We begin by defining the simplest structure: the TableKey.

It is composed by a persistent class and a primary or foreign key. This is the naming convention:

* using *Table*PK = TableKey<*Table*, *PrimaryKeyColumn*>
* using *SourceTableTargetTable*FK = TableKey<*SourceTable*, *ForeignKeyColumn*>

So imagine we have the following Entity-Relationship diagram:

Diagram

Description automatically generated

We have Fondo with a primary key column (id) that is referenced by the Inversion and the Rendimiento tables. Further assume that each table corresponds to a persistent struct of the same name (1-1 mapping between normalized tables and persistent “atoms”[[1]](#footnote-1) ).

We can therefore express the relationships or connections by declaring the following TableKeys:

* using FondoPK = TableKey<Fondo, &Fondo::id>;
* using InversionFondoFK = TableKey<Inversion, &Inversion::fkey\_fondo>;
* using RendimientoFondoFK = TableKey<Rendimiento, &Rendimiento::fkey\_fondo>;

For foreign keys we speak of the TargetTable as the one containing the primary key that the FK links to. Thus &Inversion::fkey\_fondo is a FK that targets the column id of the table Fondo, justifying the name InversionFondoFK

For each PK, a structure called PKDependencies connects it with all FKs linking to that PK. For the table Fondo we therefore declare inside a struct Fondos[[2]](#footnote-2) all PK and FK dependencies:

struct Fondos

{

using PKDependents = PKDependencies<FondoPK, InversionFondoFK, RendimientoFondoFK>;

};

Since Fondo has no FKs, we only declare the PK dependents.

However for the 2 other tables, which do have FKs, and which have PK that are not target to some FK, then we declare corresponding FKDependencies. We will focus on one of them since the other has the same structure.

struct Rendimientos

{

using FKDependents = typename FKDependencies<RendimientoFondoFK, Fondos::PKDependents>::construct::result;

};

For FKDependencies, the first template parameter is the FK TableKey RendimientoFondoFK. The second parameter is the PK dependencies of the target table Fondo. We don’t want to restate the PK dependencies again, so we reuse the Fondos::PKDependents. This structure has all the dependencies between the PK and the relevant FKs.

Let’s examine these structures in more detail.

1. TableKey
   1. Receives a persistent atom as the first template parameter and the key information as the second parameter. It exports:
      1. Table = the persistent atom[[3]](#footnote-3)
      2. Key = the PK or FK of the atom
   2. It enables setting a Table instance so we can obtain its key value
      1. setRecord() and getKeyValue()
2. PKDependencies
   1. Inputs:
      1. T = a TableKey with PK info
      2. …RefBy = includes TableKeys for all FK pointing to T’s PK
         1. This list is stored in std::tuple called reference\_list
         2. Its TableKeys are dependent on the Target’s PK
   2. Exports:
      1. Target from T
      2. Reference\_list std::tuple from RefBy…
   3. Functionality:
      1. Has\_links must check all tables that have FKs that point to the Target’s PK
      2. These tables are stored in reference\_list in FK TableKeys
      3. Thus we must check all the tables in reference\_list
      4. For each entry in reference\_list we have a Table and a Key and we call this entry a DependentClass because it’s a FK TableKey dependent on the TableKey with PK info (here called Target)
      5. Thus we look in the Key column of the Table for a value equal to the PK’s key value
      6. As soon as we find at least one record in the Table with FK = PK we return true from has\_links; else we recurse with an index – 1 until we either find a match or reach index == 0 which would correspond to position -1 in the reference\_list and thus would return false because no links were found.
3. TableConnection
   1. Inputs:
      1. FK TableKey
      2. PK TableKey
   2. Exports:
      1. Dependent = a FK TableKey
      2. Target = a PK TableKey
   3. Functionality:
      1. Foreign key exists
         1. This case is related with inserts or updates and we want to ensure that any FKs present in an instance of a persistent atom exists as a PK in the Target TableKey
         2. We do this by counting how many records in the PK TableKey have a key value equal to the FK TableKey’s value obtained by Dependent’s getKeyValue()
         3. This explores the relationship between one FK TableKey and its associated PK TableKey (known here as a TableConnection)
4. TableConnectionList
   1. Inputs:
      1. A list of TableConnection types[[4]](#footnote-4)
   2. Exports:
      1. Type at the head of the list
      2. DependentTable
   3. Functionality:
      1. Foreign key exists
         1. find list’s element count (size)
         2. call template function foreignKeyExists<index> with dep and index = size
            1. get the TableConnection at index-1 🡺 TableConnection
            2. ask TableConnection if foreignKeyExists with dep
            3. if not exists stops and returns false
            4. else calls foreignKeyExists<index-1> and continue until the end of the list (index ==0)
      2. if not exists then throw
5. FKDependencies
   1. Inputs:
      1. FK TableKey RendimientoFondoFK
      2. PKDependencies (only one!)
         1. Target = FondoPK
         2. Reference\_list = InversionFondoFK, RendimientoFondoFK
   2. Exports:
      1. tableKey RendimientoFondoFK
      2. pkDep\_list
   3. Functionality
      1. We start with an empty collection (Loki::NullType)
      2. For each element in reference\_list, we ensure tableKey is in the reference\_list of the PKDependencies
         1. In our case is RendimientoFondoFK in { InversionFondoFK, RendimientoFondoFK } is true
         2. Thus we append to the collection a TableConnection made up of tableKey, PKDependency::Target
            1. RendimientoFondoFK, FondoPK
      3. We return a TableConnectionList

TableConnection receives FK TableKey and PK TableKey (RendimientoFondoFK, FondoPK):

Dependent = RendimientoFondoFK

Target = FondoPK

int count = storage.count<typename Target::Table>(where(is\_equal(Target::Key, Dependent::getKeyValue())));

int count = storage.count<Fondo>(where(is\_equal(&Fondo::id, RendimientoFondoPK::getKeyValue())));

TableConnection has the form

1. Meaning basic persistent classes that map 1-1 with normalized tables [↑](#footnote-ref-1)
2. In plural so as not to conflict with the name of the persistent atom Fondo [↑](#footnote-ref-2)
3. We verify it is a persistent class using a requires clause (a C++ 20 feature) [↑](#footnote-ref-3)
4. Using Loki typelist: See Modern C++ Design: Generic Programming and Design Patterns Applied [↑](#footnote-ref-4)