Carbonado™ 1.2 User Guide

2010-06-15

1 Introduction

Carbonado is an extensible, high performance persistence abstraction layer for Java applications, providing a simple and consistent view to the underlying persistence technology. Although it may appear to be an object database, or an "object-relational" bridge, Carbonado respects the relational model rather than hide it.

The key difference is that Carbonado unifies the interaction to persistence technologies without mandating any of them. It can provide access to a dynamically replicated Berkeley DB via the same access patterns used for accessing a full scale RDBMS. Carbonado presents access as a relational database, complete with powerful querying features and indexing. In addition, Carbonado supports transactions, optimistic locking, joins, and LOBs. Applications developed against Carbonado are free to switch persistence technologies without imposing a significant impact – sometimes none.

The flexibility offered for access to storage extends to caching as well. Carbonado makes it easy to select and change caching technologies, with sophisticated mechanisms for electing whether to use the cache or fall back to primary storage.

There are some limitations which come along with this flexibility. For example, Carbonado queries are not as expressive as SQL selects – aggregate functions and correlated subqueries are not supported. Carbonado does allow direct SQL in certain contexts, however.

This document serves as an introduction and reference for software developers wishing to use Carbonado. Carbonado depends heavily on Java 5 features, including annotations, generics, and enumerations. Familiarity with these features is required in order to use Carbonado, as well as understand the remainder of this document. It is also assumed that the reader is familiar with how data is organized in a relational database.

2 Overview

Specifying what you are going to store and how you are going to store it in Carbonado requires two things: the Carbonado records, known as *Storables*, and the persistence layer, known as *Repositories*.

Every Storable implements or extends the Storable interface, and then follows standard JavaBean conventions for defining properties. Storables also contain operations to read and write to storage directly. In SQL terms, a Storable type definition is analogous to a table definition, Storable instances are analogous to table rows, and the properties of the Storable represent the columns.

With Carbonado, the property access method code is not user implemented, but rather is auto-generated at runtime. As such, here is all the code needed to define a simple record that can be persisted by Carbonado:

```
import com.amazon.carbonado.PrimaryKey;
import com.amazon.carbonado.Storable;

@PrimaryKey("ID")
public interface StoredMessage extends Storable {
    long getID();
    void setID(long id);

    String getMessage();
    void setMessage(String message);
}
```

To store and retrieve data, a Repository object must be set up to define what persistence technology to use and how to use it. Repositories come in many flavors, including JDBC, Berkeley DB, and special repositories which use the Composite[1] pattern to glue other repositories together in interesting and meaningful ways. Once the Repository is set up and built, the interactions with it are largely independent of the underlying implementation. Your application code will never need to know whether you are testing with a local, temporary Berkeley DB, or in production using a replicated repository with a backing RDBMS.

The following code snippet shows how the storable described above would be inserted into a repository, once the repository had been created:

```
   // Insert message.
   StoredMessage message = repo.storageFor(StoredMessage.class).prepare();
   message.setID(1);
   message.setMessage("Hello Carbonado!");
   message.insert();
```

[1] Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. See also: *Design Patterns: Elements of Reusable Object-Oriented Software.*

A small “Hello World” program fleshing out this example is provided in the appendix.

3 The Repository

Repository instances are the first layer of abstraction, bridging a persistence technology to Carbonado. All user-level interactions with Carbonado operate through a Repository, although sometimes indirectly.

Theoretically, all Repositories should operate exactly the same, no matter what the persistence layer may be. In reality, this requirement is impossible to achieve, so expect some differences in Repositories:

- **Transaction support** – Ideally, all Repositories should support full ACID transactions, with support for all isolation levels. Some Repositories may rely upon batch commits and optimistic locking as a weaker kind of transaction. Also, for databases that don't support nested transactions, committing or rolling back an inner transaction has no effect.
- **Optimistic locking** – Repositories require a record version number to support optimistic locking. Usually, all that is needed to support optimistic locking is to include a version property. Although most Repositories should be able to fully support optimistic locking, three support levels are defined. See Version for more info.
- **Schema dependence** – Repositories backed by an external schema, like an SQL database, are termed "dependent". All Storables available from such a Repository must match the external schema definition. The JDBC Repository, discussed later, implements a dependent Repository. In contrast, independent repositories rely on Storable definitions exclusively as the schema. Defining and changing Storable types is thus easier when using an independent Repository. The BDB/Berkeley DB Repository is an example of such a Repository.
- **Schema evolution** – All Repositories should support some form of schema evolution. At the very least, all independent Repositories should support adding and dropping of Storable properties without causing existing records to appear corrupt.
- **Property type support** – All Repositories must support a minimum set of property types which consists of all Java primitive types, boxed primitive types, and Strings. Support for arrays, LOBs, and custom types may vary by Repository. *Adapters* can be defined sometimes to support properties that the Repository does not natively support. DateTime property types are supported by such an Adapter, although it is automatically applied.
- **Sequence support** – Properties may optionally be annotated to support automatic value selection upon insert via a *Sequence*. Repositories that don't support this feature ignore the annotation.

Repository implementations may support additional features not defined in the standard Repository interface. These are exposed through an extensible *Capability* mechanism.

3.1 Opening Repositories

The standard way of obtaining a Repository is via a *RepositoryBuilder*. Some Repository implementations may also provide direct constructor access, but their use is discouraged.

A RepositoryBuilder is essentially a container of configuration properties, with a build method which returns a Repository instance. This design has several advantages over constructing Repository instances directly. Based on supplied configuration, the builder may decide to return a completely different kind of Repository instance.

Each kind of RepositoryBuilder contains custom configuration pertaining to the type of Repository it supports. At the very least, all Repositories must be given a name. This name may be used when constructing log messages, but some Repositories need the name to access other required configuration.

Here is an example of opening a Repository:

```
        // First step is to create a specific builder.
        BDBRepositoryBuilder builder = new BDBRepositoryBuilder();
        // All Repositories require a name to be set.
        builder.setName("demo");
        // The following options are specific to the BDB Repository.
        File envHome = new File
            (System.getProperty("java.io.tmpdir"), "carbonado-demo");
        builder.setEnvironmentHomeFile(envHome);
        builder.setTransactionWriteNoSync(true);

        // Finally, build the instance.
        Repository repo = builder.build();
```

Building Repository instances can be expensive, and building multiple instances of a given type may even be prohibited. Therefore, Repository instances should only be created once during the lifetime of an application, and be freely shared throughout. All Repository implementations must therefore be thread-safe.

3.2 Repository operations

The Repository interface is quite small, basically supporting access to Storage, transactions, and custom capabilities.

3.2.1 Storage access

All persistence and query operations pertaining to a specific Storable type go through a typed Storage instance. The following Repository method returns Storage instances:

```
    <S extends Storable> Storage<S> storageFor(Class<S> type);
```

The type parameter refers to an interface or abstract class which implements Storable. The first time this method is invoked, the type is analyzed to see if it is well formed and can be supported by the Repository. If not, a MalformedTypeException or SupportException may be thrown.

Here is an example call, operating on an existing Repository instance:

```
        Storage<StoredMessage> storage = repo.storageFor(StoredMessage.class);
```

If the Storable type is supported, subsequent invocations return a cached Storage instance. Storage instances are thread-safe and immutable. Although requesting a cached Storage instance is relatively fast, applications may choose to acquire Storage instances upon start-up and save references to them. This strategy moves the cost of the initial invocation into application startup, and it also detects problems early.

3.2.2 Transaction access

Transactions may be entered only via a call to the Repository. The use of Transactions is optional, and if not used, persist operations are auto-commit. Transactions are thread-local, and the Repository can be asked at any time if the current thread is in a transaction.

```
    Transaction enterTransaction();

    Transaction enterTransaction(IsolationLevel level);

    IsolationLevel getTransactionIsolationLevel();
```

Correct use of Transactions is discussed in detail later (§9) in this document.

3.2.3 Extra capabilities

Repositories can support custom features not provided by the standard Repository interface. Rather than extend the Repository interface, custom features are exposed as Capabilities.[2] A Capability is requested by type, and if a Repository does not support it, null is returned.

```
    <C extends Capability> C getCapability(Class<C> capabilityType);
```

Here is a small example showing a Capability being requested and used. In particular, this capability is used to interrogate a Repository for the set of Storable types it has:

```
    StorableInfoCapability cap =
        repo.getCapability(StorableInfoCapability.class);

    String[] typeNames = cap.getUserStorableTypeNames();
    ...
```

The set of supported Capabilities is documented with Repository implementations.

[2] Carbonado Repositories are often implemented as wrappers around other Repositories. If a new capability was provided by extending the Repository interface, it would get lost as soon as the Repository was wrapped. The Capability design provides a means for features to break through the layers.

3.3 Future direction

By moving custom Repository features into builders and Capabilities, the main Repository interface is kept clean. This makes it possible to implement a wide variety of Repository types without significantly impacting the design of applications which use Carbonado.

The first version of Carbonado supports a small set of Repository types, which are discussed later. Future Repository types will likely be defined to support generic remote access, data partitioning and integration with other kinds of persistence technologies.

Carbonado is designed to be extensible, and so anyone could develop their own type of Repository. Defining a new Repository can be a bit challenging, but most Carbonado users are expected to only use existing Repositories.

4 Storables

Storables represent entities that Carbonado can persist, following the Active Record design pattern[3]. All Storable types are defined as public abstract classes or interfaces, derived from the Storable interface. The actual implementation of the Storable is generated at runtime by Carbonado.

Each property that Carbonado persists is defined by two methods, an accessor and a mutator. The pair must follow JavaBean property naming conventions, and both must be abstract. No fields should be defined to represent persistent properties, as Carbonado will not examine them.

Creating Storables as interfaces is generally easier than creating as abstract classes, only in that less key strokes are required. Creating an abstract class offers the benefit of adding any convenience methods that operate directly on the Storable. Storables as abstract classes must have a public no-argument constructor, and all abstract methods must be part of a property definition.

Here's a basic Storable, defined as an abstract class:

```
import com.amazon.carbonado.PrimaryKey;
import com.amazon.carbonado.Storable;
import org.joda.time.DateTime;
import org.joda.time.Period;
import org.joda.time.PeriodType;
import org.joda.time.ReadableInstant;

@PrimaryKey("userID")
public abstract class UserInfo implements Storable {
    public abstract long getUserID();
    public abstract void setUserID(long id);

    public abstract String getFirstName();
    public abstract void setFirstName(String name);

    public abstract String getLastName();
    public abstract void setLastName(String name);

    public abstract String getFavoriteColor();
    public abstract void setFavoriteColor(String color);

    public abstract DateTime getBirthDate();
    public abstract void setBirthDate(DateTime date);

    // Convenience method; not a persisted property
    public int getAge() {
        return new Period(getBirthDate(), (ReadableInstant) null,
                          PeriodType.years()).getYears();
    }
```

[3] An Active Record is an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data. Technically speaking, however, a Storable is only an Active Record when domain logic is added. See also: *Patterns of Enterprise Application Software*.


```
}
```

All Storables must have a PrimaryKey annotation referencing at least one property. In this example, the "userID" property is the primary key. As a result, no two UserInfo records can have the same userID.

Storables can extend other Storables, which causes them to simply inherit properties and methods from the parent. Carbonado does not observe object hierarchies, however. It just treats the subclass as if it was a flat set of properties.[4]

This example shows how a common base class can be used to make it easier to define Storables which should have a common set of properties:

```
public interface TaggedEntity extends Storable {
    String getTag();
    void setTag(String tag);
}

public abstract class UserInfo implements TaggedEntity {
    ...
}

public abstract class Address implements TaggedEntity {
    ...
}
```

Upon first use, all Storables go through an extensive introspection phase, which verifies the Storable is well-formed. Otherwise, a MalformedTypeException is thrown by the Repository.storageFor method, detailing the cause.

4.1 Annotations

Carbonado Storables may be annotated to refine their definition beyond what standard Java syntax is capable of. At the very minimum, at least one annotation is required, which specifies which properties participate in the primary key.

Some annotations apply to the Storable as a whole and others apply to a single property directly. With only a few exceptions, annotations applied to a single property are placed on the accessor (get) method.

This section introduces basic annotations but does not list all of them. Advanced annotations are described in later sections.

Note: Java 5 annotations support a feature that allows them to be inherited by subclasses. Carbonado annotations do not use this feature, however. Storable subclasses must re-specify annotations at the class level, and for overridden properties.

[4] Carbonado can be described as following the *Concrete Table Inheritance* pattern. See also: *Patterns of Enterprise Application Software*.

4.1.1 PrimaryKey

Every well formed Storable requires a PrimaryKey annotation. Composite keys can be defined as well, simply by including multiple properties. Each of these properties is regarded as being a member of the primary key.

Each property listed in the PrimaryKey may be optionally prefixed with a '+' or '-' character, to indicate a preferred sort ordering. If not specified, it defaults to ascending order, which is the same as a '+' prefix.

```
@PrimaryKey({"entityTypeID", "-entityInstanceID"})
public interface Content extends Storable {
    int getEntityTypeID();
    void setEntityTypeID();

    String getEntityInstanceID();
    void setEntityInstanceID();

    ...
}
```

When inserting a new Storable instance, if the primary key matches an existing Storable's primary key (of the same type), a UniqueConstraintException is thrown.

4.1.2 AlternateKeys

Alternate keys are defined similarly to primary keys, except a Storable can contain any number of them. It is just as good as the primary key for uniquely identifying Storable instances.

Repositories are much more flexible with alternate keys than with primary keys. For example, dropping an alternate key and reconstructing it should not result in loss of data. Alternate keys are often implemented as indexes with a uniqueness constraint.

For supporting more than one alternate key definition, two kinds of annotations are used. The AlternateKeys annotation is a container for Key annotations, which are defined like PrimaryKey annotations. Preferred sort order can be identified as well using a '+' or '-' prefix on the property name.


```
@PrimaryKey("ID")
@AlternateKeys({
    @Key("fullPath"),
    @Key({"+name", "parentID"})
})
public interface FileInfo extends Storable {
     long getID();
     void setID(long id);

     String getFullPath();
     void setFullPath(String path);

     String getName();
     void setName(String name);

     long getParentID();
     void setParentID(long id);

     ...
 }
```

Like primary keys, when inserting a new Storable, if another Storable of the same type has a matching alternate key, a UniqueConstraintException is thrown.

4.1.3 Nullable

By default, Storable properties cannot be set to a null value. Passing null to the set method causes an IllegalArgumentException to be thrown. Also, inserting a Storable which has unset non-nullable properties causes a ConstraintException to be thrown.

To allow null values, apply the Nullable annotation. If the property value is a primitive, it cannot be Nullable unless it is boxed. (i.e., “int” is boxed as “Integer”)

```
public abstract class UserInfo implements Storable {
    ...

    @Nullable
    public abstract String getFavoriteColor();
    public abstract void setFavoriteColor(String color);

    ...
}
```

4.1.4 Version

A property annotated with Version is designated as the authoritative version number for the entire Storable instance. Versioning is an optional feature, but when used, the semantics of how updates work changes. Specifically, it incorporates optimistic locking checks. Some Repositories might require all Storables to be versioned.

Support for the version property falls into three categories, and all Repositories implement some or all of these. A repository may manage the version, it may respect the

version, or it may merely check the version. Most are fully capable of fully managing the version, however.

- **Manage** – The Repository takes responsibility for establishing the version on insert, and for auto-incrementing it on update. Under no circumstances should the version property be incremented manually; this can result in a false optimistic lock exception, or worse may allow the persistent record to become corrupted. Prior to incrementing, the Repository will verify that the version exactly matches the version of the current record, throwing an OptimisticLockException otherwise.

- **Respect** – Repositories which respect the version use the version to guarantee that updates are idempotent – that is, that an update is applied once and only once. These repositories will check that the version property is strictly greater than the version of the current record, and will (silently) ignore changes which fail this check.

- **Check** – Philosophically, a version property can be considered part of the identity of the storable. That is, if the storable has a version property, it cannot be considered fully specified unless that property is specified. Thus, the minimal required support for all repositories is to check that the version is specified on update. All repositories – even those which neither manage nor respect the version – will throw an IllegalStateException if the version property is not set before update.

The actual type of the version property can be anything, but some repositories might only support integers. For maximum portability, it is recommended that version properties be defined as a regular (32-bit) int.

```
public abstract class UserInfo implements Storable {
     ...

     @Version
     int getRecordVersionNumber();
     void setRecordVersionNumber(int rvn);

     ...
 }
```

The JDBCRepository requires special attention in order for version properties to be properly managed. By default, it assumes a trigger is installed on the RDMS that performs an auto-increment. A configuration option on the builder enables auto-increment to be performed by the JDBCRepository itself:


```
    JDBCRepsitoryBuilder builder = ...
    ...
    // Perform auto-increment locally
    builder.setAutoVersioningEnabled(true, "org.foo.UserInfo");
    builder.setAutoVersioningEnabled(true, "org.foo.Address");
    ...
```

If the class name argument to setAutoVersioningEnabled is null, then local auto-increment is enabled for all Storable types.

4.1.5 Sequence

A property that is annotated with @Sequence causes a value to be automatically selected for it when the Storable is inserted. This is often useful for primary keys, simplifying the process of generating new identifiers. If property set method is called prior to insert, the sequence is not invoked.

```
@PrimaryKey("userID")
public abstract class UserInfo implements Storable {
    @Sequence("USER_ID_SEQ")
    public abstract long getUserID();
    public abstract void setUserID(long id);

    ...
}
```

Sequence support may vary by Repository, and some may not support it at all. Also, the Sequence annotation requires a name parameter, although this may be ignored. Independent Repositories (Berkeley DB) can create new named sequences automatically, but for dependent Repositories (JDBC), the name must refer to an existing sequence.

If the database product being accessed by JDBC does not support sequences, a special table must be defined in order for sequences to work:

```
 CREATE TABLE CARBONADO_SEQUENCE (
     NAME           VARCHAR(100) PRIMARY KEY,
     INITIAL_VALUE  BIGINT       NOT NULL,
     NEXT_VALUE     BIGINT       NOT NULL,
     VERSION        INT          NOT NULL
 )
```

When using the sequence table, the name does not need to refer to an existing sequence. The sequence information will be inserted automatically upon first use. If you wish to always use the sequence table instead of the database's built-in support, enable this option:


```
    JDBCRepsitoryBuilder builder = ...
    ...
    // Always use sequence table for sequences
    builder.setForceStoredSequence(true);
    ...
```

4.1.6 Automatic

A property may be annotated with @Automatic, which instructs the repository to select a value for it upon insert. The actual process by which a value is automatically assigned is repository dependent. In the JDBC repository, the value might come from an auto-increment column or a database-specific trigger.

If the underlying repository doesn't automatically supply a value to an automatic property, no immediate warning is given and instead the property will be assigned a default value of null or zero. This may cause problems if the automatic property is a member of a key. Explicitly specifying a value prior to insert can sometimes be used to override the automatic value altogether.

```
@PrimaryKey("userID")
public abstract class UserInfo implements Storable {
    @Automatic
    public abstract long getUserID();
    public abstract void setUserID(long id);

    ...
}
```

Automatic properties are a good alternative for supporting JDBC databases that don't natively support sequences, but they do support auto-increment columns. If the database doesn't support auto-increment columns, a database trigger will likely need to be installed in order to emulate this behavior.

4.1.7 Alias

An Alias annotation lists alternate names for a Storable or a Storable property. An alias is used only by a dependent repository to link to entities. Without an alias, the repository may perform a best guess at finding an entity to use. Aliases may be ignored by repositories that don't require explicitly named entities.

The most common use for an alias is for a JDBC Repository, to link a Storable to a table, and its properties to the corresponding columns. Without the Alias annotation, the JDBC Repository first searches for the table or column which exactly matches the Storable or property name. Failing this, it tries again with a de-CamelCased version that uses underscores to separate name parts. For example, UserInfo is translated into USER_INFO. If any alias is specified, no automatic mapping happens.

Multiple aliases may be specified in the Alias annotation, and the Repository selects the first match. If there is no match, the Repository will not try to deduce the match itself, and throws a MismatchException instead.


```
@PrimaryKey("userID")
@Alias("USER_INFO")
public abstract class UserInfo implements Storable {
    @Alias("USER_ID")
    public abstract long getUserID();
    public abstract void setUserID(long id);

    @Alias("GIVEN_NAME")
    public abstract String getFirstName();
    public abstract void setFirstName(String name);

    @Alias({"SURNAME", "FAMILY_NAME"})
    public abstract String getLastName();
    public abstract void setLastName(String name);

    ...
}
```

4.2 Adapters

Adapters provide a means for supporting property types which are not natively supported by a given Repository. Repositories must always attempt to match properties to the best matching native type, but they may have to rely on an Adapter to make a conversion. Only one Adapter is allowed per property.

A few Adapters are bundled with Carbonado, but the Adapter design is extensible, making it easy to support new kinds of property types. The Javadocs in the AdapterDefinition annotation describe how to create new Adapters.

Note: Adapters only support one-to-one adaptation. They do not support merging multiple properties into one.

4.2.1 TrueFalseAdapter

If the Repository does not natively support boolean types, but instead the characters ‘T’ and ‘F’ are used, the TrueFalseAdapter converts these to boolean values.

```
public interface UserInfo extends Storable {
     @TrueFalseAdapter
     boolean isAdministrator();
     void setAdministrator(boolean admin);

     ...
}
```

By default, the TrueFalseAdapter is lenient, meaning that it accepts multiple values to mean true or false. It accepts the following as true: ‘T’, ‘t’, ‘Y’, ‘y’, ‘1’, and as false: ‘F’, ‘f’, ‘N’, ‘n’, ‘0’. Passing false to the annotation’s lenient parameter disables this. An IllegalArgumentException is then thrown when reading the property value. Here’s how the lenient property can be set:


```
    @TrueFalseAdapter(lenient=false)
    boolean isAdministrator();
```

4.2.2 YesNoAdapter

The YesNoAdapter is just like the TrueFalseAdapter, except it favors the values 'Y' and 'N'. It too has a lenient mode, which is enabled by default. The set of acceptable characters matched by lenient mode is the same as for TrueFalseAdapter.

4.2.3 DateTimeAdapter

DateTimeAdapter converts Joda-Time DateTime and DateMidnight objects to and from other forms. This adapter is applied automatically for all storable properties of type DateTime or DateMidnight. Carbonado does not natively support Java's Calendar objects, but Dates are supported by explicitly specifying DateTimeAdapter. Use of Joda-Time's objects is encouraged however, as they are more powerful.

With no parameters given, the DateTimeAdapter operates using the system's default time zone. Explicitly passing the time zone parameter overrides this behavior.

```
public interface UserInfo extends Storable {
    @DateTimeAdapter(timeZone="UTC")
    DateTime getModifyDateTime();

    ...
 }
```

The following example shows how to support Date and Calendar, with a DateTime property:

```
    // Get as Date
    Date date = userInfo.getModifyDateTime().toDate();

    // Get as Calendar
    Calendar cal = userInfo.getModifyDateTime().toGregorianCalendar();

    // Set from Date
    userInfo.setModifyDateTime(new DateTime(date));

    // Set from Calendar
    userInfo.setModifyDateTime(new DateTime(calendar));
```

4.2.4 TextAdapter

The TextAdapter provides a convenient means to convert LOBs to ordinary Strings. This is suitable for small text values which are expected to fit conveniently in memory. The section on LOBs explains more.

All String properties have the TextAdapter automatically applied. An optional charset parameter can be passed to the annotation, which controls character encoding. If not specified, it defaults to UTF-8. Since CLOBs natively support characters, no charset based conversion is required. Thus the charset parameter is only observed for BLOB conversion, not CLOBs.


```
public interface UserInfo extends Storable {
     @TextAdapter(charset="UTF-8")
     String getWelcomeMessage();

     ...
 }
```

4.3 Constraints

Constraints are custom annotations that can be applied to a property's *set* method. The constraint is invoked when the property is set, and it may throw an IllegalArgumentException if the set value does not satisfy the constraint. Multiple Constraints may be applied to a property, and they are executed in the same order they appear in the Storable definition.

Like Adapters, the Constraint design is extensible. The process by which new Constraints are defined is very similar to how new Adapters are defined. For more information, consult the Javadocs in the ConstraintDefinition annotation.

4.3.1 LengthConstraint

LengthConstraint limits the value of a property to lie within a specific length range. The property value may be a String, CharSequence, or any kind of array. Two optional parameters can be supplied, "min" and "max".

```
public interface UserInfo extends Storable {
     int getFirstName();

     @LengthConstraint(min=1, max=50)
     void setFirstName(String name);

     ...
 }
```

4.3.2 TextConstraint

TextConstraint limits the value of a property to be a member of a specific set. The property value may be a String, CharSequence, char, Character, or character array. Two optional parameters can be supplied, "allowed" and "disallowed". They define the sets of allowed or disallowed values.


```
public interface UserInfo extends Storable {
     char isActive();

     @TextConstraint(allowed={"Y", "N"})
     void setActive(char value);

     ...
 }
```

4.3.3 IntegerConstraint

IntegerConstraint limits the value of a property to be a member of a specific set. The property value may be a boxed or unboxed byte, short, int, long, float, double, String, CharSequence, char, Character, or character array. Four optional parameters can be supplied, “min”, “max”, “allowed”, and “disallowed”.

When an IntegerConstraint is applied to a non-numerical property (i.e., String), it is parsed as a Java int or long.

```
public interface UserInfo extends Storable {
     int getAge();

     @IntegerConstraint(min=0, max=120)
     void setAge(int value);

     int getRoleID();

     @IntegerConstraint(allowed={ROLE_REGULAR, ROLE_ADMIN})
     void setRoleID(int role);

     ...
 }
```

4.3.4 FloatConstraint

FloatConstraint limits the value of a property to be a member of a specific set. The property value may be a boxed or unboxed float, double, String, CharSequence, char, Character, or character array. Four optional parameters can be supplied, “min”, “max”, “allowed”, and “disallowed”.

When a FloatConstraint is applied to a non-numerical property (i.e., String), it is parsed as a Java float or double.


```
public interface PolarCoordinate extends Storable {
     double getTheta();

     @FloatConstraint(min=0, max=Math.PI * 2, disallowed=Double.NaN)
     void setTheta(double radians);

     ...
 }
```

4.4 Derived properties

A Storable which is defined as an abstract class can also include *derived* properties. A derived property is user defined, and it must have a method body. A property "set" method is optional.

Derived properties can be used just like a normal property in most cases. They can be used in queries, indexes, alternate keys, and they can also be used to define a version property.

```
    @Indexes(@Index("uppercaseName"))
    public abstract class UserInfo implements Storable<UserInfo> {
        /**
         * Derive an uppercase name for case-insensitive searches.
         */
        @Derived
        public String getUppercaseName() {
            String name = getName();
            return name == null ? null : name.toUpperCase();
        }

        ...
    }
```

Note: For repositories which depend on an external schema, Storable indexes are not likely to be defined, and queries against derived properties are implemented via client-side filtering or sorting.

If the derived property depends on join properties and is also used in an index or alternate key, dependencies must be listed in order for the index to be properly updated. This is covered in more detail in the indexing section.

5 CRUD operations: Create, Retrieve, Update, Delete

CRUD is an well-known acronym for Create, Retrieve, Update, and Delete. Outside of queries, all basic CRUD operations operate on a Storable instance. First, the Storable must be prepared from a Storage instance:

```
Repository repo = ...
Storage<UserInfo> storage = repo.storageFor(UserInfo.class);
UserInfo user = storage.prepare();
```

Internally, Storable implementations maintain a state for each property. This state is one of "uninitialized", "dirty", or "clean". This state is used to determine (among other things) whether all primary keys are set prior to beginning a CRUD operation. If not fully specified, an IllegalStateException might be thrown.

A freshly prepared Storable contains property values whose state is "uninitialized". If inspected, all property values are either null, zero, or false, depending on the property type.

After performing a CRUD operation, the same Storable instance can be used for performing more CRUD operations. Attempting to alter the primary key in this state results in an IllegalStateException, however. Calling delete or markPropertiesDirty allows the primary key to be altered again.

All CRUD operations have "try" variants, which return a boolean value instead of throwing certain exception types. If you expect the operation to fail, you might find it more convenient to compare a boolean value instead of catching an exception.

The following table lists all the CRUD operations and defines what false means for each try variant. A return value of true indicates success. The try variants can still throw exceptions, indicating a more serious problem in the persistence layer.

Operation	Try variant	Try variant failure (false)
`void insert()`	`boolean tryInsert()`	Insert failed because a record with matching primary key exists
`void load()`	`boolean tryLoad()`	Load failed because no matching record was found
`void update()`	`boolean tryUpdate()`	Update failed because no matching record was found
`void delete()`	`boolean tryDelete()`	Delete failed because record is already deleted

5.1 Insert

This method inserts a new persistent value for a Storable. Repository implementations usually require that properties which are not Nullable also be specified. Otherwise, a ConstraintException may be thrown.

If a record with a matching key already exists in the database, the insert method throws a UniqueConstraintException, but the tryInsert variant returns false.

```
    // Set required primary key
    user.setUserID(1);

    // Set all properties, since none are Nullable
    user.setFirstName("Indiana");
    user.setLastName("Jones");
    user.setFavoriteColor("green");
    user.setBirthDate(new DateTime("1899-07-01"));

    user.insert();
```

5.2 Load

This method loads or reloads a persisted Storable by its primary key or an alternate key. If no matching record is found, the load method throws a FetchNoneException, but the tryLoad variant returns false.

```
    // Load user 1
    user.setUserID(1);
    user.load();

    // Try to load user 2
    user = storage.prepare();
    user.setUserID(2);
    if (user.tryLoad()) {
        ...
    } else {
        // Fill in remaining properties
        ...
        user.insert();
    }
```

If a load operation fails because there is no matching record, it is assumed that the underlying record has been deleted. As a side-effect, the Storable's state will be the same as if the delete method was called on it.

The load method first checks to see if the primary key is fully specified. If it is, the load is made against the primary key. Otherwise, the load method searches for the first alternate key that is fully specified and loads against it. The order in which alternate keys are searched matches their order in the @AlternateKeys annotation. After searching for alternate keys, if no keys at all are fully specified, an IllegalStateException is thrown.

5.3 Update

The update method updates the persistent value of the Storable, regardless of whether it has actually been loaded or not. Not all properties need to be set on this object when calling update. Setting a subset causes a partial update to be performed.

After a successful update (partial or complete), all properties are set to the actual values in the storage layer. Put another way, the object is automatically reloaded after a successful update. This is done not only to fill in empty properties, but to see any value changes caused by triggers.

Updates of any storable require that the primary keys be specified. Otherwise, an IllegalStateException is thrown. If this object has a version property defined, then the update logic is a bit stricter – the version must be specified as well. If the primary key or version is unspecified, an IllegalStateException is thrown. If the primary key and version is fully specified but the version doesn't match the current record, an OptimisticLockException is thrown.

```
    // Change the favorite color of a user
    user.setUserID(1);
    user.setFavoriteColor("blue");
    if (user.tryUpdate()) {
        System.out.println("Updated: " + user);
    }
```

If no matching record is found, the update method throws a PersistNoneException, but the tryUpdate variant returns false. In either case, it is assumed that the underlying record was deleted. As a side-effect, the Storable's state will be the same as if the delete method was called on it.

5.4 Delete

This method deletes a persisted Storable by its primary key, regardless of whether it has actually been loaded or not. Calling delete does not prevent the Storable instance from being used again. All property values are still valid, including the primary key. In addition, the primary key may be altered again.

If no matching record is found, the delete method throws a PersistNoneException, but the tryDelete variant returns false.

```
    // Get rid of this guy
    user.setUserID(1);
    if (user.tryDelete()) {
        System.out.println("Deleted: " + user);
    }
```


6 Queries

A Carbonado Query is used for complex fetching and deletion operations. Query objects themselves do not hold any results – rather, they encapsulate an action. Queries are immutable, thread-safe, and can be safely shared by multiple threads.

Queries are obtained via factory methods defined in the Storage interface, and instances are cached.

```
public interface Storage <S extends Storable> {
    Query<S> query() throws FetchException;

    Query<S> query(String filter) throws FetchException;

    Query<S> query(Filter<S> filter) throws FetchException;

    ...
}
```

6.1 Filters

A Query filter can be supplied as a filter expression or a Filter object. The Filter object is an advanced feature, and is discussed later. If no filter is passed, the Query operates on all Storables of the given type. It can be used to fetch or delete all Storables in the Storage, although all Queries can be refined further.

Filter expressions match properties to values, paired against a relational operator. Expressions can be combined with logical ‘and’ and ‘or’ operations, and they can be grouped with parenthesis. A simple filter might be “lastName = ?”, and a more complex example might be “lastName = ? | (firstName >= ? & firstName < ?)”.

The full syntax is as follows:

```
Filter          = OrFilter
OrFilter        = AndFilter { "|" AndFilter }
AndFilter       = NotFilter { "&" NotFilter }
NotFilter       = [ "!" ] EntityFilter
EntityFilter    = PropertyFilter
                = ChainedFilter
                | "(" Filter ")"
PropertyFilter  = ChainedProperty RelOp "?"
RelOp           = "=" | "!=" | "<" | ">=" | ">" | "<="
ChainedFilter   = ChainedProperty "(" [ Filter ] ")"
ChainedProperty = Identifier
                | InnerJoin "." ChainedProperty
                | OuterJoin "." ChainedProperty
InnerJoin       = Identifier
OuterJoin       = "(" Identifier ")"
```

Identifiers always refer to Java bean properties, as defined in the Storable. It should be noted that in filter expressions:

- Literals are not allowed
- Logical ‘and’ operator has precedence over ‘or’
- Logical ‘not’ operator has precedence over ‘and’
- Properties cannot be compared against each other
- ‘?’ placeholders can only appear to the right of relational operators

Before a Query is fully ready to be used, all parameters must be filled in. This is done by calling one of the overloaded “with” methods or “withValues”. Invoking one of these methods returns a new Query instance, which contains all the parameter values specified up to that point.

```
Storage<UserInfo> storage = ...
Query<UserInfo> query = storage.query("lastName = ?").with("Jones");

// Operate on query
...

// Another query which matches users with a last name of Jones
// or a first name that starts with "I".
Query<UserInfo> query = storage
    .query("lastName = ? | (firstName >= ? & firstName < ?)")
    .with("Jones")
    .with("I")
    .with("J");

...
```

6.2 Cursors

To retrieve all Storables specified by a ready Query, call the fetch method, which returns a Cursor object. The Cursor is a special kind of iterator[5] for Storables.

```
Query<UserInfo> query = ...

Cursor<UserInfo> cursor = query.fetch();
while (cursor.hasNext()) {
    UserInfo user = cursor.next();
    System.out.println("User: " + user);
}
```

Cursors have a close method, which must be invoked to ensure the Cursor properly releases all resources. As a convenience, fully iterating over the Cursor causes it to automatically close when finished. Also, any exception thrown from a Cursor causes it to close. A closed Cursor behaves as if it refers to an empty result set.

If the Cursor is being used to operate only on a subset of the results, care must be taken to ensure it is properly closed. The following example demonstrates this.

[5] Cursors do not implement Iterable, and thus cannot be used in a Java 5 “foreach” loop. Cursors can throw checked exceptions, but the Iterator interface does not allow this.


```
Cursor<UserInfo> cursor = query.fetch();
try {
    while (cursor.hasNext()) {
        UserInfo user = cursor.next();
        if (user.getFirstName().length() == 7) {
            return user;
        }
    }
} finally {
    cursor.close();
}
```

Failing to close a Cursor has undefined side-effects. It can be expected that the underlying database will leak resources and locks, and over time, run out. Other processes may not be able to acquire locks, resulting in deadlocks and timeouts.

If you wish to operate on Cursor data in a Java Collection instead of iterating over results, convenience methods are provided for copying the results into a Collection or a new List. In general, this is only useful if you wish to operate on the results multiple times or pass them to something else.

6.3 Ordering

To fetch Query results in a specific order, call the Query.orderBy method, which accepts several properties to order the result by. This method returns a new Query object, which contains all parameters which may have been filled in. For controlling ascending or descending order, prefix the property with a '+' or '-' character. If the prefix is omitted, ascending order is assumed.

It should be noted that specification of ordering properties is not cumulative. Calling this method will discard any ordering properties already supplied.

```
Storage<UserInfo> storage = ...
Query<UserInfo> query = storage
    .query("lastName = ?")
    .with("Jones")
    .orderBy("firstName", "-favoriteColor");
```

6.4 Offsets and limits

A query specifies all matching results, and the fetch method attempts to retrieve all of them. If only a numerical range of results is desired, this can be achieved by skipping cursor results or closing it after a certain number have been fetched. This technique is not always very efficient, and it can also lead to poor performing query plans.

The Query.fetchSlice method allows the range to be specified explicitly, which in turn alters the query execution. If SQL is generated, it adds appropriate offset and limit information such that only the correct amount of results is streamed from the database. Repositories may also use this information to generate a better query plan.

Since repositories are not required to fetch a slice in any deterministic order, repeat executions may produce a different slice. To prevent this, only fetch slices from queries which have been given a total ordering. This ensures that the slice results are the consistent, assuming the underlying records have not been modified.

```
Storage<UserInfo> storage = ...

// Define a query with total ordering via primary key
Query<UserInfo> query = storage
    .query("lastName = ?")
    .with("Jones")
    .orderBy("userID");

// Fetch a slice of matching records
long from = (pageNumber - 1) * pageSize;
long to = from + pageSize;
Cursor<UserInfo> page = query.fetchSlice(from, to);
```

The first argument to fetchSlice is a zero-based “from” record number, which always must be specified and is inclusive. If zero, it is effectively ignored. The second argument is an optional zero-based “to” record number, which is exclusive. If null, the slice has no upper bound.

6.5 Single result queries

If you expect the query to return one result, you may find it more convenient to call the Query’s loadOne or tryLoadOne methods. This avoids having to operate on a Cursor directly.

The two methods have slightly different semantics. The loadOne method throws an exception if the actual number of results is not exactly one. The tryLoadOne method is more lenient, returning null if no results are found, but it still throws an exception if there is more than one actual result.

Whenever querying a Storable by a primary or alternate key, it is generally more efficient to load it directly than to operate on a Query. A UserInfo can be queried as:

```
Storage<UserInfo> storage = ...
// It is not recommended to load by primary key this way.
UserInfo user = storage.query("userID = ?").with(1).loadOne();
```

The implementation will likely open a Cursor internally, in order to verify that just one object was loaded. Instead, just do a normal load:

```
UserInfo user = storage.prepare();
user.setUserID(1);
user.load();
```


6.6 Counting results

The Query object has a count method which does just that – it counts the number of matches and returns it.

```
long count = query.count();
```

The actual performance of the count method will vary by repository, and in general it is not safe to assume that count does a simple lookup. The count method should not be used in conjunction with sizing a collection to hold the query results. If the entire query results are going to be pulled out, don't count them beforehand. Just store the results in an expandable collection or call the Cursor toList method.

If you wish to know if a query matches any results without requiring a full count, call the exists method:

```
boolean exists = query.exists();
```

6.7 Result set deletion

In addition to fetching, Query objects can be used for delete operations. The Query object is created in the same way as for fetching, and all parameters must be filled in. Specifying an ordering is allowed, but it is generally not useful.

Calling deleteAll deletes all matching results. If an exception is thrown, the delete is rolled back. If the Query is expected to have one match, the deleteOne or tryDeleteOne method may be called instead. Each will roll back the delete if more than one result matches, throwing a PersistMultipleException. If there are no matching results, deleteOne throws a PersistNoneException, but tryDeleteOne returns false.

If the query has no filter, calling deleteAll will delete all Storables in the Storage. An alternative is to call the Storage's truncate method instead, which may execute much faster. Transactional semantics are not guaranteed, however.

6.8 Boolean logic

Given a Query object, the filter can be further refined by combining additional filters with an 'and' or 'or' operations. The query result set can also be negated by calling the 'not' method.

```
Storage<UserInfo> storage = ...
Query<UserInfo> query = storage.query("lastName = ?).with("Jones");
query = query.or("firstName >= ? & firstName < ?");
query = query.with("I").with("J");

// Now negate the results
query = query.not();

...
```


Refining a Query using boolean logic does not discard any filled-in parameter values or ordering specification. The ‘and’ and ‘or’ operations have a restriction however, that there must be no blank parameters on the affected Query. Otherwise, an IllegalStateException is thrown. The above example makes sure to set the lastName parameter to “Jones” before calling ‘or’.

Using boolean logic is useful when constructing ad-hoc queries based on user input. For example, a web form may ask the user to filter a user search by first name, last name, or both.

```
// Start with a query that returns everything.
Query<UserInfo> query = storage.query();
if (firstName != null) {
    // Reduce results to filter on first name.
    query = query.and("firstName = ?").with(firstName);
}
if (lastName != null) {
    // Reduce results to filter on last name.
    query = query.and("lastName = ?").with(lastName);
}
...
```

6.9 Execution plans and native queries

As a debugging aid, the Query interface provides methods to print the execution plan or, for dependent Repositories, the native query as well. By default, these methods print results to System.out.

Repositories are not required to implement these methods, there is no standard format, and it is subject to change. Proper interpretation of the results may require in-depth knowledge of the persistence layer.

In general, execution plans are printed as a tree view. The leaves represent the first operations, which pass data up towards the root. The root represents the last operation performed before client code sees any results.

In the following example,

```
Query<UserInfo> query = storage
    .query("lastName = ? | (firstName >= ? & firstName < ?)")
    .with("Jones")
    .with("I")
    .with("J");

query.printPlan();
```

…the query plan might be:


```
union
  sort: [+firstName, +userID]
    index scan: UserInfo
    ...index: {properties=[+lastName], unique=false}
    ...identity filter: lastName = Jones
  sort: [+firstName], [+userID]
    index scan: UserInfo
    ...index: {properties=[+firstName], unique=false}
    ...range filter: firstName >= I & firstName < J
```

The logical ‘or’ operation caused the query to be executed with a union operation internally. Each sub-query performs an index scan, and the results are sorted before being fed into the union. Since the index on firstName already partially provides the desired ordering, its results are fed into a sort step which only sorts by userID within groups.

See the appendix for more detail regarding Carbonado's query execution plan format.

7 Joins

Join properties allow Storables to refer to related Storables, in a many-to-one, one-to-one, or one-to-many relationship. The joined Storables are usually different types, but self joins are permitted.

A Join property isn't actually persisted, like a regular property, as it is just a relationship. To define the relationship, it must refer to regular properties.

Assume we have a UserInfo Storable, as defined earlier, and now we define an Address Storable, which will be joined to a UserInfo.

```
@PrimaryKey("addressID")
public interface Address extends Storable {
    long getAddressID();
    void setAddressID(long id);

    String getAddressLine();
    void setAddressLine(String addr);

    String getPostalCode();
    void setPostalCode(String code);
}
```

7.1 Many-to-one joins

Assuming that addresses may be shared by many users, but users can have one home address; a many-to-one join can be established as follows:

```
public abstract class UserInfo implements Storable {
    ...

    public abstract long getHomeAddressID();
    public abstract void setHomeAddressID(long id);

    // Join based on property names.
    @Join(internal="homeAddressID", external="addressID")
    public abstract Address getHomeAddress() throws FetchException;
    public abstract void setHomeAddress(Address address);
    ...
}
```

The Join annotation requires two sets of properties to be defined, and in the above example, both sets have one property. The internal set specifies the properties of *this* Storable that participate in the join, and the external set specifies properties in the other Storable, which is the property's type. In this case, the external properties must exist in the Address Storable.

The internal and external property sets are intended to match each other, and so both sets must at least contain the same number of properties. Also, both must refer to existing regular properties, and the matched types must be compatible. Failing to satisfy these

conditions causes a MalformedTypeException to be thrown when calling Repository.storageFor.

Here is a more complex example joining across a pair of properties:

```
@PrimaryKey({"typeID", "instanceID"})
public interface Media extends Storable {
    // 1=books, 2=music, 3=video, etc.
    int getTypeID();
    void setTypeID(int id);

    // Other half of primary key
    long getInstanceID();
    void setInstanceID(long id);

    ...
}

@PrimaryKey({"mediaSetID", "mediaTypeID", "mediaInstanceID"})
public interface MediaSetElement extends Storable {
    long getMediaSetID();
    void setMediaSetID(long id);

    int getMediaTypeID();
    void setMediaTypeID(int id);

    long getMediaInstanceID();
    void setMediaInstanceID(long id);

    @Join(
        internal = {"mediaTypeID", "mediaInstanceID"},
        external = {"typeID", "instanceID"}
    )
    public abstract Media getMedia() throws FetchException;
    public abstract void setMedia(Media media);

    ...
}
```

When accessing a many-to-one join property which does not exist, null is returned.[6]

Join property values are cached locally in the enclosing Storable. This feature is intended to improve the performance of queries that filter on a joined property. If the join result is null, it is cached only for Nullable joins.

In the first example, an initial call to getHomeAddress results in the loading of an Address Storable by the UserInfo homeAddressID. Subsequent calls to getHomeAddress on the same Storable instance return the same Address instance.

Cached join properties are reset after setting any of the internal property values, even if the internal property value is unchanged. The following example illustrates this:

[6] Versions of Carbonado prior to 1.2 would throw a FetchNoneException for non-nullable many-to-one joins. This change in behavior was made to facilitate outer join queries.


```
MediaSetElement element = ...

// Possibly cached Media object.
Media media = element.getMedia();

// Forces locally cached Media object out.
element.setMediaInstanceID("B123456789");

// Gets new Media object.
media = element.getMedia();
```

Cached joins are also reset after a successful invocation of any of the following Storable methods:

- load or tryLoad
- insert or tryInsert
- update or tryUpdate
- delete or tryDelete
- markPropertiesClean
- markPropertiesDirty

Many-to-one joins usually require a set method, just like regular properties. Calling it does two things. If the value is not null, it sets all internal property values to match the external properties supplied by the given joined Storable. Then it caches the joined instance for later calls. No data is persisted when calling a join property set method.

So in the above example, calling setHomeAddress can be a convenient way to set the homeAddressID, if the Address object was just loaded or inserted.

Unless defined as Nullable, passing null to a join property set method causes an IllegalArgumentException to be thrown. Passing null into a Nullable join property causes it to cache the null, but it does *not* change any of the internal property values.

Internal properties of a join can be “narrower” than their matching external properties, but only for primitive numerical types. For example, an internal property defined as an int can match an external long property.

As was mentioned earlier, many-to-one join properties usually require a set method. They are not allowed if the internal and external types don't quite match.[7] For example, if an internal property cannot be null, but its matching external property can be, then the set method is not allowed. A MalformedTypeException will be thrown instead. Similarly, the set method is not allowed if any internal property is narrower.

7.2 One-to-one joins

One-to-one joins are defined the same as for many-to-one joins, except both the internal and external properties refer to complete primary keys.

[7] This behavior might change in a future version, in which case any possible conversion error is detected at runtime.

A one-to-one join is good for providing auxiliary information, not required by most records. For example, some instances of UserInfo might refer to a user who has special extra information.

```
/** Extra info for administrators */
@PrimaryKey("userID")
public interface PrivilegedUser extends Storable {
    long getUserID();
    void setUserID(long id);

    String getRights();
    void setRights(String rights);

    ...
}
```

The join may be defined as:

```
public abstract class UserInfo implements Storable {
    ...

    @Join(internal="userID", external="userID")
    @Nullable
    public abstract PrivilegedUser getAdminInfo() throws FetchException;
    public abstract void setAdminInfo(PrivilegedUser pu);
}
```

In this case, the join has also been annotated as Nullable, since not all users will have a matching record. As was for many-to-one joins, this only affects caching behavior. If the adminInfo is null, it will still be cached in the local UserInfo instance.

7.3 One-to-many joins

Logically, many-to-one and one-to-many joins are the same. In Carbonado, the difference is where you define the join, and what type of object is returned. In the example above, the join was specified in the UserInfo object, and not the Address object. The other side of the join can be specified in Address object, representing a one-to-many join.

```
public interface Address extends Storable {
    ...

    /** Return all users with this Address */
    @Join(internal="addressID", external="homeAddressID")
    Query<UserInfo> getAddressUsers() throws FetchException;
}
```

Note that the join property is represented as a Query of UserInfo instead of a single UserInfo. Also, the internal and external properties are swapped from the example for many-to-one joins.

If you make a mistake in defining a many-to-one as a one-to-many join (or vice versa), Carbonado has enough information to discover these errors, throwing a MalformedTypeException when Repository.storageFor is called.

The Query returned by the one-to-many join is a normal Query, with all the parameters filled in. You can use the standard Query operations to further refine it, by calling any of its boolean logic operations.

One-to-many join properties have a restriction in that no set method can be defined. Attempting to define one results in a MalformedTypeException.

7.4 Natural joins

Natural joins merely offer a convenient way to specify a join without having to explicitly list the internal and external properties. If no internal and external properties are listed in the Join annotation, then Carbonado tries to deduce what they are.

If the join property type is a Query, then the internal and external properties are set to match *this* Storable's primary key. The referenced join property (specified as a type parameter to Query) must have properties matching the name and type of this Storable's primary key.

If a natural join's property type is not defined by a Query, then the internal and external properties are set to match the referenced Storable's primary key. This join property must have properties matching the name and type of the referenced Storable's primary key.

In the one-to-one join example, the join was completely specified against a commonly named primary key. This meets the criteria for a natural join, and the internal and external properties need not be specified.

```
public abstract class UserInfo implements Storable {
    ...

    @Join
    @Nullable
    public abstract PrivilegedUser getAdminInfo() throws FetchException;
    public abstract void setAdminInfo(PrivilegedUser pu);
}
```

In this next example, a Shipment type is defined, which has a many-to-one relationship with a user. If the Shipment references the user with the same “userID” property, natural joins can be used.


```
@PrimaryKey("shipmentID")
public interface Shipment extends Storable {
    long getShipmentID();
    void setShipmentID(long id);

    long getUserID();
    void setUserID(long id);

    @Join
    UserInfo getUser() throws FetchException;
    void setUser(UserInfo info);

    ...
}
```

UserInfo can specify a natural one-to-many join as well:

```
@PrimaryKey("userID")
public abstract class UserInfo implements Storable {
    public abstract long getUserID();
    public abstract void setUserID(long id);

    /** Return all Shipments */
    @Join
    public abstract Query<Shipment> getShipments() throws FetchException;

    ...
}
```

7.5 Queries with joins

Many-to-one, one-to-one and one-to-many join properties can be included in Query filters, by specifying properties of the joined Storable. This example queries against a many-to-one join:

```
    Storage<UserInfo> storage = ...

    // Find all users with a specific postal code
    Query<UserInfo> query = storage.query("homeAddress.postalCode = ?");

    Cursor<UserInfo> cursor = query.with("12345").fetch();
    ...
```

The dot-separated list of properties is known as a property chain, and any number of join properties can be chained. If Address is joined to a Country object,


```
public interface Address extends Storable {
    ...

    int getCountryID();
    void setCountryID(int id);

    @Join
    Country getCountry() throws FetchException;
    void setCountry(Country country);

    Query<Join>
    UserInfo getUsers() throws FetchException;
}

@PrimaryKey("countryID")
public interface Country extends Storable {
    int getCountryID();
    void setCountryID(int id);

    String getName();
    void setName(String name);

    String getLanguage();
    void setLanguage(String language);

    @Join
    Query<Address> getAddresses() throws FetchException;
}
```

…then we can query for users in a specific country as:

```
    Storage<UserInfo> storage = ...

    // Find all users in a specific country
    Query<UserInfo> query = storage.query("homeAddress.country.name = ?");

    Cursor<UserInfo> cursor = query.with("USA").fetch();
    ...
```

7.5.1 Sub filters

Join queries with sub filters support one-to-many joins, and they also offer a convenient shorthand for specifying multiple many-to-one join properties. Consider this example query, which is written in the long form:

```
    Query<UserInfo> query = storage.query
        ("homeAddress.country.name = ? | homeAddress.country.language = ?");
```

Several shorthand equivalent queries are possible, and each is automatically converted to the original long form. The sub filter expression is bounded by parenthesis and is scoped to the join property specified immediately before it.


```
query = storage.query
    ("homeAddress.country(name = ? | language = ?)");

query = storage.query
    ("homeAddress(country.name = ? | country.language = ?)");

query = storage.query
    ("homeAddress(country(name = ? | language = ?))");
```

The above shorthand form applies to queries against many-to-one joins. The same syntax also supports queries against one-to-many joins. The SQL equivalent would be specified via a "where exists" or "where in" clause. For example, to find all countries which have registered users of a given name:

```
Query<Country> query = storage.query("addresses(users(lastName = ?))");
```

The first term in the above query filter is "addresses", which refers to the one-to-many join of Country. In the first sub filter scope, "users" refers to the one-to-many join of Address. The property filter expression "lastName = ?" is scoped to the UserInfo object.

Queries against one-to-many join properties need not specify any property filters at all. For example, to find all countries which have any registered users at all:

```
Query<Country> query = storage.query("addresses(users())");
```

The above syntax which has an empty sub filter does not work many-to-one or one-to-one joins. A MalformedFilterException will be thrown instead.

7.5.2 Outer joins

Queries against join properties require that joined Storables actually exist. In SQL terminology, this is called an *inner join.* An *outer join* does not have the existence requirement. To specify an outer join, place parenthesis around the join property:

```
Storage<UserInfo> storage = ...

// Find all users with a specific postal code
Query<UserInfo> query = storage.query("(homeAddress).postalCode = ?");

Cursor<UserInfo> cursor = query.with("12345").fetch();
...
```

The above query returns UserInfo objects which have a matching postal code for their home address, and it also returns UserInfo objects that have no home address at all. The join property can have a mix of inner and outer joins, as shown in this example:

```
Query<UserInfo> q1 = storage.query("(homeAddress).country.name = ?");
Query<UserInfo> q2 = storage.query("homeAddress.(country).name = ?");
Query<UserInfo> q3 = storage.query("(homeAddress).(country).name = ?");
```


It is illegal to place parenthesis around the last property in a chain, since it always refers to a simple property instead of a join property. A MalformedFilterException will be thrown instead.

8 Indexing

Indexes can be used to greatly improve the performance of queries, although they may make insert, update, and delete operations slower. Dependent Repositories often have their own means to define indexes, and Carbonado index annotations are ignored by dependent Repositories, like JDBC. The remainder of this section is only relevant for independent Repositories, like Berkeley DB.

Some indexes are defined implicitly, for example, primary and alternate keys. Explicit indexes are defined by placing an *Indexes* annotation on the Storable type definition. This annotation then contains a number of *Index* annotations, each representing a separate index.

```
@PrimaryKey("userID")
@Indexes({
    @Index("firstName"),
    @Index("lastName")
})
public abstract class UserInfo implements Storable {
    public abstract long getUserID();
    public abstract void setUserID(long id);

    public abstract String getFirstName();
    public abstract void setFirstName(String name);

    ...
}
```

Composite keys are defined by listing multiple properties in the Index annotation. Also, indexes can specify ascending or descending sort order by prefixing a ‘+’ or ‘-’ character to the property name.

```
@PrimaryKey("ID")
@Indexes({
    @Index("name"),
    @Index("-lastModified"),
    @Index({"length", "lastModified"})
})
public interface FileInfo extends Storable<FileInfo> {
    long getID();
    void setID(long id);

    String getName();
    void setName(String name);

    long getLength();
    void setLength(long length);

    long getLastModified();
    void setLastModified(long timestamp);

    ...
}
```


Like primary and alternate keys, index annotations are not inherited by Storable subclasses.

8.1 Index set reduction

When deciding what indexes to build, Carbonado analyzes the provided indexes and tries to reduce the set. This process eliminates creation of redundant indexes, by merging definitions together.

For example, consider the indexes “firstName”, and {“firstName”, “lastName”}. The latter index declaration is sufficient to support both indexes, and so it is the only one defined.

Independent Repositories that support indexes should have an extra capability, IndexInfoCapability, which can be interrogated to find out what set of indexes was ultimately decided upon.

8.2 Clustered indexes

Most databases support *clustered indexes* in some capacity. A clustered index defines (more or less) the physical ordering of persistent records, and there can be at most one clustered index per record type. Clustered indexes are usually faster than normal indexes and are often preferred by a query analyzer.

With Carbonado, the primary key index is always clustered, and there is no way to change this behavior.[8] To design around this, you can create a more complex primary key and supply an alternate key.

In the example UserInfo Storable, you might query by lastName so often that you’d like a clustered index for it. Instead of defining the primary key as just “userID”, create a composite {“lastName”, “userID”}. Since “userID” is unique by itself, define it as an alternate key.

8.3 Adding and dropping

To add or drop an index, simply change the Indexes annotation, recompile, and run your application. Carbonado compares the new set of indexes with what was used previously and reconciles any differences.

Index set reconciliation happens the first time a Storage instance is requested, and it will block this access until complete. Depending on how much data you have, it may take a long time to build new indexes. This process is sped up somewhat by presorting the data.

Distributed Repositories may require special procedures to add or drop indexes. If different hosts are maintaining different index sets, this can cause inconsistencies. Any special procedures should be documented with Repositories susceptible to this problem.

[8] This limitation may be addressed in a future version, possibly by supporting an optional “clustered” parameter to the @Index or @AlternateKey annotations.

8.4 Indexing derived properties

Derived properties may be indexed just like any other property. This allows for *function based indexes,* which may be used for advanced queries. The example provided in the derived properties section shows how to use a derived property index for case-insensitive searches, which is repeated here:

```
@Indexes(@Index("uppercaseName"))
public abstract class UserInfo implements Storable<UserInfo> {
    /**
     * Derive an uppercase name for case-insensitive searches.
     */
    @Derived
    public String getUppercaseName() {
        String name = getName();
        return name == null ? null : name.toUpperCase();
    }

    ...
}
```

Here's another example where a derived property index may be used for supporting efficient searches against a date time field:

```
@Indexes(@Index("auditHour"))
public abstract class AuditLog implements Storable<AuditLog> {
    ...

    public abstract DateTime getTimestamp();
    public abstract void setTimestamp(DateTime dt);

    /**
     * Derive the hour from the timestamp.
     */
    @Derived
    public String getAuditHour() {
        return getTimestamp().getHourOfDay();
    }

    ...
}
```

```
// Query for audit logs recorded at 3AM
Query<AuditLog> query = storage.query("auditHour = ?").with(3);
```

Properties derived from joins can also be indexed, which provides a powerful form of *denormalization*. The performance of join queries can be greatly improved using this technique.

Consider a query which includes a joined property:


```
    Query<UserInfo> query = users
        .query("lastName = ? & homeAddress.country.language = ?")
        .with("Jones").with("English");
```

Assuming an index exists for “lastName”, a possible query plan might be:

```
filter: homeAddress.country.language = English
  index scan: org.myorg.myapp.stored.UserInfo
  ...index: {properties=[+lastName, ~userID], unique=true}
  ...identity filter: lastName = Jones
```

What the plan shows is that the index is finding all the users with a given last name, but it must follow two joins to filter by the country's language. If this is too expensive, a better index would include the language in the UserInfo object. This is possible using a derived property:

```
@Indexes({
    @Index({"lastName", "homeLanguage"})
})
public abstract class UserInfo implements Storable {
    ...

    @Derived(from="homeAddress.country.language")
    public String getHomeLanguage() throws FetchException {
        return getHomeAddress().getCountry().getLanguage();
    }
}
```

Notice that the derived property declares a “from” dependency. This is required in order for the index to monitor changes in all dependencies and make updates. If the dependencies are not correct, the index will be inconsistent.

The index against the derived property also requires that all joins be doubly joined. If not, a MalformedTypeException is thrown:

```
com.amazon.carbonado.MalformedTypeException: org.myorg.myapp.stored.UserInfo:
Derived-from property is a join, but it is not doubly joined:
"homeLanguage" is derived from "homeAddress". Consider defining a join
property in interface org.myorg.myapp.stored.Address as:
@Join(internal="addressID", external="homeAddressID")
Query<org.myorg.myapp.stored.UserInfo> getXxx() throws FetchException
```

Following the error message's advice, the following one-to-many join is added:

```
public interface Address extends Storable {
    ...

    @Join(internal="addressID", external="homeAddressID")
    Query<UserInfo> getHomeUser() throws FetchException;
}
```

For the same reason, this join is also required:


```
public interface Country extends Storable {
    ...

    // Natural join to addresses
    @Join
    Query<Address> getAddresses() throws FetchException;
}
```

The reason the double join requirement exists is because class loading order of Storables is not guaranteed. In turn, this causes the Storable dependency graph to be inconsistently defined. The double join ensures that any Storable can discover all of its dependents and dependencies.

Rewriting the query to use the derived property:

```
    Query<UserInfo> query = users
        .query("lastName = ? & homeLanguage = ?")
        .with("Jones").with("English");
```

...yields an improved query plan of:

```
index scan: org.myorg.myapp.stored.UserInfo
...index: {properties=[+lastName, +homeLanguage, ~userID], unique=true}
...identity filter: lastName = Jones & homeLanguage = English
```

Indexes against derived properties which depend on joins do have a drawback. First, they may create additional indexes along the join path itself. A bigger drawback is the potential cost to updating a dependent property. In this example, updating the language for a single country will cause an index update for every user who lives in that country. This update will execute in a very large transaction which may exceed the limits of the underlying database. The only option might be to drop and recreate the index instead.

9 Transactions

Data retrieval and persist operations may be guarded by a transaction scope. This allows groups of operations to be treated atomically. Transaction scopes are local to the requesting thread and the Repository it was requested from.

```
Repository repo = ...
Storage<UserInfo> users = repo.storageFor(UserInfo.class);
Storage<Address> addresses = repo.storageFor(Address.class);

// Update name and address atomically
Transaction txn = repo.enterTransaction();
try {
    UserInfo user = users.prepare();
    user.setUserID(1);
    user.setName("Henry");
    user.update();

    Address addr = user.getAddress();
    addr.setPostalCode("12345");
    addr.update();

    // Commit changes
    txn.commit();
} finally {
    // Ensure transaction exits, rolling back if an exception
    txn.exit();
}
```

Transaction scopes must always be properly exited, as shown in the example above. Failure to do so has undefined consequences, but often problems show up later as excessive resource consumption and lock timeouts. Exact behavior varies by Repository.

When a transaction scope is exited, all open cursors created within the transaction are automatically closed. Any cursors created within the transaction are therefore unusable outside the transaction.

9.1 Commits

Committing a transaction is necessary to make any updates actually occur. Forgetting to commit causes all updates to be rolled back. After a commit, more changes can be made in the transaction, but any rollbacks only go as far back as the last commit.

For long running transactions, it might be necessary to commit batches of changes every so often to release resources, improve concurrency, and improve performance. Deciding whether or not to do this depends on the outcome of testing your application.

A side-effect of calling commit is the closing of all cursors opened within the transaction. If commits are being called for batching, cursors may need to be re-opened to where they left off. To do this, re-issue a query with applicable parameters.

9.2 Nested transactions

Within a transaction scope, another transaction scope may be entered. The number of nesting levels supported will vary by Repository, but this is not likely to be a problem unless you nest transactions recursively.

Nested transactions are most often created in the context of re-usable utility functions, which need to be in a transaction but cannot assume that the caller is in a transaction. Supporting nested transactions essentially makes them re-entrant.

Nested transactions are also useful to scope rollbacks, since they only go back to the start of the inner transaction. If your application depends on this behavior to work correctly, make sure the Repository in use truly supports nested transactions or savepoints. If it does not, rolling back a nested transaction has no effect. Rollbacks will only be effective at the root transaction.

Committing changes to an outer transaction forces all inner transactions to recursively commit. Likewise, exiting an outer transaction recursively exits all inner transactions. This behavior is mainly just a safeguard, since all transaction scopes should be protected by a proper try-finally statement.

9.3 Isolation level

Transaction scopes may be entered with or without an explicit isolation level[9] requested. When unspecified, the Repository chooses a suitable default, perhaps consulting any enclosing transaction. Nested transactions may request a different isolation level than was requested by an enclosing transaction.

An isolation level controls how effectively changes within the transaction are seen by other threads or processes. A low isolation level can improve concurrency, but a high level can ensure consistent behavior. Deciding whether the isolation level needs to be adjusted is usually revealed during application performance testing.

Isolation level	Definition
READ_UNCOMMITTED (degree 1)	Lowest isolation level, which indicates that dirty reads, non-repeatable reads and phantom reads can occur. This level allows modifications by one transaction to be read by another transaction before any changes have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid modification.
READ_COMMITTED (degree 2)	Indicates that dirty reads are prevented. Non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading modifications with uncommitted changes in it.
REPEATABLE_READ	Indicates that dirty reads and non-repeatable reads are

[9] Isolation level is a standard concept when referring to ACID properties of a transaction. See http://en.wikipedia.org/wiki/Transaction_isolation_level for more info.

	prevented. Phantom reads can occur. This level prohibits a transaction from reading uncommitted changes, and it also prohibits the situation where one transaction reads a record, a second transaction alters the record, and the first transaction rereads the record, getting different values the second time (a "non-repeatable read").
SNAPSHOT	Indicates that dirty reads, non-repeatable reads and phantom reads are prevented. Commits can still fail however, as snapshot isolation avoids using locks.
SERIALIZABLE (degree 3)	Highest isolation level, Indicates that dirty reads, non-repeatable reads and phantom reads are prevented. Phantoms are records returned as a result of a search, but which were not seen by the same transaction when the identical search criteria was previously used. For example, another transaction may have inserted records which match the original search.

Repositories are not required to support all isolation levels. The actual level may be escalated higher than requested. When requesting a level which is higher than supported, an UnsupportedOperationException is thrown.

Calling Reposity.getTransactionIsolationLevel returns the thread's current transaction status against that Repository. If the Repository reports an IsolationLevel of null, then the current thread is not in a transaction with that Repository.

9.4 Update mode

Transactions support a special "for update" mode which can be enabled or disabled at any time by calling setForUpdate. This mode affects lock acquisition by supporting upgrading of read locks to writes locks. Usually this means all read locks are actually write locks.

Update mode avoids deadlock caused by upgrading a read lock to a write lock, and it should always be used if a record is conditionally being updated. For example:

```
    Transaction txn = txn.enterTransaction();
    try {
        txn.setForUpdate(true);
        UserInfo user = repo.storageFor(UserInfo.class).prepare();
        user.setUserID(100);
        // Load user while holding a more exclusive lock
        user.load();
        if (!name.equals(user.getFirstName())) {
            // Update the name now without risk of deadlock
            user.setFirstName(name);
            user.update();
            txn.commit();
        }
    } finally {
        txn.exit();
    }
```


Running transactions in update mode potentially reduces concurrency, and so it should be used carefully.

9.5 Top-level transactions

Transactions are thread local, which is a convenient alternative to explicit passing of the transaction instance around. Under some circumstances it is desirable to begin a transaction in the current thread which is not nested to any existing transaction. The easiest way to do this is to create a *top-level* transaction:

```
// Enter a top-level transaction with the default isolation level
Transaction txn = repo.enterTopTransaction(null);
try {
    ...

    txn.commit();
} finally {
    txn.exit();
}
```

This kind of transaction is useful when a commit must absolutely succeed, even if the current thread is already in a transaction scope. If there was an outer transaction, then an inner commit might still be rolled back by the outer transaction.

Requesting top-level transactions can be deadlock prone if the current thread is already in a transaction scope. The top-level transaction may not be able to obtain locks held by the outer transaction.

Any new transactions which are created in the scope of the top-level transaction will nest inside it, unless they are top-level transactions themselves. After exiting a top-level transaction, the thread local transaction reverts to what it was before the top-level transaction was created.

With an ordinary nesting of transactions, exiting an outer transaction forces all inner transactions to exit first. If an inner transaction is top-level, it is not automatically exited. Top-level transactions must always be guarded by a proper try-finally statement to ensure proper exit.

9.6 Detaching transactions

An entire transaction scope can be detached from the current thread and reattached to any other thread. This is an alternative to top-level transactions, but it may also be used with a remote access layer. A servlet container, for example, does not guarantee that the same thread is used for a session. If the session includes a transaction, it must be detached and reattached for each call.

Detaching a transaction causes all nested outer and inner transactions to be detached with it. Reattaching it causes all nested transactions to be attached as well. An IllegalStateException is thrown if when attempting to attach a transaction to a thread which already has a transaction, and it is also illegal to detach a transaction from a thread which is attached to a different thread.


```
    // Remote call which enters a transaction
    public String enterTransaction() {
        Transaction txn = repository.enterTransaction();
        String txnGuid = addToSession(txn);
        txn.detach();
        return txnGuid;
    }
```

```
    // Remote call which operates on a transaction
    public void updateData(String txnGuid, String data) {
        Transaction txn = (Transaction) retrieveFromSession(txnGuid);
        txn.attach();
        try {
            // Perform updates
            ...
        } finally {
            txn.detach();
        }
    }
```

```
    // Remote call which commits and exits transaction
    public void commitTransaction(String txnGuid) {
        Transaction txn = (Transaction) retrieveFromSession(txnGuid);
        txn.commit();
        txn.exit();
    }
```

The above example omits code which ensures that transactions are not lost if the remote session is severed. One solution would be to register transactions with a timer. After waiting a predetermined amount of time, the transaction would be automatically exited.

10 Exceptions

All Carbonado checked exceptions derive from a common base class, RepositoryException. It is sub-classed into three major groups:

- FetchException
- PersistException
- SupportException

A FetchException can be thrown by any operation that attempts to read from a Repository, and PersistExceptions can be thrown by any write operation. SupportExceptions are thrown when trying to use a feature not supported by a Repository. They can be thrown when building a Repository or requesting Storage for a specific type of Storable.

10.1 Exception conversion

When writing applications that need to handle Carbonado exceptions, the easiest thing to do is to handle only RepositoryExceptions, or re-throw them. The intent of the specialized FetchException and PersistException is to declare if a method intends to read or write to the Repository:

```
    /**
     * Update user information.
     */
    public void updateUserInfo(long userID, String firstName, String lastName)
        throws PersistException;
```

The implementation of the updateUserInfo method might need to first read a UserInfo record, to decide what to do next. This operation may throw a FetchException, which cannot be thrown from a method that only declares throwing a PersistException. As a convenience, all RepositoryExceptions can be converted to a FetchException or PersistException:


```
    /**
     * Update user information.
     */
    public void updateUserInfo(long userID, String firstName, String lastName)
        throws PersistException
    {
        UserInfo ui;
        try {
            ui = this.userInfoStorage.prepare();
            ui.setUserID(userID);
            ui.load();
        } catch (FetchException e) {
            // Throw any unexpected FetchException as a PersistException,
            // to indicate failure in what we were intending to do: persist.
            throw e.toPersistException();
        }

        ...
    }
```

10.2 Deadlock and optimistic lock retry

Deadlock and optimistic lock failures are represented by FetchDeadlockException, PersistDeadlockException, and OptimisticLockException. OptimisticLockException is a subclass of PersistException. These exceptions may indicate a transient condition, and so one strategy is to retry the affected transaction again.

When retrying a transaction, the trick is to make sure you provide an upper bound on the number of retries, as well as sleeping to allow the transient condition to pass. As a convenience, a method is provided in RepositoryException for supporting back off and retry.

```
    // Retry at most three more times
    for (int retryCount = 3;;) {
        // Ensure that isolation level doesn't hold lock on loaded
        // records, allowing them to be concurrently modified.
        Transaction txn = this.repository
            .enterTransaction(IsolationLevel.READ_COMMITTED);
        try {
            ...
            myObject.load();
            ...
            myObject.update();

            txn.commit();
            break;
        } catch (OptimisticLockException e) {
            // Wait up to one second before retrying
            retryCount = e.backoff(e, retryCount, 1000);
        } finally {
            txn.exit();
        }
    }
```


The backoff method re-throws the original exception if the retry count reaches zero. Otherwise, it sleeps for a bounded random amount of milliseconds and returns a decremented retry count.

11 LOBs

LOB is an acronym for Large OBject, and a LOB property refers to a stream of data, similar to a file. There are two types of LOBs, BLOBs (Binary LOB) and CLOBs (Character LOB).

Storable properties declared as *Blob* or *Clob* have restrictions compared to regular properties. In particular, they:

- cannot be a member of a primary or alternate key
- cannot be indexed
- cannot be used in joins
- cannot be used in query filters

A LOB is accessed as if it was a file, and its data is often stored externally from the main Storable. Since LOB data is often read and written in chunks, consider accessing within a transaction scope to guard against changes.

Normal properties are updated by setting the property value. Updating a LOB property typically involves operating on the LOB itself instead of explicitly setting the property value. Setting a LOB property is useful only when completely replacing the data, which can be a relatively expensive operation. To safeguard against doing this accidentally, the newly set LOB property value must be a different instance than before.

11.1 Stream access

The primary means of accessing LOB data is via streams. Streams can be positioned to start anywhere in the LOB, for supporting random access.

Given a reference to a Blob, a stream for reading can be opened up for it by calling openInputStream, which returns a regular InputStream object. Clobs have a similar method, openReader, which returns a regular Java I/O Reader object.

To make changes to a Blob, openOutputStream returns a regular OutputStream. Clobs operate on Writers, returned from openWriter. Be sure to close these streams when finished, to ensure any buffered changes are written out.

The length of a Blob (in bytes) or Clob (in characters) can independently be accessed and changed via the getLength and setLength methods.


```
...
import com.amazon.carbonado.lob.Blob;

@PrimaryKey("imageID")
public interface Image extends Storable {
    int getImageID();
    void setImageID(int id);

    Blob getImageData();
    void setImageData(Blob blob);

    int getWidth();
    void setWidth(int width);

    ...
}
```

```
    Repository repo = ...
    Image image = repo.storageFor(Image.class).prepare();
    image.setImageID(5);
    image.load();

    // Access image data
    Blob data = image.getImageData();
    long length = data.getLength();
    InputStream in = data.openInputStream();
    ...
```

11.2 String conversion

As a convenience, the Blob and Clob interfaces provide support for reading and writing LOB data as a String. Implementations manage the streaming and buffering of the data. For Blobs, a charset may be provided to control character encoding. If not provided, it defaults to UTF-8. Since Clobs are designed specifically to support characters, explicit character encoding is not supported.

If the Blob or Clob text is intended to fit entirely in memory, consider declaring the property type as a String instead. A TextAdapter is automatically applied, such that if the Repository requires a LOB, the property still appears to be an ordinary String. If the underlying LOB is a Blob which is not encoded in UTF-8, an explicit TextAdapter annotation can be used to change the charset.

In this example, a Storable has several LOB properties, all of which are representing textual data:


```
public interface BunchOfLobs extends Storable {
    ...

    Clob getTextOne();
    void setTextOne(Clob text);

    Blob getTextTwo();
    void setTextTwo(Blob text);

    // Assume backing database stores this as a Clob
    String getTextThree();
    void setTextThree(String text);

    // Assume backing database stores this as a Blob
    @TextAdapter(charset="ISO-LATIN-1")
    String getTextFour();
    void setTextFour(String text);
}
```

The first two properties can be converted to and from a String as follows:

```
    BunchOfLobs obj = ...

    // Operate on Clob as a String
    String text = obj.getTextOne().asString();
    // Make text bold (note, no need to call update on Storable)
    obj.getTextOne().setValue("<B>" + text + "</B>");

    ...

    // Blow away Clob value by passing a new one
    obj.setTextOne(new StringClob("The quick brown fox jumps..."));

    ...

    // Operate on Blob as a String, passing optional charset
    String text = obj.getTextTwo().asString("ISO-LATIN-1");
    // Persist new value encoded as UTF-8
    obj.getTextTwo().setValue(text, "UTF-8");
```

The last two properties (textThree and textFour), were defined as Strings, and are a bit simpler to use. They work like any other property. An advantage of the TextAdapter annotation is that it prevents accidentally passing the wrong charset.

Declaring the Blob or Clob as a String can have a drawback, however. Queries always load the property value, which can be expensive. Leaving the property as a LOB allows the data to be accessed only when necessary.

12 Triggers

Carbonado supports triggers, which behave much like they do in SQL based databases. A trigger provides a means to run custom operations when records are persisted, all within the same transaction scope.

Unlike SQL databases, Carbonado triggers are not persistent. They must be added each time the system starts up. Also, Carbonado triggers do not interfere or define any SQL triggers, if the JDBC Repository is being used.

12.1 Trigger class

A trigger is invoked when a Storable is inserted, updated, deleted or loaded. In addition, it is called before and after a persist operation. If the persist operation fails, the trigger is notified of that as well. A total of ten conditions can be captured by a trigger, each of which has a corresponding method in the abstract Trigger class.

- Insert triggers
  - beforeInsert
  - afterInsert
  - failedInsert
- Update triggers
  - beforeUpdate
  - afterUpdate
  - failedUpdate
- Delete triggers
  - beforeDelete
  - afterDelete
  - failedDelete
- Load triggers
  - afterLoad

All the methods in the Trigger class are pre-defined to do nothing. Merely override methods of interest to run under those conditions.

Each method is passed the Storable instance being persisted, which is the exact same instance that a user call is making. All trigger methods that act on persists are invoked in the same transaction scope, even if the transaction is auto-commit.


```
/**
 * Trigger recursively deletes all child records referenced
 * by a FileInfo being deleted.
 */
public class CascadeDeleteTrigger extends Trigger<FileInfo> {
    public Object beforeDelete(FileInfo fileInfo) throws PersistException {
        // Call a Join property, which returns a Query, and then delete
        // the query results.
        fileInfo.getSubFiles().deleteAll();
        return null;
    }
}
```

Each of the “before” methods can return an opaque state object, which is then passed back to the matching “after” or “failed” method. This allows a before/after pair to remember any state needed to complete the triggered operation. The failed methods exist primarily to allow the optional state object to be cleaned up.

For loads, the trigger method is only called after the load. There is no corresponding “before” or “failed” load method. The after load trigger method is invoked for Storables fetched by queries as well for Storables directly loaded.

12.2 Registration

Triggers are registered by calling the addTrigger method on a Storage object. A newly added trigger is logically at the outermost nesting level. This means that its “before” methods are run before all other triggers, and its “after” methods are run after all other triggers.

```
    Storage<FileInfo> storage = ...

    Trigger<FileInfo> trigger = new CascadeDeleteTrigger();
    storage.addTrigger(trigger);
```

Registering the same Trigger multiple times has no effect, and the addTrigger method returns false in that case. The equals method of the Trigger implementation is called to determine if a duplicate is being added.

Triggers can be removed by calling the removeTrigger method. It does not matter to the remove method the order in which triggers were added. Triggers can be removed in any order.

```
    storage.removeTrigger(trigger);
```

12.3 Accessing old Storable value

Whenever a trigger method is run, it is operating on the live user Storable, which may contain non-persisted data. In order to access the existing persisted data, create a copy of the Storable and reload it.


```
public class StageCheckTrigger extends Trigger<StoredWorkflow> {
    public Object beforeUpdate(StoredWorkflow wf) throws PersistException {
        StoredWorkflow old = wf.copy();
        try {
            if (old.tryLoad()) {
                if (wf.getStage() < old.getStage()) {
                    throw new IllegalStateException
                        ("Illegal stage transition");
                }
            }
        } catch (FetchException e) {
             throw e.toPersistException();
        }
        return null;
    }
}
```

12.4 Generic Triggers

The trigger registration method accepts a trigger defined with a “super” wildcard parameter:

```
    boolean addTrigger(Trigger<? super S> trigger);
```

This wildcard parameter allows the trigger to operate on a type which need not be a specific Storable type. It can be a super class or an interface, and it doesn’t even need to be a Storable itself.

So if you have a bunch of Storables which all have a common set of properties, they can be defined in an interface that all the Storables implement. The trigger can then be defined against the common interface and be re-used for all Storable types that implement it.

13 Available Repositories and Capabilities

This section describes the minimum on how to configure standard available Repositories and describes some of their capabilities. Consult the Javadocs for each Repository for more information.

13.1 Berkeley DB

The Berkeley DB Repository is the easiest persistent Repository to set up, and is recommended when starting work on new applications. Berkeley DB is often abbreviated as “BDB”, and the product was maintained by Sleepycat Software, which is now part of the Oracle Corporation. The terms “BDB” and “Sleepycat” are both used to refer to Berkeley DB.

The “BDBRepositoryBuilder is available in the com.amazon.carbonado.repo.sleepycat package. It currently supports two Berkeley DB products: DB, and JE. By default, it assumes you wish to use JE. To select a different product, call the setProduct method on the builder, passing “DB” or “JE”.

At a minimum, the BDBRepositoryBuilder requires an environment home directory to be set. If the directory does not exist, the BDBRepositoryBuilder makes it when build is called.

In general, you’ll also want call setTransactionWriteNoSync(true) to improve the performance of writes by delaying flushes to the physical disk. It does come with a price however. In the event of a system crash you might lose the last few seconds (or minutes) of changes when the database is recovered.

13.2 JDBC

The JDBC Repository is a dependent Repository, for linking to SQL based databases. JDBCRepositoryBuilder is available in the com.amazon.carbonado.repo.jdbc package.

Although many features should work on a wide variety of database products, only a few are fully supported at this time. Expect LOBs and Sequences to not work correctly on other databases.

The recommended way to configure the JDBC Repository is to pass a javax.sql.DataSource instance to the JDBCRepositoryBuilder. This is the standard interface implemented by JDBC connection pools. If none is available, set the driver class name, URL, username, and password directly on the JDBCRepositoryBuilder.

13.2.1 Connection access

If you wish to access the JDBC connection that the current Carbonado Transaction is using, a special capability is provided, JDBCConnectionCapability. It provides direct

access the JDBC connection being used by the current transaction, which is thread-local. If no transaction is in progress, then the connection is in auto-commit mode.

All connections retrieved from this capability must be properly returned back by calling yieldConnection. Do not close the connection directly, as this interferes with the transaction's ability to properly manage it.

It is perfectly okay for other Carbonado calls to be made while the connection is in use. Also, it is okay to request more connections, although they will usually be the same instance. Failing to yield a connection has an undefined behavior.

```
Repository repo = ...

JDBCConnectionCapability cap = repo
    .getCapability(JDBCConnectionCapability.class);

Transaction txn = repo.enterTransaction();
try {
    java.sql.Connection con = cap.getConnection();
    try {
        ...
    } finally {
        cap.yieldConnection(con);
    }
    ...
    txn.commit();
} finally {
    txn.exit();
}
```

13.3 Replicated

The Replicated Repository supports total replication between two repositories. One repository is used for reading from, and the other is used for writing to. The write repository can be considered the master, and the read repository the replica.

In order to guarantee there are no inconsistencies between the master and replica, all writes should be made through the Replicated Repository. Changes made directly to the master will not be automatically propagated.

To help detect when this happens, consider using optimistic locking via the Version property. Should the master get updated via a backdoor, an update to the replica results in an OptimisticLockException. To resolve any possible inconsistencies, the Replicated Repository automatically repairs records whenever an OptimisticLockException is thrown. The exception is still passed to the caller however.

ReplicatedRepositoryBuilder is available in the com.amazon.carbonado.repo.replicated package. It just requires a master and replica repository to be configured, by providing builders for them. The methods are setMasterRepositoryBuilder and setReplicaRepositoryBuilder.

13.3.1 Master unavailable

In the event that the master repository becomes unavailable, all reads from the Replicated Repository should succeed, assuming that the replica is still available. As soon as the master is available again, writes will no longer throw exceptions. If the Replicated Repository is closed and re-opened while the master is still unavailable, a background thread attempts to open the master. Again, as soon as the master is available, writes will work again.

When the master is unavailable, any attempt to enter a transaction will likely fail. This is because it is assumed that entering a transaction is a prelude to performing a write. To ensure that the Replicated Repository provides read-only availability when the master is down, query and load operations should not be enclosed in transactions.

13.3.2 Resynchronization

To repair replication inconsistencies or fill up a new replica, use the ResyncCapability. It operates on a specific Storable type, and a filter can optionally be provided to limit which records need to be synchronized.

Also, a throttle parameter must be provided which can limit the impact of running a large resync job. It supports any value from 0.0 to 1.0. A value of 1.0 causes the resync to run at maximum speed. A value of 0.5 makes it try to run at about half speed.

13.4 Volatile Map

The MapRepositoryBuilder creates Repository instances which are backed by a volatile concurrent navigable map, and only works with JDK1.6 or higher. When the JVM exits, the contents of this repository are lost. It is the fastest repository implementation, and it is ideally suited for unit tests. The easiest way to create an instance is to call a convenience method on the builder:

```
Repository repo = MapRepositoryBuilder.newRepository();
```

This repository fully supports transactions, which also may be nested. Supported isolation levels are read committed and serializable. Other isolation levels get promoted: read uncommitted is promoted to read committed, and repeatable read is promoted to serializable.

Locks used by this repository are coarse, much like *table locks*. Loads and queries acquire read locks, and modifications acquire write locks. Within transactions, loads and queries always acquire upgradable locks, to reduce the likelihood of deadlock.

Due to the coarse nature of the locks, this repository offers the least level of concurrency, with respect to writes and transactions. Applications which acquire locks out of order are more likely to deadlock than with other repositories. This can be a beneficial feature,

since it makes it easier to reproduce such errors. For example, entering a transaction and updating Storable types in an inconsistent order is deadlock prone.

14 Advanced

This section introduces advanced features of Carbonado, which are not required for most applications.

14.1 Filter Construction

Carbonado query filters are typically specified with a filter expression string. Throughout the API, methods that accept a filter string are overloaded to also accept a Filter object. Filter objects can be constructed programmatically, which may be more convenient for ad-hoc queries than constructing a filter expression.

In general, most the necessary functionality for constructing ad-hoc queries programmatically is available in the Query object. Operating on Filter objects solely for this purpose is of little use. Manipulating Filters objects is more useful when defining new kinds of Repositories or creating specialized filtering cursors.

14.2 Specialized Cursors

Sometimes the built-in query support is not sufficient, and so a collection of specialized Cursors exists for hand-crafting result sets. Some of the Cursors available in the com.amazon.carbonado.cursor package provide features not available to ordinary queries.

14.2.1 Filtering

The most basic cursor operation involves filtering out results. The abstract FilteredCursor class is a wrapper, for which the isAllowed method must be implemented. It is given a Storable and is asked to return false if it should not be included in the result set.

A static factory method exists as well, which constructs a FilteredCursor instance from a Filter object. This makes it easy to filter results that can be expressed via simple expressions.

14.2.2 Adapting

The TransformedCursor is similar to the FilteredCursor, except it is designed to take Storables of one type and convert them into another type. The abstract transform method must be implemented, but it may simply return null to filter results.

For adapting ordinary Iterators into Cursors, use the IteratorCursor. This makes it easy to create a stream of arbitrary results from any Collection.

14.2.3 Sorting

Although Carbonado queries support ordering results by property values, sometimes more control is needed. The SortedCursor supports sorting by a custom comparator, or (as a convenience) by a list of properties.

The SortedCursor also allows you to control how results are buffered and sorted. Two sort buffer implementations are available, ArraySortBuffer and MergeSortBuffer.

ArraySortBuffer loads all results into memory, but MergeSortBuffer is able to spill results to external files if it fills with too many results.

14.2.4 Set theory operations

The supported set theory operations are union, intersection, difference, and symmetric difference. All are implemented by similarly named classes.

Each operation wraps two cursors, which must have the exact same total ordering, and a Comparator must be supplied which defines the ordering. If the sorting was performed by a SortedCursor, the Comparator can be accessed from it and be manually passed on. If the results are not sorted consistently, the set operation is likely to return a smaller result set, but the exact behavior is undefined.

The SymmetricDifferenceCursor can be useful for implementing re-sync operations between two Repositories. In order to determine which side is inconsistent, call the compareNext function instead of hasNext. It returns a comparator result indicating which source Cursor is producing the next result.

15 Glossary

This section contains a summary of the key terms used in this document.

Adapter

Annotation for supporting property types that the Repository does not natively support. Custom adapters can be defined as well.

Alias

Annotation used by dependent Repositories for binding Storable types and properties. Specifically, binds Storables types and properties to SQL tables and columns.

AlternateKeys

Annotation for identifying which properties should participate in an alternate key. An alternate key uniquely identifies records just as well as the primary key, except Repositories usually are more forgiving of changes made to an alternate key.

Alternate keys are often called unique indexes in database products, referring to the implementation instead of the concept.

Capability

Mechanism by which Repositories can support new features without bloating the main Repository interface. For Repositories that wrap other Repositories, Capabilities can easily pass through the layers.

Constraint

Annotation which restricts the values that a property can be set to. Custom constraints can be defined as well.

Cursor

Mechanism by which results are returned by a Query. Cursors follow an iterator design pattern, and they also have convenience methods for copying results to Java Collections.

Dependent Repository

Repository which is dependent on externally defined schema. For example, the JDBC Repository depends on the schema in the SQL database.

Filter

Reduces the set of results produced by a Query and is typically represented by a simple expression.

Independent Repository

Repository which is not dependent on externally defined schema. The schema is instead defined by Storable types. The Berkeley DB Repository fits this classification.

Index

An index is a special persistent data structure which can improve the performance of queries, but it may reduce the performance of write operations. In Carbonado, indexes are defined by an Index annotation which is a member of an Indexes annotation. Dependent repositories ignore Index annotations.

Isolation Level

Controls how changes within one transaction are seen by another concurrently running transaction. Isolation level in Carbonado is controlled by passing an IsolationLevel enumeration when entering a transaction.

Join

Defines a relationship between two Storables which can also be used in a query filter. Carbonado joins are specified by the Join annotation, and it supports many-to-one, one-to-one and one-to-many style joins.

Key

Annotation defining a Storable key, which is contained in an AlternateKeys annotation.

LOB

A Large OBject. The two common types are BLOB (Binary Large Object) and CLOB (Caharacter Large Object). Allows raw binary or character data to be stored.

Nullable

Annotation which defines a property as supporting null values. By default, properties cannot be set to a null value.

Optimistic Locking

Means of concurrency control which assumes most transactions don't conflict, and so locks are not immediately granted. A record version number is used to check if a conflict did arise, throwing an exception. Applications may try to perform the transaction again.

PrimaryKey

Annotation for identifying which properties should participate in the primary key. All Storables are required to define a primary key, as it uniquely identifies a Storable instance.

Query

Mechanism to find or delete Storables by means of a filter. Carbonado queries are designed to be minimal, making them easier to implement while still supporting effective automatic index selection.

Repository

Abstraction for representing a persistence technology. A Repository instance is the main Carbonado component.

RepositoryBuilder

Repository instances are created by selecting an appropriate RepositoryBuilder, passing in configuration options, and calling build.

Sequence

Annotation which allows properties to set their own value upon insert. This is often useful for primary keys, simplifying the process of generating new identifiers

Storable

Represents entities that Carbonado can persist. In SQL terms, a Storable type definition is analogous to a table definition, and Storable instances are analogous to table rows.

Storage

Provides direct and query access to Storable instances.

Transaction

Repository-level support for performing operations atomically. Carbonado transaction support may vary by Repository, from being fully ACID to being merely batch based.

Version

Optional annotation for identifying which property defines a Storable's version number. This feature is used for supporting optimistic locking.

16 Appendix: Hello World example

This example shows how to operate on the StoredMessage type defined in the introduction. Here's the StoredMessage definition again:

```
import com.amazon.carbonado.PrimaryKey;
import com.amazon.carbonado.Storable;

@PrimaryKey("ID")
public interface StoredMessage extends Storable {
    long getID();
    void setID(long id);

    String getMessage();
    void setMessage(String message);
}
```

Here's a standalone program that operates on the StoredMessage:


```
import java.io.File;

import com.amazon.carbonado.Repository;
import com.amazon.carbonado.RepositoryException;
import com.amazon.carbonado.Storage;

import com.amazon.carbonado.repo.sleepycat.BDBRepositoryBuilder;

public class HelloCarbonado {
    public static void main(String[] args) throws RepositoryException {
        // Need to build access to a Repository instance. In this
        // case, we're building access to a Repository that stores
        // data in Berkeley DB.
        BDBRepositoryBuilder builder = new BDBRepositoryBuilder();

        // All Repositories require a name to be configured.
        builder.setName("demo");

        // The BDB Repository requires a directory for storage.
        File envHome = new File
            (System.getProperty("java.io.tmpdir"), "carbonado-demo");
        builder.setEnvironmentHomeFile(envHome);
        // Improve write performance by delaying file system sync.
        builder.setTransactionWriteNoSync(true);

        // Now build the Repository instance.
        Repository repo = builder.build();

        // Access Storage for a specific type of Storable.
        Storage<StoredMessage> storage = repo.storageFor(StoredMessage.class);

        // Insert message.
        StoredMessage message = storage.prepare();
        message.setID(1);
        message.setMessage("Hello Carbonado!");
        message.insert();

        // Print the storable contents.
        System.out.println(message);

        // Update the message.
        message.setMessage("Hello World!");
        message.update();

        // Cleanly reload the message.
        message = storage.prepare();
        message.setID(1);
        message.load();

        // Print the storable contents.
        System.out.println(message);

        // Delete the message.
        message.delete();
    }
}
```

The first part of the program uses a builder to instantiate a *Repository*, which is the main access point into Carbonado. In this case, the built repository is backed by a Berkeley DB. Unless configured otherwise, this builder tries to use the Berkeley DB JE (Java Edition) product.

The Repository is asked to provide a *Storage* instance which operates on a specific Storable type, the StoredMessage example. Storage instances are used for all persistence and querying operations on a given Storable type.

Except when performing query operations, the first step in using a Storable is to ask the Storage instance to prepare an implementation. The implementation is generated by Carbonado, and customized as necessary by the Repository in use.

The rest of the program demonstrates the basic CRUD operations (Create, Read, Update, Delete), which should be easy to understand.

Although Repositories implement a close operation, all implementations should register a shutdown hook, as is the case for the Berkeley DB Repository. It is therefore safe for this example to exit the main method without explicitly closing the Repository.

17 Appendix: Query Execution Plan

When using a Berkeley DB backed repository, Carbonado uses its own rule-based query optimizer to create an execution plan. Calling Query.printPlan() dumps the plan to standard out, or you can pass in a StringBuilder to capture the query plan.

Query plans from most databases follow the same general format, and Carbonado is no exception. The query plan is represented by a tree structure, where the leaves represent data sources and nodes above them process results. The root node represents the final step before passing results into the application's cursor. Here's an example plan from Carbonado:

```
sort: [+firstName], [+birthday]
  index scan: org.myorg.myapp.stored.MyRecord
  ...index: {properties=[+lastName, +firstName, ~userID], unique=true}
  ...identity filter: lastName = ?
```

In the above plan, there are only two nodes in the tree: "sort" (the root) and "index scan" (a leaf). Each child node of the tree is indented by two spaces. In this example, the lines that start with an ellipsis are providing extended information regarding the "index scan" node.

This query execution plan shows that an index is used as the data source, and the results are sorted before being passed through to the application's cursor. The application's query is filtering on "lastName = ?" and is ordering by ("+firstName", "+birthday"). The chosen index has the properties "+lastName" and "+firstName", and it is suitable for filtering by "lastName" and ordering by "+firstName". Additional sorting is required for "+birthday", which is why a sort step is required.

17.1 Index properties

The properties of the index selected by the query plan will be prefixed by a '+', '-' or '~'. If '+', the natural ordering of the property is ascending. If '-', the natural order is descending. If '~', the ordering was not explicitly specified, and so it is implicitly ascending

The way in which non-unique indexes are defined, any remaining properties from the master record's primary key are appended to the end. They will have a '~' prefix because the property was not explicitly specified by the user index definition. Adding the properties to the index essentially makes it unique, which is why the "unique=true" property is shown. This is vestigial – all indexes as displayed by the query plan will say they are unique.

Unique indexes (alternate keys) do not put the master record's primary key into the index. They are instead stored in the index entry's value, which is hidden from the query plan. As a result, the index cannot be used to filter on the master record's primary key. In almost all cases, this is of no concern.

17.2 Data sources

Execution plan nodes that are data sources either use an index to start with a subset of results, or they fully scan over all the records. These nodes are always leaves in the tree, but they can also be the root if no additional processing is required.

17.2.1 Full scan

A full scan node simply returns every single instance of a storable type. When calling print plan, it includes the name of the storable type it iterates over.

```
full scan: org.myorg.myapp.stored.MyRecord
```

Full scans are generally considered bad unless you really want all the records. A full scan in other contexts will have additional filtering and sorting operations applied.

```
filter: firstName = ?
  full scan: org.myorg.myapp.stored.MyRecord
```

The above plan indicates that all records are examined, but only those that match a given first name will actually be returned. Defining an index on "firstName" will yield a more efficient *index scan*, discussed in the next section.

17.2.2 Index scan

An index scan attempts to fully utilize an available index to satisfy a query's filtering and ordering requirements. Printing the plan shows the name of the storable type it iterates over followed by a few more detail lines, each prefixed by an ellipsis.

```
index scan: org.myorg.myapp.stored.MyRecord
...index: {properties=[+lastName, +firstName, ~userID], unique=true}
...identity filter: lastName = ? & firstName = ?
```

The above query plan shows an index being used to find all users with a given first and last name. The same index can also be used to find a range of records, like all those with a last name that starts with "S":

```
index scan: org.myorg.myapp.stored.MyRecord
...index: {properties=[+lastName, +firstName, ~userID], unique=true}
...range filter: lastName >= S & lastName < T
```

An index can also be used for both range and identity matching. For example, finding all users whose last name is "Smith" and whose first name starts with "J":


```
index scan: org.myorg.myapp.stored.MyRecord
...index: {properties=[+lastName, +firstName, ~userID], unique=true}
...identity filter: lastName = Smith
...range filter: firstName >= J & firstName < K
```

An index can be used to produce results in a desired order, even if the natural order of the index is backwards. In this case, the scan is reversed. Here's an example where the index is used to find users of a given last name, and then sorted by first name in reverse:

```
reverse index scan: org.myorg.myapp.stored.MyRecord
...index: {properties=[+lastName, +firstName, ~userID], unique=true}
...identity filter: lastName = Smith
```

Finally, an index may be selected by the query optimizer simply because it is *clustered*. A clustered index defines the natural ordering of records, and there can be only one clustered index for a given storable type. When using Berkeley DB, the primary key is always treated as the clustered index. The plan will show if an index is clustered, which indicates that the scan will perform much better than a usual one:

```
clustered index scan: org.myorg.myapp.stored.OrderItem
...index: {properties=[+orderID, +productID], unique=true}
...identity filter: orderID = ?
```

A clustered index scan can also be reversed, in which case the plan prints "reverse clustered index scan".

17.2.3 Index key

If an index is used in such a way that it is guaranteed to produce at most one result, the plan looks like so:

```
index key match: org.myorg.myapp.stored.OrderItem
...index: {properties=[+orderID, +productID], unique=true}
...key filter: orderID = ? & productID = ?
```

This really just a special variation of an index scan, except it may run more efficiently since the data source doesn't need to open a physical cursor of its own.

17.2.4 Covering filter

Consider a query filter of "lastName >= ? & lastName < ? & firstName = ?". Probably the best index to use for this query is {"firstName", "lastName"}. It will produce a query plan like so:


```
index scan: com.myorg.myapp.stored.MyRecord
...index: {properties=[+firstName, +lastName, ~userID], unique=true}
...identity filter: firstName = ?
...range filter: lastName >= ? & lastName < ?
```

If the only available index is instead {"lastName", "firstName"}, the query plan might be:

```
filter: firstName = ?
  index scan: com.myorg.myapp.stored.MyRecord
  ...index: {properties=[+lastName, +firstName, ~userID], unique=true}
  ...range filter: lastName >= ? & lastName < ?
```

The above query plan uses the index to restrict by last name, but all records retrieved by the index scan need to go through a filter. If few records pass through the filter, they would have been loaded unnecessarily. Since the firstName property is available in the index, it is possible to perform the filter without having to fully load the master record. The actual query plan looks like this:

```
index scan: com.myorg.myapp.stored.MyRecord
...index: {properties=[+lastName, +firstName, ~userID], unique=true}
...range filter: lastName >= ? & lastName < ?
...covering filter: firstName = ?
```

The last line shows that the extra filtering is performed via the index entry itself, which offers a performance benefit. A better query plan is produced when there is a better index, but this is a trade off. Fewer indexes boosts the performance of writes. Search the web for “covering index” to find out more regarding this approach.

17.3 Data processing

If a data source doesn't produce results in exactly the desired fashion, additional processing is required before results can be passed through to the application cursor. Nodes of this kind can never be leaves in the execution plan tree.

17.3.1 Filter

If the data source provides too many results, a filter can discard results that don't match. If filtering is required, it is always performed before any other data processing steps. This reduces the work required by say, a sort operation.

```
filter: price >= 100 & price < 200
```

The above filter is simply discarding results that don't match a certain price. The full plan will show the data source, perhaps like so:


```
filter: price >= 100 & price < 200
  clustered index scan: org.myorg.myapp.stored.OrderItem
  ...index: {properties=[+orderID, +productID], unique=true}
  ...identity filter: orderID = ?
```

The above plan is looking for products in a given order, which fit a certain price range. No suitable index was found to filter on price range, which is why the filter step is included.

17.3.2 Sort

If a data source doesn't produce results in the desired order, a sort step does just that. Carbonado's sorter will sort results in memory, but it may serialize records to temporary files and merge them back together if necessary.

```
sort: [-dateAdded]
```

The above plan snippet shows a sort by descending "dateAdded". The entire plan is needed to show exactly what is going on:

```
sort: [-dateAdded]
  filter: price >= 100 & price < 200
    clustered index scan: org.myorg.myapp.stored.OrderItem
    ...index: {properties=[+orderID, +productID], unique=true}
    ...identity filter: orderID = ?
```

The query plan shows that an index is used to produce a superset of results, a filter is reducing them to certain price range, and the sort is showing recently added products first.

If the index provides partial ordering, the sort will only need to sort within groups. When it does this, two bracketed sets of properties are shown. The first set shows the ordering handled by the index, and the second set shows the sorting required in each handled group.

A query to find users under a certain age, ordered by ("+birthday", "+lastName", "+firstName") might look like:

```
sort: [+birthday], [+lastName, +firstName]
  reverse index scan: org.myorg.myapp.stored.MyRecord
  ...index: {properties=[-birthday, ~userID], unique=true}
  ...range filter: birthday > ?
```

The index has a natural ordering by descending birthday, but a reverse scan corrects that. The sort finds groups of consecutive birthdays, and then it sorts by ("+lastName",

"+firstName") within those groups. A sort of this kind produces results more quickly, and it might not need to spill to temporary files.

17.3.3 Union

Although Carbonado queries don't support explicit unions, a query plan may have a union step if the query filter contains any logical *or* operations. Each child of the union node will ultimately have a separate data source. A query for "price = ? | dateAdded = ?" might be executed as:

```
union
  index scan: org.myorg.myapp.OrderItem
  ...index: {properties=[+price, +orderID, ~productID], unique=true}
  ...identity filter: price = ?
  index scan: org.myorg.myapp.OrderItem
  ...index: {properties=[+dateAdded, +orderID, ~productID], ...
  ...identity filter: dateAdded = ?
```

Here, two index scans feed into the union, which in turn ensures that no duplicate records are returned. The union step requires that all sources which feed into it have a consistent *total ordering*. This affects what indexes are selected, and if necessary, sort steps are tacked on:

```
union
  sort: [+birthday, +userID]
    index scan: org.myorg.myapp.MyRecord
    ...index: {properties=[+lastName, +firstName, ~userID], ...
    ...identity filter: lastName = ?
  index scan: org.myorg.myapp.MyRecord
  ...index: {properties=[+birthday, ~userID], unique=true}
  ...range filter: birthday > ?
```

The above query plan might have been produced for a query filter of "lastName = ? | birthday > ?".

17.3.4 Join

Join query plans are the most complicated and can produce very tall execution plan trees. As of this writing, Carbonado only supports *nested loops* style joins. It will however fully take advantage of available indexes when joining, and it will distribute filtering and sorting into all selected indexes.

An individual join node will have an outer loop and an inner loop. The outer loop is executed once, but the inner loop is executed once for each result of the outer loop. This example is looking for orders to a specific city using "address.addressCity = ?":


```
join: org.myorg.myapp.stored.Order
...inner loop: address
  index scan: org.myorg.myapp.stored.Order
  ...index: {properties=[+addressID, ~orderID], unique=true}
  ...identity filter: addressID = ?
...outer loop
  index scan: org.myorg.myapp.stored.Address
  ...index: {properties=[+addressCity, ~addressID], unique=true}
  ...identity filter: addressCity = ?
```

An index exists on Address.addressCity and on Order.addressID. This allows the query to find matching Address records in the outer loop, and for each match, look up the corresponding Order.

In this next complex example, shipments are being filtered by "order.address.addressState = ? & order.address.addressID != ? & order.address.addressZip = ? & order.orderTotal > ? & shipmentNotes <= ?" and are ordered by ("+order.address.addressCity", "+shipmentNotes", "+order.orderTotal"):

```
sort: [+order.address.addressCity, +shipmentNotes], [+order.orderTotal]
  join: org.myorg.myapp.stored.Shipment
  ...inner loop: order
    index scan: org.myorg.myapp.stored.Shipment
    ...index: {properties=[+orderID, +shipmentNotes, ~shipmentID], ...
    ...identity filter: orderID = ?
    ...range filter: shipmentNotes <= ?
  ...outer loop
    join: org.myorg.myapp.stored.Order
    ...inner loop: address
      index scan: org.myorg.myapp.stored.Order
      ...index: {properties=[+addressID, -orderTotal, ~orderID], ...
      ...identity filter: addressID = ?
      ...range filter: orderTotal > ?
    ...outer loop
      filter: addressZip = ? & addressID != ?
        reverse index scan: org.myorg.myapp.stored.Address
        ...index: {properties=[+addressState, -addressCity,  ~addressID], ...
        ...identity filter: addressState = ?
```

This example is somewhat contrived, but it shows how complicated a query plan can get with a multi-way join. It should also be noted that the root sort operation is able to take advantage of the natural ordering of the indexes, and so it only needs to sort within groups by "order.orderTotal".

18 Appendix: Recommended practices

Storables are fairly simple objects, but listed here are a few recommended practices for avoiding trouble.

18.1 Scope Storables privately

Although Storables are defined as being totally public, avoid exposing Storables in your application's published interface. Exposing Storable instances directly can encourage misuse, bypassing any special domain logic that must be applied when operating on persisted data.

Create appropriate layers of abstraction to separate application features from persistence. The classic Model-View-Controller[10] pattern is appropriate, with Storables representing the model.

To make Storable definitions appear separate from your published interface, either place them in a separate package or follow a naming convention. For example, place all Storables in a sub-package named "stored", or prefix all Storable names with "Stored".

18.2 Primary key design

Although all Repositories should support some sort of schema evolution, care must be taken when modifying a Storable's primary key if persisted data already exists in the database. Changing the primary key may either be impossible, or it might make it very difficult to retrieve old records.

To prevent this problem from happening, pay close attention when designing the primary key. Are the property types general enough to support future types of records? If numerical, does the property have enough precision? Should more properties be added to the key?

One simple way to support evolution of the primary key is to rely on alternate keys instead. Define the primary key as a long, with a Sequence annotation to create identifiers for you. Then define a composite alternate key which would have otherwise served as the primary key. Repositories are much more forgiving when alternate keys change, since they are often implemented as indexes. They can be rebuilt with no loss of data.

Unfortunately, alternate keys cannot be used for all CRUD operations, making them a bit more cumbersome to use than primary keys. Loads can be performed by alternate key, but updates and deletes cannot.

18.3 Versioning

As described earlier, adding a version property enables optimistic locking. In general, it's a good idea to include a version property. This makes it easier to use Storables in a wider

[10] MVC splits user interface interaction into three distinct roles. See also: *Design Patterns: Elements of Reusable Object-Oriented Software.*

variety of applications. For example, a web form for editing content can have a hidden field containing the version of the content. If another user has edited the content before the first user submitted the form, the optimistic lock exception detects this.

Databases that support replication may provide weak consistency guarantees, and so a version property can be used to detect inconsistencies. How the inconsistency is repaired is up to the application. Some distributed databases may require a version property, to guarantee idempotent updates.

Adding a version property does restrict the functionality of update operations, however, since a valid record version must be provided to the update. Also, the extra version property increases the storage requirements of a Storable slightly, and so it might not be appropriate in applications that must be frugal.

