ReStore

Relational Database Persistency for Smalltalk Objects

Developers' Manual

https://github.com/rko281

Version History
2019-02-22: Initial version
2019-04-08: Added Pharo MySQL, ByteArray, Dictionaries, ReadStreams, Additional Functions

Contents

Introduction

ReStore is a framework for Dolphin Smalltalk and Pharo which enables objects to be stored in and read from relational databases (SQLite, PostgreSQL, MySQL etc.).

ReStore aims to make relational persistency as simple as possible, creating and maintaining the database structure itself and providing access to stored objects via familiar Smalltalk messages. This allows you to take advantage of the power and flexibility of relational storage with no specialist knowledge beyond the ability to install and configure your chosen database.

1. Getting Started

To install ReStore in your image follow the instructions on the GitHub project page for your Smalltalk dialect:

- Dolphin Smalltalk – https://github.com/rko281/ReStore
- Pharo – https://github.com/rko281/ReStoreForPharo

The class **SSWReStore** represents a ReStore session/connection; following installation a default singleton instance of SSWReStore is created and assigned to the global variable **ReStore**. We will use this global default throughout most of this document (see chapter 8 for information on working with multiple ReStore instances).

Choosing a Database

ReStore supports several different databases via the **SSWSQLDialect** class hierarchy. Currently defined SQL Dialects are:

- SQLite
- MySQL / MariaDB
- PostgreSQL
- SQL Server
- Access

Each subclass defines the different behavior, data types, functions etc. supported by a particular database. ReStore automatically selects the appropriate subclass after connecting to your chosen database; this ensures your application code is independent of database choice, enabling you to switch databases easily if required. For example, for simplicity and speed you may use SQLite during development, then deploy to PostgreSQL for better scalability.

Configuring ReStore

After choosing and installing your database you must tell ReStore how to connect to it; the method for doing this varies by Smalltalk dialect:

Dolphin Smalltalk

ReStore for Dolphin accesses databases via ODBC. You must first create a Data Source Name (DSN) using the driver for your chosen database via the ODBC control panel. Since Dolphin is a 32-bit application ensure that you use the 32-bit ODBC control panel – you can open this from your Dolphin image by evaluating

```
ReStore openODBC
```

Once the DSN is created you can configure ReStore to use it as follows:

```
ReStore dsn: 'MyDataSourceName'
```

Pharo

Pharo currently supports SQLite, MySQL and PostgreSQL. You must create the appropriate connection object then assign this to ReStore as follows:

```
"SQLite – see https://github.com/pharo-rdbms/Pharo-SQLite3 for more information"
ReStore connection: (SSWSQLite3Connection on: (Smalltalk imageDirectory / 'test.db') fullName)

"PostgreSQL – see https://github.com/svenvc/P3 for more information"
ReStore connection: (SSWP3Connection new url: 'psql://user:pwd@192.168.1.234:5432/database')

"MySQL – see https://github.com/pharo-rdbms/Pharo-MySQL for more information"
ReStore connection:
	(SSWMySQLConnection new
		connectionSpec:
			(MySQLDriverSpec new
				db: 'database'; host: '192.168.1.234'; port: 3306;
				user: 'user'; password: 'pwd';
				yourself);
		yourself)
```

Connecting and Disconnecting

Once you have configured ReStore for your chosen database you may connect to and disconnect from the database as follows:

```
ReStore connect.
ReStore disconnect.
```

2. Defining the Object Model

The first step in creating a ReStore application is to define your data model − the structure of your model classes. This allows ReStore to automatically create database rows from your objects, and also to create the actual database table in which those rows will exist.

Defining the structure of a class is done with the class method reStoreDefinition. This method should list the name of each persistent instance variable in the class, and define the type of object held in that instance variable. The 'type' of object will be a **class**, a **parameterized class**, or a **collection**. These different types are highlighted in the following example for a hypothetical **CustomerOrder** class**:**

reStoreDefinition

```
^super reStoreDefinition
        define: #orderDate as: Date;                                    "Class"
        define: #customer as: Customer;                                 "Class"
        define: #items as: (OrderedCollection of: CustomerOrderItem);   "Collection"
        define: #totalPrice as: (ScaledDecimal withPrecision: 8 scale: 2);  "Parameterized Class"
        yourself
```

2.1 Simple Classes

In the simplest case, just the class of object held is needed. Supported classes are:

- **Integer**
- **Float**
- **Boolean**
- **String**
- **Date**
- **Time**
- **DateAndTime**

Example for a hypothetical **Person** class:

```
define: #surname as: String;
define: #dateOfBirth as: Date;
define: #salary as: Float;
define: #isMarried as: Boolean;
```

Additionally, any other class defining a reStoreDefinition method may be used. This allows your classes to reference each other or even themselves:

```
define: #gender as: Gender;
define: #address as: Address;
define: #spouse as: Person;
```

2.2 Parameterized Classes

A parameterized class defines not only the class of an object but also additional information that may be required in relation to the class.

String

In Smalltalk an instance of String can be any size, with (to all intents) no upper bound. Within relational databases, however, there are usually three different types of Strings:

1. Fixed sized − usually referred to as a **CHAR**
2. Variable sized with some upper limit on the number of characters − this is usually referred to as a **VARCHAR**
3. An unbound, variable sized String − names vary; **LONGTEXT**, **TEXT**, **MEMO** etc.

For these reasons ReStore allows you to parameterize a String definition to enable the best choice of database type to be made. For Strings of a known, fixed number of characters (e.g. a postal/zip code, or a product code), you can specify a CHAR−type String using the String class method fixedSize:

```
define: #productCode as: (String fixedSize: 8);
```

For a variable sized String with a known maximum number of characters (VARCHAR) the method maxSize: is used:

```
define: #surname as: (String maxSize: 100);
```

Finally, if you just specify **String** (i.e. unparameterized) then a default value will be used as the maximum size of that String. This value will vary from database to database, but the net effect is usually to cause a LONGTEXT−type String to be used, although some databases may use an intermediate type with a large upper limit (e.g. MEDIUMTEXT). Example:

```
define: #notes as: String;
```

ByteArray

ReStore offers support for storing ByteArrays in a BLOB-type database column:

```
define: #imageData as: ByteArray;
```

Similar to with Strings, you may optionally specify a maximum size for the ByteArray – this will help ReStore choose the most appropriate BLOB type where the database offers multiple types with different (or no) maximum size:

```
define: #thumbnailImageData as: (ByteArray maxSize: 8192);
```

ScaledDecimal

Within Smalltalk an instance of ScaledDecimal has a **scale** − this defines the number of digits after the decimal point. In ReStore, when defining an instance variable as a ScaledDecimal as a minimum you must give the scale of that ScaledDecimal:

```
define: #totalPrice as: (ScaledDecimal withScale: 2);
```

Most relational databases support a type similar to ScaledDecimal (NUMERIC, DECIMAL etc.) but in addition to scale there is usually also **precision** − the total number of digits that may be held, including the scale. If you specify just a scale (as in the above example) a default precision of 15 will be used. Alternatively, you may specify the precision yourself:

```
define: #totalPrice as: (ScaledDecimal withPrecision: 8 scale: 2);
```

2.3 Unique IDs

Within ReStore every persistent object is automatically allocated an auto-incremented integer ID, unique to itself within its class. This happens completely transparently – you do not need to define this or store it within an instance variable of your class. However there may be times where you wish to access this unique ID from your application code, for example to use as a customer or order reference.

Where this is the case you can declare the corresponding instance variable in your class as follows:

```
defineAsID: #customerNo;
```

This defines the instance variable *customerNo* as an Integer that holds the unique ID of the object. You do not need to instantiate this value yourself – ReStore will automatically allocate the next available ID when the object is persisted. Should you wish to allocate the unique ID yourself however, you can simply assign it prior to storing the object and ReStore will use the assigned value instead. In this latter case it is up to your application code to ensure the ID remains unique.

2.4 Collections

Defining a Collection in ReStore is done by specifying an example (template) collection. Supported collection classes are:

- **OrderedCollection**
- **SortedCollection**
- **Array**
- **Set**
- **Dictionary**[1]

In addition to the collection class, the template collection must also specify the class of object held within that collection which can be either another persistent class, or a base or parameterized class. This is done through the Collection method **of:** (for convenience, this is defined as both a class and instance method). Example:

```
define: #middleNames as: (OrderedCollection of: (String maxSize: 100));
define: #friends as: (OrderedCollection of: Person);
```

For a SortedCollection, by default the implementation of <= in the referenced class is used to define the sort order. This can be changed by specifying a sort block:

```
define: #children as: ((SortedCollection sortBlock: [ :c1 :c2 | c1 age > c2 age]) of: Person);
```[2]

For an Array, since it is a fixed−size collection its size must be specified:

```
define: #parents as: ((Array new: 2) of: Person);
```

Note that collections in ReStore are homogenous, i.e. all elements must be of the same class.[3]

[1] See section 2.6 for information on Dictionaries
[2] See section 5.3 for limitations on ReStore sort block definitions
[3] Except in the case of inheritance – see 2.9

2.5 Owned Collections

In ReStore, collections specified as described above are termed **General Collections**. For readers with knowledge of relational databases they are equivalent to a many−to−many relationship. The relationships are stored using an intermediate table; ReStore takes care of creating this table and automatically adding/removing the mapping entries.

General Collections closely match the flexible nature of Smalltalk collections. Frequently however collections actually fulfill the following more limited criteria:

- each member object holds a reference to the owner of the collection
- each member object appears in the collection only once

In this case a more efficient form of collection can be used − an **Owned Collection**. This is equivalent to a one−to−many relationship in relational database terminology.

As an example, consider the hypothetical classes **Customer** and **CustomerOrder**. Each CustomerOrder knows its customer through its *customer* instance variable, defined as follows:

```
define: #customer as: Customer;
```

A Customer has a collection of CustomerOrders each of which appears only once. Thus the Customer's *orders* collection can be specified as:

```
define: #orders as: (OrderedCollection of: CustomerOrder owner: #customer);
```

This defines the Customer's *orders* instance variable as an owned collection of CustomerOrders, with the CustomerOrder instance variable *customer* holding a reference to the owning Customer.

Why use an owned collection? Owned collections are usually quicker to store and retrieve from the database compared to general collections since there is no intermediate mapping table – the relationship is defined solely by the instance variables of the two objects.

2.6 Dictionaries

The main difference between Dictionaries and the other collection classes detailed previously is that each element of a Dictionary is essentially storing two objects - the **key** and the **value** – so the specification of a Dictionary must define the class of both objects. This is done by providing an Association between the key class and the value class as the parameter to of:

```
define: #profitByDate as: (Dictionary of: Date -> Float);

define: #productQuantities as: (Dictionary of: Product -> Integer);
```

Like other collections, the class of elements for both key and value can be any other persistent class, and will be the same for all elements of that collection (except in the case of inheritance).

Cache Dictionaries

For reasons of logic or efficiency you may have a Dictionary where the key is some attribute of the value. For example, say you have a **Company** object - this has a potentially large collection of addresses of individual offices:

Company	**Offices**
Widgets International, Inc.	Schupstraat 24, **Antwerp** ... The Widgets Centre, **Zagreb**
Transdiffussion Worldwide	Westerstraat 93, **Amsterdam** ... Transdifussion House, **Zurich**

Your application has a requirement to quickly find the Office in a particular city - hence you would like to store the collection of CompanyOffices as a Dictionary mapping city name to actual CompanyOffice. Since this arrangement is effectively a lookup cache, ReStore terms such structures **Cache Dictionaries**.

In this scenario ReStore offers an optimised form of Dictionary which will read a collection of objects from the database and automatically assemble a cache ready for fast and efficient lookup in your application code. To specify such a Dictionary, the key of the Association which is the parameter to of: should be the name of the instance variable of the value class which holds the key object, for example:

```
define: #cityOffices as: (Dictionary of: #city -> CompanyOffice);
```

Cache dictionaries can also take advantage of any owner link in the target object in the same way as other owned collections:

```
define: #cityOffices as: (Dictionary of: #city -> CompanyOffice owner: #company);
```

Multi-Value Cache Dictionaries

In the above Company-Office Dictionary example it is implicit that there can only be one CompanyOffice per city. If this is not guaranteed to be the case we need to use a different cache implementation. A common approach to a scenario like this is to use a Dictionary with a collection of objects as its values, often implemented like this:

```
addOffice: aCompanyOffice

    ^(self cityOffices at: aCompanyOffice city ifAbsentPut: [OrderedCollection new]) add: aCompanyOffice


removeOffice: aCompanyOffice

    ^self cityOffices at: aCompanyOffice city ifPresent: [ :offices | offices remove: aCompanyOffice]
```

ReStore terms such structures **Multi-Value Cache Dictionaries** and allows you to specify them in your class's reStoreDefinition as follows:

```
    define: #cityOffices as: (Dictionary of: #city -> (OrderedCollection of: CompanyOffice));
```

Again, an owner link can be included in the specification:

```
    define: #cityOffices as: (Dictionary of: #city -> (OrderedCollection of: CompanyOffice) owner: #company);
```

By specifying a cache dictionary or multi-value cache dictionary, ReStore will read a collection of objects from the database and automatically assemble a cache dictionary ready for efficient and fast lookup in your application code.

Derived Keys

In addition to defining a cache based on an instance variable (as in the above examples), ReStore also allows you to specify a Block which derives the key from the value object. For example, if city isn't held directly by the CompanyOffice object but instead within its address instance variable you could define the cache as follows:

```
    define: #cityOffices as: (Dictionary of: [ :office | office address city] -> CompanyOffice);
```

Blocks can also be used with multi-value and owned Dictionary specs, for example:

```
define: #cityOffices as:
    (Dictionary
        of: [ :office | office address city] -> (OrderedCollection of: CompanyOffice)
        owner: #company);
```

2.7 Dependent Relationships

When defining your object model, you may notice subtle differences in the relationships between objects. As an example, consider the following partial specification of class **Person**:

```
define: #address as: Address;
define: #gender as: Gender;
```

At first glance the relationships described here are the same − each instance of **Person** will hold an instance of **Address** in its *address* instance variable, and an instance of **Gender** in *gender*. However there is a difference − an instance of Gender will likely be referenced by many different Person objects, whereas the object held in *address* will only belong to one individual Person.

The significance of this is as follows: when the contents of a Person's *address* instance variable is overwritten (by another instance of Address, or nil) the replaced Address object is no longer referenced from any other persistent object, and so should be removed from the database (deleted). Further, when an instance of Person is itself removed from the database, then its *address* object should also be deleted for the same reason. On the other hand, since the object held in *gender* is shared by other objects it should not be deleted in either of these cases.

In ReStore, relationships such as that between a Person and its Address are termed **dependent relationships** − the object held in *address* is **dependent on** the owning instance of Person for its existence. By explicitly declaring this dependency ReStore knows to delete the instance of Address when overwritten, or when the owning Person instance is deleted.

Declaring a dependent relationship is done by sending the message dependent to the class of dependent object in the class specification method:

```
define: #address as: Address dependent;
define: #gender as: Gender;
```

Collection−based relationships may also be dependent. Returning to our **CustomerOrder** example class, this contains an owned collection of **CustomerOrderItem** instances which define the component parts of the order (product, quantity etc.). This collection can also be declared dependent by sending the message dependent to the class of dependent object in the collection definition:

```
define: #items as: (OrderedCollection of: CustomerOrderItem dependent owner: #order);
```

2.8 Inlined Classes

We have previously seen ReStore instance variable definitions that reference other persistent model classes, for example:

```
define: #address as: Address dependent;
```

Within the database, an instance of **Person** will be stored in the **`person`** table, with a reference to its instance of **Address**, which will be stored separately in the **`address`** table.

This is a fairly standard scenario, however it does mean that to fetch a Person and its Address from the database requires two read operations (one for the Person and one for the Address). This is probably not an issue if you do not always require a person's address (indeed it may be an advantage, as it avoids fetching data that isn't required), however there may be cases where this is not optimal for your application.

Where this is the case you can ask ReStore to store the instance of the referenced class within the database row of the owning object. This is termed *inlining* and is done using the method **inlined**:

```
define: #address as: Address inlined;
```

With the *address* instance variable defined as inlined, the database representation of the Address object will be stored as part of the owning Person object, directly within the **`person`** table. This means only one read operation is necessary to fetch a Person and its Address.

Note that since a Person's *address* is now stored within the Person record in the database, it is automatically dependent so there is no need to declare this separately.

2.9 Inheritance

At the start of this chapter, it was mentioned that defining a class for ReStore enables a database table to be automatically constructed for that class. With a hierarchy of related classes it is necessary to decide whether these should share a single database table, or if each class exist in its own individual table.

By default, a hierarchy of classes will share a single table – this opens up some important possibilities.

Firstly, if you define an instance variable or collection in another class as holding instances of the superclass of a hierarchy then that instance variable/collection can hold an instance of the superclass or any of the subclasses sharing the superclass table. To illustrate this we will return to the CustomerOrderItem class introduced in the previous section. Let's say this contains a *product* instance variable defined as follows:

```
define: #product as: Product;
```

Product is the superclass of a number of classes representing different product types:

Product	**-** *productCode, supplier, stockLevel*
Book	*- author, title, isbn, publisher*
MusicProduct	*- artist, title, catNo*
Vinyl	*- vinylWeight*
CD	

If Product and its subclasses share a table then CustomerOrderItem's *product* instance variable can hold an instance of Book, CD, Vinyl or any other subclass of Product.

A second advantage is that when you need to query for objects in the database, a query for instances of the superclass of the hierarchy will also find instances of any subclasses sharing that table:

```
"Find all Products from a particular supplier. Uses ReStore querying methods – see X.X"
Product storedInstances select: [ :item | item supplier = aSupplier]
```

This will find instances of Book, CD, Vinyl or any other subclass of Product supplied by aSupplier.

Disadvantages of Sharing a Table

When constructing the single shared table for a hierarchy of classes, ReStore will allocate a column for every non−collection instance variable in each class. Thus, when for example an instance of CD is stored in the table, the columns corresponding to *author, isbn, publisher* (Book instance variables) and *vinylWeight* (Vinyl) will be empty. This is effectively wasted space in the table.

If there are large differences in the instance variables held by different subclasses within a hierarchy you may want to store instances of the subclasses in different tables to avoid this waste of space. This does mean that the advantages of table sharing listed above are lost. However, if there *are* large differences between the classes in the hierarchy then it is likely that they are not related in a meaningful way and so the loss of these advantages may not be important.

A common situation where this is the case is where there is one abstract superclass, defining a few common attributes for all model object classes in a particular application:

MyModel	**-** *description, dateCreated*
Person	*- firstName, surname, address*
Address	*- line1, postcode*

In a case like this it is highly unlikely that you would want instances of Person and Address to share a table. To turn off the default behavior of hierarchies sharing a table, MyModel should implement the following class method:

```
shouldSubclassesInheritPersistency

    ^false
```

Should you wish to turn table sharing 'on' again in a sub−hierarchy, you would simply override this method to return **true**.

Individual subclasses may also choose to 'opt out' of sharing a table by implementing the following method:

```
shouldInheritPersistency

    ^false
```

Note, however, that a subclass cannot 'opt in' to table sharing where it has been specifically turned off by a superclass implementation of shouldSubclassesInheritPersistency.

2.10 Creating and Maintaining the Database

Once you have successfully defined the classes forming your object model, you are ready to begin working with ReStore.

Let's assume you have connected the global ReStore instance as described in section 1. You then need to tell it which classes are to be persistent − these are the classes for which you have defined reStoreDefinition methods. To do this, you use the methods **addClass:** or to add a whole hierarchy of classes, **addClassWithSubclasses:**

```
ReStore
    addClass: Customer;
    addClass: CustomerOrder;
    addClass: CustomerOrderItem
    addClassWithSubclasses: Product
```

You now have a ReStore object, connected to the database and containing all required classes. However, none of the corresponding tables exist in the database. Fortunately, ReStore can create all the tables for you with one simple instruction:

```
ReStore synchronizeAllClasses
```

Maintaining Tables

One of the advantages of Smalltalk is its highly interactive nature, which allows you to rapidly develop and refine applications. Over time your classes are likely to change as you redevelop and refactor your code. Unfortunately, this evolutionary development process can leave stored data (in files, or a relational or object database) in an incompatible state, requiring manual intervention or additional coding to bring it up to date with your current object model.

ReStore helps overcome these problems and preserve the rapid development benefits of Smalltalk by automatically reconfiguring your database tables to match your object model. When you have made changes to your object classes, simply re−add them to ReStore (as above) and re−evaluate:

```
ReStore synchronizeAllClasses
```

Resynchronizing in this way allows ReStore to create new tables (for new classes), add new columns (for new instance variables) and remove redundant columns (where you have removed instance variables).

Renaming a Class

A change that ReStore cannot handle automatically with synchronizeAllClasses is where you have renamed a class. In this case, if you were to use synchronizeAllClasses, ReStore would simply add an empty table with a name based on the new name of the class, leaving the old table (and its data) in the database, but inaccessible.

To overcome this, ReStore allows you to explicitly state when you have renamed a class. As an example, let's say you renamed the class **Address** to **PostalAddress**:

```
ReStore renamedClass: Address from: #PostalAddress
```

Using this technique, the original table for Address would simply be renamed to match PostalAddress, preserving all its data.

Removing Classes

Similar to renaming a class, you must use a specific message to tell ReStore that you no longer require a particular class in your data model (and that its corresponding table can be removed from the database):

```
ReStore destroyClass: <redundant class>
```

In more drastic situations (e.g. to purge all data from a database), you may evaluate:

```
ReStore destroyAllClasses
```

Note that this will only remove from the database tables associated with classes known to ReStore (i.e. those that have been added with addClass: or addClassWithSubclasses:).

Renaming an Instance Variable

A further change that cannot by handled directly via synchronizeAllClasses is where an instance variable has been renamed. Using synchronizeAllClasses in this case would cause ReStore to add a new (empty) column for the new instance variable, and delete the previous column, with the loss of all data contained in that column.

Similar to renaming a class, ReStore provides a simple message which can be used to inform it of the change of instance variable name and instruct it to update the database structure accordingly. Let's say you have renamed the **Customer** instance variable *firstName* to *forename*; you would inform ReStore of this change as follows:

```
ReStore renamedInstVar: #forename from: #firstName in: Person
```

This method will prompt ReStore to update the table associated with Person, renaming the column previously corresponding to *firstName* so it matches *forename*.

3. Storing Objects – Transactions

You have successfully completed your class specifications, connected ReStore, and constructed your tables. You are now ready to create and store persistent objects.

One of the simplest ways to do this is to allow ReStore to keep track of your persistent objects by wrapping all changes in a **Transaction**. Transactions are a way of packaging together a batch of changes to persistent objects. During the transaction there is no actual interaction with the database, all changes are purely within your image. At the end of the transaction, you may **commit** any changes to the database, at which point they become persistent. Alternatively, changes may be **rolled back**, leaving the database unchanged and the objects in memory in their original state, prior to the start of the transaction.

An important trait of transactions is that they are **atomic** – if the transaction is successfully committed, all changes made within the transaction will be persisted to the database. However if the commit fails, no changes are written to the database[4]. This prevents related changes being only partly committed, which could leave the database in an inconsistent state.

[4] Refer to chapter 7 for reasons why transactions may fail and how to handle this

3.1 Storing a New Object

When you first create an object in your image it is non-persistent, i.e. it does not exist in the database. You need to explicitly ask ReStore to persist the new object – this is done via the message **store**

```
"Store a new object in the database – first begin a new transaction"
ReStore beginTransaction.

"Create the object"
johnSmith := Customer new firstName: 'John'; surname: 'Smith'; yourself.

"Ask ReStore to store johnSmith in the database – nothing is written to the database at this point"
johnSmith store.

"Commit the transaction – johnSmith will now be stored in the database:"
ReStore commitTransaction
```

Automatic Storing of Referenced Objects - 1

An important point to note is that, where a stored object references other non-persistent objects, these will automatically be stored along with the referring object. For example, if we modify the above example to also create an Address object for the new customer:

```
ReStore beginTransaction.

johnSmith := Customer new firstName: 'John'; surname: 'Smith'; yourself.
johnSmith address: (Address new line1: '123 Some Street'; yourself).
johnSmith store.

ReStore commitTransaction
```

Since the new Address object is referenced from the stored Customer object, the Address object will automatically also be persisted – there is no need to explicitly store this object. This applies whenever a stored object references other non-persistent objects.

3.2 Updating a Persistent Object

Once you have a persistent object you may want to update its database representation with changes made in your image. Provided you make these changes within a transaction then there is nothing additional you need to do other than commit the transaction at the end – ReStore will automatically detect all changes made to persistent objects:

```
"First being a new transaction"
ReStore beginTransaction.

"Update the persistent object – ReStore will automatically add any changes to the active transaction"
johnSmith dateOfBith: (Date fromString: '01/02/1990').
johnSmith addOrder: CustomerOrder new.

"Commit – all changes made since the start of the transaction are automatically stored"
ReStore commitTransaction
```

Automatic Storing of Referenced Objects - 2

Something to note in the above example is the addition of a new CustomerOrder:

```
johnSmith addOrder: CustomerOrder new.
```

Similar to the Address object in the previous section, the new CustomerOrder object will automatically be stored in the database when the transaction commits, since it is referenced from the already-persistent *johnSmith* object. Again, there is no need to explicitly store this object.

Block Transactions

As a more convenient alternative to a beginTransaction... commitTransaction sequence, it is possible to use the message evaluateAsTransaction: and make all your changes within a block:

```
"Equivalent to the above using a block"
ReStore evaluateAsTransaction:
    [johnSmith
        dateOfBith: (Date fromString: '01/02/1990');
        addOrder: CustomerOrder new]
```

3.3 Reverting Changes

ReStore recognizes changes to a persistent object by storing a copy of the object prior to those changes being made. Providing those changes are made within a transaction then the changes can be automatically reverted by **rolling back** the transaction.

This can be useful within application code by providing an easy way to implement “Undo” functionality:

```
"Selectively commit or rollback changes to a persistent object"
ReStore beginTransaction.
johnSmith surname: 'Smythe'.

(Object confirm: 'Save Changes?')
    ifTrue: [ReStore commitTransaction "johnSmith is now named 'John Smythe' in the database"]
    ifFalse: [ReStore rollbackTransaction "johnSmith's surname in the image is reverted to 'Smith' "]
```

In a similar way it is also possible to easily implement a dialog with OK/Cancel functionality without any additional code – simply begin a transaction when opening the dialog, then either commit or rollback depending on whether the user clicks OK or Cancel.

3.4 Deleting Persistent Objects

Finally, you may also wish to delete persistent objects from the database. This is done with the message **unstore**:

```
"Delete an object from the database"
ReStore evaluateAsTransaction: [johnSmith unstore].
```

Automatic Deletion of Dependent Objects

Similar to the automatic storing of referenced objects discussed earlier, referenced objects may also automatically be unstored when their referring object is unstored. However this is subject to whether the reStoreDefinition of the referring object defines the relationship as **dependent** (see section 2.7).

Assuming the *address* and *orders* definitions of Customer are dependent:

```
define: #address as: Address dependent;
define: #orders as: (OrderedCollection of: CustomerOrder owner: #customer);
```

...then the Address object and all CustomerOrder objects referred to by *johnSmith* will also be deleted when *johnSmith* is deleted.

4. Storing Objects - Manually

As we saw in the previous chapter, when a transaction is active (following a beginTransaction) ReStore will track changes to persistent objects and add those objects to the active transaction. All changes are then committed (or rolled back) in a single step.

This automatic tracking of changes means that no special actions are required to mark an object as changed, and so your application does not need to consider which objects it is likely to change. Thus fully persistent applications are quick and easy to construct.

In some circumstances however you may want to take more control of the updating of your persistent objects. One scenario where this is desirable is where you prefer to structure your application as a number of *mode-less* screens. By mode-less, we mean that the user is free to switch between different screens at will, as opposed to *modal* screens, which restrict the user to working on one screen at once.

If each mode-less screen has its own 'Save' or 'Apply Changes' button then the single transaction model is not appropriate - sending commitTransaction will commit all changes made, regardless of from which screen they originated.

What is required is for each screen to manually notify ReStore of which objects it has changed (or is likely to have changed). This chapter discusses the mechanisms ReStore provides to accomplish this.

4.1 Storing an Object

ReStore allows you to explicitly ask an object to store itself (if not yet persistent) or to store any changes made to itself (if already persistent) by simply sending it the message **store**. This effectively begins a transaction, adds the receiver object to the transaction, and then finally commits the transaction. Thus the following examples are equivalent:

```
"Storing a new object - with transaction:"
ReStore evaluateAsTransaction:
        [(Customer new firstName: 'John'; surname: 'Smith'; yourself) store].

"Storing a new object - without transaction:"
(Customer new firstName: 'John'; surname: 'Smith'; yourself) store.


"Updating an existing object - with transaction:"
ReStore evaluateAsTransaction: [johnSmith middleName: 'David'.].

"Updating an existing object - without transaction:"
johnSmith middleName: 'David'; store.
```

Note that the same rules on the automatic storing of referenced non-persistent objects apply here as when using transactions.

unstore

Similar to store, you may send **unstore** to explicitly an object to ask it to delete itself from the database.

```
"Deleting an object - with transaction:"
ReStore evaluateAsTransaction: [johnSmith unstore].

"Deleting an object - without transaction:"
johnSmith unstore.
```

Again, the rules on the automatic unstore of dependent objects apply.

Note that **store** and **unstore** are the same messages that are used to tell ReStore to persist new objects and delete existing objects within a transaction (see 3.1 and 3.4). In the transaction use-case they do not take effect immediately but add their change to the current transaction, whereas in the non-transaction use-case they act immediately. Thus their behaviour changes depending on whether they are used whilst a transaction is active or not.

4.2 Rollback and Refresh

Similar to when working with transactions you may rollback any changes made to a persistent object; this is done by simply sending it the message **rollback**

```
johnSmith rollback
```

This will revert the object to its state prior to the changes being made. Similar to rollbackTransaction you can use this to implement an easy “Undo” or “Cancel Changes” function.

Object Refresh

rollback only reverts the object back to its previous state within the current image.

In a multi-user scenario it is also possible that another user may have changed the object within the database. If you wish to bring the object up-to-date with its current state in the database you can send it the message **refresh**

```
johnSmith refresh
```

Note that refresh effectively performs a rollback for objects which have not been updated in the database (compared to your image).

If your application has multiple concurrent users you may wish to use refresh prior to displaying an object to ensure the user sees the latest version.

4.3 Checking for Changes

When working with persistent objects it can be useful to know whether the user has made any changes compared to the version of the object originally fetched from the database. For example you may want to selectively enable an “Apply Changes” button depending on whether any changes have actually been made.

You could track when a change is made in your own code, but ReStore enables you to do this automatically by asking a persistent object if it **hasChanged**. Example:

```
johnSmith hasChanged. "false"

johnSmith middleName: 'David'.
johnSmith hasChanged. "true"

johnSmith store.
johnSmith hasChanged. "false"
```

Note that hasChanged only compares the current state of the object to the last version fetched from or stored in the database by your image. It does not compare it with the current version in the database, where it may have been updated by another user.

4.4 Dependent Objects

A complication with the manual way of managing changes (when compared to using transactions) is tracking all objects that may be changed by a particular area of code. As an extreme example let's consider an application where a customer can edit any of their own details plus their active orders. Thus we have the following potentially changed objects:

- a Customer
- their *address* object
- their *customerOrders* collection
- each element of their *customerOrders* collection
- each CustomerOrder's *items* collection
- each element of each CustomerOrder's *items* collection

As you can see this quickly becomes a complicated set of objects to keep track of. Fortunately ReStore can help avoid the need for your code to explicitly store each of these objects by using information from the objects' reStoreDefinition.

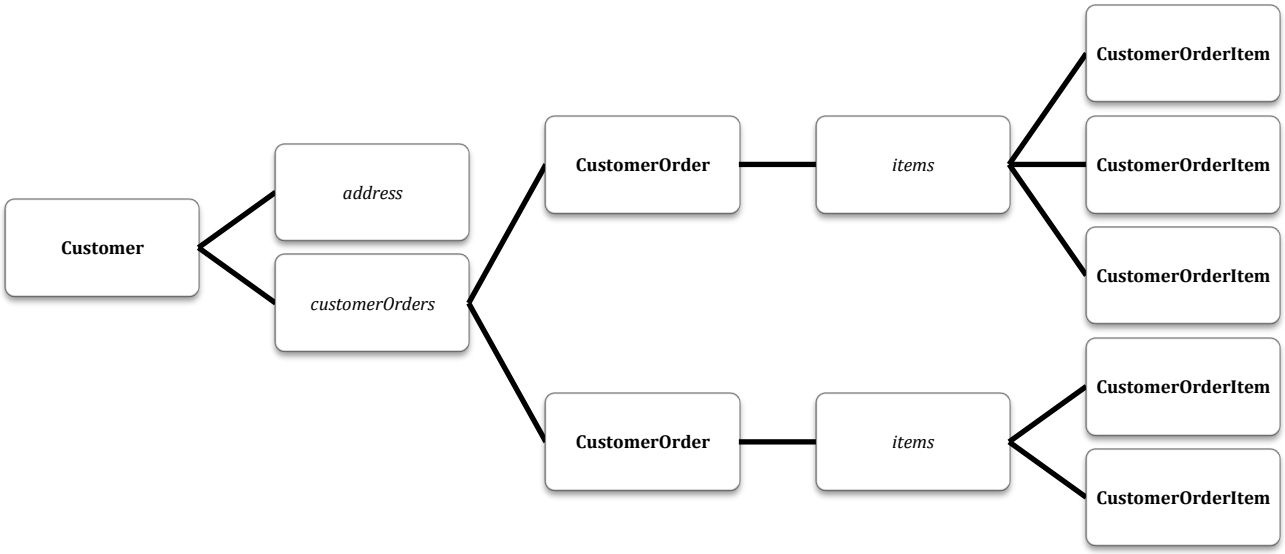
In section 2.7 we considered the concept of **dependent** relationships between objects and stated that one advantage of defining these is that dependents of an object are automatically deleted from the database when no longer referenced from that object. A further advantage is that dependent objects are also stored when their owning object is stored. So, assuming **Customer** is defined as follows:

```
define: #address as: Address dependent;
define: #customerOrders as: (OrderedCollection of: CustomerOrder dependent owner: #customer);
```

...and if CustomerOrder is defined:

```
define: #items as: (OrderedCollection of: CustomerOrderItem dependent owner: #customer);
```

This gives us a hierarchy of dependent relationships that can be visualized as follows:



With the dependent relationships defined in this way, the application only needs to send store to the Customer object for any changes to all of its dependent objects (and their dependents, recursively) to be stored in the database.

Additionally, the other messages discussed in this chapter (refresh, rollback, hasChanged) also apply to dependent objects in the same way as store, giving the same simplicity advantages.

4.5 Working with Multiple Objects

The examples so far in this chapter have all involved working with individual objects (potentially with their set of dependent objects).

Depending on your application you may want to send store, refresh etc. to a collection of objects at once. You could repeatedly send store to each object in turn, though in addition to being long-winded this will create an individual transaction for each invocation of store. If one of these transactions fails this could potentially leave your database in an inconsistent state, with some changes committed and others not.

Fortunately all the messages discussed in this chapter have variants that enable you to work with collections of objects; this wraps each operation in a single overall transaction which will either succeed or fail in its entirety. These messages are defined in Collection and are the same as the messages for individual objects, appended with **All**:

- **storeAll**
- **unstoreAll**
- **rollbackAll**
- **refreshAll**

You may also check if any member of a collection has been changed:

- **hasAnyChanged**

To use these, simply assemble a collection of the objects you are working with then send the appropriate message:

```
(Array with: customer1 with: customer2 with: customer3) storeAll.
```

5. Querying – Introduction

Once you have started storing objects, you need some way to find and bring them back into your image. ReStore provides a sophisticated querying mechanism based around the semantics of standard Smalltalk collection enumeration methods (select:, detect: etc.).

Stored Instances

If you want to obtain all in-memory instances of a class in your image, you send the message allInstances to that class. ReStore provides a similar message - **storedInstances** - that answers a collection-like object representing all instances of a class that exist in the database.

Note this is a not a real collection – no objects are fetched from the database by sending storedInstances. Instead this virtual collection is a starting point for actual querying using enumeration methods (select:, detect: etc.) in almost exactly the same manner as a regular collection. The block arguments to the enumeration methods are not executed in the image but translated to SQL and run directly in the database, transferring computation to the database server and so minimizing the amount of data transferred.

Thus querying your database is efficient (no objects are actually fetched from the database until actually required) and as simple as using regular Smalltalk collection methods. In addition you can use much of the same code for both real Smalltalk collections and virtual, database-resident collections.

Object Identity

Once you have a persistent object in memory then you may traverse its references to other persistent objects exactly as you would with a regular non-persistent object. These linked objects are automatically and transparently fetched from the database the first time they are sent a message.

Note that ReStore will always maintain the **identity** of objects returned from the database − that is, if you have a persistent object in memory, and that object is also included in the results of a subsequent query, or is referenced from another persistent object, then the existing in−memory object will be used. This also means that cyclical structures − Object A references Object B references Object A − are not a problem.

5.1 Enumerating Stored Instances

The following enumeration methods work in the same way as their Collection counterparts:

- **select:**
- **reject:**
- **detect:**
- **detect:ifNone:**
- **collect:**[5]
- **select:thenCollect:**
- **allSatisfy:**
- **anySatisfy:**

Examples:

```
"Obtain a virtual collection representing all Customers in the database"
allCustomers := Customer storedInstances.

"Retrieve all Customers with the surname Smith"
allCustomers select: [ :each | each surname = 'Smith'].

"Find Customer number 12345"
allCustomers detect: [ :each | each customerNo = 12345].

"Do we have any Customers in London?"
allCustomers anySatisfy: [ :each | each address city = 'London'].

"Who has placed an order today?"
allCustomers select: [ :each | each customerOrders anySatisfy: [ :order | order date = Date today]].
```

To re-emphasize, no objects are actually fetched from the database until actually required. Only the second, third and last examples above will actually fetch objects from the database, and then only the ones matching the query block. This ensures that the absolute minimum of data is transferred from the database, making querying quick and efficient.

[5] See 6.4 for information on the use of collect:

5.2 Enumeration Blocks

The blocks passed to enumeration methods (select: etc.) are not actually evaluated with each object in the database; instead the block is analyzed by ReStore and converted into an equivalent SQL query. This means there are certain limitations on the messages that can be sent within the block.

Instance Variable Accessors – Single Objects

Any instance variable of the class holding a single (non-collection) object may be used via its equivalently-named accessor message. Where the instance variable holds another persistent object then that object's instance variable accessors may then be used (recursively). The following previous examples demonstrate this:

```
allCustomers select: [ :each | each surname = 'Smith'].
allCustomers detect: [ :each | each customerNo = 12345].
allCustomers anySatisfy: [ :each | each address city = 'London'].
```

Instance Variable Accessors – Collection Objects

Any instance variable of the class holding a collection object may also be used via its equivalently-named accessor message. Subsequent messages should make sense to a collection; examples:

```
allCustomers reject: [ :each | each customerOrders isEmpty].
allCustomers select: [ :each | each customerOrders size > 10].
allCustomers select: [ :each | each customerOrders anySatisfy: [ :order | order date = Date today]].
```

Other Local Methods

In addition to a class's accessor methods you may also refer to other methods defined by the class which themselves obey the same rules as query blocks. For example, if Customer defines the following method:

```
fullName

    ^self firstName, ' ', self surname
```

...then the following is a valid enumeration block:

```
allCustomers select: [ :each | each fullName = 'John Smith'].
```

Condition Messages

The following condition messages may be used:

- =, ~=
- <,<=, >, >=
- between:and:
- isNil, notNil
- isEmpty
- match:
- includes:

Logical Operators

Conditions may be combined using the logical operators AND and OR, for example:

```
"Do we have any Smiths in London?"
allCustomers anySatisfy: [ :each | each surname = 'Smith' and: [each address city = 'London']]
```
[6]

Messages and Functions

When dealing with a basic object (String, Number, Date etc.) within an enumeration block it is possible to send certain standard Smalltalk messages to these objects – these are translated into SQL functions that are then executed by the database. Translating messages to functions in this way transfers work from the Smalltalk image to the database, reducing the amount of data transferred and making querying more efficient.

The actual messages supported vary from database to database depending on the functions supported but commonly include the following:

String messages

- ,
- size
- asUppercase
- asLowercase
- trimBlanks

Numeric messages

- Mathematical operators (+, -, *, /, \\ etc.)
- asInteger
- abs

See definitions of commonFunctions and dialectSpecificFunctions in the **SSWSQLDialect** hierarchy for further examples. You may also add your own function translations (see section 6.6).

[6] Note that ReStore for Pharo currently cannot always translate block-style logical operations (and:, or:) correctly. You are recommended to use & and | instead, e.g. (each surname = 'Smith') & (each address city = 'London')

5.3 Sorting

As with a standard SortedCollection it is possible to define a sort order for a storedInstances collection via the use of a **sort block** - a two-argument block defining the relative ordering of two instances. A sort block is defined on a storedInstances collection by sending the message **sortBlock:** - once defined, all database activity against that collection will be ordered by use of equivalent SQL ORDER directives. For example, the following sort block:

```
[ :p1 :p2 | p1 surname <= p2 surname]
```

...translates to the following SQL

```
ORDER BY SURNAME ASC
```

Limitations on Sort Blocks

As with enumeration blocks, sort blocks used within ReStore are subject to certain limitations. Firstly, the sorting must be defined only in terms of the instance variables of the class of object held by the collection. Secondly, the sort block must be defined **logically**, not **procedurally**. This is not normally a concern where the sorting is defined on a single attribute (as with the surname example above) but is important where more than one attribute is used.

As an example, consider the following way of defining a sort order on (firstly) the *surname* of a Customer object, and then (if surnames are equal) on the *firstName*:

```
[ :p1 :p2 |
(p1 surname = p2 surname)
    ifTrue: [p1 firstName <= p2 firstName]
    ifFalse: [p1 surname < p2 surname]]
```

This is a **procedural** definition, in that a flow of program control is defined - firstly an = test is applied, then, depending on the result, either a *firstName* or surname comparison. Such a sort block is not suitable for use with an storedInstances collection - it should be redefined **logically** as follows:

```
[ :p1 :p2 |
(p1 surname < p2 surname) |
    ((p1 surname = p2 surname) & (p1 firstName <= p2 firstName))]
```

In this implementation, the comparison is defined purely in terms of <, = , & (logical AND) and | (logical OR). ReStore is able to translate such a sort block into the equivalent SQL order directives.

Note these limitations also apply to the use of SortedCollections in a class definition (see 2.4) and to the implementation of <= in a persistent class where such an implementation is used to define the sort order.

5.4 Other Collection Messages

In addition to enumeration messages certain other standard Collection messages may be sent to a storedInstances virtual collection.

asOrderedCollection

By sending asOrderedCollection you can convert an storedInstances collection into a regular OrderedCollection. An advantage of this is that the limitations on enumeration blocks no longer apply − you are now dealing with a real Smalltalk collection.

Implicit in this is that all objects forming the collection are fetched from the database into your Smalltalk image; if this is a large number then asOrderedCollection may take a while to execute. Together with the overhead of allocating the memory for a significant number of objects, asOrderedCollection should be used with care.

asSortedCollection

Similar to asOrderedCollection, asSortedCollection and asSortedCollection: can convert a storedInstances collection into a regular SortedCollection.

If the storedInstances collection already has a sort defined (via sortBlock:), then asSortedCollection will use that sort block, otherwise the default sort for the class of object in the collection will be used (i.e. its implementation of <=). asSortedCollection:, like the standard Smalltalk implementation, takes the sort block to use as its parameter.

As with sortBlock:, the sort block or implementation of <= must conform to the rules for use with ReStore (see 5.3).

size

As with a standard collection size will return the number of elements of an storedInstances collection. However, rather than fetching the entire collection from the database this is done by issuing a 'count' query whose sole result is the size of the collection, thus the overhead is minimal. It can be a good idea to use size before asOrderedCollection to check that the collection to be fetched is not unreasonably large.

isEmpty

isEmpty is provided for convenience. This simply uses size to test for a zero-element collection, and hence has the same performance benefits.

first, last

These messages will fetch just the first or last object in the storedInstances collection without needing to fetch all members of the collection. You may also use the related parameterized first: and last: messages to fetch the first or last “N” elements. Like a regular collection, these messages will raise an error if the collection has no members.

5.5 Query by Example

ReStore also supports an alternative way of querying which can be used in addition to the enumeration-based methods discussed so far. This is termed **Query by Example**.

Querying by example is accomplished by creating a template object which is 'similar to' the instance(s) you wish to find in the database. By 'similar to', we mean that the template object has certain of its instance variables filled, and 'similar' instances will have the same instance variables filled with the same values.

Once a template object has been created, sending it the message **similarInstances** will result in a storedInstances collection of matching objects from the database. This collection can then be used in the same ways discussed earlier (e.g. asOrderedCollection to return an actual Smalltalk collection, satisfying: to further refine etc.). Example:

```
template := Customer new.
template surname: 'Smith'.
allSmiths := template similarInstances.

template address city: 'London'.
allSmithsInLondon := template similarInstances.
```

Query by example is particularly useful when developing a 'Find'-type user interface. The model of the screen would be the template object; as the user fills in the details to be found, the template object is also filled in. When the user hits 'Find', it is only necessary to send similarInstances to the template object to find the required instances.

Wildcards

One potential disadvantage with the use of template objects is that they require a 'hard' match - e.g. in the above example someone named 'Smithe' would not be included in the resulting collection.

In a block-based query, it is possible to use the match: message to look for a partial match such as this (e.g. 'Smith*' match: surname). To check for such partial matches when querying by example, it is necessary to send the message **asWildcard** to the String denoting a partial match:

```
template := Customer new.
template surname: 'Smith*' asWildcard.
allSmithsEtc := template similarInstances.
```

Requiring nil

Query by example relies on only the aspects of interest being set in the template object - any attribute which is unset will be **nil**, and is thus ignored when determining the similar instances.

Occasionally it can be useful for an unset attribute to be a requirement of the similar objects. For example, to determine any unsent CustomerOrder objects it may be necessary to check for instances with a *despatchDate* of nil. Such a requirement can be expressed by sending the message **required** to nil:

```
template := CustomerOrder new.
template despatchDate: nil required.
allUnsentOrders := template similarInstances.
```

6 Querying – Advanced

This chapter continues discussing querying in ReStore, with an emphasis on more advanced topics and techniques.

6.1 Refining a Query

Sometimes you may wish to progressively refine a storedInstances collection without actually requiring the member objects of that collection to be in memory. This may be due to the structure of your user interface, or possibly to avoid one large enumeration block containing many separate conditions logically ANDed together. Refining a query in this way can be done easily and efficiently using the method **satisfying:**

satisfying: performs in a similar way to select:. However, instead of returning a real collection of objects matching the discriminator block (which would entail fetching all those objects from the database) satisfying: returns another storedInstances collection. You may then select:, reject: etc. against this block, query its size or isEmpty etc., or alternatively refine the collection further with further satisfying: messages. Example:

```
allSmiths := Customer storedInstances satisfying: [ :each | each surname = 'Smith'].
allJohnSmiths := allSmiths satisfying: [ :each | each firstName = 'John'].
specificJohnSmith := allJohnSmiths detect: [ :each | each dateOfBirth = aDate]
```

The important point to grasp is that no objects are fetched from the database when using satisfying: – this makes it very efficient. In the above example, if select: were used instead of satisfying: possibly several thousand Customer instances would have been fetched from the database by the *surname* part of the query. Most of these would be discarded as the query was refined – this would make the whole process very inefficient. By instead using satisfying: there is no interaction with the database until the final detect:.

withoutInheritance

It was noted in section 2.9 that hierarchies of classes which share a single table allow queries against a superclass to also return instances of subclasses. Sometimes this is not the desired effect; in this case a new storedInstances collection, specifically excluding the subclasses, can be obtained by using the message withoutInheritance.

As an example, consider a hierarchy representing printers and combined printer/scanners:

Printer	**-** *printSize, printResolution*
PrinterScanner	*- scanSize, scanResolution*

The query:

```
Printer storedInstances select: [ :each | each printSize = 'A4'].
```

...would return both Printers and PrinterScanners with A4 print size. If only Printer instances are required this can be achieved by specifying withoutInheritance:

```
Printer storedInstances withoutInheritance select: [ :each | each printSize = 'A4'].
```

6.2 Expanding a Query

Sometimes it may be convenient or necessary to expand an storedInstances collection to encompass more elements. Again, this can be due to the structure of a user interface, or to avoid a single enumeration block with multiple conditions logically ORed together.
This can be accomplished easily by 'adding' together storedInstances collections using the standard Smalltalk addition method addAll:. For example, to find all Customers named 'Smith' or 'Jones':

```
allSmithsOrJones := Customer storedInstances satisfying: [ :each | each surname = 'Smith'].
allSmithsOrJones addAll: (Customer storedInstances satisfying: [ :each | each surname = 'Jones']).
```

There are a couple of points to note with the use of addAll:. Firstly the message behaves like a regular Collection addition method in that it returns the message *argument*, not the receiver. Secondly, you may only add together storedInstances collections based on the same model class (or an inherited class sharing the same table).

6.3 Incremental fetch with ReadStreams

In section 5.4 we noted that using asOrderedCollection to fetch all elements of a large collection should be used with caution due to speed and memory considerations.

One way to mitigate this is by fetching a few elements at a time – for example a user interface may display a page with the first 50 members of a collection, and provide a "Next" button to fetch the next 50 elements (and then the next 50, and so on).

ReStore allows you to implement this style of behavior easily using normal Smalltalk ReadStream semantics by asking any storedInstances collection for its **readStream**

```
Customer storedInstances readStream

(Customer storedInstances satisfying: [ :each | each surname = 'Smith']) readStream
```

Once you have obtained a stream onto the collection you may then incrementally fetch elements of the collection using standard ReadStream messages (next, next:, upTo: etc.). Since only the data for the requested elements is fetched from the database, good performance can be maintained even for very large collections.

Note that the ability to fetch data incrementally is dependent on support from the underlying database interface. At the time of writing incremental fetch is supported by ODBC for Dolphin and SQLite and MySQL for Pharo. The Pharo P3 PostgreSQL interface currently always fetches all matching results, though you may still stream over these and ReStore will not reify the data into objects until needed.

6.4 Reporting with collect:

In addition to the enumeration messages discussed in the previous chapter (select:, detect: etc.), you can also use **collect:** with a storedInstances collection. In this case the enumeration block is used to fetch individual items of data from the database - this allows you to construct simple reports on your stored data. For example, the following block:

```
Customer storedInstances collect: [ :each | each surname]
```

...will fetch the surnames of all Customers, without fetching the Customer objects themselves.

Obviously one isolated item of data is not much use on its own, so ReStore allows you to fetch multiple items in a single query via the use of a new binary operator: **||** (double vertical bar). This concatenates together each item of data, returning the result as an Array. Using this technique it is possible to expand the above example to something more useful:

```
Customer storedInstances collect: [ :each | each firstName || each surname || each dateOfBirth]
```

The block argument to collect: is subject to the same rules as with the other enumeration messages. This means it is also possible to use messages defined by the class (where they satisfy the rules for query blocks) and to perform simple transformations on the server using supported message-to-function transformations:

```
Customer storedInstances collect: [ :each | each fullName || each dateOfBirth year]
```

Refining the report

Since the receiver of collect: is a storedInstances collection you are able to use the satisfying: message to perform the collect: query over a subset of your data, for example:

```
allCustomers := Customer storedInstances
ukCustomers := allCustomers satisfying: [ :each | each address country = 'United Kingdom'].
ukCustomers collect: [ :each | each fullName || each dateOfBirth year]

allOrders := CustomerOrder storedInstances.
recentOrders := allOrders satisfying: [ :each | each orderDate year = Date today year].
recentOrders collect: [ :each | each customer fullName || each totalPrice]
```

6.5 Aggregate Queries

Relational databases support **aggregate queries**, allowing calculations to be performed on groups of data, typically returning totals, counts and other numeric functions. ReStore allows you to create such queries from a storedInstances collection via the new enumeration message **project:** and the messages **count**, **sum**, **average**, **minimum** and **maximum**:

```
"Number of recent orders by customer"
recentOrders project: [ :each | each customer customerNo || each count].

"Total recent order value by customer"
recentOrders project: [ :each | each customer customerNo || each totalPrice sum].
```

Aggregate messages can also be combined in a single query, for example:

```
recentOrders project:
    [ :each |
    each customer customerNo || each count || each totalPrice sum ||
    each totalPrice average || each totalPrice minimum || each totalPrice maximum]
```

Using project: and aggregate messages allows your application to quickly produce summary reports with all calculation done on the server and the minimum of data transferred. Thus they can be very quick.

Grouping

Within a collect: block, any element not subject to an aggregate message is used to **group** the results. In the above examples we used each customer's unique *customerNo* to group results by individual customers. If instead we had used each customer's name:

```
recentOrders project: [ :each | each customer fullName || each count]
```

...then any customers with the same name would have had their order statistics grouped together into a single result – almost certainly not what you want for this particular report.

You can avoid this by additionally including the unique ID of the object why by which you wish to group the results:

```
recentOrders project: [ :each | each customer customerNo || each customer fullName || each count]
```

Note this is possible even where the unique ID is not defined as one of the object's own instance variables, by using the special message **_id**. For example if we didn't maintain a customer number as part of the Customer object, the above example could be written:

```
recentOrders project: [ :each | each customer _id || each customer fullName || each count]
```

6.6 Additional Functions

In “Messages and Functions” (Section 5.2) it was mentioned that within query blocks it is possible to use certain Smalltalk messages that can be mapped to equivalent SQL functions. For example the String message asUppercase is mapped to the SQL function `UPPER()`.

ReStore provides a default set of such translations however it is possible to specify additional translations depending on your requirements and the functions supported by your chosen database. By creating additional translations you can transfer more of the querying effort to the database, increasing the performance of your application.

As an example we will look at adding the SQLite function `RTRIM()`. When used with a single String argument the function removes any spaces from the right-hand side of the String (i.e. removes any trailing spaces) - this is roughly equivalent to the Pharo String method trimRight.

We can add a translation between the message trimRight and the function `RTRIM()` to a connected ReStore instance as follows:

```
ReStore translateMessage: #trimRight toFunction: 'RTRIM(%1)' asSQLFunction
```

With this translation added you can now use the message in query blocks, for example:

```
Customer storedInstances collect: [ :each | each fullName trimRight]
Customer storedInstances select: [ :each | each fullName trimRight = 'John Smith']
```

Notes:

1. Function translations are held by the sqlDialect object of ReStore which is created and reinitialized each time ReStore connects, based on the type of database. Therefore you should add your custom function translations to ReStore only after connecting, and re-add them if you subsequently disconnect and reconnect
2. Parameters within the SQL function are represented by numbered placeholders %1, %2 etc.. The first parameter %1 refers to the receiver of the message whilst additional parameters correspond to the arguments of the message – so function parameter %2 corresponds to the first message argument, %3 to the second etc.
3. The method trimRight does not actually exist in Dolphin Smalltalk, however you can still add the translation since the method itself is not actually evaluated (though browsers will give an “undefined selector” warning)
4. The translated SQL does not have to be a single function but can be any valid SQL expression. Let’s say you have added a method plusPercent: as an extension to Number and would like to use an equivalent to this in a query block. You can do this by adding a translation from plusPercent: to an equivalent calculation in SQL:

```
ReStore
    translateMessage: #plusPercent:
    toFunction: '%1 * (1 + (%2 / 100))' asSQLFunction
```

Specifying the Result Type

With message-function translations there is an implicit assumption that the result of the function is of the same type as the receiver. Whilst this assumption holds in many cases – numerical functions (+, - etc.), String concatenation, and the examples discussed above – this is not always the case. For example, the String message size is mapped to the SQL function `LEN()` (or some variant depending on the database) whose result is an Integer, not a String. In cases like this ReStore allows you to explicitly state the class of the result object.

As an example we will consider a further extension method to Number asPercentageString which displays a floating point number in the range 0-1 as a percentage String, e.g. '50%'. When creating a translation to a matching SQL expression we must specify the class of the result as String, since it does not match the receiver class of Number:

```
ReStore
    translateMessage: #asPercentageString
    toFunction: ('CAST((%1*100) as char) || "%%"' asSQLFunctionWithResultClass: String)
```

Notes:

1. This SQL expression is specific to SQLite; it will differ for other databases. For example in MySQL it would be 'CONCAT(CAST((%1*100) as char), "%%")'
2. Note the doubled percent sign "%%" at the end of the expression – this is needed to escape the percent sign since it is a feature of the numbered parameter placeholders
3. ReStore provides a number of convenience methods to declare functions similar to this more succinctly: asSQLFunctionStringResult, asSQLFunctionIntegerResult and asSQLFunctionBooleanResult

Result Type based on Argument Type

There is a further possibility when translating a function – the result type may vary based on the argument type. As an example consider a Number method ifPositive:ifNegative: which returns either the first or second argument depending on the sign of the receiver, e.g.

```
123 ifPositive: 1 ifNegative: -1
-123 ifPositive: 'POS' ifNegative: 'NEG'
```

This can be translated to SQL using the `CASE` function available in most databases, however we need to declare that the result type will be neither the receiver class or a fixed class, but the class of one of the arguments (we assume that all arguments and thus possible results are of the same class). This can be done as follows:

```
ReStore
    translateMessage: #ifPositive:ifNegative:
    toFunction: ('CASE WHEN %1>=0 THEN %2 ELSE %3 END' asSQLFunction resultParamIndex: 2)
```

By adding "resultParamIndex: 2" to the declaration we state the result of the function will be the same type as parameter %2.

6.7 Data-Modifying Queries

In addition to returning objects, storedInstances collections can also be used to modify data within the database.

modify:

modify: is a type of enumeration message where the argument block is expected to update some aspect of the member objects. However it is important to understand that the update is done **directly within the database** – this makes the update very efficient (since no objects are fetched from the database) but has the limitation that any objects already in memory (either in your image, or that of another user) will not be updated. Thus modify: is most suitable for 'supervisor'−type activities, ideally when no other image is connected to the database.

As an example of its use, let's say that you've just added the instance variable *country* to the class **Address**, in preparation for handling overseas customers. All current addresses are presumed to be in your home country; you wish to update these to make this explicit. This is an ideal application for modify:

```
Customer storedInstances modify: [ :each | each address country: 'United Kingdom'].
```

unstoreAll / unstore:

unstoreAll can be used to quickly delete from the database all members of a storedInstances collection. For example, the following would delete all CustomerOrder objects from the database:

```
CustomerOrder storedInstances unstoreAll
```

unstore: operates in a similar way, but takes as its parameter a select:-style block which restricts the objects to delete. The following would delete all CustomerOrders placed before 2015:

```
CustomerOrder storedInstances unstore: [ :each | each orderDate year < 2015]
```

6.8 Performance Considerations

At the start of the previous chapter we mentioned that, after reading an object from the database with an initial query, you may traverse the object's links to other objects; these are then automatically fetched from the database. Depending on the structure of your application this can lead to a potential performance issue known as the "1+N problem".

Consider a screen that allows you to find multiple Customer instances based on certain criteria. The results list of this screen shows the details of each matching customer (name etc.) plus the number of CustomerOrders placed by that customer.

Fetching the list of matching customers can be done with a single[7] query. However as each customer's details are displayed the collection of orders for that customer will be fetched from the database in order to display its size. Thus we have one query returning N customers, then N queries to fetch each customer's orders.

You can avoid this performance penalty by preconfiguring a storedInstances collection to fetch any related object or collection of objects for every member of the storedInstances collection using a single query. This is done via the messages **include:** and **includeAll:**

```
template := Customer new.
template surname: 'Sm*' asWildcard.
matchingCustomers := template similarInstances.
matchingCustomers include: #customerOrders.
matchingCustomers := matchingCustomers asSortedCollection.
```

Now, when asSortedCollection is used to fetch the matching customers, one additional query will automatically be issued to fetch the *customerOrders* of all matching customers.

include[All]: can also be used to fetch single related objects as well as collections. For example, if your results list also needed information from the *address* instance variable of a Customer, you would add this to the list of instance variables to be fetched:

```
matchingCustomers includeAll: #(#customerOrders #address).
```

[7] Depending on the size of the storedInstances collection ReStore may actually use more than one query. By default collections are split into pages of 100 objects, with one query used for each page – thus 2 queries for a collection of 120, 3 for 240 etc.. The figure of 100 is configurable - see definitions of readAllBatchSize for further information.

With Real Collections

An alternative scenario that can also be affected by the 1+N problem is where you already have a real collection of objects in your image, but you want to fetch the contents of certain instance variables for all members of that collection.

Let's twist our current example of a "Find Customer" screen to demonstrate this. Let's say that the initial list of customers only displays information directly held in the Customer object. Thus at this stage we have no need to user includeAll: to fetch additional objects – let's remove this from our previous code:

```
template := Customer new.
template surname: 'Sm*' asWildcard.
matchingCustomers := template similarInstances asSortedCollection.
```

However, once the user has located the customers of interest they can then click a "Details" button to display additional information for the matching customers – again, let's say this will be based on their orders and address. By allowing this information to be fetched in the standard way (transparently, on demand) we would incur two queries for each Customer (one for the *address*, one for the *customerOrders*). However we can avoid this by instructing the (real) SortedCollection to fetch this information for all its elements using **fetchAll:** (or **fetch:** for a single instance variable):

```
matchingCustomers fetchAll: #(#customerOrders #address).
```

Now, similar to a storedInstances collection and includeAll:, the real collection will issue a single query for each requested instance variable, fetching the contents of those variables for all members of the collection.

By careful use of include[All]: and fetch[All]: you can avoid the 1+N problem from impacting the performance of your application.

7 Update Clashes

Where an application is the sole user of a database, transactions should always succeed. However where more than one user is interacting with the database there is the possibility for an update clash. Consider the following example:

Time	User A	User B	Database
00:00	Begin editing Person 'John Smith'		'John Smith'
00:01		Begin editing Person 'John Smith'	'John Smith'
00:09	Change *surname* to **'Smythe'**		'John Smith'
00:10		Change *surname* to **'Smithe'**	'John Smith'
00:14	commitTransaction / store		'John **Smythe**'
00:15		commitTransaction / store	???

What should happen when User B commits their change? If the change is allowed to succeed, then User A's changes will be overwritten. If it fails, then User B will potentially lose their changes − is this acceptable?

By default ReStore records a version number for each object in the database; each time the object is updated the version is incremented by 1. This allows ReStore to detect where another database user has changed an object, and thus to avoid overwriting another user's changes from being overwritten.

Where this happens, the originating commitTransaction or store message will fail – this chapter discusses how to handle this.

7.1 Handling Transaction Failures

When working with transactions the result of an update clash is that the transaction will fail to commit. To check for this, you should test the result of commitTransaction − this will return a Boolean to indicate whether the transaction succeeded (**true**) or failed (**false**).

Note that when a transaction fails to commit you have not lost the changes made during that failed transaction – the transaction is still active, the changed objects retain their changed state in memory and your changes still exist in the transaction. However none of the changes have been committed to the database − an update clash on one single object prevents the whole transaction from committing.

Rollback and Refresh

The simplest action to take is to fail the whole transaction, rollback the changes made and update the clashing object(s) with the latest versions from the database. This is done as follows:

ReStore **rollbackAndRefreshTransaction**

Following rollbackAndRefreshTransaction, you could (for example) issue a notification of the clash to the user ("Another user has changed the data you were editing..."), present them with the refreshed objects and ask them to repeat their actions.

Refresh and Rollforward

An alternative approach is to merge the changes in the transaction with the changes in the database. This is done as follows:

ReStore **refreshAndRollforwardTransaction**

With refreshAndRollforwardTransaction the object(s) causing the update clash are refreshed with their latest versions from the database and the changes made in the transaction are re−applied onto the refreshed objects. Following a refreshAndRollforwardTransaction, you will need to send commitTransaction again, remembering that this second commit may also fail if any objects have been changed further since the refresh.

Using refreshAndRollforwardTransaction avoids the user having to repeat their changes, but you should bear in mind that this also allows the user to overwrite another user's changes without having first viewed them.

7.2 Handling Store Failures

Just like when using transactions, a database update made via store or storeAll may also fail due to an update clash. In this case the store[All] message will answer **false** (and conversely **true** in the event of a successful database update).

Unlike when using transactions, following a store failure there is no active transaction left open. However the object's changes still exist in the image and so you need to decide how to proceed.

Object Refresh

If you wish to bring the changed object(s) up-to-date with its current state in the database you can send the messages **refresh** and **refreshAll**

```
changedObject refresh

changedObjects refreshAll
```

As noted in 4.2, refresh[All] effectively performs a rollback for objects that have not been updated in the database compared to your image. Thus it is equivalent to rollbackAndRefreshTransaction when working with transactions.

Handling Store Failure with Transactions

There is an alternative method of handling store[All] failures that enables you to catch a failure while the transaction is still active. To do this you should trap the exception **StoreFailure** when attempting the store operation; you may then directly interact with the failed transaction (the exception parameter providing a wrapper onto this). Example:

```
[object store] on: StoreFailure do: [ :transaction | transaction rollbackAndRefresh]
```

See the *operations* method category of StoreFailure for other supported actions.

7.3 Minimizing Update Clashes

ReStore will attempt to minimize update clashes by handling them automatically wherever possible. When a clash is detected, ReStore will compare three versions of the object:

- the object as it was prior to the current change (version 1)
- the object as it is in memory, following the change (version 2)
- the object as it is currently stored in the database (version 3)

If the changes made in the database (1>>3) are independent of the changes made in memory (1>>2) ReStore will avoid an update clash by refreshing 2 with the changes made between 1 and 3. It will then commit this merged version of the object, giving version 4. For example:

Time	**User A**	**User B**	**Database**
00:00	Begin editing Person 'John Smith'		'John Smith'
00:01		Begin editing Person 'John Smith'	'John Smith'
00:09	Change *surname* to **'Smythe'**		'John Smith'
00:10		Change *firstName* to **'James'**	'John Smith'
00:14	Commit Transaction		'John **Smythe**'
00:15		Commit Transaction **fails; update clash detected**	'John Smythe'
00:16		Merge changes, recommit **succeeds**	'**James** Smythe'

If this automatic merging is not acceptable for a particular object (or particular instance variables of an object) you can override the method **mergeUpdate:from:** to prevent this by returning **false** – see the comment of the Object implementation of this method for further information.

7.4 Handling Clashes Automatically

If there is a 'full' update clash (i.e. the changes 1>>2 and 1>>3 affect one or more of the same instance variables) ReStore will ask the changed object itself to try to resolve the change. ReStore does this by sending the message **handleUpdateClash:from:to:** to the affected object(s) for each clashing instance variable − the arguments to the message being the changed instance variable (a Symbol), the previous version of the object (version 1) and the current database version of the object (version 3).

By implementing handleUpdateClash:from:to: in your model classes you can update the receiver (i.e. version 2, the in−memory version of the object) to resolve the clash. The result of this method should be a Boolean indicating whether the clash was resolved. If all update−clashing objects in a transaction are able to resolve their own clashes, then the transaction will automatically re-commit successfully with no further intervention.

As an example we'll return to the previous **CustomerOrder** example, in particular the **Product** class. The most volatile attribute of a Product is likely to be its *stockLevel* since this will change every time an order is made for that item. A popular item is likely to be ordered by many users at one time, and so an update clash on *stockLevel* is highly likely. Product can attempt to resolve *stockLevel* update clashes by implementing handleUpdateClash:from:to:

```
handleUpdateClash: aSymbol from: oldVersion to: newVersion

    | myStockChange mergedStockLevel |

    (aSymbol = #numberInStock) ifFalse: [^false].

    myStockChange := self stockLevel – oldVersion stockLevel.
    mergedStockLevel := newVersion stockLevel + myStockChange.

    ^mergedStockLevel >= 0
        ifTrue: [self stockLevel: newStockLevel. true]
        ifFalse: [false]
```

Firstly, the method checks whether the clashing attribute (aSymbol) is actually *stockLevel*; if not it makes no further attempt to handle the clash.

Next, the relative change in *stockLevel* between version1 (oldVersion) and version 2 (self) is calculated. The method then applies this change of stock level to the *stockLevel* as currently stored in the database (newVersion), to give the new, merged, stock level.

Finally the method checks that the merged stock level change is not negative − this would be invalid and so cannot be handled. Assuming this is not the case, however, the merged *stockLevel* is applied to the receiver and true is returned to denote that the update clash has been handled successfully.

By targeting implementations of handleUpdateClash:from:to: at situations where update clashes are likely, you can improve the usability of your applications by ensuring that most transactions commit successfully on the first attempt.

8. Working with Multiple ReStores

So far in this manual we have used the singleton default instance of **SSWReStore**, accessed via the **ReStore** global variable. This is a simple and convenient way of using ReStore where only one ReStore instance per image is necessary.

It is equally possible to use multiple instances of SSWReStore, for example in a web application server you may choose to have one SSWReStore instance per user connection.

Each instance of SSWReStore is completely independent and shares no objects or state with another instance. This allows you to have an active transaction in one instance but not another, instances connected to different databases, or instances connected to the same database but with different user credentials (e.g. separate user and admin instances) for added security.

8.1 Manually Specifying a ReStore

At its simplest you can just create your own instance of SSWReStore:

```
myReStore := SSWReStore new
```

Once created and connected you may pass your own SSWReStore instance as a parameter to ReStore messages which normally assume the default singleton instance; this will cause your instance to be used instead:

```
"Storing a new object in default ReStore"
Customer new store.

"Storing a new object in myReStore"
Customer new storeIn: myReStore.[8]


"Storing multiple new objects in default ReStore"
aCollection storeAll.

"Storing multiple new objects in myReStore"
aCollection storeAllIn: myReStore.


"Accessing stored instances in default ReStore"
Customer storedInstances.

"Accessing stored instances in myReStore"
Customer storedInstancesIn: myReStore.


"Accessing similar instances in default ReStore"
templateObject similarInstances.

"Accessing similar instances in myReStore"
templateObject similarInstancesIn: myReStore.
```

[8] Persistent objects know which SSWReStore instance they belong to, so when storing an already-persistent object it is not necessary to specify the instance. For consistency you may chose to do so; in this case ReStore will sanity-check that the parameter matches the SSWReStore to which the object belongs

8.2 Using Affinity

In addition to specifying the SSWReStore instance manually, ReStore also provides a way for the "current" or "active" ReStore to be selected automatically. This is done by defining the **affinity** of a particular ReStore instance.

Process Affinity

A simple example of this is **process affinity** – having a separate SSWReStore instance for a particular process. ReStore provides a straightforward way to configure this:

```
aProcess reStore: myReStore.
```

With this defined, myReStore will be used in place of the default singleton ReStore whenever aProcess is active. Thus you may use store, storeAll and storedInstances as normal without specifying myReStore as a parameter.

If you wish to access the currently-active instance of SSWReStore (e.g. to begin a transaction) you can do so as follows:

```
SSWReStore default
```

You may remove a process affinity as follows:

```
aProcess reStore: nil.
```

General Affinity

Process affinity is a special case of **general affinity** – the ability to specify a particular SSWReStore instance for an arbitrary "current state" of your image. For example, under a web application framework you may have a dynamic variable named **CurrentSession** defining the active session when handling a request. You can bind a particular SSWReStore instance to a particular session as follows

```
myReStore
    affiliateWith: aSession
    using: [ :session | CurrentSession value == session].
```

With this affinity defined, myReStore will be used in place of the default singleton ReStore whenever aSession is the active value of CurrrentSession. Again, this means you may use store, storeAll and storedInstances as normal without specifying myReStore as a parameter, and you may access the currently-active ReStore instance via "SSWReStore default"

You may remove a general affinity as follows:

```
myReStore disaffiliateWith: aSession
```