row version. When retrieved it refers to the version valid at the start of the transaction, but it can be used at any time subsequently (inside or outside the transaction) to see if the row has been updated by this or any other transaction (this is the only violation of transaction isolation in Pyrrho).

There is a method in the PyrrhoConnect subclass of IDbConnection for verifying a check string:

|  |  |
| --- | --- |
| bool Check(string ch) | Check to see if a given CHECK pseudocolumn value is still current, i.e.  the row has not been modified by a later transaction. |

Unlike other DBMS, the check cookie is just a note of a transaction log position and so is persistent.

In IBM’s solidDB, the concurrency control mode can be set per table using the ALTER TABLE t SET OPTIMISTIC/PESSEMISTIC statement, or for all tables using the General.Pessimistic setting in solid.ini. With the default optimistic setting:

1. Each time that the server reads a record to try to update it, the server makes a copy of the version number of the record and stores that copy for later reference.
2. When it is time to commit the transaction, the server compares the original version number that it read against the version number of the currently committed data.
   * If the version numbers are the same, then no one else changed the record and the system can write the updated value.
   * If the originally read value and the current value on the disk are not the same, then someone has changed the data since it was read, and the current operation is probably out-of-date. Thus the system discards the version of the data, aborts the transaction, and returns an error message.
   * The step of checking the version numbers is called validation. The validation can be performed at the commit time (normal validation) or at the time of writing each statement (early validation). In solidDB, early validation is the default method (General.TransactionEarlyValidate=yes).

Each time a record is updated, the version number is updated as well.

Each user's transaction sees the database as it was at the time that the transaction started. This way the data that each user sees is consistent throughout the transaction, and users are able to concurrently access the database. Even though the optimistic concurrency control mechanism is sometimes called optimistic locking, it is not a true locking scheme—the system does not place any locks when optimistic concurrency control is used. The term locking is used because optimistic concurrency control serves the same purpose as pessimistic locking by preventing overlapping updates. When you use optimistic locking, you do not find out that there is a conflict until just before you write the updated data. In pessimistic locking, you find out there is a conflict as soon as you try to read the data.

To use an analogy with banks, pessimistic locking is like having a guard at the bank door who checks your account number when you try to enter; if someone else (a spouse, or a merchant to whom you wrote a check) is already in the bank accessing your account, then you cannot enter until that other person finishes her transaction and leaves. Optimistic locking, on the other hand, allows you to walk into the bank at any time and try to do your business, but at the risk that as you are walking out the door the bank guard will tell you that your transaction conflicted with someone else's and you will have to go back and do the transaction again.

With pessimistic locking, the first user to request a lock, gets it. Once you have the lock, no other user or connection can override your lock.

SQL Server’s optimistic concurrency uses versioning: they call it “snapshot isolation”. When a record in a table or index is updated, the new record is stamped with the transaction sequence\_number of the transaction that is doing the update. The previous version of the record is stored in the version store, and the new record contains a pointer to the old record in the version store. Old records in the version store may contain pointers to even older versions. All the old versions of a particular record are chained in a linked list, and SQL Server may need to follow several pointers in a list to reach the right version. Version records need to be kept in the version store only as long as there are there are operations that might require them.

## 5.7 Transaction Profiling

In the current literature, transaction profiling is usually associated with detection of security breaches: a transaction that does not fit any known business process deserves investigation. This paper presents a new model for lightweight transaction profiling that also focuses on traffic optimisation, by considering failed transactions also. Where transactions repeatedly fail because of lock-timeout or transaction conflict, there are two options: redesign of the task, or scheduling it to take place during maintenance periods. In experiments the mechanisms proposed here added less than 10% to DBMS memory usage and had no appreciable effect on transaction timings. These mechanisms can bring immediate benefits in database security and can assist in improvements to access control and role management for a live system.

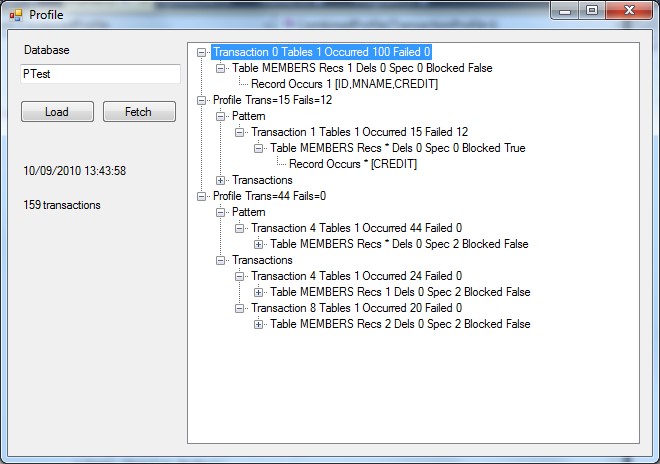
The creation of transaction profiles is very important for proactive security monitoring, and for database traffic management. A poorly designed database, or careless application design, can lead to transactions that lock more data than they need to, and thus degrade performance of the whole system. Where transactions repeatedly fail because of lock-timeout or transaction conflict, there are two options: redesign of the task, or scheduling it to take place during maintenance periods.

Pyrrho includes a mechanism for diagnosing incidents where database transactions fail. Generally, a transaction *T* has a read a set R*T* of data items and is committing changes to a set W*T* of data items, and typically these sets overlap. With optimistic concurrency control, transactions are unaware of concurrent transactions but a transaction commit will fail if another transaction has committed a change to any data item in R*T* W*T*. To analyse such failures, Pyrrho assists with aggregating failures of this type, using a supplemental log that records the read and write details for each transaction along with the success or failure of the transaction.

## 5.8 Profiling implementation

If profiling is turned on for a database, Pyrrho maintains a transaction profile, which is persisted not in the database itself, but in an XML file: this is because it is a record not of the entire database activity, but just the periods for which profiling is enabled. Profiles can be deleted without harming the database in any way.

There is a convenience utility called ProfileViewer which displays the profile in a readable tree-view format. The profile can either be “fetched” from the server (assuming profiling is enabled), or “loaded” from the XML file (in which case ProfileViewer expects to find the xml file in its working folder).



Profiling has a negligible effect on performance and memory use. Profiling can be enabled for all databases, or in the configuration of individual databases.

The purpose of gathering or storing profile information is to understand and monitor the causes of transaction conflicts. Performance tuning and database design should seek to minimise failed transactions during normal operation. It is inevitable that an unusual operation, such as changing the schema or making an update affecting all rows of a table, will be hard to commit during heavy traffic, because a conflicting transaction will probably occur in the meantime.

When profiling is turned off or on for a database called *name* profiling information is destructively saved as or if available loaded from an XML document with name *name.*xml*.* Thus a database administrator can carefully take a database offline by throttling, and then turning off profiling to record a snapshot before shutting down a server, and in this way a full profile of normal operations can be maintained. This level of completeness for profile information will not be achieved if the database server is simply killed.

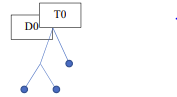
If profiling is enabled, any failed transaction will report its profile. The system profile table will contain the number of successful and failed transactions recorded for this profile: the number of successful transactions will be based on the entire history of the database, while the number of failed transactions recorded will be based on the available information from recorded periods of full profiling (or since the time profiling was enabled for the server).

If profiling is turned on, a set of system tables (Profile$, Profile$ReadConstraint, Profile$Record, Profile$RecordColumn, and Profile$Table) enable inspection of the real-time state of the profile information, always excluding any information about transactions in progress. As with other system tables these tables are not persisted but instrument the running server by exposing in-memory data structures as if they were database tables. The profile viewer described in section 5.8 obtains profile information from these tables or from the XML document, and also groups profiles with similar pattern (for example where everything is the same apart from the number of affected rows).

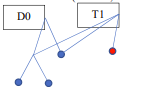
## 5.9 PyrrhoDBMS: safe optimistic transactions

To motivate Prhho’s approach to transactions, consider the following explanation (Crowe, Matalonga, Laiho, [DBKDA 2019](C://Users/Malcolm/Downloads/dbkda_2019_2_10_50029%20(1).pdf)) for StrongDBMS, a much simpler optimistic database built using immutable components. As mentioned in section 3.8 Pyrrho’s data structures have the property that any change creates a new root, but shared many existing nodes with the previous version.

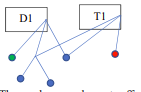
Let us start with a database D initially synchronized with the database file. There will be data structures including a list of objects and a list of names, but we show just one in the illustration.

A transaction T starts by using the same data structures as the database.

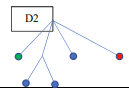
When the transaction makes a change, it creates new root nodes as required, but continues to share the rest of the data structures (indexes, lists of objects etc.).



If another transaction commits in the meantime, the database will be replaced by a new one sharing the same data structures apart from the new root node. These changes do not affect the transaction’s data structures.



When our transaction commits, (it can only do so after checking for no conflicts), its new information is added into the database’s data structures, creating new root nodes as required.



### 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=p0p1…pn etc.

The verification step performed by Pyrrho 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 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 LocalTransaction.check(Physical p) which is called from Commit1.

### Entity Integrity

The main entity integrity mechanism is contained in LocalTransaction. 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 a linked list of values
* A pointer to the next IndexConstraint.

During LocalTransaction.AddRecord and LocalTransaction.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 and discussed next, this list is tested for conflict.

### Referential Integrity

The main referential integrity mechanism is contained in LocalTransaction. 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 a linked list of values
* A pointer to the next ReferenceDeletionConstraint.

Concurrency control for referential constraints for insertions 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 a linked list of values
* A pointer to the next ReferenceInsertionConstraint.

For distributed databases all the above checking information needs to be sent to the transaction master for verification.

# Chapter 6: Role Based Security

In any organisation, the allocation of responsibilities to individuals varies over time, and so it makes sense to assign permissions not to individual users, but to roles. Roles are associated with business processes, not job descriptions: any one individual might be assigned a number of roles in different business processes such as validate travel expenses, assign salesman to region, publish telephone directory. It is a good idea to have a number of roles: if there are none than there is no traceability and no security: anyone can do anything and nobody will ever know why. It is also a good idea that every operation on a database declares the role being exercised, so that the action can be checked for validity. The principles of transparency and accountability mean that people need to explain what they are doing – it is not enough simply to say I am doing this because I can.

Security analysis begins with an account of who is allowed to enter or modify data and on what basis, and who is allowed to read that data. Permissions can be granted to applications as well as to users, but in that case the application takes on the responsibility for allowing different individuals to carry out different operations, and for best results it is these that should be recorded in the transaction record. For a particular database and application, it will be clear at any stage what role is being exercised. The same considerations apply to stored procedures and to methods of structured types.

From this discussion we see that database operations should be granted to roles, and roles should be granted to users. In SQL there is also the possibility of allowing users to administer a role, and allowing a role to grant privileges to other roles.

For an example, suppose we have a warehouse containing products, being ordered by customers. We can imagine that the list of products is maintained by a Manager role, that a Clerk can add a new customer or take a new order, that a Storeman can manually alter stock levels, that a Deliveryman can record that a delivery has been done, etc. We expect none of this information is confidential, except that customer address information is only visible to the Clerk and Deliveryman. Individuals might be allowed to adopt more than one role, but it should always be clear what role they are currently in. So if John the Clerk is temporarily allowed to do something in the Manager’s absence, we should be able to look back to see what he did in that role.

### Example

For another example, suppose a small sporting club (such as squash or tennis) wishes to allow members to record their matches for ranking purposes:

Members: (id int primary key, firstname char)

Played: (id int primary key, winner int references members, loser int references members, agreed boolean)

For simplicity we give everyone select access to both these tables.

**Grant select on members to public**

**Grant select on played to public**

Although Pyrrho records which user makes changes, it will save time if users are not allowed to make arbitrary changes to the Played table. Instead we will have procedure Claim(won,beat) and Agree(id), so that the Agree procedure is effective only when executed by the loser. With some simple assumptions on user names, the two procedures could be as simple as[[11]](#footnote-11):

**[Create procedure claim(won int,beat int) insert into played(winner,loser) values(claim.won,claim.beat)]**

**[Create procedure agree(p int) update played set agreed=true where winner=agree.p and loser in (select m.id from members m where current\_user like ('%'||firstname))]**

We want all members of the club to be able to execute these procedures. We could simply grant execute on these procedures to public. However, it is better practice to grant these permissions instead to a role (say, membergames) and allow any member to use this role:

**Create role membergames 'Matches between members for ranking purposes'**

**Grant execute on procedure claim(int,int) to role membergames**

**Grant execute on procedure agree(int) to role membergames**

**Grant membergames to public**

This example could be extended by considering the actual use made of the Played table in calculating the current rankings, etc.

## 6.1 Application or DBMS based security

In many commercial environments, DBMS security is a neglected topic. This is partially excusable if the only way to use a database is through a set of applications whose use is subject to strong authentication and authorisation mechanisms, but it should be recognised that a good security structure in the databases can lead to greatly enhanced forensic opportunities. These will be lost if the tables in a database are all public, or (worse) if the tables all belong to the database administrator and the database administrator identity is used by all applications.

In many commercial DBMS, it is very hard to discover who made a particular change to the database, or when, or what value was there before the change occurred. Although transaction logs can be maintained, they are often discarded after a time. It is better practice to retain the logs, or to adopt a DBMS design such as Pyrrho’s where the transaction log is inseparable from the database.

On the other hand, the use of DBMS-based security makes a database much less portable. It is no longer a simple matter of copying the database tables to another machine or domain, since the authorisation identifiers will be different.

At v7, Pyrrho puts no security objects in a database by default: however, as long as this remains the case the database can only be accessed by the server’s account. The first role to be defined becomes the schema role, and then the first user to be defined becomes the owner of the database and the administrator of the owning role. Once there are these first security objects, the database can be accessed by the owner or according to the grants they make.

For example, if a student develops a database at home, there is an extra step required to ensure that the database can be accessed using the workplace server, namely to grant all privileges on the base tables to the student’s user identity in the workplace:

**grant all privileges on players to "DOMAIN\user"**

or to PUBLIC of course.

## 6.2 Forensic investigation of a database

Pyrrho supports two kinds of investigation of a database.

First, full log tables are maintained. These are accessible to the current owner of the database, or to an investigator specified in the server configuration file. The log files allow tracing back to discover the full history of any object: when it was created, what changes to it were made, and when it was dropped. In each case, full transaction details are recorded: user, role and timestamp. Since objects can be renamed, logs use numeric identifiers to refer to objects in the database. Full details of the log tables are given in chapter 8. Using these tables it is always possible to obtain details of when and by whom entries were made in the database.

One extension to SQL2011 syntax which assists with forensic investigation is the pseudo-table ROWS(n) where n is the “Pos” attribute of the table concerned in “Sys$Table” (see Appendix 8). For example, suppose we want a complete history of all insert, update and delete operations on table BOOK. Then lookup BOOK in Sys$Table:

**select "Pos" from "Sys$Table" where "Name"='BOOK'**

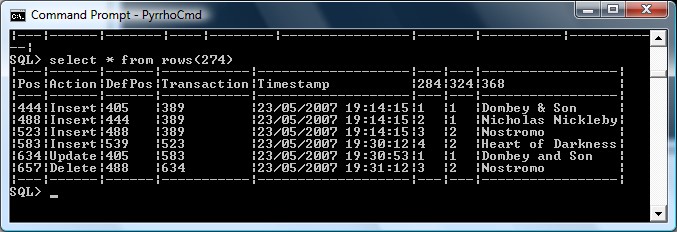
If this yields 274, then the required history is

**select \* from rows(274)**

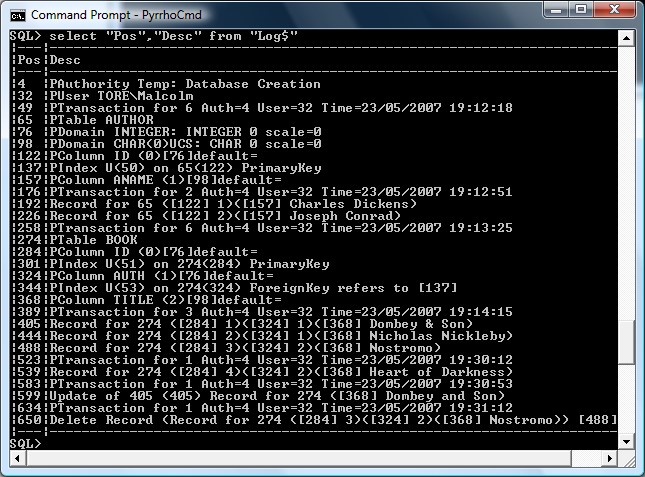
These can of course be combined:

**select \* from rows((select "Pos" from "Sys$Table" where "Name"='BOOK'))**

The second set of parentheses is needed in SQL2011 here to force a scalar subquery.



The Log$ table gives a semi-readable account of all transactions:

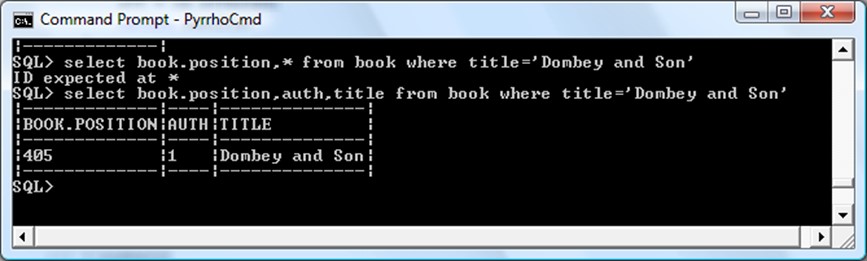


The system log refers to columns and tables by their uniquely identifying number rather than by name. Note also that the Update record shows which field(s) have been modified. Most of the System and log tables have a column called “Pos” which gives the defining position of the relevant entry.

There is a pseudo-column in every table called POSITION which allows the defining position of current records in the database to be retrieved using ordinary queries, e.g. in the above example

**select book.position from book where title='Dombey and Son'**

would give 405. This value is in fact the defining position of the first record with the same primary key as ‘Dombey and Son’, rather than the place where the current title was set, and so it can be used to find other relevant log entries for this record.



Finally in this section we consider some security precautions.

An authorisation identifier is like a user. Users are defined in SQL by granting them privileges. It is assumed that there is some implementation defined way of associating a particular session with an authorisation identifier. For example in SQL Server, users are defined within SQL Server or by means of Windows integrated authentication. In Pyrrho the user identity is taken from Windows in the form “domain\user” (or in Linux from the connection string), and cannot be changed in a session.

In v7 of Pyrrho databases are by default anonymous: there is no role or user information, but as long as this remains the case, unless it is empty, the only user who can use the database is the owner of the server instance. Once a role is created, the first user granted this role becomes the owner of the file: they can then access the file and create more roles and grant roles to other users. Alternatively, a user can simply user the PyrrhoCmd command line program giving a database file name that does not yet exist, and then in the current session they need to create a role and grant it to themselves. There is a lot more information below on Privileges and Roles, starting in section 6.3.

SQL recognises a predefined authorisation identifier \_SYSTEM . Apart from \_SYSTEM, any authorisation identifier can be a granted privileges or roles.

Let us consider a few other security aspects.

1. **The database file and its location**. Databases in Pyrrho consist of a single file, with the same (case-sensitive) name as the database. By default the file is located in the same folder as the server, but the person starting the server can specify a different default folder. If the server is stopped (or has not opened this database since it started) the file can be copied or even deleted. An obvious security precaution is to limit access to the database folder and/or place it on a server or device that ordinary users cannot directly access.
2. **The connection information**. The server listens on a fixed port, which is 5433 by default. Using a less well-kown port and requiring access via a specific TCP address are easy ways of making the system less discoverable.
3. **Backups.** A feature of Pyrrho is that it uses append storage. This means that a copy of the database file taken last week is a snapshot of the database as it stood last week. To copy the database file, the server first needs to be stopped (and then it reloads the whole thing from the file), so it is not very practical to take backups every few minutes.

Stopping the server is safe, but obviously temporarily makes all databases inaccessible. In previous versions of Pyrrho there was a way of detaching (closing) an individual database, which would lose merely all transactions in progress for that database, but there was no way of alerting the users involved.

Some additional utilities can be imagined for the scenario of an unreadable database file: (a) we can get the database to report the current append position on a regular basis, and use this to create a truncated database file to any such highwatermark. (b) since the first entry for any transaction contains a timestamp, we could write a utility that truncates the file just before that record. Neither of these utilities has been written since such an event has never happened with a released version of the DBMS.

The Pyrrho manual contains a section on Troubleshooting which deals with such aspects of database administration for Pyrrho.

## 6.3 Privileges

The SQL standard says that a **privilege** authorises a given category of action to be performed by a specified authorisation identifier on a specified object, such as a table, a column, a domain, a userdefined type, or a routine. The actions that can be specified are insert, update, delete, select, references, usage, under, trigger and execute. Insert, update, select and references can be for whole tables or views, or can be limited to a specified set of columns. Usage privileges apply to domains or user-defined types. Under privileges apply to structured types, and execute privileges apply to routines.

The security model in the SQL standard is based on the GRANT and REVOKE statement. There are two versions of each, for granting or revoking privileges to grantees and for granting or revoking roles to grantees.

A grantable privilege is a privilege that may be granted by a grant privilege statement: it can specify WITH GRANT OPTION in which case the grantee can grant it to others.

Every database object has an owner. This is initially set to the creator of the object, and \_SYSTEM is considered to have granted the owner all privileges when the object is created. On creation a database has a default role with the same name as the database, and the owner of the database can use this role to create the starting set of objects for the database.

The normal way for ownership of a Pyrrho database to be changed is for the database owner to invoke the Pyrrho-specific GRANT OWNER statement. This is implemented as part of the normal database service, and it is good practice to ensure that owners of database objects are user identities that are still available in the operating system.

## 6.4 Roles

A role can be created by the CREATE ROLE statement: initially the owner has administrative rights on the role (granted by \_SYSTEM). The grant role statement is used to allow this role to authorisation idnetifiers, and if WITH ADMIN OPTION the grantee may grant it to others.

Each role is an authorisation identifier. SQL recognises a special role called PUBLIC which is associated with any user and is the owner of standard domains.

In the SQL standard an authorisation identifier is permitted any action granted to it directly or through a role. An authorisation identifier is enabled if it is the current user identifier, the current role name, or the name of a role that is applicable for the current role. A privilege is **current** if it is applicable for an enabled authorisation identifier.

In Pyrrho DBMS only one role is current at any point in a session. A user (authorisation identifier) must choose a single role as the session role, and can modify this within the session using the SET ROLE statement to another role they have been granted. Thus in Pyrrho, the only enabled authorisation identifiers are the current user name and the current role.

## 6.7 The role stack

During query execution, any invoked operation is checked to see that the current role has been authorised to carry out the operation: at the start of the analysis the current role is set to the session role. When the operation involves transferring control to a routine, and the current role is permitted to execute the routine, the current role becomes the role of the routine (the “definer’s role”), and is restored on exit from the routine.

When the transaction is committed, all of the modifications will be recorded against the original user’s id and the session role. When reviewing such a record, it is important to remember that the changes may have been made by a routine operating with different permissions.

In Pyrrho, these permissions affect the metadata that can be viewed in a session, e.g. default values, view definitions and routine definitions. If these are examined by the current user (using the system tables) the SQL text may contain identifiers that have a different name or are not available to the current user and role. Pyrrho displays the identifiers that are appropriate to the current user and role or <hidden> if this data is not viewable from the current user and role.

## 6.8 Revoking privileges

In the SQL standard it is often a complex matter to discover by what route a particular user is entitled to take a particular action. There are many parts of the SQL standard where a schema change results in a cascade of grants of permissions. Unfortunately, it then becomes very unclear what effect revoking a privilege from a user or role will have, as a simple statement of the form

**revoke all privileges on T from "*System*\Fred"**

may allow Fred to retain privileges that he has acquired through some complex chain of grants.

Some DBMS regard this position as unsatisfactory, and in Pyrrho the semantics of grant and revoke operate somewhat differently from the standard. The effect is that a revoke statement of the above form will actually leave Fred with no privileges on the specified object (and any consequential privileges are also removed, in a cascade). The derogation from the SQL standard in this respect is extensively documented in the Pyrrho manual.

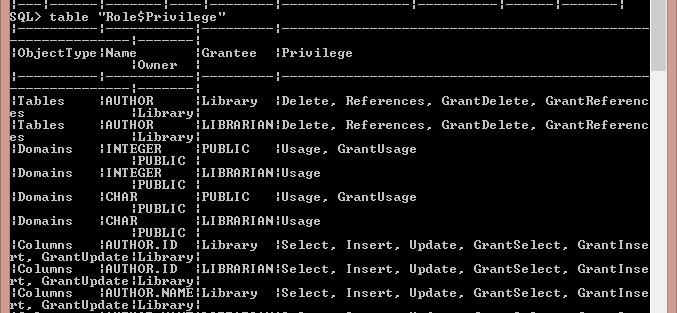
Apart from the owner privilege (which can be held by just one user), granting privileges directly to users is deprecated. It is recommended to grant roles to users instead. Similarly, attempting to create a hierarchy of roles is also deprecated, and in Pyrrho the grant of role A to role B has the effect only of granting role A to all users authorised to use role B at the time of the grant: it does not create a permanent relationship between the roles; revoking a role from a role does nothing, and all roles are in the root namespace. This behaviour appears to be a departure from SQL2011.

Similarly, a grant of privileges does not create any permanent relationship between roles. For example, granting Select on a Table implies granting select on all of the current columns. The grant can be repeated later if new columns are added, or the new columns can be granted. Similarly in Pyrrho, access to a column can be revoked even though the role was previously granted access to the whole table.

Granting a role to a user is different: it means that the user is entitled to exercise the role, and any privileges that the role has at the time of use.

## 6.9 Verifying privileges

The DBMS should provide one or more system tables to verify the current permissions on an object. In Pyrrho, this is done by the “Role$Privilege” table. Here is an example:



As can be seen in the illustration, the command prompt is not the best way to study the contents of this table. The fields in this system table are ObjectType, Name, Grantee, Privilege and Owner. Note the different privileges associated with database objects (e.g. tables and columns).

# Chapter 7: Role-based modelling and legacy data

In the last chapter we examined roles and security. The roles assigned to a user, and the permissions assigned to the roles, control what a user is able to do. For the reasons outlined in the last chapter, Pyrrho requires the user to exercise one role at a time. In this chapter we explore a major advantage of this approach, in that schema changes made by a user in Pyrrho are local to the role, so that each role may have a different data model. We will see that this allows data analytics to operate conveniently, and in real time, on the physical database.

On Windows and Linux systems, the user identity is obtained from the operating system. The user can specify another role in the connection string, or specified by the SET ROLE statement, provided this role has been assigned to them. Pyrrho allows database objects to be renamed or altered by holders of the appropriate permissions: but such renaming and alteration applies to the current role, so that a database object can have different names in different roles.

By default all roles in a Pyrrho database have a default data model based on the base tables, their columns, and can use foreign keys as navigable properties. Composite keys use the list notation for values e.g. (3,4). The default data model can be modified on a per-role basis to provide more user-friendly entity and column names, and user-friendly descriptions of these entities and properties. Tables and columns can be flagged as entities and attributes as desired.

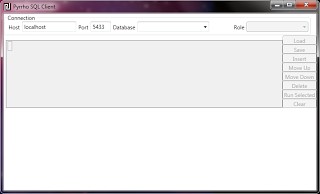
For example, roles could be defined for users in different countries, using entity names, property names and descriptions appropriate to the language of the country, giving access to localised columns or views. These defined views or generated columns could even have specific data types targeting specific roles, since they impose no overhead unless they are explicitly used.

Roles that are granted usage of an object will not see any subsequent name changes applied in the parent role, but the role administrator can define new names. Stored procedures, view definitions, generation rules etc use the definer’s permissions for execution. If the code is examined in the different roles that use them objects will be referred to using the viewing role’s names. If such embedded code refers to objects inaccessible to the viewer, the code will be reported as “(definer’s code)”.

Apart from object names, only the owner of an object can modify objects (ALTER). This includes changes to object constraints and triggers, and inevitably such modifications can disrupt the use of the object by other roles, procedures etc. References in code in other roles can introduce restrictions on dropping of objects, but as usual, cascades override restrictions, and in Pyrrho, revoking privileges always causes a cascade. Granting select on a table must include at least one notnull column. Granting insert privileges for a role must include any notnull columns that do not have default values, and cannot include generated columns.

### An example

We walk through a simple database example, about a library database. Either start up the PyrrhoSQL client as shown,



If you are using the PyrrhoSQL client shown, give the Database name as Library and click Connect. If you are using an ordinary command prompt give the command

**pyrrhocmd Library**

We begin by controlling access to the database, and then start to build a simple database of books and authors.

**create role Library**

**grant Library to "*Yoursystem*\*Yourname*"**

**create table author(id int primary key,name char not null)**

**create table book(id int primary key, title char not null, aid int references author)**

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

**insert into book values(10,'Lord Jim',2),(11,'Nicholas Nickleby',1)**

**table book**



This looks okay to a database specialist but the Librarian is not impressed. He wants the author’s name in the book table: after feebly trying to explain about joins, I provide a special generated column in this table using the standard SQL2008 syntax: alter table book add aname char generated always as (select name from author a where a.id=aid)



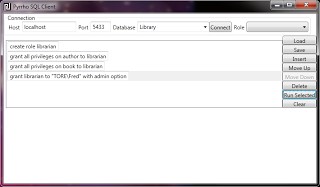
This pleases him a bit but he wants more reader-friendly names and to hide these numeric columns. So I add a new role for the Librarian, and allow Fred the admin option so he can define his preferred column headings:

**create role librarian**

**grant all privileges on author to librarian**

**grant all privileges on book to librarian**

**grant librarian to "COMP10059\Fred" with admin option**



(A generation rule in SQL2011 is not allowed to contain a query expression. Otherwise there are no Pyrrho extensions here.)

Fred can now log in to the system with his Librarian role. With the PyrrhoSQL client shown below, we make sure Fred login with the LIBRARIAN role selected from the drop-down. If he uses the command line, then after starting with pyrrhocmd Library, he needs

**set role librarian**

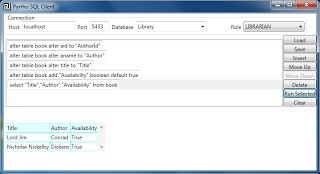
or he won’t see much. He decides to rename some columns (this is a Pyrrho extension), define a new column called Availability, and to create a role for his readers with a simpler table structure:

**alter table book alter aid to "AuthorId"**

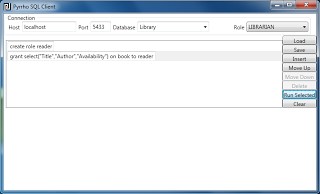
**alter table book alter aname to "Author"**

**alter table book alter title to "Title"**

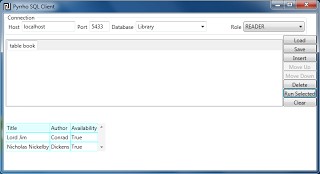
**alter table book add "Availability" boolean default true select "Title","Author","Availability" from book**

 create role reader

**grant select("Title","Author","Availability" on book to reader**

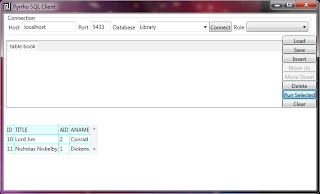


The only columns the Reader can see are the ones granted, so Reader can say simply “table book” to see these:

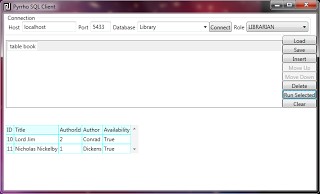


Note that the Author data comes from a table that is otherwise inaccessible to the Reader, because the generation rule uses “definer’s rights”.

Now this is how things stand. The database objects as viewed from the default “Library” role have not changed:



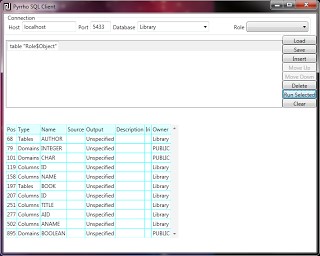
From the Librarian role we have:



and as we have seen the Reader does not see the numeric fields.

## 7.1 How the mechanism works

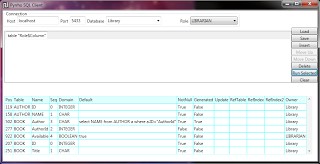
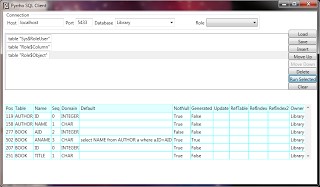
The database creator has set up the following objects in the Library database: tables AUTHOR(ID,NAME) and BOOK(ID,TITLE,AID,ANAME). In the “Role$Object” system table we can see these objects as owned by the default database role, and the PUBLIC standard types that have been used: INTEGER and CHAR as required for these tables, and the BOOLEAN standard type that the librarian used for his new Availability column.

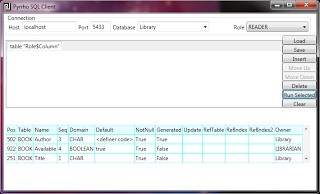


In this table we can also see that database objects can have other role-based metadata such as an output flag (this can be Entity or Attribute as we will see later), a human-readable Description, and an Iri for Web metadata.

In the corresponding tables for the other roles, we see different metadata for different sets of objects. The LIBRARIAN role renamed three of these objects, and defined the Availability column, and the READER role contains just a few entries. As at the end of the last blog posting, the database owner cannot use this role: it was created by the LIBRARIAN and had not yet been made public. Fred can get us the entries, and also make the role PUBLIC so anyone can use it.

Instead of looking at the Role$Object table for each role, let’s instead look at the Role$Column table: the first is for “Library”, the second for “LIBRARIAN”, the third for “READER”:

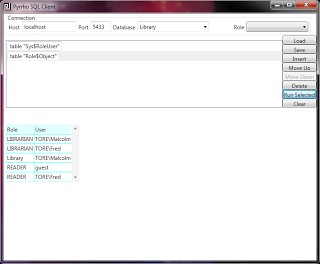




As expected, we see the librarian’s column names in the second two tables. But the biggest difference is the way that the Default value for the ANAME or Author column is shown. None of these exactly matches the actual definition used (select name from author a where a.id=aid). The first screenshot shows the identifiers capitalised, the second uses the LIBRARIAN’s name Authorid for the AID column, and in the third the code cannot be displayed, since the READER role does not know about the AUTHOR table. In fact, Pyrrho uses numeric identifiers internally (select “158” from “68” a where a.”119”=”277”), and, if possible, displays the code appropriately for the viewing role.

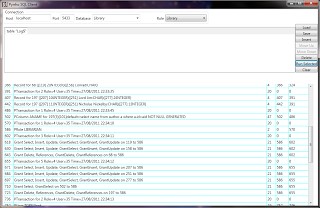
There are four blank columns in these tables Update can specify a set of actual assignments to be carried out if a generated column is assigned to. The next three columns are used for reverse relationship navigation and are specified using Pyrrho’s new REFLECTS syntax.

Let’s examine the list of users allowed to login to roles (this Sys$RoleUser table looks the same from any role allowed to see it):



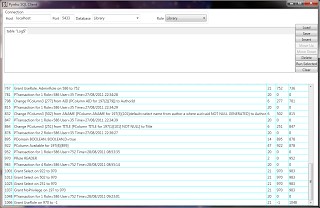
“guest” appears in this list because Fred has ordered “grant reader to public”. Finally in this section I’d like to show some entries from what the database file actually contains. These are from the Log$ table.

The first shows where I create the LIBRARIAN role and grant all privileges on table AUTHOR with grant option. I’d have liked to crop this image to show only the lines from 570 to 643:



We can see that both these transactions are by Role=4, which is the default role for the database (always the first database object to be created). User=35 is TORE\Malcolm, the database creator. So role Library is defining the librarian role. Granting all privileges on the AUTHOR table, unless it is restricted to a particular set of columns, implies granting all privileges on all the columns too. And “all privileges” means all the privileges held by the grantor, in this case also all of the grant options. So the single SQL statement has been expanded into three Grant records in the database. 586, as we can see, refers to the Librarian. Generally, log entries refer to database objects by number rather than by name. Pyrrho has always done this because objects can be renamed. The mechanism now works really well for role-based naming, so that the new version is backwards compatible with existing Pyrrho databases.

The second extract is the last few entries in the log, from 878 on, where the user is Fred:



Here the user is Fred, and the role is LIBRARIAN. The transactions correspond to the four SQL statements:

alter table book add “Available” boolean default true create role reader

grant select(“Title”,”Author”,”Available”) on book to reader grant reader to public

The Grant NoPrivilege entry is probably not required, but at present it ensures that table BOOK (197) is entered in the namespace for READER (970). One other oddity in this list is where the “Avaialble” column is defined. The figures 197(4)[895] are for the table BOOK, the table’s owner, and the new column’s domain, which is a slightly odd collection of data to display in the Log entry (the table’s owner is not actually mentioned in the log entry).

## 7.2 Legacy Data

When importing tables from an existing database, it is good to take the opportunity for some minor redesign. For example, additional integrity constraints can be added, or data types can be simplified, for example by relaxing field size constraints. Keywords that imply such sizes, e.g. DOUBLE PRECISION, BIGINT etc are not supported in Pyrrho, which provides maximum precision by default. National character sets are deprecated since they make data locale-specific: universal character sets are used by default.

A more important area for attention is Pyrrho’s security model. This offers an opportunity for improving the security of the business process. Pyrrho’s default settings are that the database creator’s default role is the schema role, and this will generally allow all desired operations to be performed. But database administrators should take advantage of Pyrrho’s facilities here. Full details are given in Chapter 5, but the following notes provide an executive-level overview of the approach.

The first thing to note is that Pyrrho expects the operating system to handle user authentication so that there is no way for a user to pretend to be someone else: a custom encryption of the connection string is used to ensure this. There is an implicit business requirement to know which staff took the responsibility for data changes (corresponding to initials in former paper-based systems), and Pyrrho’s approach is that it is undesirable for the database management system to force anonymity on such operations by disguising the staff responsible behind a faked-up application identity.

This means that users of the database must be identified and granted permissions. Where the number of authorised staff is large, mechanisms for authorising new users can be automated. Generally it is useful to use the role mechanism to simplify the granting of groups of permissions to the users. Generally a role should be created with the same rights as each application in the legacy system.

Existing users and roles can be imported from the existing database: assuming users are identified in the existing database by their login identities. Where applications have been given user identities in the legacy system, this should generally be replaced by roles. Ideally each business process should have a role to enable associated database changes to be tracked. Each connection to Pyrrho is for a role, and this can enable a good record of the reasons for changes to data.

# Chapter 8: Application Development and the Web

## An example

## 8.1 Connection

## 8.2 Data Definition

## 8.3 Retrieving information

## 8.4 Transactions

## 8.5 The REST service

By default, Pyrrho will try to set up a REST service on ports http 8180 and https 8133, using Windows authentication. (You can supply your own server certificate for transport layer security and/or specify different ports.) Users who have not been granted roles in the database will be able to access public data.

From the viewpoint of this book the importance of considering these features here is that the results are delivered by the standard database engine, so that any updates will be subject to the same access control, constraint checking, triggers etc.

*proto***://***host*:*port*/d*atabase*/*role*{/Selector}{/Processing}

Selector matches[[12]](#footnote-12):

[**table** ]Table\_id

[**procedure** ]*Procedure*\_id [‘(‘ Parameters ‘)’]

**[where** ]*Column\_*id=string

[**select** ]*Column\_*id{,*Column\_*id}

[**key** ]string **of** *Table\_*id[‘(‘*Column*\_id{,*Column\_*id}’)’]

**of** *Table\_*id[’(‘*Table\_*id’)’]

Appending another selector is used to restrict a list of data to match a given primary key value or named column values, or to navigate to another list by following a foreign key, or supply the current result as the parameters of a named procedure, function, or method. The **of** keyword is used for reverse navigation of foreign key relationships, and for navigating many-many relationships: the column list if present is for choosing a foreign key or secondary table. See examples below.

Processing matches:

**distinct [***Column*\_id{, *Column\_*id}] **ascending** *Column*\_id{, *Column\_*id} **descending** *Column*\_id{, *Column\_*id} **skip** *Int\_*string **count** *Int*\_string

The Http/https Accept and Content-Type headers control the formatting used. At present the supported formats are XML (text/xml), HTML (text/html, only for responses) and SQL (text/plain). The Pyrrho distribution includes a REST client which makes it easier to use PUT, POST and DELETE. A URL can be used to GET a single item, a list of rows or single items, PUT an update to a list of items, POST one or more new rows for a table, or DELETE a list of rows. Thus GET and POST are very different operations: for example, POST does not even return data. All tables referenced by selectors must have primary keys.

For example with an obvious data model, GET http://Sales/Sales/Orders/1234 returns a single row from the Orders table, GET http://Sales/Sales/Orders/Total>50.0/OrderDate/distinct returns a list of dates when large orders were placed, GET http://Sales/Sales/Orders/OrderDate,Total returns just the dates and totals, GET http://Sales/Sales/Orders/1234/of OrderItem returns a list of rows from the OrderItem table, and GET http://Sales/Sales/Orders/1234/CUST/Customer/NAME returns the name of the customer who placed order 1234. The response will contain a list of rows: if HTML has been requested it will display as a table (or a chart if specified by the Metadata flags, sec 7.2). Using HTML will also localise the output for dates etc for the client. Metadata can also specify XSL transforms so that simple business objects can be generated directly by Pyrrho.

PUT http://Sales/Sales/Orders/1234/DeliveryDate with text/plain content of ((date’2011-07-20’)) will update the DeliveryDate cell in a row of the Orders table. PUT content consists of an array of rows, whose type must match the rowset returned by the URL. If the array has more than one row, commas can be used as separators. XML format can also be used, which should match the data format returned by the URL.

POST http://Sales/Sales/Orders will create one or more new rows in the Orders table. In Pyrrho an integer primary key can be left unspecified. In SQL (text/plain) format, column names can be included in the row format, e.g. (NAME:’Fred’,”DoB”:date’2007-10-22’): if no names are provided, all columns are expected. Remember that the REST service is case-sensitive for database object names. XML format can also be used, in which case column values for the new row can be supplied either as attributes or child nodes irrespective of the data model. A mime type of text/csv has been added to facilitate import from Excel spreadsheets.

If no Selector or Processing components are provided, the target is the Role itself. For this target GET returns a set of C# class definitions for POCO use, as described in section 3.6.

There are some extra columns in the Log$ table: the record type, a related previous log entry if any, and the start of the transaction.

For simplicity we will continue to use the example database from the last two postings, with the following additional steps. These create a new role "Web" with read-only access to the tables, and some metadata for XML formatting of the output. The details are given at the end of this section. For now let’s just consider GET actions. (As we will see in the next posting, one simple way of handling moderate security for PUT, DELETE and POST is to grant Insert, Update and Delete to a role with a hard-to-guess name.)

We can use REST to explore the Web role of the database:

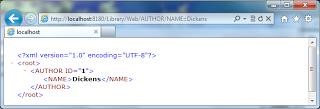
For the Author table, AUTHOR has been declared as an entity, and ID has been declared as an attribute, so the output has a different form:



We can limit the output to particular columns:



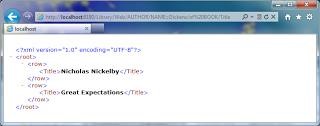
We can select rows satisfying a particular set of conditions:



We can navigate foreign keys:



We can reverse-navigate foreign keys using the OF keyword (see below)



The relationship of this Library database to the one in the section 7.1 is (approximately) as follows:

In role Library:

**Create role "Web"**

**Grant select on author to "Web"**

**Grant select on book to "Web"**

**Insert into book(title,aid) values('Great Expectations',1)**

In role “Web”:

**Alter table author entity**

**Alter table author alter id attribute**

## 8.6 Data Visualisation and CSS

The system table Role$Object shows the metadata defined for database objects. Some of these are for control of Pyrrho’s Web services.

In particular, Pyrrho offers some simple charts that can be obtained via the Web services.

The Metadata flags and the string “constraint” are Pyrrho extensions., Attribute and Entity affect XML output in the role, Histogram, Legend, Line, Points, Pie (for table, view or function metadata), Caption, X and Y (for column or subobject metadata) affect HTML output in the role. HTML responses will have JavaScript added to draw the data visualisations specified. The string is for a description, and for X and Y columns is used to label the axes of charts. If the table’s description string begins with a < it will be included in the HTML so that it can supply CSS style or script. If the table’s string metadata begins with <? it should be an XSL transform, and Pyrrho will generate and then transform the return data as XML.

## 8.7 REST and POCO

# Chapter 9: Classic Database topics

## 9.1 Portability and Internationalisation

In SQL, for the convenience of users, time values without specified time zones are assumed to be local. This approach is easy to criticise: after all, the client and server are frequently in different time zones. Accordingly, in Pyrrho any time value without a specified timezone is assumed to be coordinated universal time (UTC) and this is how it is stored in the database.

### DateTime

SQL specifies its own special syntax for dates and times, but on output from the database, Pyrrho uses the operating system’s regional settings, and/or the conventions of the target language (such as C#) to represent the datetime.

|  |  |
| --- | --- |
| Keyword | Meaning |
| YEAR | Year |
| MONTH | Month within year |
| DAY | Day within month |
| HOUR | Hour within day |
| MINUTE | Minute within hour |
| SECOND | Second and possibly fraction of a second within minute |
| TIMEZONE\_HOUR | Hour value of time zone displacement |
| TIMEZONE\_MINUTE | Minute value of time zone displacement |

There is an ordering of the significance of <primary datetime field>s. This is, from most significant to least significant: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

The <primary datetime field>s other than SECOND contain non-negative integer values, constrained by the natural rules for dates using the Gregorian calendar. SECOND, however, can be defined to have a <time fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value, a non-negative exact numeric value.

There are three classes of datetime data types defined within this part of ISO/IEC 9075:

* DATE — contains the <primary datetime field>s YEAR, MONTH, and DAY.
* TIME — contains the <primary datetime field>s HOUR, MINUTE, and SECOND
* TIMESTAMP — contains the <primary datetime field>s YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

Items of type datetime are comparable only if they have the same <primary datetime field>s.

### Intervals

There are two classes of intervals. One class, called *year-month* intervals, has an express or implied datetime precision that includes no fields other than YEAR and MONTH, though not both are required. The other class, called *day-time* intervals, has an express or implied interval precision that can include any fields other than YEAR or MONTH.

A year-month interval is made up of a contiguous subset of those fields.

|  |  |
| --- | --- |
| Keyword | Meaning |
| YEAR | Years |
| MONTH | Months |

A daytime interval is made up of a contiguous subset of those fields.

|  |  |
| --- | --- |
| Keyword | Meaning |
| DAY | Days |
| HOUR | Hours |
| MINUTE | Minutes |
| SECOND | Seconds and possibly fractions of a second |

The actual subset of fields that comprise a value of either type of interval is defined by an <interval qualifier> and this subset is known as the precision of the value.

Within a value of type interval, the first field is constrained only by the <interval leading field precision> of the associated <interval qualifier>.

|  |  |
| --- | --- |
| Keyword | Valid values of INTERVAL fields |
| YEAR | Unconstrained except by <interval leading field precision> |
| MONTH | Months (within years) (0-11) |
| DAY | Unconstrained except by <interval leading field precision> |
| HOUR | Hours (within days) (0-23) |
| MINUTE | Minutes (within hours) (0-59) |
| SECOND | Seconds (within minutes) (0-59.999...) |

### Localisation and collations

Pyrrho uses the character set names as specified in SQL2011. Specifying a character set restricts the values that can be used, not the format of those values. By default, the UCS character set is used.

By default, the UNICODE collation is used, and all collation names supported by the .NET framework are supported by Pyrrho. CHAR uses standard culture-independent Unicode. NCHAR is no longer supported and is silently converted to CHAR. UCS\_BINARY is supported.

The SQL2011 standard specifies a locale-neutral interface language to the server, for example using a locale-independent format for dates in SQL statements.

Pyrrho maintains this, with locale-neutral database files. From this viewpoint, localisation is a matter for client configuration, since once dates, currency amounts etc are being displayed in the client, the client application will use the UI properties set in the client’s operating system.

Some areas of localisation are notorious: no two people in Scotland will agree on how to sort the names Mabon, Mackay, MacKay, McKay, Macdonald, McDonald, MacDonald, M’Donald, Martin, and the placement of names starting with de, or the treatment of double surnames in Spain, present wellknown problems.

The Unicode reports ([http://www.unicode.org/reports/tr10/)](http://www.unicode.org/reports/tr10/) give examples including conventions for orderings based on accents, whether case is considered, or punctuation, or combination characters.

Fortunately, the makers of operating systems have solved these problems for us, with satisfactory libraries to associate acceptable behaviours with a given locale. While the result will not please everyone, at least it will be consistent across a large number of applications written for that operating system.

Thus, ordering of strings on the server side (for example with the SQL ORDER BY clause), will need to be informed about the locale context for the strings being sorted. The SQL standard assumes that the collation to use is defined for each table, or within a given SQL statement or expression. In designing localised applications, programmers need to select the right combination of server side settings to give an entirely satisfactory localised experience.

At the server side, views and updatable generated columns provide opportunities for localisation, which can be targeted by defining roles for specific locales. With the help of the role-based models in Chapter 7, a role with the Chinese locale (for example) could see Chinese names for tables and columns, and there could be role-specific “generated always” columns that defined Chinese versions of informative messages for a user application. The localised version of the application would then attach to the appropriate role and use the appropriate column to get the localised resources for the application. This could be a very powerful tool to add to the localisation mechanisms available in ASP.NET for example.

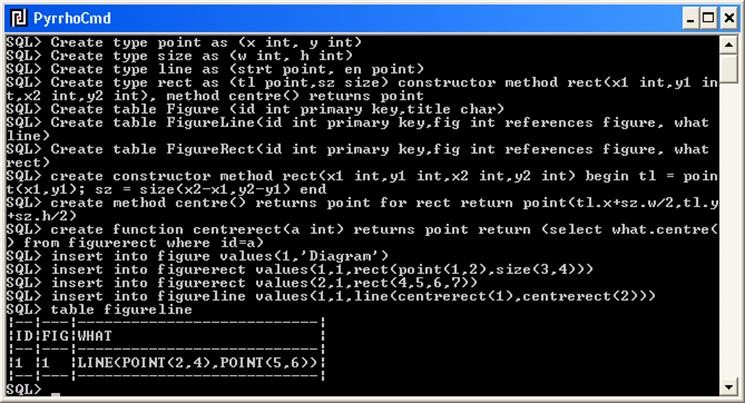
There are evident gaps in this provision. For example, it could be a property of a role to set the locale, so that the collations used for ordering string data are set by default to match the role’s locale.

## 9.2 Using structured types

Structured types, multisets and arrays can be stored in tables. There is a difference between (say) a table with certain columns, a multiset of rows with similarly named fields and a multiset of a structured type with similarly named attributes, even though in an element of each of these the value of a column, field or attribute respectively is referenced by syntax of the form a.b . Some constructs within SQL2011 overcome these differences: for example the INSERT statement uses a set of values of a compatible row type to insert data into a table, and TABLE v constructs a table out of a multiset v. The type model in Pyrrho allows user-defined types to be simple or structured, they can define XML data types (e.g. for RDF/OWL use) have an associated URI and constraints.

To use structured types, it is necessary to CREATE TYPE for the structured type: this indicates the attributes and methods that instances of the type will have. Then a table (for example) can be defined that has a column whose vales belong to this type. At this stage the table could even be populated since (there is an implicit constructor for any structured type); but before any methods can be invoked they need to be given bodies using the CREATE METHOD construct. Note that you cannot have a type with the same name as a table or a domain (since a type has features of both).

Values of a structured type can be created (using a constructor function), assigned to variables, used as parameters to suitably declared routines, used as the source of methods, and placed in suitably declared fields or columns.



(Note that the coordinates have been declared as int, so the first point here is not (2.5, 4)).

Arrays and multisets of known types do not need explicit type declaration. Their use can be specified by the use of the keyword ARRAY or MULTISET following the type definition of a column or domain.

You can define ordering functions for any user-defined type. Ordering functions can simply specify equality, or allow < and > comparison (FULL). The ordering function can specify a relative ordering, in which case the corresponding routine takes two parameters of the udt, and returns -1, 0 or 1. A mapping ordering function defines a numeric value for each value of the udt which can then be compared. The other option you can specify (STATE) merely tests for componentwise equality.

## 9.3 Stored Procedures and Methods

In SQL2011 the syntax :v is not supported for variable references, and instead variables are identified by qualified identifier chains of form a.b.v . The syntax ? for parameters is not supported either.

In SQL2011-2-11.50 we see that procedures never have a returns clause (functions should be used if a value is to be returned), and procedure parameters can be declared IN, OUT or INOUT and can be RESULT parameters. Variables can be ROW types and collection types. For functions, TABLE is a valid RETURNS type (it is strictly speaking a “multiset” in SQL2011 terminology). From SQL2011-2-6.39 we see that RETURN TABLE ( queryexpression ) is a valid return statement.

Here are some outlines of procedure statements specified in SQL2011-4 and supported in Pyrrho. Complete syntax summaries for Pyrrho are given in Appendix 6.

The operation of the security model for routines is rather subtle. All routines operate with definer’s rights by default, but access to them is controlled according to the current role.

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

create function booksby(auth char) returns table(title char) return table(select title from author a inner join book b on a.id =

b.authid where aname = booksby.auth )

This example also shows that a routine body is a single procedure statement (possibly a compound

BEGIN..END statement). If you use the command line utility PyrrhoCmd, very long SQL statements such as the last one above can be enclosed in square brackets and supplied on several lines as described in section 1.7.

The above function can be referenced by statements such as

select \* from table(booksby('Charles Dickens'))

The keyword table in this example is required by SQL2011-2(7.6).

However, methods of a structured type van have a returns clause.

## 9.4 Condition handling statements

The following condition handling apparatus (as specified in SQL2011) is also supported. The predefined conditions are SQLSTATE string, SQLEXCEPTION, SQLWARNING and NOT FOUND, but any identifier can be used . All of the following can appear where statements are expected, and handlers apply anywhere in the scope where they are declared.

DECLARE CONTINUE HANDLER FOR conditions statement

DECLARE EXIT HANDLER FOR conditions statement

DECLARE UNDO HANDLER FOR conditions statement

UNDO is defined in the SQL standard (04-4.8): it offers more fine-grained behaviour than rollback, as it merely removes any changes made in the scope of the handler.

SIGNAL condition setlist

Here the options for condition are SQLSTATE string or any identifier. The setlist allows a set of keywords defined n the SQL standard, corresponding to items in the diagnostics area. For example, you can pass a reason in the diagnostic area using the MESSAGE\_TEXT keyword.

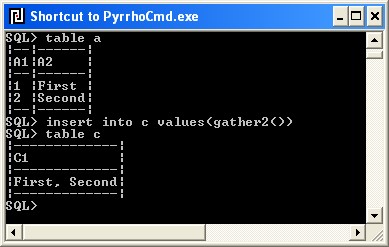
### Examples

The following functions perform the same task. The first uses a handler, while the second uses a for statement.

create function gather1() returns char begin declare c cursor for select a2 from a; declare done Boolean default false; declare continue handler for sqlstate '02000' set done=true; declare a char default ''; declare p char; open c; repeat fetch c into p; if not done then if a = '' then set a = p else set a = a || ', ' || p end if end if until done end repeat; close c; return a

end

|  |
| --- |
| create function gather2() returns char begin declare b char default ''; for select a2 from a do  if b='' then  set b = a2 else  set b = b || ', ' || a2 end if  end for; return b end |



## 9.5 Triggers

## 9.6 Views

## 9.7 REST and Views

# References

Ceri, S.; Pelagatti, G. (1984): Distributed Database Design: Principles and Systems (McGraw-Hill)

Cheung, A., Arden, O. (2012): Automatic Partitioning of Database Applications, Proceedings of the VLDB Endowment, 5, p. 1471-1482

Codd, E. F. (1985) Does your DBMS run by the rules? *ComputerWorld* Oct 21, 1985.

Crowe, M. K. (2005): Transactions in the Pyrrho Database Engine, in Hamza M.H. (ed) Proceedings of the IASTED International Conference on Databases and Applications *DBA 2005*, Innsbruck, February

2005 (IASTED, Anaheim) ISBN 0-88986-460-8 ISSN 1027-2666, p.71-76,

[https://www.researchgate.net/publication/220984896\_Transactions\_in\_the\_Pyrrho\_Database\_Engi ne](https://www.researchgate.net/publication/220984896_Transactions_in_the_Pyrrho_Database_Engine)

Crowe, M. K. (2007): An introduction to the source code of the Pyrrho DBMS. *Computing and Information Systems Technical Reports*, **40**, University of Paisley. An updated version is included in the Pyrrho distribution as SourceIntro.docx.

Crowe, M. K. and Laux, F. (2020): Reconsidering Optimistic Algorithms for Relational DBMS, in *DBKDA 2020, The Twelfth International Conferences on Advances in Databases, Knowledge and Data Applications*, Lisbon (ThinkMind Digital Library) **ISSN:** 2308-4332, **ISBN:** 978-1-61208-790-0 <https://www.thinkmind.org/index.php?view=article&articleid=dbkda_2020_1_50_50030>

Curino, C.; Jones, E. P. C.; Popa, R. A.; Malviya, N.; Wu, E.; Madden, S.; Balakrishnan, H.; Zeldovich, N. (2011): "Relational Cloud: A Database-as-a-Service for the Cloud." 5th Biennial Conference on Innovative Data Systems Research, CIDR 2011, January 9-12, 2011 Asilomar, California.

Elbushra, M. M., Lindström, J., (2014): Eventual Consistent Databases: State of the Art, *Open Journal of Databases* **1**, p.26-41, ISSN 2199-3459 [http://www.ronpub.com/publications/OJDBv1i1n03\_Elbushra.pdf](http://www.ronpub.com/publications/OJDB-v1i1n03_Elbushra.pdf)

Fernandez, E.B., Washizaki, H., Yoshioka, N., VanHilst, M. (2011): An Approach to Model-based

Development of Secure and Reliable Systems, Availability, Reliability and Security (ARES), 2011 Sixth International Conference on , vol., no., pp.260-265, 22-26 Aug. 2011, doi: 10.1109/ARES.2011.45

Floridi, Luciano: Sextus Empiricus: The transmission and recovery of Pyrrhonism (American Philological Association, 2002) ISBN 0195146719

Fonseca, J; Vieira M,; Madeira H.: (2008) Online Detection of Malicious Data Access using DBMS

Auditing, Proceedings of the 2008 ACM symposium on Applied Computing, Fortaleza, Brazil, p. 10131020

Frank, L. (2011); , Architecture for Integrating Heterogeneous Distributed Databases Using Supplier

Integrated E-Commerce Systems as an Example, *Computer and Management (CAMAN), 2011*

*International Conference on* , pp.1-4, 19-21 May 2011, doi: 10.1109/CAMAN.2011.5778783

Garus, K. (2012): Version-Based Optimistic Concurrency Control in JPA/Hibernate,

<http://squirrel.pl/blog/2012/11/02/version-based-optimistic-concurrency-control-in-jpahibernate/>

Guenther, J. (2012): Optimistic Concurrency in Entity Framework,

[http://it.toolbox.com/blogs/aspnet-development/optimistic-concurrency-in-entity-framework50651](http://it.toolbox.com/blogs/aspnet-development/optimistic-concurrency-in-entity-framework-50651)

Haritsa, J. R., Carey, M. J., & Livny, M. (1990, December). Dynamic real-time optimistic concurrency control. In *Real-Time Systems Symposium, 1990. Proceedings., 11th* (pp. 94-103). IEEE.

IBM (2011) SolidDB v7.0 announcement, [http://www-](http://www-01.ibm.com/common/ssi/rep_ca/1/897/ENUS211-491/ENUS211-491.PDF)

[01.ibm.com/common/ssi/rep\_ca/1/897/ENUS211-491/ENUS211-491.PDF](http://www-01.ibm.com/common/ssi/rep_ca/1/897/ENUS211-491/ENUS211-491.PDF)

Iyer, S. (2009): New approaches to securing the database, *Network Security*, Volume 2009, Issue 11, November 2009, Pages 4-8, ISSN 1353-4858, 10.1016/S1353-4858(09)70120-5.

Kaspi, S., Venkataraman, S. (2014): Performance Analysis of Concurrency Control Mechamisms for OLTP Databases, *International Joiurnal of Information and Education Technology,* **4**, p. 313-318.

Kung, H.T., Robinson, J.T. (1981): On optimistic methods of concurrency control, *ACM Transactions of Database Stystems*, **6**, p. 213-226.

Laertius, Diogenes (3rd cent): The Lives and Opinions of Eminent Philosophers, trans. C. D. Yonge (London 1895)

Lessner, T., Laux, F., Connolly, T., Crowe, M. (2012): Transactional Composition and Concurrency Control in Disconnected Computing, *International Journal of Advances in Software*, **4**, p.442-460.

Mensacé, D. A., Nakamishi, T.: Optimistic versus Pessimistic ConcurrencyControl Mechanisms in Database Management Systems, *Information Systems* **7**, p.13-27

Moorthy, M. K., Seetharaman, A., Mohamed, Z., Gopalan, M., & San, L. H. (2011). The impact of information technology on internal auditing. *African Journal of Business Management*, *5*(9), 35233539.

Moreau, L. (2010): *The foundations for provenance on the Web*, (Now Publishers Inc)

Oh, S., Park, S (2003): Task-role-based access control model, *Information Systems* **28**, p.533-562.

Ouyang, X., Nellans, D., Wipfel, R., Flynn, D., Panda, D.K. (2011), Beyond block I/O: Rethinking traditional storage primitives, *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* , pp.301-311, IEEE, doi: 10.1109/HPCA.2011.5749738

Raab, F., Kohler, W., Shah, A. (2001): Overview of the TPC-C benchmark: The Order-Entry Benchmark, Transaction Processing Performance Council. www.tpc.org.

SQL2011: ISO/IEC 9075-2:2008 Information Technology – Database Languages – SQL – Part 2:

Foundation (SQL/Foundation); ISO/IEC 9075-4:2008: Information Technology – Database Languages

– SQL – Part 4: Persistent Stored Modules;ISO/IEC 9075-14:2008: Information technology —

Database languages — SQL — Part 14::XML-Related Specifications (SQL/XML) (International Standards Organisation, 2011)

SWI-Prolog, [www.swi-prolog.org.](http://www.swi-prolog.org/)

Thomson, A., Diamond, T., Weng, S. C., Ren, K., Shao, P., & Abadi, D. J. (2012, May). Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (pp. 1-12). ACM.

Tozun, P., Porobic, D., Branco M. (2012): Transaction Processing, <http://dias.epfl.ch/OLTP>

Wust, J., Boese, J. H., Renkes, F., Blessing, S., Krueger, J., & Plattner, H. (2012, October). Efficient logging for enterprise workloads on column-oriented in-memory databases. In *Proceedings of the 21st ACM international conference on Information and knowledge management* (pp. 2085-2089). ACM.

## Appendix 1.1 Using PyrrhoDB with Windows Forms

At one time Pyrrho also had DataAdapters, but now it stays with the simple ADO.NET approach so as to be more compatible with Linux. So for the final exercise in this lab, let us go back to Windows Forms, but avoid DataAdapters.

I downloaded the latest version of Pyrrho from [www.pyrrhodb.com,](http://www.pyrrhodb.com/) and extracted the files to C:\Temp. I created shortcuts on the desktop for the PyrrhoSvr application in C:\Temp\Pyrrho, and started up the server by right-clicking the shortcut. Leave the PyrrhoSvr window running.

To create the database use the command line tool PyrrhoCmd in C:\Temp\Pyrrho. Start a new ordinary Command Prompt window and change directory to \Temp\Pyrrho.

**PyrrhoCmd club**

At the SQL> prompt issue the following commands (the square brackets indicate a command that may continue over several lines of input):

**create table members(id int primary key,firstame char, surname char)**

**insert into members values(56,'Fred','Bloggs')**

**insert into members values(78,'Mary','Black')**

**[create table played(id int primary key,playedon date,winner int references members,loser int references members,agreed boolean)]**

**insert into played values(32,date'2021-01-09',56,78,false)**

You can verify this has worked in the usual way (select \* from members) etc.

Now start up Visual studio and make a new Windows Forms project.

This time add Reference and browse to C:\Temp\Pyrrho\PyrrhoLink.dll .

In the two cs files in the project delete the using lines for System.Threading.Tasks, and add

**using Pyrrho;**

As before, place a DataGridView in the design surface. Double-click the design surface (outside of the DataGridView).

In the class, declare PyrrhoConnect conn; In the Form1() constructor method, add the lines conn = new PyrrhoConnect(“Files=club”); conn.Open();

In the Form1\_Load method, give the code as follows:

var cmd = conn.CreateCommand(); cmd.CommandText = “select \* from members”; var rdr = cmd.ExecuteReader();

dataGridView1.ColumnCount = rdr.FieldCount; dataGridView1.ColumnHeadersVisible = true;

for (int j=0;j<rdr.FieldCount;j++) dataGridView1.Columns[j].Name = rdr.GetName(j);

while (rdr.Read())

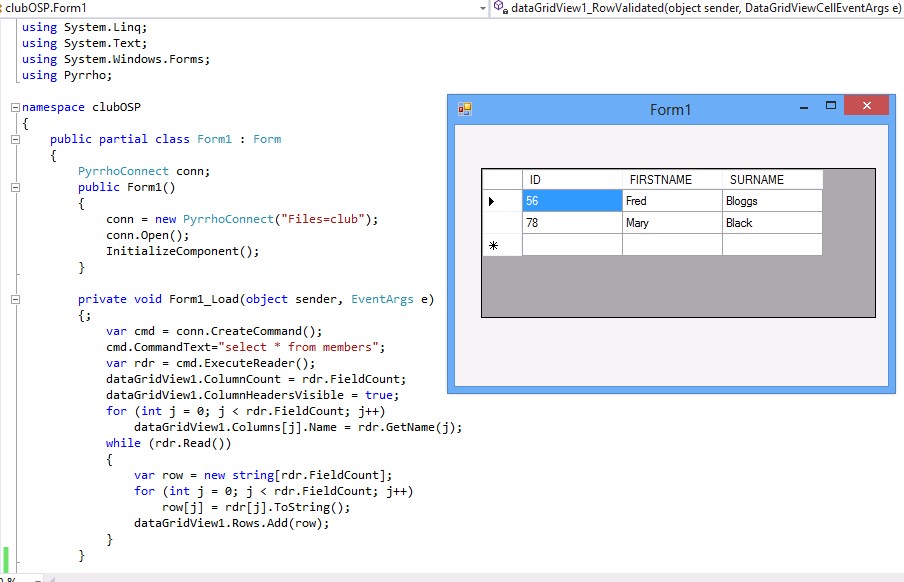
{

var row = new string[rdr.FieldCount];

for (int j=0;j<rdr.FieldCount;j++) row[j] = rdr[j].ToString();

dataGridView1.Rows.Add(row);

}



Stop the program. As before, we need to implement updates. In the designer, left-click the DataGridView, select the Events tab in the properties window, and double-click RowValidated, repeat for CellValidated and RowEnter. Add declarations in the Form1 class for bool newRow=false; int id; bool[] cellChanged = new bool[3]; Enter the following code in the event handlers:

private void dataGridView1\_RowValidated(object sender, DataGridViewCellEventArgs e)

{

var r = dataGridView1.Rows[e.RowIndex]; var cm = ""; if (newRow) {

cm = "insert into members values(" + r.Cells[0].Value.ToString() +

",'" + r.Cells[1].Value.ToString() + "','" + r.Cells[2].Value.ToString() + "')"; newRow = false;

} else { var s = ""; var c = ""; if (cellChanged[0])

{

s += c+"id=" + r.Cells[0].Value.ToString(); c = ",";

}

if (cellChanged[1])

{

s += c+"firstname='" + r.Cells[1].Value.ToString()+"'"; c = ",";

}

if (cellChanged[2])

{

s += c+"surname='" + r.Cells[2].Value.ToString()+"'"; c = ",";

} if (s != "")

cm = "update members set " + s + " where id=" + id;

} if (cm!="") try { conn.Act(cm); } catch { }

}

private void dataGridView1\_RowEnter(object sender, DataGridViewCellEventArgs e)

{

var r = dataGridView1.Rows[e.RowIndex]; if (r!=null && !newRow && r.Cells[0].Value!=null) int.TryParse(r.Cells[0].Value.ToString(), out id); for (int j = 0; j < 3; j++) cellChanged[j] = false;

}

private void dataGridView1\_CellValidated(object sender, DataGridViewCellEventArgs e)

{

cellChanged[e.ColumnIndex] = true; if (!newRow)

{

var r = dataGridView1.Rows[e.RowIndex];

int.TryParse(r.Cells[0].Value.ToString(), out id);

}

}

## A1.2 Using Java with Pyrrho

There are several analogues to Windows Forms in Java, such as Swing. But just for fun let’s do a very simple console application that retrieves data from a database.

You will need the Java Development Kit (JDK) installed on your computer. The Pyrrho distribution includes a library called OSPJC.

First set up the Path and CLASSPATH environment variables. (Start with

Computer>Properties>Advanced System Settings>Environment Variables, and use Edit and New respectively).

Add the following to Path:

;C:\Program Files (x86)\Java\jdk1.7.0\_09\bin

Set CLASSPATH to

.;C:\Program Files (x86)\Java\jdk1.7.0\_09\lib;C:\PyrrhoDB\OSP\OSPJC\bin

Alternatively, copy the org folder in OSPC\bin to the current directory and use –cp . with the javac and java commands below.

The following program assumes that database Temp has a table called A with columns B int and C char.

Use Notepad or something to make a test program JCTest.java (and remember to use All files when saving):

import org.pyrrhodb.Connection; import java.sql.Statement; import java.sql.ResultSet; public class JCTest {

public static void main(String args[]) throws Exception

{

Connection conn =

Connection.getConnection("localhost","Temp","COMP10059\\Student","Temp");

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("select \* from a"); for (boolean b = rs.first();b;b=rs.next())

{

System.out.println(""+rs.getInt("B")+"; "+rs.getString("C"));

}

}

}

Note that the public class has the same name as the file. You need to remember that Java is case sensitive while SQL isn’t, so use e.g. B and C even if you declared your table with lower-case names, e.g. create table a(b int, c char). Note that both pyrrhodb and java.sql define a Connection class, so you can’t use \* above.

Now compile it with

javac JCTest.java and run it with

java JCTest

As an exercise try out your own database. Note the column names in getInt() etc are case sensitive. You should be able to do update, delete etc using stmt.ExecuteNonQuery().

## A1.3 Using PHP with Pyrrho

Rename the C:\Program Files (x86)\PHP folder to PHP1. Copy the PHP folder in Pyrrho51.iso to \Program Files (x86).

Not in an Administrator command prompt, change direct to \OSP .

Install the OSPLink.dll in the gacutil:

“C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools”\gacutil –i OSPLink.dll

Register the assembly:

C:\Windows\Microsoft.NET\Framework\v4.0.30319\regasm /tlb OSPLink.dll

Now try the PHP version of the above program. Get Notepad to put the following in try.php:

<?php

$conn = new COM("OSPLink") or die("Cannot connect");

$conn->ConnectionString="Files=Temp";

$conn->Open();

$rdr=$conn->Execute("table a");

$row=$rdr->Read(); while(!is\_int($row))

{

print($row[0].': '.$row[1]."\r\n");

$row = $rdr->Read();

}

?>

Now at a command prompt try php try.php

# Appendix 2: DBTech’s Transactions mystery tour

I’ve included this as an appendix because I found it a genuinely astonishing experience to work with Martti Laiho on this DBTech tutorial (see www.DBTEchNet.org). We never really finished it but it led me to make a number of changes.

I’ve included the complete text of the tutorial as published. Don’t enter the text in blue though as it’s either comments or specific to the Linux version they were using.

DBTechNet / DBTech VET

Martti Laiho 2012-12-19

SQL Transactions

----------------

Appendix 1 / Pyrrho

Welcome to the "Transaction Mystery Tour" using Pyrrho DBMS. Following experiments with the Pyrrho transactions can be tested using the free DebianDB database laboratory of DBTechNet or on Windows.

----------------------- on Linux ------------------------ For the first steps on using Pyrrho in DebianDB see the "Quick Start Guide" at

http://www.dbtechnet.org/download/QuickStartToDebianDBv05.pdf

Downloading OSP.zip from http://pyrrhodb.uws.ac.uk/OSP.zip and unzipping it as

root@debianDB:/home/student/Downloads# unzip -Z OSP.zip -d /opt

Login as student/Student1

To start the Pyrrho server enter following commands in a Gnome Terminal window

# Pyrrho Server cd /opt/OSP mono OSP.exe

For client sessions in following experiments open 2 parallel Gnome Terminal windows

starting the client using following commands

# Pyrrho Client cd /opt/OSP

mono PyrrhoCmd.exe TestDB

----------------------- on Windows ----------------------

REM Pyrrho Server on Windows Unzip Pyrrho.zip to C: \Temp\Pyrrho

CD \Temp\Pyrrho

PyrrhoSvr

REM Pyrrho Client on Windows

CD \Temp\Pyrrho

Start PyrrhoSvr in one window and start pyrrhocmd in another.

**PyrrhoCmd testdb**

Here’s an example of the sort of thing that went wrong. Note that in the comments above, on Linux the instructions say to call the database TestDB. Whoever wrote the instructions clearly reasoned that Windows file names are not case sensitive. This is not actually true. If the database is really called TestDB, then the above command will result in lots of error messages! To check this, try creating a database in one case and then get the case wrong, as in:



Actually this used to cause all sorts of internal errors.

---------------------------------------------------------

Notes: As DEFAULT Pyrrho uses AUTOCOMMIT mode.

Pyrrho does not accept the word "WORK" after COMMIT.

SMALLINT is not a supported data type.

---------------------------------------------------------

-- A1.1 Experimenting with single explicit transactions

---------------------------------------------------------

-- Autocommit mode ?

CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si INTEGER);

INSERT INTO T (id, s) VALUES (1, 'first');

SELECT \* FROM T;

ROLLBACK; -- what happens?

Here’s another example, which I’ve left for your amusement. It you type exactly what is above, Pyrrho does nothing at all:

 because there is no transaction active, and the PyrrhoCmd program was confused by the strange text. If you just say rollback, PyrrhoCmd actually tries to say something useful:



I actually added the warnings such as “There is no current transaction” because of this tutorial. Even worse was the way that Martti and his students kept pasting in things like Let’s try and because this is not correct SQL the transaction would roll back.

SELECT \* FROM T;

---------------------------------------------------------

-- Transactional mode

-- testing also use of a DDL command in a transaction!

BEGIN TRANSACTION;

CREATE TABLE T2 (id INT NOT NULL PRIMARY KEY, s VARCHAR(30));

INSERT INTO T (id, s) VALUES (2, 'second');

INSERT INTO T2 (id, s) VALUES (2, 'second');

SELECT \* FROM T ;

ROLLBACK;

SELECT \* FROM T;

SELECT \* FROM T2; -- What has happened to T2 ?

COMMIT;

------------------------------------------------------------------ -- Testing if an error would lead to automatic rollback in Pyrrho?

------------------------------------------------------------------

-- Let's first implement the DUAL table like in Oracle -- for single line info queries

I’ve since included the from static trick in Pyrrho.

BEGIN TRANSACTION;

SELECT CURRENT\_DATE;

COMMIT;

--

BEGIN TRANSACTION;

INSERT INTO T (id, s) VALUES (2, 'Error test starts here');

-- division by zero should fail

SELECT (1/0) AS dummy FROM DUAL;

COMMIT;

BEGIN TRANSACTION;

-- updating an non-existing row

UPDATE T SET s = 'foo' WHERE id = 9999;

I’ve since included the 0 records message only very recently. I didn’t regard this as an error at all. Very reluctantly I implemented the NOT\_FOUND handler in stored procedures for this situation.

-- deleting an non-existing row

DELETE FROM T WHERE ida = 7777 ;

--

BEGIN TRANSACTION;

INSERT INTO T (id, s) VALUES (2, 'Error test starts here'); INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');

--

BEGIN TRANSACTION;

INSERT INTO T (id, s) VALUES (3, 'how about inserting too long string value?');

INSERT INTO T (id, s, si) VALUES (4, 'Integer overflow?',

9223372036854775808);

INSERT INTO T (id, s) VALUES (5, 'Testing an INSERT after errors');

SELECT \* FROM T;

COMMIT;

BEGIN TRANSACTION;

DELETE FROM T WHERE id > 1;

COMMIT;

------------------------------------------------------------

-- So, what have we found out of automatic rollback in Pyrrho ?

------------------------------------------------------------

-- Testing the database Recovery

BEGIN TRANSACTION;

INSERT INTO T (id, s) VALUES (9, 'Let''s see what happens if ..'); SELECT \* FROM T;

-- <close the client!!!>

#------------------------------------------------------------------

#-- Starting a new terminal window and connecting to our TestDB

#-- we can study what happened to our latest uncommitted transaction

#-- just by listing the contents of table T

# Pyrrho Client cd /opt/OSP mono PyrrhoCmd.exe TestDB SELECT \* FROM T;

COMMIT;

I don’t regard closing the client and restarting it as database Recovery – what do you think?

-----------------------------------------------------------

-- A1.2 Experimenting with Transaction Logic

-----------------------------------------------------------

-- Experiment 1: COMMIT and ROLLBACK -- in session of client A:

# Pyrrho Client cd /opt/OSP

mono PyrrhoCmd.exe TestDB

BEGIN TRANSACTION;

[CREATE TABLE Accounts ( acctID INTEGER NOT NULL PRIMARY KEY, balance INTEGER NOT NULL

CONSTRAINT uncreditable\_account CHECK (balance >= 0))]

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- let''s try the bank transfer

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;

UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

SELECT \* FROM Accounts;

ROLLBACK;

-----------------------------------------------------------

-- Experiment 2: Transaction logic

-- following commands are "sensitive" to previous balance values BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;

I think you should do something else just here…

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;

COMMIT;

SELECT \* FROM Accounts; -- Do we have a problem ?

-- Experiment 2b Transaction logic

-- restoring the original contents

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

#------------------- Advanced topic -----------------------

# Experiment with IF structure

# See the Stored Procedure at the end of this file ?

#------------------- end of the Advanced topic ----------------------- ------------------------------------------------------------------------

-- A1.3 Experimenting with Concurrent Transactions

-- For concurrency experiments we will open two parallel

-- sessions

-- Note: SERIALIZABLE is the default and only supported

-- isolation level in Pyrrho! So some of our test cases are

-- not relevalnt in Pyrrho.

-- To start with fresh contents we enter following commands on a session

-- restoring the original contents

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- Experiment 3 Simulating "Lost update problem"

-- 1. client A starts

BEGIN TRANSACTION;

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 2. client B starts

BEGIN TRANSACTION;

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 3. client A continues

UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. client B continues

UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;

-- 5. client A continues

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

COMMIT;

-- 6. client B continues

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

COMMIT;

-- Note: In this experiment we did not have the real "Lost Update Problem",

-- but after A commits its transaction B usually can proceed and overwrite

-- the update made by A. We call this reliability problem as

-- "Blind Overwriting" or "Dirty Write".

-- The optimistic concurrency control of Pyrrho solves the Dirty Write Problem!!!

---------------------------------------------------------------------------

-

-- Experiment 4 "Lost Update Problem" fixed by locks

-- (competition on a single resource)

--

-- Note: this experiment is not relevant with Pyrrho!

--

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- client A starts

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- client B starts

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- client A continues

UPDATE Accounts SET balance = balance - 200 WHERE acctID = 101;

-- client B continues

UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;

-- the client which survived will commit

SELECT acctID, balance FROM Accounts WHERE acctID = 101;

COMMIT;

-----------------------------------------------------------

-- Experiment 5

---

-- Competition on two resources in different order

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- 1. Client A starts

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;

-- 2. Client B starts

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 200 WHERE acctID = 202;

-- 3. Client A continues

UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

-- 4. Client B continues

UPDATE Accounts SET balance = balance + 200 WHERE acctID = 101;

-- 5. Client A continues

COMMIT;

-- 6. Client B continues

COMMIT;

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- In the following we will experiment concurrency anomalies i.e.

-- data reliability risks known by ISO SQL standard

-- can we identify those?

-- how can we fix the experiments?

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Experiment 6 Dirty Read ?

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- 1. client A starts

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;

UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

SELECT \* FROM Accounts;

-- 2. Client B starts - will this be a Dirty Read ?

BEGIN TRANSACTION;

SELECT \* FROM Accounts;

COMMIT;

-- 3. Client A continues

ROLLBACK;

SELECT \* FROM Accounts;

-----------------------------------------------------------

-- Experiment 7 Unrepeatable Read ?

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- 1. client A starts

BEGIN TRANSACTION;

-- Accounts having balance > 500 euros

SELECT \* FROM Accounts WHERE balance > 500;

-- 2. Client B starts

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance - 400 WHERE acctID = 101;

UPDATE Accounts SET balance = balance + 400 WHERE acctID = 202;

SELECT \* FROM Accounts;

COMMIT;

-- 3. Client A continues

-- Let's see the results after client B's transaction:

SELECT \* FROM Accounts WHERE balance > 500;

COMMIT;

-----------------------------------------------------------

-- Experiment 8 Insert Phantom ?

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- 1. client A starts

BEGIN TRANSACTION;

-- Accounts having balance > 1000 euros:

SELECT \* FROM Accounts WHERE balance > 1000;

-- 2. Client B starts

BEGIN TRANSACTION;

INSERT INTO Accounts (acctID,balance) VALUES (303,3000);

SELECT \* FROM Accounts;

COMMIT;

-- 3. Client A continues

-- Let’s see the results:

SELECT \* FROM Accounts WHERE balance > 1000;

COMMIT;

-----------------------------------------------------------

-- Experiment 9 Update Phantom ?

-- First restoring the original contents by client A

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

-- 1. client A starts

BEGIN TRANSACTION;

-- Accounts having balance > 1000 euros:

SELECT \* FROM Accounts WHERE balance > 1000

-- 2. Client B starts

BEGIN TRANSACTION;

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 101;

SELECT \* FROM Accounts;

COMMIT;

-- 3. Client A continues

-- Let’s see the results:

SELECT \* FROM Accounts WHERE balance > 1000;

COMMIT;

===============================================================

Note1: Multi-line commands need to be enclosed in [ ] brackets

Note2: COMMIT is not allowed in a procedure/function

-- The following test is copied from the Pyrrho manual and works fine

CREATE TABLE A (A2 CHAR(1));

[CREATE FUNCTION GATHER1() RETURNS CHAR

BEGIN

DECLARE C CURSOR FOR SELECT A2 FROM A;

DECLARE DONE BOOLEAN DEFAULT FALSE;

DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET DONE=TRUE;

DECLARE A CHAR DEFAULT '';

DECLARE P CHAR;

OPEN C;

REPEAT

FETCH C INTO P;

IF NOT DONE THEN

IF A = '' THEN

SET A = P

ELSE

SET A = A || ', ' || P

END IF

END IF

UNTIL DONE END REPEAT;

CLOSE C;

RETURN A;

END;]

-------------------------------- BEGIN TRANSACTION;

drop function BankTransfer (int,int,int);

[CREATE FUNCTION BankTransfer

(IN fromAcct INT,

IN toAcct INT,

IN amount INT)

RETURNS VARCHAR(100)

BEGIN

DECLARE msg VARCHAR(100) ;

DECLARE NOT\_FOUND BOOLEAN DEFAULT FALSE;

DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET NOT\_FOUND=TRUE;

SET msg = 'ok';

UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;

IF (NOT\_FOUND) THEN

SET msg = '\* Unknown from account ' || CAST(fromAcct AS

VARCHAR(10))

ELSE

BEGIN

UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;

IF (NOT\_FOUND) THEN

SET msg = '\* Unknown from account ' || CAST(toAcct AS

VARCHAR(10))

ELSE

SET msg = 'OK'

END IF

END

END IF;

RETURN msg

|| ' toAcct: ' || CAST(toAcct AS VARCHAR(10))

|| ' amount: ' || CAST(amount AS VARCHAR(10)) ; END;]

COMMIT;

BEGIN TRANSACTION;

DELETE FROM Accounts;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT \* FROM Accounts;

COMMIT;

BEGIN TRANSACTION;

SELECT BankTransfer (101, 202, 100) FROM DUAL;

SELECT \* FROM Accounts;

ROLLBACK;

BEGIN TRANSACTION;

SELECT BankTransfer (100, 202, 100) FROM DUAL;

SELECT \* FROM Accounts;

ROLLBACK;

BEGIN TRANSACTION;

SELECT BankTransfer (101, 200, 100) FROM DUAL;

SELECT \* FROM Accounts;

ROLLBACK;

BEGIN TRANSACTION;

SELECT BankTransfer (101, 202, 2000) FROM DUAL;

SELECT \* FROM Accounts;

ROLLBACK;

SQL> BEGIN TRANSACTION;

SQL-T>DELETE FROM Accounts;

SQL-T>INSERT INTO Accounts (acctID,balance) VALUES (101,1000);

SQL-T>INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SQL-T>SELECT \* FROM Accounts;

|------|-------|

|ACCTID|BALANCE|

|------|-------|

|101 |1000 |

|202 |2000 |

|------|-------|

SQL-T>COMMIT;

SQL> BEGIN TRANSACTION;

SQL-T>SELECT BankTransfer (101, 202, 100) FROM DUAL;

|-----|

|Col$1|

|-----|

|OK |

|-----|

SQL-T>SELECT \* FROM Accounts;

Internal error: Object reference not set to an instance of an object.

The transaction has been rolled back

SQL> ROLLBACK;

There is no current transaction

SQL>

SQL> SELECT \* FROM Accounts;

|------|-------|

|ACCTID|BALANCE|

|------|-------|

|101 |1000 |

|202 |2000 |

|------|-------|

SQL> BEGIN TRANSACTION;

SQL-T>SELECT BankTransfer (101, 202, 100) FROM DUAL; |-----|

|Col$1|

|-----|

|OK |

|-----|

SQL-T>SELECT \* FROM Accounts;

Internal error: Object reference not set to an instance of an object.

The transaction has been rolled back

SQL> BEGIN TRANSACTION;

SQL-T>SELECT BankTransfer (100, 202, 100) FROM DUAL;

|--------------------------|

|Col$1 |

|--------------------------|

|\* Unknown from account 100|

|--------------------------|

SQL-T>SELECT \* FROM Accounts;

Internal error: Object reference not set to an instance of an object.

The transaction has been rolled back

SQL> SELECT \* FROM Accounts;

|------|-------|

|ACCTID|BALANCE|

|------|-------|

|101 |1000 |

|202 |2000 |

|------|-------|

SQL> BEGIN TRANSACTION;

SQL-T>SELECT BankTransfer (101, 200, 100) FROM DUAL;

|-----|

|Col$1|

|-----|

|OK |

|-----|

SQL-T>SELECT \* FROM Accounts;

Internal error: Object reference not set to an instance of an object.

The transaction has been rolled back

SQL> SELECT \* FROM Accounts;

|------|-------|

|ACCTID|BALANCE|

|------|-------|

|101 |1000 |

|202 |2000 |

|------|-------|

SQL> BEGIN TRANSACTION;

SQL-T>SELECT BankTransfer (101, 202, 2000) FROM DUAL;

|-----|

|Col$1|

|-----|

|OK |

|-----|

SQL-T>SELECT \* FROM Accounts;

Internal error: Object reference not set to an instance of an object.

The transaction has been rolled back

SQL> SELECT \* FROM Accounts;

|------|-------|

|ACCTID|BALANCE|

|------|-------|

|101 |1000 |

|202 |2000 |

|------|-------|

SQL>

-- Unsolved problems:

-- Why the function return an empty string ?

[CREATE PROCEDURE TransferTest

(fromAcct INT, toAcct INT, amount INT)

IF BankTransfer (101, 202, 100) = 'OK' THEN

COMMIT ELSE ROLLBACK

END IF;]

BEGIN TRANSACTION;

SELECT BankTransfer (101, 202, 100) FROM DUAL;

# Appendix 3: Roles and Security

## A3.1: The sporting club

1. Recall that in Lab 1 we set up the following tables in a database called club:

Members: (id int primary key, firstname char)

Played: (id int primary key, winner int references members, loser int references members, agreed boolean)

with the following commands

create table members(id int primary key,firstame char, surname char) insert into members values(56,’Fred’,’Bloggs’) insert into members values(78,’Mary’,’Black’)

[create table played(id int primary key,playedon date,winner int references members,loser int references members,agreed boolean)]

insert into played values(32,date’2014-01-09’,56,78,false)

1. For simplicity we give everyone select access to both these tables.

Grant select on members to public

Grant select on played to public

1. Although Pyrrho records which user makes changes, it will save time if users are not allowed to make arbitrary changes to the Played table. Instead we will have procedure Claim(won,beat) and Agree(id), so that the Agree procedure is effective only when executed by the loser. With some simple assumptions on user names, the two procedures could be as simple as:

[Create procedure claim(won int,beat int) insert into played(winner,loser) values(claim.won,claim.beat)]

[Create procedure agree(p int) update played set agreed=true where winner=agree.p and not agreed and loser in (select m.id from members m where current\_user like ('%'||firstname))]

1. We want all members of the club to be able to execute these procedures. We could simply grant execute on these procedures to public. However, it is better practice to grant these permissions instead to a role (say, membergames) and allow any member to use this role:

Create role membergames 'Matches between members for ranking purposes'

Grant execute on procedure claim(int,int) to role membergames

Grant execute on procedure agree(int) to role membergames

Grant membergames to public

1. Check the above procedures work using some sample data, for example call claim(56,78) . Alter the Claim procedure to record the current\_date as the date of the match. (Alter procedure claim…)
2. Now create users Fred and Mary (say) in Windows using the Control Panel.
3. Log in as Fred and check that Fred select \* from the tables but not make changes directly. Get Fred to claim a win.
4. Log in as Mary and accept.

This example could be extended by considering the actual use made of the Played table in calculating the current rankings, etc.

## A3.2: A forensic investigation

Examine the log of the club database as described in the lecture. Look at the last few transactions. What are the user and role? Try writing select statements in the manner described in the lecture to find the previous values of things that have been changed.

Also look at the (current) “Role$Privilege” table.

## A3.3: Some more PyrrhoDB stuff

I will be updating the OSPnn.iso files every so often, so always use the most recent (highest numbered). Use this to *overwrite* the folder you have previously installed on your virtual machine (and then you will be able to use the new binaries while retaining any databases you have developed). When Pyrrho starts up it gives the date of the server.

There is a manual in the distribution on your virtual machine and from the PyrrhoDB website [http://pyrrhodb.com](http://pyrrhodb.com/) . A lot of the same material is available in the Sample code section of the website, including full reference details for the SQL that Pyrrho uses. There is also the blog, which is accessible from the website. Here are a few pointers that you might find useful in your coursework:

1. You can use the pyrrhodb command line to import data from a text file. You first need to create a table with suitable columns to receive the data. The latest version of the manual says (sec 4.3.6)

A textfile containing rows for a table can similarly be added using a command such as

#### insert into directors values ~rowsfile

Simple data can be provided in a csv or similar file. The first line containing column headings and exposed spaces in the file are ignored. Data items in the given file are separated by exposed commas or tabs. Rows are parenthesized groups (optionally separated by commas), or provided without parentheses but separated by exposed semicolons or newlines. Characters such as commas etc are not considered to be separators if they are within a quoted string or a structure enclosed in braces, parentheses, brackets, or pointy brackets.

Of course if the file is really called rowsfile.txt you need to say so: ~rowsfile.txt .

1. You can also use the REST service (there is a RESTclient in the distribution) to POST csv data to a table. However, this only works for tables that have a primary key. For this service, don’t post the first line of a csv file since this contains column headers. (Don’t forget to run the OSP server as administrator: if you get the Access denied message on start-up, the REST service will not be available.)
2. You can copy data from one table to another by using the INSERT..SELECT combination as in

[insert into mytable select somecolumn,anothercolumn from anothertable]

1. You can get PyrrhoDB to draw charts if you want. You can add metadata to tables and columns. There is a [blog post](http://pyrrhodb.blogspot.co.uk/search?updated-min=2011-01-01T00:00:00-08:00&updated-max=2012-01-01T00:00:00-08:00&max-results=16) about this but we don’t use the SERIES keyword any more. Things like HISTOGRAM, PIE, X, Y are keywords. For example (assuming table mytable has suitable data columns mycaptions and mycolumn)

alter table mytable histogram alter table mytable alter column mycaptions x alter table mytable alter column mycolumn y Use the HTTP/REST service to view the histogram in a browser: the URL you want is of form [http://localhost:8180/mydb/myrole/mytable.](http://localhost:8180/mydb/myrole/mytable) If your database has no roles defined yet, the default role has the same name as the database.

1. The rules for case-sensitivity come separately from Windows and SQL, and are a little confusing.

In the command line and the URL, the database file name (and thus the name of Pyrrho’s default Role) are case sensitive. By default, lower case letters remain in lower case. So if you create a database called mydb, the actual file name on the server will be mydb.osp. If you need to use these lower case identifiers in SQL, you need to enclose them in double quotes. (The default role for example will be “mydb”.)

In SQL, the names of database objects that you create are not case sensitive unless they are enclosed in double quotes. In the absence of double quotes, lower case letters will be turned into upper case. If you create a table called mytable in SQL, the actual name will be MYTABLE, and so if it is mentioned in a URL, you will need to use capitals.

Se we end up with URL’s like <http://localhost:8180/mydb/mydb/MYTABLE>.

1. You can view the metadata such as PIE, HISTOGRAM etc for database objects such as tables, views, procedures and table columns by using the Role$Object sytem table (on the command line you would say table “Role$Object”). For columns of views or stored functions there is another system table called “Role$Subobject” which does not seem to be in the manual just now .

You should be able to modify metadata for objects and subobjects by using the keywords ADD and DROP (you don’t really need ADD as it is the default behaviour) e.g. alter table mytable drop histogram alter view myview alter someviewcolumn drop x

alter function myfunc(int) alter someresultcolumn y etc.

1. There are ways of supplying style sheets hidden in the Description metadata string. The manual’s notes on Metadata read (sec 7.2):

Metadata = ATTRIBUTE | CAPTION | ENTITY | HISTOGRAM | LEGEND | LINE | POINTS | PIE | X | Y | CAPPED | USEPOWEROF2SIZES | BACKGROUND | DROPDUPS | SPARSE | MAX’(‘int’)’ | SIZE’(‘int’)’ | string | iri .

The Metadata flags and the string “constraint” are Pyrrho extensions., Attribute and Entity affect XML output in the role, Caption, Histogram, Legend, Line, Points, Pie, X and Y affect HTML output in the role. Other metadata keywords are for MongoDB. The string is for a description, but if it begins with a < it will be included in the HTML so that it can supply CSS style or script. If the string begins with <? it should be an XSL transform, and Pyrrho will generate and then transform the return data as XML. HTML responses will have JavaScript added to draw the data visualisations specified.

# Appendix 4: Pyrrho SQL Syntax

A clickable HTML version of these details is available at [www.pyrrhodb.com.](http://www.pyrrhodb.com/)

In this section capital letters indicate key words: many of these words are reserved words (see the website for a list of reserved words). Tokens such as id, int, string are shown as all lower case words. Mixed case is used for grammar symbols defined in the following productions. The characters = . [ ] { } are part of the production syntax. Characters that appear in the input are enclosed in single quotes, thus ‘,’ . Where an identifier representing an object name is required, and the type of object is not obvious from the context, locutions such as Role\_id are used.

There are three tokens: xmlname, iri and xml, which are used in XPath below, which are extensions to SQL2011. These tokens are not enclosed in single or double quotes, but may contain string literals that are enclosed in quotes. xmlname represents a case-sensitive sequence of letters and digits, iri is an IRI enclosed in <> and xml represents any Xml content not including an exposed , ] or ). In SQL all string literals are enclosed in single quotes, case-sensitive identifiers or containing special characters are enclosed in double quotes.

## A4.1 Statements

Sql = SqlStatement [‘;’] .

SqlStatement = Alter

| BEGIN TRANSACTION [WITH PROVENANCE string ]

| Call

| COMMIT

| CreateClause

| CursorSpecification

| DeleteSearched

| DropStatement

| Grant

| Insert

| Rename

| Revoke

| ROLLBACK

| SET AUTHORIZATION ‘=’ CURATED

| SET PROFILING ‘=’ Value

| SET ROLE id [FOR DATABASE id]

| SET TIMEOUT ‘=’ int

| UpdateSearched

| HTTP HttpRest .

The above statements can be issued at command level. You SELECT multiple rows from tables using the CursorSpecification. Inside procedures and functions there is a different set. (Note that “direct SQL” statements are in both lists.)

Apart from SET ROLE, the above SET statements are available only to the database owner. SET AUTHORIZATION = CURATED makes all further transaction log information PUBLIC (it is not reversible).

Statement = Assignment

| Call

| CaseStatement

| Close

| CompoundStatement

| BREAK

| Declaration

| DeletePositioned

| DeleteSearched

| Fetch

| ForStatement

| GetDiagnostics

| IfStatement

| Insert

| ITERATE label

| LEAVE label

| LoopStatement

| Open

| Repeat

| RETURN Value

| ROLLBACK

| SelectSingle

| Signal

| UpdatePositioned

| UpdateSearched

| While

| HTTP HttpRest .

HttpRest = (ADD|UPDATE) url\_Value data\_Value [AS mime\_string]

| DELETE url\_Value .

Here ADD and UPDATE are used as the SQL analogues of POST and PUT. For Http GET see Value . By design in Pyrrho, the execution of ROLLBACK causes immediate exit of the current transaction with SQLSTATE 25005, and premature COMMIT is not supported, so that while ROLLBACK is in both lists above, COMMIT is nly in one.

## A4.2 Data Definition

As is usual for a practical DBMS Pyrrho’s Alter statements are richer than SQL2011.

Alter = ALTER DOMAIN id AlterDomain

| ALTER FUNCTION id ‘(‘ Parameters ‘)’ RETURNS Type AlterBody

| ALTER PROCEDURE id ‘(‘ Parameters ‘)’ AlterBody

| ALTER Method AlterBody

| ALTER TABLE id AlterTable

| ALTER TYPE id AlterType

| ALTER VIEW id AlterView .

Procedures, functions and methods are distinguished by their name and arity (number of parameters) .

Method = MethodType METHOD id ‘(‘ Parameters ’)’ [RETURNS Type] [FOR id].

Parameters = Parameter {‘,’ Parameter } .

Parameter = id Type .

The specification of IN, OUT, INOUT and RESULT is not (yet) supported.

MethodType = [ OVERRIDING | INSTANCE | STATIC | CONSTRUCTOR ] .

The default method type is INSTANCE. All OVERRIDING methods are instance methods.

AlterDomain = SET DEFAULT Default

|  |  |
| --- | --- |
| | | DROP DEFAULT |
| | | TYPE Type |
| | | AlterCheck . |
| AlterBody = | AlterOp { ‘,’ AlterOp } . |
| AlterOp = | TO id |
| | | Statement |
| | | [ADD|DROP] { Metadata } . |
| Default = | Literal | DateTimeFunction | CURRENT\_USER | CURRENT\_ROLE | NULL | ARRAY'('')' |

| MULTISET'('')' .

AlterCheck = ADD CheckConstraint

| [ADD|DROP] { Metadata } | DROP CONSTRAINT id .

Note that anonymous constraints can be dropped by finding the system-generated id in the Sys$TableCheck, Sys$ColumnCheck or Sys$DomainCheck table (see section 8.1).

CheckConstraint = [ CONSTRAINT id ] CHECK ‘(‘ [XMLOption] SearchCondition ‘)’ .

XMLOption = WITH XMLNAMESPACES ‘(‘ XMLNDec {‘,’ XMLNDec } ‘)’ .

XMLNDec = (string AS id) | (DEFAULT string) | (NO DEFAULT) .

The following standard namespaces and prefixes are predefined:

‘http://www.w3.org/1999/02/22-rdf-syntax-ns#’ AS rdf

‘http://www.w3.org/2000/01/rdf-schema#’ AS rdfs

‘http://www.w3.org/2001/XMLSchema#’ AS xsd

‘http://www.w3.org/2002/07/owl#’ AS owl

AlterTable = TO id

| ADD ColumnDefinition

| ALTER [COLUMN] id AlterColumn

| DROP [COLUMN] id DropAction

| (ADD|DROP) (TableConstraintDef | VersioningClause)

| ADD TablePeriodDefinition [AddPeriodColumnList]

| ALTER PERIOD id TO id

| DROP TablePeriodDefinition

| AlterCheck

| [ADD|DROP] { Metadata }.

AlterColumn = TO id

| POSITION int

| (SET|DROP) ((NOT NULL)|ColumnConstraint )

| AlterDomain

| GenerationRule

| [ADD|DROP] { Metadata } .

When columns are renamed, Pyrrho cascades the change to SQL referring to the columns.

AlterType = TO id

| ADD ( Member | Method )

| DROP ( Member\_id | Routine)

| Representation

| [DROP] { Metadata }

| ALTER Member\_id AlterMember .

Other details of a Method can be changed with the ALTER METHOD statement (see Alter above).

Member = id Type [DEFAULT Value] Collate .

AlterMember = TO id

| [DROP] { Metadata }

| TYPE Type

| SET DEFAULT Value

| DROP DEFAULT .

AlterView = SET (INSERT|UPDATE|DELETE|) TO SqlStatement

| SET SOURCE TO QueryExpression

| TO id

| [ADD|DROP] { Metadata } .

Metadata = ATTRIBUTE | CAPTION | ENTITY | HISTOGRAM | LEGEND | LINE | POINTS | PIE | X |

Y | CAPPED | USEPOWEROF2SIZES | BACKGROUND | DROPDUPS | SPARSE | MAX’(‘int’)’ | SIZE’(‘int’)’ | string | iri .

The Metadata flags and the string “constraint” are Pyrrho extensions., Attribute and Entity affect

XML output in the role, Histogram, Legend, Line, Points, Pie (for table, view or function metadata), Caption, X and Y (for column or subobject metadata) affect HTML output in the role. HTML responses will have JavaScript added to draw the data visualisations specified. Other metadata keywords are for MongoDB. The string is for a description, and for X and Y columns is used to label the axes of charts. If the table’s description string begins with a < it will be included in the HTML so that it can supply CSS style or script. If the table’s string metadata begins with <? it should be an XSL transform, and Pyrrho will generate and then transform the return data as XML.

AddPeriodColumnList = ADD [COLUMN] Start\_ColumnDefinition ADD [COLUMN] End\_ColumnDefinition .

Create = CREATE ROLE id [Description\_string]

| CREATE DOMAIN id [AS] DomainDefinition

| CREATE FUNCTION id ‘(‘ Parameters ‘)’ RETURNS Type Statement

| CREATE ORDERING FOR UDType\_id (EQUALS ONLY|ORDER FULL) BY Ordering

| CREATE PROCEDURE id ‘(‘ Parameters ‘)’ Statement

| CREATE Method Statement

| CREATE TABLE id TableContents [UriType] {Metadata}

| CREATE TRIGGER id (BEFORE|AFTER) Event ON id [ RefObj ] Trigger

| CREATE TYPE id ((UNDER id )|AS Representation)[ Method {‘,’ Method} ]

| CREATE ViewDefinition

| CREATE XMLNAMESPACES XMLNDec { ‘,’ XMLNDec }

| CREATE INDEX Table\_id Name\_id [UNIQUE] Cols Metadata .

Method bodies in SQL2011 are specified by CREATE METHOD once the type has been created...In Pyrrho types UNDER or Representation must be specified (not both). CREATE XMLNAMESPACES is for creating a persistent association of namespace uris with identifiers.

Representation = (StandardType|Table\_id|‘(‘ Member {‘,’ Member }’)’)[UriType] {CheckConstraint} . UriType = [Abbrev\_id]‘^^’( [Namespace\_id] ‘:’ id | uri ) .

Syntax with UriType is a Pyrrho extension. Abbrev\_id can only be supplied within a CREATE DOMAIN statement. See section 5.2.1.

DomainDefinition = StandardType [UriType] [DEFAULT Default] { CheckConstraint } Collate .

Ordering = (RELATIVE|MAP) WITH Routine | STATE .

TableContents = ‘(‘ TableClause {‘,’ TableClause } ‘)’ { VerisoningClause } | OF Type\_id [‘(‘ TypedTableElement {‘,’ TypedTableElement} ‘)’]

| AS Subquery .

VersioningClause = WITH (SYSTEM|APPLICATION) VERSIONING .

WITH APPLICATION VERSIONING is Pyrrho specific: see section 5.2.3.

TableClause = ColumnDefinition {Metadata} | TableConstraint | TablePeriodDefinition .

ColumnDefinition = id Type [DEFAULT Default] {ColumnConstraint|CheckConstraint} Collate

| id GenerationRule

| id Table\_id ‘.’ Column\_id.

The last version is a convenience form for lookup tables, e.g. if a.b has domain int then a.b is a shorthand for int check (value in (select b from a)).

GenerationRule = GENERATED ALWAYS AS ‘(‘Value‘)’ [ UPDATE ‘(‘ Assignments ‘)’ ] | GENERATED ALWAYS AS ROW (START| END) .

The update option here is an innovation in Pyrrho.

ColumnConstraint = [CONSTRAINT id ] ColumnConstraintDef .

ColumnConstraintDef = NOT NULL

| PRIMARY KEY

| REFERENCES id [ Cols ] [USING ‘(‘Values’)’] { ReferentialAction }

| UNIQUE

| DEFAULT Value | GenerationRule .

The Using expression here is an extension to SQL2011 behaviour. See section 5.2.2. A column default value overrides a domain default value.

TableConstraint = [ CONSTRAINT id ] TableConstraintDef .

TableConstraintDef= UNIQUE Cols

| PRIMARY KEY Cols

| FOREIGN KEY Cols REFERENCES id [Cols] [USING ‘(‘Values’)’] { ReferentialAction } .

The Using expression here is an extension to SQL2011 behaviour. See section 5.2.2.

TablePeriodDefinition= PERIOD FOR PeriodName ‘(‘ Column\_id ‘,’ Column\_id ‘)’ .

PeriodName = SYSTEM\_TIME | id .

TypedTableElement = ColumnOptionsPart | TableCnstraint .

ColumnOptionsPart = id WITH OPTIONS ‘(‘ ColumnOption {‘,’ ColumnOption } ‘)’ .

ColumnOption = (SCOPE Table\_id) | (DEFAULT Value) | ColumnConstraint .

Values = Value {‘,’ Value } .

Cols = ‘(‘ColRef { ‘,’ColRef } [‘,’ PERIOD ApplicationTime\_id ] ‘)’.

The period syntax here can only be used in a foreign key constraint declaration where both tables have application time period definitions, and allows them to be matched up.

ColRef = Column\_id { ‘.’ Field\_id } [DESC] .

The Field\_id syntax is Pyrrho specific and can be used to reference fields of structured types or documents. The desc option is for the MongoDB Create Index syntax and is allowed only for document field references.

ReferentialAction = ON (DELETE|UPDATE) (CASCADE| SET DEFAULT|RESTRICT) .

The default ReferentialAction is RESTRICT. Constraints are retrospective: they cannot be applied if existing data conflicts with them A new default value will be applied to any existing null values, but dropping or changing a default value has no retrospective effect (since there are no null values to apply it to).

ViewDefinition = VIEW id AS QueryExpression [UPDATE SqlStatement] [INSERT SqlStatement] [DELETE SqlStatement] {Metadata} .

This is an extension to SQL2011 syntax to provide simpler mechanisms for indirect tables. Note that all of these can use Web services to access remote data.

TriggerDefinition = TRIGGER id (BEFORE|(INSTEAD OF)|AFTER) Event ON id [RefObj] Trigger .

In Pyrrho, triggers can be defined on views: this is very useful when used with roles.

Event = INSERT | DELETE | (UPDATE [ OF id { ‘,’ id } ] ) .

RefObj = REFERENCING { (OLD|NEW)[ROW|TABLE][AS] id } .

In this syntax, the default is ROW; TABLE cannot be specified for a BEFORE trigger; OLD cannot be specified for an INSERT trigger; NEW cannot be specified for a DELETE trigger.

Trigger = FOR EACH (ROW|STATEMENT)[TriggerCond](Statement|(BEGIN ATOMIC Statements END))

.

TriggerCond = WHEN ‘(‘ SearchCondition ‘)’ .

DropStatement = DROP DropObject DropAction .

DropObject = ObjectName

| ORDERING FOR id

| ROLE id

| TRIGGER id

| XMLNAMESPACES (id|DEFAULT) {‘,’ (id|DEFAULT) } | INDEX id .

DropAction = | RESTRICT | CASCADE .

The default DropAction is RESTRICT.

Rename =SET ObjectName TO id .

## A4.3 Access Control

Grant = GRANT Privileges TO GranteeList [ WITH GRANT OPTION ]

| GRANT Role\_id { ‘,’ Role\_id } TO GranteeList [ WITH ADMIN OPTION ] .

Grant can only be used in single-database connections (section 3.4). For roles see section 5.5.

Revoke = REVOKE [GRANT OPTION FOR] Privileges FROM GranteeList

| REVOKE [ADMIN OPTION FOR] Role\_id { ‘,’ Role\_id } FROM GranteeList .

Revoke can only be used in single-database connections. Revoke withdraws the specified privileges in a cascade, irrespective of the origin of any privileges held by the affected grantees: this is a change to SQL2011 behaviour. (See also sections 5.5 and 7.13.) Privileges = ObjectPrivileges ON ObjectName.

ObjectPrivileges = ALL PRIVILEGES | Action { ‘,’ Action } .

Action = SELECT [ ‘(‘ id { ‘,’ id } ‘)’ ]

| DELETE

| INSERT [ ‘(‘ id { ‘,’ id } ‘)’ ]

| UPDATE [ ‘(‘ id { ‘,’ id } ‘)’ ]

| REFERENCES [ ‘(‘ id { ‘,’ id } ‘)’ ]

| USAGE

| TRIGGER | EXECUTE | OWNER .

The owner privilege (Pyrrho-specific) can only be granted by the owner of the object (or the database) and results in a transfer of ownership of that object to a single user or role (not PUBLIC).. Ownership always implies grant option for the owner privilege. References here can be to columns, methods, fields or properties depending on the type of object referenced by the objectname.

ObjectName = TABLE id

| DOMAIN id

| TYPE id

| Routine

| VIEW id

| DATABASE .

The Pyrrho-specific OWNER Privilege is the only grantable privilege on the Pyrrho-specific DATABASE ObjectName, and results in a transfer of ownership of the database to the grantee, who must be a user (not a role).

GranteeList = PUBLIC | Grantee { ‘,’ Grantee } .

Grantee = [USER] id | ROLE id .

See section 5.5 for the use of roles in Pyrrho.

Routine = PROCEDURE id [‘(‘Type, {‘,’ Type }’)’]

| FUNCTION id [‘(‘Type, {‘,’ Type }’)’]

| [ MethodType ] METHOD id [‘(‘Type, {‘,’ Type }’)’] [FOR id ]

| TRIGGER id .

## A4.4 Type

Type = (StandardType | DefinedType | Domain\_id | Type\_id)[UriType] .

StandardType = BOOLEAN | CharacterType | FloatType | IntegerType | LobType | NumericType | DateTimeType | IntervalType | XML | PASSWORD | DOCUMENT | DOCARRAY | OBJECT .

The last four types are Pyrrho-specific: Password values show as \*\*\*\*\*\*\*, Document is as in http://bsonspec.org, DocArray is for the array variant used in Bson, and Object is ObjectId as in MongoDB, Documents and ObjectIds are transmitted to clients as subtypes of byte[] data, using Bson and byte[12] format respectively. All four types have automatic conversion from strings: Json to Bson for Document and DocArray, while ObjectId strings can be of hex digits or have four integer fields separated by colons: time:machine:process:count . A top-level document without an \_id will be given an \_id similar to MongoDB for the local machine. Documents are considered equal if corresponding fields match .

CharacterType = (([NATIONAL] CHARACTER) | CHAR | NCHAR | VARCHAR) [VARYING] [‘(‘int ‘)’] [CHARACTER SET id ] Collate .

National and varying are ignored, and the names are regarded as equivalent in Pyrrho .

Collate = [ COLLATE id ] .

There is no need to specify COLLATE UNICODE, since this is the default collation. COLLATE

UCS\_BASIC is supported but deprecated. For the list of available collations, see .NET documentation.

FloatType = (FLOAT|REAL|DOUBLE PRECISION) [‘(‘int’,’int’)’] .

The names here are regarded as equivalent in Pyrrho .

IntegerType = INT | INTEGER | BIGINT | SMALLINT .

All these integer types are regarded as equivalent in Pyrrho .

LobType = ([NATIONAL] CHARACTER |BINARY) LARGE OBJECT | BLOB | CLOB | NCLOB .

National is ignored, the character large object types are regarded as equivalent to CHAR since they represent unbounded character strings, and of course BINARY LARGE OBJECT is the same as BLOB.

NumericType = (NUMERIC|DECIMAL|DEC) [‘(‘int’,’int’)’] .

The names here are regarded as equivalent in Pyrrho .

DateTimeType = (DATE | TIME | TIMESTAMP) ([IntervalField [ TO IntervalField ]] | ['(' int ')']).

The use of IntervalFields when declaring DateTimeType is an addition to the SQL standard.

IntervalType = INTERVAL IntervalField [ TO IntervalField ] .

IntervalField = YEAR | MONTH | DAY | HOUR | MINUTE | SECOND [‘(‘ int ‘)’] .

DefinedType = (ROW|TABLE) Representation

| Type ARRAY

| Type MULTISET .

The TABLE alternative here is a Pyrrho extension to SQL2011, but currently there is no difference.

## A4.5 Data Manipulation

Insert = INSERT [WITH PROVENANCE string] [XMLOption] INTO Table\_id [ Cols ] Table\_Value

.

For example is INSERT INTO t VALUES (4,5) , or is INSERT INTO t SELECT c,d FROM e .

UpdatePositioned = UPDATE [XMLOption] Table\_id Assignment WHERE CURRENT OF Cursor\_id .

UpdateSearched = UPDATE [XMLOption] Table\_id Assignment [WhereClause] .

DeletePositioned = DELETE [XMLOption] FROM Table\_id WHERE CURRENT OF Cursor\_id .

DeleteSearched = DELETE [XMLOption] FROM Table\_id [ WhereClause] .

CursorSpecification = [XMLOption] QueryExpression [FOR ((READ ONLY)|(UPDATE [OF id {‘,’ id}))]] .

A simple table query is defined (SQL2011-02 14.1SR18c) as a CursorSpecification in which the QueryExpression is a QueryTerm that is a QueryPrimary that is a QuerySpecification.

QueryExpression = QueryExpressionBody [OrderByClause][FetchFirstClause] .

QueryExpressionBody = QueryTerm

| QueryExpression ( UNION | EXCEPT ) [ ALL | DISTINCT ] QueryTerm .

DISTINCT is the default and discards duplicates from both operands.

QueryTerm = QueryPrimary | QueryTerm INTERSECT [ ALL | DISTINCT ] QueryPrimary .

DISTINCT is the default.

QueryPrimary = SimpleTable | ‘(‘ QueryExpressionBody [OrderByClause][FetchFirstClause] ‘)’ .

SimpleTable = QuerySpecification | Value | TABLE id .

QuerySpecification = SELECT [ ALL | DISTINCT ] SelectList TableExpression .

SelectList = ‘\*’ | SelectItem { ‘,’ SelectItem } .

SelectItem = (Value [AS id ]) | ([id{‘.’id}’.’]‘\*’) .

TableExpression = FromClause [ WhereClause ] [ GroupByClause ] [ HavingClause ] [WindowClause] .

GroupByClause and HavingClause are used with aggregate functions.

FromClause = FROM TableReference { ‘,’ TableReference } .

WhereClause = WHERE BooleanExpr .

GroupByClause = GROUP BY [DISTINCT|ALL] GroupingSet { ‘,’ GroupingSet } .

GroupingSet = OrdinaryGroup | RollCube | GroupingSpec | ‘(‘’)’.

OrdinaryGroup = ColumnRef [Collate] | ‘(‘ ColumnRef [Collate] { ‘,’ ColumnRef [Collate] } ‘)’ .

RollCube = (ROLLUP|CUBE) ‘(‘ OrdinaryGroup { ‘,’ OrdinaryGroup } ‘)’ .

GroupingSpec = GROUPING SETS ‘(‘ GroupingSet { ‘,’ GroupingSet } ‘)’ .

HavingClause = HAVING BooleanExpr .

WindowClause = WINDOW WindowDef { ‘,’ WindowDef } .

Window clauses are only useful with window functions, which are discussed in section A6.7.

WindowDef = id AS ‘(‘ WindowDetails ‘)’ .

WindowDetails = [Window\_id] [ PartitionClause] [ OrderByClause ] [ WindowFrame ] .

PartitionClause = PARTITION BY OrdinaryGroup .

WindowFrame = (ROWS|RANGE) (WindowStart|WindowBetween) [ Exclusion ] .

WindowStart = ((Value | UNBOUNDED) PRECEDING) | (CURRENT ROW) .

WindowBetween = BETWEEN WindowBound AND WindowBound .

WindowBound = WindowStart | ((Value | UNBOUNDED) FOLLOWING ) .

Exclusion = EXCLUDE ((CURRENT ROW)|GROUP|TIES|(NO OTHERS)) .

TableReference = TableFactor Alias | JoinedTable .

TableFactor = Table\_id [FOR SYSTEM\_TIME [TimePeriodSpecification ]]

| View\_id

| ROWS '(' int [ ',' int ] ')'

| Table\_FunctionCall

| Subquery

| '(' TableReference ')'

| TABLE ‘(‘ Value ‘)’

| UNNEST ‘(‘ Value ‘)’

| XMLTABLE ‘(‘ [XMLOption] xml [PASSING NamedValue {‘,’ NamedValue}] XmlColumns ‘)’

| ‘[‘ [ Document\_Value { ‘,’ Document\_Value }] ‘]’ .

ROWS(..) is a Pyrrho extension (for table and cell logs), and the last two options above are also Pyrrho-specific: static is for a single query that does not access a table, and the other allows a specific list of documents to be supplied (static is actually equaivalent to [{}] ).

Alias = [[AS] id [ Cols ]] .

TimePeriodSpecification = AS OF Value

| BETWEEN [ASYMMETRIC|SYMMETRIC] Value AND Value | FROM Value TO Value .

This syntax is slightly more general than in SQL2011 – see section 5.2.3.

Subquery = '(' QueryExpression ')' .

Subqueries return different sorts of values depending on the context, including simple values (scalars, structures, arrays, multisets, etc), rows and tables.

JoinedTable = TableReference CROSS JOIN TableFactor

| TableReference NATURAL [JoinType] JOIN TableFactor

| TableReference [JoinType] JOIN TableFactor USING '('Cols')' [TO '('Cols')']

| TableReference [JoinType] JOIN TableReference ON SearchCondition .

The TO part is an extension to named column joins for cases where the tables being joined are related using an adapter function (see section 5.2.1), and then both sets of named columns are in the result rowset.

JoinType = INNER | ( LEFT | RIGHT | FULL ) [OUTER] .

SearchCondition = BooleanExpr .

OrderByClause = ORDER BY OrderSpec { ‘,’ OrderSpec } .

OrderSpec = Value [ ASC | DESC ] [ NULLS ( FIRST | LAST )] .

The default order is ascending, nulls first.

FetchFirstClause = FETCH FIRST [ int ] (ROW|ROWS) ONLY .

XmlColumns = COLUMNS XmlColumn { ‘,’ XmlColumn }.

XmlColumn = id Type [ DEFAULT Value ] [ PATH str ] .

## A4.6 Value

|  |  |
| --- | --- |
| Value = | Literal |
| | | Value BinaryOp Value |
| | | ‘-‘ Value |
| | | ‘(‘ Value ‘)’ |
| | | Value Collate |
| | | Value ‘[‘ Value ‘]’ |
| | | Value AS Type |
| | | ColumnRef |
| | | VariableRef |
| | | (SYSTEM\_TIME|Period\_id|(PERIOD’(‘Value,Value’)’)) |
| | | VALUE |
| | | ROW |
| | | Value ‘.’ Member\_id |
| | | MethodCall |
| | | NEW MethodCall |
| | | FunctionCall |
| | | VALUES ‘(‘ Value { ‘,’ Value } ‘)’ { ‘,’ ‘(‘ Value { ‘,’ Value } ‘)’ } |

| QueryExpression

| Subquery

| ROW ‘(‘ Value { ‘,’ Value } ‘)’

| ‘{‘ mongo\_spec ‘}’

| (MULTISET |ARRAY) ((‘[‘Value { ‘,’ Value } ‘]’)| Subquery)

| TREAT ‘(‘ Value AS Sub\_Type ‘)’

| CURRENT\_USER

| CURRENT\_ROLE

| HTTP GET url\_Value [ AS mime\_string ] .

The VALUE keyword is used in Check Constraints, ROW if not followed by ‘(‘ can be used for testing the type of the currently selected row. Subqueries that are mulitset-valued must be part of the MULTISET constructor syntax, and if row-valued must be part of the TABLE constructor syntax above. Collate if specified applies to an immediately preceding Boolean expression, affecting comparison operands etc. HTTP GET gives a string value by default unless used within an INSERT statement, in which case Pyrrho will attempt to convert to the expected row type. The mime string is used for retrieval of a particular content type from the server. QueryExpression and VALUES.. give a Table\_Value whose type is contextually determined. The AS syntax above is allowed only in parameter lists and methodcalls. A mongo\_spec is like the comma-separated key-value list in a JavaScript objects (Json): keys are case-sensitive and are enclosed in single or double quotes, and values are numerics, or single-or double-quoted strings, or regular expressions, or embedded JavaScript objects or arrays enclosed in []. Strings starting with $ have special meanings and can be used in SelectItems to refer to values in the curret context (e.g. “$a.b” ).

BinaryOp = ‘+’ | ‘-‘ | ‘\*’ | ‘/’ | ‘||’ | MultisetOp .

|| is used in array and string concatenation.

VariableRef = { Scope\_id ‘.’ } Variable\_id .

ColumnRef = [ TableOrAlias\_id ‘.’ ] ColRef

| TableOrAlias\_id ‘.’ (PROVENANCE| CHECK).

The use of the PROVENANCE and CHECK pseudo-columns is a change to SQL2011 behaviour. CHECK is a row versioning cookie derived from a string type.

MultisetOp = MULTISET ( UNION | INTERSECT | EXCEPT ) ( ALL | DISTINCT ) .

|  |  |
| --- | --- |
| Literal = | int |
| | | float |
| | | string |
| | | TRUE | FALSE |
| | | ‘X’ ‘’’ { hexit } ‘’’ |
| | | id ‘^^’ (Domain\_id|Type\_id|[Namepsace\_id]’:’id|uri) |
| | | DATE date\_string |
| | | TIME time\_string |
| | | TIMESTAMP timestamp\_string |
| | | INTERVAL [‘-‘] interval\_string IntervalQualifier . |

Strings are enclosed in single quotes. Two single quotes in a string represent one single quote. Hexits are hexadecimal digits 0-9, A-F, a-f and are used for binary objects.

The syntax with ^^ is special in Pyrrho and is added for RDF/OWL support, e.g. "2.5"^^units:ampere).

In the RDF/OWL syntax, @ in the double quoted part is a special character and introduces a locale. Note however that without the ^^, “2.5” would be an id and not a string, and the <> do not normally behave like quotes.

Dates, times and intervals use string (single quoted) values and are not locale-dependent. For full details see SQL2011: e.g.

* a date has format like DATE ‘yyyy-mm-dd’ ,
* a time has format like TIME ‘hh:mm:ss’ or TIME ‘hh:mm:ss.sss’ , • a timestamp is like TIMESTAMP ‘yyyy-mm-dd hh:mm:ss.ss’,
* an interval is like e.g.
  + INTERVAL ‘yyy’ YEAR,
  + INTERVAL ‘yy-mm’ YEAR TO MONTH,
  + INTERVAL ‘m’ MONTH,
  + INTERVAL ‘d hh:mm:ss’ DAY(1) TO SECOND,  INTERVAL ‘sss.ss’ SECOND(3,2) etc.

IntervalQualifier = StartField TO EndField | DateTimeField .

StartField = IntervalField [‘(‘ int ‘)’] .

EndField = IntervalField | SECOND [‘(‘ int ‘)’] .

DateTimeField = StartField | SECOND [‘(‘ int [‘,’ int]’)’] .

The ints here represent precision for the leading field and/or the fractional seconds.

IntervalField = YEAR | MONTH | DAY | HOUR | MINUTE .

## A4.7 Boolean Expressions

BooleanExpr = BooleanTerm | BooleanExpr OR BooleanTerm .

BooleanTerm = BooleanFactor | BooleanTerm AND BooleanFactor .

BooleanFactor = [NOT] BooleanTest .

BooleanTest = Predicate | ‘(‘ BooleanExpr ‘)’ | Boolean\_Value .

Predicate = Any | Between | Comparison | Contains | Every | Exists | In | Like | Like\_Regex | Member | Null | Of | PeriodBinary | Similar | Some | Unique.

Any = ANY ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Between = Value [NOT] BETWEEN [SYMMETRIC|ASYMMETRIC] Value AND Value .

Comparison = Value CompOp Value .

CompOp = ‘=’ | ‘<>’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ .

Contains = PeriodPredicand CONTAINS (PeriodPredicand | DateTime\_Value) .

Every = EVERY ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Exists = EXISTS Table\_Subquery | XMLEXISTS ‘(‘ XmlQuery ‘)’.

FuncOpt = [FILTER ‘(‘ WHERE SearchCondition ‘)’] [OVER WindowSpec] .

The presence of the OVER keyword makes a window function. In accordance with SQL2011-02 section 4.15.3, window functions can only be used in the select list of a QuerySpecification or SelectSingle or the order by clause of a “simple table query” as defined in section 7.5 above. Thus window functions cannot be used within expressions or as function arguments.

In = Value [NOT] IN ‘(‘ Table\_Subquery | ( Value { ‘,’ Value } ) ‘)’ .

Like = Value [NOT] LIKE Value [ ESCAPE Value ].

Like\_Regex = Value [NOT] LIKE\_REGEX Pattern\_Value [FLAG Value]

Member = Value [ NOT ] MEMBER OF Value .

Null = Value IS [NOT] NULL .

Of = Value IS [NOT] ( OF ‘(‘ [ONLY] Type {‘,’[ONLY] Type } ‘)’ | (CONTENT | DOCUMENT | VALID) ).

Similar = Value [ NOT ] SIMILAR TO Regex\_Value [ ESCAPE char ].

The Regex\_Value is an expression, whose value is a character string that parses recursively as follows: RE = RT {‘|’ RT } . RT = RF { RF } . RF = RP [‘\*’|’+’|’?’| ‘{‘ unsigned [‘,’ unsigned ] ‘}’] . RP = {RC} | [‘RR {RR} | [‘^’ {RR}]’]’|’(‘[unsigned]RE’)’. RR =RC | (char’-‘char)|(’[‘ ’:’ id ‘:’’]’). RC= char | ’\’char .

Some = SOME ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Unique = UNIQUE Table\_Subquery .

PeriodBinary = PeriodPredicand (OVERLAPS | EQUALS | [IMMEDIATELY] (PRECEDES | SUCCEEDS)) PeriodPredicand .

See also Contains above.

PreiodPredicand = { id ‘.’ } id | PERIOD ‘(‘ Value ‘,’ Value ‘)’ .

## A4.8 SQL Functions

FunctionCall = NumericValueFunction | StringValueFunction | DateTimeFunction | SetFunctions | TypeCast | XMLFunction | UserFunctionCall | MethodCall .

NumericValueFunction = AbsoluteValue | Avg | Ceiling | Coalesce | Correlation | Count |

Covariance | Exponential | Extract | Floor | Grouping | Last | LengthExpression | Maximum |

Minimum | Modulus | NaturalLogarithm | Next | Nullif | Occurrences\_Regex | Percentile | Position | Position\_Regex | PowerFunction | Rank | Regression | RowNumber | Schema | SquareRoot | StandardDeviation | Sum | Variance .

AbsoluteValue = ABS ‘(‘ Value ‘)’ .

Avg = AVG ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Ceiling = (CEIL|CEILING) ‘(‘ Value ‘)’ .

Coalesce = COALESCE ‘(‘ Value {‘,’ Value } ‘)’

Correlation = CORR ‘(‘ Value ‘,’ Value ‘)’ FuncOpt .

Count = COUNT ’(‘ ‘\*’ ‘)’

| COUNT ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Covariance = (COVAR\_POP|COVAR\_SAMP) ‘(‘ Value ‘,’ Value ‘)’ FuncOpt .

Exponential = EXP ‘(‘ Value ‘)’ .

Extract = EXTRACT ‘(‘ ExtractField FROM Value ‘)’ .

ExtractField = YEAR | MONTH | DAY | HOUR | MINUTE | SECOND.

Floor = FLOOR ‘(‘ Value ‘)’ .

Grouping = GROUPING ‘(‘ ColumnRef { ‘,’ ColumnRef } ‘)’ .

Last = LAST ['(' ColumnRef ')' OVER WindowSpec ] .

LengthExpression = (CHAR\_LENGTH|CHARACTER\_LENGTH|OCTET\_LENGTH) ‘(‘ Value ‘)’ .

Maximum = MAX ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Minimum = MIN ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Modulus = MOD ‘(‘ Value ‘,’ Value ‘)’ .

NaturalLogarithm = LN ‘(‘ Value ‘)’ .

Next = NEXT ['(' ColumnRef ')' OVER WindowSpec ] .

Nullif = NULLIF ‘(‘ Value ‘,’ Value ‘)’ .

Occurrences\_Regex = OCCURRENCES\_REGEX ‘(‘Pattern\_Value [FLAG Options\_Value] IN String\_Value [FROM StartPos\_Value ] [USING (CHARACTERS|OCTETS)] ‘)’ .

Percentile = (PERCENTILE\_CONT|PERCENTILE\_DISC) ‘(‘ Value ‘)’ WithinGroup .

WindowSpec = Window\_id | ‘(‘ WindowDetails ‘)’ .

WithinGroup = WITHIN GROUP ‘(‘ OrderByClause ‘)’ .

Position = POSITION [‘(‘Value IN Value ‘)’] .

Without parameters POSITION gives a Pyrrho log entry (see section 3.5).

Position\_Regex = POSITION\_REGEX ‘(‘ [START|AFTER] Pattern\_Value [FLAG Options\_Value] IN String\_Value [FROM StartPos\_Value] [USING (CHARACTERS|OCTETS)] [OCCURRENCE RegexOccurrence\_Value] [GROUP RegexCaptureGroup\_Value] ‘)’ .

PowerFunction = POWER ‘(‘ Value ‘,’ Value ‘)’ .

Rank = (CUME\_DIST|DENSE\_RANK|PERCENT\_RANK|RANK) ‘(‘’)’ OVER WindowSpec

| (DENSE\_RANK|PERCENT\_RANK|RANK|CUME\_DIST) ‘(‘ Value {‘,’ Value } ‘)’ WithinGroup .

Regression = (REGR\_SLOPE|REGR\_INTERCEPT|REGR\_COUNT|REGR\_R2|REGR\_AVVGX| REGR\_AVGY|REGR\_SXX|REGR\_SXY|REGR\_SYY) ‘(‘ Value ‘,’ Value ‘)’ FuncOpt .

RowNumber = ROW\_NUMBER ‘(‘’)’ OVER WindowSpec .

Schema = SCHEMA ‘(‘ ObjectName [ COLUMN id ]‘)’ .

Added for Pyrrho: returns a number identifying the most recent schema change affecting the specified object (including any change to this object by another name in another role). Note the sytax of ObjectName given in sec 7.4 above above uses keyword prefixes such as TABLE. The COLUMN syntax shown can only be used with tables.

SquareRoot = SQRT ‘(‘ Value ‘)’ .

StandardDeviation = (STDDEV\_POP|STDDEV\_SAMP) ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Sum = SUM ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Variance = (VAR\_POP|VAR\_SAMP) ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

DateTimeFunction = CURRENT\_DATE | CURRENT\_TIME | LOCALTIME | CURRENT\_TIMESTAMP | LOCALTIMESTAMP .

StringValueFunction = Normalize | Substring | Substring\_Regex | Translate\_Regex | Fold | Trim | Provenance | XmlAgg .

Normalize= NORMALIZE '(' Value ')' .

Substring = SUBSTRING ‘(‘ Value FROM Value [ FOR Value ] ‘)’ | SUBSTRING ‘(‘ Value SIMILAR Value ESCAPE Value ‘)’.

Substring\_Regex = SUBSTRING\_REGEX ‘(‘ Pattern\_Value [FLAG Flag\_Value] IN Value [FROM StartPos\_Value] [USING (CHARACTERS|OCTETS)] [OCCURRENCE RegexOccurrence\_Value] [GROUP RegexCaptreGroup\_Value] ‘)’.

Translate\_Regex = TRANSLATE\_REGEX ‘(‘ Pattern\_Value [FLAG Flag\_Value] IN Value [WITH Replacement\_Value] [FROM StartPos\_Value] [USING (CHARACTERS|OCTETS)] [OCCURRENCE RegexOccurrence\_Value] [GROUP RegexCaptreGroup\_Value] ‘)’.

Fold = (UPPER|LOWER) ‘(‘ Value ‘)’ .

Trim = TRIM ‘(‘ [[LEADING|TRAILING|BOTH] [character] FROM] Value ‘)’ .

Provenance = PROVENANCE | TYPE\_URI [ ‘(‘ Value ‘)’ ] .

XmlAgg = XMLAGG ‘(‘ Value [ OrderByClause ] ‘)’ .

SetFunction = Cardinality | Collect | Element | Fusion | Intersect | Set .

Collect = COLLECT ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Fusion = FUSION ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Intersect = INTERSECTION ‘(‘ [DISTINCT|ALL] Value) ’)’ FuncOpt .

Cardinality = CARDINALITY ‘(‘ Value ‘)’ .

Element = ELEMENT ‘(‘ Value ‘)’ .

Set = SET ‘(‘ Value ‘)’ .

Typecast = (CAST | XMLCAST) ‘(‘ Value AS Type ‘)’ | TREAT ‘(‘ Value AS Sub\_Type ‘)’ .

## A4.9 Statements

Assignment = SET Target ‘=’ Value { ‘,’ Target ‘=’ Value }

| SET ‘(‘ Target { ‘,’ Target } ‘)’ ‘=’ Value .

For a simple assignment of form Target = Value, the keyword SET can be omitted.

Target = id { ‘.’ id } .

Targets which directly contain parameter lists are not supported in the SQL2011 standard.

Call = CALL Procedure\_id ‘(‘ [ Value { ‘,’ Value } ] ‘)’ | MethodCall .

Inside a procedure declaration the CALL keyword can be omitted.

CaseStatement = CASE Value { WHEN Values THEN Statements }[ ELSE Statements ]END CASE | CASE { WHEN SearchCondition THEN Statements } [ ELSE Statements ] END CASE .

There must be at least one WHEN in the forms shown above.

Close = CLOSE id .

CompoundStatement = Label BEGIN [XMLDec] Statements END .

XMLDec = DECLARE Namespace ‘;’ .

Declaration = DECLARE id { ‘,’ id } Type

| DECLARE id CURSOR FOR CursorSpecification

| DECLARE HandlerType HANDLER FOR ConditionList Statement .

Declarations of identifiers, cursors, and handlers are specific to a scope in a SQL routine.

HandlerType = CONTINUE | EXIT | UNDO .

ConditionList = Condition { ‘,’ Condition } .

Condition = ConditionCode | SQLEXCEPTION | SQLWARNING | (NOT FOUND) .

The ConditionCode not\_found is acceptable as an alternative to not found.

Signal = SIGNAL ConditionCode [ SET CondInfo’=’Value{‘,’CondInfo’=’Value}] | RESIGNAL [ConditionCode ] [ SET CondInfo’=’Value{‘,’CondInfo’=’Value}] .

ConditionCode = Condition\_id | SQLSTATE string .

CondInfo = CLASS\_ORIGIN|SUBCLASS\_ORIGIN|CONSTRAINT\_CATALOG| CONSTRAINT\_SCHEMA| CONSTRAINT\_NAME|CATALOG\_NAME|SCHEMA\_NAME| TABLE\_NAME|COLUMN\_NAME|CURSOR\_NAME|MESSAGE\_TEXT .

GetDiagnostics = GET DIAGNOSTICS Target ‘=’ ItemName { ‘,’ Target ‘=’ ItemName }.

ItemName = NUMBER | MORE | COMMAND\_FUNCTION | COMMAND\_FUNCTION\_CODE |

DYNAMIC\_FUNCTION | DYNAMIC\_FUNCTION\_CODE | ROW\_COUNT | TRANSACTIONS\_COMMITTED

| TRANSACTIONS\_ROLLED\_BACK | TRANSACTION\_ACTIVE | CATALOG\_NAME | CLASS\_ORIGIN |

COLUMN\_NAME | CONDITION\_NUMBER | CONNECTION\_NAME | CONSTRAINT\_CATALOG |

CONSTRAINT\_NAME | CONSTRAINT\_SCHEMA | CURSOR\_NAME | MESSAGE\_LENGTH |

MESSAGE\_OCTET\_LENGTH | MESSAGE\_TEXT | PARAMETER\_MODE | PARAMETER\_NAME |

PARAMETER\_ORDINAL\_POSITION | RETURNED\_SQLSTATE | ROUTINE\_CATALOG | ROUTINE\_NAME | ROUTINE\_SCHEMA | SCHEMA\_NAME | SERVER\_NAME | SPECIFIC\_NAME | SUBCLASS\_ORIGIN | TABLE\_NAME | TRIGGER\_CATALOG | TRIGGER\_NAME | TRIGGER\_SCHEMA .

SQLSTATE strings are 5 characters in length, comprising a 2-character class and and a 3 character subclass. See the table in section 8.1.1.

Fetch = FETCH [How] Cursor\_id INTO VariableRef { ‘,’ VariableRef } .

How = NEXT | PRIOR | FIRST | LAST | ((ABSOLUTE | RELATIVE) Value )) .

ForStatement = Label FOR [ For\_id AS ][ id CURSOR FOR ] QueryExpression DO Statements END FOR [Label\_id] .

IfStatement = IF BooleanExpr THEN Statements { ELSEIF BooleanExpr THEN Statements } [ ELSE Statements ] END IF .

Label = [ label ‘:’ ] .

LoopStatement = Label LOOP Statements END LOOP .

Open = OPEN id .

Repeat = Label REPEAT Statements UNTIL BooleanExpr END REPEAT .

SelectSingle = QueryExpresion INTO VariableRef { ‘,’ VariableRef } .

Statements = Statement { ‘;’ Statement } .

While = Label WHILE SearchCondition DO Statements END WHILE .

UserFunctionCall = Id ‘(‘ [ Value {‘,’ Value}] ‘)’ .

MethodCall = Value ‘.’ Method\_id ‘(‘ [ Value { ‘,’ Value } ] ‘)’

| ‘(‘ Value AS Type ‘)’ ‘.’ Method\_id ‘(‘ [ Value { ‘,’ Value } ] ‘)’ | Type’::’ Method\_id ‘(‘ [ Value { ‘,’ Value } ] ‘)’ .

## A4.10 XML Support

XMLFunction = XMLComment | XMLConcat | XMLDocument | XMLElement | XMLForest | XMLParse | XMLProc | XMLQuery | XMLText | XMLValidate.

XMLComment = XMLCOMMENT ‘(‘ Value ‘)’ .

XMLConcat = XMLCONCAT ‘(‘ Value {‘,’ Value } ‘)’ .

XMLDocument = XMLDOCUMENT ‘(‘ Value ‘)’ .

XMLElement = XMLELEMENT ‘(‘ NAME id [ ‘,’ Namespace ] [‘,’ AttributeSpec ]{ ‘,’ Value } ‘)’ .

Namespace = XMLNAMESPACES ‘(‘ NamespaceDefault |( string AS id {‘,’ string AS id }) ‘)’ .

NamespaceDefault = (DEFAULT string) | (NO DEFAULT) .

AttributeSpec = XMLATTRIBUTES ‘(‘ NamedValue {‘,’ NamedValue }‘)’ .

NamedValue = Value [ AS id ] .

XMLForest = XMLFOREST ‘(‘ [ Namespace ‘,’] NamedValue { ‘,’ NamedValue } ‘)’ .

XMLParse = XMLPARSE ‘(‘ CONTENT Value ‘)’ .

XMLProc = XMLPI ‘(‘ NAME id [‘,’ Value ] ‘)’ .

XMLQuery = XMLQUERY ‘(‘ Value , xpath\_xml ‘)’ .

This syntax seems to be non-standard in Pyrrho but allows extraction from an xml Value using an XPath expression

XMLText = XMLTEXT‘(‘ xml ‘)’ .

XMLValidate = XMLVALIDATE‘(‘ (DOCUMENT|CONTENT|SEQUENCE) Value ‘)’ .

# Appendix 5 Pyrrho’s condition codes

Standard SQL exception handling allows stored procedures to set up handlers for various conditions that arise in the database engine. The SQL standard specifies a number of these, but most in the list below are specific to Pyrrho. Diagnostic information about the causes of the condition is also available through use of the GET DIAGNOSTICS mechanisms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number** | **Message Template** | **ISO** | **Pyrrho** | **Comments** |
| 00000 | Successful completion | y | y | Not an exception |
| 01000 | Warning | y | n |  |
| 01001 | Warning – cursor operation conflict | y | n | Not reported |
| 01002 | Warning – disconnect error | y | n | Condition 2E000 raised instead |
| 01003 | Warning – null value eliminated in set function | y | y |  |
| 01004 | Warning – string data, right truncation | y | n | 01004 is used for fixed length binary data: see 22001 instead |
| 01005 | Warning – insufficient item descriptor areas | y | n |  |
| 01006 | Warning – privilege not revoked | y | n | Condition 42105 raised instead |
| 01007 | Warning – privilege not granted | y | n | Condition 42105 raised instead |
| 01005 | Warning – insufficient item descriptor areas | y | n | Cannot occur |
| 01009 | Warning – search condition too long for information schema | y | n | Cannot occur |
| 0100A | Warning – query expression too long for information schema | y | n | Cannot occur |
| 0100B | Warning – default value too long for information schema | y | n | Cannot occur |
| 0100C | Warning – result sets returned | y | n |  |
| 0100D | Warning – additional result sets returned | y | n |  |
| 0100E | Warning – attempt to return too many result parameters | y | y |  |
| 0100F | Warning – statement too long for information schema | y | n | Cannot occur |
| 01012 | Warning – invalid number of conditions | y | n | Not reported |
| 0102F | Warning – array data, right truncation | y | n | Cannot occur |
| 02000 | No data | y | y | Not an exception |
| 02001 | No additional result sets returned | y | n |  |
| 07000 | Dynamic SQL error | y | n |  |
| 07001 | Using clause does not match dynamic parameter specifications | y | n |  |
| 07002 | Using clause does not match target specifications | y | n |  |
| 07003 | Cursor specification cannot be executed | y | n |  |
| 07004 | Using clause required for dynamic parameters | y | n |  |
| 07005 | Prepared statement not a cursor specification | y | n |  |
| 07006 | Restricted data type attribute violation | y | n |  |
| 07007 | Using clause required for result fields | y | n |  |
| 07008 | Invalid descriptor count | y | n |  |
| 07009 | Invalid descriptr index | y | n |  |
| 0700B | Data type transform function violation | y | n |  |
| 0700C | Undefined DATA value | y | n |  |
| 0700D | Invalid DATA target | y | n |  |
| 0700E | Invalid LEVEL value | y | n |  |
| 0700F | Invalid DATETIME\_INTERVAL\_CODE | y | n |  |
| 0700G | Invalid pass-through surrogate value | y | n |  |
| 0700H | PIPE ROW not during PTF execution | y | n |  |
| 08000 | Connection exception | y | n | See 2E |
| 08001 | SQL-client unable to establish SQL-connection | y | y |  |
| 08002 | Connection name in use | y | y |  |
| 08003 | Connection does not exist | y | y |  |
| 08004 | SQL-Server rejected establishment of SQL-connection | y | y |  |
| 08006 | Connection failure | y | y |  |
| 08007 | Connection exception – transaction resolution unknown | y | y |  |
| 08C00 | Client-side threading violation for reader | n | y |  |
| 08C01 | Client-side threading violation for command | n | y |  |
| 08C02 | Client-side threading violation for a transaction | n | y |  |
| 08C03 | An explicit transaction is already active in this thread and connecion | n | y |  |
| 08C04 | A reader is already open in this thread and connection | n | y |  |
| 08C05 | Conflict with an open reader in this thread and connection | n | y |  |
| 08C06 | Cannot change connection properties during a transaction | n | y |  |
| 09000 | Triggered action exception | y | n | Pyrrho uses a single diagnostics area |
| 0A000 | Feature not supported | y | n |  |
| 0A001 | Feature not supported – multiple server transactions | y | n | Pyrrho supports multiple serevrs |
| 0D000 | Invalid target type specification | y | y |  |
| 0E000 | Invalid schema name list specification | y | n | S071 is not supported |
| 0F000 | Locator exception | y | n | T561 is not supported |
| 0F001 | Locator exception – invalid specification | y | n | T561 is not supported |
| 0L000 | Invalid grantor | y | n | Condition 42105 is raised |
| 0M000 | Invalid SQL-invoked procedure reference | y | n | T471 is not supported |
| 0P000 | Invalid role specification | y | n | Condition 42105 is raised |
| 0S000 | Invalid transform group specification | y | n | S241 is not supported |
| 0T000 | Target table disagrees with cursor specification | y | y |  |
| 0U000 | Attempt to assign to non-updatable column | y | y |  |
| 0V000 | Attempt to assign to ordering column | y | n | B031 is not supported |
| 0W000 | Prohibited statement encountered during trigger execution | y | n | See 27000 |
| 0W001 | Trigger error - modify table modified by data change delta table | y | n | See 27001 |
| 0Z000 | Diagnostics exception | y | n |  |
| 0Z001 | Maximum number of stacked diagnostics areas exceeded | y | n | Cannot occur |
| 11000 | Prohibited column reference encountered during trigger execution | y | n |  |
| 21000 | Cardinaility violation | y | y |  |
| 22000 | Data exception | y | y |  |
| 22001 | String data, right truncation | y | y |  |
| 22002 | Null value, no indicator parameter | y | n |  |
| 22003 | Numeric value out of range | y | y |  |
| 22004 | Null value not allowed | y | y |  |
| 22005 | Error in assignment | y | y |  |
| 22006 | Invalid interval format | y | n |  |
| 22007 | Invalid datetime format: ? | y | y | Diagnostic info added |
| 22008 | Datetime field overflow: ? | y | y | Diagnostic info added |
| 22009 | Invalid time zone displacement value | y | n |  |
| 2200B | Escape character conflict | y | n |  |
| 2200C | Invalid use of escape character | y | n |  |
| 2200D | Invalid escape octet | y | n |  |
| 2200E | Null value in array target | y | n |  |
| 2200F | Zero-length character string | y | n |  |
| 2200G | Most specific type mismatch | y | y |  |
| 2200H | Sequence generator limit exceeded | y | n |  |
| 2200J | Nonidentical notations with the same name | y | n |  |
| 2200K | Nonidentical unparsed entities with the same name | y | n |  |
| 2200N | Invalid XML content | y | y |  |
| 2200P | Interval value out of range | y | n |  |
| 2200Q | Multiset value overflow | y | n | Cannot occur |
| 2200S | Invalid XML comment | y | n |  |
| 22010 | Invalid indicator parameter value | y | n |  |
| 22011 | Substring error | y | n |  |
| 22012 | Division by zero | y | y |  |
| 22013 | Invalid preceding or following size in window function | y | n |  |
| 22014 | Invalid argument for NTILE function | y | n |  |
| 22015 | Interval field overflow | y | n |  |
| 22016 | Invalid argument for NTH\_VALUE function | y | n |  |
| 22018 | Invalid character value for cast | y | n |  |
| 22019 | Invalid escape character | y | y |  |
| 2201B | Invalid regular expression | y | y |  |
| 2201C | Null row not permitted in value | y | n |  |
| 2201E | Invalid argument for natural logarithm | y | n |  |
| 2201F | Invalid argument for power function | y | n |  |
| 2201G | Invalid argument for width bucket function | y | n |  |
| 2201H | Invalid row version | y | n |  |
| 2201M | Namespace ? not defined | y | y |  |
| 2201S | Invalid XQuery regular expression | y | n |  |
| 2201T | Invalid XQuery option flag | y | n |  |
| 2201U | Attempt to replace a zero-length string | y | n |  |
| 2201V | Invalid XQuery replacement string | y | n |  |
| 2201W | Invalid row count in a fetch first clause | y | n |  |
| 2201X | Invalid row count in result offset clause | y | n |  |
| 2201Y | Zero-length binary string | y | n |  |
| 22020 | Invalid period value | y | n |  |
| 22021 | Character not in repertoire | y | n |  |
| 22022 | Indicator overflow | y | n |  |
| 22023 | Invalid parameter value | y | n |  |
| 22024 | Unterminated C string | y | n |  |
| 22025 | Invalid escape sequence | y | y |  |
| 22026 | String data length mismatch | y | n |  |
| 22027 | Trim error | y | n |  |
| 22029 | Noncharacter in UCS string | y | n |  |
| 2202D | Null value substituted for mutator subject parameter | y | n |  |
| 2202E | Array element error | y | n |  |
| 2202F | Array data, right truncation | y | n |  |
| 2202G | Invalid repeat argument in a sample clause | y | n |  |
| 2202H | Invalid sample size | y | n |  |
| 2202J | Invalid argument for row pattern navigation operation | y | b |  |
| 2202K | Skip to non-existent row | y | n |  |
| 2202L | Skip to first row of match | y | n |  |
| 22030 | Duplicate JSON object key value | y | n |  |
| 22031 | Invalid argument for SQL/JSON datetime function | y | n |  |
| 22032 | Invalid JSON text | y | n |  |
| 22033 | Invalid SQL/JSON subscript | y | n |  |
| 22034 | More than one SQL/JSON item |  |  |  |
| 22035 | No SQL/JSON item | y | n |  |
| 22036 | Non-numeric SQL/JSON i8tem | y | n |  |
| 22037 | Non-unique keys in JSON object | y | n |  |
| 22038 | Singleton SQL/JSON item required | y | n |  |
| 22039 | SQL/JSON array not found | y | n |  |
| 2203A | SQL/JSON member not found | y | n |  |
| 2203B | SQL/JSON number not found | y | n |  |
| 2203C | SQL/JSON object not found | y | n |  |
| 2203D | Too many JSON array elements | y | n |  |
| 2203E | Too many JSON object members | y | n |  |
| 2203F | SQL/JSON scalar required | y | n |  |
| 22041 | Invalid RDF format | n | y | OWL type extension to SQL |
| 22102 | Type mismatch on concatenate | n | y |  |
| 22103 | Multiset element not found | n | y |  |
| 22104 | Incompatible multisets for union | n | y |  |
| 22105 | Incompatible multisets for intersection | n | y |  |
| 22106 | Incompatible multisets for except | n | y |  |
| 22107 | Exponent expected | n | y |  |
| 22108 | Type error in aggregation operation | n | y |  |
| 22109 | Too few arguments | n | y |  |
| 22110 | Too many arguments | n | y |  |
| 22111 | Circular dependency found | n | y |  |
| 22201 | Unexpected type ? for comparison with Decimal | n | y |  |
| 22202 | Incomparable types | n | y |  |
| 22203 | Loss of precision on conversion | n | y |  |
| 22204 | Query expected | n | y |  |
| 22205 | Null value found in table ? | n | y |  |
| 22206 | Null value not allowed in column ? | n | y |  |
| 22207 | Row has incorrect length | n | y |  |
| 22208 | Mixing named and unnamed columns is not supported | n | y |  |
| 22209 | AutoKey is not available for ? | n | y |  |
| 22210 | Illegal assignment of sensitive value | n | y |  |
| 22211 | Domain ? Check constraint fails | n | y |  |
| 22212 | Column ? Check constraint fails | n | y |  |
| 22300 | Bad document format | n | y | Document Extension to SQL |
| 23000 | Integrity constraint violation | y | y |  |
| 23001 | RESTRICT: ? referenced in ? | y | y | A referenced object cannot be deleted |
| 23103 | This record cannot be updated | n | y | usually integrity violation |
| 24000 | Invalid cursor state | y | y |  |
| 24101 | Cursor is not open | n | y |  |
| 25000 | Invalid transaction state | y | y |  |
| 25001 | Active SQL-transaction | y | y |  |
| 25002 | Branch transaction already active | y | n |  |
| 25003 | Inappropriate access mode for branch transaction | y | n |  |
| 25004 | Inappropriate isolation level for branch transaction | y | n |  |
| 25005 | No active SQL-transaction for branch transaction | y | n |  |
| 25006 | Read-only SQL-transaction | y | n |  |
| 25007 | Schema and data statement mixing not supported | y | n |  |
| 25008 | Held cursor requires same isolation level | y | n |  |
| 26000 | Invalid SQL statement name | y | y |  |
| 27000 | Triggered data change violation | y | n |  |
| 27001 | Trigger exception – modify table modified by data change delta table | y | n |  |
| 28000 | Invalid authorization specification | y | y | No role ? in database ? |
| 28101 | Unknown grantee kind | n | y |  |
| 28102 | Unknown grantee ? | n | y |  |
| 28104 | Users can only be added to roles | n | y |  |
| 28105 | Grant of select: entire row is nullable | n | y |  |
| 28106 | Grant of insert must include all notnull columns | n | y |  |
| 28107 | Grant of insert cannot include generated column ? | n | y |  |
| 28108 | Grant of update : column ? is not updatable | n | y |  |
| 2B000 | Dependent privilege descriptors still exist | y | n |  |
| 2C000 | Invalid character set name | y | n |  |
| 2C001 | Cannot drop SQL-session default character vset | y | n |  |
| 2D000 | Invalid transaction termination | y | y |  |
| 2E000 | Invalid connection name | y | y |  |
| 2E104 | Database is read-only | n | y |  |
| 2E105 | Invalid user for database ? | n | y |  |
| 2E106 | This operation requires a single-database session | n | y |  |
| 2E108 | Stop time was specified, so database is read-only | n | y |  |
| 2E110 | Unauthorized HTTP access | n | y |  |
| 2E111 | User ? can access no columns of table ? | n | y |  |
| 2E201 | Connection is not open | n | y | See also 080nn |
| 2E202 | A reader is already open | n | y |  |
| 2E203 | Unexpected reply | n | y |  |
| 2E204 | Bad data type ? (internal) | n | y |  |
| 2E205 | Stream closed | n | y |  |
| 2E206 | Internal error: ? | n | y |  |
| 2E208 | Badly formatted connection string ? | n | y |  |
| 2E209 | Unexpected element ? in connection string | n | y |  |
| 2E210 | LOCAL database server does not support distributed or partitioned operation | n | y |  |
| *2E300* | *The calling assembly does not have type ?* | *n* | *y* |  |
| *2E301* | *Type ? doesn’t have a default constructor* | *n* | *y* |  |
| *2E302* | *Type ? doesn’t define field ?* | *n* | *y* |  |
| 2E303 | Types ? and ? do not match | n | y |  |
| 2E304 | Get rurl should begine with / | n | y | REST service |
| 2E305 | No data returned by rurl ? | n | y | REST service |
| 2E307 | Obtain an up-to-date schema for ? from Role$Class | n | y |  |
| 2F000 | SQL routine exception | y | n |  |
| 2F002 | Modifying SQL-data not permitted | y | n |  |
| 2F003 | Prohibited SQL-statement attempted | y | y |  |
| 2F004 | Reading SQL-data not permitted | y | n |  |
| 2F005 | Function executed no return statement | y | n |  |
| 2H000 | Invalid collation name | y | y |  |
| 30000 | Invalid SQL statement identifier | y | n |  |
| 33000 | Invalid SQL descriptor name | y | y |  |
| 33001 | Error in prepared statement parameters | n | y |  |
| 34000 | Invalid cursor name | y | y |  |
| 35000 | Invalid condition number | y | n |  |
| 36000 | Cursor sensitivity exception | y | n |  |
| 36001 | Cursor sensitivity exception – request rejected | y | n |  |
| 36002 | Cursor sensitivity exception – request failed | y | n |  |
| 38000 | External routine exception | y | n |  |
| 38001 | External routine – containing SQL not permitted | y | n |  |
| 38002 | External routine – modifying SQL-data not permitted | y | n |  |
| 38003 | External routine – prohiobited SQL-statement attempted | y | n |  |
| 38004 | External routine – reading SQL-data not permitted | y | n |  |
| 39000 | External routine invocation exception | y | n |  |
| 39004 | External routine invocation – null value not allowed | y | n |  |
| 3B000 | Savepoint exception | y | n |  |
| 3B001 | Savepoint exception – invalid specification | y | n |  |
| 3B002 | Too many savepoints | y | n |  |
| 3C000 | Ambiguous cursor name | y | n |  |
| 3D000 | Invalid catalog specification | y | y |  |
| 3D001 | Database ? not open | n | y |  |
| 3D005 | Requested operation not supported by this edition of Pyrrho | n | y |  |
| 3D006 | Database ? incorrectly terminated or damaged | n | y |  |
| 3D007 | Database is not append storage | n | y | Server is append storage version |
| 3D008 | Database is append storage | n | y | Server is not for append storage |
| 3D010 | Invalid Password | n | y |  |
| 3F000 | Invalid schema name | y | n |  |
| 40000 | Transaction rollback | y | y |  |
| 40001 | Transaction Serialisation Failure | y | y |  |
| 40002 | Transaction rollback – integrity constraint violation | y | n |  |
| 40003 | Transaction rollback – statement completion unknown | y | y |  |
| 40004 | Transaction rollback – triggered action exception | y | n |  |
| 40005 | Transaction rollback – new key conflict with empty query | n | y |  |
| 40006 | Transaction conflict: Read constraint for ? | n | y |  |
| 40007 | Transaction conflict: Read conflict for ? | n | y |  |
| 40008 | Transaction conflict: Read conflict for table ? | n | y |  |
| 40009 | Transaction conflict: Read conflict for record ? | n | y |  |
| 40010 | Object ? has just been dropped | n | y |  |
| 40011 | Supertype ? has just been dropped | n | y |  |
| 40012 | Table ? has just been dropped | n | y |  |
| 40013 | Column ? has just been dropped | n | y |  |
| 40014 | Record ? has just been deleted | n | y |  |
| 40015 | Type ? has just been dropped | n | y |  |
| 40016 | Domain ? has just been dropped | n | y |  |
| 40017 | Index ? has just been dropped | n | y |  |
| 40021 | Supertype ? has just been changed | n | y |  |
| 40022 | Another domain ? has just been defined | n | y |  |
| 40023 | Period ? has just been changed | n | y |  |
| 40024 | Versioning has just been defined | n | y |  |
| 40025 | Table ? has just been altered | n | y |  |
| 40026 | Integrity constraint: ? has just been added | n | y |  |
| 40027 | Integrity constraint: ? has just been referenced | n | y |  |
| 40029 | Record ? has just been updated | n | y |  |
| 40030 | A conflicting table ? has just been defined | n | y |  |
| 40031 | A conflicting view ? has just been defined | n | y |  |
| 40032 | A conflicting object ? has just been defined | n | y |  |
| 40033 | A conflicting trigger for ? has just been defined | n | y |  |
| 40034 | Table ? has just been renamed | n | y |  |
| 40035 | A conflicting role ? has just been defined | n | y |  |
| 40036 | A conflicting routine ? has just been defined | n | y |  |
| 40037 | An ordering now uses function ? | n | y |  |
| 40038 | Type ? has just been renamed | n | y |  |
| 40039 | A conflicting method ? for ? has just been defined | n | y |  |
| 40040 | A conflicting period for ? has just been defined | n | y |  |
| 40041 | Conflicting metadata for ? has just been defined | n | y |  |
| 40042 | A conflicting index for ? has just been defined | n | y |  |
| 40043 | Columns of table ? have just been changed | n | y |  |
| 40044 | Column ? has just been altered | n | y |  |
| 40045 | A conflicting column ? has just been defined | n | y |  |
| 40046 | A conflicting check ? has just been defined | n | y |  |
| 40047 | Target object ? has just been renamed | n | y |  |
| 40048 | A conflicting ordering for ? has just been defined | n | y |  |
| 40049 | Ordering definition conflicts with drop of ? | n | y |  |
| 40050 | A conflicting namespace change has occurred | n | y |  |
| 40051 | Conflict with grant/revoke on ? | n | y |  |
| 40052 | Conflicting routine modify for ? | n | y |  |
| 40053 | Domain ? has just been used for insert | n | y |  |
| 40054 | Domain ? has just been used for update | n | y |  |
| 40055 | An insert conflicts with drop of ? | n | y |  |
| 40056 | An update conflicts with drop of ? | n | y |  |
| 40057 | A delete conflicts with drop of ? | n | y |  |
| 40058 | An index change conflicts with drop of ? | n | y |  |
| 40059 | A constraint change conflicts with drop of ? | n | y |  |
| 40060 | A method change conflicts with drop of type ? | n | y |  |
| 40068 | Domain ? has just been altered, conflicts with drop | n | y |  |
| 40069 | Method ? has just been changed, conflicts with drop | n | y |  |
| 40070 | A new ordering conflicts with drop of type ? | n | y |  |
| 40071 | A period definition conflicts with drop of ? | n | y |  |
| 40072 | A versioning change conflicts with drop of period ? | n | y |  |
| 40073 | A read conflicts with drop of ? | n | y |  |
| 40074 | A delete conflicts with update of ? | n | y |  |
| 40075 | A new reference conflicts with deletion of ? | n | y |  |
| 40076 | A conflicting domain or type ? has just been defined | n | y |  |
| 40077 | A conflicting change on ? has just been done | n | y |  |
| 40078 | Read conflict with alter of ? | n | y |  |
| 40079 | Insert conflict with alter of ? | n | y |  |
| 40080 | Update conflict with alter of ? | n | y |  |
| 40081 | Alter conflicts with drop of ? | n | y |  |
| 40082 | ETag validation failure | n | y |  |
| 40083 | Secondary connection conflict on ? | n | y | Remote connection snapshots differ |
| 42000 | Syntax error or access rule violation at ? | y | y |  |
| 42101 | Illegal character ? | n | y |  |
| 42102 | Name cannot be null | n | y |  |
| 42103 | Key must have at least one column | n | y |  |
| 42104 | Proposed name conflicts with existing database object (e.g. table already exists) | n | y |  |
| 42105 | Access denied ? | n | y |  |
| 42107 | Table ? undefined | n | y |  |
| 42108 | Procedure ? not found | n | y |  |
| 42109 | Assignment target ? not found | n | y |  |
| 42111 | The given key is not found in the referenced table | n | y |  |
| 42112 | Column ? not found | n | y |  |
| 42113 | Multiset operand required, not ? | n | y |  |
| 42115 | Unexpected object type ? ? for GRANT | n | y |  |
| 42116 | Role revoke has ADMIN option not GRANT | n | y |  |
| 42117 | Privilege revoke has GRANT option not ADMIN | n | y |  |
| 42118 | Unsupported CREATE ? | n | y |  |
| 42119 | Domain ? not found in database ? | n | y |  |
| 4211A | Unknown privilege ? | n | y |  |
| 42120 | Domain or type must be specified for base column ? | n | y |  |
| 42123 | NO ACTION is not supported | n | y |  |
| 42124 | Colon expected .. | n | y |  |
| 42125 | Unknown Alter type ? | n | y |  |
| 42126 | Unknown SET operation | n | y |  |
| 42127 | Table expected | n | y |  |
| 42128 | Illegal aggregation operation | n | y |  |
| 42129 | WHEN expected | n | y |  |
| 42131 | Invalid POSITION ? | n | y |  |
| 42132 | Method ? not found in type ? | n | y |  |
| 42133 | Type ? not found | n | y |  |
| 42134 | FOR phrase is required | n | y |  |
| 42135 | Object ? not found | n | y |  |
| 42138 | Field selector ? not defined for ? | n | y |  |
| 42139 | :: on non-type | n | y |  |
| 42140 | :: requires a static method | n | y |  |
| 42142 | NEW requires a user-defined type constructor | n | y |  |
| 42143 | ? specified more than once | n | y |  |
| 42146 | OLD specified on insert trigger or NEW specified on delete trigger | n | y |  |
| 42147 | Cannot have two primary keys for table ? | n | y |  |
| 42148 | FOR EACH ROW not specified | n | y |  |
| 42149 | Cannot specify OLD/NEW TABLE for before trigger | n | y |  |
| 42150 | Malformed SQL input (non-terminated string) | n | y |  |
| 42151 | Bad join condition | n | y |  |
| 42152 | Non-distributable where condition for update/delete | n | y |  |
| 42153 | Table ? already exists | n | y |  |
| 42154 | Unimplemented or illegal function ? | n | y |  |
| 42156 | Column ? is already in table ? | n | y |  |
| 42157 | END label ? does not match start label ? | n | y |  |
| 42158 | ? is not the primary key for ? | n | y |  |
| 42159 | ? is not a foreign key for ? | n | y |  |
| 42160 | ? has no unique constraint | n | y |  |
| 42161 | ? expected at ? | n | y |  |
| 42162 | Table period definition for ? has not been defined | n | y |  |
| 42163 | Generated column ? cannot be used in a contraint | n | y |  |
| 42164 | Table ? has no primary key | n | y |  |
| 42166 | Domain ? already exists | n | y |  |
| 42167 | A routine with name ? and arity ? already exists | n | y |  |
| 42168 | AS GET needs a schema definition | n | y |  |
| 42169 | Ambiguous column name ? needs alias | n | y |  |
| 42170 | Column ? must be aggregated or grouped | n | y |  |
| 42171 | A table cannot be placed in a column | n | y |  |
| 42172 | Identifier ? already declared in this block | n | y |  |
| 42173 | Method ? not defined | n | y |  |
| 44000 | With check option violation | y | y |  |
| 44001 | Domain check ? fails for column ? in table ? | n | y |  |
| 44002 | Table check ? fails for table ? | n | y |  |
| 44003 | Column check ? fails for column ? in table ? | n | y |  |
| 44004 | Column ? in Table ? contains null values, not null cannot be set | n | y |  |
| 44005 | Column ? in Table ? contains values, generation rule cannot be set | n | y |  |
| HZ000 | Remote Database Access error | y | n |  |

# Appendix 6: Pyrrho’s System Tables

There are three sets of system tables, and these give read-only access using SQL for the database owner to internal information held in the DBMS engine as seen from the current transaction. For more details, see the Pyrrho Manual Pyrrho.docx. Since the names of these tables contain a $, they need to be enclosed in double-quotes in SQL. These system tables are for a single current database; the Role$ tables give the properties as seen from the current role; while the Log$ tables give access to the defining records.

All of these tables are virtual in the sense that the contents are generated in this form only transiently and when required from the DBMS engine’s internal data structures.

|  |  |
| --- | --- |
| **System Table Name** | **Type of information** |
| Sys$Audit | Access to audit records created in accordance with enforcement settings |
| Sys$AuditKey | Details of audit records |
| Sys$Classification | Record classification information for current records (levels A,B,C). Level D=unclassified is not included in this table |
| Sys$ClassifiedColumnData | Column classification supplementing the above |
| Sys$Enforcement | As specified in Mandatory Access Control |
| Sys$Role | A list of all roles known for this database |
| Sys&RoleUser | A list of users granted each role |
| Sys$ServerConfiguration | The current set of configuration properties for this server |
| Sys$User | A list of all users known in the current database |
| Role$Class | Base tables, key columns, and C# class definitions |
| Role$Column | Columns of base tables |
| Role$ColumnCheck | Column check constraints |
| Role$ColumnPrivilege | Role access privileges on table columns |
| Role$Domain | Domain and type information accessible from the current role |
| Role$DomainCheck | Domain check constraints |
| Role$Index | With standard SQL indexes are created to implement integrity and referential constraints |
| Role$IndexKey | Key information for indexes |
| Role$Java | Base tables, key columns, and Java class definitions |
| Role$Method | Methods defined for user-defined types |
| Role$Object | Metadata for database objects |
| Role$Parameter | Parameter information for user-defined methods and procedures |
| Role$PrimaryKey | Primary key information for base tables |
| Role$Privilege | Access privileges that have been granted on database objects |
| Role$Procedure | User-defined Procedures |
| Role$Python | Base tables, key columns, and Python class definitions |
| Role$Subobject | Metadata for database subobjects (e.g. view columns) |
| Role$Table | Base tables in the current database |
| Role$TableCheck | Table check constraints as currently defined |
| Role$TablePeriod | User-defined periods |
| Role$Trigger | User-defined triggers |
| Role$TriggerUpdateColumn | Column update information specified for user-defined triggers |
| Role$Type | User-defined types |
| Role$View | User-defined Views |
| Log$ | The transaction log: a readable synopsis of each entry in the log |
| Log$Alter | Details for ALTER records in the log (see also CHANGE, EDIT, MODIFY) |
| Log$Change | Details for CHANGE records in the log |
| Log$Check | Details for user-defined check conditions |
| Log$Classification | Details of all log entries that change data classification of objects. (See also Log$Update for classification of records) |
| Log$Column | Details for column definitions |
| Log$Conflicts | Details of conflicting log entries for each log entry |
| Log$DateType | Details for user-defined date types |
| Log$Delete | Details for DELETE records in the log |
| Log$Domain | Details for DOMAINS specified in this database |
| Log$Drop | Details for DROP entries in the log |
| Log$Edit | Details for EDIT records in the log |
| Log$Enforcement | Details of alterations made to enforcement for tables etc |
| Log$Grant | Details for GRANT records in the log |
| Log$Index | Details for INDEX records in the log (when indexes are created) |
| Log$IndexKey | Details for keys of indexes |
| Log$Metadata | Details of METADATA entries in the log |
| Log$Modify | Details of MODIFY entries in the log |
| Log$Ordering | Details of user-defined orderings |
| Log$Procedure | Details of user-defined procedures |
| Log$Record | Details for records in the log (insert and update) |
| Log$RecordField | Details for records in the log (insert and update) |
| Log$Revoke | Details of REVOKE records in the log |
| Log$Role | Details of changes to the Role object |
| Log$Table | Details of metadata for user-defined tables |
| Log$TablePeriod | Details of table period definition |
| Log$Transaction | Details of all transactions in the Log |
| Log$TransactionParticipant | Details of participants for distributed transactions |
| Log$Trigger | Details of user-defined triggers |
| Log$TriggeredAction | Details of changes made to the database by triggers |
| Log$TriggerUpdateColumn | Update column details for user-defined triggers |
| Log$Type | User-defined types. See also Log$Domain, and Log$TypeMethod |
| Log$TypeMethod | Details of methods defined for user-defined types |
| Log$Update | Details of UPDATE records in the log (See also Log$Record) |
| Log$User | Details of USER records in the log |
| Log$View | Details of VIEW definitions in the log |
| ROWS(*nnn*) | Details of operations involving table defined at *nnn* in the log |
| ROWS(*rrr*,*ccc*) | Details of operations involving the table cell at the row defined at *rrr* and column defined at *ccc* in the log. |
| Profile$ | If profiling is turned on these tables are available: profile identity |
| Profile$ReadConstraint |  |
| Profile$Record |  |
| Profile$RecordColumn |  |
| Profile$Table |  |

# Appendix 7: Pyrrho’s Connection String

Pyrrho allows multi-database connections (with some limitations: see section 2.8.3). More than one database on the same server can be specified using a comma-separated list of Files, but two mechanisms are provided for more complex scenarios. In Android and embedded editions of Pyrrho, the Host and Port fields are not provided, but remote hosts and ports can be specified for a database using the syntax *file*@*host*[:*port*:[*user*[:*role*]]] . And the connections string can be compound using a Json-like notation, see below.

By default, only the first database in a multi-database connection is modifiable using the connection, but the Modify flag is available for changing this behaviour. In addition, a Role can be specified per database.

The syntax of ConnectionString is similar to ADO.NET.

ConnectionString = Setting {‘;’Setting} .

Setting = id’=’val{‘,’val} .

The possible fields in the connection string are as follows:

|  |  |  |
| --- | --- | --- |
| **Field** | **Default value** | **Explanation** |
| Files |  | One or more comma-separated database file names  (excluding the .pfl or .osp extension). Characters ,;= within names are not allowed. If the only field in the connection string is Files, the prefix Files= can be omitted. |
| Host | 127.0.0.1 | The name of the machine providing the service. |
| Locale |  | The name of the locale to be used for error reporting. The default is supplied by the .NET framework. |
| Modify |  | The default value is true for the first file in the connection, and false for others. If the value true is specified then it applies to all of the Files in the current connection string. |
| Port | 5433 | The port on which the server is listening |
| Provider | PyrrhoDBMS |  |
| Role | databasename | A role name selected as the session role. If this field is not specified, the session role will be the default database role if the user is the database owner or has been granted this role (it has the same name as the database), or else the guest role, which can access only PUBLIC objects. |
| Stop |  | If a value is specified, this means that Pyrrho is to load the database as it was at some past time. |
| User |  | This field is supplied by infrastructure |

A CompoundConnectionString format is available for complicated multi-file connections.

CompoundConnectionString = ConnectionString | ‘[{‘ ConnectionString {‘},{‘ ConnectionString }‘}]’ .

If *cs1* and *cs2* are ConnectionStrings as specified above, then [{*cs1*},{*cs2*}] is permitted. For example [{A},{Files=B,Modify=true},{C}] . For this purpose the 3 separator sequences “[{“ “},{“ “}]”are assumed not to occur anywhere in the individual connection strings (in particular they must not themselves be compound). The User field is supplied by infrastructure for the first of these connection strings but applies to all.

# Appendix 8. Demonstrating the shareable B-Tree data structure

# Appendix 9. The TPCC-C benchmark

# Appendix 10. Pyrrho’s transaction commit process

# Appendix 11. Localisation and Globalisation

The Pyrrho DBMS file format makes very few assumptions about machine architecture beyond mandating Unicode characters (UTF-8 encoding), universal time, and the use of 8-bit bytes. The SQL standard mandates a standard format for dates and times.

The Pyrrho server for Windows and Linux/Mono uses native 64-bit longs but these are encoded as byte arrays for writing to the database file. It is a very simple matter to modify the server for a different machine architecture. The client library localises these to the locale of the client, which may be different from that of the server. Support for multiple cultures is mandated in the SQL standard, and Pyrrho supports these for database operations. In this way Pyrrho database files are independent of locale and machine architecture.

However, users of database products expect more (while retaining the neutral culture for SQL itself):

* Localisation of server error messages and warnings (by client locale)
* Localisation of client utility error messages and prompts, and user interface (e.g. a right-to-left interface)
* Localisation of the server’s system tables (e.g. column names in Chinese)

Early editions of Pyrrho included language some support for languages including Chinese and Arabic There are some clues in the manuals and source code as to how this missing localisation files can be reinstated/.

# Index to Syntax

|  |  |
| --- | --- |
| AbsoluteValue, 194  Action, 187  AddPeriodColumnList, 184  Alias, 190  Alter, 182  AlterBody, 183  AlterCheck, 183  AlterColumn, 183  AlterDomain, 183  AlterMember, 184  AlterOp, 183  AlterTable, 183  AlterType, 184  AlterView, 184  Any, 193  Assignment, 196  AttributeSpec, 198  Authority, 211  Avg, 194  Between, 193  BinaryOp, 192  BooleanExpr, 193  BooleanFactor, 193  BooleanTerm, 193  BooleanTest, 193  Call, 197  Cardinality, 196  CaseStatement, 197  Ceiling, 194  CharacterType, 188  CheckConstraint, 183  Close, 197  Coalesce, 194  Collate, 188  Collect, 196  ColRef, 186  Cols, 186  ColumnConstraint, 185  ColumnConstraintDef, 185  ColumnDefinition, 185  ColumnOption, 186  ColumnOptionsPart, 186  ColumnRef, 192  Comparison, 193  CompOp, 193  CompoundStatement, 197  CondInfo, 197  Condition, 197  ConditionCode, 197  ConditionList, 197  Contains, 193  Correlation, 194  Count, 194  Covariance, 195  Create, 184  CursorSpecification, 189 | DatabaseError, 206  Date, 207  DateTimeField, 193  DateTimeFunction, 196  DateTimeType, 189  Declaration, 197  Default, 183  DefinedType, 189  DeletePositioned, 189  DeleteSearched, 189  DomainDefinition, 185  DropAction, 187  DropObject, 187  DropStatement, 187  Element, 196  EndField, 193  Event, 186  Every, 193  Exclusion, 190  Exists, 193  Exponential, 195  Extract, 195  ExtractField, 195  Fetch, 198  FetchFirstClause, 191  Files, 211  FloatType, 188  Floor, 195  Fold, 196  ForStatement, 198  FromClause, 190  FuncOpt, 194  FunctionCall, 194  Fusion, 196  GenerationRule, 185  GetDiagnostics, 197  GetFileNames, 208  Grant, 187  Grantee, 188  GranteeList, 188  GroupByClause, 190  Grouping, 195  GroupingSet, 190  GroupingSpec, 190  HandlerType, 197  HavingClause, 190  Host, 211  How, 198  HttpRest, 182  IfStatement, 198  In, 194  Insert, 189  IntegerType, 188  Intersect, 196  IntervalField, 189, 193  IntervalQualifier, 193 |

|  |  |
| --- | --- |
| IntervalType, 189  ItemName, 197  JoinedTable, 191  JoinType, 191  Label, 198  Last, 195  LengthExpression, 195  Like, 194  Like\_Regex, 194  Literal, 192  LobType, 188  Locale, 211  LoopStatement, 198  Maximum, 195  Member, 184, 194  Metadata, 121, 184  Method, 182  MethodCall, 198  MethodType, 183  Minimum, 195  Modulus, 195  MultisetOp, 192  NamedValue, 198  Namespace, 198  NamespaceDefault, 198  NaturalLogarithm, 195  Next, 195  Normalize, 196  Null, 194  Nullif, 195  NumericType, 189  NumericValueFunction, 194  ObjectName, 187  ObjectPrivileges, 187 Occurrences\_Regex, 195 **of**, 70 Of, 194  Open, 198  OrderByClause, 191  Ordering, 185  OrderSpec, 191  OrdinaryGroup, 190  Parameter, 183  Parameters, 182  PartitionClause, 190  Percentile, 195  PeriodBinary, 194  PeriodName, 186  Port, 211  Position, 195  Position\_Regex, 195  PowerFunction, 195  Predicate, 193  PreiodPredicand, 194  Privileges, 187  Provenance, 196  Provider, 211  PyrrhoArray, 206, 207  PyrrhoConnect, 206  PyrrhoInterval, 206, 209 | PyrrhoRow, 206, 209, 210  QueryExpression, 189  QueryExpressionBody, 189  QueryPrimary, 189  QuerySpecification, 189  QueryTerm, 189  Rank, 195  ReferentialAction, 186  RefObj, 186  Regression, 195  Rename, 187  Repeat, 198  Representation, 185  ResetReader, 209  Revoke, 187  RollCube, 190  Routine, 188  RowNumber, 195  Schema, 195  SearchCondition, 191  SelectItem, 189  SelectList, 189  SelectSingle, 198  Set, 196  SetAuthority, 209  SetFunction, 196  Signal, 197  Similar, 194  SimpleTable, 189  Some, 194  Sql, 181  SqlStatement, 181  SquareRoot, 196  StandardDeviation, 196  StandardType, 188  StartField, 193  Statement, 181  Statements, 198  StringValueFunction, 196  Subquery, 191  Substring, 196  Substring\_Regex, 196  Sum, 196  TableClause, 185  TableConstraint, 186  TableConstraintDef, 186  TableContents, 185  TableExpression, 189  TableFactor, 190  TablePeriodDefinition, 186  TableReference, 190  Target, 197  TicksPerSecond, 209  TimePeriodSpecification, 191  TransactionConflict, 206  Translate\_Regex, 196  Trigger, 186  TriggerCond, 187  TriggerDefinition, 186  Trim, 196 |

|  |  |
| --- | --- |
| Type, 188  Typecast, 196  TypedTableElement, 186  Unique, 194  UpdatePositioned, 189  UpdateSearched, 189  UriType, 185  UserFunctionCall, 198  Value, 191  Values, 186  VariableRef, 192  Variance, 196  VersioningClause, 185  ViewDefinition, 186  WhereClause, 190  While, 198  WindowBetween, 190  WindowBound, 190  WindowClause, 190  WindowDef, 190  WindowDetails, 190  WindowFrame, 190 | WindowSpec, 195  WindowStart, 190  WithinGroup, 195  XmlAgg, 196  XmlColumn, 191  XmlColumns, 191  XMLComment, 198  XMLConcat, 198  XMLDec, 197  XMLDocument, 198  XMLElement, 198  XMLForest, 198  XMLFunction, 198  XMLNDec, 183  XMLOption, 183  XMLParse, 198  XMLProc, 199  XMLQuery, 199  XMLText, 199  XMLValidate, 199  XSL, 73, 121 |

215

[View publication stats](https://www.researchgate.net/publication/275645497)

1. However, it is a good idea to have a separate folder for the databases. It is simplest to copy the server executable to the folder you plan to use. See section 1.2. [↑](#footnote-ref-1)
2. This design decision in Pyrrho is discussed further in section 3.7. [↑](#footnote-ref-2)
3. Compensation mechanisms should be supported by the DBMS if required by business logic. But they should not be introduced merely because the DBMS does not support Web applications or transactions properly. Pyrrho offers row-version checking and other “forensic” methods to explore the dependency of later events on particular transactions, but the use in the literature of phrases such as “automatic compensation” merely indicates poor transaction design or support. [↑](#footnote-ref-3)
4. Both are shown as -1 until the log records the definition of the first role or user. [↑](#footnote-ref-4)
5. However, if such a transaction is explicit (beginning with begin transaction and ending with commit) the commit will report conflict if another transaction has changed anything it has read. [↑](#footnote-ref-5)
6. The end-of-file marker includes a kind of digital signature to guard against tampering with the database contents. [↑](#footnote-ref-6)
7. The idea that all internals of the database engine should be exposed in relational tables is a consequence of Codd’s (1985) principles. Since Pyrrho’s transaction logs are durable, all their details are exposed in system tables. See Appendix 8. [↑](#footnote-ref-7)
8. The SQL standard has a detailed and more complex proposal for adding JSON objects to relational databases. [↑](#footnote-ref-8)
9. Pyrrho requires single quotes to enclose string literal: but the standard in Json is to use double quotes. Field names in JSON are case sensitive, so for simplicity here we used upper case, to match the default case in SQL. (Official documents about Json sometimes suggest that field names should be enclosed in quotes.) [↑](#footnote-ref-9)
10. In other DBMS long integers are called BIGINT. [↑](#footnote-ref-10)
11. The square brackets surround SQL input that takes more than one line [↑](#footnote-ref-11)
12. The optional keywords here are less restrictive than might appear: In this syntax views and tables can be used interchangeably, so that the keyword **table** if present may be followed by a view. Similarly, the keyword **procedure** if present may be followed by a function call. [↑](#footnote-ref-12)