Case study **27**

Conway's Game of Life

Objective. This study is about games, worlds, life and death, borderline SQL applications and dramatic database optimization. The goal of the project is to implement the graphical animation of Conway's cellular automata, aka *Game of Life*. A game of life is made up of an infinite array of cells in which live a population of small *animals*, each of them occupying one cell. The transition of one state of the population to the next one is specified by a set of simple computing rules. The goal of the game is to observe and study the evolution of the population. A game of life is implemented as a table in a database in which each row contains the coordinates and the content of a cell. The algorithms developed in this study load the initial state of a population then compute the next states thanks to the evolution rules. Finally, they visualize this evolution as an animated cartoon. The contribution of this study is twofold. It stresses the importance of database and algorithm optimization (the last version is 1,400 times faster than the first one) and it shows that relational databases and SQL may be quite efficient to develop matrix manipulation procedures (the SQL version is nearly 7 times faster than the equivalent Python program).

This study is also a tribute to E. F. Codd, the inventor of the relational model of databases, who first studied self-replicating cellular automata.

Keywords. cellular automata, replicating system, Conway, glider, Codd, matrix manipulation, algorithm optimization, database optimization, declarative algorithm, table indexing, in-memory database, CTE, recursive query, vector graphics, animated simulation, Python.

27.1 Introduction

Cellular automata are mathematical objects invented by J. H. Conway in 1970 in search for self replicating systems. A cellular automaton lives in a 2-dimensional grid made up of square cells. It has an initial state that evolves as a sequence of states derived according to a small set of elementary transformation rules. Each state is defined by certain cells containing a small critter of which we only know that it is present or not. The presence of such an animal in a cell (which is then called a *live cell*) is represented by a black square while an empty cell (a *dead cell*) is white.

In Figure 27.1, the grid shows the initial state of the most simple *glider* (top), a popular object that moves like a wriggling, hyperkinetic worm crawling down the grid diagonal. It comprises 5 live cells. After 4 steps, it recovers its form but one step down along the diagonal. And so on. Forever ...

The evolution rules are very simple. To compute the next state of a cell of an automaton, we count those of its eight neighbors that are live. Let N be this number. Then we apply the following transformation rules:

– if N = 2, the cell contents is preserved, be it live or dead,

– if N = 3, the cell becomes (or remains) live,

– for all the other values of N (0, 1, > 3), the cell becomes dead.

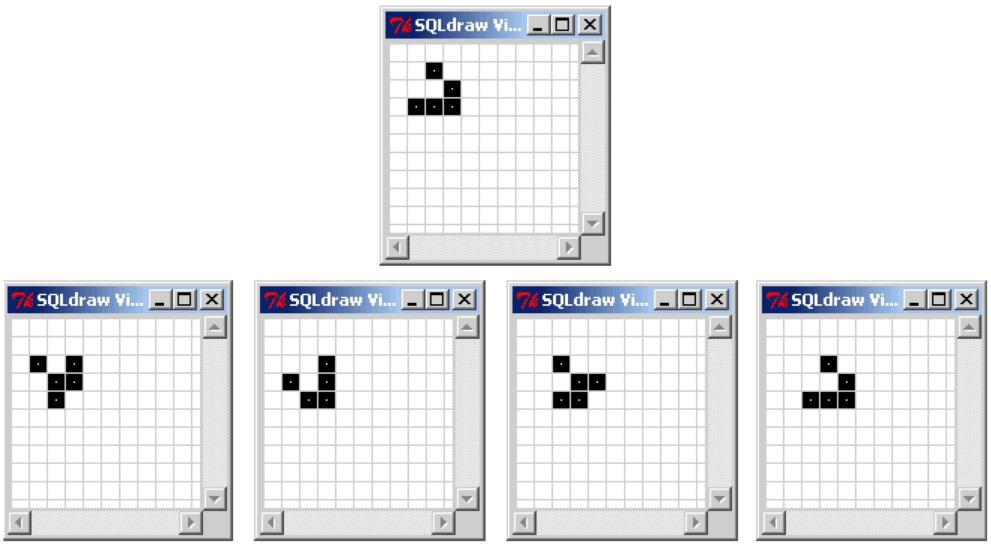


Figure 27.1 - The starting first five states of a simple Glider

For reasons that are not hard to guess, this system has been given the popular name of *Game of Life*. It has triggered much research among computer science theoretists[1], epidemiologists studying how diseases spread and vanish[2] and in some gamer communities[3]. In particular, several families of patterns have been identified: *still* (each state is the same as the initial one), *gliders* (moving patterns), *guns* (that oscillate while producing *shells*, that may be gliders), *oscillators*, *mirrors*, *bumpers*, *spaceships*, etc. Some patterns move, grow infinitely[4], vanish, oscillate, replicate themselves or produce other patterns. Just like societies in real life! See http://en.wikipedia.org/wiki/Conway's_Game_of_Life as a good starting point.

This topic may come as a surprise in a text devoted to databases and database programming[5]. Indeed, animating cellular automata mainly requires much computation and fast graphical rendering. Not precisely an application domain in which database technology is renowned for! Or is it?

However, by trying to develop cellular automata with the SQLfast environment, we will encounter several interesting challenges the solution of which can be generalized to other, more mainstream, database problems.

In addition, we will once more appreciate the expressiveness and power of SQL. As an appetizer, just consider that cellular automata can be elegantly and efficiently implemented **in 2 (two) SQL queries**.

27.2 World and automata representation

Cellular automata live in a infinite world. Due to reason that are not worth being made explicit, we will reduce them to finite worlds, represented by arrays. A world is represented by table CELL in which each row describes a cell of an array. CELL has three columns:

– **I**, that indicates the line number of the cell,

– **J**, that represents the column number of the cell,

– **C**, that indicates whether the cell is live (`1`) or dead (`0`).

For instance, row `(4,4,1)` tells that cell `4,4` is live while row `(3,3,0)` denotes a dead cell. Table CELL is defined as follows:

```
create table CELL(I integer, J integer, C integer);
```

1. The rules mentioned above define a system that have been proved to be Turing-complete.
2. Thanks to more sophisticated automata, with more than two states (*healthy*, *infected*, *immune/dead*) and probabilistic evolution rules
3. OK, not quite the same as that of WoW!
4. Seemingly, because this cannot be neither observed nor proved.
5. It is interesting to note that E. F. Codd, the inventor of the relational model of databases, formerly worked on cellular automata. See for instance E. F. Codd, *Cellular Automata*, Academic Press, New York (1968)

Its primary key is (I,J) but we do not declare it for now. Similarly, we do not declare column *not null*. The scripts that will make use of this table (as well as of the second one) are guaranteed to insert correct data.

Computing the next state of an automaton requires that we can analyze all the cells of the current state unmodified. Therefore, we need an additional array in which we will store the data required to compute the next state without altering the current state. In this array, we will store, for each cell, the number of live neighbours. Then, from this information, we will be able to update the contents of table CELL to create the next state by applying the evolution rules.

This additional array is implemented as table SCORE, with the same structure as CELL, and in which column N represents the number of live neighbors of cell (I,J):

```
create table SCORE(I integer, J integer, N integer);
```

For instance, row `(2,3,1)` of SCORE tells that cell `(2,3)` in CELL has only `1` live neighbor, which is no good news for its viability! Same for the cell corresponding to row (`3,3,5`). On the contrary, row (`4,4,2`) tell that cell (`4,4`) will keep its former state.

Now, we are able to create the database (Script 27.1). Since it will be used only to compute the successive steps, we declare it as a non persistent, *in-memory* database. A world that comprises **`maxLin`** lines and **`maxCol`** columns is initialized by the insertion of **`maxLin`** x **`maxCol`** cells with `C = 0` (Script 27.2). To make the graphical layout more elegant, and to make cell evaluation easier, we leave the first line and column empty, that is, they are not part of the world. For our experiments, a 50 X 50 square world will prove comfortable enough. Later, we will show that this algorithm is not fast enough for our ambition, so that we will call it *First* (or *naive*) *version*.

We also install the initial state of the *glider* of Figure 27.1 (Script 27.3).

```
createDB InMemory;
create table CELL (I integer,J integer,C integer);
create table SCORE(I integer,J integer,N integer);
```

Script 27.1 - Life Game: creation of the database

```
for I = [0,$maxLin$];
  for J = [0,$maxCol$];
    insert into CELL values ($I$,$J$,0);
  endfor;
endfor;
```

Script 27.2 - Life Game: creation of a **maxLin x maxCol** world - First version

```
update CELL set C = 1 where I = 1 and J = 2;
update CELL set C = 1 where I = 2 and J = 3;
update CELL set C = 1 where I = 3 and J = 1;
update CELL set C = 1 where I = 3 and J = 2;
update CELL set C = 1 where I = 3 and J = 3;
```

Script 27.3 - Life Game: creation of a Glider

27.3 Computing the next state

To compute the next state of the automaton we proceed in two steps: computing the number of live neighbor cells and updating the current state.

First, we compute the number of live neighbors of each cell and we store these numbers in table SCORE. This can be done in a single **insert** SQL query. The current cell under examination is denoted by alias **C0** while its neighborhood is explored by alias **C1**. The relation between **C0** and **C1** (i.e., the *join condition*) is that coordinate **I** of **C1** is between **C0.I**-1 and **C0.I**+1 and coordinate **J** of **C1** is between **C0.J**-1 and **C0.J**+1. But we must exclude **C0** itself of course. Value **N** of cell **C0** is the sum of the values of **C** of its neighbor cells. Hence the query of Script 27.4.

The second step consists in updating column **C** of each **CELL** row according to the number of its live neighbors, previously stored in **SCORE**. This is expressed by a `case` construct that assigns the appropriate (0,1) value to column **C** (Script 27.5).

```
insert into SCORE(I,J,N)
select C0.I, C0.J, sum(C1.C)
from   CELL C0, CELL C1
where  C1.I between (C0.I - 1) and (C0.I + 1)
and    C1.J between (C0.J - 1) and (C0.J + 1)
and    not (C1.I = C0.I and C1.J = C0.J)
group by C0.I, C0.J;
```

Script 27.4 - Computing the neighborhood of the current state - First version

27.4 Graphical display of automaton evolution

Technically speaking, the job is done. However, it would be nice to show graphically, as a *cartoon*, the successive states of an automaton. We suggest to convert these states into a sequence of vector graphics pictures expressed in the SQLdraw language. These pictures, written in a text file with extension ***.draw**, can then be rendered through statement **showDrawing** (see Chapters 13 and 21 of the Tutorial).

```
update CELL
set C = (select case
                when N<2 then 0
                when N=2 then CELL.C
                when N=3 then 1
                when N>3 then 0 end
         from SCORE S
         where S.I = CELL.I and S.J = CELL.J);
```

Script 27.5 - Deriving the next state from the current one - First version

The first part of this file, shown in Script 27.6, defines the useful part of the world (**area** `192,192`) expressed in pixels. Its origin (`0,0`) is at the top-left corner of the graphical space. In this parts the border of the cells are drawn as a grid made up of two polylines:

`polyline "Vgrid",1,"gray80"`: defines the vertical lines of the grid

`polyline "Hgrid",1,"gray80"`: defines the horizontal lines of the grid.

A polyline is a contiguous chain of segments defined by their connection points. Statement **polyline** assigns a name to the polyline and specifies the thickness and the color of its segments (the fourth parameter is ignored). It is followed by the list of the coordinates of its points.

```
area 192,192
polyline "Vgrid",1,"gray80",""
0,192
0,0
0,192
12,192
12,0
...
polyline "Hgrid",1,"gray80",""
192,0
0,0
192,0
192,12
0,12
...
```

Script 27.6 - SQLdraw script creating the grid of a game

When a state is available, we generate the SQLdraw command **points** that displays its current population. This command comprises a header, follows by a series of points, each positioned on a live cell. The header assigns a name to the set of points and specifies its shape: thickness and color of its border polygon, filling color and the point type, here type 23 (see the point catalogue in Chapter 21 of the Tutorial). Point type 23 is a black 11x11 pixel square with a small white hole in its center. If we allow room for a separation of `1` pixel for the grid, each cell is `12` pixel large. Each point is defined by its coordinates and an optional label. The coordinates of a point are those of its center, so, the graphical coordinates of cell I,J are (6 + 12 * J, 6 + 12 * I).

Animation is created as follows. Once a state has been drawn, a wait period of, say, 150 ms., is started (`wait 150`), after which, the last state is erased (`delete last`). Then, the next state is drawn.

For instance, the first two states of the *Glider* automaton (Figure 27.1) can be displayed by Script 27.7.

```
points "life-0",1,"black","black",23
30,18,""
42,30,""
18,42,""
30,42,""
42,42,""
wait 150
delete last
points "life-1",1,"black","black",23
18,30,""
42,30,""
30,42,""
42,42,""
30,54,""
wait 150
delete last
```

Script 27.7 - SQLdraw script of the first two states of the *Glider*

27.5 Generation of SQLdraw scripts.

Script 27.8 generates the **`points`** command of the current state from the contents of table CELL. Variables Lnbr and Cnbr specify the numbers of the last line and column of the world. Note that the `delete` statement is lacking. It is generated elsewhere to avoid erasing the very last state.

27.6 Packaging the LIFE application

We have enough material to define the architecture of our LIFE application. Remember that the application available in directory **Case_Life_Game** may be a bit different (more sophisticated) from the description given below. More on this in the last section.

The main script could look like Script 27.9.

```
write points "life-$seq$",1,"black","black",23;
for I = [0,$Lnbr$[;
   for J = [0,$Cnbr$[;
      extract R = select C from CELL where I = $I$ and J = $J$;
      if ($R$ = 1);
         compute X = 6 + $J$ * 12;
         compute Y = 6 + $I$ * 12;
         write $X$,$Y$,"";
      endif;
   endfor;
endfor;
if ($Delay$ > 0) write wait $Delay$;
```

Script 27.8 - Generation of the SQLdraw script of the current state - First version

It first comprises a simple dialogue to collect the useful world size (variables `Lnbr` and `Cnbr`), the number of successive states to compute (variable `Niter`) and the wait delay, in ms (variable `Delay`). Then, it calls the three database creation procedures: _LIFE-Create-Database.sql for the database, _LIFE-Create-World.sql that inserts the world cells in the database and _LIFE-Create-Glider.sql that installs the initial state of the *Glider* of Figure 27.1. Procedure _LIFE-Compute-History.sql is in charge of computing and generating the `Niter` successive states of the automaton in text file `LIFEGAME-Glider.draw`. Finally, it shows the animated rendering of the Niter states through statement **`showDrawing`**.

```
set Lnbr,Cnbr,Niter,Delay = 15,15,40,150;
ask-u Lnbr,Cnbr,Niter,Delay = [Simulation parameters]
      Lines:|Columns:|Iterations:|Delay (ms):;
   if ('$DIALOGbutton$' = 'Cancel') exit;
execSQL _LIFE-create-Database.sql;
execSQL _LIFE-create-World.sql;
execSQL _LIFE-create-Glider.sql;
execSQL _LIFE-compute-History.sql;
showDrawing LIFEGAME-Glider.draw;
```

Script 27.9 - The main script of the LIFE application

Procedure _LIFE-Create-Database.sql includes the statements of Script 27.1. Procedure _LIFE-Create-World.sql is defined from Script 27.2 while procedure _LIFE-Create-Glider.sql is defined from Script 27.3.

The most interesting procedure obviously is _LIFE-Compute-History.sql, that does all the job. Its code is shown in Script 27.10. It starts by opening external file LIFE-GAME-Glider.draw, that will receive the SQLdraw statements [1] and finishes by redirecting the output channel to `window`, which closes the external file [7].

Statements [2] and [3] insert the points (the *live cells*) of the starting state of the automaton. For each of the following states,

– statement [4] inserts an SQLdraw `delete` statement to erase the last state from the drawing area,

– statement [5] computes the next state as suggested in Scripts 27.4 and 27.5,

– statement [6] inserts into the SQLdraw script the points (the *live cells*) of the current state.

```
outputOpen LIFE-draw.draw;                                      [1]
set seq = 0;                                                    [2]
execSQL _LIFE-Generate-Current-State.sql;                       [3]
for seq = [1,Niter];
   write delete last;                                           [4]
   execSQL _LIFE-Compute-Next-Step.sql;                         [5]
   execSQL _LIFE-Generate-Current-State.sql;                    [6]
endfor;
outputAppend window;                                            [7]
```

Script 27.10 - Procedure *_LIFE-Compute-History.sql*

27.7 First performance analysis

We would like the LIFE application to show the result as soon as the parameters have been entered. Unfortunately, this is not the case, even if the speed of computing and generating 30 states of the *Glider* automaton in a 15x15 world seems to be acceptable with this respect: 2.6 s. We guess than wider worlds, larger automata examined through hundreds of states will require much longer computing and generation time.

Trying to improving the performance of the LIFE application is an interesting challenge. It will allow us to better understand how a database and a database application work and to experiment with various ways to boost the performance of a database.

To measure the impact of performance improvement techniques, we need a reference point. Computing 100 successive states of the *Glider* automaton in a 60x60 world seems a good starting point: it is sufficiently small to imply a *bearable* computing time with the current version and sufficiently large to let us compare the successive optimization. The run times of the main four procedures are shown in Figure 27.2. A total of about **14** minutes is quite impressive! Not surprisingly, the major part of the cost (> 99.6%) is that of procedure LIFE-compute-History.sql.

procedure	raw version
Create-Database	0.000
Create-World	2.7
Create-Glider	0.002
Compute-History	835.715
Total	**838.417**

Figure 27.2 - Computing times of the raw version (Glider, 60X60 world, 100 iterations)

27.8 Optimization 1: indexing tables

Let us first examine the way the database is used. Procedure LIFE-Create-World.sql is a sequential process while LIFE-Create-Glider.sql makes a small number of random updates. There is no real room for improvement in the first three procedure, at least for now.

Procedure LIFE-Compute-History.sql is our main target. It includes heavy database access and update queries:

- In procedure LIFE-Compute-Next-Step, query 27.4 executes a *self-join* followed by a *group by* operator on table CELL. Both operators are based on columns **I** and **J**.
- In procedure LIFE-Compute-Next-Step, query 27.5 is a massive update of table CELL from table SCORE. The 60x60 cell updates require each a random access to SCORE based on (**I**, **J**) values.
- In procedure LIFE-Generate-Current-State, the double loop executes, for each state, 60x60 random accesses to table CELL, that is, 3,600 `select` queries on column **I** and **J**, that is, 360,000 queries for the 100 states.

From this simplified analysis, we conclude that an index on column **I** and **J** on both tables CELL and SCORE should decrease access time in all these queries.

Let us try that. We change the contents of procedure LIFE-Create-Database.sql by Script 27.11, that declares these indexes.

```
createOrReplaceDB LIFE.db;
create table CELL (I integer,J integer,C integer);
create table SCORE(I integer,J integer,N integer);
create index XCELL  on CELL (I,J);
create index XSCORE on SCORE (I,J);
```

Script 27.11 - Life Game - Creation of the *LIFE.db* database with indexes

The resulting execution times (Figure 27.3) are quite encouraging.

procedure	raw version	indexed
Create-Database	0.000	0.001
Create-World	2.7	2.7
Create-Glider	0.002	0.001
Compute-History	**835.715**	**442.752**
Total	**838.417**	**445.454**

Figure 27.3 - Indexing the database more than halves the run time

Encouraging but still excessive. Waiting more than 7 minutes for the show will not attract a large audience! Actually, procedure LIFE-Compute-History.sql mixes database queries and non-database statements, so that its global execution time figure is too gross to allow us to identify other ways of improvement.

Splitting this time into computation time (LIFE-Compute-Next-Step) and generation time (LIFE-Generate-Current-State) could be quite informative. We change the code of script LIFE-Compute-History.sql a little bit, by deactivating the two statements `execSQL LIFE-Generate-Current-State.sql` (we merely prefix them with the `'--'` comment indicator). The resulting times when executing the raw and indexed versions are stunning (Figure 27.4): the time drops from 566.45 s. to 6.36 s.! Subtracting these times from the global times provide the execution time of LIFE-Generate-Current-State.[6]

6. To be quite precise, this time also includes the execution time of the proper statements of procedure LIFE-compute-History.sql and of procedure LIFE-generate-grid.sql. However, these times are very small and can be ignored. For instance, generating the 60x60 grid takes 0.4 sec.

procedure	**raw version**	**indexed**
Create-Database	0.000	0.001
Create-World	2.7	2.7
Create-Glider	0.002	0.001
Compute-History	835.715	442.752
- Compute-Next-Step		**16.547**
- Generate-Current-State		**426.205**
Total	**838.417**	**445.454**

Figure 27.4 - Splitting times between computing and generation

Clearly, our optimization effort should now concentrate on the **generation process**.

27.9 Optimization 2: *windowing* table access

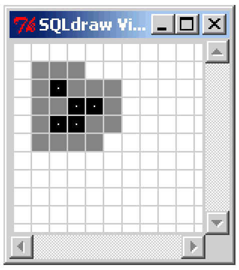
The SQL queries of procedures LIFE-compute-History.sql and LIFE-Generate-Current-State visit all the cells of the world (and, in parallel, those of the SCORE table, which has the same size). In fact, when we consider the unitary transition from one state to the next one, we observe that the set of cells that could change often is quite small.

Considering the fourth state of the *Glider* automaton represented in Figure 27.1, the cells that are likely to change in the next state are (1) the five live cells and (2) their seventeen neighbor cells. They are shown in the left array of Figure 27.5: live cells in black and their neighbors in gray. So, we only need to compute the next state for these cells, ignoring all the white cells. Similarly, when generating the SQLdraw statements of the current state, examining the white cells would be pure wasted time.

However, the area defined in this way can be irregular and quite difficult to define. We will therefore consider a simpler area, a bit larger than necessary, but easier to compute: the *bounding rectangle*. It comprises the smallest rectangle that completely contains the automaton plus a border one cell large (Figure 27.5, right array).

The coordinates (I1,I2,J1,J2) of this bounding rectangle can be computed by the following SQL query:

```
select min(I)-1 as I1, max(I)+1 as I2,
       min(J)-1 as J1, max(J)+1 as J2
from CELL where C = 1;
```



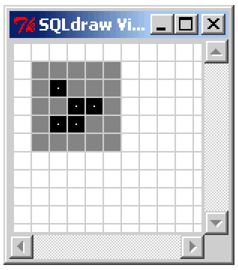


Figure 27.5 - The bounding area of an automaton (strict and rectangular)

This formula must be refined a little bit to avoid negative values for the cells at the edge of the world:

```
select max(min(I)-1,0) as I1, min(max(I)+1,$Lnbr$-1) as I2,
       max(min(J)-1,0) as J1, min(max(J)+1,$Cnbr$-1) as J2
from CELL where C = 1;
```

Its translation in the SQLfast syntax is straightforward:

```
extract I1,I2,J1,J2 =
        select max(min(I)-1,0),min(max(I)+1,$Lnbr$-1),
               max(min(J)-1,0),min(max(J)+1,$Cnbr$-1)
        from CELL where C = 1;
```

We can now adapt (what we will call *windowing*) the SQL queries so that they only examine and process the cells that lie in the (I1,I2,J1,J2) rectangle (Scripts 27.12 and 27.13). Let us now measure the effect of this optimization (Figure 27.6).

procedure	**raw**	**indexed**	**window**
Create-Database	0.000	0.001	0.001
Create-World	2.7	2.7	2.7
Create-Glider	0.002	0.001	0.001
Compute-History	835.715	**442.752**	**4.17**
- Compute-Next-Step		**16.547**	**0.33**
- Generate-Current-State		**426.205**	**3.84**
Total	**838.417**	**445.454**	**6.87**

Figure 27.6 - The effect of windowing SQL queries

The improvement is really dramatic: the execution time of LIFE-Compute-History has been divided by more than **100**!

```
insert into SCORE(I,J,N)
   select C0.I, C0.J, sum(C1.C)
   from   CELL C0, CELL C1
   where  C0.I between $I1$ and $I2$
   and    C0.J between $J1$ and $J2$
   and    C1.I between (C0.I - 1) and (C0.I + 1)
   and    C1.J between (C0.J - 1) and (C0.J + 1)
   and    not (C1.I = C0.I and C1.J = C0.J)
   group by C0.I, C0.J;

update CELL
set C = (select case
                when N<2 then 0
                when N=2 then CELL.C
                when N=3 then 1
                when N>3 then 0
                end
         from SCORE S
         where S.I = CELL.I and S.J = CELL.J)
where  I between $I1$ and $I2$
and    J between $J1$ and $J2$;
```

Script 27.12 - *Windowing* the SQL queries of procedure LIFE-Compute-History.sql

```
for I = [$I1$,$I2$[;
   for J = [$J1$,$J2$[;
      extract R = select C from CELL where I = $I$ and J = $J$;
      if ($R$ = 1);
         compute X = 6 + $J$ * 12;
         compute Y = 6 + $I$ * 12;
         write $X$,$Y$,"";
      endif;
   endfor;
endfor;
```

Script 27.13 - *Windowing* the loops of procedure LIFE-generate-Current-State.sql

27.10 Optimization 3: building a cheap world

At this stage, one of the highest cost becomes that of creating the world (2.7 sec.) by procedure LIFE-Create-World.sql. The reason is the same as for the generation process: the SQLfast interpretation of the two embedded loops. There is several ways to decrease the building cost. We will examine three of them.

1. Executing a predefined script that creates the world

   Instead of generating on the fly and executing the `insert` queries as shown in Script 27.2, we store these queries in a standard SQLfast script file for further reuse. The cost of creating the world is that of executing this script,. This cost is much lower (about 0.05 s.) but this technique requires that we create one or several such predefined scripts.

2. Using a preloaded database

   The idea is similar: we create a collection of predefined databases in which worlds of various sizes have been preloaded. Same improvement and draw-back as above.

3. Creating the world with a recursive query

   We keep Script 27.2, but we translate the double loop into a recursive CTE. The script is shown in Script 27.14. Outstanding result: the 60 X 60 world is created in **0.008** s. instead of **2.7**. And no cumbersome additional scripts or databases!

   This technique is quite elegant, but also is a bit less intuitive for users less familiar with recursive queries, and therefore deserves some explanations.

   The first recursive CTE, **INIT1**, creates rows (I,0,0) for I in [0,maxLin]. In other terms, it creates the cells of the *first column* of the maxLin X maxCol world.

   The second CTE, **INIT2**, starts from these cells to create the remaining cells of each line of the world.

```
with recursive
  INIT1(I,J,C)
  as (select 0,0,0
        union
      select I+1,J,0 from INIT1 where I < $maxLin$),
  INIT2(I,J,C)
  as (select * from INIT1
        union
      select I,J+1,0 from INIT2 where J < $maxCol$)
insert into CELL select * from INIT2;
```

Script 27.14 - Building a world through a recursive query

Considering the last technique, we get the result shown in Figure 27.7. The total execution time now drops to **4.18** sec., less than **1** sec. of which being spent by the execution of SQL queries. We can admit that this figure is quite satisfying.

procedure	raw	indexed	window	world
Create-Database	0.000	0.001	0.001	0.001
Create-World	2.7	2.7	**2.7**	**0.008**
Create-Glider	0.002	0.001	0.001	0.001
Compute-History	835.715	442.752	4.17	4.17
Total	**838.417**	**445.454**	**6.87**	**4.18**

Figure 27.7 - Creating a world through a recursive query

27.11 Optimization 4: let SQL generate vector graphics

Now, the highest cost is again that of procedure Compute-History, and more specifically procedure Generate-Current-State (run time 3.84 s.). Its code is that of Script 27.13. It comprises two embedded loops parsing the live cells of the current world window. For each of these cells, two elementary computations and the writing of their result as a short string.

This can easily be done by an SQL query. Let us convert the current script into pure SQL. We consider temporary table LINE in which we will insert the SQLdraw commands that draw the points of the successive generations of the automaton:

```
LINE(TXT)
```

The rewriting of Script 27.13 as an SQL query is shown in Script 27.15. The contents of table LINE is then copied in the SQLdraw file.[7]

The run time of this generation dramatically drops, from **3.84 s.** to 0.593 - 0.33 = **0.26 s.** (Figure 27.8).

```
insert into LINE
  select cast(6+I*12 as char)||','||cast(6+J*12 as char)||',""'
  from   CELL
  where  C = 1
  and    I between $I1$ and $I2$ and J between $J1$ and $J2$;
```

Script 27.15 - Rewriting Script 27.13 as a pure SQL query

7. This way of working could be disputed. Indeed, we postulate that the order of the rows of the result set will be that of the insertion queries. Since table LINE has no key nor index, most DBMS will implement it as a pure sequential structure, *first in, first out*. If we are uncomfortable with this hypothesis, we could add columns STEP, that represents the current step, and ORD, that specifies the position of the command in the SQLdraw file. When writing in this file, the TXT values are output in `order by STEP,ORD`. This will just add a dozen milliseconds to the total cost.

procedure	raw	indexed	window	world	graphics
Create-Database	0.000	0.001	0.001	0.001	0.001
Create-World	2.7	2.7	2.7	0.008	0.008
Create-Glider	0.002	0.001	0.001	0.001	0.001
Compute-History	835.715	442.752	4.17	4.17	0.593
Total	**838.417**	**445.454**	**6.87**	**4.18**	**0.603**

Figure 27.8 - Generating vector graphics with an SQL query

27.12 Optimization 5: Rewriting the application in Python

Of course, rewriting the procedures in a standard programming language, like Java or Python, should lead to still better times. Quite surprisingly, the improvement is more subtile than it could be thought. Indeed, the generation of the SQLdraw file through a Python program that has the same structure as the SQLfast application (including windowing) and that implements CELL and SCORE by lists of lists instead of database tables leads to a run time of **3.52** s., that is, nearly **6 times slower** than SQLfast version.[8] If we focus on the core computing, that is on the work of procedure **Compute-Next-Step.sql**, we observe that computing the 100 states by the Python program costs **2.48** s., to compare with **0.33** s. for the SQLfast application: the SQL engine appears to be **more than 7.5 times faster** than Python.

procedure	raw	indexed	window	world	graphics	Python
Create-Database	0.000	0.001	0.001	0.001	0.001	
Create-World	2.7	2.7	2.7	0.008	0.008	
Create-Glider	0.002	0.001	0.001	0.001	0.001	
Compute-History	835.715	442.752	4.17	4.17	0.593	
Total	**838.417**	**445.454**	**6.87**	**4.18**	**0.603**	**3.52**

Figure 27.9 - Generating drawing with an SQL query

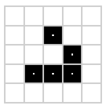
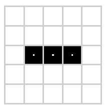
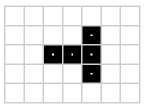
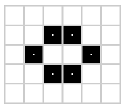
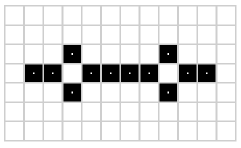
To be fair, this is just one experiment, on one small automaton, implemented in a dynamic language that is not known for its lightening speed (C or even Java could

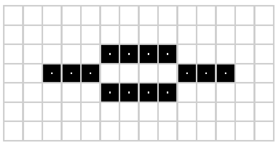
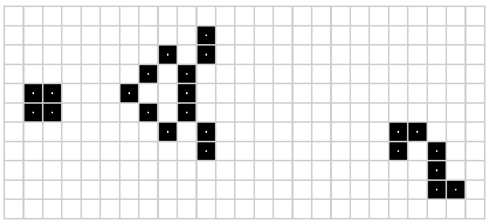
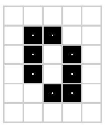
8. Program **LIFE-Game-in-Python.py** is included in the SQLfast distribution.

do better). However, it provides an interesting information on the ability of an SQL engine to efficiently perform matrix manipulation.

27.13 Some representative Conway's automata

The literature and the many dedicated web sites provide countless automata, the starting states of which range from three to several thousands live cells. For instance, try video clip http://www.youtube.com/watch?v=C2vgICfQawE for a dramatic presentation of outstanding patterns, some of them comprising hundreds of thousands of cells. We present here below some of the most popular cellular automata.

Starting state	Name and family
	**Elementary glider**. This automaton moves but does not grow. It has a period of 4 states.
	**Blinker**. The simplest oscillator with two distinct states.
	**Blinker2**. After a dozen evolution states, explodes into four simple Blinkers.
	**Beehive**. Still automaton. Each life cell has exactly two live neighbors.
	**Pentadecathlon**. Oscillator with a period of 15 states.

	**FlyingSaucer**. Transforms into six simple Blinker.
	**Buckaroo**. Oscillator with a period of 30 states.
	**Upside-Down**. Expands then shrinks into four still 2x2 squares.

27.14 Family life

The initial state of the *Gosper Gun* automaton comprises four entities: two bumpers (the left and right side 2x2 squares) and two generating entities (we will call them the *parents*) moving horizontally in opposite directions (Figure 27.10). Together, the parents form the so-called *Gun*. When a parent touches a bumper, it moves in the reverse direction, toward its partner. When the parents meet, they generate a little glider (as is natural for parents), after what they reverse their direction and target the bumpers.

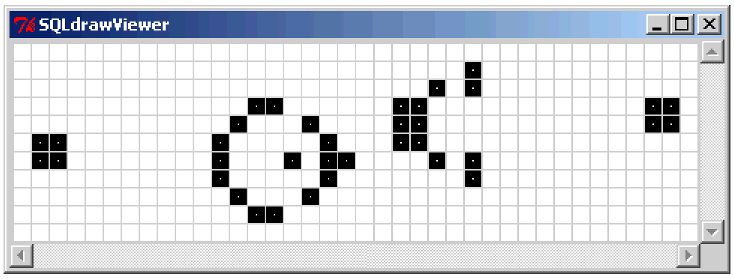


Figure 27.10 - Initial state of the Gosper Glider Gun

The *Gosper Gun* has a period of 30 states. This means that every 30 iterations, the couple of parents (the *Gun*) recovers its initial state (Figure 27.11) and has generated a new glider. The total time to compute and generate 90 states, that is three periods, is **2.6** s.

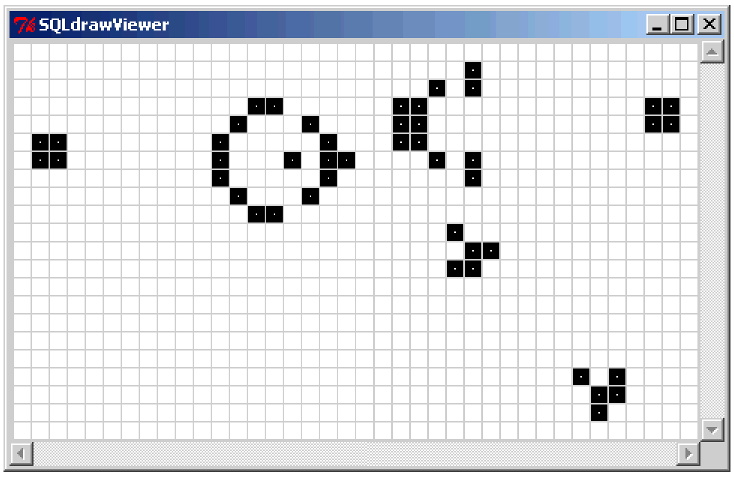


Figure 27.11 - The Gosper Gun at its 60th states. Two gliders have been generated.

27.15 Lessons learned

As it has been usual in most case studies developed so far, the very goal of this one is not to solve a specific problem, i.e., animating Conway's cellular automata,[9] but rather to illustrate general strategies of problem solving. One of the most interesting aspect of this study is the various ways a slow naive solution can be improved. Let us recall and comment the design decisions leading to the creation of worlds and automata and to the computing of their successive states.

1. **Using an *in-memory* database**. The main cost of data intensive applications is that of external memory access (such as reading from and writing to magnetic disk). Maintaining the data needed by the application in main memory allows keeping this cost as low as possible. Making the whole database reside in main memory is one of the simplest way to avoid this cost.[10] This is partic-

9. Let us be honest, solving this problem is fun but not terribly useful in the real life!
10. The other technique consists in assigning a large buffer to the database so that the data are read only once from the disk, then kept in the buffer, so that further access will find them in main memory, saving costly disk access. This technique requires less main memory space but may be less efficient than an *in-memory* database.

ularly effective when the database is transient (which is the case of our database) and fairly large.

2. **Dropping integrity constraints**. As a general rule, the critical properties of the data must be expressed as integrity constraints. *Uniqueness* constraints must be ensured by *primary* and *unique keys*, *referential* constraints must be translated into *foreign keys* and *mandatory* values into `not null` constraints. More complex properties can be expressed by `check` predicates or by `triggers`. This is quite true for general purpose databases, that are modified by several agents, from external sources, both likely to be unreliable. Of course, this security has a cost, the ROI being a high data quality. In some cases however, this cost can be avoided, notably when only one reliable process is in charge of updating the data. This (hopefully!) is the case of the application we have developed in this study. This is why we have declared no primary nor foreign keys and no `not null` constraints.
3. **Creating indexes**. An index, if carefully designed, reduces the cost of accessing a set of selected rows in a table (for reading or updating). And this, all the more so as the table is large and this set is a small part of this table. Though we have experimented on very small tables, we have observed a drastic improvement in execution time of the application.
4. **Reducing the problem space**. The first, naive, algorithms were built to explore, analyze and potentially update all the cells of the worlds. This process was useless for most of these cells. Identifying the smallest possible subset of cells likely to be usefully examined and modified has lead to an impressive gain of execution time. This identification is an essential part of the problem modeling process.
5. **From procedural to declarative expressions**. This last aspect may be one of the most important in the context of this series of case studies. Most programmers, be they professional or casual, tend to apply a procedural reasoning when they have to solve problems such as those of this case study. The main pattern of this reasoning could be informally translated as:

   for each element **`e`** of set **`E`**, execute action **`A`** on **`e`**.

   Hence the multiple loops that structure the algorithms of the naive version. Most such loops translate a higher level declarative reasoning that can be expressed into an SQL query, recursive[11] or not. These queries are preprocessed by the optimizer of the SQL engine and translated into an access plan quite often much more efficient than the procedural version. In our application, the gain was very dramatic.

Relational databases and the SQL language are often considered to be poorly fit to matrix manipulation. Surely, they are not the primary tools we can think about for

11. Many recursive procedures can be converted into loop-based structures. What we suggest here is to apply the converse and to let the SQL engine apply this conversion. This approach is natural to functional and logic programmers, but less familiar to mainstream developers.

such operations. However, this study shows that, in some cases, when they are carefully crafted, SQL applications can provide simple, intuitive and quite efficient solutions without requiring the use of special-purpose languages and environments.

Note: trying alternative algorithms

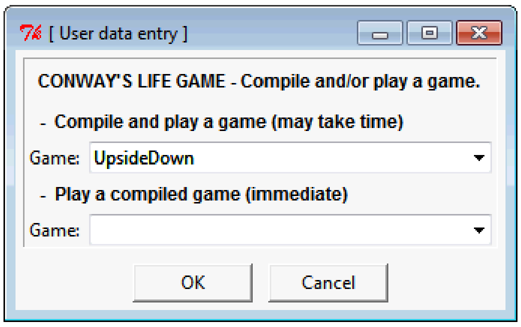
The technique applied in this study may be the most intuitive but it is not the only one that have been documented. The alternative approach described below is as simple but may be more efficient:

> *Instead of visiting all the neighbors of each cell, be it live or dead, as we did in this study, we examine* ***each live cell*** *and add 1 to the score of to all its neighbors. An index*[12] *on column C of CELL is likely to help a lot.*

Its implementation is fairly easy and is left as an exercise.

27.16 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Life_Game**. Actually, they can be run from main script **LifeGame-MAIN.sql**, that displays the selection box of Figure 27.12, through which the user can choose to compile and run a new game or to run a game previously compiled. We call *compiling a game* the transformation of the specification of this game (such as through Script 27.3) into executable code, that is, the corresponding SQLdraw file.


[User data entry]

CONWAY'S LIFE GAME - Compile and/or play a game.

- Compile and play a game (may take time)

Game: UpsideDown

- Play a compiled game (immediate)

Game:

Figure 27.12 - Selecting a game to compile and run, or a script to run

12. In particularly a partial index that references live cells only (`where C = 1`).

The standard format of the scripts that create cellular automata (*games of life*) is shown in Script 27.16. It includes the `insert` queries that create the initial state in table CELL as well as the optimal settings of the following parameters:

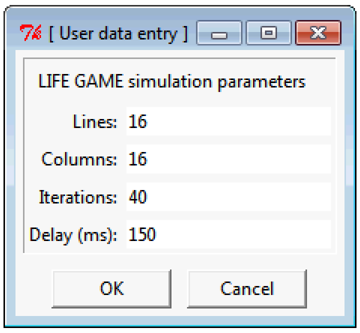
- Lnbr: number of rows of the world shown in the graphical window
- Cnbr: number of columns of the world shown in the graphical window
- Niter: number of states to compute
- Delay: wait time between each state when the automaton is animated.

```
set Lnbr = 15;
set Cnbr = 15;
set Niter = 40;
set Delay = 150;

update CELL set C = 1 where I = 1 and J = 2;
update CELL set C = 1 where I = 2 and J = 3;
update CELL set C = 1 where I = 3 and J = 1;
update CELL set C = 1 where I = 3 and J = 2;
update CELL set C = 1 where I = 3 and J = 3;
```

Script 27.16 - Creation of the standard Glider (script Create-LIFEGAME-Glider.sql)

If the setting statements are missing or incomplete, a new data entry box opens, letting the user set the parameters at its preferred values (Figure 27.13).



LIFE GAME simulation parameters

Lines: 16
Columns: 16
Iterations: 40
Delay (ms): 150

Figure 27.13 - Completing the parameters of the selected game

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.