Representing Uniqueness Constraints in Object-Relational Mapping
The Natural Entity Framework

Mark J. Olah, David Mohr, and Darko Stefanovic

Department of Computer Science, University of New Mexico
1 University of New Mexico, Albuquerque, NM, USA 87131
{mjo,dmohr,darko}@cs.unm.edu

Abstract. Object-oriented languages model data as transient objects, while relational databases store data persistently using a relational data model. The process of making objects persistent by storing their state as relational tuples is called *object-relational mapping* (ORM). This process is nuanced and complex as there are many fundamental differences between the relational model and the object model. In this work we address the difficulties in representing entity identity and uniqueness consistently, efficiently, and succinctly in ORM. We introduce the *natural entity* framework, which: (1) provides a strong concept of value-based persistent object identity; (2) allows the programmer to simultaneously specify natural and surrogate key constraints consistently in the object and relational representations; (3) provides object constructors and initializers that disambiguate the semantics of persistent object creation and retrieval; and (4) automates the mapping of inheritance hierarchies that respect natural key constraints and allows for efficient polymorphic queries and associations.

1 Introduction

In an object-oriented (OO) language, data are represented as objects, but objects are transient—they do not persist beyond a particular process or between subsequent executions of a program. To make the data persistent and accessible for concurrent processes in a structured form, an *object-relational mapping (ORM)* can be used to store objects as tuples in a relational database.[1] An ORM is a method for translating between a data model expressed as a class hierarchy and a data model expressed as a relational schema. ORM software packages allow a program to create, read, update, delete, and query objects stored persistently in a relational database using object and class methods of an OO programming language.

Designing an ORM presents many challenges because the object data model and the relational data model differ profoundly in how they represent, store, and access data. We focus in this work on just one facet of the mapping between the models: the concept of *identity and uniqueness*. Both data models are used to abstractly represent

[1] There are other possibilities, beyond the scope of this paper, such as using a persistent object store and a programming language that supports persistence natively. Without going into the merits of different approaches, we concentrate on ORM because of its widespread use.

sets of physical or conceptual entities. An entity has multiple properties; the values of these properties may affect entity identity and entity uniqueness. However, the concepts of identity and uniqueness have different semantics in the object model and in the relational model [8].

In relational models uniqueness is a value-based notion defined by relational keys. A *key* is a minimal set of attributes (columns) of a relation that uniquely identifies a particular tuple (row). It can be a *surrogate key*, an artificial value introduced solely to distinguish tuples; or it can be a *natural key*, consisting of attributes that correspond to meaningful, real-world, properties of the entities. The attributes in a natural key represent those properties of an entity that define its identity and uniqueness in the context of the application and are well known to the users of the entity. A natural key is a concise description that can be used to query for the existence of a specific individual entity. Every relation must specify a *primary key*, which is used as the default identifier for a tuple. For practical reasons this is often a surrogate key. However, when a natural key exists, it often makes sense to declare its existence as well by enforcing a uniqueness constraint on the natural key attributes. This prevents the database from maintaining two copies of data that represent the same entity. Additionally, declaring a natural key results in the database maintaining an index on the natural key attributes, which allows queries involving the natural key to be optimized [6].

In contrast, in object models value and identity are independent. While an OO execution environment enforces the uniqueness of object identities, this imposes no constraints on the values of objects. Hence, when real-world entities are represented by objects, there can be many distinct objects having the same values for a set of natural attributes and thus representing the same entity. There are no mechanisms to prevent this error-prone duplication of entity representations, and typically no universal mechanism to query for the existence of an object based on its value.

This fundamental difference in how uniqueness and identity are defined in relational databases and in OO programming languages leads to problems when data representing real-world entities are made persistent with a relational database, but are operated on as in-memory objects. If there are multiple in-memory objects all denoting the same entity, which object represents the true current state of that entity, and which one corresponds to the database's current state, i.e., the tuple representing the entity? This question becomes even more confusing when there are multiple execution contexts operating on entities concurrently.

Our real-world motivation for developing the natural entity framework comes from the experience of writing scientific computing simulations, which are distributed, concurrent applications with persistent state. Some of our examples will be drawn from this field; similar modeling and representation problems are encountered in the business world and in web-based applications.

To properly model the concept of entity uniqueness and identity at both the object and the relational level, we propose a new framework of constraints and semantics for object construction and interactions that can be enforced in modern ORM systems and strongly object-oriented languages. Our *natural entity* framework provides a base class `NaturalEntity` with the functionality described in this paper. Natural entities are persistent objects in an OO execution environment that directly enforce value-based uniqueness constraints on natural attribute values. Other ORMs allow natural keys and uniqueness constraints to be declared on the relational model, but they do not enforce these constraints on the object model, or in the inheritance hierarchy. Making these constraints explicit allows persistent objects to more directly represent the semantics of relational tuples used to store their state. This simplifies the programmer's conceptual model and reduces potential problems with concurrency, entity identity, and uniqueness.

In contrast to creating regular objects, there is overhead when checking for value-based uniqueness, but this overhead is not higher than manual enforcement of uniqueness. The proposed natural entities are otherwise normal objects that exist alongside, and interact with, other objects, and they can be queried and used polymorphically. Hence, the natural entity framework does not reduce the expressiveness of the OO language, and a programmer is free to represent entities using persistent objects that do not enforce uniqueness constraints, or using regular non-persistent objects. However, only through the use of the natural entity framework can the programmer maintain the value-based uniqueness constraints for in-memory objects.

The primary contribution of the natural entity framework is that it allows the ORM to manage and enforce value-based object identity and uniqueness on in-memory objects. These value-based constraints match the constraints imposed by natural keys on the relations that store the persistent state of the natural entities. Thus the object model for natural entities is modified to more closely match the relational model.

This provides several advantages: (1) natural entities have a strong concept of value-based identity and uniqueness, accessible through object attributes and methods that prevent multiple in-memory objects from representing the same conceptual entity (Sec. 3); (2) the ORM can use an identity map to provide fast value-based queries for in-memory objects and a uniqueness constraint to provide fast queries for archived objects (Sec. 4); (3) natural entities have constructor methods that automatically manage the uniqueness constraints for in-memory objects and disambiguate object construction from object retrieval (Sec. 5); and, (4) natural entities inheritance hierarchies can be mapped automatically to a relational schema that uses the appropriate constraints and relations to maintain natural key uniqueness constraints and to allow polymorphic queries (Sec. 6).

Given these features, the natural entity framework provides functionality that is lacking in modern ORM systems and presents an abstraction that is easy to understand and implement, allowing the programmer to spend more time on solving the actual problems at hand. We found this to be the case in our work on scientific simulations, and we offer this description in the belief that the framework will be broadly applicable.

2 Background

To be specific about how the concept of uniqueness constraints is implemented, here we summarize the terminology used for relational models and OO programming languages.

2.1 Relational Model

A *relation* is a tuple of attributes denoted $R = R(A_1, \ldots, A_n)$. The attributes come from some domain $\mathbf{A}$, and each attribute A_i has a type τ_i, (written $A_i : \tau_i$), where $\tau_i \in \mathbf{T}$ for some set $\mathbf{T}$ of basic types. For brevity we omit type signatures where they are not essential to the discussion. A *relation instance* is a set of tuples from the domain $(A_1 \times \ldots \times A_n)$ that represents the current factual state of the relation. When it is not otherwise confusing, the term *relation* is used to describe both the relation's schema (attributes, types, and constraints) and its time-varying instances (the tuples and their values). In the concrete context of a relational database, a relation specifies the names and types of the columns of a table, and an instance specifies a set of table rows and their values.

A non-empty set $k \subset \{A_1, \ldots A_n\}$ is a key of relation $R(A_1, \ldots A_n)$ if for any instance of the relation, the value of the attributes in k uniquely determines a tuple and no proper subset of k is also a key. Thus, a key is a minimal set of attributes that can be used to define the identity of a tuple. A relation may have many keys. A key is *simple* if it consists of a single attribute, otherwise it is *compound*. Each table must have a *primary key*, which is used as the canonical set of attributes for identifying a row for the purpose of database operations and references between tuples of relations. Primary key attributes are underlined in the notation for a relation to highlight their role (e.g., $R(\underline{A_1}, \underline{A_2}, A_3)$ has a primary key $\{A_1, A_2\}$.) Associations between relations are expressed with a *foreign key constraint* that restricts a set of attributes to values that come from the relational instance state of a separate set of attributes that form a key [3].

A *relational schema* is a set $\mathbb{R} = \{R_1, \ldots, R_m\}$ of relations along with constraints. A relational database provides a set of types and mechanisms to define relational schemas over those types. It maintains instances for each relation that obey all the restrictions and allows queries to create, read, update, and delete tuples.

2.2 Object Model

An *object* lives in memory and has identity, type, state, and behavior. An object's state is given by the values of a collection of named attributes that come from a set of types $\mathbf{T}'$.[2] In strongly object-oriented languages, objects have a concept of identity independent of their attribute values or addressability [9]. This allows references to objects to be tested if they refer to the same object, and hence forms a definition for object uniqueness.

An object's type is some class C. A class creates objects: it defines names and types for each attribute, and the set of *methods* that operate on the state of an object. These methods define the behavior of the object. An object that belongs to a class is said to be an *instance* of that class.

Inheritance. A set of classes $\mathbb{C} = \{C_1, \ldots, C_k\}$ is called a *class schema*. Classes have a concept of inheritance. If C_i inherits from C_j, we write $C_i <: C_j$, and the class C_i inherits all of the attributes and methods of C_j. The inheritance relation is reflexive, transitive, and antisymmetric, and so defines a partial ordering on the class schema, called the *inheritance hierarchy*. This relation represents specialization as objects of class C_i now can represent all the state and behavior of C_j, but can also add or modify attributes and methods. Thus, if $C_i <: C_j$ and o is an instance of C_i, then o is also an instance of C_j.

[2] The set of OO types $\mathbf{T}'$ may, but does not necessarily, intersect with the set of types $\mathbf{T}$ used in the relational schema. They will almost certainly not be identical.

This property is called *polymorphism* and allows objects to act as an instance of any class more general than their own.

The maximal elements in the hierarchy are called the *base classes*. In many languages multiple inheritance is possible, so a class can inherit directly from more than one class. Multiple inheritance is not a focus of this paper, though the implications are briefly considered. In a single inheritance class schema, the inheritance hierarchy is not a general lattice, but a forest of *inheritance trees*, each rooted at a single base class. For single inheritance hierarchies we can uniquely define the super relation $\text{Super}(C_i) = C_j$ if $C_i <: C_j$ and $C_i <: C_k <: C_j$ implies $C_k = C_i$ or $C_k = C_j$. In other words, the super relation determines the smallest class larger than a given class, called the immediate superclass. Conversely, C_i is said to be a subclass of C_j.

A class can be *abstract* or *concrete*. There cannot be objects belonging to an abstract class, only to concrete classes. Abstract classes are only used to be inherited from by other classes.

2.3 Object-Relational Mapping

The object and relational models are general enough to apply to most modern OO languages and relational databases, hence they form a good basis for describing how objects can be mapped to relations. An ORM is a mapping from a class schema $\mathbb{C}$ to a relational schema $\mathbb{R}$ that provides a correspondence between objects in $\mathbb{C}$ and tuples (or sets of tuples) from relations in $\mathbb{R}$.

In this mapping attributes of an object with type $t_1 \in \mathbf{T}'$ are mapped to one or more tuple element with type(s) $\tau_i \in \mathbf{T}$. Since the types available in a programming language (subtly) differ from those available in databases, this mapping of types is a necessity, and may not be 1-to-1. However, for most uses the type differences have no practical effect, and we leave exploring the implications for value-based identity as future work.

3 Object Identity and Uniqueness

The central issue addressed by the natural entity framework is consistently representing real-world entities that possess a concept of uniqueness described succinctly by the values of one or more well known (natural) attributes, i.e., a natural key.

Identity in OO languages. Like objects in the natural world, objects in a programming language have concepts of identity and uniqueness. Many OO programming languages (Python, Smalltalk, Java, Ruby, etc.) have a strong concept of object uniqueness in that each object has an associated immutable internal id(entifier), distinct from the references used to access it [9]. Such an id is called a *surrogate object id* since it has no relation to the value or meaning of the object. It merely serves to define the identity of the object and allows comparing the identity to those of other objects, as there is a bijection from object ids to objects [14].

Identity in Relational Databases. Identity in relational databases is a value-based property determined by a designated primary key. The primary keys should be unique, immutable, and non-null. The database maintains a uniqueness constraint on the primary key, preventing duplicate tuples, and uses an index to quickly select tuples by their primary key or detect violations of the uniqueness constraint. The primary key is also used to define foreign key relationships.

Because of all these important requirements placed on the primary key, it often makes sense to use a surrogate key as the primary key, even when there is a well-known natural key. There are many good reasons to prefer surrogate keys as primary keys, most of which arise from the fact that using surrogate keys allows the relational schema to decouple identity and value [4]. This allows more flexibility when the relational model needs to be updated or refactored [1]. Other benefits arise due to the fact that surrogate keys are simple (consist of a singleton attribute) and are typically small integral types. Natural keys in contrast are often compound and may include strings and other types that require more space as foreign keys. Since the primary key is always used to represent entity relationships through foreign key constraints, having a small, simple primary key reduces space usage and simplifies join operations. Simple integral keys are also often faster for use in selects against the primary key. For these reasons, ORMs often use surrogate primary keys by default [5].

However, natural keys are still useful and have some desirable characteristics. Declaring a natural key communicates to the database that the relational model has a logical uniqueness constraint on the natural key attributes and prevents a single conceptual entity from being represented by more than one tuple. Additionally, the database can then maintain a uniqueness constraint and index on the natural key. The presence of an index allows clients to quickly retrieve objects by their natural key-values, or determine that no such object exists. This can lead to distinct performance advantages for natural keys in some situations [11].

3.1 Identity in the Natural Entity Framework

The natural entity framework, like other ORM tools, must reconcile the semantics of object identity in OO languages and tuple identity in relational databases. Our goal is to enforce the uniqueness of entity representation across both data models as determined by natural key attributes, but we simultaneously want to support polymorphic queries, efficient entity relationships, and flexibility for refactoring databases.

To achieve these objectives, the natural entity framework enforces the simultaneous use of surrogate primary keys and auxiliary natural keys. This dual-key representation achieves advantages of both surrogate and natural keys. In particular, our surrogate keys are unique within each inheritance hierarchy rooted at the `NaturalEntity` class. This uniformity of primary keys allows us to use a single top level relation to define a primary key for every object belonging to the class hierarchy. This makes polymorphic queries and associations much more efficient and uniform than they could be with natural keys. Indeed, without a uniform key for the entire inheritance tree, representing polymorphic associations would become problematic as there would be no single foreign key constraint that could be used to represent an association. Hence, surrogate primary keys are necessary for polymorphism and flexibility, but they do not fulfill the need for maintaining value-based uniqueness. This is achieved by the auxiliary natural keys. These keys require a separate index, which comes at the cost of storage space and maintenance time. However, this index is exactly what ensures the logical value-based uniqueness of natural entities, and it is heavily used by constructors (Sec. 5) and other common queries against the natural key, thus it is both necessary and useful.

4 Management of Persistent States and Concurrency

Building on the concepts of object and relational identity, an ORM must have a way to track and manage the identity of in-memory objects. Unlike transient objects, which have a limited scope and lifetime, persistent objects must maintain their identity permanently and consistently across concurrent processes. To simplify the tracking of persistent objects and their modifications, modern ORM packages provide the concept of a *session manager*. The natural entity framework relies on a session manager to manage the persistent state of in-memory persistent objects and enforce the uniqueness constraints for natural entities.

Our principal contribution is to provide additional constructor methods which make explicit the assumptions about the state of a persistent object when it is created and prevent the user from violating the value-based uniqueness constraints.

4.1 Transactions

The session manager has transactional semantics and manages a set of persistent objects by implementing the *unit of work* concept [5]. It tracks object creation, modification, and deletion. The session manager delegates large parts of this work to the database by using transactions. This ensures a consistent database state, even when objects are modified concurrently by other processes. It follows that the concurrency guarantees are largely provided by the transaction. The session manager supplies methods to control the global transactional state for an execution context. The `begin()` method starts a transaction and is implicitly called as needed if no transaction is currently in progress. The `flush()` method sends pending modifications to the database, but does not end the transaction. The `commit()` method commits a transaction, and this implies a flush operation if there are still pending changes. Finally, the `rollback()` method undoes all database changes made during the transaction.

4.2 Object States

From the perspective of an OO execution environment, reasoning about persistent objects is much more complicated than standard transient objects because the data representing the object can be stored in memory, in one or more relations in the RDBMs, and/or in the memory of other concurrent processes. The session manager acts as the single point of persistence management for an OO execution environment. It determines how a persistent object relates to its external relational state in the database. Any object of a class that derives from a persistent base class, such as `NaturalEntity`, will be understood by the session manager to be in one of the following six states:

- *Transient* – The object is not managed as persistent by the session, while a corresponding tuple with the same natural key in the database may or may not exist; there is no operational connection with any persistent object.
- *Pending* – The object does not yet have a permanent record but has been successfully added to the session and will be added to the database when the session state is flushed to the database. Until the object is successfully flushed it has not yet been assigned a primary key.
- *Persistent Clean* – The object has a primary key and a corresponding representation in the database. No persistently managed attributes have changed values, so no updates need to be sent to the database.
- *Persistent Dirty* – The same as a persistent clean object, except the value of one or more of the persistently maintained attributes has been changed, so that an SQL update operation is needed to save the state of the object. Copies of this object in other sessions do not know about the changes and may have made conflicting changes of their own.
- *Expired* – The object's state is no longer valid because it was created in a session that has been committed or rolled back, so its state needs to be reloaded from the database. This reloading is done transparently by the session manager when necessary.
- *Archived* – The object is not part of the store but is persistently stored in the database. Strictly speaking, this is not a state of an object, since no corresponding object exists in the session, but conceptually the tuple in database represents an object that is not currently loaded.

It is important to remember that the identity of a persistent object is provided by the natural key, and maintained through transactions and the constraint imposed by the database key. In case of conflicting concurrent transactions, e.g., simultaneous inserts or deletes, one of the concurrent processes will be prevented from committing its changes by an exception. In Fig. 1 we show the effect of various operations on the persistent state of an object, but omit the expired state and other effects that occur at transaction boundaries. The effect of commits is to expire all pending and persistent objects and the session manager updates any identity maps of persistent objects accordingly (Sec. 5.1).

5 Object Creation

Maintaining a value-based uniqueness constraint for persistent objects causes difficulties with object creation. Normally, the programming environment's concept of object identity is all that determines object uniqueness. When an object constructor is called, a new object with a unique object id is always created, and an initializer method is called. However, natural entity classes with value-based uniqueness constraints necessitate different semantics. First, the constructor must be given the values for each of the natural key attributes since they must not be null. Given the natural key value, the constructor is presented with several possibilities: (1) an object with those values already exists in memory so we are not allowed to create a new object with a new object id and the same natural key values; (2) an object with those values exists in an archived state, so it must

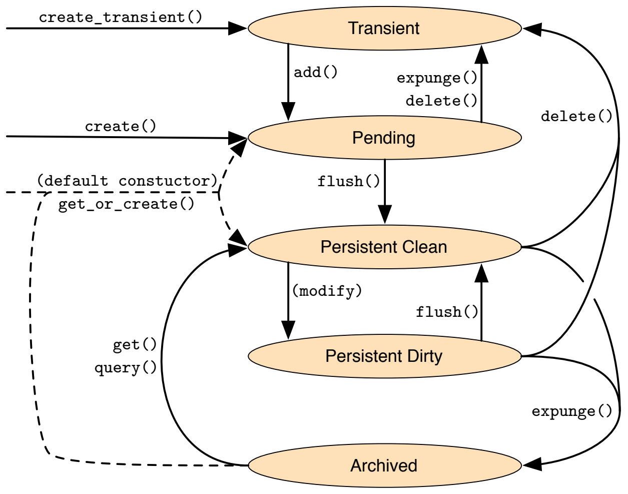

Fig. 1. Persistent object states and effect of constructors and session commands within a single transaction context. The effects of transaction boundaries and the expired state are omitted for clarity.

be loaded from the database; or, (3) there is no persistent or in-memory object with the given natural key, so a new object should be created and added to the database.

Such a constructor requires a natural-keyed dictionary of in-memory persistent objects, i.e., an identity map (Sec. 5.1), and a mechanism to query for the existence of archived objects. Both of these can be provided efficiently by the session manger, but they nevertheless impose a significant cost, especially when the round trip time for remote database queries is involved. Unfortunately, such queries are necessary if we wish to maintain the consistency constraints; allowing the constructor to make new objects without regard to the natural key values would result in duplicate objects in memory. Furthermore, the cost of frequent queries can be reduced by allowing the caching of natural keys or prefetching of objects (particularly when the database transaction isolation prevents non-repeatable reads). When queries are necessary they can be handled efficiently because of the unique index maintained on the natural key attributes.

Together all of these considerations impose a significant change to the semantics of object creation, and can lead to conceptual problems for programmers. The natural entity framework addresses this conceptual ambiguity by providing additional constructor methods with different semantics. These constructors allow programmers to explicitly state their intentions or assumptions when creating an object.

- `get()` - A constructor that takes the natural key and returns the object uniquely identified by that key, either by returning a reference to an in-memory object representing that entity, or by loading an archived object from the database and returning it in the persistent clean state. If no such object exists, an exception is raised.

- `create()` - A constructor that takes the natural key and returns a newly created object in the pending state, but only if no persistent object with the same natural key exists in memory or in the archived state. An exception is raised if the object already exists.
- `get_or_create()` - A constructor with the combined semantics of the `get()` and `create()`. It takes the natural key and either returns an existing persistent object, or returns a newly created object in the pending state. This is the default constructor.
- `create_transient()` - A constructor with normal transient object semantics that always returns a new object in the transient state. It can take arbitrary arguments and ignores the uniqueness constraints.

The `get_or_create()` constructor does whatever it takes to get a reference to the unique object that has the provided natural key. It will find that object if it is in memory and return a reference, or it will look in the database for an archived version and return it, and if no such persistent object exists, it will construct a new object and make it persistent by moving it to the pending state. In our application domain we found that the `get_or_create()` has the appropriate semantics in the vast majority of situations, and therefore we have made it the default constructor, which results in particularly succinct code (e.g., in Python `var=ClassName(...)`).

The `create()` and `get()` constructors are used in cases where the existence or non-existence of a particular `NaturalEntity`object represent a logical error, and the programmer would like an exception to be raised so that the error is not silently ignored.

Finally the `create_transient()` constructor has several uses when the normal semantics of the natural entity construction are too rigid. Unlike the other constructors, `create_transient()` does not need to be given the natural key, and does not use any database connections or in-memory identity maps. This is useful for testing object behavior without using a database. Transient objects are also useful when the user does not wish to immediately pay the cost of the database query to check for archived objects. Furthermore, they support situations where not all of the natural key attributes are immediately available, but it makes sense to partially construct a `NaturalEntity` object, and then finish filling in the natural key attributes later. This is often the case in GUI or web-based applications where objects are built up sequentially by user actions. A transient object can be made persistent by using the `add()` method, which will check that all natural key attributes are specified and will raise an exception if the object already exists.

5.1 Identity Map

When the (non-transient) constructors are called, they are provided with the complete natural key for the desired object. If an object with that natural key already exists in memory in the pending, expired, persistent clean, or persistent dirty states, it would be incorrect to construct and return a new object. Instead we must return a reference to the in-memory object. The ORM's session manager is able to track the persistent state of objects, but it also needs a way to look up objects by their natural key. This is a common requirement for ORMs, which Fowler calls the *identity map* pattern [5]. The purpose of an identity map is simply to map database keys to in-memory objects. When working with persistent objects, sometimes different parts of the code need access to the same data object without understanding whether that object is already in memory. The solution is to keep a global registry (or identity map) of in-memory objects keyed by their primary key. Normally, this identity map is stored in the session manager object, and it is used for internal ORM lookups of foreign key mappings. However, when primary keys are surrogates, it is awkward for a user to make use of this identity map, because the surrogates by definition are meaningless and often obscured from the user. It is much more common for a user to query using natural key attributes, and the constructors must be able to do this efficiently for in-memory objects. Hence, the natural entity system implements an auxiliary identity map, keyed on the natural key attributes. The identity map only stores in-memory persistent objects, i.e., transient objects are excluded. If an object is removed from the persistent store with the `delete()` method, it becomes transient. Thus, a constructor will not return a reference to a deleted object, even if that object is still in memory.

5.2 Initialization

Since the `NaturalEntity` constructors have multiple possible mechanisms for retrieving or creating objects, the concept of initialization also needs to be refined. For natural entities there are three distinct ways a new in-memory object could be created and require initialization: (1) it could be created as a transient object; (2) it could be retrieved from an archived state in the database; or, (3) it could be created as a new persistent object in the pending state. (In the case where the constructor already found the object in memory through the identity map, no initialization is needed.) The `NaturalEntity` class provides three different initializers that will be called by the constructor in each of the three cases.

- `initialize()` – This method is called when a new persistent object is created. The object will be in the pending state and the object's (immutable) natural key attributes will have been set to the values provided to the constructor.
- `reinitialize()` – This method is called when an archived object is brought into memory by a constructor. The object will be in the persistent clean state and all persisted attributes (including the natural key attributes) will have been set by the ORM system.
- `initialize_transient()` – This method is called if and only if the object is constructed with the `create_transient()` method. The object will be in the transient state, and any supplied natural key attributes will have been set, but those omitted by the user (which is permitted for transient objects) will have no default value.

5.3 Object Creation Semantics in Other ORMs

The multiple constructors of the natural entity framework represent a departure from the normal mechanism of persistent object creation presented by modern ORMs. In many modern ORM systems, all objects are initially created as transients, and only after a call to an `add()` method are they moved to a pending (or equivalent) state [13, 10]. The difficulty with this mechanism is that it does not allow the ORM to directly manage value-based object uniqueness. If an object with identical natural key already exists in the database, then the next time the session state is flushed, an exception will be raised when the database prevents the SQL INSERT command from violating the uniqueness constraint on the natural key. This failure mode can be eliminated by always first querying for a particular natural key before attempting to create and add an object with that key. This common ORM idiom is often required in code manipulating objects with natural keys. The constructors available for `NaturalEntity` classes make the assumptions of the programmer explicit, succinct, and less error-prone. Instead of remembering to first check if an object already exists before creating it, a programmer can just create a `NaturalEntity` object by passing the natural key to the constructor, and the system will automatically do the right thing; i.e., return the unique object with given natural key. Thus the overhead of the natural entity constructors is comparable to what would be required by any other implementation that wishes to protect against failures due to duplicate objects.

6 Mapping Natural Entity Inheritance Hierarchies

All natural entity classes must inherit from the `NaturalEntity` class, thus we must map all the classes in each inheritance subtree rooted at `NaturalEntity` into a relational schema. The natural entity system supports flexible mapping of hierarchies to relations, which allows polymorphic queries and associations, as well as different natural keys for separate subtrees of the inheritance hierarchy. The user only needs to supply minimal information about the desired inheritance mapping strategy and the ORM can automatically construct the appropriate tables and constraints. As an example we consider a distributed computer simulation system with two inheritance hierarchies, an abstract `Experiment` class with two concrete subclasses and an abstract `Measurement` class also with two concrete classes (Fig. 2). An `Experiment` has a one-to-many relationship with measurements, so each `Measurement` has a foreign key to the `Experiment` hierarchy's primary key—a polymorphic association. We examine natural keys in the relation further in Sec. 6.2.

6.1 Inheritance Mapping Strategies

The relational data model has no built-in concept of inheritance, but support for inheritance and polymorphism can be enforced by appropriately structuring the relational schema and queries. There are three standard methods for mapping inheritance hierarchies to a relational schema [5]: (1) the *single table* strategy maps all classes in an inheritance hierarchy to a single table; (2) the *class table* strategy maps each class to its own table; and (3) the *concrete table* strategy maps only concrete classes to tables.

The single and class table strategies are particularly useful for polymorphic queries and associations as for every class in the hierarchy they store the class name (i.e., the type) and a surrogate object id in a single top level table. Concrete table inheritance lacks these properties and is not considered further.

Single and class table strategies are distinguished by the technique they use to represent the differing attributes for classes in the hierarchy. Single table inheritance has

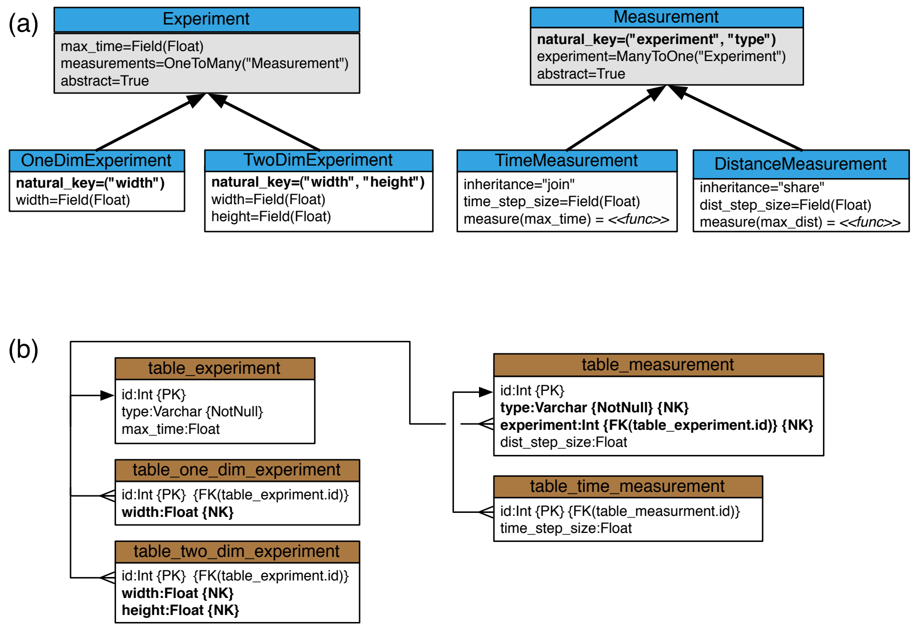

Fig. 2. (a) A simple example of a class schema with two inheritance hierarchies, abstract classes, multiple natural key bases, polymorphic associations, and both shared and joined inheritance mappings. The text in each class entry is close to the actual amount of code needed to specify this hierarchy. We use syntax that is similar to our Python-based reference implementation of the natural entity framework. (b) The relational schema generated by the natural entity framework from the class schema in (a). The foreign key constraints are shown.

a single relation which includes all attributes of all classes in the hierarchy. It allows polymorphism by permitting attributes to be null for objects that do not include them. In contrast, class table inheritance only includes non-inherited attributes in each class table. It permits polymorphic queries by using joins on the primary surrogate key to retrieve attribute values from all the relations that store an object's state. These differences lead to quantifiable performance and space trade-offs [7]. Modern ORMs allow the user to specify a mixture of these strategies within a single inheritance hierarchy [2]. When mixing strategies, the single table approach is called *shared* or horizontal mapping, while the class-table approach is called *joined* or vertical mapping [12]. Shared table inheritance works best when the cost of additional join operations needed to load rows is a limiting factor, or when a portion of the class hierarchy shares almost all of the same persistent attributes. Joined table inheritance works best when database space is constrained, or in portions of the hierarchy where few persistent attributes are shared between classes.

In the natural entity framework each class in a hierarchy only needs to specify if it will use the shared or joined inheritance strategy and the ORM can automatically derive the relational schema.

6.2 Natural Keys and Inheritance

Every concrete class that derives from `NaturalEntity` must define or inherit a natural key, so that the constructor can enforce the value-based uniqueness constraint. Abstract classes need not define a natural key, and any class that has no natural key must be declared as abstract.

Because of the option to use joined inheritance, an individual object can have its attributes stored in several relations, but there is always a relation that stores the attributes declared specifically in a class. This is the *primary relation* of the class.

Consider a class C that defines a natural key and that has no superclass which also defines a natural key (i.e., it has only abstract superclasses). The natural key results in a uniqueness constraint which is implemented by the database. A constraint can typically only be defined on attributes in a single table and not on joined tables. It follows that exactly one of the relations representing C must enforce this constraint. None of C's superclasses could have a natural key constraint, as enforcing a uniqueness constraint on $\text{Super}(C)$'s primary relation would prevent other subclasses of $\text{Super}(C)$ from defining different natural keys. Hence, the natural key constraint for C must be enforced in C's primary relation. This implies that all C's natural key attributes must be defined in C and cannot be inherited, or they would not be present in C's primary relation. Finally, note that any subclass of C will inherit C's natural key attributes, and because these attributes have a uniqueness constraint defined on the relation that stores them, the subclass must also inherit the natural key from C.

Therefore in any inheritance chain, i.e., starting at a concrete class and following the super relation to a base class, there is exactly one class that declares a natural key. Such a class is called a *natural key base*, as all classes that inherit from the natural key base share the same natural key constraint and store their natural key attributes in the primary relation of the natural key base. Furthermore, a natural key base, must use the joined inheritance mapping strategy, because if C is a natural key base, $\text{Super}(C)$ does not have a natural key, and so the natural key attributes and uniqueness constraint must be defined in a separate relation from $\text{Super}(C)$'s primary relation.

Hence, when mapping a class hierarchy to a relational schema, the mapping will require: (1) a single table for the root class to store the primary key and object type; (2) a table for each natural key base (unless the class is also the root); and (3) a table for each class that uses joined inheritance (unless the class is a natural key root or the base class).

Full-fledged multiple inheritance does not fit into the semantic model of object identity in this paper. However, the concept of mixins (additional abstract base classes) is easily supported, because a mixin does not define entity identity or uniqueness.

6.3 Type as a Natural Key Attribute

A natural key base will pass on its natural key to all of its subclasses, and thus only one object of *any* derived class may have a given natural key value. Sometimes this is too restrictive a condition on the classes. Because the natural key distinguishes objects based on their value, but not their type, it restricts cases where objects have identical values but different behavior because their respective classes have different methods.

For example, consider the class structure of the distributed simulation system in Fig. 2. The `Measurement` class defines a simple natural key as a foreign key relationship to the `Experiment` it measures. An experiment should be able to include both a `TimeMeasurement` and `DistanceMeasurement` instance. However, because these objects have the same natural key this becomes impossible. The two measurement subclasses have the same attributes, but the meaning of the attributes differs due to different method implementations. Thus, it can make sense to have more than one measurement object with the same natural key, provided they belong to different classes. This can be accomplished by adding the implicit type attribute to the natural key base's primary relation and thus adding the type to the uniqueness constraint. This allows multiple `Measurements` to belong to a single `Experiment`, provided they are from different classes.

In the natural entity framework the type can optionally be declared to be part of the natural key of a class to allow this distinction when it is required. The type attribute is automatically managed by the ORM, since it is always present as an attribute of any object in the OO programming language.

7 Conclusion

The natural entity framework provides an OO interface for programming with objects that have a strongly enforced concept of value-based uniqueness. These semantics require restrictions on object creation, initialization, inheritance, and relational structure.

The constructor methods of natural entities provide a consistent interface that distinguishes the different mechanisms by which a persistent class may be created and initialized. These constructors prevent the ORM from representing the same conceptual entity with different in-memory objects by ensuring that the value-based natural key constraints are maintained for all natural entity objects in the execution environment.

Enforcing value-based object identity changes the semantics of object models in the context of OO languages. However, these constraints only apply to objects from classes that inherit from `NaturalEntity`. Thus natural entities can coexist with objects of other less-strict persistent classes, as well as normal transient objects. Hence the natural entity framework makes it easier for a programmer to reason about object uniqueness for those entities which require it, but does not otherwise constrain the expressiveness of programs or programming languages. This level of flexibility makes a performance evaluation or validation of the framework complicated, as the natural entity framework will only be used in applications that benefit from value-based uniqueness constraints, and hence the specific application context is essential to the performance characteristics. In future work, we will quantify the performance of the Natural Entity framework under different application workloads and degrees of concurrency. Our own experience tells us that many applications have classes of persistent objects that logically require value-based uniqueness, and easily enforcing these constraints has been an invaluable tool in writing correct scientific software.

The natural entity framework can be implemented in any OO language that supports a strong concept of object identity. It relies on the facilities and abstractions provided by modern ORMs. Object and class introspection, and the ability to instrument object construction and destruction are helpful features in making the implementation easy to use. Our reference implementation in Python is built on top of the SQLAlchemy ORM, and the Elixir extension.

Acknowledgments. We thank the conference reviewers for their incisive and detailed comments. This material is based upon work supported by the National Science Foundation under grants 0829896 and 1028238.

References

1. Ambler, S.W.: Agile Database Techniques. Wiley, Indianapolis, IN (2003)
2. Cabibbo, L.: Managing inheritance hierarchies in object/relational mapping tools. In: CAiSE 2005. pp. 135–150 (2005)
3. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM 13(6), 377–387 (1970)
4. Codd, E.F.: Extending the database relational model to capture more meaning. ACM Trans. Database Syst. 4(4), 397–434 (1979)
5. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Boston, MA (2003)
6. Helman, P.: The Science of Database Management. Richard D. Irwin Inc., Burr Ride, IL (1994)
7. Holder, S., Buchan, J., MacDonell, S.G.: Towards a metrics suite for object-relational mappings. In: Model-Based Software and Data Integration, pp. 43–54. Springer (2008)
8. Ireland, C., Bowers, D., Newton, M., Waugh, K.: A classification of object-relational impedance mismatch. In: Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications. pp. 36–43. IEEE Computer Society (2009)
9. Khoshafian, S., Copeland, G.P.: Object identity. In: OOPSLA 1986. pp. 406–416 (1986)
10. Kowark, T., Hirschfeld, R., Haupt, M.: Object-relational mapping with SqueakSave. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 87–100. IWST '09, ACM (2009)
11. Link, S., Lukovic, I., Mogin, P.: Performance evaluation of natural and surrogate key database architectures. Tech. rep., Victoria University of Wellington, Wellington, NZ (2010)
12. Mork, P., Bernstein, P., Melnik, S.: Teaching a schema translator to produce O/R views. In: Conceptual Modeling - ER 2007, LNCS, vol. 4801, pp. 102–119. Springer (2007)
13. O'Neil, E.J.: Object/relational mapping 2008: Hibernate and the entity data model (EDM). In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. pp. 1351–1356. ACM (2008)
14. Wieringa, R., de Jonge, W.: Object identifiers, keys, and surrogates– object identifiers revisited. Theory and Practice of Object Systems 1(2), 101–114 (1995)