USING POLYHEDRA® CL FOR INTEGRITY AND PERFORMANCE
How Polyhedra's CL triggers allow applications to be simplified, data integrity to be assured, and overall performance to be improved.

Contents

Abstract

Traditional database management systems provide locking and parallel transaction execution mechanisms to allow application programmers to perform updates that maintain an application-level integrity policy, whilst not freezing out other database activity whilst is in progress. The problem is that the use of such mechanisms increase the number of interactions between the database server and its client applications, and this impose a performance hit that can be unacceptable in embedded, real-time applications This paper describes how active database management systems can avoid the need for using locking, with subsequent benefits in system performance and also in the simplification of application code.

ENEA

Introduction

One of the main problems when designing a database is the specification of the business rules that will go around it – in effect, an application-level integrity policy that will need to be enforced in some way. Most Database Management Systems, (DBMS) will enforce a basic set of integrity policies - for example, relational database management systems will guarantee, among other things, that primary key attribute values uniquely identify records, that each attribute value is of the correct type - but in general, a database designer will want to go further than this. Techniques such as normalisation help, but are not sufficient, so with traditional DBMS, the database designer not only specifies the database schema but also a set of rules that each application writer - the person coding up client processes that make use of the database - must follow whenever interrogating and updating the database.

To illustrate this, consider a simple application area: a registry of births, deaths and marriages, to be implemented using relational databases. This could consist of two tables, one to hold information about people - their names and their dates of birth and death - and the other to hold information about their relationships via marriage. To capture the fact that in most countries it is illegal to be married to more than one person at a given time, the database schema is supplemented by guidelines to the application writers, so that whenever the database is updated this rule is maintained. (In a simple case such as this, the database designer could also be the application writer - but in larger applications there will be many more people writing client applications, each addressing part of the problem, than there would be people designing the database structures, so it becomes essential to formalise the rules that application writers are to follow.) Thus in our example application, when updating the database to record a death it is necessary to cancel any related marriage. Likewise, when recording a new marriage it is first necessary to ensure that records exist for the people involved, that neither person is marked as dead, and that there is no non-cancelled marriage record relating to either of them.

It is at this stage that the problems of concurrency arises: if the database queries and the database update are performed as independent operations, then there is the possibility (maybe only slight, but real enough) that another application will update the database between the queries and the update, in such a way to invalidate the previous checks; for example, the intruding operation might record the death of one of the parties involved. Traditionally, this is handled by allowing the application database to 'lock' the database (so that no other client can interfere with the data in such a way that 'my' change results in an integrity violation); for performance reasons, a fine granularity is offered, so that whilst one application is busy other applications can still perform various other, non-conflicting operations. With this approach, the process of recording a marriage might be:

- signal the database that we wish to start a new ‘transaction’;
- search for the record for one of the parties involved, using a DBMS primitive that indicates that the record - if found - is to be locked;
- if the search is unsuccessful, signal the end of the transaction, otherwise check whether the person is flagged as dead. If so, cancel the transaction, which will automatically cancel the lock previously established;
- assuming the previous search succeeded and the person is alive, now repeat the procedure search for the other person being married, cancelling the transaction as before if problems are noted - in which case, both locks are automatically cancelled;
- now check for existing marriages, and simultaneously lock the marriage table so that no other client process can add a record that might clash with what is being done; and, finally,
- check the results of the marriage queries, and either cancel the whole transaction or insert a new marriage record and signal that this completes the transaction: in either case, the outstanding locks will be cancelled.

In practice, some of these steps may be rolled together, but as can be seen, the operation is not simple, and requires a number of interactions with the database; worst of all perhaps, it puts the onus on the application coder to properly implement the rules specified by the database designer. It also significantly complicates the design and implementation of the database management system, which has to be able to cope with multiple simultaneous transactions, each of which may be cancelled at any time, requiring the cancellation of any locks and the unwinding of any changes so far performed within the transaction. The DBMS also has to check for conflicting lock requirements, holding up or cancelling transactions as and where needed to avoid or resolve conflicts. Finally, the developers of the database management system have to provide a rich set of locking mechanisms, to allow the database designer to avoid unnecessary restrictions on the kind of transactions that can be performed in parallel.

Particularly in real-time systems where performance is critical, resources are limited but where the typical operations are not complex, alternative techniques are needed. The rest of this paper considers one such technique, called active databases, shows how it can reduce the need for concurrent transactions, and describes a commercially-available active DBMS that support these features. A sample database definition, complete with active code, is given as an appendix.

Active databases

The definition of an active DBMS used in this paper is a DBMS that provides the database designer the opportunity to attach code to a schema that is automatically triggered when changes occur; an active database is one whereby the database designer has made use of this facility. To be useful, the granularity of the code triggering should be as fine as possible, so that different code is run according to the change that has been made, but the critical points are that:

- the code is run as part of the same operation that requested the update;
- the code has the ability to specify additional changes which are done along with the requested change; and,
- there is a mechanism of cancelling the operation complete with any changes made by the triggered code (causing the database to roll back to the state prior to the operation).

Going back to our earlier example, the operation of recording a new marriage can be simplified enormously. The rule to be followed by the application engineer reduces down to the single step: insert a new marriage record. All the rest of the checks are performed automatically, partly by the DBMS as a result of careful schema design, and partly by the code attached to the schema by the database designer.

In both approaches, the responsibility for specifying the integrity policy lies with the database designer, and part of the enforcement mechanism is embedded in the database schema developed by the database designer. However, instead of just specifying the rules for the remaining part of the enforcement mechanism, the database designer also encodes these rules and attaches them to the database.

Note that this approach does not restrict the application coder, merely removes some of the responsibility and reduces the number of database interactions that are required; even if the application performs its own checks, no locking is needed since the final checks are performed by the active code as part of the insert operation. Thus as far as overall integrity is concerned it does not matter if the application performs a number of steps:

- query the database to retrieve the records for the people, which might be expressed in SQL as:

```
select id, birthdata, birthplace, deathdate
       from person
       where name = 'John Doe'
       or    name = 'Anne Smith';
```

- analyse the response to determine that at least one John Doe and one Alice Smith is alive, and then to choose - perhaps interacting with a human user - which of these is to be married. Note that this stage involves no interaction with the database, that the DBMS need not have any recollection of the earlier query, and that the database may change during this phase;
- finally, add the database record for the new marriage, for example

```
insert into marriage
              (locationcode, id, husband, wife, mdate)
       values (12345, 1289, 568746282, 834772012, now ());
```

... or just simply perform the last step, using information supplied from driving licences, say. In terms of performance, however, avoiding the need to query and lock the database can lead to some quite substantial improvements, as the number of round trips between client and server are reduced.

Other benefits from this approach include the ability at a later stage to enhance the integrity rules without the need to rewrite the existing applications. For example, it is obvious that the active database could add in checks such as whether the people getting married are of legal age and - where it is a legal requirement! - of differing gender; also, if the database were to be altered so as to be able to record parentage, it would then be possible to check whether the want-to-be newlyweds are too closely related.

Note that not all active DBMS'es offer the same facilities, and thus some may allow the database designer to enforce stricter rules than others would allow. For example, a DBMS that runs the triggered code after all the client-requested changes have occurred might mean that the active code does not work quite as expected if related changes are made to the database within a single transaction - if in our earlier example a batch of changes entered at the end of a month include both a person's marriage and also a record of his or her subsequent death in a plane crash, then the active code might reject the operation because the marriage refers to someone who (by the time the test is done) is already marked as dead. Obviously, more careful coding (such as checking the death date against the marriage date should the death date be specified, instead of just insisting that the death date is null) would in this instance have averted the problem, but in general the database designers is made much easier if:

- the appropriate active code is performed by the DBMS after each individual command within the operation requested by the client;
- the coding language is powerful, yet clear to read;
- the DBMS allows a very fine granularity to the way active code is attached to the database, for example allowing different code to be triggered depending on which attributes are altered; and,
- if the triggered code requests changes to the database, then this may itself trigger further active code to be triggered, to be performed in-line.

Denormalisation

It was mentioned earlier that it is common to normalise relational databases to improve database integrity. In a normalised database, the schema is designed so that a given item of information is kept only in one place, and that no information is explicitly kept that can be deduced from other information in the database. Thus, the principle of normalisation would say that one does not have a flag on the person table to say whether they are married, since an appropriately-designed query on the marriage table will reveal this fact; this helps overall database integrity, since if there is no such flag there is no risk that its value differs from that revealed by the query. However, normalisation can be expensive in operation, since a query to determine, say, married men requires more than one table to be inspected...

```
select p.id, p.name
       from  person p, marriage m
       where m.husband = p.ID
       and   m.enddate is null;
```

... and a query to find eligible bachelors between the age of 18 and 30 becomes very complex. If using an active DBMS, however, it becomes possible to add in extra attributes so as to simplify queries (and/or the active code itself) without risking integrity. Thus, suppose the active database code maintained a 'married' attribute in records of type person, then the eligible bachelors query becomes a fairly simple query on a single table:

```
select id, name from person
       where male = true
       and   born + years (25) > now ()
       and   born + years (18) < now ()
       and   married is null and died is null;
```

(The exact syntax of the query will vary according the DBMS in use.)

An active database management system

Real-time applications have some database requirements that can put considerable strain on conventional databases: in particular, the ability to run on cheap processing equipment, to run in small amounts of memory, to be fast (simple queries and transactions should take at most a few milliseconds even when using slow processors), the ability to run with no backing store, and the ability to be able to keep clients up to date cheaply. On the other hand, the amount of data that needs to be held is typically small - a database with 500,000 records is usually considered large - and most transactions are small (and succeed), so it becomes possible to design DBMS'es specifically for this kind of use.

The Polyhedra DBMS (http://www.polyhedra.com/) was designed and built from scratch for this marketplace. For speed and coding simplicity, it normally uses an extremely simple transactional model: complete serialisation. A client initiates a transaction by sending down a series of changes; no changes are made in the database until the complete set is received, at which time the transaction is added to the end of a FIFO queue. The database management system processes transactions from the front of the queue in turn, and each is run to completion - unless a failure is detected, in which case the system “rolls back” the database to its state prior to the transaction. Read requests are handled by the same queuing mechanisms, so clients can only see the state of the database “between transactions”. However, as Polyhedra is an active DBMS, one can see from this paper that using this simplistic transaction model is not as restrictive as users of traditional database management systems might suppose. The appendix uses the Polyhedra trigger langauge in its description of the schema and trigger code for a marriage database implementing a sophisticated series of checks.

A supplementary approach - optimistic concurrency control.

In order to keep clients up to date, Polyhedra has an 'active query' mechanism; if a client launches an active query, the server responds with an initial set of results, and then remembers the query until it is cancelled by the client. When a change occurs in the database that invalidates the previously-returned results, the DBMS will automatically detect this, and calculate and despatch a 'delta' - a set of change information to allow the client to bring itself up to date. This mechanism avoids the need for clients to poll the database to detect changes, and so, although it takes work in the DBMS to track active queries and calculate deltas, the overall workload is reduced for a given level of responsiveness. This active query mechanism has another purpose: as described in the following paragraphs, it can be used to supplement the active code mechanism to give the appearance of concurrent transactions without the complexity and overheads of locking.

In essence, to use this mechanism, the client launches a number of active queries, and then can 'block' a group of them which flags the database not to send up any deltas for those queries. At a later stage, the queries can be 'unblocked', in which case a single set of deltas might be issued to bring the client up to date; alternatively, rather than just unlocking the queries, a set of changes can be sent through to the DBMS, in which case one of three things can happen:

1. One or more of the requested changes may clash with a change that occurred in the database and that would have been reported to the client if the query(s) had not been blocked - in this case, the changes are abandoned, the queries unblocked, and deltas generated to get the client up to date;
2. no clashes are detected, but an integrity violation is noted (either by the DBMS itself or by the active code attached to the database), in which case the changes are rolled back, the queries unblocked, and deltas generated to bring the client up to date; or,
3. no clashes are detected, and the changes can be implemented, together with any changes specified by the active code; the transaction completes, the queries unblocked and deltas generated to bring the client up to date with the resultant state.

This 'optimistic concurrency' mechanism does not enforce integrity - that is done by the active code attached to the database - but is useful where users - in this case, typically humans acting through client processes - are making a number of changes to an object, and want to be sure that if someone is making clashing changes to the same object at the same time then only one of those sets of changes succeed.

Summary

Using active databases allows the database designer to move much of the integrity enforcement from the client applications into the database; with fine-granularity triggers, the need for locking and concurrent transactions can be engineered out of the application. Real-time databases can rely on this and avoid the need for providing any locking or parallel execution support, especially if a high-level optimistic concurrency control mechanism can be provided to supplement the active nature of the database.

Appendix: the marriage database

Given below is the schema and active code for a marriage database, showing many of the features discussed in the paper. The schema is written in the dialect of SQL, and the active code is written in the CL programming language developed for and used in the Polyhedra active relational database management system.

The schema

The example below makes use of Polyhedra's 'create schema' extension, which allows a group of tables to be introduced together; these tables may reference any of the tables in the group being introduced, in addition to already-existing tables. (The 'traditional' approach is to define the tables without the columns that make forward references, and then use 'alter table' to add in the missing columns once all the tables have been defined. Such an approach makes it difficult to read the table definition files, and thus adds scope for errors.)

```
create schema
    create table person

    (   persistent
    ,   id      integer primary key
    ,   name    char not null
    ,   born    datetime
    ,   died    datetime
    ,   ismale  bool
    ,   married integer references marriage
    )
    create table marriage

    (   persistent
    ,   id       integer primary key
    ,   wife     integer not null references person
    ,   husband  integer not null references person
    ,   started  datetime not null
    ,   location char
    ,   ended    datetime
    ,   whyended char
    )
;
```

The active code

The active code mechanism in Polyhedra allows the database designer to attach code fragments, or 'methods', written in a special coding language called CL with the various tables in the database. CL is reasonably self-explanatory, once you note that it is case-insensitive and that:

- the 'on create' method are triggered immediately after the object have been created in the relevant table, and 'on set <attributename>' methods are triggered by changes in the values of an attribute – in both cases after any entity and referential integrity checks have been performed, and before any other changes are made by the SQL or CL code that triggered the method.
- backslash '\' is the line continuation character, and '--' flags that the rest of the line is a comment;
- the 'of' operator can be used to reference a field in another record, and CL sees foreign key attributess as pointers to the referenced record; and,

- ampersand ‘&’ and double-ampersand ‘&&’ are string concatenation operators, with the latter causing an additional space character between the strings being concatenated.

```
script person
    on create
        if born > now () then
            abort transaction "date of birth must not be in the future"
        else if exists died then
            -- the record has been created with the birth date and death date set -
            -- which is fine, as long as both dates are in the past and in a
            -- sensible order...
            if died < born then
                abort transaction "cannot die before being born"
            else if died > now () then
                abort transaction "cannot know when" && name && "will die"
            end if
        end if
    end create
    on set died
        if not exists died then
            abort transaction "cannot raise the dead"
        else if died < born then
            abort transaction "date of death must not precede birth"
        else if died > now () then
            abort transaction "cannot know when" && name && "will die"
        else if exists married then
            -- tell the marriage record that I am dead and thus the marriage is over;
            -- a trigger on the marriage.ended column will do what else is needed...
            set whyended of married to "death of" && name
            set ended    of married to died
        end if
    end set died
    on  set ismale
        -- we could rule out gender changes, or rule it out if the person is
        -- currently married, or cancel the marriage.
        -- let's do the last of these...
        if exists married then

            set whyended of married to name && "had a sex change"
            set ended of married to now ()
        end if
    end set ismale
end script
```

```
script marriage
    -- (the CL 'script' statement introduces a group of methods to be attached to the
    -- named table. 'end script' flags the end of the group of methods.)
    on  create
        -- (in Polyhedra, 'on create' methods are triggered when objects are created in
        -- the database, whether by means of an SQL INSERT statement or by CL code.)
        -- perform various checks:
        if started > now () then
            abort transaction "start date cannot be in the future"
        else if exists ended then
            -- the 'ended' field is not null - so this record does not affect the
            -- status of who is married to whom. Consequently, we can ignore the record.
            exit -- skip the rest of the handler.
        else if exists married of husband then
            abort transaction name of husband && "is already married to" && \
                              name of wife of married of husband
        else if exists married of wife then
            abort transaction name of wife && "is already married to" && \
                              name of husband of married of wife
        else if not ismale of husband then
            abort transaction "husband should be male," && name of husband && "is not"
        else if ismale of wife then
            abort transaction "wife should be female," && name of wife && "is not"
        else if exists died of husband or exists died of wife then
            abort transaction "both spouses must be alive to marry"
        end if
        -- to have got here, all checks have passed: update the other records in
        -- the database for consistency
        set married of husband to me
        set married of wife to me
    end create
    on  set husband
        -- on set handlers are triggered when an attribute is altered (whether by an
        -- SQL UPDATE, or a CL set statement); here, we can simply ban the change.
        abort transaction "cannot alter the husband of a marriage"
    end set husband
    on  set wife
        abort transaction "cannot alter the wife of a marriage"
    end set wife
    on  set started
        abort transaction "cannot alter the start date of a marriage"
    end set started
    on  set ended
        -- one flags a marriage as over by setting this attribute non-null
        if not exists ended then
            -- end date altered from non-null to null: application error
            abort transaction "cannot unset end date of a marriage"
        else if ended < started then
            abort transaction "a marriage cannot end before it begins!"
        else if married of husband = me then
            -- marriage was current; update the database to reflect the fact that it
            -- isn't. No need to check 'married of wife = me' unless really paranoid!
            set married of husband to null
            set married of wife    to null
        else
            -- the marriage was not the current marriage of the husband & wife - so
            -- something must be trying to alter the end date of a marriage record
            -- that was already marked as ended. complain.

            abort transaction "cannot alter end date of a marriage"
        end if
    end set ended
end script
```

Further code can be added, depending on one's level of paranoia - for example, one can add code to ensure that the married attribute of a person is always either null or pointing at a live marriage record that itself points at this record. In addition, the inbuilt security mechanisms can restrict who can make what alterations, and attributes (such as married) can be set up so that no-one can alter them directly - in which case the only alterations possible are those done by triggered code (which runs fully privileged, on behalf of the database owner).

Sample SQL

The following SQL statements illustrate how the above database can be populated and queried by clients.

- First, create some initial records in the person table...

```
insert into person (id, name, born, ismale) values
                   (1, 'Adam', date ('01-jan-1951'), true);
insert into person (id, name, born, ismale) values
                   (3, 'Ben', date ('01-dec-1953'), true);
insert into person (id, name, born, ismale) values
                   (5, 'Chris', date ('01-nov-1955'), true);
insert into person (id, name, born, ismale) values
                   (7, 'David', date ('01-oct-1962'), true);
insert into person (id, name, born, ismale) values
                   (9, 'Edward', date ('01-sep-1959'), true);
insert into person (id, name, born, ismale) values
                   (11, 'George', date ('01-aug-1961'), true);
insert into person (id, name, born, ismale) values
                   (13, 'Hugh', date ('01-jul-1963'), true);
insert into person (id, name, born, ismale) values
                   (15, 'Ian', date ('01-jun-1965'), true);
insert into person (id, name, born, ismale) values
                   (17, 'John', date ('01-may-1967'), true);
```

```
insert into person (id, name, born, ismale) values
                   (19, 'Keith', date ('01-apr-1969'), true);
```

```
insert into person (id, name, born, ismale) values
                   (100, 'Anne', date ('01-jan-1951'), false);
insert into person (id, name, born, ismale) values
                   (102, 'Betty', date ('02-feb-1952'), false);
insert into person (id, name, born, ismale) values
                   (104, 'Carol', date ('03-mar-1953'), false);
insert into person (id, name, born, ismale) values
                   (106, 'Doris', date ('04-apr-1954'), false);
insert into person (id, name, born, ismale) values
                   (108, 'Emma', date ('05-may-1955'), false);
insert into person (id, name, born, ismale) values
                   (110, 'Fiona', date ('06-jun-1956'), false);
insert into person (id, name, born, ismale) values
                   (112, 'Gina', date ('07-jul-1957'), false);
insert into person (id, name, born, ismale) values
                   (114, 'Helen', date ('08-aug-1958'), false);
insert into person (id, name, born, ismale) values
                   (116, 'Ina', date ('09-sep-1959'), false);
insert into person (id, name, born, ismale) values
                   (118, 'Jane', date ('10-oct-1960'), false);
commit;
```

- Now, create some valid `marriage` records...

```
insert into marriage (id, husband,wife,started, location) values
                     (1, 1, 100, date ('11-jul-1970'), 'London');
insert into marriage (id, husband,wife,started, location) values
                     (2, 3, 102, date ('21-jun-1972'), 'London');
insert into marriage (id, husband,wife,started, location) values
                     (3, 5, 104, date ('07-may-1971'), 'Cambridge');
insert into marriage (id, husband,wife,started, location) values
                     (4, 7, 106, date ('11-jul-1970'), 'Edinburgh');
insert into marriage (id, husband,wife,started, location) values
                     (5, 9, 108, date ('30-Dec-1977'), 'Edinburgh');
insert into marriage (id, husband,wife,started, location) values
                     (6, 11, 110, date ('10-Mar-1978'), 'Edinburgh');
commit;
```

- … a divorce and a remarriage...

```
update marriage set ended=date('03-Apr-1983'), whyended='Divorce'
       where id = 3;
insert into marriage (id, husband,wife,started, location) values
                     (7, 5, 112, date ('15-jul-1983'), 'London');
commit;
```

- a death and a remarriage...

```
update person set died=date('09-Feb-1985') where id = 1;
insert into marriage (id, husband,wife,started, location) values
                     (9, 13, 100, date ('15-May-1986'), 'Cardiff');
commit;
```

- … two more divorces...

```
update marriage set ended=date('24-Aug-1986'), whyended='Divorce'
       where id = 8;
update marriage set ended=date('18-Sep-1986'), whyended='Divorce'
       where id = 6;
update marriage set ended=date('02-Oct-1986'), whyended='Divorce'
       where id = 7;
commit;
```

- … and end with 4 weddings and a funeral.

```
insert into marriage (id, husband,wife,started, location) values
                     (10, 11, 112, date ('10-May-1990'), 'London');
insert into marriage (id, husband,wife,started, location) values
                     (11, 15, 114, date ('11-Jun-1990'), 'London');
insert into marriage (id, husband,wife,started, location) values
                     (12, 17, 116, date ('12-Jul-1990'), 'London');
insert into marriage (id, husband,wife,started, location) values
                     (13, 19, 104, date ('13-Aug-1990'), 'London');
update person set died=date('09-Feb-1985') where id = 106;
commit;
```

We can now look at some sample queries and show the results that would be obtained assuming the above SQL had been used to set up the contents of the database.

- show the married people:

  This could be as simple as 'select id, name from person where married is not null', or it could be enhanced to show the person's age at the time of marriage. This latter request requires information from both the person and the marriage table, but as one is restricting the query using a foreign key in one table to select records in the other, the query can be performed efficiently.

```
select p.id id, p.name name, ismale,
       the_year(m.started) married,
       the_year(m.started-p.born) aged
from   person p, marriage m
where  p.married=m.id;
```

id	name	ismale	married	aged
3	Ben	TRUE	1972	19
9	Edward	TRUE	1977	19
11	George	TRUE	1990	29
13	Hugh	TRUE	1986	23
15	Ian	TRUE	1990	26
17	John	TRUE	1990	24
19	Keith	TRUE	1990	22
100	Anne	FALSE	1986	36
102	Betty	FALSE	1972	21
104	Carol	FALSE	1990	38
108	Emma	FALSE	1977	23
112	Gina	FALSE	1990	33
114	Helen	FALSE	1990	32
116	Ina	FALSE	1990	31

- list the dead people:

```
select id, name dead, ismale, the_year(died-born) aged
from   person
where  died is not null;
```

id	dead	ismale	aged
1	Adam	TRUE	35
106	Doris	FALSE	31

- list the unmarried people:

```
select id, name, ismale, the_year(now()-born) aged
from   person
where  married is null and died is null;
```

id	name	ismale	aged
5	Chris	TRUE	44
7	David	TRUE	37
110	Fiona	FALSE	43
118	Jane	FALSE	39

- display information about the marriages:

Here, in order to display the names of the people being married we a performing a 3-table join (marriage x person x person), but as in the earlier example we are using foreign keys in one table to select records in the others, and so it can be performed efficiently.

```
select   the_year(m.started) year, location,
         h.name husband, the_year(m.started-h.born) age_h,
         w.name wife,    the_year(m.started-w.born) age_w,
         the_year(m.ended) ended, whyended
from     marriage m, person h, person w
where    m.husband=h.id and m.wife=w.id
order by m.started;
```

year	location	husband	age_h	wife	age_w	ended	whyended
1970	Edinburgh	David	8	Doris	17	1998	death of Doris
1970	London	Adam	20	Anne	20	1998	death of Adam
1971	Cambridge	Chris	16	Carol	19	1983	Divorce
1972	London	Ben	19	Betty	21	NULL	NULL
1977	Edinburgh	Edward	19	Emma	23	NULL	NULL
1978	Edinburgh	George	17	Fiona	22	1986	Divorce
1983	London	Chris	28	Gina	27	1986	Divorce
1986	Cardiff	Hugh	23	Anne	36	NULL	NULL
1990	London	George	29	Gina	33	NULL	NULL
1990	London	Ian	26	Helen	32	NULL	NULL
1990	London	John	24	Ina	31	NULL	NULL

1990	London	Keith	22	Carol	38	NULL	NULL

We end up by looking at the responses produced when various 'bad' changes are made to the database structure.

```
SQL> update person set died=NULL where id = 1;
SQL> commit;
Error: cannot raise the dead
```

```
SQL> insert into marriage (id, husband,wife,started, location) values
                          (100, 1, 110, date ('10-Aug-1995'), 'London');
SQL> commit;
Error: both spouses must be alive to marry
```

```
SQL> insert into marriage (id, husband,wife,started, location) values
                          (101, 110, 5, date ('10-Aug-1995'), 'London');
SQL> commit;
Error: husband should be male, Fiona is not
```

```
SQL> insert into marriage (id, husband,wife,started, location) values
                          (102, 5, 110, date ('10-Aug-2095'), 'London');
SQL> commit;
Error: start date cannot be in the future
```

```
SQL> insert into marriage (id, husband,wife,started, location) values
                          (102, 19, 110, date ('10-Aug-1995'), 'London');
SQL> commit;
Error: Keith is already married to Carol
```

```
SQL> update marriage set husband = 5 where husband = 19;
SQL> commit;
Error: cannot alter the husband of a marriage
```

```
SQL> update marriage set ended = date ('01-Jan-1930') where husband = 19;
SQL> commit;
Error: a marriage cannot end before it begins!
```

```
SQL> update person set died =  date ('01-Jan-2130') where id = 19;
SQL> commit;
Error: cannot know when Keith will die
```