Chapter 1

Database Access from Phat

1.1 Keys, KeyPtr's and the Key Manager

This style of database access is intended to have been created by experts for use by various users, and the interface therefore has little or no direct reference to the actual database. All reference to objects residing in the database is by means of a key, an object deriving from TPhDbKey, which encapsulates the SQL concept of a primary key to a table – it is an object which refers to exactly one 'object' somehow stored in Oracle.[1]

1.1.1 Creating TPhDbKeys

There are two ways to obtain a key. One is directly, through a class's constructor. See for example the constructor of TPhCalKey or TPhRunKey. This requires you to know, exactly, all the information that the Oracle tables are keyed on. If you need for some reason a KeyPtr and therefore a key to an object which is not in the database, you can use a TPhLocalKey. See the class documentation of that class for more information.

The second way is to use a key class's Select function. This function is designed to return "the best" key[2] satisfying some specification. See for example TPhCalKey::Select. This would be used in the following way:

```
TPhCalKey::Select("SIPED",2121,"PHAT5");
```

where this will return you the "best" (most recent) calibration key which is a silicon pedestal set, for method PHAT5 (the currently used one) and also valid for run 2121.

[1] Though that object need not be actually stored in one uniquely referenced row of a single table.

[2] It actually returns a lazy-evaluation object called a TPhDbKeySelector, but usually you do not need to worry about this distinction.

1.1.2 TPhKeyPtr's

The TPhKeyPtr is a template class. Objects of the class function for all intents and purposes (to the user) just like a pointer. One specifies what the TPhKeyPtr is a pointer to by the template parameter, thus:

```
TPhKeyPtr<TPhRun> run;
```

declares 'run' as a key pointer to a TPhRun object. If one has a pointer to a valid object (which the above declaration does not do, more below), it could refer to the actual run object thus:

```
run->GetDate();  // Calls the TPhRun GetDate() method.
func(*run); // Calls func, passing it the run object
            // (or a reference to it).
```

Many functions in Phat and ROOT require a raw pointer. Most of the time, the TPhKeyPtr can be passed just like a pointer to a const object. Unfortunately, many ROOT and Phat functions are not declared const-correctly, and therefore a cast to a pointer to a non-const is needed.

```
const_cast<TPhRun *>(run.GetPtr()) // can be passed to a function
                                   // taking TObject *
```

Note well that using this const_cast to actually modify the object pointed to by the KeyPtr may have unpredictable results ('undefined behavior', as the C++ standard calls it). The need for this should be greatly diminished with the coming of ROOT 3.0.

1.1.3 Reading TPhKeyPtr's

The first way of making a TPhKeyPtr is to read in an object which already exists in the database. To do this, you just ask for that object by its key:

```
TPhRunKey runkey(1543);
TPhKeyPtr<TPhRun> run(runkey);
```

would make run a pointer to a TPhRun object for run 1543. One can condense this syntax and use temporaries:

```
TPhKeyPtr<TPhRun> run(TPhRunKey(1543));
TPhKeyPtr<TPhPedestalSet>
  peds(TPhCalKey::Select("SIPED",2121,"PHAT5");
```

where the first statement does the same as in the previous example, and the second statement creates a pointer to a pedestal object referred to by the key discussed in the example in the previous section – the "best" set for run 2121.

There are two advantages to using KeyPtr's in this way. First, when you stream an object out to a Root file, any KeyPtr's in the object will write only the key – not the whole object pointed to. When the object is read back in, the key manager will consult the database and recover the object by its stored key, and reattach the pointer. Second, the key manager performs caching of objects, so that objects will generally not be read twice from the database.

1.1.4 Writing TPhKeyPtr's

The second way of making a TPhKeyPtr is to write out an object which is not yet in the database to the database, defining its key in the process. The way this is expressed is by making a pairing of a Key and its object – which does an implicit write to the database.[3] For example:

```
TPhPedestalSet *mypeds;
// I've just calculated mypeds for run 2119 somewhere else.
TPhKeyPtr<TPhPedestalSet> peds(TPhCalKey("SIPED",2119,"PHAT5"),mypeds);
```

This is very similar to the previous, except for the explicit specifying of an actual key, not asking for the "best" key – note that the previous Select function could well have returned this pedestal set, if it had been in the database and no other sets were in for runs 2120 or 2121 – and the explicit specifying of the object pointed to, in this case mypeds. It is the fact that these are explicitly specified that makes the system write the new association into the database.

1.1.5 Caveats to using KeyPtr's

Once you create a KeyPtr referring to an object, the TPhKeyManager thinks it 'owns' the object, and might delete it out from under you (if you've discarded your KeyPtr), or might give someone else a pointer to that object without telling you. Here are some rules to follow when working with KeyPtrs:

- Don't make KeyPtr's to objects which are not now or soon to be stored in the database.
  The current implementation forbids this, but that may change in the future.

- Don't change an object pointed to by a KeyPtr unless you know what you must do.
  The current implementation also forbids this, but this may change since there are too many ROOT functions which are not const-correct. If it is changed, changing the object a KeyPtr points would cause others hard-to-find bugs.

- Don't take pointers to the object pointed to by the keyptr without keeping the KeyPtr itself around. Remember that the KeyPtr is destroyed when you leave scopes!
  If you do this, it is possible that the Key Manager will think you're done

[3] This may be made more explicit in the future.

with the object and delete it. Or it might not, if someone else has a KeyPtr to it somewhere. This is another source of hard-to-find bugs.

1.2 Using SQL directly

One can use SQL directly, if one understands the structure of the database itself (discussion of which is beyond the scope of this document). One uses the interface supplied by three abstract base classes to SQL: *TPhSQLDatabase, TPhSQLConnection,* and *TPhSQLStatement.*

In order to execute a statement, one must follow several steps. First, one must initialize the database (once per Phat session). The normal way of doing this is:

```
gPhDb = new TPhDatabase();
```

TPhDatabase uses TPhSQLDatabase, and is aware that the (current) Phobos database uses Oracle.

Second, one must authenticate yourself to this database. You must do this whenever you want to change the user you're logging in to Oracle as. To do this:

```
gPhDb->Authenticate("userid","password");
```

Third, one gets a connection. You should only keep this connection as long as you need to, since otherwise multiple calls to the database can interfere with each other. The usual way of doing this is:

```
TPhDatabase::TConn activate(gPhDb);
TPhSQLConnection *conn = gPhDb.Connection();
```

The first statement creates a local object that ensures that the connection is released when you leave the scope of the function you are in. It only needs to be done once in a function, even if you refer to the connection more than once in a function. If for some reason you do not want the connection to be released upon leaving the function, you must explicitly use the functions TPhDatabase::Activate and ::Release – and be careful about early returns from your function!

Fourth, one creates a statement. This is obtained from the connection:

```
TPhSQLStatement *s =
  conn->NewStatement("SELECT X FROM TABLE");
```

Note that the statement must be deleted by the user.

Next, one binds and defines input and output variables (see below), and then executes the statement. The default execution tells ORACLE to execute the statement only once, and returns true if the execution succeeds.

```
Bool_t ret = s->Execute();
```

If the statement being executed is a select statement, the default execution will return one row of the selected output into the output variables, but will return kFALSE if there is no data. To fetch later rows, the Fetch() method should be used. Fetch retuns the number of rows fetched, or -1 if there is another error.

```
if (s->Execute()) {
  Int_t rows = 0;
  do {
    // Process data...
  } while ((rows = s->Fetch()) > 0);
  if (rows < 0) {
    // Handle error
  } else {
    // End of data.
  }
} else {
  // Can't distinguish 'real error' from 'no data returned'.
  }
```

Alternatively, in order to distinguish these errors one can tell Execute not to fetch data for a select statement, by passing a 0 parameter. Then Execute will only fail if there is an error; if there is no data, the first call to fetch will return 0.

```
if (s->Execute(0)) {
  Int_t rows = 0;
  while ((rows = s->Fetch())) > 0) {
    // Process data
    }
  if (rows < 0) {
    // Handle error fetching data.
  } else {
    // End of data, or never was any
  }
} else {
  // Handle error in executing statement.
}
```

1.2.1 Binding Variables

Binding, in ORACLE-speak, is setting a variable as an input parameter to an SQL query. While one *can* write such variables directly into the query, using strstreams, fprintf, or similar mechanisms, it is more efficient (particularly for repeated invocations of the same query with different parameters) to use these locally named variables. To use this, first one puts a placeholder, which starts with the ':' character, in the SQL statement where you want the variable to be used:

```
TPhSQLStatement *s =
  conn->NewStatement("SELECT X FROM RUNTABLE WHERE RUN = :inrun");
```

Here ":inrun" is a placeholder for a run number that we will pass to this select statement. Some C++ variable needs to exist corresponding to this SQL placeholder. Then, you tell ORACLE where it is by passing its address:

```
Int_t run_number = 7;
s->BindInt(":inrun",&run_number);
```

Now, calling the 'Execute' method will select the desired data for run number 7.

1.2.2 Defining Variables

Defining is setting up variables for ORACLE to put the results of a select statement into. This is the only way to get information out of ORACLE directly.

1.3 Adding a new key scheme to Phat

Before you decide to add a new key scheme, check to see that one of the existing ones cannot be used. Eventually, inevitably, someone will want to add a new piece of information to be stored in the database under a key. The system has been designed to allow this extension using C++ polymorphism and virtual functions. However, in order for the system to work right, the user must take care. The would-be extender is advised to read carefully the documentation herein and in the TPhDbKey class before beginning. In some cases, failure to override functions correctly may lead to improper behavior for the entire system.

In reading the following sections, one is advised to have the header file T-PhDbKey.h handy. This declares most of the functions discussed, as well as the TPhDbKeySelector.

1.3.1 Creating the table(s)

Before the key can be added, the schema on the database side must be considered and set up. Setting up a schema in SQL is beyond the scope of this document. What is important to the Phat side of things in that job, however, is making sure that there is a single set of information (the 'key') which can be used to refer to a complete set of information; in Phat this will be the object pointed to by the KeyPtr. In Oracle, this may be represented by a table with a single primary key and bits of subsidiary information; or it may be referred to by multiple rows (even in multiple tables) associated through a common key field.

1.3.2 The DbKeyManager under the hood

Before we go into detail about creating the key, it is useful to know something about how the key manager works, and where it will be calling virtual functions on the TPhDbKey objects which you will be defining.

First, the key manager keeps an internal hash table of the keys which are outstanding. These are reference-counted through the TPhKeyPtr's, which increment and decrement the references in their destructors. When someone creates a new KeyPtr, the key manager uses the ROOT hash table it has as a member to look up the key and see if it already has it. If it does, it simply increments the reference counts and returns the key with the object. For this to work correctly, it calls the DbKey's Hash() and IsEqual() functions, and note well that the default root comparisons for these functions will NOT work (they compare addresses, not values of the objects).

If the key manager does NOT have the key, it relies on one further layer of indirection to retrieve the appropriate object. It asks its 'implementation' object to fetch the object for the appropriate key. There are currently two implementation classes, though in principle more could be made. They use a virtual function to make the decision whether to call the GetFromSQLConnection or GetFromFile function on the TPhDbKey. If it calls GetFromFile, that in turn calls StringKey() to get the key string that it will look the object up by in the ROOT file, and reads the object using the root functionality.

Finally, a word about selectors. If the key manager is passed a selector, it will immediately pass the selector to its implementation; no caching is done for this. Then the implementation will call one of the two virtual functions on the selector in order to find the correct key from the SQL connection or from the list of those in the current file.

1.3.3 Defining the Key object

The key object should derive from TPhDbKey. There are several important virtual functions which need to be overridden for TPhDbKey, because these objects in general need to be compared with other TPhDbKeys by the KeyManager to determine if they are the same. Sometimes you can derive from an object derived from TPhDbKey (such as TPhTimeKey, for example) to help you out. In cases like that, many of the functions only need to be overridden if new data members are added in your class.

In this definition, please note the distinction between signifying parameters (those which actually define the key) and side-band data. If all of the signifying parameters are set, they completely define the key; the side-band data is just additional information in the database for that key that can be retrieve. In the DbKey object, such parameters should be defined as 'mutable' (see your C++ reference). As an example, the run number is a piece of side band data for the TPhCalKey, which uses instead the time as (one of) its actual signifying parameter.

The relevant virtual functions are:

- Bool_t IsEqual(TObject *obj) Must return true if the two objects compare equal, and must deal with being able to compare the object to any other TPhDbKey object as the TObject parameter. Note that ROOT compares just the addresses by default, and this is in general not desired behavior for the key structures. In general this will compare for equality all of the parameters of the key which are significant parameters.

- ULong_t Hash() This function must be defined in such a way that if two objects compare equal, they return the same hash (the converse need not be true, but the more unique you can make the hash, the better).

- TString StringKey() const This function must return a string which fully specifies the type of the DbKey, and encodes all of the signifying parameters of the DbKey.

- TObject * GetFromSQLConnection(TPhSQLConnection &)
  void WriteToSQLConnection(TPhSQLConnection &, const TObject &)
  These functions will read and write objects in and out of the database according to the key.

- void SetSideBandData(const TPhDbKey &) const This function must set any sideband data in the current key to equal that in the key passed as the parameter. See 1.4.4 for a discussion of why this is needed.

In addition, it is likely that one or more selector objects will be desired for the key. In this context, we can no longer delay discussion of the TPhDbKeySelector. A TPhDbKeySelector object (by which I mean anything deriving from the abstract class TPhDbKeySelector) is a delayed-evaluation object, which is designed to go to the database on request (usually by the DbKeyManager) and retrieve the information needed to build the 'best' key matching the parameters the selector was given (in its constructor). They should be returned by value from the TPhDbKey's Select function (see TPhCalKey for some examples), which will usually be a trivial wrapper of the constructor of the TPhDbKeySelector subclass.

For the TPhDbKeySelector, one will normally have to override two functions:

- TPhDbKey *GetKeyFromSQLConnection(TPhSQLConnection &)

- TPhDbKey *GetKeyFromList(const TCollection &)

In both cases, the key selector is responsible for consulting its parameters and finding the 'best' key either stored in the database or in the TCollection it is passed, and returning that key. [4]

[4] Aside: Since getting a list of all keys of a particular type is often useful in this, the template FilteredList is defined in DbKeyHelper.h to assist you in doing this. If ROOT allowed a filtering iterator type (you can build one in the STL), it would be used here instead.

1.4 A Detailed example – Adding TPhTrigMap and TPhTrigKey

As an example of adding a new key structure to Phat, I will discuss the design of a pair of keys which I have come close to (but have not finished) adding to Phat myself. This pair of keys is designed to describe the trigger conditions for a given run.

For purposes of the example, I will first describe the functionality desired, and then describe how the keys, their constructors, overridden virtual member functions, and corresponding selectors work. What I will *not* do here is discuss how the actual information should be stored and retrieved in the objects in Phat. In other words, I will not write the SQL statements and various Phat wrappers around those SQL statements, and the logic to store the results in the objects.

1.4.1 Conceptual elements

Going into the overall framework are several concepts, independent of the implementation. Our goal in the design is to independently map these concepts into structures both the the relational database (ORACLE) and in Phat; and then design the keys that allow you translate between these representations.

- Trigger Bit: A trigger bit is the hardware-based index of the element that is being used by the trigger circuitry to decide whether to take a bit. A given trigger bit may be set up with different meanings in hardware at different times.
- Trigger Map: The trigger map supplies a mapping between a human-defined name (eg, PP*PN3, ZCAL, or CENT) and the specific trigger bits. It does not specify the scaledowns for that mapping. It is intended that the trigger map would remain constant over the medium-term, only changer when trigger experts change some aspect of the trigger hardware, (eg adding a new trigger).
- Scale Down: A trigger's scale down is the n such that the DAQ takes 1/n events with that trigger.
- Trigger Configuration: The configuration is a set of trigger names paired with their respective scaledowns, defining logically what the trigger is doing in a particular run. The trigger configuration is intended to be allowed to change on the relatively short-term (even run-to-run). It is possible to have the same trigger configuration representing different sets of trigger bits at different times, if the trigger map changes, but the same trigger configuration should result in the same 'mix' of data at all times.

1.4.2 ORACLE tables

Given these frameworks, the ORACLE tables used in this setup are as follows. I will not discuss the issues of SQL design that went into them here; that is appropriate for an SQL textbook.

Column	Type	Notes
TRIGGER_CONFIGS		
config_name	String	Primary Key
Description	String	
TRIGGER_CONFIG_BITS		
config_name	String	Refs TRIGGER_CONFIGS.config_name. Primary Key(1)
Trigger_config_bit	String	Refs TRIGGER_BITS.Trigger_bit. Primary Key(2)
Scaledown	Integer	
TRIGGER_BITS		
Trigger_bit	String	Primary Key
Description	String	
TRIGGER_BIT_MAP		
Validity_date	String	Primary Key (1)
Trigger_bitnum	Integer	Primary Key (2)
Trigger_word	Integer	Primary Key (3)
Trigger_bit	String	Refs TRIGGER_BITS.Trigger_bit

I wish to make a few comments about this design (which is not the only imaginable one). First, the hardware level trigger bit number is described by two numbers: a trigger word and a trigger bit number. Second, the list of all triggers used in the experiment is maintained in Trigger_bits. Finally, note that trigger_bit_map does not contain any explicit reference to a single object like a 'trigger map', containing instead only the changes from time to time as to what particular trigger bits represent. Essentially, this means a new 'trigger map' object is defined for each different validity_date in the trigger_bit_map table. It is done this way in anticipation that the most frequent change will simply be to add a trigger bit to an existing setup.

1.4.3 Data Classes

The data classes being used will correspond almost exactly to the concepts in 1.4.1, with one exception. The scaledown will be represented by a simple integer. [5] I will only sketch the classes here. For more details, look at the classes TPhTrigMap and TPhTrigConfig in the CVS repository, which are where the design currently stands.

- TPhTrigMap defines an isomorphism between raw trigger bits and tirgger names, and can get either either one given the other.
- TPhTrigMap::TTriggerBit is defined as a subclass of TPhTrigMap in the belief that few other class in the offline framework will ever need to be concerned about the hardware representation of the triggers. It is basically just a wrapper for a pair of integers.

[5] It need not be. For example, present practice has the scaledowns restricted to powers of two. A class could be designed that understands that requirement. But for simplicity I will not do so.

- TPhTrigConfig is basically mapping between trigger names and scale-downs.

1.4.4 Key Classes

The key classes are defined to correspond to the keys by which we look up and retrieve the data objects. These I will present in somewhat more detail. Only TPhTrigKey exists in the current repository. I will consider them in reverse order for pedagogical reasons.

TPhTrigKey

TPhTrigKey (which is a key to the TriggerConfigs) is defined as follows:

```
class TPhTrigKey: public TPhNameKey {
public:
  TPhTrigKey(const TString &trigger_set_name):
    TPhNameKey(trigger_set_name) {}

  virtual TObject * GetFromSQLConnection(TPhSQLConnection &conn)
    const;
  virtual void WriteToSQLConnection(TPhSQLConnection &conn,
    const TObject &obj);

  virtual void SetSideBandData(const TPhDbKey &) const;

  void SetDescription(const TString &descrip) {
    fDescription = descrip;
  }

protected:
  virtual const TString & TableName() const;
  virtual const TString & FieldName() const;
private:
  mutable TString fDescription;

ClassDef(TPhTrigKey,1)
};
```

In this case, much of the work has been done quite simply, by deriving from a pre-existing class, TPhNameKey, intended as an abstract base class for classes with a single string field as their field. This defines correctly all of the various hash, equality, and stringkey functions. A few things have to be done nevertheless.

First, the two (pure in TPhNameKey) virtual functions TableName and FieldName have been overridden so that the TPhNameKey code knows how to do its work.

Second, GetFromSQLConnection of course has to be written, because the extent of the data being read is not known to TPhNameKey. Similarly for WriteToSQLConnection.

Third is the function SetSideBandData. Note that the class itself is really just adding the TString fDescription (pulled from the field of the same name in the ORACLE table) to the TPhNameKey code, and that this field is side band data – it is not relevant to the key itself, and it is not being stored with the Phat object either. [6] The function SetSideBandData is just:

```
void TPhTrigKey::SetSideBandData(const TPhDbKey &key) const {
  TPhNameKey::SetSideBandData(key);
  if (const TPhTrigKey * tkey = dynamic_cast<const TPhTrigKey *>(&key)) {
    fDescription = tkey->fDescription;
  }
```

Note the call to the parent-class' SetSideBandData function, and the check that the type of the passed-in TPhDbKey is correct (it will be unless something else is screwed up, but...)

The fact that the data is sideband is incorporated in the key object by defining the as field `mutable`, which allows the above code to work even though it is a `const` member function. This is needed because a user may, for example, create a new TPhTrigKey on his own, with the constructor TPhTrigKey(TString &). The user will probably supply no (or a wrong!) description. Later when he uses the key to read in an object from the database, we want the description in your (possibly const-qualified) key to match the one in the database. This is achieved by calling the function SetSideBandData for the user's key.

One thing that probably should be added to the design is a Select function which gets the right trigger configuration for a particular run number, which would have to go to another table in the ORACLE database in order to decide that.

TPhTrigMapKey

Here I want to note that I could achieve much the same ends by deriving TPhTrigMapKey from TPhTimeKey, which supplies various functions and selectors useful for keys based on specific validity times. See TPhFECMapKey as an example. For pedagogical purposes I will not do so here. I have also omitted the sideband text description data, which would be handled similarly to the above. Here is the interface to TPhTrigMapKey:

```
class TPhTrigMapKey: public TPhDbKey {
public:
   TPhTrigMapKey(): TPhDbKey() {}
   TPhTrigMapKey(TDatime time): fTime(time) {}
```

[6] At least, not in this design. It *could* be stored with the data object, but my view was that the data object should contain just what it needed. Besides, this helps illustrate aspects of the key system.

```
   virtual ULong_t Hash();
   virtual Bool_t IsEqual(TObject *obj);
   virtual TString StringKey() const;
   Bool_t operator==(const TPhTrigMapKey &a) const;

   virtual TObject * GetFromSQLConnection(TPhSQLConnection &conn) const;
   virtual void WriteToSQLConnection(TPhSQLConnection &conn,
                              const TObject &obj);

   // Getter for the validity date omitted omitted.
   TPhTrigMapKeySelector Select(TDatime &time);
private:
   TDatime fTime;
ClassDef(TPhTrigMapKey,1)
};
```

Note that this key has an fTime, which should be restricted to the possible values of the validity_time in the database. This needs to be done in code in GetFrom- and WriteToSQLConnection, which I will not show. Getting the key for the 'best' time is the business of the Selector, which I will discuss more below.

First, I will show how the Hash and IsEqual are overridden. Remember, they *must* be defined in such a way that if IsEqual is true for two objects, the hash is the same.

```
ULong_t TPhTrigMapKey::Hash() {
  return IsA()->Hash() ^ fTime.Get();
}
```

This returns an integer, being the bitwise xor of the class's hash function, and the data field.

```
Bool_t TPhTrigMapKey::IsEqual(TObject *obj) {
  return ClassEqual(*this,*obj);
}
```

ClassEqual is a helpful template function (defined in phatutil/inc/DBKeyHelper.h) which checks to see that the classes of `this` and `*obj` match, and if so returns the value of operator==.

```
inline Bool_t TPhTrigMapKey::operator==(const TPhTrigMapKey &a) const {
  return fTime == a.fTime;
}
```

Finally, we have the check of the equality of the validity times.

The function StringKey() is quite simple, just outputting "TPhTrigMap-Key:126781", where the number is a translation of the time into an integer:

```
TString TPhTrigMapKey::StringKey() const {
  ostrstream buf;
  buf << "TPhTrigMapKey:" << fValidityDate.Get();
  return TString(buf.str());
}
```

This result will be used as the ROOT key for the TriggerMap if it gets written to a ROOT file.

The Selector

Finally, the hardest part. This must be written to select the latest time in the database before the current time. In showing this, I have omitted a fair bit of generalization work that I have used in other places here.

```
class TPhTrigMapKeySelector: public TPhDbKeySelector {
public:
  TPhTrigMapKeySelector(TDatime time);

  TPhDbKey* GetKeyFromSQLConnection(TPhSQLConnection &conn) const;
  TPhDbKey* GetKeyFromList(const TCollection &keylist) const;

private:
  TDatime fTime;

ClassDefDoc(TPhTrigKeySelector,1)

};
```

Note the ClassDefDoc, because TPhDbKeySelector is not a TObject.[7] Here, I will demonstrate the implementation of the virtual override of GetKeyFromSQLConnection, which requires some understanding of the SQL-using parts of the previous. This function returns a pointer to a new key, which the caller is responsible to delete, chosen as the 'best one' the selector can find in the database.

```
TPhDbKey *
TPhTrigMapKeySelector::GetKeyFromSQLConnection(TPhSQLConnection &conn)
const {
  TPhSQLStatement *s = conn.NewStatement(
"Select max(validity_date) from PHOFFLINE.TRIGGER_BIT_MAP "
"where validity_date <= :indate\");
  s->BindDate(":indate",&fTime);
  TDatime outdate;
  s->DefineDate(1,&outdate);
```

[7] This is because the selectors are often returned by value, and I want to avoid the TObject overhead in that process.

```
  if (s->Execute()) {
     delete s;
     return new TPhTrigMapKey(outdate);
  } else {
    delete s;
    return 0;
  }
}
```