Analgesia safety checklist: PDA implementation details

Part I.

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

1 Introduction

This document contains details of the database underlying our PDA-based Analgesia application. Readers are assumed to have read the preceding two documents in the series:

1. *The Analgesia Safety Checklist*
2. *PDA data capture based on a form template*

We also assume that you understand the fundamentals of relational databases (particularly SQL), what an entity class is, normalisation (first, second and third normal form), and notation for time complexity of a function, i.e. things like $O(n.\log(n))$.[1]

This documentation and code is released under the GNU Public Licence (GPL), a copy of which can be obtained at: http://www.gnu.org/copyleft/gpl.html. Please note the conditions of this licence. This document is Copyright ©J van Schalkwyk, 2005–2007.

The SQL component of the PDA program fall into four logical components. The first two are more general, and deal with data; the second pair specifies how the data are presented to the user on both the PDA and the desktop machine:

1. Creation of fundamental data tables (These tables permit modelling of medical processes);
2. Creation and population of analgesia-specific tables;
3. Creation of tables which specify user-interface menu structure;
4. User interface details.

This document (Part I) covers only the first pair. The SQL statements dealing with the specifics of the user interface are covered in Part II. Before we consider the data tables and their population for the purposes of the Analgesia program, we must first identify *entity classes* of interest. This is done in the following section (3), followed by the creation of tables relevant to management of these entity classes.

[1] Sometimes called 'Big-O notation'. See e.g. http://en.wikipedia.org/wiki/Time_complexity

1.1 A note on syntax

Before we discuss SQL, note that we are using DogWagger 2.0 to integrate our documentation and code. One small 'idiosyncrasy' of DogWagger is that if you are creating a single line of code from multiple lines of source code, *leading* spaces are suppressed, but trailing ones aren't. This allows indentation of source code *and* insertion of whitespace. Some text editors by default (e.g. WinEdt) suppress trailing spaces, so this feature must be disabled in the Word processor,[2] as must wrapping of verbatim statements.

The SQL syntax in our specification could be considered somewhat 'retro'. In general, we have kept to SQL-92 style statements. In particular, we have avoided 'new' join syntax, mainly because this is implemented in different ways in different databases. We also avoid compound primary keys.[3]

Our own personal naming conventions are somewhat idiosyncratic — keys and foreign references are named to allow more modern joins, but for all other table components we allocate a different name to each database column, and every column within a particular table begins with the same letter. See the usage for an idea of how this works.[4]

In addition, we generally represent table names in UPPERCASE, columns in *italics*.

There is *one* 'feature' worth noting. We use an *integer* type for all primary keys. Unfortunately, in SQL we have absolutely no guarantee that on various systems, a key of a particular bit length will be supported. We thus limit all such keys to values between 0 and (1 billion minus 1).[5]

1.2 A philosophical note

Some of our data manipulation is bound to get up the nose of the SQL purist.[6] Noting that SQL is not Turing-complete, we have the awkward options of embedding arbitrary code within SQL, relying on rather arbitrary extensions to SQL, or the option we choose, which is embodied in the following statement:

```
  "Even if a hippo goes to Fred Astaire for a hundred
```

[2]For WinEdt, say Document:Document Settings—Defaults—Trim Spaces.

[3]This latter idiosyncracy is technically a violation of third normal form, but *effectively* we do preserve 3NF, and having a single primary key per table has massive benefits, especially in the context of our stripped-down PDA SQL.

[4]Heck, rename them if you find our format unpalatable!

[5]Federal Information Processing Standards (FIPS 127-1, now 'withdrawn') mandate an integer size of at least 9 decimal digits, which fits our choice rather well.

[6]Other parts may be just plain wrong — my SQL abilities are limited.

```
  years, it will still never be chosen as a
  dance instructor"
```

In other words, let SQL do the straightforward SQL; use external code to do the rest. We use our own *scripting language* to tie SQL statements together and link them to our user interface! This approach is admittedly a blessing and a curse. We can do what we want, but at the expense of introducing our own scripting language, which by design runs in the same fashion on both PDA and desktop, the latter as a Perl program.

The presence of the scripting language does *not* impair users when they query the SQL database on the desktop. We have taken some pains to make both our SQL code, and more importantly, our database, as vanilla-flavoured as possible.

A consequence of our scripted data entry into the database is that we devolve fancy data formatting and checking to the scripting, rather than having the SQL throw out frequent exceptions as it chokes on data. A certain amount of co-operation, trust, and lack of hippos is necessary!

1.3 Another crippling SQL limitation

Another significant limitation of SQL is its lack of constants (replaced if you desire by creating constant functions, that is, you can define a function called 'Pi()' which when used will return a constant value). We toyed with the idea of either creating yet another layer of SQL pre-processing for our .SQL files, or alternatively defining a whole bunch of cumbersome constant functions, but rejected both approaches as unnecessarily complex.

Why might we want such constants in the first place? Especially when you get to Part II (*AnalgesiaDB2.tex*) you'll see that there is a crying need for named constants to replace a host of numeric values. For example to create an [Exit] button in a menu, we might wish to say something like:

```
INSERT INTO ITEM (iID, iType, iText)
  VALUES (123, button, 'Exit');
```

. . . but instead, owing to the lack of constant values, we must say:

```
INSERT INTO ITEM (iID, iType, iText)
  VALUES (123, 2, 'Exit');
```

. . . which is far less explicit as we now have to 'understand' that the code value 2 refers to a button and not, say, a text field, checkbox or popmenu. Rather sad.

2 Overall database structure

Our database is relatively simple and can be described in a few sentences! Central is the concept of a PROCESS, which refers to a PERSON (‘Subject’), is a particular type of process, and has defined start and end times. OBServations are made on processes, and various data are associated with each ‘EPOCH’.

2.1 A concise schema

Here’s a brief schema of our table structure. Any table A references table B if there’s an arrow pointing from A to B. Tables with a star* after them are deliberately denormalised.

```
  PERSON <---)
PROCTYPE <---)PROCESS|<-EPOCH|<---ACTOR2(-->AROLE
                     |       |          (-->PERSON
                     |       |<---PAINERROR-->errtype
                     |       |<---PERSDATA*-->person
                     |       |<---MEDSCORE
                     |       |<---SURGTYPEOB-->SURGTYPE
                     |       |<---NONEVENT
                     |       |<---MEASURE
                     |       |<---PAINSCORE
                     |       |<---RXOBS
                     |       |<---INFUSIONOBS
                     |       |<---PCA
                     |       |<---PCASETTINGS
                     |       |<---RGNOBS
                     |       |<---COMMENT
                     |       |<---ISPROBLEM
                     |       |<---BADOBS*(-->person
                     |                   (-->BED-->ROOM-->WARD
                     |<--STOPPROC|<--WHYSTOP
                     |<---)
                          )RX
    DRUGFORM<---)DRUG <---)
```

Despite appearances, ACTOR2 is not denormalised as the left hand reference is to a subject (patient) via EPOCH and PROCESS, and the right-hand reference is to a role player different from the patient.[7]

2.2 Brief table descriptions

Various people apart from the Subject may also be associated with a process, and we describe them and their roles using an ACTOR table. Roles are described in the AROLE table.

In addition we make observations on processes via the EPOCH table. Each 'EPOCH' actually represents an epoch in the lifespan of the process, usually during a user session when they visit various patients. The EPOCH has a creation (start) time, as well as an identified observer, and refers to one particular process.[8]

Particular processes are specially important to us. These types of processes include hospital admission and drug administration via a variety of routes, including epidural, intravenous and oral.

A multiplicity of tables references the 'EPOCH' epochs. One extremely important such table is the *BED observation* called BADOBS,[9] so named because of a convenient denormalisation we have introduced — the table not only references a bedspace and an EPOCH on an admission process, but also contains the 'Person' field which directly references a particular subject (patient). This denormalisation speeds certain queries enormously, as will be seen.

Several DRUGs may be associated with a particular process. In addition, contemporaneous processes might exist, so we model the presence of an epidural in sombody's back as an epidural process, and a distinct epidural *infusion* process which is associated with one or more drugs. At present we have no complex structural dependencies or associations between such processes, rather constraining such associations through software constructs and checks.

At the end of this section we list many small subsidiary tables which reference EPOCH, but first let's look at drug administration. We use a table called RX to associate a given DRUG with a particular PROCESS. The RX table is fairly flexible (perhaps too flexible) in that it also specifies a rate, concentration and even (alternatively) a dose and interval. We subsequently create drug OBServations on the relevant PROCESS, and then, using the RXOBS table we record daily total doses. For infusions we have a very specific table called INFUSIONOBS which simply records a rate and concentration.

Things become a little more complex because of the existence of Patient Controlled Analgesia (pca) — to record pca details, we require a separate PCA table

[7]The ACTOR2 table is currently disabled in our database, but we've retained the reference for potential future development'.

[8]We now also indirectly record the EPOCH end time!

[9]Formerly we called the 'EPOCH' table 'OBS'.

which references an EPOCH on a pca PROCESS, and contains the number of tries and hits derived from the pca pump in the past 24 hours. In addition we need to record on a daily basis what the actual pca settings are, for which we use the PCASETTINGS table, which also references an EPOCH on the PCA PROCESS!

Regional procedures also have a particular table associated with them: RGNOBS documents site checks, particularly for epidurals.

Each DRUG table entry describes the trade name and formulation of the drug. Previously we referenced the PHARM table which contains the drug's generic name, but this requirement has been removed. (See *AnalgesiaDB2.tex* for the rationale behind this simplification).

Finally, here is the promised list of 'minor' tables linked to EPOCH:

1. PERSDATA, describing properties of individual people;
2. MEDSCORE, allocated to scoring systems, in particular ASA rating;
3. SURGTYPEOB, describing the type of surgery in broad detail (a subsidiary table is SURGTYPE);
4. NONEVENT, a slightly odd table which documents the *observed* absence of something (for example, 'no epidural'!)
5. MEASURE used to describe body measurements such as weight;
6. PAINSCORE, used to document pain on rest and movement;

Finally, any EPOCH entry can have one or more general comments associated with it, although we discourage use of such comments to record analysable data!

2.3 PDA considerations

The limiting factor in our database will be PDA speed, more than anything else. Memory size is also a consideration, but should be secondary, especially as modern PDAs have at least several tens of megabytes of memory. The executable footprint of our database will be in the region of just 200 K or less, and we anticipate that our usual database will occupy under a megabyte *in toto*.

Let's look at the size issue in a little more detail. We assume the following:

1. Each PDA will have no more than about fifty patients on it at any one time. It is unreasonable to assume that the user of the individual PDA will be concerned with management of more than about 50 patients;[10]

[10]You can see from the following that a hundred or so shouldn't be a problem.

2. Each patient will, *on average* spend five days on the pain service. Except in high acuity settings with complex pain patients, this assumption seems reasonable, or even excessive.
3. There will be about fifty (or fewer) PDA-documented observations on a single patient on a single day.[11]
4. Most data items will be simple menu-driven choices, allowing storage in four or fewer bytes.

Based on the above, we will store about 200 bytes per patient per day, a maximum of 10 K per day, and need to retain just 50 K of patient data in the PDA at any one time, a number which might be increased several-fold if we liberally use time-stamps and so forth.

Let's now consider data storage and retrieval on the PDA. If we have a proper relational database implemented on the PDA (and this is most desirable), then we may well have queries involving three or more joins on tables. Let's consider the following model (which is *not* exactly the one we will use):

1. Several simultaneous processes involving a particular patient are modelled using a PROCESS table;
2. Observations of such processes (for example, observations of pain scores associated with the process 'admission under the analgesia team') are recorded in an OBSERVATION table.

If there are (at worst case) fifty patients and fifty observations, then we end up with 2500 observations per day, or 12 500 over five days. Remembering that we can only conveniently fit 64K into a PalmOS database file (although databases can be extended over several such chunks), this implies that each observation would have to be pretty tiny. But even such considerations are secondary to considerations of speed.

If we refer to the patient (by their unique database ID) in PROCESS, and retain a datum in OBSERVATION, then we need to consider several 'speed' issues:

- Adding new data items should not be a problem, provided we order items by a progressively incrementing key.[12] The new item is placed at the top of the file where it belongs;

[11]In many circumstances, far fewer; fifty a day is pushing it in terms of user compliance with data entry!

[12]Note however that we will not use autoincrementing keys per se, as these are not core SQL components, and different databases implement these differently!

- Deletions, and re-ordering of whole tables will be infrequently used on the PDA (if at all).[13]

- Rapidly retrieving recorded data may be a significant issue, if handled poorly. Let's look at this retrieval in more detail.

Let's assume we have about 10 000 records in the OBSERVATION table, and (say) ten processes per patient, that is, 500 processes active in the PROCESS table. If we perform a join on the two tables, the Cartesian join gives us 5 million items, but we can optimise our search a bit.

On a machine with the entire database in RAM (as occurs with most current PDAs), let's analyse an arbitrary query along the lines of:

```
SELECT MAX(OBSERVATION.value)
  FROM OBSERVATION, PROCESS
WHERE
  OBSERVATION.process = PROCESS.prockey AND
  PROCESS.patient = 1 AND PROCESS.proctype = 2
  AND OBSERVATION.obstype = 34
```

If we start with the PROCESS table, then we only have 500 records to examine in order to identify the relevant process; we can then join on the OBSERVATION table to find the desired observation(s). A small problem is that we then need to either repeatedly plod through each of 10 000 records, or we require an index, in which case we could perform a binary search, generally requiring just $\log_2(n)$ record comparisons,[14] to locate a relevant record. Note however that the relationship is 1:many, so we would then still have to examine a fair number of adjacent records for relevance. *In addition*, there is the overhead of maintaining the index, which is substantial, in that keeping the ordering of items in the index requires significant movement of indices around in memory.

An alternative approach is to start with the OBSERVATION table — an initial plod through this, followed by a quick search on PROCESS (which search doesn't require a separate index, and is also $O(\log n)$. The drawback is the number of records in OBSERVATION. If we have many observations on the PDA screen for a particular patient, there might be unacceptable delays in populating these fields.

A further problem is representation of data within the OBSERVATION table, where we are faced with several unpalatable options — numerous data fields, one

[13]In fact, we will not delete previously recorded items; if a user wishes to delete an item, we will merely flag the item as 'deleted by the user' and retain it — this is good medical database practice.

[14]where n is the number of records in OBSERVATION

per datum; or cumbersome mapping of data into a selection of fields; or ugly text representation of all data!

All of the above leads us where we might initially have suspected we were going — individual menus for observations, either referring directly to processes, or perhaps indirectly via an OBSERVATION table. The OBSERVATION table might be made smaller if we appreciate that we aren't really concerned with second-by-second accuracy for each and every datum: data observed in one menu might reasonably be stamped with the same timestamp (This approach would also have the advantage of removing similar 'duplicate' data from individual menus devoted to particular data items).

Working from the 'bottom up' in our data retrieval now requires examination of perhaps a few hundred items in the relevant menu, followed by sequential joins onto OBSERVATION and PROCESS, each requiring *no* index, and $O(\log n)$ search times.[15] We can also decrease the size of the OBSERVATION table by several fold, if we group related items in menus which depend on this table.

Although search-time wise it's probably quite reasonable to have a vast number of tiny tables, one for each datum, we will take the middle ground and group logically related items in their own tables.[16]

[15] Note that in real implementation, the optimised index-free join on three tables is $O(k^2 . \log_2(p) . \log_2(o) . n)$ where p,o, and n are respectively the number of items in the PROCESS, OBSERVATION and DATUM tables, and k is a constant describing the clumsiness or otherwise of binary table searches.

[16] Initially I did the one-table, lumped together, everything as text approach, but on reflection, such an approach is probably a little silly!

3 SQL entity classes

There is a very short list of important entity classes. These are:

1. person;
2. process;
3. epoch;
4. drug;
5. BED (with the associated room and ward entities)

There should also be a **device** entity class, but in this implementation we will not go to the length of recording each device used for PCA or regional infusions.[17] We will explore the minimalist nature of the above five entity classes as we work through them, but important ideas are as follows:

- A person is a person; their *role* is very much subservient to their being a person.
- Description of what is going on is very much patient-centric. Most on-going, well, processes involving that patient can be represented under the PROCESS rubric. There can be many concurrent processes involving one patient.
- *Observations* are always attached to processes, along the time-line of the process;

We consider each entity class under the corresponding table, in the following section. It's worth noting that we have no entity class for a **diagnosis**! This is mainly because at present our database does not contain sufficient detail to accurately associate a diagnosis with the relevant corroborative (primary) evidence. A statement of diagnosis is always an opinion; we believe that many current databases err because when they assert a 'diagnosis', they provide neither corroborative evidence, nor do they identify the person making the diagnosis. You will also see that costing, billing codes etc are not represented in our application. Our focus is that of a *clinical* database.

[17]It seems logical to represent the device in the PROCESS table, implying that if the device changes, we stop that process and replace it with a new one associated with the new device.

4 Fundamental data tables

Data tables correspond to our entity classes.

4.1 person

It makes sense to represent all people within the database, as, well, people. This approach also prevents the anomaly where 'patients' and 'staff' are represented in different tables and then — lo and behold — a staff member becomes a 'patient', introducing an uncomfortable denormalisation.

Each person requires a unique identifier. It is *very* unwise to use an externally available identifier such as the hospital number as our unique identifier, because multiple hospital numbers are sometimes attached to the same patient, despite attempts to prevent such actions. We thus generate a unique ID internally for each patient.

As we don't use auto-incrementing fields (not part of the SQL standard), we need an 'external' mechanism for such key generation, which topic is discussed in Section 14.3 below. The problem of unique IDs also touches on the problem of synchronisation. This is covered in section 14.4.

Here's our minimalist PERSON table:

```
CREATE TABLE PERSON (
  cold integer,
  person integer,
      constraint badPersonKey primary key (person),
  pBorn timestamp,
  pDied timestamp,
  pMade timestamp,
  pTokens decimal(32,0),
  pStatus integer,
  pUserName varchar (32),
  pFlags integer default 0
              );
```

Note the unique ID (*person*), as well as timestamps of when the record was created (*pMade*), date and where available time of birth, and likewise for death. We allow null values in our database, so a living patient is identified by having a null value for *pDied*.

The *cold* field is of great importance. It is present in each and every table which is exported to the PDA, and allows us to select which records we will export. We only export records to the PDA if their cold value is NULL. Special Perl routines set the cold flag in records which are, as the name suggests, 'cold' and not desired on the PDA.

Other fields deserve description — *pTokens* will be an encrypted field governing user access to the PDA[18]. We might, but here don't constrain pStatus to depend on a separate table (STATUS) which describes the role of the individual represented on the PDA, with bit flags representing a user (doctor or nurse), and/or patient role.[19] *pFlags* is at present unused, but the lowest order bit could be set to 1 in circumstances where the birth *time* is known, and likewise for the next bit up, which might be used to signal a known time of death.[20] pUserName was added subsequently to facilitate desktop user management.

The following description of the STATUS table isn't actually used within our database code: we simply rely on knowledge of the values, but don't need the actual table!

```
CREATE TABLE STATUS (
  status integer,
      constraint badStatus primary key (status),
  pstStatus varchar(32)
              );
```

The STATUS table would be a frilly accessory, with status containing the required bit-flags to identify a patient. The flags are as follows:

Flag mask (hex)	*Meaning*
01	patient
02	pain team member
04	nurse
08	doctor
10	junior
20	senior

Table 1: Bit flags for STATUS table

[NOTE: SEE ACTUAL USAGE. DO NOT RELY ON THE ABOVE TABLE AT PRESENT]

[18]Perhaps best removed to another table?

[19]It would be best to design software around the concept that if both user and patient flags are set, that individual's access to the data should be regarded as a special case, governed by a discrete set of rules appropriate to relevant local policies.

[20]This arrangement allows us to by default set these times to zero in the corresponding timestamp fields, without the anomaly that where the time actually *is* zero, we have no way of distinguishing between this and a default value.

To fulfil key dependencies, STATUS should cover all possible combinations used. Here's code we might use to populate STATUS:

```
INSERT INTO STATUS (status, pstStatus)
          VALUES (1, 'Patient'),
                 (2, 'Anaesthetist'),
                 (4, 'Pain team nurse'),
                 (10, 'Consultant anaesthetist'),
                 (12, 'Paint team consultant'),
                 (14, 'Pain team consultant anaesthetist'),
                 (6, 'Pain team anaesthetist);
```

4.1.1 'Temporal denormalisation' and other problems

Further information needs to be attached to the **person**, but we must be cautious. Many databases are bedevilled by having a fixed field for, say, *surname*, which becomes a problem with aliases, marriages and deed poll name changes. In other words, with time, apparently invariant data changes. If the database cannot accommodate what I call 'temporal denormalisation' then either the old data have to be thrown away to make way for the new, or arbitrary hacks have to be performed to retain them.

Another problem is that of viewpoint — who is making a data assertion.[21] This is best illustrated with problems of gender representation. The gender perspective of say, a bed manager, may be at odds with that of the biologist and the patient. Our approach is to store such information as an *observation*! A person's particulars are stored as reports attributable to a person (source), taken at a particular time. Conflicts can be resolved by considering the nature of the datum and who reported it. For example, we can envisage gender data being stored with the ability to attribute the source of information to the most important person ('I declare that I am female'), as well as phenotypic gender, karyotypic gender, legal gender, and so forth.

[21] All our 'statements of fact' are theory laden.

4.2 process

```
CREATE TABLE PROCESS (
  cold integer,
  process integer,
      constraint badProcessKey primary key (process),
  Person integer,
      constraint badPatientref foreign key (Person)
          references PERSON ON UPDATE CASCADE,
  ProcType integer,
      constraint badProcessType foreign key (ProcType)
          references PROCTYPE,
  rStart timestamp,
  rEnd timestamp,
  rCreated timestamp,
  rPlanner integer,
      constraint badPlannerref foreign key (rPlanner)
          references PERSON ON UPDATE CASCADE
                ); 
```

The complex course of a patient through hospital is mapped using a fairly simple concept — that of the process. Note that a process (continuous or continual) may extend over several days (or even weeks) or be extremely brief.

The process clearly refers to a particular patient, and has a start and end time, but what of the rest? It's good to know who is the architect or creator of the process (*rPlanner*), as well as when the process was conceptualised — *rCreated*. Processes may be of several types, a concept detailed in the PROCTYPE table described in the next section.

The 'ON UPDATE CASCADE' modification to constraints in the above anticipates our later use of temporary keys in the PERSON table, which we then update (See AnalgesiaDB2 document). This comment also applies to the tables EPOCH, ACTOR2 and (eugh) BADOBS.

Process code	*Meaning*
1	Data observation
3	Hospital admission process
5	Post-discharge
50	Enteral drug administration
110	Epidural catheter access
210	Epidural drug administration
310	Epidural drug PCEA
100	Spinal access
200	Spinal drug administration
190	Intravenous access
290	IV drug infusion
291	IV drug boluses
390	IV drug PCA
280	Transdermal drug administration
276	Subcutaneous drug administration
270	Per-rectal drug administration
266	Nasal drug administration
500	Surgery

Table 2: Coding of process types

4.2.1 Types of Process

The process table refers to a ‘type of process’, which in the SQL implementation must be defined before the actual process table. Here’s the relevant, trivial table:

```
CREATE TABLE PROCTYPE (
  cold integer,
  proctype integer,
      constraint badPROCTYPEKey primary key (proctype),
  rptNature varchar(32)  );
```

Although this table might be considered not to need a cold field, we nevertheless include one to facilitate later export of all tables to the PDA. A brief preliminary list of types of process is shown in Table 2.[22]

Regional codes have dispersed among them interscalene, infraclavicular, axillary, interpleural, femoral, sciatic and incisional infusions. A tentative coding scheme for these is:

[22]For supplementary processes, see section 8.

Process code	*Meaning*
120	interscalene catheter access
220	interscalene infusion
125	infraclavicular catheter access
225	infraclavicular infusion
130	axillary catheter access
230	axillary infusion
135	interpleural catheter access
235	interpleural infusion
140	femoral catheter access
240	femoral infusion
145	sciatic catheter access
245	sciatic infusion
150	incisional catheter access
250	incisional infusion

4.2.2 People participating in a process

There will be role players in any process. We need the ability to describe their function. Such characterisation is tricky — on the one hand, we have the concept of several players intermittently associating themselves with the process; on the other we wish to keep the data model clean. (Our current database includes the AROLE table but we don't use it a present)!

One approach would be to create a separate process for each player, and then somehow associate such processes with a 'dominant' process. This approach is flexible but somewhat wasteful. Another is to merely associate players ('actors') with a process without regard to timing of association. There are problems with this approach as actors come and go in a particular process. A low-waste, flexible approach is to *observe* the presence of a particular actor — an ACTOR2 table qualifies the observation, linking it to one or more actors.[23]

[23] We also must describe the AROLE of the actor.

4.3 epoch

As discussed above, encoding of observations of a process is tricky. Here's the relevant table:

```
CREATE TABLE EPOCH (
  cold integer,
  epoch integer,
      constraint badObsKey primary key (epoch),
  oMade timestamp,
  oLength integer,
  Person integer,
      constraint badObserver foreign key (Person)
          references PERSON ON UPDATE CASCADE,
  Process integer,
      constraint badObsProcess foreign key (Process)
          references PROCESS ON UPDATE CASCADE
                );
```

We have renamed the EPOCH table (it was previously OBS, short for observation) because we believe it's important to have a duration attached to an epoch. There are at least two possible ways to timestamp the end of the epoch, the first being an actual TIMESTAMP, the second by inserting a duration for the epoch. We have chosen the latter for three reasons, firstly the duration is more succinct in terms of PDA storage (4 bytes as opposed to 14 bytes), secondly the granularity of the duration can be made finer (1 billion milliseconds gives us 11 days!), and thirdly it's easier to pull out epoch duration (don't need to subtract two timestamps for every interrogation of a datum). The oLength integer specifies the epoch duration in milliseconds, although in our current usage, we will merely enter seconds multiplied by 1000.

We then in turn create several subsidiary tables which describe parameters attached to the epoch. Here's one such table — ACTOR2:

```
CREATE TABLE ACTOR2 (
  cold integer,
  actor2 integer,
    constraint badActorkey primary key (actor2),
  Epoch integer,
    constraint badactObs foreign key (Epoch)
       references EPOCH,
  Arole integer,
    constraint badRole foreign key (Arole)
       references AROLE,
  Person integer,
    constraint badActor foreign key (Person)
       references PERSON ON UPDATE CASCADE
                 );
```

You can see that several actors might be associated with a single epoch, which is economical and not unreasonable — several people may be present on a ward round or at the start of an operation, for example. See how ACTOR2 itself has a subsidiary table, AROLE, which is nothing more than a little ‘frill’:

```
CREATE TABLE AROLE (
  cold integer,
  arole integer,
    constraint badROLEkey primary key (arole),
  acrRoleName varchar (32),
    constraint badactdescription
      check (acrRoleName is not null)
                     );
```

```
INSERT INTO AROLE (arole, acrRoleName)
  VALUES (1, 'Pain team consultant');
```

The actual use of ACTOR2 in our current database will be minimal, as we will only use it to document the presence of a consultant on the pain ward round.[24] The potential is however there. Other tables which refer to EPOCH are described in a later section (Section 6).

4.4 drug

Drug therapy can be complex and confusing — we try to cut through this by representing different basic modalities as processes. For example, epidural administration of drugs will be a process distinct from intravenous administration. Each mode (epidural, spinal, regional nerve blocks, intravenous, subcutaneous, transdermal, and rectal) should be represented by a different process type. PCA (or PCEA) can logically be associated with a variety of basic modes, notably epidural and IV administration. We will thus have two different process codes for epidural infusion without PCEA, and epidural infusion *with* PCEA. Likewise for intravenous infusion versus intravenous PCA.[25]

[24]An update: even this has been removed in the current version.
[25]With the option for similar extension of certain other modalities.

Several drugs might be associated with one process. In addition, drug therapy associated with a particular modality may change (for example, one might change from infusion of epidural bupivacaine 0.25% to 0.125% with fentanyl). It is tricky to model such changes.

Let's now use an epidural to describe our approach, although the pattern is generic and can be used for IV infusions, or whatever.

We will go through the following steps:

1. Create an 'epidural process' which describes epidural *access*, that is the presence of an epidural catheter;

2. Have separate epidural *infusion* processes; every time a change is made to the *type* of infusion, the previous infusion ceases and a new one is created. Rather empirically, rate changes are recorded as observations;

3. Associate one or more drugs directly with the process describing epidural infusion (not the 'epidural process' itself). This association is subject to the above condition — a drug change forces termination of that infusion, and creation of a new infusion process!

4. Is it reasonable to have a 'null infusion' to explicitly describe lack of infusion without removal of the epidural catheter itself?[26]

5. We could conceivably create a supervisory table associating the epidural process with subsidiary infusions; however such a table would be more for the convenience of analysis than PDA representation of the epidural.

There are several other ways of implementing the above, including recording drugs and their changes as observations (which I believe is too passive — observations of such alterations should be used to *drive* process changes, and not merely be recorded); or creating a hierarchy of processes and sub processes, either through a self-referential field in the PROCESS table, or a sub-process table. The latter approach becomes complex and difficult to analyse, and necessitates clumsy correlated sub-queries. I believe that in context, my approach is reasonable.

Here are two relevant tables, the DRUG table, which describes the actual drug (using the trade name where relevant, although a 'generic' option is encouraged), and the RX table which permits association of one or more drugs with a process. Note that there is no *database* reason why a drug might not be associated with other arbitrary processes that make no semantic sense; we constrain such behaviour in our scripting rather than in our database design.

[26]Perhaps a better option is to record such an absence as a NONEVENT entry!

```
CREATE TABLE DRUG   (
  cold integer,
  drug integer,
      constraint badDrugKey primary key (drug),
  dTrade varchar(32),
  DrugForm integer,
      constraint badDrugformulation foreign key
        (DrugForm) references DRUGFORM
               );

CREATE TABLE RX (
  cold integer,
  rx integer,
      constraint badRXID primary key (rx),
  Drug integer,
      constraint badGivenDrug foreign key (Drug)
          references DRUG,
  Process integer,
      constraint badRxProc foreign key (Process)
          references PROCESS,
  gConcentration integer,
  gRate integer,
  gDose integer,
  gInterval integer
          );
```

Concentration, rate, dose and interval variables permit fairly flexible dosing, either intermittent or continuous.[27] Null values can be used where appropriate. Concentration will be in micrograms/ml, rate in microlitres/hr, dose in micrograms, and interval in seconds.

Note the apparent denormalisation implied by the inclusion of a drug formulation (under DRUG) when we associate a drug with a particular process. In fact, this permits us to *document* errors where the *incorrect* formulation is given by a particular route!

Here's the DRUGFORM table which describes the formulation:

[27]Examine me carefully; might we have a separate *regimen* table??

```
CREATE TABLE DRUGFORM (
  cold integer,
  drugform integer,
      constraint badFormKey primary key (drugform),
  dformText varchar(32)
                );
```

Formerly we used the PHARM table but for now this has been disabled:

```
CREATE TABLE PHARM (
  cold integer,
  pharm integer,
    constraint badPharmKey primary key (pharm),
  hName varchar (32)
                 );
```

A new table related to the DRUG table is DRUGUSAGE. It permits a lot of flexibility, because it describes how a particular DRUG is used. Clearly an epidural formulation will be used epidurally, but what of, say, morphine 1mg/ml? This might be used for intravenous boluses, or for PCA. Ideally, we would also wish to describe whether certain drugs are used as anti-nauseants without using an artificial classification based on the values of primary keys. We will use the DRUGUSAGE table to code for such eventualities. (See ListDrugs in *AnalgesiaDB2.tex*). We might formally specify the USAGE table, but for now this will be implicit, with the following codes:

drUsage code	*Meaning*
1	PCA
2	IV bolus
3	Special infusion
4	anti-nauseant
5	oral analgesic
6	other analgesic

There is clearly room for adding many other options, but we will be parsimonious at present. Here's the actual DRUGUSAGE table:

```
CREATE TABLE DRUGUSAGE (
  cold integer,
  drugusage integer,
      constraint badDrugUsage primary key (drugusage),
  Drug integer,
      constraint badUDrug foreign key (Drug)
        references DRUG,
  drUsage integer
                ) ;
```

We might additionally constrain drUsage to be IS NOT NULL.
[WE MUST STILL ADD A SEED IN THE UIDS GENERATOR TABLE.]

4.5 BED

Tracking movement of patients from bed to bed and ward to ward can be tricky. We will not be content with simply recording a ward and room against a patient. Although we won't activate full bed-by-bed recording in our initial version of the PDA database, the potential is there to do so! Here's a tentative 'bed-epoch' table:

```
CREATE TABLE BEDOBS2 (
  bedobs2 integer,
      constraint badBedobskey primary key (bedobs2),
  Epoch integer,
      constraint badBedObs foreign key (Epoch)
          references EPOCH,
  Bed integer,
      constraint badBedObs2 foreign key (Bed)
          references BED
               ) ;
```

This is not the final table we use. First let's examine the actual, inter-related BED, WARD and ROOM tables:

```
CREATE TABLE WARD (
  cold integer,
  ward integer,
      constraint badWardKey primary key (ward),
  swrdText varchar(16)
            ) ;

CREATE TABLE ROOM (
  cold integer,
  room integer,
      constraint badRoomKey primary key (room),
  Ward integer,
      constraint badRoomWard foreign key (Ward)
      references WARD,
  srmText varchar(16));

CREATE TABLE BED (
  cold integer,
  bed integer,
      constraint badBedspaceKey primary key (bed),
  Room integer,
      constraint BadBedRoom foreign key (Room)
          references ROOM,
  sName varchar(8)    );
```

Despite the above, it's still not trivial to get all the patients for a particular ward! In the case where we fail to resolve associations to the level of detail of bedspaces, it's useful to create a generic bedspace for a particular ward or particular room, thus allowing association in the absence of detailed characterisation!

4.5.1 Obtaining ward occupants

It's rather tricky trying to find who is in a particular ward. First we must find the most recent bed epoch for each person:

```
 SELECT MAX(EPOCH.epoch) FROM BEDOBS2, EPOCH,
PROCESS
 WHERE BEDOBS2.Epoch = EPOCH.epoch AND
   EPOCH.Process = PROCESS.process
 GROUP BY PROCESS.Person;
```

The assumption in the above is that epoch keys *always* increase with time![28] We then need to take the resulting observations, find which ones correspond to the correct ward, and pull out the relevant patients!

Things would be massively simplified if we performed the following wicked denormalisation on the BEDOBS2 (aka 'BADOBS') table (we do):

```
CREATE TABLE BADOBS (
  cold integer,
  badobs integer,
```

[28]This in turn implies that temporary generated keys on the PDA *must* always be greater than permanent keys originating on the desktop!

```
        constraint badBadobskey primary key (badobs),
    Epoch integer,
        constraint badBadObs foreign key (Epoch)
            references EPOCH,
    Bed integer,
        constraint badBadObs2 foreign key (Bed)
            references BED,
    boInactive integer,
    boFlag integer,
    Person integer,
        constraint aaghBadObs foreign key (Person)
            references PERSON ON UPDATE CASCADE
                 );
```

With this structure and the ‘boInactive’ field, which we set for a particular patient whenever we change BED, it becomes trivial to find the current patients in a particular ward. (See GetPatients4Ward in *AnalgesiaDB2.tex*).

We introduced the boFlag field to allow us to identify (and flag) certain individuals. We examine each individual for a particular property, and set this flag to 1 if the person meets our requirements. Of particular interest are patients for evening review (PM review process is active), those who have been identified as having problems, and those who haven’t been seen today (ie. with no observations having the current date). Identifying each such individual poses its own problems.

Note that on *discharge* we reset the value of boFlag to NULL. This trick allows us to easily discern discharged patients, without obliterating their BADOBS table entry.

5 Small general-purpose tables

As noted above in section 4.1.1, it's a good idea not to commit to data structures which won't accommodate alterations such as surname changes. At the cost of more complexity, we consequently farm these data out as observations.

We have also discussed two reasons why it may be unwise to store values for all such observations within a single epoch table — there are time penalties, as well as problems of encoding associated with such an approach.

We thus construct several small, simple tables which reference the EPOCH table, and contain general-purpose information. Here are two:

5.1 Data concerning people

```
CREATE TABLE PERSDATA (
  cold integer,
  persdata integer,
    constraint badPersDataKey primary key (persdata),
  Epoch integer,
    constraint badPersDataObs foreign key (Epoch)
      references EPOCH,
  pdoSurname varchar(32),
  pdoForename varchar(32),
  pdoHospNo varchar(16),
  pdoPerson integer,
    constraint badPdPerson foreign key (pdoPerson)
      references PERSON ON UPDATE CASCADE,
  pdoGender integer   );
```

The most noteworthy field is pdoPerson, which represents a *wicked* denormalisation. We do this to speed searches on the PERSDATA table, which otherwise need a complex sequence of joins.

The *pdoGender* field will at present encode just four options (null for unknown, 0 for unclear, 1 for female, 2 for male) with the option to reference an external table of relevant complexity.[29]

Note that if any one of these fields changes, for example, if someone marries and changes their surname, then a *new epoch* will be created together with a new reference to this epoch in PERSDATA. If this action is performed, then we will *only* place a new value in the requisite field, and will leave all other data fields

[29]Of negligible utility in most contexts, and almost certainly in the pain database context, but of relevance with e.g. gender alteration, intersex, pseudohermaphroditism etc.

NULL, preventing duplication of other data. The correct method of interrogating such data is thus to look for all relevant observations on the 'Data epoch' process for that patient, and then order them by time to get more recent data, ignoring null values. Such data storage depends on sane behaviour on the part of updating programs, and is not constrained by the database structure.[30]

The downside of such an approach is the complex query required to pull out the relevant datum. Let's say we want the gender. Given the relevant process code (say 123), one option for finding the most recent relevant epoch is:

```
SELECT MAX(PERSDATA.persdata) FROM PERSDATA, EPOCH
 WHERE
   pdoGender IS NOT NULL
   AND PERSDATA.Epoch = EPOCH.oKey
   AND EPOCH.Process = 123;
```

This isn't as bad as it looks, especially on the PDA where the PERSDATA table will be small. Time consumed is $O(n.\log_2(o))$, where n and o are the number of items in the PERSDATA and EPOCH tables respectively. There are ways we might speed things marginally. For example, we might only export the 'most recent' composite data from desktop to PDA, provided we enforce the requirement that there is no reverse transferral of such data back to the desktop (This is reasonable, as any alteration will involve creation of a new row in PERSDATA, as noted above).

To retrieve gender directly we can then condense our query to:

```
SELECT PERSDATA.pdoGender FROM PERSDATA, EPOCH
 WHERE
   PERSDATA.Epoch = EPOCH.oKey
   AND EPOCH.Process = 123
```

This approach may well not be worth the effort. We could of course 'flatten' things and refer PERSDATA directly to PROCESS; this approach would necessitate either abandoning fidelity, or having two of the EPOCH fields represented in PERSDATA — oMade and Person. Nasty. We create the pdoPerson field in preference.

[30]An alternative would be to have even more fiddly data tables and get rid of the nulls. Chris Date may have some good points, but I can find ample justification for use of the odd null!

5.2 ASA rating

The American Society of Anesthesiologists (ASA) physical status classification system was created in 1941 as a simple rating scale, and has been revised, with the addition of an 'E' status for emergency cases. The scale is shown in Table 5.[31]

Score	*Description*
1	A completely healthy patient
2	Patient with mild systemic disease
3	Severe systemic disease, not incapacitating
4	Incapacitating disease which is a constant threat to life
5	Moribund patient not expected to live 24 hours with or without surgery

Table 5: ASA rating system.

The ASA rating of an individual may vary with time (and observer!) As we are now accustomed, we will code this rating through a table referencing the epoch table. Because of the propensity of people in the medical field to use overall scoring systems, it seems reasonable to generalise our approach, allowing multiple scores to be represented without adding a plethora of new tables, thus:

```
CREATE TABLE MEDSCORE (
  cold integer,
  medscore integer,
    constraint badMedScoreKey primary key (medscore),
  Epoch integer,
    constraint badMedObsRef foreign key (Epoch)
      references EPOCH,
  msoNature integer,
  msoValue integer    );
```

Due to the current trivial character of msoNature, we decline to provide a table which it references — we will be content to allow just two values in msoNature (1 = ASA 'E' score, 2 = ASA 1–5) with corresponding values in msoValue (The only valid entry in msoValue is 1 if E was documented, as 'non-E' is, as it were, the default;[32] we simply record 1–5 in msoValue in the case of msoNature being 2). Extension is possible with other scoring systems.

[31]Some add a 6 for the brain-dead organ donor; some a G modifier for the pregnant state.
[32]Arguably we might have a separate 'non-E' box in our interface!

5.3 Operation data

We have already described surgery in terms of a PROCESS (Section 4.2, and table 2). Associated tables will be used to describe the site and type of surgery. Because several sites and types may be involved during a single operation, it seems reasonable to fit these tables into our epoch paradigm. We can then also invoke any programming machinery we use to examine the EPOCH table, when we need to access surgical information.[33] Here is the SURGTYPEOB table (the SURGSITEOB table has been removed as discussed in *AnalgesiaDB2.tex*):

```
CREATE TABLE SURGTYPEOB (
  cold integer,
  surgtypeob integer,
      constraint badSurgTypeObsKey primary key (surgtypeob),
  Epoch integer,
      constraint badSurgTypeObsRef foreign key (Epoch)
         references EPOCH,
  SurgType integer,
      constraint badSurgTypeRef foreign key (SurgType)
         references SURGTYPE
                  );


CREATE TABLE SURGSITEOB (
  cold integer,
  surgsiteob integer,
      constraint badSurgSiteObsKey primary key (surgsiteob),
  Epoch integer,
      constraint badSurgSiteObsRef foreign key (Epoch)
         references EPOCH,
  SurgSite integer,
      constraint badSurgSiteRef foreign key (SurgSite)
         references SURGSITE
                  );
```

The SURGTYPE and SURGSITE tables are fairly trivial lists, containing values from the following rather tentative tables. We've removed the SURGSITE table. First, type of surgery, as shown in Table 6.

[33] Alternative approaches such as having multiple concurrent sub-processes, etc. are less appealing.

```
CREATE TABLE SURGTYPE (
  cold integer,
  surgtype integer,
    constraint badSurgtypeKey primary key (surgtype),
  ctText varchar(32)
                      );
```

Code	*Meaning*
1	general
49	endoscopy
99	plastics
149	orthopaedic
199	neurosurgery
249	ophthalmic
299	dental
399	ORL
449	cardiac
499	thoracic
599	vascular
649	hepatobiliary
699	colorectal
799	urology/renal
899	gynaecology
999	obstetrics

Table 6: Coding of types of surgery

Similar to type coding, is the data table we formerly used to encode sites of surgical intervention.

```
CREATE TABLE SURGSITE (
  cold integer,
  surgsite integer,
    constraint badSurgsiteKey primary key (surgsite),
  csText varchar(32)
                      );
```

Unfortunately, international codings for body site (even where we ‘simply’ look at injury coding) are best described as a stuffup. ICD-9 doesn’t interoperate with ICD-10; both are baroque in their complexity. Tools like the Barell Matrix (designed to mitigate ICD-9) don't work with ICD-10. ICD-10 S codes (S00–99) cover injuries. There are other tools — the IAIABC body part coding, something called DN36. The complex and detailed ICECI classification of injury doesn't even refer to the site of the injury on the person, as it's meant to be used in concert with ICD! We formerly kept things very simple, as shown in Table 7, but have now at least temporarily abandoned site coding as unnecessary, insufficiently explicit, and cumbersome.

Code	*Meaning*
1	upper abdomen
2	lower abdomen
3	abdomen
119	lumbar region
149	pelvis
199	perineum
299	thorax
399	upper limb
499	lower limb
599	eye
699	head
799	neck

Table 7: Coding of surgical sites

The above table has the capacity for refinement of codes, to include more detail.[34] We might use: head (610–660: ear, nose, teeth, mouth, facial, craniotomy), neck (710–740: vertebral column & cord, larynx, trachea, thyroid), upper limb (310–380: shoulder, upper arm, elbow, forearm, wrist, hand, fingers, and thumb), thorax (210–240: vertebrae & cord, lateral thoracotomy, sternotomy, other chest wall), lumbar detail (100–110: vertebrae & cord, flank incisions), and lower limb refinements (410–480: hip, thigh, knee, leg, ankle, foot, great toe and other toes).

[34] When we present sites in menus, it's important that we both keep these simple, and, within a single list, preferably present more detailed site information *before* more general site information, to enhance appropriate coding. For example, eye should be encountered before head if they're in the same list!

5.4 ‘Not-observations’

It is sometimes necessary to record the *absence* of something, for example, the absence of a regional infusion, IV PCA, oral therapy, or ‘other modalities of therapy’. Rather than creating phantom processes to reflect such ‘non-events’, we have a separate NONEVENT table. Here it is:

```
CREATE TABLE NONEVENT (
  coId integer,
  nonevent integer,
    constraint badNoneventKey primary key (nonevent),
  Epoch integer,
    constraint badNoEvObs foreign key (Epoch)
      references EPOCH,
  noCode integer,
  noValue integer
                      );
```

Okay, we should have a table to describe the various codes, but at present we will simply use the codes 1, 2, 3 and 4 for a regional infusion, IV PCA, oral therapy and ‘other modalities’ respectively!

Initially we thought that we would only use the existence of a noCode value to indicate the *absence* of the associated therapy, but we then realised that it made more sense to represent both the user’s recording of absence and *presence* in the same table, hence the noValue field. If the noValue field contains a 1, then the event is *absent*, if it’s a zero, then the event is indeed present!

5.5 Additional EPOCH-related tables

A few other observations may be useful, notably the *weight*, recorded in the MEASURE table. Kilograms will be the entry value, but we will record an integer weight in grams in the table, for more flexibility!

```
CREATE TABLE MEASURE (
  coId integer,
  measure integer,
    constraint badMsrKey primary key (measure),
  Epoch integer,
    constraint badMsrObsRef foreign key (Epoch)
      references EPOCH,
  meWt integer );
```

We might at some stage also include meHt, the height in millimetres.

6 Analgesia-related tables

Here we describe all tables referencing the epoch table, provided they are relevant to the description of pain or administration of analgesia.

6.1 Pain scores

It's reasonable to group pain scores — they will usually be recorded together as part of one epoch. It also makes sense to have null values in unused fields. The table is simple:

```
CREATE TABLE PAINSCORE (
  cold integer,
  painscore integer,
    constraint badPainScoreKey primary key (painscore),
  Epoch integer,
    constraint badPsObsRef foreign key (Epoch)
      references EPOCH,
  psoRest integer,
  psoMovement integer,
  psoCough integer     );
```

Owing to the simple nature of the three pain score fields (values from 0–10 for pain at rest/on movement, and preferably just a 0/1 for inadequate cough/adequate cough), we do not at this stage link these to tables, but if more arcane values are required, this might be done.

6.2 Drug therapy

The simplest form of documentation of drug observations is merely to record whether the drug is being given. This approach might be taken for oral agents, but even here we would encourage recording of a total amount given between 08:00 and 07:59 inclusive, every day.

This observation of daily total dose is of particular importance with drug therapy using morphine and several other opiates. Ideally that period of time should be 24 hours, but as visits to the patient cannot necessarily be scheduled with clockwork accuracy, we must draw a line in the sand, hence the above time specification. Here's the table:

```
CREATE TABLE RXOBS (
  cold integer,
  rxobs integer,
    constraint badRxObsKey primary key (rxobs),
  Epoch integer,
    constraint badRxObsRef foreign key (Epoch)
      references EPOCH,
  rxoTotal integer,
  rxoDoses integer
                      );
```

The value in rxoTotal is in micrograms; if this value is left as null, then we record the *presence* of stated therapy, but no dose. The type of therapy can always be deduced via EPOCH (and thence the attached process, and its associated values in the RX table). [35]

We include the separate rxoDoses for several reasons, the most important being that with some therapy, we give intermittent doses (e.g. top-ups of the epidural) and we need to record this.[36]

6.3 Continuous infusions

A basal infusion via a PCA pump and a continuous infusion are effectively the same thing; likewise for an epidural infusion.[37] We record such infusions using the INFUSIONOBS table.

```
CREATE TABLE INFUSIONOBS (
  cold integer,
  infusionobs integer,
    constraint badInfKey primary key (infusionobs),
  Epoch integer,
    constraint badInfObsRef foreign key (Epoch)
      references EPOCH,
  inoRate integer,
```

[35]It might still be desirable to introduce an apparent denormalisation — having a reference within RXOBS to DRUG, or to do this using a separate table, so that *errors* in drug administration can be well documented! Investigate this!

[36]Another motivation for this field is that it gives us some idea as to whether the 'total dose' was given as one big slug or several smaller doses. Ideally each dose should be recorded, together with time and amount, but this isn't feasible at present. [REVIEW ME]

[37]Provided we aren't infusing bupivacaine intravenously :-(we're not here talking about correctness, just mechanism.

```
  inoConc integer
                    );
```

[DO WE NEED THE INOCONC AS THIS SHOULD BE PART OF THE RX ????? [38]]

The infusion referred to can be derived from the EPOCH table, and might be one of a variety of types.

Details are in order. The rate is in microlitres/hr; drug concentration in micrograms/ml. Stat doses and top-ups should be recorded separately.

6.4 PCA observations

PCA may be provided with both intravenous and epidural infusions, the latter as PCEA. We have coded PC(E)A as separate types of **process** for each. In order to record PCA observations, we use the following table:

```
CREATE TABLE PCA (
  cold integer,
  pca integer,
    constraint badPcaObsKey primary key (pca),
  Epoch integer,
    constraint badPcaObsRef foreign key (Epoch)
      references EPOCH,
  pcoTries integer,
  pcoGood integer
                    );
```

For *pcoTries* and *pcoGood* to be meaningful, the number must be since a prior event, and the logical event is when the last reading was taken (and usually, when the pump was last cleared). Cumulative totals should not be recorded, only changes since the last recording. We are not here recording total dose; if desired, this total can be recorded using the RXOBS table (Section 6.2).

Basal infusion associated with PCA is recorded separately, as described above in section 6.3.

Here's the separate table for observation of PCA settings:

[38] Do we need a separate type of observation — that of the type of drug? Then can meticulously document errors. to explore.

```
CREATE TABLE PCASETTINGS (
  cold integer,
  pcasettings integer,
    constraint badPcaSetKey primary key (pcasettings),
  Epoch integer,
    constraint badPcaSetObsRef foreign key (Epoch)
      references EPOCH,
  pseDose integer,
  pseLockout integer,
  pseLimitInterval integer,
  pseDoseLimit integer
                     );
```

For maximum flexibility, we will represent pseDose in micrograms, and pseLockout in *seconds*, not minutes. What about e.g. 4 hourly dosing limits? We also need a limit interval (again in seconds), as well as a limiting dose, in micrograms.[39] If we're feeling conservative as regards wasted space, we might consider moving *pseLimitInterval* and *pseDoseLimit* to a separate table![40]

6.5 Epidural and other regional observation

Multiple regional data items will generally be observed at nearly the same time. This multiplicity should be reflected in 'regional data tables' attached to the regional process. There is an added complexity, in that we have at least two processes — one reflecting the presence of a regional catheter, and another describing the current infusion.

Continuous epidural infusion can be adequately described using the INFUSIONOBS table (See section 6.3), and PCEA using the PCA table (Section 6.4). Documentation of observations related to the catheter and its function are distinct, and contained in the following table:

```
CREATE TABLE RGNOBS (
  cold integer,
  rgnobs integer,
    constraint badRgnObsKey primary key (rgnobs),
  Epoch integer,
    constraint badRgnObsRef foreign key (Epoch)
```

[39] Other more complex regimens are possible, but we won't here allow for such finesse!

[40] Think about *initial* settings and how we encode these. What about cross-checking? What about cessation of infusions (Stop button?)

```
        references EPOCH,
    rgoPressure integer,

    rgoSite integer,
    rgoMotor integer,

    rgoLevel integer,
    rgoMobile integer,

    rgoHeadache integer,
    rgoSitePain integer,
    rgoBackache integer,
    rgoInconti integer
                        );
```

We have a fairly large number of columns within RGNOBS. This allows for coding of a variety of observations, including epidural follow-up.[41]

A value of 0 is at present used to indicate any motor dysfunction, 1 normal motor function.[42]

The way we have designed our tables mandates that if the drug (or concentration of drug) being infused changes, we terminate that particular process and replace it with a new one. Our response to menu entries documenting such changes will thus not be to 'document the observation' but to make the necessary changes to data in the PROCESS table! [43]

Ultimately we might create further tables to explain rgoLevel, rgoPressure, rgoSite, and rgoMotor, although initially we will simply record null for 'not observed', 0 for 'no problem' and 1 for 'problem present'. We've already added a further field — rgoMobile to indicate whether the patient is mobilising or not.[44]

[41]We might extend some questions reserved for follow-up to make them routine for all visits, but this may impair their effectiveness as one-off follow-up questions.

[42]CHECK ME: This might not be desirable!

[43]But do we need to 'affirm' that the current infusion is continuing — surely yes? Hmm. This is implicit in the recording of other observations; in fact, we could leave these blank and simply create an entry to document ongoing infusion!

[44]We might consider moving this elsewhere?

6.6 Other modalities

Transdermal, rectal, and other administration of drugs is reflected in the relevant processes. The existence of administration and doses can adequately be represented in the tables already defined above.

7 Reasons for stopping

We have recently added another table. It is highly desirable to be able to determine why PCA and epidural or other regional infusions have been stopped. We therefore create a table which associates a principal reason (in the estimation of the person doing the recording) with a particular process which has been stopped. The associative table is STOPPROC, but first we need a list of reasons for stopping, stored in the WHYSTOP table:

```
CREATE TABLE WHYSTOP (
  cold integer,
  whystop integer,
    constraint badWhyStop primary key (whystop),
  wText varchar(32)
                     );
```

Let’s populate this table:

```
INSERT INTO WHYSTOP (whystop, wText)
  VALUES ( 1, 'no longer needed' ),
         ( 2, 'ineffective' ),
         ( 3, 'nausea'),
         ( 4, 'hypotension'),
         ( 5, 'CNS effects'),
         ( 6, 'accident/error' ),
         ( 7, 'local infection' ),
         ( 8, 'local haematoma' ),
         ( 9, 'other local complications' ),
         ( 10, 'risk of infection' ),
         ( 11, 'bleeding risk' ),
         ( 12, 'patient request' ),
         ( 13, 'staff demand' ),
         ( 14, 'systemic complication' ),
         ( 15, 'not tolerated' ),
         ( 16, 'dictates of protocol' );
```

Finally, the STOPPROC table which links a process with a WHYSTOP value:

```
CREATE TABLE STOPPROC (
  cold integer,
  stopproc integer,
    constraint badStopp primary key (stopproc),
  Process integer,
    constraint badStopProc foreign key (Process)
      references PROCESS,
  Whystop integer,
    constraint badStopWhy foreign key (Whystop)
      references WHYSTOP
                    );
```

We use this table in preference to having a StopProc field within all PROCESS tables for reasons of parsimony in the PROCESS table.

At present it's still possible (See *AnalgesiaDB2.tex* to terminate a PCA or epidural process without demanding a reason, but such terminations are limited to alterations to infusions — when an infusion is altered, a new (similar) process is created and supplants the former one. This approach allows us to make 'minor' changes without demanding a reason.

8 Modelling of problems

There are several possible ways we might model problems such as renal failure or coagulopathy.[45] One unattractive option is to simply record them as observations, unappealing because it fails to highlight their ongoing nature, as well as relegating them to the status of unconfirmed 'diagnoses'. Better is to have full blown processes to represent such problems. Here is a supplemental list of processes that refer to problems:

Process code	*Meaning*
1000	substantial renal dysfunction
1010	heart failure
1020	respiratory failure
1030	liver failure
1040	anticoagulation
1050	coagulopathy
1060	chronic pain
1070	chronic opiate use
1080	allergy alert

[45]See *PDA data capture based on a form template* for the relevant list of items and data screen.

1090	CNS dysfunction
1100	PM observation
1110	sedation
1120	hypotension
1130	nausea
1140	bowel opening

Table 8: Process codes for problems

We've conveniently lumped problems such as nausea, hypotension and sedation, as well as the necessity for 'PM observation', together with the other 'problem' processes! Bowel opening is substantially relevant where opiates are being used!

Several objections can be raised to the above, somewhat simplistic coding. We provide no definition of any of the above, leaving it up to the individual clinician. Many other disorders are simply *not* coded at present. There is potential overlap between such items as 'anticoagulation' and 'coagulopathy'.[46] A bland generalisation like 'allergy alert' is not necessarily a liability, as it can indeed be qualified by other tables, for example, observations as to the specific nature of the allergy. There is plenty of room to fill in other 'nonstandard' problems, but we would advise caution and consensus in this regard.[47] It might be reasonable to reserve key values of, say, over 10000, for custom processes limited to a particular (local) coding system.

9 Comments

It seems a good idea to have the facility to attach arbitrary comments to observations. The danger with this approach is that vital information can be relegated to obscure comments, and be effectively lost, especially if typos are included in the comment, or details are otherwise obfuscated!

We will take the course of associating arbitrary text comments with observations in the EPOCH table, thus:

```
CREATE TABLE COMMENT (
  cold integer,
```

[46]This might be resolved by mandating that the latter is only selected when the coagulation disturbance is/would be present, independent of any administered agent.

[47]Stuffing the PROCESS table with a host of marginally relevant processes has the potential to bring the database to its knees.

```
  comment integer,
    constraint badCmtId primary key (comment),
  Epoch integer,
    constraint badCmtObs foreign key (Epoch)
      references EPOCH,
  cText varchar(255));
```

We allow a fairly substantial number of characters for comments, without overdoing it. Several unordered comments can be attached to an epoch, and via this association we can derive timestamp, observer and process. Because the comments are generic comments on the epoch, we discourage very specific 'data-containing' comments associated with other tables depending on EPOCH. We can nevertheless separate out comments by PROCESS, and so have 'epidural comments', general observation comments, and so forth!

10 Observing problems

How do we model the observation of problems such as hypotension, nausea, and sedation? How can we indicate in a generic fashion that we 'consider the patient to have major problems'?

One rather arbitrary approach is only to create an instance of the 'problem process' when the problem is present, but there are several disadvantages of this approach. Most important is that the observation that the problem 'doesn't exist at present' must be stored elsewhere. In addition, we create multiple disjointed problems if the problem waxes and wanes, or comes and goes. It's silly to terminate the process every time the blood pressure rises, and then re-establish it with the next drop in pressure. So we create distinct processes: blood pressure, nausea (or absence thereof) and 'level of sedation' (which can have the value 'none'). We then make observations as to the state of these processes.[48]

The most basic table we might attach to such processes is simply:

```
CREATE TABLE ISPROBLEM (
  cold integer,
  isproblem integer,
    constraint badprId primary key (isproblem),
  Epoch integer,
    constraint badProbs foreign key (Epoch)
      references EPOCH,
  prIsOrNot integer);
```

The `prIsOrNot` field simply has a one or zero depending on whether the problem is or isn't present!

[48]Even consider having an 'is there a problem' process!

11 Documenting Errors

We introduce the PAINERROR table, which is attached to an Epoch, documenting the type of error detected, the date on which it occurred, and a mandatory comment. First the subsidiary ERRTYPE:

```
CREATE TABLE ERRTYPE (
  cold integer,
  errtype integer,
    constraint baderrtype primary key (errtype),
  etText varchar (32))
```

Now for the main error table.

```
CREATE TABLE PAINERROR (
  cold integer,
  painerror integer,
    constraint badpnerr primary key (painerror),
  Epoch integer,
    constraint badPEepoch foreign key (Epoch)
      references EPOCH,
  Errtype integer,
    constraint badPEtype foreign key (Errtype)
      references ERRTYPE,
  ErrStamp timestamp,
  peText varchar (128)
  );
```

Although we introduce the capacity for timestamping the error, at present we will only record the date, and leave the timestamp as '00:00:00'. Let's populate the ERRTYPE table here:

```
INSERT INTO ERRTYPE (errtype, etText)
  VALUES (10,'Prescription'),
         (20,'Administration'),
         (30,'Pump malfunction'),
         (40,'Disconnection'),
         (50,'Cessation'),
         (60,'Anticoagulation'),
         (70,'Other management'),
         (101,'Bad PDA data');
```

12 Printing

In the ‘production’ version of our database, it’s clearly desirable to print reports, both for individual patients, and summary statistics. We would like to maintain an audit trail of printing activity, so we will create a new table which records such activities. Each PROCESS will, when printed (i.e. when involved in a printing procedure) acquire an associated entry in the PRINTED table, together with a timestamp and the identity of the individual logged on and doing the printing. Here’s the table:

```
CREATE TABLE PRINTED (
    cold integer,
    printed integer,
      constraint badPrinting primary key (printed),
    Process integer,
      constraint badPrintProc foreign key (Process)
        references PROCESS,
    prTime timestamp,
    Person integer,
      constraint badPrintPerson foreign key (Person)
        references PERSON);
```

We will also create a related uPrinted column in the UIDS table below, but will not export the PRINTED table to the PDA, and therefore do not create xTABLE and xCOLUMN entries.[49]

[49]Note that because of the lack of these entries, we do not automatically export and restore this table if we backup and restore the database! [FIX ME]

13 Population of data tables

The following SQL populates the various tables described above.

13.1 Populating type of process

Here are fundamental codes and text labels for the various procedures and other processes alluded to above. The first two sections refer to analgesic processes and details of catheter access, the third section representing problems has been moved to the CSV import section of *PerlPgm.tex* but should not be tinkered with lightly as any changes to codes of 1000 and more must be associated with appropriate changes in menu items and associated scripting!

```
INSERT INTO PROCTYPE (proctype, rptNature)
  VALUES ( 1, 'Data observation' ),
    ( 3, 'Hospital admission process' ),
    ( 5, 'Post-discharge' ),
    ( 99, 'Check at 24h'),
    ( 100, 'Spinal [plain]' ),
    ( 101, 'Spinal [morphine]' ),
    ( 103, 'Spinal catheter insertion' ),
    ( 109, 'CSE' ),
    ( 110, 'Epidural catheter' ),
    ( 190, 'Intravenous access' ),
    ( 210, 'Epidural drug administration' ),
    ( 266, 'Nasal drug administration' ),
    ( 270, 'Per-rectal drug administration' ),
    ( 276, 'Subcutaneous drug administration' ),
    ( 280, 'Transdermal drug administration' ),
    ( 290, 'IV drug infusion' ),
    ( 291, 'IV drug boluses' ),
    ( 310, 'PCEA' ),
    ( 390, 'IV drug PCA' ),
    ( 500, 'Surgery' );

INSERT INTO PROCTYPE (proctype, rptNature)
  VALUES  ( 50, 'enteral drug administration' ),
    ( 115, 'spinal catheter' ),
    ( 215, 'spinal infusion' ),
    ( 315, 'spinal PCA' ),
    ( 120, 'interscalene catheter' ),
    ( 220, 'interscalene infusion' ),
    ( 320, 'interscalene PCA' ),
    ( 125, 'infraclavicular catheter' ),
    ( 225, 'infraclavicular infusion' ),
    ( 325, 'infraclavicular PCA' ),
    ( 130, 'axillary catheter' ),
    ( 230, 'axillary infusion' ),
    ( 330, 'axillary PCA' ),
    ( 135, 'interpleural catheter' ),
    ( 235, 'interpleural infusion' ),
    ( 335, 'interpleural PCA' ),
    ( 140, 'femoral catheter' ),
    ( 240, 'femoral infusion' ),
    ( 340, 'femoral PCA' ),
    ( 145, 'sciatic catheter' ),
    ( 245, 'sciatic infusion' ),
    ( 345, 'sciatic PCA' ),
    ( 150, 'incisional catheter' ),
    ( 350, 'incisional PCA' ),
    ( 250, 'incisional infusion' );
```

(Does anyone ever use the above odd PCA modalities)? Finally we have some amendments (as of 2007-11-08):

```
INSERT INTO PROCTYPE (proctype, rptNature)
 VALUES (203, 'Spinal infusion');

INSERT INTO PROCTYPE (proctype, rptNature)
 VALUES (303, 'Spinal PCA');

INSERT INTO PROCTYPE (proctype, rptNature)
 VALUES (199, 'Validation error');
```

The first two are to fix up *use* of a spinal catheter, the last is a temporary error-finding false process type.

13.2 Type of surgery

We have codes for 'generic' surgery in various fields such as plastics and orthpaedics. We have left gaps, anticipating that some might wish to refine coding, but note that coding by site is represented elsewhere — this approach is powerful because site and type codes can be combined, but the overhead of checking for appropriate combinations of site and type might be considerable and the alternative (allowing rubbish) will antagonise enough people to make us want to completely suppress 'surgical site'.

```
INSERT INTO SURGTYPE (surgtype, ctText)
  VALUES (  0, 'unspecified' ),
         (  1, 'general' );
```

Apart from 'general' we also allow for 'unspecified' to identify where database import routines fail to determine a code. The remaining codes have been moved to CSV import in the file *PerlPgm.tex*!

13.3 Surgical site (brief)

Again, there is room for refinement by site of the following coding, but don't rush off and do this as we've disabled usage of this table in *AnalgesiaDB2.tex*.

```
INSERT INTO SURGSITE (surgsite, csText)
  VALUES (  1, 'upper abd.' ),
    (  2, 'lower abd.' ),
    (  3, 'abdomen' ),
    (  119, 'lumbar' ),
    (  149, 'pelvis' ),
    (  199, 'perineum' );

INSERT INTO SURGSITE (surgsite, csText)
  VALUES  (  299, 'thorax' ),
    (  399, 'upper limb' ),
    (  499, 'lower limb' ),
    (  599, 'eye' ),
    (  699, 'head' ),
    (  799, 'neck' );


*
--- the end of this file specification!
```

13.4 Other data tables

It is probably wise to move population of the DRUGFORM, PHARM, WARD, ROOM, and BED tables to a separate SQL file, as these may vary dramatically from hospital to hospital, and even more so between regions and countries. (We might still profitably include 'generic' versions here)!

14 ‘Appendices’

14.1 META tables

These SQL tables record the structure of other tables we create! For a detailed examination of how these tables are populated, see the relevant subroutines and their documentation in the document PerlPgm.pdf.

xTABLE

This table contains basic information (just the name, at present) about tables created.

```
CREATE TABLE xTABLE (
  cold integer,
  xTaKey integer,
    constraint BadTableKey primary key (xTaKey),
  xTaName varchar (15),
    constraint NullTableName check (xTaName IS NOT NULL)
  );
```

xCOLUMN

xCOLUMN details the properties of each column in each table.

```
CREATE TABLE xCOLUMN (
  cold integer,
  xCoKey integer,
    constraint BadColumnKey primary key (xCoKey),
  xCoName varchar (30),
    constraint NullColumnName check (xCoName IS NOT NULL),
  xCoType varchar(1),
    constraint NullColumnType check (xCoType IS NOT NULL),
  xCoSize decimal(4,0),
  xCoScale decimal(2,0) default 0,
  xCoTable decimal (8,0),
    constraint ColInTable foreign key (xCoTable)
       references xTABLE,
    constraint ColHasNoTable check (xCoTable IS NOT NULL),
  xCoDefault varchar(64)
  );
```

The xCoDefault field is a bit problematic. We really need some form of ‘polymorphic’ column, but we will simply have this as varchar, which is clumsy. (Most other solutions are more cumbersome, e.g. having a vast number of different fields, with a separate table containing these fields).

Possible column types are listed in Table 9:

Code	*Meaning*
F	floating point
D	date
T	time
S	stamp (date+time)
I	integer
N	numeric (precision, scale)
V	varchar

Table 9: Codes for various datum types

Note that size for type N translates to precision! For all except type N, scale will be 0.

xLIMIT

This table records constraints on table columns.

```
CREATE TABLE xLIMIT (
  cold integer,
  xLiKey integer,
    constraint BadLimitKey primary key (xLiKey),
  xLiName varchar (30),
  xLiType varchar(1),
  xLiColumn decimal(8,0),
    constraint LimitBadColumn foreign key (xLiColumn)
      references xCOLUMN,
    constraint LimitHasNoColumn check (xLiColumn IS NOT NULL),
  xLiTable decimal(8,0),
    constraint LimitBadTable foreign key (xLiTable)
      references xTABLE,
  xLiCheck varchar (256)
  );
```

[Do we REALLY require cold fields for the x-tables??] xLiCheck contains the C (check) expression! Not used much at present. We have a drastically curtailed list of possible constraints, contained in table 10.

Code	*Meaning*
P	primary key
F	foreign key
C	check
X	check IS NOT NULL
N	check IS NULL

Table 10: Various constraints

The general purpose C (check) constraint isn't used much at present because of the tight coupling between input routines and database routines.

14.2 A password table

At present we do not use passwords on the PDA, but on the desktop machine we've set up a (we hope temporary) 'front end' for users that is written in AutoIt, and perches over Windows, allowing users to log-in. To allow us to service this log-in, we have written Perl script that runs on the desktop. This uses the following table, which needs to be manually inserted into the database, and is *not* exported to the PDA as it is not represented in our export scripts, and also has no representation in the meta-tables (Section 14.1).

```
CREATE TABLE USERPWD (
  cold integer,
  Person integer,
   constraint BadPwdPerson foreign key (Person)
     references PERSON,
   pwd varchar(32)    )
```

Note that this table is denormalised (might be subsumed by PERSON) and has no primary key, although Person is effectively a primary key. We have removed it from PERSON deliberately so as not to export the passwords to the PDA. If this table were exported to the PDA, it would be wise to export the md5-encrypted passwords contained in the *pwd* field only after they have been further md5'd with a salt value.

The reference to PERSON does not use ON UPDATE CASCADE, as this feature should never be required if the table is used appropriately.

14.3 The UIDS ‘generator’ table

Because we don’t use auto-incrementing keys, we make use of a ‘key-generating’ table called UIDS.[50]

In the current implementation, tables such as ward and BED are ‘closed’, and therefore don’t have generator columns in UIDS. Likewise (at present) for surgtype. All open tables have a corresponding column entry in UIDS, which is simply named from the table with a prefixed ‘u’, thus:

```
CREATE TABLE UIDS (
  cold integer,
  uKey integer,
  uStatus integer,
  uPerson integer,
  uPROCTYPE integer,
  uProcess integer,
  uEpoch integer,
  uActor2 integer,
  uDrugform integer,
  uAROLE integer,
  uDrug integer,
  uRx integer,
  uInfusionobs integer,
  uPharm integer,
  uPersdata integer,
  uMedscore integer,
  uPainscore integer,
  uRxobs integer,
  uPca integer,
  uSurgtypeob integer,
  uPcasettings integer,
  uRgnobs integer,
  uBadobs integer,
  uNonevent integer,
  uIsProblem integer,
  uStopProc integer,

  uxTable  integer,
  uxColumn integer,
  uxLimit  integer,
  uComment integer,
  uMeasure integer,
  uPrinted integer,
  uPainerror integer
```

[50]Note the implications of this decision for concurrent access and synchronisation; a proper multiuser implementation on the desktop requires use of semaphores, as well as complex recovery mechanisms for the case where critically timed crashes occur during interrogation.

```
                  );
```

Note the references to xTable and so forth (as uxTable) which allows us to generate keys for the metatables!

Populating the metatables

The code to populate the above tables is tedious, as we refer to components of the above tables within themselves!

```
INSERT INTO xTABLE (xTaKey, xTaName) VALUES (1, ’xTABLE’);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (1,      ’xTaKey’,  ’N’,        8,        1);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (2,     ’xTaName’,  ’V’,       15,        1);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (49,   ’cold’,    ’I’,   4,      1);
-- the above added 6/11/2006 !

INSERT INTO xTABLE (xTaKey, xTaName) VALUES (2, ’xCOLUMN’);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (3,    ’xCoKey’,  ’N’,        8,        2);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (4,    ’xCoName’, ’V’,       30,        2);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (5,    ’xCoType’, ’V’,        1,        2);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (6,    ’xCoSize’, ’N’,        4,        2);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable,
                    xCoDefault)
               VALUES (7,    ’xCoScale’,’N’,        2,       2,
                    ’0’);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (8,    ’xCoTable’,’N’,        8,        2);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (50,   ’cold’,    ’I’,   4,      2);
-- the above added 6/11/2006 !

INSERT INTO xTABLE (xTaKey, xTaName) VALUES (3, ’xLIMIT’);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (9,    ’xLiKey’,  ’N’,      8,        3);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (10,   ’xLiName’, ’V’,    30,        3);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (11,   ’xLiType’, ’V’,     1,        3);
```

```
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (12,   ’xLiColumn’,’N’,   8,        3);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (13,   ’xLiTable’,’N’,    8,        3);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (14,   ’xLiCheck’,’V’,  256,        3);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (51,   ’cold’,    ’I’,   4,      3);
-- the above added 6/11/2006 !

INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (1,    ’BadTableKey’,’P’, 1);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (2,    ’BadColumnKey’,’P’, 3);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (3,    ’BadLimitKey’,’P’, 9);

INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn, xLiTable)
              VALUES (4,    ’ColInTable’,’F’, 8 ,1 );
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn, xLiTable)
              VALUES (5,    ’LimitBadColumn’,’F’, 12 ,2);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn, xLiTable)
              VALUES (6,    ’LimitBadTable’,’F’, 13 ,1);

INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (7,    ’ColHasNoTable’,’X’, 8);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (8,    ’LimitHasNoColumn’,’X’, 12);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (9,    ’NullTableName’,’X’, 2);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (10,   ’NullColumnName’,’X’, 4);
INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (11,   ’NullColumnType’,’X’, 5);
```

We also have to populate the UIDS table, and represent it within the metatables:

```
INSERT INTO UIDS (uKey, uStatus, uPerson, uPROCTYPE,
                  uProcess, uEpoch, uActor2,
                   uDrugform, uAROLE,
                  uDrug, uRx, uInfusionobs,
                   uPharm, uPersdata, uMedscore, uPainscore,
                  uRxobs, uPca, uSurgtypeob,
                   uPcasettings, uRgnobs, uBadobs, uNonevent,
                   uIsProblem, uStopproc,
                   uxTable, uxColumn, uxLimit, uComment,
                   uMeasure, uPrinted)
          VALUES (1,    8,       1000,     10,
                  2000,    3000, 1,
                   11,        2,
                  1,     100, 4000,
                   5000,   6000,      7000,      8000,
                  9000,   10000, 11000,
                   12000,         13000,    14000, 15000,
                   16000, 18000,
                   100, 200, 300, 400,
                   17000, 1);

INSERT INTO xTABLE (xTaKey, xTaName) VALUES (4, 'UIDS');

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (15,   'uKey',     'I',   4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (48,   'cold',     'I',   4,      4);
-- the above added 6/11/2006.
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (16,   'uStatus', 'I',   4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (17,   'uPerson', 'I',   4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (18,   'uPROCTYPE','I',4,      4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (19,   'uProcess','I',   4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (20,   'uEpoch',    'I',   4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (21,   'uActor2', 'I',   4,      4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (22,   'uDrugform','I',  4,      4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize,xCoTable)
               VALUES (23,   'uAROLE','I',   4,      4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (24,   'uDrug',    'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (25,   'uRx',      'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (29,   'uInfusionobs','I',4,      4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (30,   'uPharm',  'I',   4,       4);
```

```
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (31,   'uPersdata','I',  4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (32,   'uMedscore','I',  4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (33,   'uPainscore','I', 4,       4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (34,   'uRxobs',  'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (35,   'uPca',    'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (36,   'uSurgtypeob','I',4,       4);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (38,   'uPcasettings','I',4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (39,   'uRgnobs', 'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (40,   'uBadobs', 'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (45,   'uNonevent', 'I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (46,   'uIsProblem', 'I',   4,       4);

INSERT INTO xLIMIT (xLiKey, xLiName, xLiType, xLiColumn)
              VALUES (12,  'BadUidsKey','P', 15);

INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (41,   'uxTable','I',    4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (42,   'uxColumn','I',   4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (43,   'uxLimit','I',    4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (44,   'uComment','I',    4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (47,   'uMeasure','I',    4,       4);
INSERT INTO xCOLUMN (xCoKey, xCoName, xCoType, xCoSize, xCoTable)
               VALUES (52,  'uStopProc','I',    4,       4);
*
-- MUST have the star to indicate 'EOF' (our quirk)
```

Note that for any new tables with keys in UIDS, we must manually insert an xCOLUMN reference, or problems will occur on the PDA when trying to find a new KEY.

The (default) xCoScale for each of the above is zero. Several of the tables are ‘closed’ from the perspective of scripting, and so are not represented in UIDS.

The *uids* field is the primary key of the UIDS table, but in our current application only one row will be present, with uids equal to one.

14.4 Synchronisation Issues

It is conceivable that two different users might each be capturing data on separate PDAs. Each then synchronises their PDA with the desktop database. What about collision between keys/two 'different' patient records being created? The solution to this problem should be rule-based: at synchronisation the program performing this action *must* check for identical hospital numbers, and perhaps even other similarities such as identical (or even similar) surnames and forenames. User input should be requested with very similar matches, perhaps with a recommended course of action. The process should be as follows:

1. First user synchronises their PDA. The 'master' unique ID of the patient, generated on the reference (desktop) system is moved to the PDA, overwriting the temporarily generated PDA value (if not previously synchronised);

2. A second user synchronises PDA 2, and the desktop program notes the collision. The original ('master') ID is moved to this system too.

3. The desktop program keeps a record of the transaction, noting the collision.

You can see that ID generation for *all* tables on any one PDA is only temporary, and the desktop machine must coordinate 'final' IDs. Old IDs on a PDA (known to exist on the desktop) must be retained; new IDs will be generated and then replaced with old ones at the next synchronisation! Such synchronisation will be facilitated (and errors avoided) if each PDA has its own 'number space' for IDs.[51]

14.5 Major Problems

Okaaaay. We had a biiig problem with Ocelot. Our multiline insert statements only *appeared* to succeed. The program didn't actually execute some of them! (Longer ones). Rather than rewriting every bloody line (again), we arranged things so that each compound line is split up before submission to a single line. (See documentation for Perl program).

[51]For example, The system might use IDs between 0 and 999 M-1, individual PDAs using parts of the space from 999 M up to 1000 M-1.