EXTENDING DATABASE FUNCTIONALITY THROUGH EMPRESS PERSISTENT STORED MODULES

Serge Savchenko
Empress Software Inc.

This paper reviews the relevance of Persistent Stored Modules in the user driven augmentation of the existing functionality in a Relational Database Management System. The second part of the article includes a step-by-step example of a Persistent Stored Module implementation, which provides a solution to a real life data retrieval challenge.

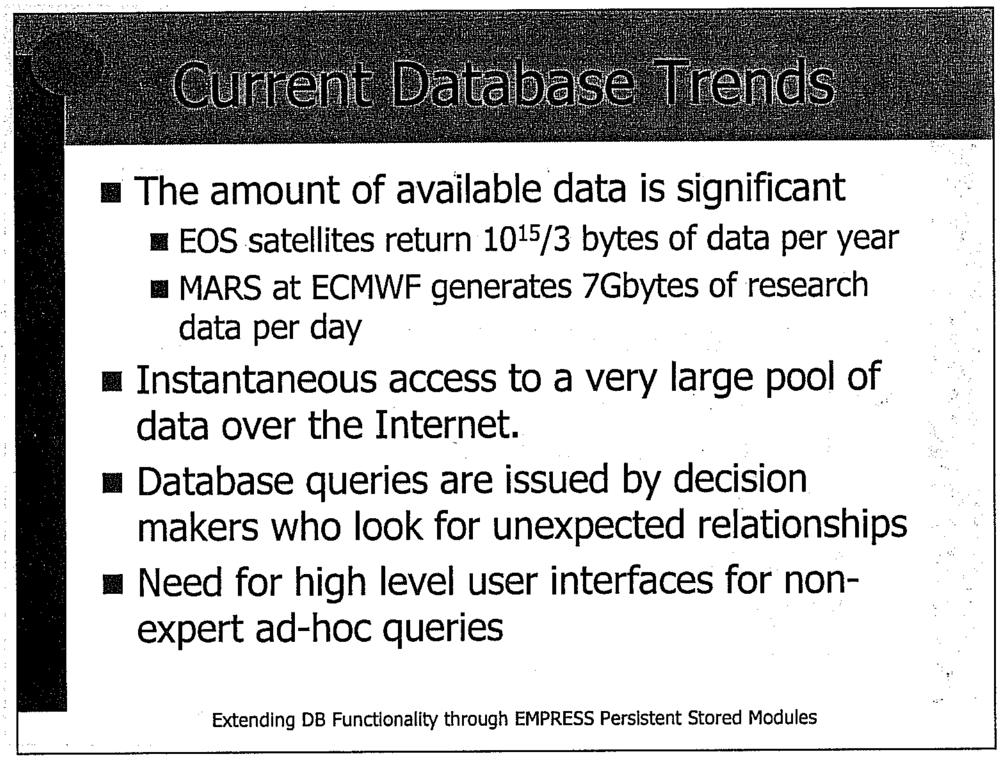
1. INTRODUCTION

The explosion in digitized information, hardware capacity and computer communication has placed significantly higher demands on database systems. While at its core database systems still store data and provide access to information, the methodology of data retrieval has been elevated to a new level of complexity.

There is no lack of research data available today. The Earth Observation Systems (EOS) alone return $10^{15}/3$ bytes of data reflecting trends in the Earth atmosphere, oceans and land mass. The ECMWF Meteorological Archival and Retrieval System (MARS) grows at the rate of 7 GB per day. In some cases, the process of data retrieval and analysis buckles under the rate of new data acquisition.

The Internet significantly increases the total pool of research data, multiplying the challenge of data retrieval and data analysis many folds.

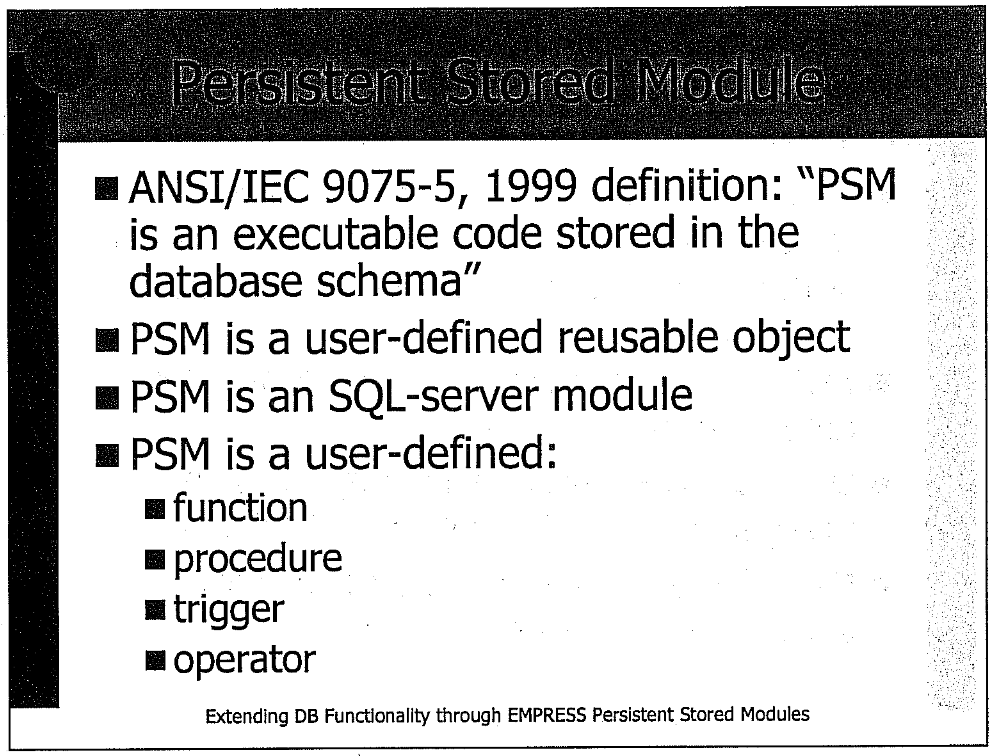
Empress Software has opened a new chapter in RDBMS technology by allowing users to extend the existing capability of their database engines through the use of Persistent Stored Modules.



This paper emphasizes the ease in development of new Persistent Stored Modules and their inclusion into a database schema. The result is a radically different paradigm for a database system where algorithms are placed "next" to user data. The new model unites the executable code and user data under the umbrella of the term "database."

This paper is about the benefits of such a model. In order to illustrate the use of a PSM, an example citing a data retrieval problem and a PSM based solution for it are included in latter part of the article.

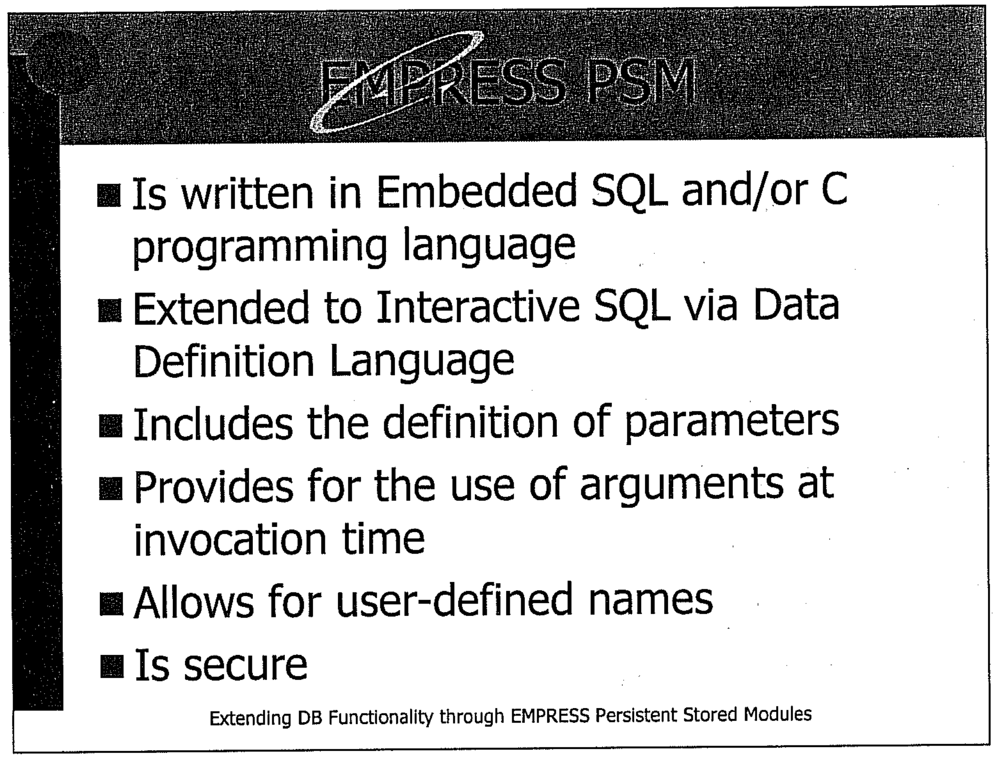
The Summary briefly discusses the main advantages of the Persistent Stored Module implementation.



2. PERSISTENT STORED MODULE DEFINITION

Both ANSI/IEC9075-5, 1996 and soon to be formalized SQL-3 standards define Persistent Stored Module (PSM) as "an executable code stored in the database schema". Thus by definition, PSM is a database schema object. The Module itself can contain one or more routines which are addressable elements. Thereby each routine is also a schema object. Each and every schema object is managed within the scope of the database.

The routines within the PSM are also known as *User-Defined Function, Stored Triggers and Procedures* or *Operators*.



3. PERSISTENT STORED MODULE

IN EMPRESS SOFTWARE IMPLEMENTATION

Empress Software has implemented its PSM in accordance with the ANSI standard. The main benefit of adhering to the widely recognized standard is popularization and therefore a degree of brand name independence. Here are other benefits of the EMPRESS PSM implementation:

- A user can use the widespread C-programming language and/or Embedded SQL to develop Empress schema objects (user-defined routines), therefore there is no need to learn another proprietary programming language in order to code the routines for Empress Persistent Stored Modules.

- Empress Data Definition Language (DDL) was augmented at the Interactive SQL level; here are the examples of the additional commands:

  CREATE MODULE,

  DISPLAY MODULE,

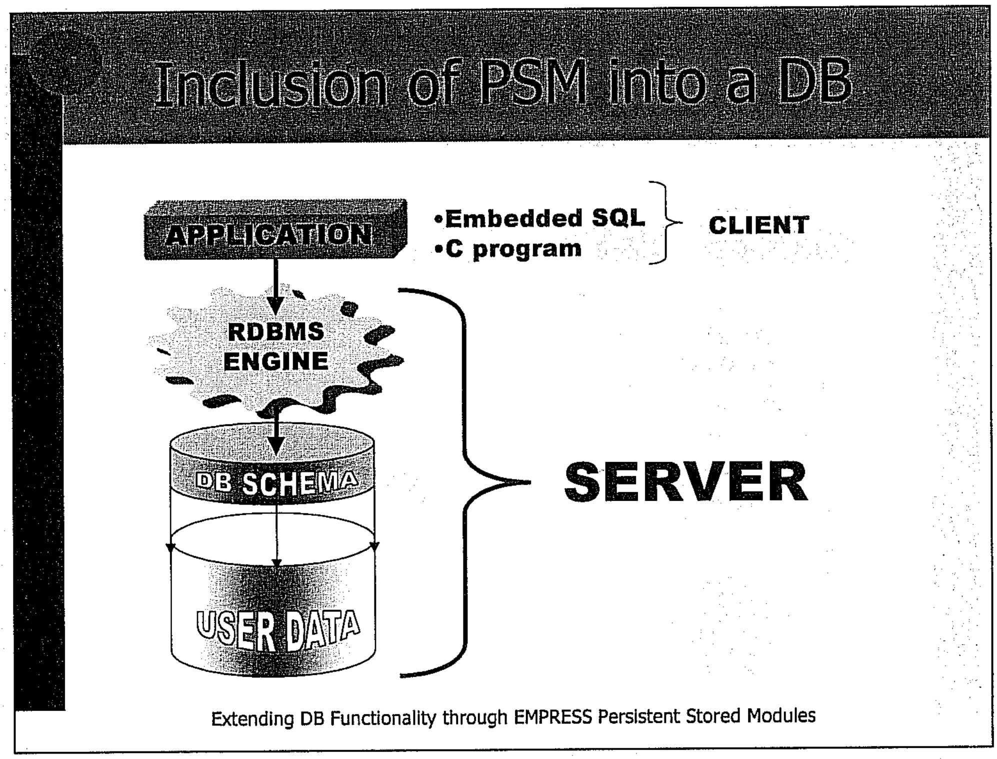
  UPDATE MODULE,

  CALL

  and others. All of these commands make handling of the executables objects in the database easy and natural.

- User-defined functions and procedures include the definition of parameters thus allowing for the use of arguments at the execution time. This is a very important property if the EMPRESS PSM implementation as it provides for a dynamic mechanism, which will process relevant user data at the time of invocation.

- PSM's are governed by the same authorization methods as user data in the database.



4. FUNCTIONAL DIVISION OF A TYPICAL DATABASE APPLICATION

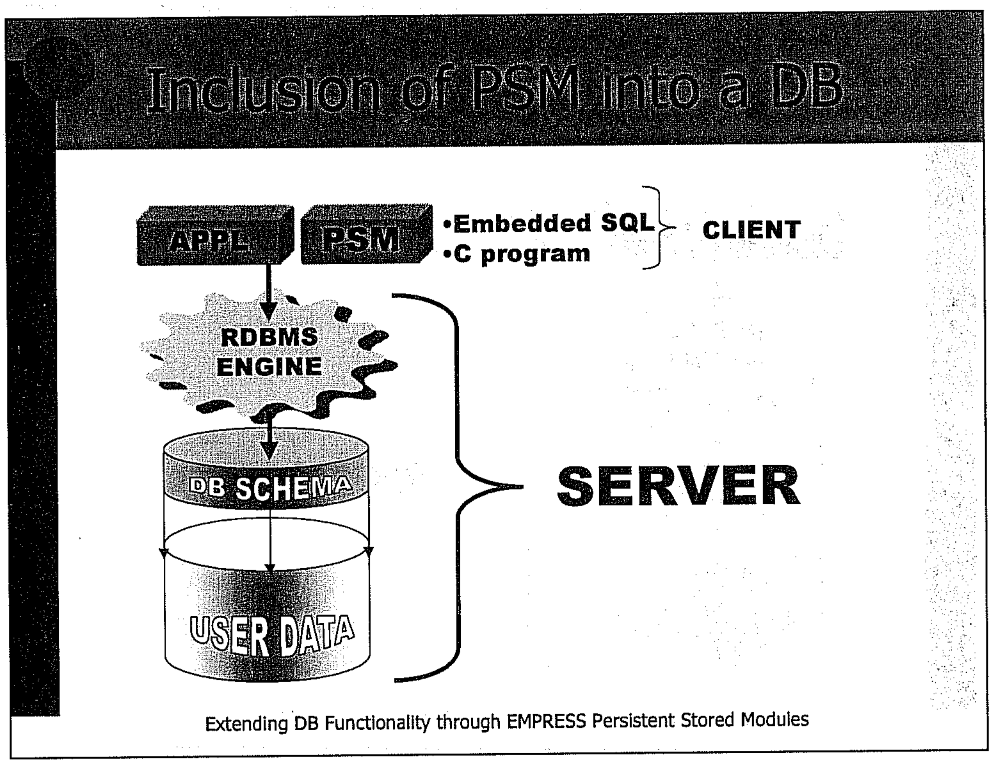
Any database driven application can be divided into four conceptual components:

a) user data set

b) database scheme

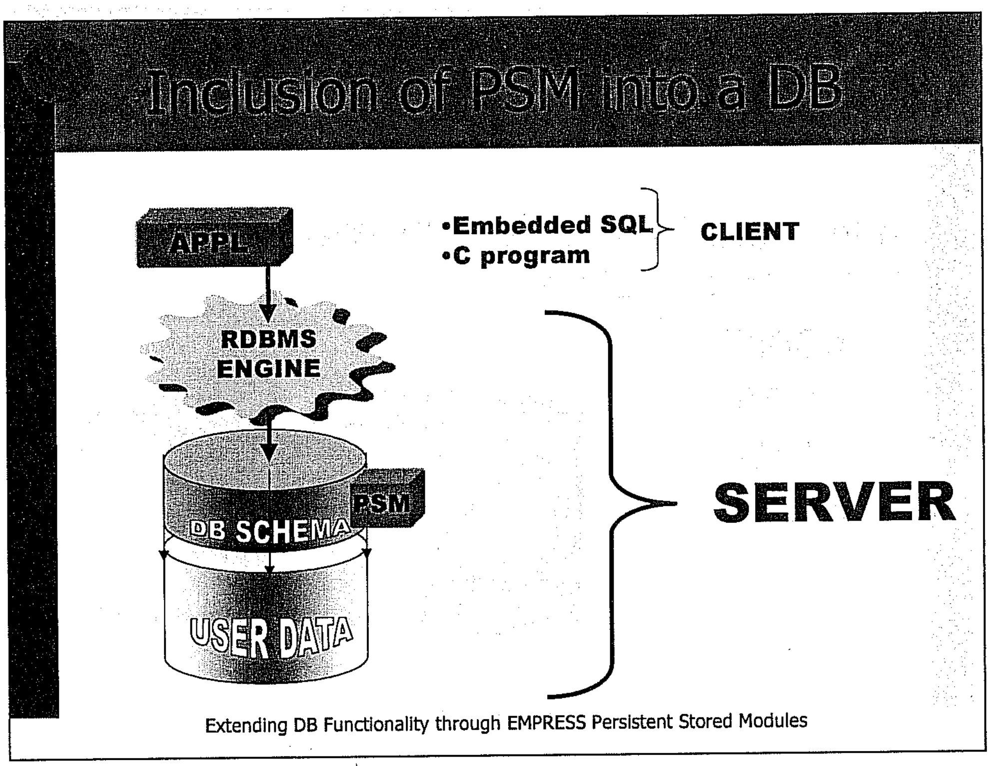
c) database engine

d) application



For the purpose of this paper the parts mentioned in a, b and c are referred to as "database server". By the same token, the application portion is informally referred to as "client". This is a software definition and it should not be mixed up with the hardware client/server paradigm, for the *database server* and *client* can reside on the same physical machine.

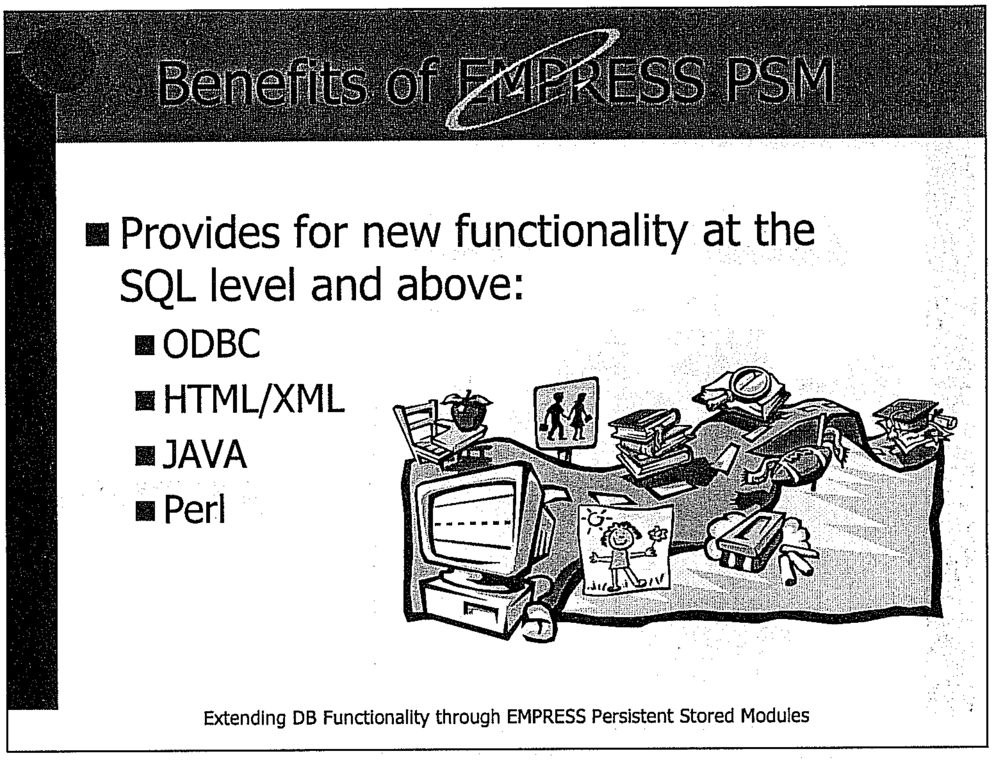
The application *client* contains procedural logic of the database-based application. A typical database application would usually contain two conceptual parts: operational logic and user interface. Theoretically, it would be possible to divide such an application along this dividing line and convert the portion with the operational logic into a PSM. Of course, a brand new PSM can be developed without re-writing an existing application.



5. THE PLACEMENT OF A PERSISTENT STORED MODULE

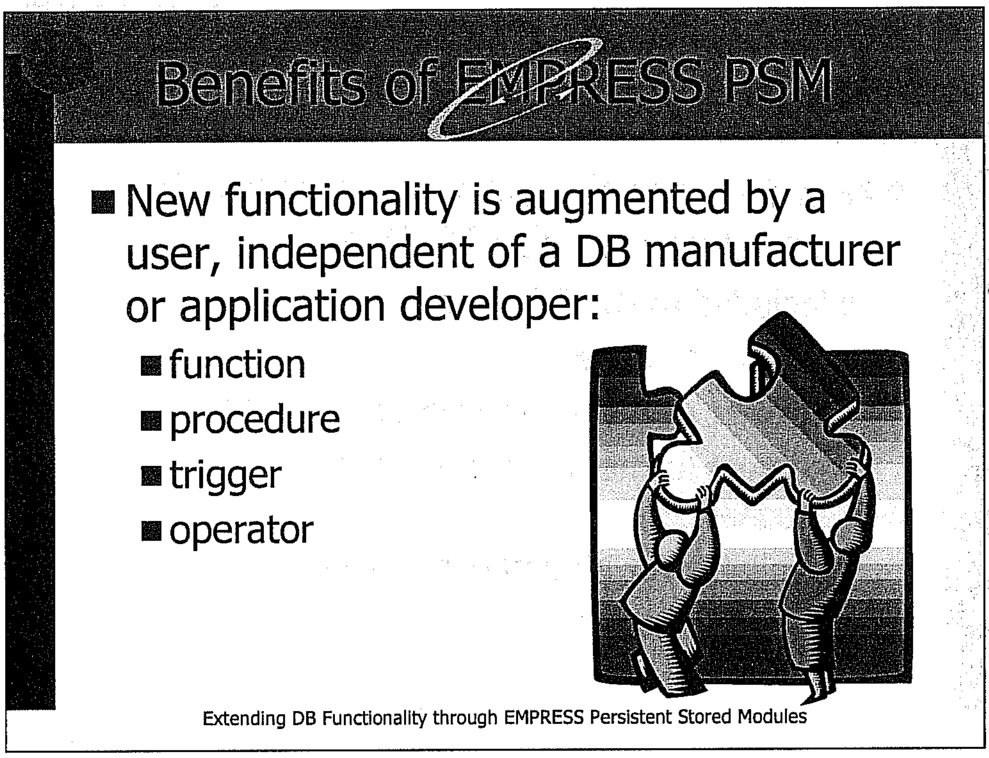
The inclusion of the operational logic into a PSM translates into the following statement:

logic that pertains to data kept in the database is stored in the database itself, below the database engine layer.



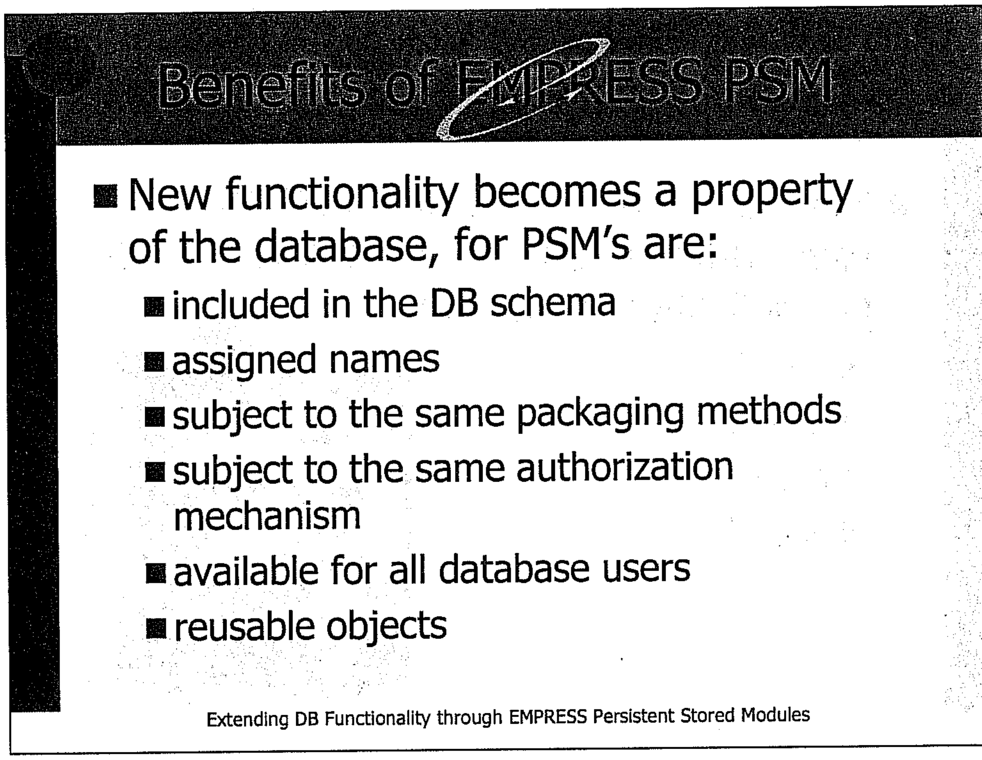
6. EXTENDING OPERATIONAL LOGIC TO OTHER USER INTERFACES

One of the major advantages of storing operational logic underneath the database engine layer is the ability of many other applications written in different languages to access this logic in the same manner as it would access the data stored in the database. In other words, the functionality embedded into a PSM now can be extended to other user interfaces



7. AUGMENTATION OF EXISTING FUNCTIONALITY

The ability to store executable logic in the database schema also allows the owner of the database to easily add new user-defined functions, procedures and operators to their database. So, if a user is in need of a certain statistical function that is not a part of the database system distribution, this function can be effortlessly made a part of this database. Thus, the user of the database system gains a certain degree of independency from the manufacturer of the database system.



8. OPERATIONAL LOGIC BECOMES A PART OF THE DATABASE

There are many good reasons for why user data and procedural logic responsible for processing that data should be stored in the same repository called database. One entity now includes the "processing unit" and "raw material". Each PSM is assigned a user-defined name, which makes it unique and distinct. Data Definition Language at the Interactive SQL level provides for management of PSM's. PSM's are also subject to the same authorization methods imposed on the entire database system. Should an existing application be updated or converted into a different end-user interface, the business logic stored in PSM's will be re-used by new applications thus providing greater fluidity in moving away from obsolete applications and/or environments to new ones.

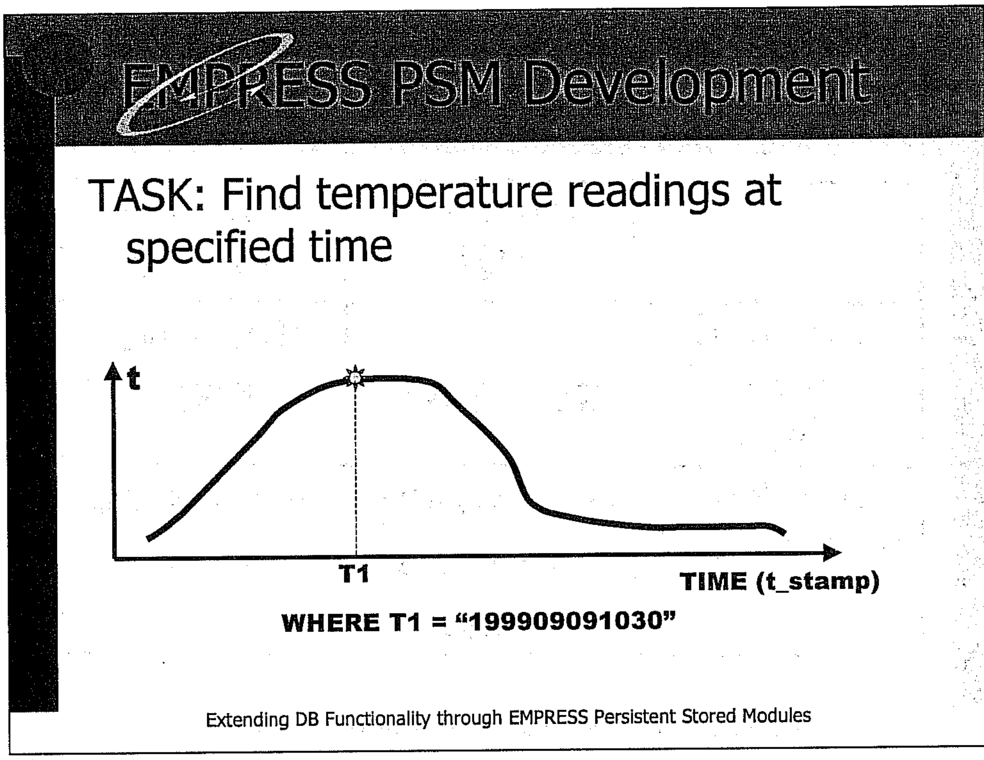
PSM Classification

	written in SQL	written in non-SQL
invoked from SQL	**SQL PSM**	**External PSM**
invoked outside SQL	**Externally Invoked PSM**	**Non-SQL PSM**

Extending DB Functionality through EMPRESS Persistent Stored Modules

9. PSM CLASSIFICATION

The ANSI standard defines four categories of PSM's. The programming language used for coding the user-defined routines and the method of invocation define the category of a particular PSM. The EMPRESS implementation of Persistent Stored Modules supports all four types.



10. EXAMPLE OF A PSM IMPLEMENTATION

Let's examine a real life example in order to better understand the applicability of a PSM and the method of its inclusion into an existing database.

Our Task is to find temperature readings (*t*) at specified time (*T1*) in a chronological log stored in database table *Thermolog*; *Thermolog* is a table in the database called *db*.

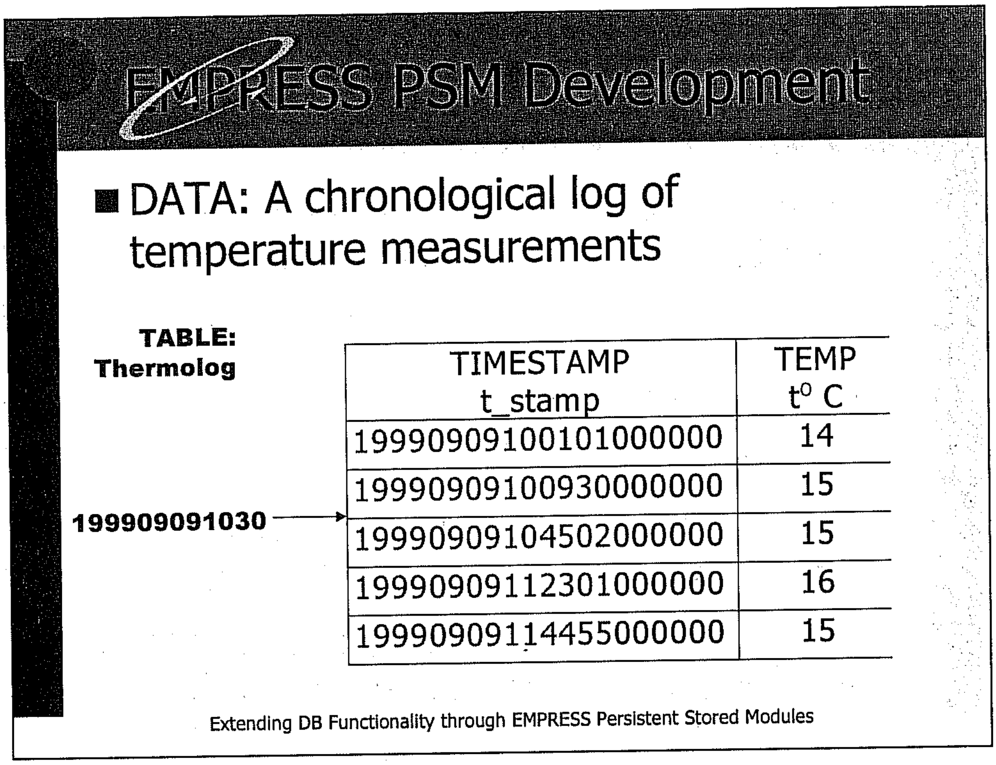


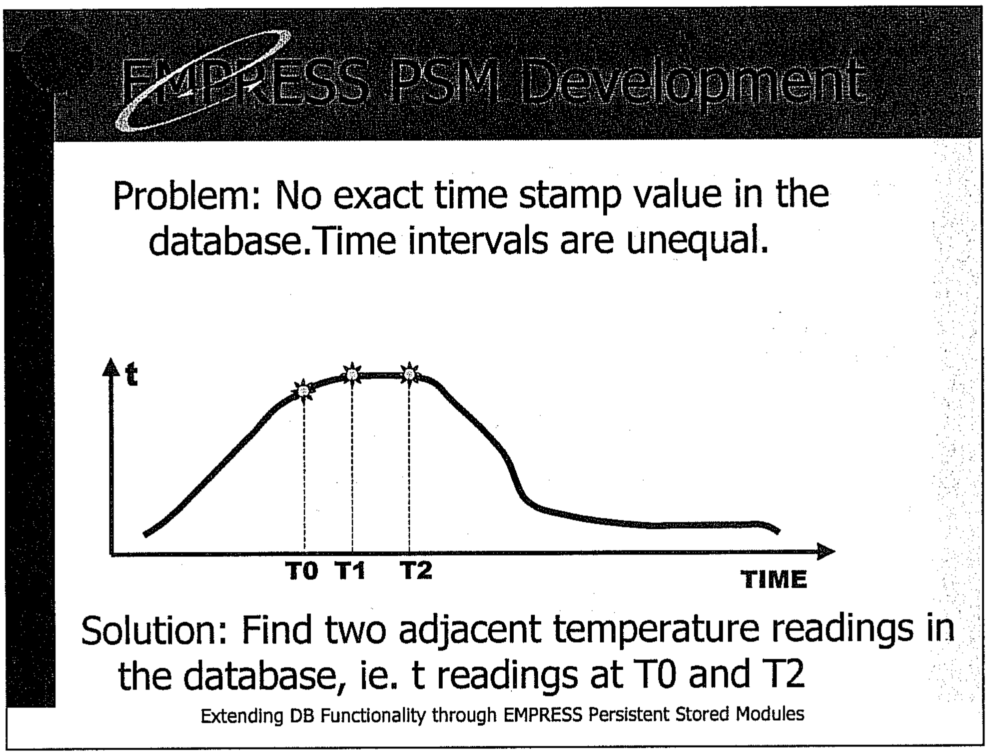
Table *Thermolog* must contain at least two attributes:

t_stamp representing timestamp readings in the log

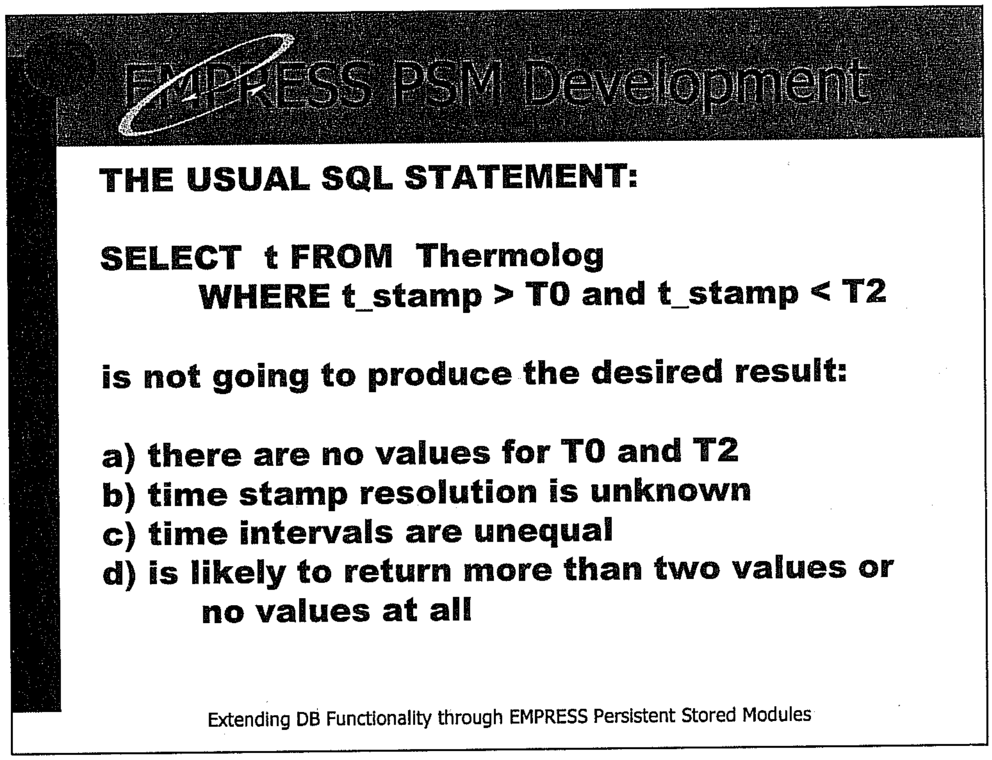
(in case of EMPRESS RDBMS the timestamp data type has the resolution of a microsecond)

and *t* representing the actual temperature measurement.

It is very likely (especially if the timestamp resolution is high) there won't be a precise match for the time value T1 in the chronological log.



There are several approaches one might take in order to solve this problem. One approach is to find temperature readings that were taken just before and just after time T1. In other words, two closest time stamps: T0 and T2 where T0 will be less than T1 and T2 will be greater than T1.



The seemingly fit SQL statement:

SELECT *t* FROM *Thermolog* WHERE *t_stamp* > T0 and *t_stamp* < T2 is

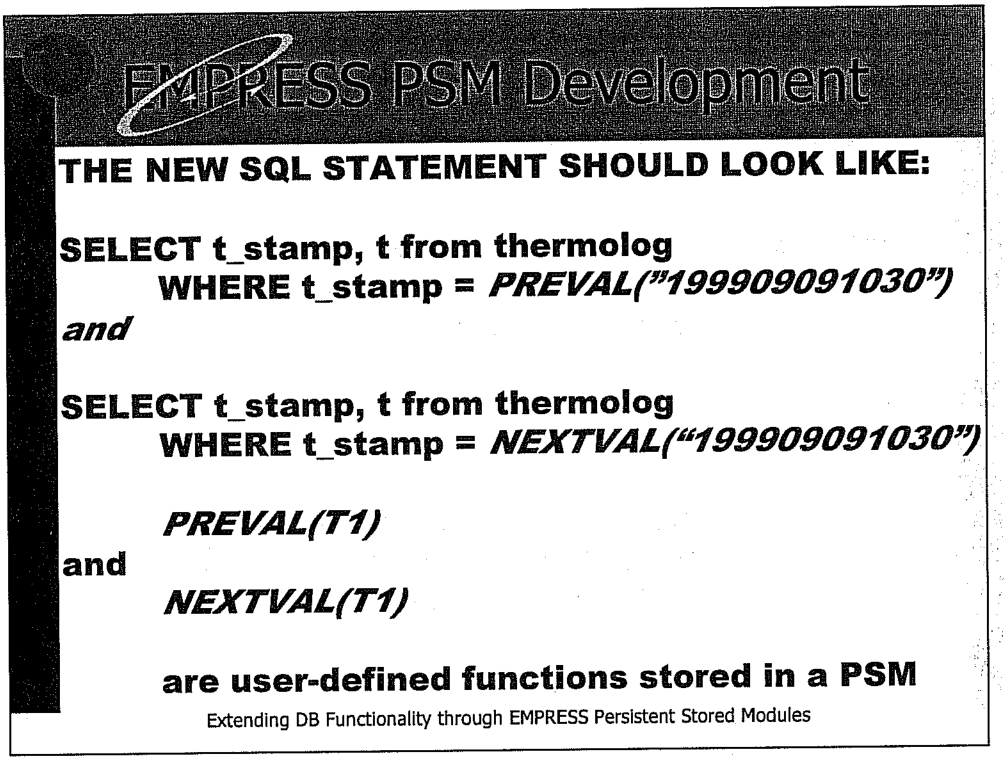
unlikely to produce the desired result for:

1) the values for T0 and T2 are not known, it is possible to find them, but it will take at least two additional SQL statements and it also requires specialized date type functions which might not be a part of the database engine;

2) timestamp resolution is unknown; it could be minutes, if entries made manually or microsecond if the data was inserted by an application;

3) in order to cover all the bases an assumption that entries were made at irregular intervals must be made;

4) even if "common sense" guesses are made as to what the values of T0 and T2 might be, the select statement is likely to return too many values or none at all.

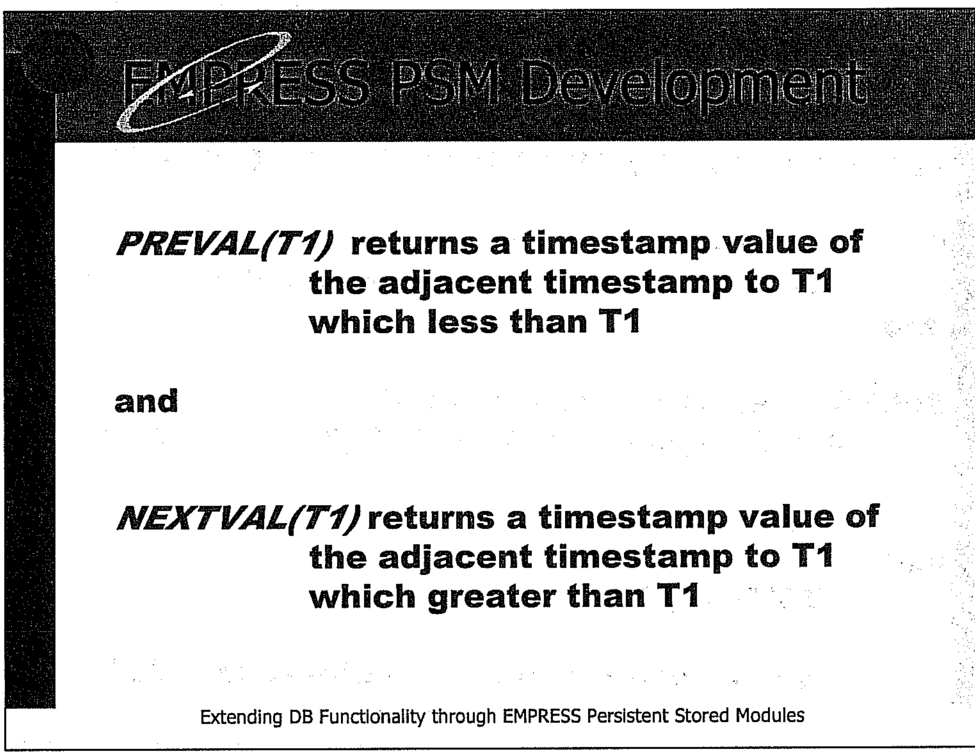


In order to obtain the expected result each and every time a SELECT statement is issued, we need to agree that each SQL statement must return just one value. Since we are looking for two temperature readings at T0 and T2, we require two SELECT statements, each returning a single temperature measurement. Therefore, the new SQL statements should be as follows:

SELECT *t_stamp, t* FROM *Thermolog* WHERE *t_stamp* = **PREVAL**("*199909091030*")

SELECT *t_stamp, t* FROM *Thermolog* WHERE *t_stamp* = **NEXTVAL**("*199909091030*")

where PREVAL(T1) and *NEXTVAL(T1)* are user-defined functions stored in a PSM

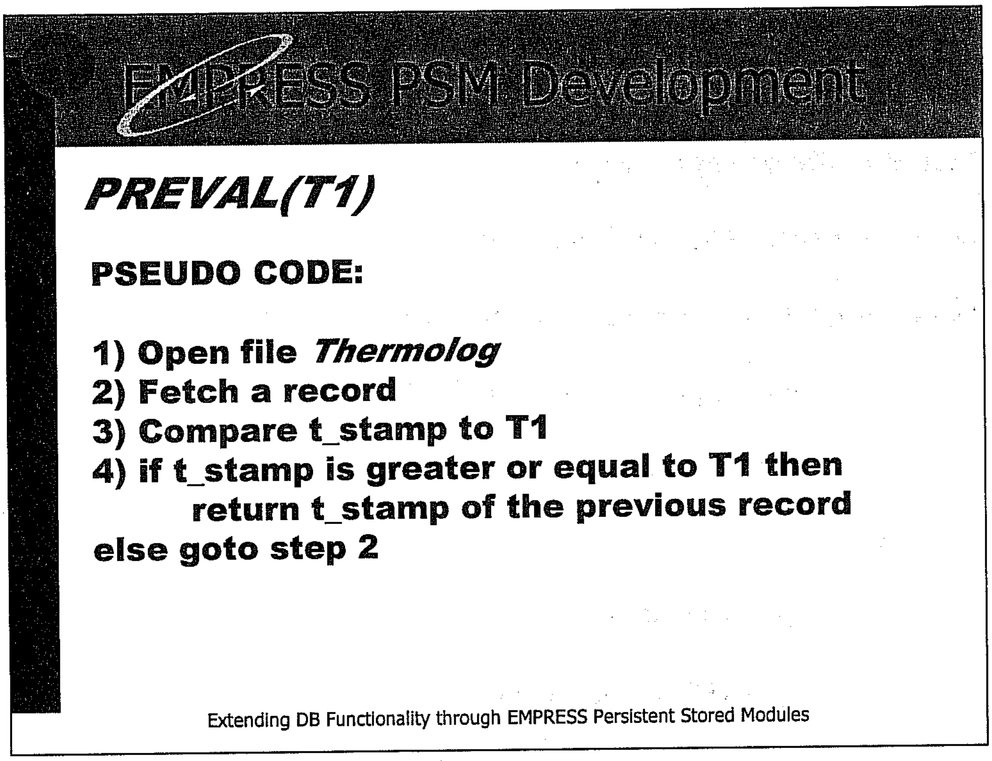


The definitions of the user-defined functions *Preval* and *Nextval*

PREVAL(T1) returns a timestamp value of the adjacent timestamp to T1,which is less or equal than T1

and

NEXTVAL(T1) returns a timestamp value of the adjacent timestamp to T1, which is greater or equal than T1



The Algorithm for the User-defined Function *Preval()*

Since both functions PREVAL() and NEXTVAL() are similar in nature, we will focus on one of them – PREVAL().

Here's the pseudo code for the function PREVAL:

(Table Thermolog is ordered in chronological order)

Open file ***Thermolog*** (point the log file)

Fetch next record (read in time stamp)

Compare t_stamp to T1 (compare time stamp to "199909091030")

if t_stamp is greater or equal to T1 then

return t_stamp of previous record

else

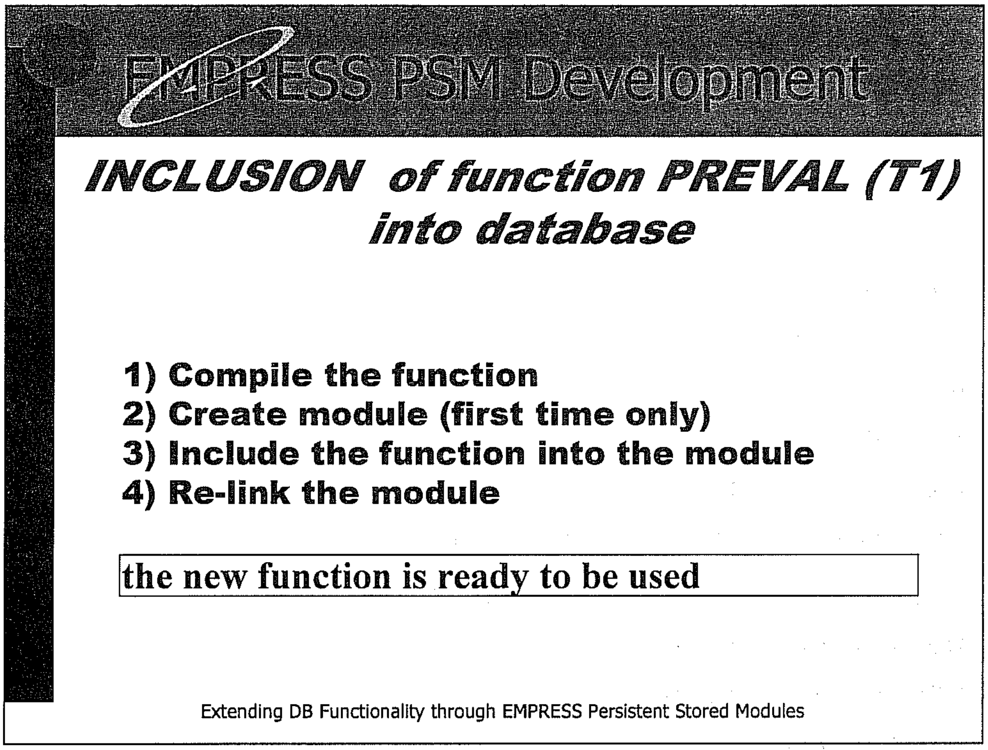
record t_stamp of this record

go to "Fetch next record"

(Continue until the condition is met or end-of-file)

Here's the source code for the user-defined function *PREVAL*. The function is written in C and Embedded SQL.

```
GLOBAL_SHARED_FUNC char  *preval (char *timestamp)
{
        char   *pretmp;
        pretmp = (char*) mspsm_malloc (32);
        pretmp[0] = '\0';
        EXEC SQL INIT;
        EXEC SQL DATABASE IS "./db";  check ("DATABASE");
        EXEC SQL DECLARE log_entry CURSOR FOR
        SELECT t_stamp FROM Thermolog order by timestamp;
        check( "declare cursor" );
        EXEC SQL OPEN log_entry;
        check( "open cursor" );
        while (SQLCODE != 100){
                EXEC SQL FETCH log_entry t_stamp INTO :str;
                if ( SQLCODE == 100 ) break; check( "fetch " );
                if( strcmp(str, t_stamp ) >= 0 )
                {
                EXEC SQL CLOSE log_entry;
                EXEC SQL CLOSE_TABLE "Thermolog";
                return pretmp;
                }
                strcpy(pretmp, str);
                }
        EXEC SQL CLOSE log_entry;
        EXEC SQL CLOSE_TABLE "Thermology";
        return pretmp;
}
```



11. THE INCLUSION OF THE USER-DEFINED FUNCTION PREVAL

Compile the function

empesql preval.c

(*prepare Embedded SQL statement, stored in *mpout.c**)

emppsmcc –inplace –O –o preval.dll mpout.c $MSPATH

(*compiles the C program mpout.c and produces Dynamically loadable executable stored in *preval.dll**)

Create module (first time only)

CREATE MODULE TIMING FUNCTION PREVAL (Generic char) RETURNS GENERIC Char EXTERNAL NAME PREVAL; END MODULE; (*Create a PSM called TIMING, which includes the user-defined function PREVAL(T1)*)

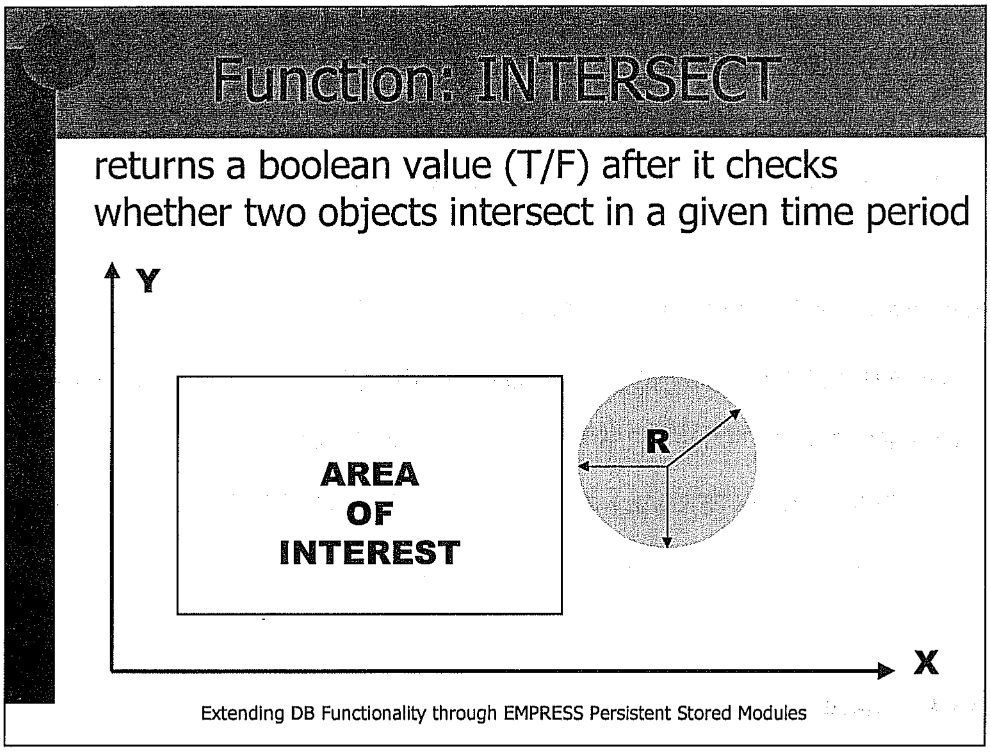
Include the function into the module

CREATE MODULE TIMING FUNCTION PREVAL (Generic char) RETURNS GENERIC Char EXTERNAL NAME PREVAL; FUNCTION NEXTVAL (Generic char) RETURNS GENERIC Char EXTERNAL NAME NEXTVAL; END MODULE; (*add the user-defined function NEXTVAL(T1) to MODULE TIMING*)

Re-link the module

UPDATE MODULE TIMING FROM "preval.dll";

Now a new PSM called *TIMING* is created and stored in the schema of the database *db*. This module becomes a schema object of the *database server*, and as such will be dynamically loaded and executed at the run time. *TIMING* includes two user-defined functions *PREVAL(T1)* and *NEXTVAL (T1)* which are now also database schema objects. Any user who has access to the table *Thermolog* also gains access to the two functions and therefore can query the table using the functions *PREVAL()* and *NEXTVAL()*. These functions can be called upon by the user working at any of the following interface levels: Interactive SQL, Embedded SQL, Microsoft Query (MS-Access, MS-EXCEL), any ODBC compatible client, as well as Perl DBI, HTML/XML, JAVA. In other words, user data stored in the table *Thermolog* and the executable code stored in the routines *PREVAL()* and *NEXTVAL()* had become parts of one entity (database *db*) and therefore are meant for "public consumption".

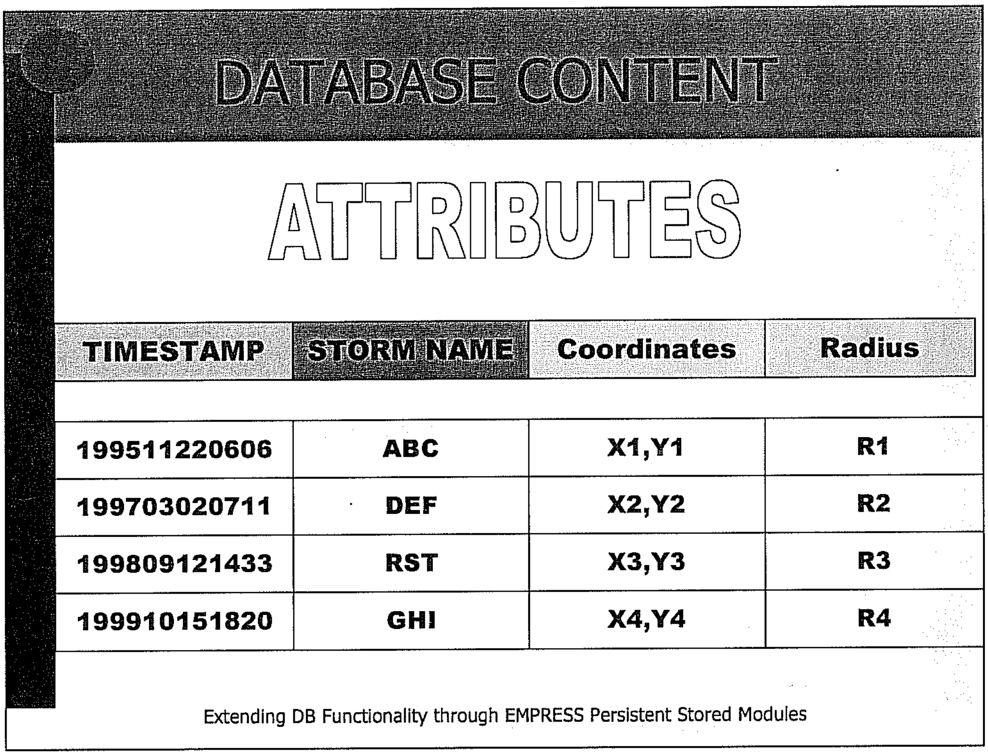


12. ANOTHER EXAMPLE OF A PSM APPLICATION

The executable logic stored in user-defined routines can be of greater complexity. Our next example includes a sophisticated algorithm, which deals with geometrical figures defined on a plane of coordinates. The user-defined function *rectangle_circle_intersect*(X1,Y1,X2,Y2, X3,Y3, R) returns a Boolean value of True or False if finds that the two figures intersect.

The rectangle is defined by two points on the plane of coordinates: the first point X1, Y1 and the second point diagonally opposed to the first is denoted by X2,Y2. The circle is defined by the centre X3,Y3 and the length of the radius R.

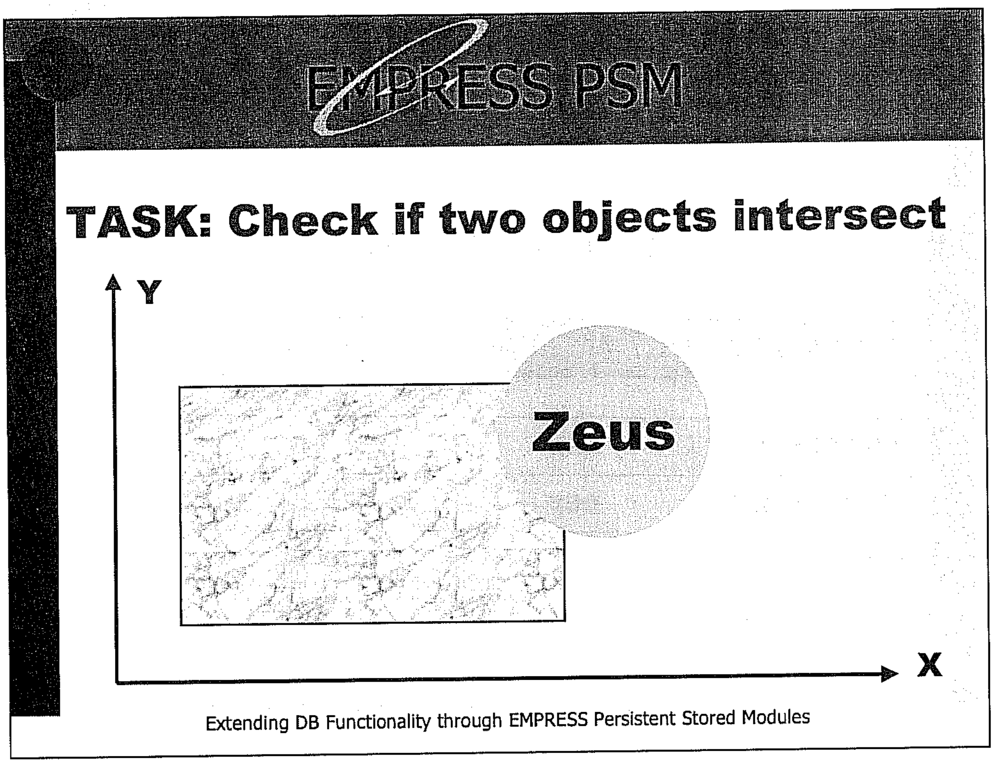
The source code for this routine can be found in the Appendix A.



TIMESTAMP	STORM NAME	Coordinates	Radius
199511220606	ABC	X1,Y1	R1
199703020711	DEF	X2,Y2	R2
199809121433	RST	X3,Y3	R3
199910151820	GHI	X4,Y4	R4

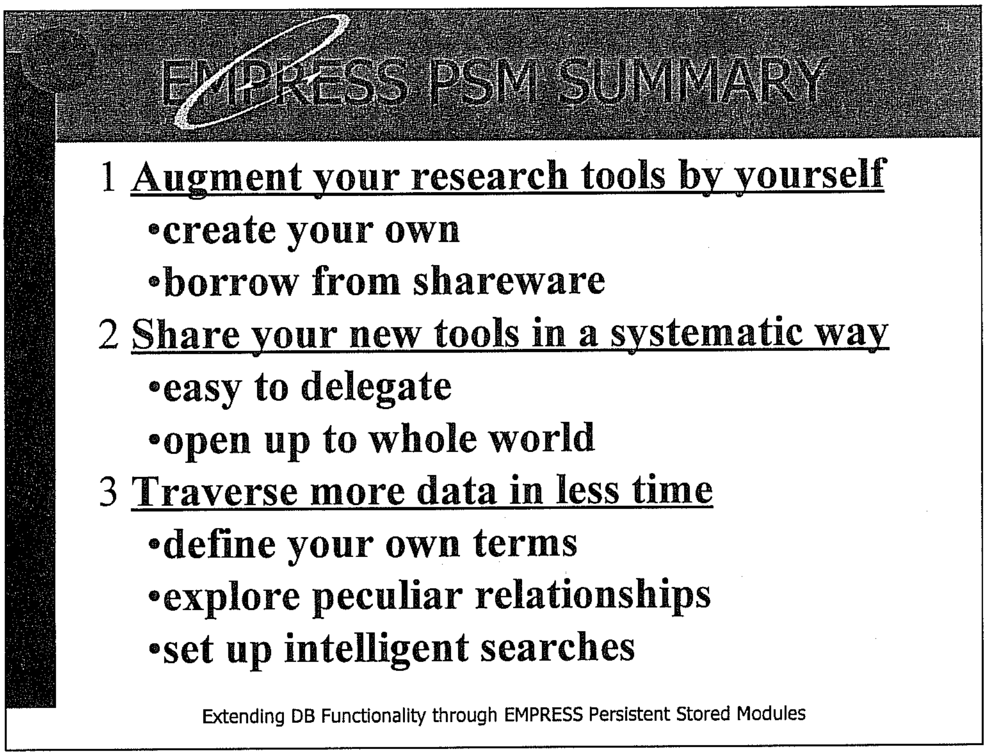
Once the function *rectangle_circle_intersect*(X1,Y1,X2,Y2, X3,Y3, R) is compiled, it becomes possible to perform a spatial operation "intersect" over a set of data with physical dimensions and addressable coordinates.

Let's take a look at the table of storms where the attributes include: timestamp, the storm name, coordinates and the dimension of the storm.



The rectangle can be defined by a user as the area of special interest, then the table Storm can be traversed using the function *rectangle_circle_intersect()* in order to identify the storm that had touched at the area of interest.

It is obvious how this or similar functions can be added to an existing database containing current or historical data and achieve rudimentary functionality of a Geographical Information System without acquiring one.



THE SUMMARY

In conclusion,

The main benefits of the PSM technology are:

The faculty to augment the existing functionality of a database engine through the inclusion of executable logic as database schema objects is extended to the database user level.

The new functionality is stored using the same methods as data, therefore equating the simplicity in accessing data kept in the database to that of using the new functionality.

The combination the new functionality and the ease of use allow for the construction of intelligent database mining while searching for "unusual" relationships.

The PSM implementation by Empress Software empowers users to step above the strict constrains of a relational database and take steps in the direction of the object-oriented technology.

Reference:

Date, C.J. with Hugh Darwen, 1997: A Guide To The SQL Standard.
Fourth Edition. Addison-Wesley, Don Mill, Ontario Appendix E.

Fortier, Dr. Paul J., 1999: SQL-3 Implementing the Object-Relational Database.
McGraw-Hill, Toronto pp. 305-322

Raoult, Baudouin, 1997: Architecture of the new MARS server.
Sixth Workshop on Meteorological Operational Systems, 17-21 November, 1997,
ECMWF, Reading, UK, p.90

Silberschatz, Avi, Stonebreaker, Mike, Ullman, Jeff, editors, 1996: Database Research:
Achievements and Opportunities Into the 21st Century.
SIGMOD Record, Vol. 25, No.1, March 1996.
The Association of Computing Machinery, New York, NY, USA pp. 52-58

APPENDIX A.
USER-DEFINED FUNCTION *rectangle_circle_intersect*
(The function name <u>box_circle_intersection</u> is used in this example)

```
#include        <usrfns.h>
#include        <stdlib.h>
#include        <stdio.h>
#include        <math.h>

#define EARTH_CIRCUMFERENCE   40075.0
typedef struct {
        double  x;
        double  y;
} Point;

typedef struct {
        Point   centre;
        double  radius;
} Circle;

/* This function will return the key that fits where a box that is describe 4 points. A point is
intersection of longitude and a latitude.
        * (long1, lat1) ------------------------- * (long2, lat1)
        |                                         |
        |                                         |
        |                                         |
        |                                         |
        * (long1, lat2) ------------------------- * (long2, lat2)*/

static  msbool Check_point (
                                double  boundary_longitude,
                                double  boundary_latitude,
                                double  current_longitude,
                                double  current_latitude,
                                msbool  west);
static  msbool west = true;
static  msbool east = false;

/*
* Imports:
*       par1:   longitude of current record
*       par2:   latitude of current record
*       par3:   longitude of Northwest Corner
*       par4:   latitude of Northwest Corner
*       par5:   longitude of Southeast Corner
*       par6:   latitude of Southeast Corner
*
* Exports:
*       returns true if within region
*
*/
```

```
GLOBAL_SHARED_FUNC    msbool box_point_intersection (
                            double longitude,
                            double latitude,
                            double long1,
                            double lat1,
                            double long2,
                            double lat2)
{
        double x_offset;
        double y_offset;

        if (long1 < 0.0)
                y_offset = fabs (long1);
        if (lat1 < 0.0)
                x_offset = fabs (lat1);

        /* Check the north west corner */
        if (!Check_point (
                long1 + y_offset,
                lat1 + x_offset,
                longitude + y_offset,
                latitude + x_offset,
                west))
                        return false;

        y_offset = x_offset = 0.0;
        if (long2 < 0.0)
                y_offset = fabs (long2);
        if (lat2 < 0.0)
                x_offset = fabs (lat2);
        if (! Check_point (
                long2 + y_offset,
                lat2 + x_offset,
                longitude + y_offset,
                latitude + x_offset,
                east))
                        return false;
        return true;

}

static    msbool Check_point (
                                double boundary_longitude,
                                double boundary_latitude,
                                double current_longitude,
                                double current_latitude,
                                msbool west)
{
        /* Check the East longitude line */
        if (!west)
        {
```

```
			if (current_longitude > boundary_longitude)
			{
					return false;
			}
			/* Check the latitude line */
			if (current_latitude > boundary_latitude)
			{
					return false;
			}
	}
	else
	{
			/* Check the West longitude line */
			if (current_longitude < boundary_longitude)
			{
					return false;
			}

			/* Check the South latitude line */
			if (current_latitude < boundary_latitude)
			{
					return false;
			}
	}

	return true;
}

static	msbool intersection (
				Point p1,
				Point p2,
				Circle c)
{

	double dx;
	double dy;
	double dr2;
	double determinant;
	double discriminant;
	Point i1;
	Point i2;			/* intersection points */
	double intermediate;	/* intermediate step in equation */
	/* distances for the 3 colinear points */
	double d1;
	double d2;
	double d3;
	msbool ret;

	p1.x -= c.centre.x;		/* move system so circle is on origin */
	p2.x -= c.centre.x;
	p1.y -= c.centre.y;
```

```
p2.y -= c.centre.y;

/* step one, If the discriminant is >0 then there are intersections */
dx = p2.x - p1.x;
dy = p2.y - p1.y;
dr2 = dx*dx + dy*dy;

determinant = p1.x*p2.y - p2.x*p1.y;

discriminant = ((c.radius * c.radius) * dr2) -
               (determinant * determinant);

/* two point intersection of line ... We have an intersection */
ret = false;
if (discriminant >= 0.0)
{
        /* step two, find the intersection points */
        intermediate = dx * sqrt(discriminant);

        /* significant of dy */
        if(dy<0)
        {
                intermediate *= -1.0;
        }
        if (dr2 > 0.0)
        {
                i1.x=((determinant * dy) + intermediate) / dr2;
                i2.x=((determinant * dy) - intermediate) / dr2;
        }
        else
        {
                /* False value in, false value out....
                  This happens only if the line segment is zero
                  length, so it is going to return false.*/
                return false;
        }

        if(dy<0)
        {
                /* absolute of dy */
                dy*=-1.0;
        }
        intermediate = dy * sqrt(discriminant);
        i1.y = ((-1.0 * determinant) * dx + intermediate) / dr2;
        i2.y = ((-1.0 * determinant) * dx - intermediate) / dr2;


        /* step 3, check if one of the points is between p1 and p2 */
        /* d1 = length of line segment */
        /* d2 = distance from p1 to intersection point */
        /* d3 = distance from p2 to intersection point */
```

```
                /* check both intersection points. here's intersection 1 */
                d1 = sqrt(((p1.x-p2.x)*(p1.x-p2.x))+
                            ((p1.y-p2.y)*(p1.y-p2.y)));
                d2 = sqrt(((p1.x-i1.x)*(p1.x-i1.x))+
                            ((p1.y-i1.y)*(p1.y-i1.y)));
                d3 = sqrt(((p2.x-i1.x)*(p2.x-i1.x))
                            +((p2.y-i1.y)*(p2.y-i1.y)));
                if(d2 + d3 - d1<0.0001) /* some tolerance for rounding errors */
                        ret=true;

                /* and intersection 2 */
                d1=sqrt(((p1.x-p2.x)*(p1.x-p2.x))
                            +((p1.y-p2.y)*(p1.y-p2.y)));
                d2=sqrt(((p1.x-i2.x)*(p1.x-i2.x))
                            +((p1.y-i2.y)*(p1.y-i2.y)));
                d3=sqrt(((p2.x-i2.x)*(p2.x-i2.x))
                            +((p2.y-i2.y)*(p2.y-i2.y)));
                if(d2+d3-d1<0.0001) /* some tolerance for rounding errors */
                        ret=true;
        }

        return(ret);
}

/* This function will return true is the storm intersects the square
        a box that is describe 4 points. A point is intersection of
        longitude and a latitude.
        * (long1, lat1) ------------------------- * (long2, lat1)
        |                                         |
        |                                         |
        |                                         |
        |                                         |
        * (long1, lat2) ------------------------- * (long2, lat2)
*/

/*
* Imports:
*       par1:   longitude of current record
*       par2:   latitude of current record
*       par3:   radius of storm system
*       par4:   longitude of Northwest Corner
*       par5:   latitude of Northwest Corner
*       par6:   longitude of Southeast Corner
*       par7:   latitude of Southeast Corner
*
* Exports:
*       returns true if within region
*
*/

GLOBAL_SHARED_FUNC      msbool box_circle_intersection (
```

```
                    double  longitude,
                    double  latitude,
                    double  radius,
                    double  long1,
                    double  lat1,
                    double  long2,
                    double  lat2)
{
        /* Calculate if the distance */
        Circle   Storm;
        Point    Northwest;
        Point    Southwest;
        Point    Northeast;
        Point    Southeast;

        /* Is the storm point within the box */
        if (box_point_intersection (longitude, latitude, long1, lat1,long2, lat2))
                return true;

        Storm.radius = radius * (360.0/EARTH_CIRCUMFERENCE);
        Storm.centre.x = latitude;
        Storm.centre.y = longitude;

        /* We make the assumption that we are working only near the
          equator and that the earth is flat.
          Therefore we won't have to get into any heavy duty
          scaling function for distances that a degree represents.  */

        Northwest.y = long1;
        Northwest.x = lat1;
        Southwest.y = long1;
        Southwest.x = lat2;
        if (intersection (Northwest, Southwest, Storm))
                return true;

        Northeast.y = long2;
        Northeast.x = lat1;
        Southeast.y = long2;
        Southeast.x = lat2;
        if (intersection (Northeast, Southeast, Storm))
                return true;

        /* It needs to check the latitude intersections */
        if (intersection (Northwest, Northeast, Storm))
                return true;

        if (intersection (Southwest, Southeast, Storm))
                return true;

        return false;
}
```